

Nota : sur la page web http://www.i3s.unice.fr/~bernot/Enseignement/GB3_Python1 vous trouverez, avec un retard de quelques jours par rapport au cours, des notes de cours et les feuilles de TD.

1 Installation de Python

Vérifiez tout d'abord que Python ne soit pas déjà installé sur votre machine. Si oui, passez cette partie du TD...

- Cherchez (par exemple *via google*) la page officielle de Python et choisissez la version stable la plus récente.
- Ne pas utiliser les « sources » : ce sont des textes de programmes qui doivent être compilés pour fabriquer les commandes utiles pour le langage Python (python, IDLE, etc.). Mieux vaut utiliser les installations déjà compilées, qui sont alors dépendantes du système que vous utilisez : Windows, Linux, Mac OS 10, etc.
- Chargez celle qui convient. Cela installera dans vos menus diverses commandes utiles pour Python.

Lancez parmi ces commandes « IDLE ».

- Une fenêtre apparaît, avec « >>> » et le curseur qui attend que vous commenciez à programmer, et donnera les résultats au fur et à mesure de vos commandes.
- Dans le menu de IDLE, *File/New* permet d'accéder à un éditeur. Dans cet éditeur on peut modifier à loisir un texte de programme en Python mais, contrairement à la première fenêtre IDLE, cela ne l'exécute pas au fur et à mesure des lignes tapées. Pour exécuter le programme, il faut choisir *Run* dans le menu de l'éditeur : cela demande un nom de fichier la première fois, où le texte du programme sera sauvegardé avant d'être exécuté.
- Ultérieurement, dans le menu de IDLE, *File/Open* permet d'éditer un ancien fichier de programme de son choix. Il faut donc choisir des noms de fichier « parlants »...

2 Premiers calculs

Exercice 1 : Tapez les exemples vus en cours. À cette occasion, on découvre entre autres :

- que les accents ne sont pas bien gérés, mieux vaut les éviter
- 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
- etc.

Exercice 2 : 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.

3 Définition de la syntaxe des expressions booléennes

Un programme dans un langage donné est une suite de symboles, construite en respectant certaines règles. La *syntaxe* explique comment sont construits les mots et les phrases valides du langage (les expressions), la *sémantique* en donne le sens.

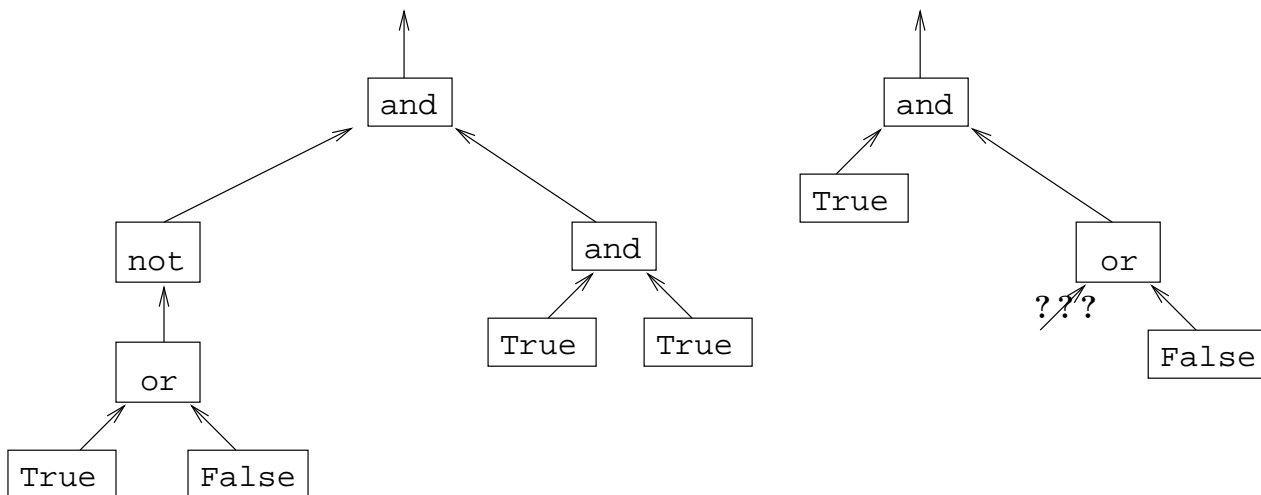
Ainsi, une expression booléenne est :

- une constante booléenne (`True` ou `False`)
- ou une expression unaire de la forme « `not exp` » où `exp` est une expression booléenne plus simple
- ou une expression binaire de la forme « `exp1 and exp2` » où `exp1` et `exp2` sont des expressions booléennes plus simples
- ou une expression binaire de la forme « `exp1 or exp2` » où `exp1` et `exp2` sont des expressions booléennes plus simples

Exemples d'expressions correctes avec ces règles : « `True` », « `True and False` », « `(True and False) or True` ».

Expressions incorrectes : « `True and` », « `True and or False` » ou « `True not False` ». Elles comportent des *erreurs syntaxiques* parce qu'elles ne peuvent pas être construites en utilisant uniquement les règles précédentes.

Pour respecter ces règles, il faut pouvoir fabriquer un *arbre syntaxique* tel que, en chaque nœud de l'arbre, l'opération booléenne qui sera calculée en dernier en constitue la *racine*, et les expressions qu'elle relie en sont les *sous-arbres*. Par exemple l'expression « `not(True or False) and (True and True)` » est correcte alors que « `True and or False` » ne l'est pas. En effet, elles conduisent respectivement aux formes d'arbre syntaxique suivantes :



Exercice 3 : Tracez les arbres syntaxiques des expressions précédentes. Vérifier que l'arbre est bien formé pour les expressions correctes, mal formé ou impossible à compléter sinon.

Un opérateur binaire peut être placé, comme ici, en position *infixe* c'est-à-dire, entre ses arguments. Pour certains opérateurs ou dans certains langages de programmation, il peut aussi être placé en position *préfixe* (devant ses arguments : « or (exp,exp) ») ou plus rarement *postfixe* (derrière ses arguments : « exp exp or »). Dans le langage Python, and et or sont infixes mais les fonctions que vous programmerez vous-mêmes seront préfixes.

4 Priorité et parenthésage à gauche

Nous allons maintenant travailler avec des expressions arithmétiques que nous pouvons définir de la façon suivante. Une expression arithmétique est :

- un nombre entier relatif
- ou une expression binaire de la forme « $exp1 + exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques
- ou une expression binaire de la forme « $exp1 - exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques
- ou une expression binaire de la forme « $exp1 * exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques
- ou une expression binaire de la forme « $exp1 / exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques
- ou une expression binaire de la forme « $exp1 \% exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques
- ou enfin une expression binaire de la forme « $exp1 ** exp2$ » où $exp1$ et $exp2$ sont des expressions arithmétiques

Exercice 4 : Tracez les arbres syntaxiques des expressions suivantes : « $(1 + (5 \% 2)) - 4$ », « $(5 ** 2) - (3 / 2)$ » et « $(4 ** / 4) +$ ».

Vérifiez pour chacune si l'arbre est bien formé (expressions correctes) ou mal formé et dans ce cas indiquez les malformations.

L'expression $2 * 3 + 4$ peut *a priori* être interprétée comme $A = (2 * 3) + 4$ ou comme $B = 2 * (3 + 4)$.

Exercice 5 : De combien de façon peut-on *a priori* évaluer l' expression $2*8/4+5$?

Il y a donc des ambiguïtés si l'on ne met pas toutes les parenthèses. Une solution consisterait à ajouter des parenthèses pour lever l'ambiguïté mais, pour limiter l'usage de parenthèses dans la notation infix, on peut définir un ordre d'utilisation, appelé *priorité* sur les opérateurs.

Une opération $op1$ est prioritaire devant $op2$ si $op1$ « choisit » ses arguments avant $op2$. Par exemple, « $*$ » est prioritaire devant « $+$ », $2 * 3 + 4$ se lit donc $(2 * 3) + 4$. Si l'on souhaite un groupement différent des arguments, il faut alors utiliser des parenthèses : $2 * 2 + 4$ vaut 8 alors que $2 * (2 + 4)$ vaut 12. Si deux opérations ont même priorité, leurs arguments sont lus de gauche à droite. Par exemple, « $*$ » et « $/$ » ont même priorité donc $2 * 3 / 4$ signifie $(2 * 3) / 4$.

En Python :

- les opérations sur les nombres ont la priorité, avec de plus $*$ / $\%$ et $**$ prioritaires sur $+$ et $-$
- les comparaisons viennent ensuite
- enfin des opérations booléennes
- les opérateurs de même priorité sont parenthésés à gauche.

Exercice 6 : Inventez une demi-douzaine d'expressions dont le résultat, une fois calculé sous Python, permet de mettre en évidence ces priorités.