Annexe B. Aide-mémoire

B.1 Environnement de TP

Deux environnements de travail sont utilisés. L'un, *Python Tutor*, est disponible depuis le site web du cours. Il est utilisable depuis n'importe quel navigateur web supportant javascript. L'intérêt de Python Tutor est qu'il permet d'observer l'exécution du programme pas à pas. Le défaut est qu'il est difficile de sauvegarder ses programmes, et il limite le nombre de pas à 10000 seulement.

L'autre, *spyder*, contient à la fois un éditeur de texte et un interpréteur interactif, ce qui permet à la fois de facilement enregistrer ses programmes, et d'expérimenter avec. Il est disponible sur les postes du CREMI, mais vous pouvez également l'installer sur votre propre ordinateur, cf les instructions sur moodle.

Selon les situations, on préfèrera donc utiliser l'un ou l'autre.

Environnement Python Tutor

L'environnement Python Tutor est disponible depuis le site web du cours :
https://services.emi.u-bordeaux.fr/pythontutor/visualize-initinfo.html
Il vous demande votre *login* et *mot de passe* CREMI.

Il suffit alors de taper le code et lancer l'exécution.

On obtient alors une page avec à gauche le code en cours d'exécution ainsi qu'une barre de progression et quelques boutons permettant d'avancer et reculer dans l'exécution, et à droite l'état des variables du programme. On peut ainsi observer l'évolution des variables pendant l'exécution pas à pas.

Le lien *Éditer le code* permet de retourner modifier le programme, avant de l'exécuter de nouveau.

Environnement de travail Linux

Deux systèmes d'exploitation peuvent être installés sur les ordinateurs personnels auquels vous avez accès : Linux et Windows. Les travaux pratiques seront entièrement réalisés dans le cadre du système d'exploitation Linux.

Connexion/Déconnexion

Tapez votre nom de *login*, puis votre *mot de passe* dans la fenêtre de connexion [1]. À la fin du TP, il ne faut **pas oublier** de se déconnecter (se "déloguer" du système).

1. Si le système Linux n'est pas déjà chargé, il faut redémarrer l'ordinateur et choisir l'option Linux au moment opportun.

B.2 Comprendre les messages d'erreur

Nous reprenons ici la liste des messages d'erreur que vous verrez souvent, et dans les pages suivantes des explications du problème et ses solutions potentielles. Cette liste est également disponible, de manière plus pratique, sur la page moodle du cours, "Comprendre les messages d'erreur".

- B.2.1 : NameError : name 'blablaBla' is not defined
- B.2.2 : SyntaxError : invalid syntax
- B.2.3 : SyntaxError : expected an indented block
- B.2.4 : SyntaxError : unindent does not match any outer indentation
- B.2.5 : SyntaxError : unexpected indent
- B.2.6 : SyntaxError : can't assign to operator
- B.2.7 : SyntaxError : can't assign to function call
- B.2.8 : SyntaxError : unexpected EOF while parsing
- B.2.9 : ValueError : math domain error
- B.2.10 : TypeError : 'list' object cannot be interpreted as an integer
- B.2.11 : TypeError : f() takes 2 positional arguments but 3 were given
- B.2.12 : TypeError : f() missing 1 required positional argument : 'y'
- B.2.13 : TypeError : 'int' object is not iterable
- B.2.14 : TypeError : 'int' object is not callable
- B.2.15 : TypeError : unorderable types : ___c_node() <= int()
- B.2.16 : IndexError : image index out of range
- B.2.17 : ImportError : No module named 'bibimages'
- B.2.18 : FileNotFoundError : [Errno 2] No such file or directory : 'tgv2005.dot'

B.2.1 NameError : name 'blablaBla' is not defined

Erreur : le nom 'blablabla' n'est pas défini. Soit vous utilisez la variable (ou la fonction) blablabla avant de la définir soit vous vous êtes trompé en tapant le nom de la variable (ou de la fonction) blablabla. Attention : python est sensible à l'utilisation des majuscules. Il peut aussi arriver que vous ayez oublié des guillemets autour d'une chaîne de caractères (voir exemple 3).

Exemple 1 :

```
def blablaBla(x):
    return x * x
y = blablabla(2)
```

```
NameError: name 'blablabla' is not defined
```

Origine du problème : nous avons mal orthographié “blablabla” : il y un B majuscule dans le 3ième “bla” de la définition de la fonction et pas de majuscule dans l’appel de la fonction.

Exemple 2 :

```
j = 5
x = 2 * i
i = 4
```

```
NameError: name 'i' is not defined
```

Origine du problème : on utilise le nom ’`i`’ dans cette expression alors qu’il n’a jamais été défini avant. Soit on aurait dû initialiser la variable ’`i`’ avant (déplacer l’instruction `i = 4` avant l’instruction `x = 2 * i`), soit on a mal orthographié la variable (on voulait taper ’`j`’ au lieu de ’`i`’).

Exemple 3 : (un peu plus subtil)

```
def f(x):
    m = x*x
    return m

f(5)
y = m
```

```
NameError: name 'm' is not defined
```

Origine du problème : la variable “`m`” est une variable locale à la fonction “`f`”, elle ne “vit” qu’à l’intérieur de la fonction “`f`”.

Exemple 4 :

```
monImage = ouvrirImage(teapot.png)
```

Origine du problème : comme on a oublié les guillemets, python pense que teapot est le nom d’une variable... qui n’est donc pas définie.

Solution :

```
monImage = ouvrirImage("teapot.png")
```

B.2.2 SyntaxError : invalid syntax

Erreur : erreur de syntaxe. Cela veut dire que l’interpréteur est perdu car il n’arrive pas à déchiffrer votre programme. L’origine du problème n’est pas forcément à l’endroit pointé par l’interpréteur, mais peut prendre sa source plus en amont dans le code, comme nous allons le voir dans les exemples ci-dessous.

Exemple 1 :

```
def f(x)
        ^
SyntaxError: invalid syntax
```

Origine du problème : on a oublié les ' :' à la fin de la ligne.
Solution :

```
def f(x):
```

Exemple 2 :

```
if i = 6:
     ^
```

Origine du problème : après le "`if`", `python` s'attend à une expression booléenne, Ici, il se retrouve avec une affectation, et est donc perdu.
Solution : remplacer l'opérateur d'affectation "`=`" par l'opérateur de test d'égalité "`==`"

```
if i == 6:
```

Exemple 3 :

```
 if i == 9
          ^
SyntaxError: invalid syntax
```

Origine du problème : on a oublié les "`:`" à la fin de la ligne.
Solution :

```
if i==9 :
```

Exemple 4 :

```
def f(x):
    return sqrt((x*x - x)
print(f(5))
^
```

Origine du problème : il manque une parenthèse fermante à la ligne précédente.
Solution :

```
def f(x):
    return sqrt((x*x - x))
print(f(5))
```

Exemple 5 :

```
def f(x):
    return sqrt((x*x - x)

print f(5)
     ^
```

Origine du problème : en python 3 (contrairement à python 2), `print` est une fonction comme les autres. Pour appeler cette fonction, il faut donc utiliser des parenthèses.

Solution :

```
def f(x):
    return sqrt((x*x - x)

print (f(5))
```

B.2.3 SyntaxError : expected an indented block

Erreur : problème d'indentation. Le plus fréquemment, il manque une ou plusieurs tabulations à l'endroit pointé par le message d'erreur (voir exemple 1). Parfois, l'origine du problème vient de plus haut dans votre programme. Par exemple, lorsqu'on a commencé une fonction, un `if` ou un `for`, que l'on n'a pas fait suivre par une instruction (voir exemple 2) ou qu'on a commenté l'instruction en question (exemple 3).

Exemple 1 :

```
if i == 9:
toto = 9
```

Solution :

```
if i == 9 :
    toto = 9
```

Exemple 2 :

```
def f(x):

y = 5*2
```

Origine du problème : vous avez déclaré une fonction sans définir son code.

Solution : rajouter un `return` dans votre fonction.

```
def f(x):
    return

y = 5*2
```

Exemple 3 :

```
for i in range(5):
##   print(i)

y = 5*2
```

Origine du problème : vous avez commenté une partie du code, ce qui perturbe son interprétation.

Solution : commenter tout le bloc.

```
##for i in range(5):
##    print(i)

y = 5*2
```

B.2.4 SyntaxError : unindent does not match any outer indentation level

```
if i == 9:
    titi = 5
  toto = 4
  ^
SyntaxError: unindent does not match any outer indentation level
```

Origine du problème : le niveau d'indentation (nombre d'espaces) ne correspond à aucun niveau d'indentation externe. Dans l'exemple ci-dessus, l'instruction "`toto = 9`" est soit trop à gauche, soit trop à droite. Cela arrive souvent lorsqu'on mélange les caractères de tabulation et d'espace.

Solution :

```
if i == 9:
    titi = 5
    toto = 4
```

ou (selon le contexte) :

```
if i == 9:
    titi = 5
toto = 4
```

B.2.5 SyntaxError : unexpected indent

Exemple :

```
def f(x):
    y = x*x
        return y
```

Origine du problème : il y a un niveau d'indentation de trop (il y a trop de tabulations) sur cette ligne.

Solution : supprimer les tabulations superflues.

```
def f(x):
    y = x*x
    return y
```

B.2.6 SyntaxError : can't assign to operator

```
i+1 = 5
^
```

Origine du problème : on ne peut pas affecter une valeur à une expression qui ne représente pas une variable. Ici, à gauche du symbole d'affectation "`=`", on trouve l'expression "`i + 1`, qui ne définit pas une variable dans laquelle stocker la valeur qui est à droite du symbole d'affectation. Ici, on voulait peut-être écrire "`i = 5 - 1`".

B.2.7 SyntaxError : can't assign to function call

```
def f(x):
    return x*x

f(5) = y
^
SyntaxError: can't assign to function call
```

Origine du problème : on ne peut pas affecter une valeur au résultat d'un appel de fonction. En effet, le résultat d'un appel de fonction est une valeur constante (par exemple, "12"), et non une variable permettant de stocker une valeur.

Solution : vous vouliez probablement écrire :

```
def f(x):
    return x*x

y = f(5)
```

B.2.8 SyntaxError : unexpected EOF while parsing

Erreur : on arrive à la fin du fichier (EOF = End Of File) alors qu'on n'a pas terminé l'instruction ou le bloc d'instruction en cours.

Exemple :

```
def f(n):
    s = 0
    for i in range(n):

### fin du fichier ####
```

Solution : compléter la fonction :

```
def f(n):
    s = 0
    for i in range(n):
        s = s + i
    return s
### fin du fichier ####
```

B.2.9 ValueError : math domain error

Exemple :

```
y = -5
x = sqrt (y)
```

Origine du problème : la fonction `sqrt` ne prend en paramètre que des nombres positifs.

B.2.10 TypeError : 'list' object cannot be interpreted as an integer

Erreur : un objet de type "`list`" ne peut pas être interprété comme un entier.

Exemple 1 :

```
L = [5,2,3]
for i in range(L):
    ...
```

Ici on passe une liste en paramètre à la fonction range() alors qu'elle s'attend à avoir un entier.

Solution : ne pas utiliser la fonction `range`, mais directement la liste :

```
L = [5,2,3]
for elt in L:
    ...
```

B.2.11 TypeError : f() takes 2 positional arguments but 3 were given

Erreur : la fonction "`f()`" prend normalement 2 arguments et on l'appelle en lui en passant 3.

Exemple :

```
def f(x,y):
    return x + y

print(f(5,6,7))
```

Solution : appeler la fonction avec le bon nombre d'arguments

B.2.12 TypeError : f() missing 1 required positional argument : 'y'

Erreur : on appelle la fonction 'f()' en ne lui passant pas assez d'arguments.

Exemple :

```
def f(x,y):
    return x + y

print(f(5))
```

Solution : appeler la fonction avec le bon nombre d'arguments

B.2.13 TypeError : 'int' object is not iterable

Erreur : l'interpréteur s'attend à avoir une liste, mais se retrouve avec un entier.

Exemple :

```
n = 5
for i in n:
    print(i)
```

Solution :

```
n = 5
for i in range (n):
    print(i)
```

B.2.14 TypeError : 'int' object is not callable

Erreur : vous utilisez une variable (ici de type entier) comme si c'était une fonctionn c'est-à-dire en lui passant des arguments entre parenthèses.

Exemple 1 :

```
def f(x):
    return x*x

y = 2
z = y(5)
```

Solution : appeler la fonction `f` et non la variable `y` :

```
def f(x):
    return x*x

y = 2
z = f(5)
```

Exemple 2 :

```
monImage = nouvelleImage(300,200)
for y in range(largeurImage(monImage)):
    colorierPixel(monImage,0,y(255,255,255))
```

Origine du problème : ici, on a oublié une virgule.
Solution :

```
monImage = nouvelleImage(300,200)
for y in range(largeurImage(monImage)):
    colorierPixel(monImage,0,y,(255,255,255))
```

B.2.15 TypeError : unorderable types : __c_node() <= int()

Erreur : on essaye de comparer des choses incomparables, comme par exemple un sommet avec un nombre.

paragraphExemple :

```
c = 5
for s in listeSommets(G):
    if s <= c:
```

Solution : ne pas utiliser le sommet `s` mais un entier (comme par exemple le degré de `s`) :

```
c = 5
for s in listeSommets(G):
    if degre(s) <= c:
```

B.2.16 IndexError : image index out of range

Erreur : vous essayez d'accéder à un pixel qui n'est pas dans l'image.

paragraphExemple :

```
monImage = nouvelleImage(300,200)
colorierPixel(monImage,200,200,(255,255,255))
```

Origine du problème : ici, l'image est de largeur 200 et le pixel le plus à droite a pour abscisse 199 (car la numérotation commence à 0).

B.2.17 ImportError : No module named 'bibimages'

```
from bibimages import *
```

Origine du problème : python ne trouve pas le fichier `bibimages.py`. Il y a deux causes principales à ce problème :

1. soit vous n'avez pas encore récupéré le fichier `bibimages.py` sur la page https://moodle.u-bordeaux.fr/course/view.php?id=14677 ;
2. soit vous l'avez bien récupérée, mais elle est dans un autre répertoire.

B.2.18 FileNotFoundError : [Errno 2] No such file or directory : 'tgv2005.dot'

Erreur : le fichier `'tgv2005.dot'` n'est pas présent dans le répertoire courant. Soit vous n'avez pas téléchargé ce fichier, soit vous l'avez téléchargé dans un autre répertoire.

Solution : vérifiez le contenu du répertoire courant.

B.3 Utilisation de la bibliothèque de graphes

Il faut commencer par importer le module `bibgraphes` :

```
from bibgraphes import *
```

Attention à respecter la distinction entre minuscules et majuscules :

La fonction `ouvrirGraphe` permet d'ouvrir un graphe existant au format `.dot`. Le format `.dot` est très standard et utilisé très largement pour sauvegarder des graphes, en fait c'est un simple fichier texte, vous pouvez l'ouvrir pour voir à quoi il ressemble ! On peut également écrire des graphes, notamment pour sauvegarder un coloriage.

`ouvrirGraphe(nom:str) -> graphe`	Ouvre le fichier *nom* et retourne le graphe contenu dedans, par exemple : `tgv = ouvrirGraphe("tgv2005.dot")`
`ecrireGraphe(G:graphe, nom:str)`	Sauvegarde le graphe `G` dans le fichier *nom*, par exemple : `ecrireGraphe(tgv, "fichier.dot")`
`afficherGraphe(G:graphe)`	dessine le *graphe* `G`, par exemple : `afficherGraphe(tgv)`

L'argument `G` est un graphe	
`listeSommets(G:graphe) -> list`	retourne la *liste* des *sommets* de `G`, par exemple : `l = listeSommets(tgv)`
`nbSommets(G:graphe) -> int`	retourne le *nombre* de sommets de `G`, c'est-à-dire la *taille* de la liste précédente, par exemple : `n = nbSommets(tgv)`
`sommetNom(G:graphe, etiquette:str) -> sommet`	retourne le *sommet* de `G` désigné par son *nom* (`etiquette`), par exemple : `bdx = sommetNom(tgv, "Bordeaux")`

La fonction `afficherGraphe` (aussi appelée `dessiner`) comporte des paramètres facultatifs :

`afficherGraphe(G:graphe,algo='neato')`	dessine le graphe `G` en utilisant un algorithme où les arêtes sont traitées comme des ressorts, par exemple : `afficherGraphe(tgv, algo='neato')`
`afficherGraphe(G:graphe,algo='circo')`	dessine le graphe `G` en utilisant un algorithme de placement des sommets sur un cercle, par exemple : `afficherGraphe(tgv, algo='circo')`

L'argument `s` est un sommet	
`listeVoisins(s:sommet) -> list`	retourne la *liste* des *voisins* du sommet `s`, par exemple : `L = listeVoisins(bdx)`
`degre(s:sommet) -> int`	retourne le *degré* du sommet `s`, qui est aussi la *taille* de la liste des voisins, par exemple : `n = degre(bdx)`
`nomSommet(s:sommet) -> str`	retourne le *nom* du sommet `s`, par exemple : `etiquette = nomSommet(bdx)`
`colorierSommet(s:sommet, c:str)`	colorie le *sommet* `s` avec la *couleur* `c`, par exemple : `colorierSommet(bdx, "red")` (ou `"green"`, `"blue"`, `"white"`, `"cyan"`, `"yellow"`, ...)
`couleurSommet(s:sommet) -> str`	retourne la *couleur* du *sommet* `s`, par exemple : `c = couleurSommet(bdx)`

`marquerSommet(s:sommet)`
 	
`demarquerSommet(s:sommet)`
	marque le sommet `s`, par exemple :
	`marquerSommet(bdx)`
	démarque le sommet `s`, par exemple :
	`demarquerSommet(bdx)`

`estMarqueSommet(s:sommet) -> bool`	retourne `True` si `s` est marqué, `False` sinon, par exemple : `b = estMarqueSommet(bdx)`
`listeAretesIncidentes(s)`	retourne la *liste* des arêtes *incidentes* à `s`, par exemple : `L = listeAretesIncidentes(bdx)`

L'argument `a` est une arête	
`nomArete(a:arete) -> str`	retourne le *nom* (étiquette) de `a`, par exemple : `nom = nomArete(a)`

`marquerArete(a:arete)`
 	
`demarquerArete(a:arete)`
	marque l'arête `a`, par exemple :
	`marquerArete(a)`
	démarque l'arête `a`, par exemple :
	`demarquerArete(a)`

`estMarqueeArete(a:arete) -> bool`	retourne `True` si `a` est marquée, `False` sinon, par exemple : `b = estMarqueeArete(a)`

Arguments : un sommet `s` et une arête `a`	
`sommetVoisin(s:sommet,a:arete)`	retourne le voisin du sommet `s` en suivant l'arête `a`, par exemple : `s = sommetVoisin(bdx, a)`

L'argument `u` est une liste	
`melange(u:list) -> list`	retourne une copie de la liste `u` dont les éléments ont été permutés de façon aléatoire, par exemple : `L = melange(L)`
`elementAleatoireListe(u:list)`	retourne un élément choisi aléatoirement dans la liste `u` si celle-ci est non vide. Si la liste `u` est vide la fonction retourne une erreur `IndexError`. Par exemple : `s = elementAleatoireListe(listeSommets(G))` où `G` contient un graphe

Les fonctions suivantes permettent de construire des graphes de taille variable :

`construireComplet(n:int) -> graphe`	retourne le graphe complet K_n , par exemple : `K5 = construireComplet(5)`
`construireBipartiComplet(m:int,n:int) -> graphe`	retourne $K_{m,n}$, par exemple : `K34 = construireBipartiComplet(3,4)`
`construireArbre(d:int,h:int) -> graphe`	retourne l'arbre de hauteur h dont chaque sommet possède d fils, par exemple : `arbre = construireArbre(2,3)`
`construireGrille(m:int,n:int) -> graphe`	retourne la grille rectangulaire avec m lignes et n colonnes, par exemple : `grille = construireGrille(4,6)`
`construireTriangle(n:int) -> graphe`	retourne la grille triangulaire d'ordre n, par exemple : `t5 = construireTriangle(5)`

B.4 Utilisation de la bibliothèque d'images

Il faut commencer par importer le module `bibimages.py` :

```
from bibimages import *
```

Les fonctions suivantes deviennent alors disponibles :

`ouvrirImage(nom:str) -> image`	Ouvre le fichier *nom* et retourne l'image qu'il contient, par exemple : `img = ouvrirImage("teapot.png")`
`nouvelleImage(large:int, haut:int) -> image`	Retourne une image de taille $large \times haut$, initialement noire, par exemple : `img = nouvelleImage(300, 200)`
`ecrireImage(img:image, nom:str)`	Sauvegarde l'image *img* dans le fichier *nom*, par exemple : `ecrireImage(img, "toto.png")`
`afficherImage(img:image)`	Affiche l'image *img*, par exemple : `afficherImage(img)`
`largeurImage(img:image)`	Retourne le nombre de colonnes de pixels contenues dans *img*, par exemple : `l = largeurImage(img)`
`hauteurImage(img:image)`	Retourne le nombre de lignes de pixels contenues dans *img*, par exemple : `h = hauteurImage(img)`
`colorierPixel(img:image, x:int, y:int, (r,g,b))`	Peint le pixel (x, y) dans l'image *img* de la couleur (r, g, b), par exemple : `colorierPixel(img, 10, 10, (255, 0, 0))`
`couleurPixel(img:image, x:int, y:int)`	Retourne la couleur du pixel (x, y) dans l'image *img*, par exemple : `(r,g,b) = couleurPixel(img, 10, 10)`

B.5 Rappel de la syntaxe

Les caractères "_____" représentent l'indentation obligatoire.

Affectation : `variable = expression`

Opérateurs mathématiques : opérateurs usuels `+,-,*,/`, division entière `//`, reste de la division entière `%` .

Opérateurs booléens : comparaison `<,>,<=,>=`, égalité `==,!=`, combinaison `and,or,not`

```
if (condition):
    instructions exécutées quand
    condition est vraie
```

Exemple :

```
if age <= age_reduction:
    prix = prix / 2
```

```
if (condition):
    instructions exécutées quand
    condition est vraie
else:
    instructions exécutées quand
    condition est fausse
```

Exemple :

```
if age <= age_reduction:
    prix = 5
else:
    prix = 10
```

```
if (condition1):
    instructions exécutées quand
    condition1 est vraie
elif (condition2):
    instructions exécutées quand
    condition2 est vraie
else:
    instructions exécutées quand
    condition1 et condition2 sont fausses
```

Exemple :

```
if age <= age_reduction1:
    prix = 5
elif age >= age_reduction2:
    prix = 7
else:
    prix = 10
```

Définition d'une fonction :

```
def fonction(parametres, avec, virgules):
    instructions exécutées quand
    fonction est appelée
    return valeur
```

Exemple :

```
def f(n,m):
    if n < m:
        return n
    return m
```

Appel d'une fonction :

```
fonction(arguments,a,fournir)
```

Exemple :

```
f(1,3)
```

```
for variable in une_liste:
    instructions exécutées avec variable
    contenant successivement les valeurs
    de une_liste
```

Exemple :

```
s = 0
for a in range(1,10):
    s = s + a
```

```
while (condition):
    instructions exécutées tant que
    condition est vraie
```

Exemple :

```
i = 1
while i < n:
    i = i * 2
```

La fonction `range` permet de générer des listes d'entiers utilisables par la primitive `for` :

`list(range(10))`	`[0,1,2,3,4,5,6,7,8,9]`
`list(range(3,7))`	`[3,4,5,6]`
`list(range(1,20,4))`	`[1,5,9,13,17]`

Note : Lorsque l'on a une fonction qui prend en paramètre une liste `L`, on n'utilise pas `range` puisque l'on a déjà une liste pour `for` :

```
def f(L):
  for x in L:
    ...
```

Inversement, si la fonction prend en paramètre un entier `n`, on a besoin d'utiliser `range` pour fabriquer une liste pour `for` :

```
def f(n):
  for i in range(n):
    ...
```