Travaux Pratiques
Initiation à la programmation avec Python 3
Feuille n.1

Basé sur le cours de Jean-Vincent Loddo

1 Utiliser les variables

Les variables sont des boîtes dans lesquelles nous pouvons stocker des **valeurs**. En Python, les valeurs peuvent être de plusiers **types**, par exemple des entiers (`int`), des nombres réels (en virgule flottante) (`float`), des booléens (`bool`) ou des chaînes de caractères (`str`). Ouvrez un interpréteur Python (version 3) dans un terminal, si possible `ipython3`, sinon `python3`, et tapez la séquence de commandes suivantes :

```
X="un bon TP"
print("Je vous souhaite",X)
```

Transformez ces deux lignes en script (programme) en les plaçant dans un fichier, par exemple `foo.py`, en ajoutant le shebang [1] `#!/usr/bin/python3` comme première ligne. Vous pouvez utiliser les éditeurs de texte `emacs` ou `gedit` pour écrire et modifier votre programme. Rendez ensuite le fichier exécutable (`chmod +x foo.py`) et exécutez-le dans un terminal (`./foo.py`). Vous êtes en ce moment à la fois le **développeur** de ce petit programme (celui qui l'édite) et un **utilisateur** (quelqu'un qui l'exécute). Essayez à présent (en tant que développeur) de remplacer la ligne d'affectation de la variable `X` par une ligne qui demandera à l'utilisateur (donc à vous même, plus tard) de saisir une phrase qui sera stockée dans `X` :

```
#!/usr/bin/python3
print('Je suis un gentil petit programme. Que voulez-vous que je vous souhaite ?')
X = input()
print('Alors je vous souhaite',X)
```

Remark 1.1. Pour écrire des chaînes de caractères en Python on peut les entourer soit par des guillemets simple (`'`) que par des doubles (`"`). Si la chaîne contient des retours à la ligne, il faudra utiliser trois guillemets simple (`'''`) ou trois double (`"""`).

L'outil (fonction) `input` permet d'afficher un message avant de gérer la saisie de l'utilisateur. Vous pouvez en profiter pour éliminer la 2ème ligne (c'est-à-dire le 1er `print`), ou pour aller à la ligne après la première phrase affichée. Testez la version suivante :

```
#!/usr/bin/python3
print('Je suis un gentil petit programme.')
X = input('Que voulez-vous que je vous souhaite ? ')
print('Alors je vous souhaite',X)
```

1. Wikipedia : Le shebang, représenté par #!, est un en-tête d'un fichier texte qui indique au système d'exploitation que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes ou instructions) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

2 La programmation est un jeu de Lego

La programmation est un jeu de Lego : vous assemblez les valeurs (`'Je vous souhaite'`, un caractère blanc et le contenu de `X`) pour construire de nouvelles valeurs et vous assemblez les services rendus par des (sous-)programmes (`print` et `input`) pour construire de nouveaux (sous-)programmes. Dans la suite on vous demandera de taper du code pour tester et comprendre son fonctionnement. Dans ce code, il pourra y avoir des **commentaires** (que vous ne devez pas saisir). En Python, tout ce qui suit le caractère `#` est un commentaire ignoré par l'interpréteur (donc une ligne qui **commence** par `#` est **complètement** ignorée).

Assemblage des *valeurs* en Python. En Python, l'assemblage (ou construction) de valeurs dépend de leur type, voici quelques **exemples à tester** avec l'interpréteur `ipython3` ou `python3` :

1. s'il s'agit de nombres, entiers (`int` ou `long`) ou flottants (`float`), on peut construire de nouvelles valeurs par les opérateurs arithmétiques `+ - * / %` ou avec des outils de base ("fonctions" ou "opérateurs") que Python comprend comme la plupart des langages. Essayez chaque expression (calcul) à gauche (pas besoin d'écrire les commentaires à partir du caractère `#`) :

```
1 + (3 * 5) // 2          # (rend 8) des opérations sur les entiers => résultat entier
1 + (3 * 5) / 2           # (rend 8.5) / est la division entre flottants => résultat flottant
1.0 + (3 * 5) // 2        # (rend 8.0) il y a un flottant (1.0) => résultat flottant
11 % 3                    # (rend 2) l'opérateur % donne le reste de la division entière
2.71 / 3.14               # une division entre flottants
2 ** 10                   # (rend 1024) l'opérateur ** est la puissance, ici on calcule 2^10
round(3.1415 * 10)        # (rend 31) arrondir un flottant
round(3.1415 * 10, 2)     # (rend 31.42) arrondir au n-ème chiffre décimal, ici n=2
```

2. s'il s'agit de booléens (valeurs de vérité, type `bool`), on peut les combiner avec les opérateurs logiques, ce qui est bien pratique pour les prises de décision du robot (`if-then-else`, `while`). Essayez :

```
(1>10)                    # True ou False ?
(5<7)                     # True ou False ?
(1>10) or (5<7)           # True ou False ?
A=(1>10)                  # Une affectation ne rend pas de résultat
A                         # True ou False ?
X=7                       # Une affectation ne rend pas de résultat
B=(5<X)                   # Une affectation ne rend pas de résultat
B                         # True ou False ?
C=False                   # Une affectation ne rend pas de résultat
A and B                   # True ou False ?
A or B or C               # True ou False ?
not (B and C)             # True ou False ?
```

Remark 2.1. Plusieurs **instructions** peuvent être placées sur une même ligne en les séparant par des `";"`. Par exemple, on aurait pu mettre sur une même ligne toutes les affectations de variables précédentes :

```
A=(1>10); B=(5<7); C=False
```

Remark 2.2. Si **toutes** les instructions sont des **affectations**, alors elles peuvent être réunies sur une même ligne en séparant les variables à gauche et les valeurs correspondantes à droite par des `","`. Par exemple :

```
A, B, C = (1>10), (5<7), False
```

3. s'il s'agit de chaînes de caractère (`str`), on fabrique de nouvelles chaînes de plusieurs manières possibles. On va distinguer trois façons habituelles :
   (a) on "colle" les chaînes entre elles (**concaténation** ou **juxtaposition**) avec `+` et `*` :

```
'salut' + 'le monde'      # rendu ?
X = 'salut'               # Une affectation ne rend pas de résultat
X + 'le monde'            # rendu ?
X + ' le monde'           # rendu ?
Y="Abc"                   # Une affectation ...
Y=Y*4                     # Une affectation ...
print(Y)                  # print AFFICHE sur le terminal (mais ne rend pas de résultat)
Z=print(Y)                # ERREUR ! justement parce qu'il ne rend pas de résultat !
```

(b) on “extrait” (**extraction**) des sous-chaînes avec `[]` :

```
Z="ABCDEFGHIJK"          # Une affectation ne rend pas de résultat
Z[0]                     # rendu ?
Z[1]                     # rendu ?
Z[0:2]                   # rendu ?
Z[2:6]                   # rendu ?
Z[:3]                    # rendu ?
Z[3:]                    # rendu ?
Z[0:10:2]                # rendu ?
```

(c) on “agence” des n-plets de pièces (de tout type) dans un gabarit (**chaîne de format**) avec l’opérateur `%` (le même symbole utilisé pour le reste de la division entière). Ça s’appelle le “formatage de chaînes de caractères” ou “string formatting”, mais nous pourrons continuer de l’appeler “**agencement**”[2] :

Dans l’agencement, on précise comment la pièce doit être représentée, par exemple : comme un entier avec `%d`, comme un flottant avec `%f`, comme une chaîne avec `%s`) :

```
"Bonjour %s, comment allez-vous ?" % "Emmanuel"
"Chers %s et %s, comment allez-vous ?" % ("Emmanuel","Brigitte")
"Voici une chaîne: %s, voici un entier: %d, voici un flottant: %f" % (X, 42, 3.14)
```

Remark 2.3. Les opérateurs `+` et `%` sur les chaînes de caractères s’écrivent de la même manière que l’addition et le reste de la division (modulo) entre nombres (entiers ou réels flottants) mais n’ont aucun autre rapport avec ces opérations arithmétiques ; Python comprend qu’il s’agit de faire une concaténation si les deux arguments sont des chaînes (`str`), il comprend qu’il s’agit de faire une addition s’il s’agit de nombres (`int`, `long`, `float`). De même, concernant `%`, il comprend si faire le reste de la division ou l’agencement. **Attention** : pas tous les langages de programmation sont aussi permissifs que Python. Si on est content de ce choix des concepteurs, on dira che Python est plutôt *cool*, sinon on dira qu’il est *laxiste* parce qu’il laisse passer trop de bugs détectables. Chaque programmeur est libre d’avoir son opinion...

3 Comment rendre sa copie à chaque fin de séance TP

Déposer simplement votre fichier source (`.py`) sur **Moodle** dans le répertoire du module **M1207**. Si vous avez travaillé sur plusieurs fichiers, créez une archive (`.tar.gz`) et déposez-la, par exemple :

```
tar -czvf compte-rendu-seance1.tar.gz  exo1.py exo2.py exo3.py ...
```

ou, en utilisant les motifs de fichiers du shell (mais attention à ne pas en mettre plus que nécessaire) :

```
tar -czvf compte-rendu-seance1.tar.gz  exo*.py
```

Une méthode alternative est de faire une simple concaténation de vos fichiers source, par exemple :

```
head -n 10000 exo*.py > compte-rendu-seance1.py
```

(on suppose ici que vos fichiers contiennent moins de dix mille lignes...)

2. Image source (équipe) à l’adresse `https://openclipart.org/image/2400px/svg_to_png/227299/boys_n_ball.png`

4 Exercices

Construire un programme **indépendant** (*shebang*, `chmod +x`) pour chaque exercice. Ceci n'empêche pas de **tester des bouts de code dans l'interpréteur, avant de les intégrer dans le programme**.

Tous les résultats présentés par les programmes doivent être clairement compréhensibles à l'utilisateur (sinon les programmes ne rendent pas vraiment leur service !).

Remarque 1 : après l'appel à la fonction `input()`, qui rend une chaîne de caractère (type `str`), il faudra faire appel à la fonction `int()` ou à la fonction `float()` pour convertir cette chaîne (par exemple `"123"`) dans le nombre correspondant (entier `123` ou flottant `123.0`).

Remarque 2 : on construira des programmes dont le **résultat** sera une chaîne de caractère déposée (par `print`) sur la sortie standard du programme (normalement le terminal). Ce mécanisme n'est pas à confondre avec le résultat rendu par les **fonctions**, qui sont des sous-programmes :

- but de nos premiers programmes : rendre (avec `print`) à l'utilisateur ou à un autre programme (penser aux tubes `|`) une chaîne de caractère sur la sortie standard
- but d'un sous-programme : rendre (avec `return`) une valeur, d'un certain type (pas forcément `str`), à une autre partie du code ayant besoin de cette valeur
- les sous-programmes qui se comportent comme le programme principal, et ne rendent aucun résultat aux autres sous-programmes, s'appellent des **procédures**, et `print` est un des exemples plus simples. Dans la section dédiée à la tortue, nous écrirons d'autres procédures.

Pour souligner ce caractère spécifique des programmes que nous construisons aujourd'hui par rapport aux sous-programmes que nous construirons dans les prochains TP, dans la suite de cet énoncé le mot **affiche** sera en gras et suivi du mot `print`. En effet, vous devez l'avoir déjà lu quelque part, `print` ne rend aucun résultat, dans le sens où il ne rend aucun résultat exploitable par un autre sous-programme.

4.1 Types, conversions et opérateurs de base

1. Écrire et tester un programme `triple.py` qui demande à l'utilisateur un nombre et **affiche** (`print`) ensuite le triple de ce nombre.

2. Écrire et tester un programme `rectangle.py` qui demande à l'utilisateur la *longueur* et la *largeur* du rectangle, puis calcule et **affiche** (`print`) le *périmètre* et la *surface*. L'affichage se fera sur deux lignes, une pour chaque résultat.

3. Écrire et tester un programme `cercle.py` qui demande à l'utilisateur le *rayon* du cercle, puis calcule et **affiche** (`print`) le *diamètre* (deux fois le rayon), le *périmètre* (π fois le diamètre) et la *surface* (π fois le carré du rayon). En ce qui concerne π, utilisez simplement une variable nommée `PI` affectée à `3.141592`. L'affichage se fera sur trois lignes, une pour chaque résultat.

4. Écrire et tester un programme `sms.py` qui rend le service de fabriquer un SMS d'amour (oui, c'est très utile) : le programme demande le nom du destinataire et **affiche** (`print`) "*Je suis passée à côté d'un ange qui ma demandé : C'est quoi ton vœu ? je lui ai répondu : ça serait d'être à côté de ... qui lit en ce moment ce message.* "[3] où les pointillés seront évidemment remplacés par le nom du destinataire. Utiliser plusieurs lignes avec les séparateurs triple quote (`'''` ou `"""`).

4.2 Conditionnelle ("prise de décision")

1. Écrire et tester un programme `maximum.py` qui demande à l'utilisateur deux nombres, puis **affiche** (`print`) la phrase "*Le nombre ... est certainement plus grand que ...* ", en remplissant correctement les pointillés.

2. Modifier le programme précédent en prévoyant d'**afficher** (`print`) éventuellement la phrase "*Les deux nombres sont égaux*", lorsque les deux nombres saisis seront égaux. Résoudre avec deux conditionnelles (`if-else`) imbriquées.

3. Source `http://message-d-amour.blogspot.fr/`

3. Réutiliser le code écrit dans `rectangle.py` et `cercle.py` pour écrire un programme `geometrie.py` qui demande à l'utilisateur le type de figure sur lequel il souhaite travailler (rectangle ou cercle), et ensuite se comporte comme `rectangle.py` ou `cercle.py` selon le choix de l'utilisateur.

4. Modifier `sms.py` de façon que le programme demande en plus l'âge du destinataire. Si celui-ci est jeune (moins de 13 ans), il **affichera** (`print`) le SMS "*Je peux jouer à CoD, mes parents sont partis chez tante Éloïse*". En revanche, si le destinataire est un peu trop âgé (plus de 110 ans) afficher le message d'erreur "*Inutile de fabriquer un SMS, pensez au courrier ordinaire*". Dans les cas intermédiaires, il aura le comportement précédent.

5 La tortue

Dans l'interpréteur Python, l'instruction suivante :

```
from turtle import *
```

a l'effet d'apporter un certain nombre d'outils pour dessiner sur un écran graphique. Le trait correspond à la piste laissée derrière elle par une tortue virtuelle dont on peut contrôler les mouvements. Les fonctions principales qui deviennent accessibles sont :

`reset()`	on efface le tableau et on recommence
`position()`	pour connaître sa position actuelle
`goto(x,y)`	aller à la position de coordonnées (x,y)
`forward(x)`	avancer de x pixels
`backward(x)`	reculer de x pixels
`up()`	relever le crayon pour se déplacer sans laisser de traces
`down()`	abaisser le crayon pour tracer
`color(x)`	utiliser la couleur du crayon x
`left(x)`	tourner à gauche d'un angle x
`right(x)`	tourner à droite d'un angle x
`width(x)`	utiliser un tracé d'épaisseur x
`fill(x)`	remplir ou pas (x de type `bool`) un contour fermé
`write(x)`	écrire le texte x (de type `str`)

Avec l'interpréteur `ipython3` vous pouvez avoir de l'aide sur toutes ces fonctions en utilisant le point d'interrogation juste après le nom de la fonction. C'est une des raisons pour lesquelles cet interpréteur est préférable à `python3` (sans le `"i"`). Par exemple, il vous suffira de taper `color?` pour avoir de l'aide sur la fonction `color()`.

5.1 Exercices

1. Dessiner un carré de longueur 100 pixels. Écrire ensuite un programme indépendant `dessine_carre.py` qui demande à l'utilisateur la longueur en pixels, puis dessine le carré de cette longueur.
2. Dessiner un rectangle de longueur 200 pixels et hauteur 100 pixels. Écrire ensuite un programme indépendant `dessine_rectangle.py` qui demande à l'utilisateur la longueur et la hauteur en pixels, puis dessine le rectangle correspondant.
3. Dessiner le début d'une spirale rectangulaire dont le premier trait sera de longueur 20. Écrire ensuite un programme indépendant `dessine_spirale.py` qui demande à l'utilisateur la longueur du trait initial en pixels, puis dessine (un début) de spirale.
4. Répéter l'exercice 1. en faisant un premier carré, puis un second de taille double, contenant le premier.
5. Répéter l'exercice 2. en faisant un premier rectangle, puis un second de taille double, contenant le premier.