UE41 **Cours n°1** GB3, 1^{er} semestre 2017-2018
Cours de Python Gilles Bernot

Les notes de cours et exercices sont disponibles (avec un peu de retard par rapport au déroulement du cours) à l'adresse web suivante :

`http://www.i3s.unice.fr/~bernot/Enseignement/Master_Python1/`

1 L'*art* de programmer

Utiliser un langage de programmation pour faire enchaîner à l'ordinateur, rapidement, des opérations sur des données n'est pas très difficile. Cependant, avec un style de programmation approximatif, on peut vite écrire des programmes incompréhensibles et c'est alors la porte ouverte à de nombreuses erreurs. Ce cours a pour objectif de vous apprendre à écrire des programmes *propres*, sans « bidouillages », et tellement logiquement écrits qu'on pourra les modifier plus tard, quand on aura même oublié comment ils marchent : en programmation, tout est dans **le *style* et l'*élégance***...

On va utiliser le langage *Python* comme support mais le choix du langage n'est pas très important car avec les méthodes de programmation « dans les règles de l'art » acquise dans ce cours, il sera facile de passer d'un langage à un autre. Les concepts de base d'une programmation bien menée sont les mêmes dans tous les langages.

2 Comment fonctionne un ordinateur, dans les grandes lignes

Avant d'apprendre les premiers éléments de programmation, cette partie du cours a pour objectif de démystifier les ordinateurs (et de vous donner les principaux critères utiles qui font la qualité d'un ordinateur).

2.1 Le processeur et la mémoire

Le cœur d'un ordinateur est son *processeur* : c'est une sorte de « centrale de manipulations » qui effectue des transformations électriques rapides, correspondant à diverses manipulations de *symboles* codés par des « signaux électriques ». Il travaille de concert avec une *mémoire* qui alimente le processeur en *données* et en *instructions* à réaliser : sur le fond, la mémoire d'ordinateur stocke simplement une longue suite de symboles qui peuvent représenter aussi bien des données que des instructions.

L'élément de base des codages sus-mentionnés est l'information binaire : « le courant peut passer » ou au contraire « le courant ne peut pas passer ». Cette information minimale est appelée un *bit*. Un bit peut donc prendre deux valeurs, 0 ou 1, et est matériellement réalisé par un genre de transistor largement miniaturisé.

Un symbole est codé par une séquence de bits. Ainsi avec seulement 2 bits on peut déjà encoder 4 symboles différents.

> *Par exemple on pourrait décider que 00 code « A », symbole pour l'Adénine, que 01 code « T », symbole de la Thymine, que 10 code « G », symbole de la Guanine, et que 11 code « C », symbole de la Cytosine.*

Avec 3 bits, on peut encoder 8 symboles différents (4 commençant par le bit 0 et 4 commençant par le bit 1), *etc.* En pratique, on compte les volumes d'information en *octets*. Un octet (« byte » en anglais) est formé de 8 bits et peut donc encoder $2^8 = 256$ symboles différents. Le code le plus utilisé est le code ASCII [1], qui encode sur un octet (presque) toutes les lettres (majuscules, minuscules), les ponctuations, l'espace, des caractères dits de contrôle, *etc.* Pour encoder les lettres accentuées et les lettres spécifiques à certaines langues, il faut faire appel à un autre code qui peut utiliser 2 octets pour ces lettres particulières ; le plus courant est le code UTF-8.

L'électronique n'a aucune difficulté à manipuler les bits : éteindre un bit, l'allumer, inverser sa valeur, tester sa valeur... Il existe ainsi un ensemble (limité) d'*instructions* qui manipulent les bits (souvent par groupes de 8 bits donc par octet, ou même par groupes de plusieurs octets). Ainsi un ordinateur est en fait un *manipulateur électronique de symboles* (encodés par des octets). De plus, chaque instruction de manipulation est elle même codée par une séquence de bits, exactement comme les symboles.

Les instructions successives à effectuer sont placées en des endroits déterminés de la mémoire et sont lues les unes après les autres, ce qui conduit à enchaîner des commandes et finalement à faire des transformations aussi complexes que l'on veut de la mémoire. La mémoire sert donc à la fois à stocker le programme des instructions successives à

1. American Standard Code for Information Interchange

effectuer et à stocker les données/symboles que ce programme manipule. Pour ce faire le proceseur possède une zone dans laquelle il note la place de la mémoire où se trouve la prochaine instruction à effectuer, une zone dans laquelle il a recopié l'instruction en cours, et quelques zones qui servent de la même façon à manipuler les données. Ces zones sont appelées des registres.

Maintenant que l'on a vu comment transitent les instructions successives à effectuer dans le processeur, parlons des données : il ne servirait à rien de faire tant de manipulations si elles ne sortaient jamais des registres du processeur. Pour cela il y a des instructions de stockage des registres du processeur vers la mémoire, et pour alimenter le processeur en données, il existe inversement des instructions de chargement des données qui peuvent copier n'importe quel emplacement de la mémoire vers les registres de données. Ainsi les suites d'instructions commencent souvent par charger certaines places de la mémoires dans les registres, puis font divers transformations de ces symboles, et enfin stockent de nouveau en mémoire les résultats présents dans les registres.

2.2 Les périphériques

Ainsi, on voit que si l'on conçoit un programme judicieux d'instructions successives, on peut modifier à volonté l'état de la mémoire pour lui faire stocker des résultats de manipulations de symboles, aussi complexes et sophistiquées que l'on veut. La question qui se pose maintenant est d'exploiter cette mémoire en la montrant à l'extérieur sous une forme compréhensible pour l'Homme, ou réexploitable ultérieurement. C'est le rôles des *périphériques* :

- Avec ce que l'on a vu jusqu'à maintenant, s'il y a une coupure de courant alors tout est perdu car les « mini-transistors » ne conservent pas leur état (0 ou 1) sans courant. Donc certains périphériques sont des mémoires « de masse » comme les disques durs, les disquettes, les CDrom, DVDrom, Blu-ray *etc.* ou certaines mémoires « flash » (=clefs USB) ayant une durée raisonnable de conservation des données.
- Il faut également pouvoir intervenir et entrer les données et les ordres dans l'ordinateur par exemple *via* un clavier, une souris, une manette de jeu, des instruments de mesure, *etc.* Ces périphériques « d'entrées » agissent en modifiant certaines parties de mémoires (indiquant les touches enfoncées, les déplacements de souris ou de doigts sur un écran tactile, *etc*) et les programmes qui gèrent le système informatique « surveillent » tout simplement régulièrement ces mémoires particulières.
- Accéder aux résultats et les visualiser est également nécessaire ; l'ordinateur doit pouvoir montrer à l'utilisateur les résultats des manipulations symboliques par exemple via une carte graphique et un écran, une imprimante ou divers appareillages de réalité virtuelle... De la même façon le rôle de ces périphériques « de sortie » (comme la carte graphique) est simplement de transcrire (par exemple en pixels de couleur) le contenu de « sa » zone de mémoire ; zone de mémoire que les programmes peuvent naturellement modifier à volonté.
- Plus généralement, entrées et sorties combinées donnent lieu à des périphériques « de communication » qui permettent non seulement de communiquer avec l'Homme mais aussi avec d'autres ordinateurs par exemple *via* des réseaux.

3 Quelques unités de capacité d'information

- Un *bit* est une valeur logique (vrai/faux codé en 0/1).
- Un *octet* (en anglais : byte) est une suite de 8 bits ; il permet de coder par exemple des entiers de 0 à 255, ou des caractères, ou des instructions comme déjà mentionné.
- Un *Ko* se prononce un *kilo-octet* (en anglais : Kb, Kilo-byte) ; c'est une suite de 1024 octets (pas exactement 1000 parce que 1024 est une puissance de 2, ce qui facilite l'adressage sur ordinateur).
- Un *Mo* se prononce *Méga-octet* (en anglais : Mb, Mega-byte) ; c'est une suite de 1024 Ko.
- Un *Go* se prononce *Giga-octet* (en anglais : Gb, Giga-byte) ; suite de 1024 Mo.
- Un *Tera-octet* est une suite de 1024 Go, puis viennent les *Peta-octet*, *Exa-octet*, *etc.*

Actuellement les valeurs significatives lorsque vous achetez un ordinateur sont :

- La vitesse à laquelle se succèdent les opérations élémentaires faites par le ou les processeurs de l'ordinateur : de 1 à 5 Giga-opérations par seconde. Il s'agit donc d'une *fréquence* exprimée en Hertz (par exemple 3,5GHz).
- Le nombre de processeurs mis ensemble, c'est-à-dire combien d'opérations élémentaires se font en même temps : de 1 à 8 (mono-core, dual-core, quad-core, 8-core/octo-core) sur les ordinateurs individuels.
- La taille de la mémoire : souvent de 1 à 32 Giga-octets.
- La taille du disque dur dans lequel seront stockés vos programmes (applications, outils bureautiques, lecteurs divers, jeux, *etc.*) et vos données (textes, images, musique, films, bases de données, *etc.*) : de 200 Giga-octets à 10 Tera-octets.
- La résolution de l'écran : de 800 à 3000 pixels de large et de 700 à 2000 pixels de haut, soit de 600 Kilo-pixels à 6 Méga-pixels environ.

4 Les langages de programmation

Ce qu'on vient de voir sur la structure d'un ordinateur donne déjà des capacités de programmation importantes : il suffit de mettre en mémoire une suite d'instructions à effectuer, et le processeur les chargera et les effectuera les unes après les autres. C'est ce qu'on appelle le *langage machine*. Comme son nom l'indique, c'est un langage très efficace pour une machine... mais il est très indigeste à lire (et à écrire) pour un être humain. Plutôt que d'écrire du langage machine, l'humain préfère écrire des ordres à l'ordinateur dans un langage plus évolué (par exemple Python).

C'est là qu'intervient la *compilation* :

1. Un programme écrit par un humain en Python est finalement un texte, c'est-à-dire une suite de caractères qui peut donc être stockée dans la mémoire de l'ordinateur.
2. Les programmes en langage machine sont des suites d'instructions qui doivent aussi être stockées en mémoire.
3. *Or* un programme en langage machine est finalement un processus qui transforme des données en mémoire en d'autres données en mémoire, quelles qu'elles soient. Un programme en langage machine peut donc considérer un programme écrit en Python comme des données, de même qu'il peut considérer un autre programme en langage machine comme des données.
4. *Donc*, si quelques spécialistes se chargent de la corvée d'écrire un programme en langage machine qui transforme :
   — un (texte de) programme en python
   — en une suite d'instructions en langage machine
   alors nous pouvons écrire un texte en Python, laisser le programme des spécialistes en faire un programme en langage machine, et faire tourner le résultat.

Cette technique peut s'appliquer de différentes façons que nous ne détaillerons pas ici. Elle est appelée selon les cas la *compilation* ou l'*interprétation* du langage (ici Python). Ceci permet d'avoir l'impression que c'est le programme écrit en Python qui « tourne » directement sur l'ordinateur.

> *D'une certaine façon, l'interprétation d'un langage de programmation joue pour l'informatique un rôle inverse de la transcription et la traduction pour la biologie moléculaire : la transcription et la traduction permettent de construire des structures de plus haut niveau à partir du « langage machine » qu'est le génome, alors que l'interprétation produit du langage machine à partir de textes bien structurés.*

Grâce à cette technique classique, la notion de *langage de programmation* devient plus large qu'une simple suite d'instructions données à l'ordinateur.

Un langage de programmation est un « vocabulaire » restreint et des règles de formation de « phrases » très strictes pour donner des instructions à un ordinateur. Le *moins* par rapport au français ou aux mathématiques reste malgré tout sa pauvreté, mais le *plus* est qu'aucune « phrase » (=*expression*) n'est ambigüe : il n'existe pas plurieurs interprétations possibles.

On peut alors :

— regrouper puis abstraire un grand nombre de données élémentaires (nombres, caractères...) pour caractériser une *structure de données* abstraite (`protéine` = suite de ses acides aminés *et* ses conformations possibles *et* ses sites actifs en fonction de conditions, *etc*).

— regrouper des suites de commandes élémentaires (additions, multiplications, statistiques, classifications, décisions...) pour organiser le *contrôle du programme* et le rendre plus lisible et logique (déterminer si deux protéines ont de l'`affinité` = inventorier les conditions du compartiment qui les contient, calculer la conformation la plus probable, inventorier les sites actifs qui en résultent, comparer deux à deux si un domaine d'une protéine a de l'affinité avec un domaine de l'autre protéine, *etc*).

Donc : ce qui définit un langage de programmation, c'est

— la façon de représenter symboliquement les **structures de données** (e.g. protéine)

— et la façon de gérer le **contrôle** des programmes (e.g. que faire et dans quel ordre pour résoudre une question d'affinité de protéines).

5 Un exemple en Python

On reviendra plus précisément sur chacune des notions utilisées ici. Il s'agit simplement pour l'instant de *voir* à quoi ressemble un programme et son utilisation.

```
>>> borne = 100
>>> def evalue (n) :
...   if n < (borne / 2) :
```

```
...     return("petit")
...   elif n < borne :
...     return("moyen")
...   else :
...     return("grand")
...
>>> evalue (70)
'moyen'
>>> evalue (150)
'grand'
>>> borne = 50
>>> evalue (70)
'grand'
```

Selon le système d'exploitation avec lequel on travaille, la présentation peut différer (les « ⋙ » ou les « ... ») mais en revanche, ce qu'on tape au clavier et les réponses de l'ordinateur sont *toujours* identiques.

- La première ligne « `borne = 100` » est une *affectation* de variable. Elle a pour effet qu'à partir de cette ligne, il est équivalent d'écrire `borne` ou d'écrire `100`. On dit que `borne` est une *variable* et que sa *valeur* est 100. L'avantage de cette ligne est double :
  - Partout où l'on a utilisé `borne` au lieu de `100`, il sera facile de « changer d'avis » et de considérer que finalement la borne ne vaut que 50. Il suffit de faire une nouvelle affectation *sans avoir besoin de chercher partout* où se trouvait le nombre 100 et le remplacer par 50. Toutes les valeurs qui peuvent changer un jour (taux de TVA, mesure de diverses quantités, *etc*) doivent donc être mises dans des variables.
  - Après des affectations de variables, les programmes sont plus lisibles car les noms de variables permettent de leur donner une signification. On peut écrire des programmes compréhensibles où les valeurs sont remplacées par des noms parlants, qui aident à la compréhension. L'idéal serait de pouvoir programme presque comme en français « si l'entier `n` est plus petit que la moitié de la borne alors il est considéré comme petit, sinon s'il reste plus petit que la borne alors il est moyen, sinon il est grand. »
- Le mot clef `def` ressemble lui aussi à une affectation en ce sens qu'il donne un nom à quelquechose. Il s'agit de donner un nom (`evalue`) à un petit programme qui fait des manipulations symboliques en fonction d'une valeur qu'on lui donne (`n`). Ici, le nom `evalue`, chaque fois qu'il sera appelé, lancera les comparaisons du nombre qu'on lui donne entre parenthèses avec la `borne` et/ou sa moitié, et il imprimera à l'écran le résultat qui en découle (grand, moyen ou petit).
  La construction `if..elif..else..` n'a rien de difficile :
  - `elif..` est une contraction de « else if.. »
  - juste derrière le « `if` » et avant le « : » on écrit la condition qui, si elle est vraie, conduit à effectuer les commandes qui suivent, et si par contre elle est fausse, conduit à effectuer les commandes qui suivent le « `else` ». C'est logique.
- Enfin la commande `return` signale ce que doit être le résultat de la fonction. Lorsque ce résultat n'est pas utilisé par ailleurs, il est imprimé à l'écran.

On dit que la « fonction `evalue` prend `n` en entrée » ou encore « en argument », pour dire que le résultat que fournit `evalue` dépend de la valeur « mise à la place de `n` » entre les parenthèses. On remarque que ce résultat dépend aussi de `borne`, mais cette fois de manière implicite (c'est-à-dire que `borne` n'est pas un argument de `evalue`).

Puisqu'un langage de programmation est défini par ses *structures de données* et par son *contrôle*, il suffit de connaître les parties les plus utiles de ces deux aspects en Python pour savoir programmer en Python. Mieux : sur le fond, les principales structures de données et les principales primitives de contrôle *sont les mêmes* pour tous les langages dits impératifs (qui eux-mêmes représentent la quasi-totalité des langages industriels). Les seules variations d'un langage à l'autre sont les choix des mots clefs : par exemple certains langages utilisent « `function` », « `procedure` » ou encore « `let` » au lieu de « `def` ». C'est dire à quel point un cours de programmation peut être universel, sous réserve de ne pas se noyer dans les particularités de détail du langage choisi.

Nous allons commencer par les structures de données les plus simples.

6 Les structures de données

Une structure de données est définie par quatre choses :

1. *Un type* qui est simplement un nom permettant de classifier les expressions syntaxiques relevant de cette structure de donnée. Par exemple, `"grand"`, avec ses guillemets autour, est une donnée de type `string` (chaîne

de caractères en français) et `"petit"` et `"moyen"` sont deux autres données de type `string` elles aussi. Python abrège `string` en `str`.

2. *Un ensemble* qui définit avec précision quelles sont les *valeurs pertinentes* de ce type. Par exemple l'ensemble de toutes les suites de caractères, de n'importe quelle longueur, sachant qu'un caractère peut aussi bien être une lettre qu'un chiffre, une ponctuation ou un caractère dit « de contrôle ».
3. *La liste des opérations* que l'ordinateur peut utiliser pour effectuer des « calculs » sur les valeurs précédentes. Cela comprend le nom de l'opération (par exemple la `/` fait partie des opérations sur les nombres) et comment l'utiliser, c'est-à-dire quels *arguments* elle accepte et la nature du résultat. Dans l'exemple précédent, on a vu par exemple que `/` accepte deux arguments de type `float` (comme on le verra plus loin) et cette division a pour résultat encore un nombre (de type `float` donc).
4. *La sémantique des opérations* c'est-à-dire une description précise du résultat fourni en fonction des arguments. Dans notre exemple, « `/` » est la division habituelle des nombres (avec virgule).

Il existe en Python une commande `type`, qui prend en argument une valeur ou une variable quelconque et fournit comme résultat le type de cette valeur :

```
>>> type(2.5)
<class 'float'>
>>> type("toto")
<class 'str'>
```

7 Les booléens

Il s'agit d'un ensemble de seulement 2 données pertinentes, dites *valeurs de vérité* : {`True`,`False`}. Le type est appelé `bool`, du nom de George Boole [1815-1864] : mathématicien anglais qui s'est intéressé aux propriétés algébriques des valeurs de vérité.

Opérations qui travaillent dessus :

Opération	Entrée	Sortie
`not`	`bool`	`bool`
`and`	`bool`×`bool`	`bool`
`or`	`bool`×`bool`	`bool`

De manière similaire aux tables de multiplications, on écrit facilement les tables de ces fonctions puisque le type est fini (et petit).

Exemples de calculs sur les booléens :

```
>>> False
False
>>> type(True)
<class 'bool'>
>>> true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> True and False
False
>>> True and
  File "<stdin>", line 1
    True and
            ^
SyntaxError: invalid syntax
```

Comme on le voit, l'opération `type` écrit bien le type d'une donnée.

8 Les entiers relatifs

Théoriquement le type `int` correspond à l'ensemble $\mathbb{Z}$ des mathématiques. En fait il est borné en Python, avec borne dépendante de la machine, mais assez grande pour ignorer ce fait ici. (`int` comme « integers »).

Les opérations qui travaillent dessus :

Opérations	Entrée	Sortie
`+` `-` `*` `//` `%` `**`	`int`×`int`	`int`
`==` `<` `>` `<=` `>=` `!=`	`int`×`int`	`bool`

Exemples de calculs sur les entiers :

```
>>> type(46)
<class 'int'>
>>> 5 * 7
35
>>> 5 // 2
2
>>> 5 % 2
1
>>> 5 ** 3
125
```

On remarque que la division `//` est *euclidienne* (pas de virgule dans le résultat) et que `%` donne le reste de la division euclidienne.

Pour les comparaisons :

```
>>> 5 < 3
False
>>> 5 == 3 + 2
True
>>> 5 = 3 + 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 5 <= 5
True
```

Noter la différence entre « `==` » (qui effectue une comparaison et fournit un booléen en résultat) et « `=` » qui est une affectation (et doit avoir un nom de variable à sa gauche et une valeur à sa droite). On remarque aussi que les opérations de calcul `+` `-` `*` `//` `%` `**` sont *prioritaires* sur les opérations de comparaison `==` `<` `>` `<=` `>=` `!=` car sinon « `5 == 3 + 2` » n'aurait pas eu plus de sens que « `False + 2` » (on n'additionne pas un booléen et un entier).

9 Les nombres réels

On les baptisent « les nombres flottants » pour des raisons historiques.

Le type `float` représente l'ensemble des nombres réels, avec cependant une précision limitée par la machine, mais les erreurs seront négligeables dans le cadre de ce cours. Par abus d'approximations on considère donc que `float` correspond à l'ensemble $\mathbb{R}$.

On dispose des opérations suivantes :

Opérations	Entrée	Sortie
`+` `-` `*` `/` `**`	`float`×`float`	`float`
`int`	`float`	`int`
`float`	`int`	`float`
`==` `<` `>` `<=` `>=` `!=`	`float`×`float`	`bool`

Exemples d'utilisation :

```
>>> type(46.8)
<class 'float'>
>>> 5.0 / 2.0
2.5
>>> 7.5 * 2
15.0
```

```
>>> int(7.5 * 2)
15
>>> int(7.75)
7
>>> float(3)
3.0
```

Noter la différence entre `/` et `//` : `/` est la division exacte sur les réels uniquement, et `//` est la division euclidienne sur les entiers relatifs uniquement.

10 Les chaînes de caractères

Dans l'encodage le plus courant, il y a 256 « *caractères* » qui couvrent à peu près tout ce que l'on peut taper au clavier en une fois, en utilisant les touches de nombres, lettres ou ponctuations, éventuellement en combinaison avec la touche *majuscule* (*shift* en anglais) ou bien la touche *contrôle*. Une *chaîne de caractères*, comme son nom l'indique, est une suite finie de caractères.

Le type des chaînes de caractères est généralement appelé `string` mais en Python il est appelé de manière abrégée `str`. Pour produire une chaîne de caractères, il suffit de l'écrire entre guillemets, simple ou doubles :

```
>>> "toto"
'toto'
>>> 'toto'
'toto'
>>> toto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'toto' is not defined
>>> toto = 56
>>> toto
56
```

Les guillemets sont nécessaires car sinon, Python interprète la chaîne de caractères comme le nom d'une variable, et la commande est alors interprétée par Python comme « donner la valeur de la variable `toto` ».

Une opération courante sur les chaînes de caractères est la concaténation, notée avec un « + ». Les autres opérations fréquentes sont la longueur (nombre de caractères), notée `len`, l'inclusion de chaînes de caractères, notée `in`, et les comparaisons (ordre du dictionnaire, plus précisément, ordre lexicographique).

```
>>> "toto" + "tutu"
'tototutu'
>>> len("toto")
4
>>> "bc" in "abcde"
True
>>> "bd" in "abcde"
False
>>> "toto" < "tutu"
True
>>> "ab" < "aab"
False
>>> "a b" == "ab"
False
```

Noter l'absence d'espace entre `"toto"` et `"tutu"` après concaténation.

Opérations	Entrée	Sortie
`+`	`str`×`str`	`str`
`len`	`str`	`int`
`in`	`str`×`str`	`bool`
`==` `<` `>` `<=` `>=` `!=`	`str`×`str`	`bool`

11 Mise en pratique : premières fonctions et procédures en Python

Exercice 1 : Écrivez en python une fonction `carre` qui prend en entrée un nombre entier relatif (de type `int`) et retourne en sortie son carré.

Par exemple `carre(5)` retourne `25`.

Faites des essais d'utilisation avec plusieurs valeurs entières.
Faites aussi un essai avec une valeur réelle (type `float`) : ça marche aussi ! pourquoi à votre avis ?

Exercice 2 : Écrivez en python une procédure `double` qui prend en entrée un nombre entier relatif `n` et imprime à l'écran `"le double de ... est ..."` qui indique à combien est égal le double de `n`.

Par exemple `double(6)` imprime à l'écran `"le double de 6 est 12"` (mais ne retourne aucun valeur en tant que fonction).

Faites 2 versions, l'une utilisant la concaténation `+`, l'autre avec l'opération de substitution `%`.

Exercice 3 : Quels résultats donneraient respectivement les expressions `1 + carre(2)` et `1 + double(2)` ?

Vérifiez vos prédictions *après* les avoir formulées...

Exercice 4 : Écrivez une fonction `triangle` de 3 arguments `a`, `b` et `c` qui indique si ces 3 réels définissent les côtés d'un triangle, en suivant le procédé suivant : le plus grand des côtés doit être inférieur à la somme des deux autres.

Exercice 5 : Écrivez en python une fonction `nucl` qui prend en entrées une chaîne de caractères `c` (de type `str`) supposée représenter un codon et un nombre entier `i` (de type `int`) et retourne en sortie le `i`-ième nucléotide du condon `c`.

Par exemple `nucl("ATC",2)` retourne `"T"`.

La fonction doit de plus imprimer un message d'erreur à l'écran si `c` n'est pas un codon (longueur différente de 3 ou contient d'autres lettres que `A`, `T`, `G` ou `C`) et si `i` n'est pas compris entre 1 et 3.

À l'occasion de ces premiers exercices, on découvre entre autres :
- que l'indentation (les marges) doivent impérativement être scrupuleusement respectées,
- qu'il faut une ligne vide à la fin d'un « `def ... :` » pour que l'ordinateur comprenne qu'on a fini la définition de fonction,
- que les calculs que l'on écrit doivent tous respecter une forme « arborescente » dans laquelle le type retourné par un sous-arbre de calcul doit coïncider avec celui attendu par chaque fonction en argument, en particulier il ne faut pas écrire « `a >= b and c` » qui est mal typé si `a`, `b` et `c` sont des entiers, alors que l'on voulait écrire « `a >= b and a >= c` »
- *etc.*