TD 1.1 : Premiers programmes

Bases de la Programmation Impérative – Chapitre 1 : bases
version du 20 octobre 2021

L'objectif de ce premier TD est d'être capable d'écrire **et de comprendre** les premiers programmes python que nous réaliserons en TP.

Attention : comme nous l'avons vu dans le premier cours, python est un langage de haut niveau. Par conséquent, bien que les programmes que nous allons voir dans ce premier TD soient petits et simples, ils cachent beaucoup de choses. L'objectif ici n'est pas de rentrer dans toutes ces choses cachées, mais de comprendre le nécessaire pour aborder les premiers TP.

Exercice 1 : portée des variables

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2
3  une_var_glob = 7
4
5  def f(un_param):
6      une_var_loc = 4
7      une_autre_var_loc = un_param + une_var_loc
8      print(une_autre_var_loc)
9
10 f(une_var_glob)
11 print(une_var_glob)
```

Question 1

Comme nous l'avons fait en cours (diapositive 21 ici), exécutons pas à pas le programme en modifiant sur papier un tableau contenant les variables accessibles à chaque étape du programme. Ce tableau indiquera pour chaque variable : son nom, son type, sa valeur et sa portée. Nous noterons également ce qui s'affiche sur la sortie standard.

Cliquez ici pour révéler la correction.

Après avoir exécuté la ligne 3, les variables sont les suivantes :

Nom	Type	Valeur	Portée
une_var_glob	int	7	Globale

Ensuite, l'interpréteur rencontre une **définition**. Celle-ci n'est pas exécutée pour le moment, elle le sera quand on **l'appellera**. L'interpréteur passe donc à la ligne 10.

L'appel à `f` fait à la ligne 10 dirige le flot de contrôle dans la fonction. Avant d'exécuter la première ligne de cette fonction, c'est à dire la ligne 6, une variable locale est créée pour chaque paramètre avec pour valeur, la valeur donnée en argument au moment de l'appel à la fonction. Avant d'exécuter la ligne 6, les variables sont donc les suivantes :

Nom	Type	Valeur	Portée
une_var_glob	int	7	Globale
un_param	int	7	Fonction f

Après avoir exécuté la ligne 6, les variables sont les suivantes :

Nom	Type	Valeur	Portée
une_var_glob	int	7	Globale
un_param	int	7	Fonction f

Nom	Type	Valeur	Portée
une_var_loc	int	4	Fonction f

Après avoir exécuté la ligne 7, les variables sont les suivantes :

Nom	Type	Valeur	Portée
une_var_glob	int	7	Globale
un_param	int	7	Fonction f
une_var_loc	int	4	Fonction f
une_autre_var_loc	int	11	Fonction f

L'exécution de la ligne 8 va donc afficher `11` sur la sortie standard.

Une fois que l'interpréteur est sorti de la fonction `f`, toutes les variables locales à la fonction disparaissent et on a donc :

Nom	Type	Valeur	Portée
une_var_glob	int	7	Globale

L'exécution de la ligne 11 va afficher `7` sur la sortie standard.

Exercice 2 : portée des variables avec même nom

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2
3  a = 7
4
5  def f(a):
6      b = 8
7      a = a + b
8      print(a)
9
10 a = 11
11 print(a)
12 f(a)
13 print(a)
```

Question 1

Même question que dans l'exercice précédent.

Cliquez ici pour révéler la correction.

Après avoir exécuté la ligne 3, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	7	Globale

Ensuite l'interpréteur voit la définition de `f` puis passe à la ligne 10. Après avoir exécuté la ligne 10, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	11	Globale

L'exécution de la ligne 11 va afficher `11`.

La fonction `f` est ensuite appelée avec `a` comme argument. Au début de l'exécution de la fonction `f`, c'est à dire avant que la ligne 6 soit exécutée, les variables sont donc les suivantes. On notera ici qu'il existe **deux variables `a` distinctes**.

Nom	Type	Valeur	Portée
a	int	11	Globale
a	int	11	Fonction f

Après avoir exécuté la ligne 6, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	11	Globale
a	int	11	Fonction f
b	int	8	Fonction f

Après avoir exécuté la ligne 7, les variables sont les suivantes. On notera ici que la variable `a` fait référence à la variable `a` locale à la fonction. En effet, lorsqu'une variable est utilisée, la sémantique de python indique que l'association entre le nom de la variable et une variable existante se fait en premier lieu en cherchant dans les variables locales. Si et seulement si la variable n'est pas trouvée dans les variables locales, alors elle sera cherchée dans les variables globales.

Nom	Type	Valeur	Portée
a	int	11	Globale
a	int	19	Fonction f
b	int	8	Fonction f

L'exécution de la ligne 8 va afficher `19`.

Une fois que l'interpréteur est sorti de la fonction `f`, toutes les variables locales à la fonction disparaissent et on a donc :

Nom	Type	Valeur	Portée
a	int	11	Globale

L'exécution de la ligne 13 va afficher `11` sur la sortie standard.

Exercice 3 : typage dynamique

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2
3  a = "1"
4  b = "3"
5  print(a + b)
6
7  a = 1
8  b = 3
9  print(a + b)
10
11 a = "1"
12 b = 3
13 print(a + b)
14
15 t1 = (1, 3)
16 print(t1[0] + t1[1])
17
18 t2 = (1, "3")
19 print(t2[0] + t2[1])
```

Question 1
Même question que dans les exercices précédents.

Cliquez ici pour révéler la correction.
Après avoir exécuté les lignes 3 et 4 les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	str	“1”	Globale
b	str	“3”	Globale

En python, l'opérateur `+` entre deux chaînes de caractères correspond à une concaténation des deux chaînes. Autrement dit, on “colle” les deux chaînes. Le résultat de cette opération est de type chaîne de caractères également. À la ligne 5, l'interpréteur doit commencer par évaluer `a + b`. Il va donc commencer par effectuer la concaténation qui va renvoyer la chaîne `"13"`. Cette chaîne est ensuite donnée en argument de l'appel à la fonction `print` qui va donc afficher `13` (la chaîne de caractères).

Après avoir exécuté les lignes 7 et 8 les variables sont les suivantes. Ici il faut noter une propriété fondamentale du langage python : **le type de la valeur associée à une variable peut changer au cours de l'exécution du programme**. On dit que python est un langage à **typage dynamique**, en opposition aux langages à **typage statique** (par exemple C que nous verrons au second semestre) dans lesquels les variables ont un type défini une fois pour toute dans le code que l'on écrit. Par conséquent, dans les langages à *typage statique* une variable ne peut pas changer de type.

Nous n'abordons pas dans ce premier TD la question des avantages et des inconvénients du typage dynamique et du typage statique. Ceux-ci devraient apparaître à nos yeux au cours de nos études ici à l'Ensimag. Néanmoins, afin de nous aider à garder le moral pendant nos TP de BPI, soulignons quand même un des avantages du typage dynamique :
“L'avantage du typage dynamique c'est qu'il diminue les frais de coiffeur puisqu'on s'arrache les cheveux à le déboguer”

Nom	Type	Valeur	Portée
a	int	1	Globale
b	int	3	Globale

En python, l'opérateur `+` entre deux entiers correspond à l'addition entière. À la ligne 9, l'interpréteur calcule donc d'abord `1 + 3` qui va donner l'entier `4` qui sera donné en argument de l'appel à la fonction `print` qui va donc afficher `4` (l'entier).

Après avoir exécuté les lignes 11 et 12 les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	str	“1”	Globale
b	int	3	Globale

En python, l'opérateur `+` ne peut pas être appelé entre une chaîne de caractères et un entier. L'exécution de la ligne 13 va donc générer une erreur `TypeError`. On dit que python est un langage **fortement typé** : il n'y a pas de conversion implicite d'un type vers un autre.

Si ce que l'on voulait faire ici était une concaténation entre la chaîne de caractères `a` et la représentation sous forme de chaîne de caractères de l'entier `b`, c'est à dire `"3"`dans l'exemple, alors il faudrait écrire la conversion explicitement dans notre code à l'aide de la fonction `str` : `print(a + str(b))`.

En continuant l'exécution du programme supposant que nous l'ayons corrigé en utilisant une conversion explicite, après avoir exécuté la ligne 15 les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	str	“1”	Globale
b	int	3	Globale
t	tuple	(1, 3)	Globale

La ligne 16 va donc afficher `4` car ici `+` correspond à l'addition entière.

Après avoir exécuté la ligne 18 les variables sont les suivantes. On notera ici qu'un tuple peut contenir des valeurs

de différent types.

Nom	Type	Valeur	Portée
a	str	“1”	Globale
b	int	3	Globale
t	tuple	(1, “3”)	Globale

Comme nous l'avons déjà vu, en python l'opérateur + ne peut pas être appelé entre un entier et une chaîne de caractères. L'exécution de la ligne 19 va donc générer une erreur `TypeError`.

Exercice 4 : mes propres types

On considère le programme suivant :

```
1  #!/usr/bin/env python3
2  """Illustration des namedtuple"""
3
4  import collections
5
6  bob = ("Robert", 23)
7  bill = ("William", 47)
8
9  # affiche le prénom de bob et l'age de bill (beurk)
10 print(bob[0])
11 print(bill[1])
12
13 bill[1] = 43
14
15 Personne = collections.namedtuple("Personne", "prenom, age")
16
17 bob_nt = Personne("Robert", 23)
18 bill_nt = Personne("William", 47)
19
20 # affiche le prénom de bob et l'age de bill (plus lisible !)
21 print(bob_nt.prenom)
22 print(bill_nt.age)
23
24 bill_nt.age = 43
```

Question 1
Même question que dans les exercices précédents.

Cliquez ici pour révéler la correction.
Après avoir exécuté les lignes 4, 6 et 7 les variables sont les suivantes :

Nom	Type	Valeur	Portée
bob	tuple	(“Robert”, 23)	Globale
bill	tuple	(“William”, 47)	Globale

L'exécution des lignes 10 et 11 va donc afficher `Robert` puis `47`. En utilisant des tuples, comme l'indique le commentaire, il est difficile de savoir ce que représentent `bob[0]` et `bill[1]` sans aller se référer à la création des tuples. Ici, cela ne pose pas trop de difficultés, mais dans un grand programme cela peut assez vite rendre le code très difficile à lire.

À la ligne 13, on veut changer la valeur du second attribut du tuple `bill` (pour rajeunir ce dernier). L'exécution de cette ligne va générer une erreur `TypeError` car les tuples python sont **immuables**, c'est à dire que leur valeur ne peut jamais être changée une fois que le tuple a été créé. Autrement dit, on ne pas changer les attributs ni ajouter/supprimer d'éléments à un tuple.

La motivation derrière l'immuabilité de certains types en python concerne l'implémentation du langage, c'est à dire l'implémentation de l'interpréteur lui même. Concrètement, le fait que certains types soient immuables permet de

nombreuses optimisations dans l'implémentation de l'interpréteur.

On suppose maintenant avoir commenté la ligne 13 qui cause une erreur, et on continue donc à la ligne 15.

L'exécution de la ligne 15 va utiliser la fonction `namedtuple` du module `collections` qui a été importé ligne 4 pour créer un nouveau type appelé `Personne` composé de deux attributs respectivement nommés `prenom` et `age`. Ici, il faut se demander quel est le type de la variable `Personne` ainsi créée ? Comme l'indique le fait que nous utilisions une majuscule au début du nom de cette variable (ce n'est pas obligatoire mais une convention pour faciliter la lecture du code), la variable `Personne` est de type "type défini par namedtuple". Cette variable va donc nous permettre par la suite de créer des personnes.

Les variables sont donc les suivantes après l'exécution de la ligne 15. Pour ne pas rentrer dans les détails de python car cela ne nous intéresse pas pour le moment, on notera `???` la valeur de la variable `Personne`.

Nom	Type	Valeur	Portée
bob	tuple	("Robert", 23)	Globale
bill	tuple	("William", 47)	Globale
Personne	type défini par namedtuple	???	Globale

Les lignes 17 et 18 vont créer deux personnes. Une fois ces deux lignes exécutées les variables sont donc les suivantes :

Nom	Type	Valeur	Portée
bob	tuple	("Robert", 23)	Globale
bill	tuple	("William", 47)	Globale
Personne	type défini par namedtuple	???	Globale
bob_nt	Personne	("Robert", 23)	Globale
bill_nt	Personne	("William", 47)	Globale

L'exécution des lignes 21 et 22 affiche dont `Robert` et 47. Le code est bien plus facile à lire que le code utilisant des tuples car nous voyons ici explicitement ce qui est affiché.

Comme leur nom l'indique, les namedtuples sont des tuples, et donc sont également immuables. L'exécution de la ligne 24 va donc générer une erreur `TypeError`.

Exercice 5 : et dans un autre langage, ça donne quoi ?

On considère le programme suivant :

```ruby
#!/usr/bin/env ruby

a = 2010

def f(p)
  puts "bienvenue "
  return p + 11
end

puts "à l'Ensimag "

a = f(a)
puts "en " + String(a)
puts "(signé Yoda)"
```

Question 1

Même question que dans les exercices précédents.

Cliquez ici pour révéler la correction.

Dans cet exercice on va voir que les raisonnements effectués pour du code python sont valide également pour le langage ruby.

Après avoir exécuté la ligne 3, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	2010	Globale

On passe ensuite à la définition de la fonction puis à la ligne 10. L'exécution de cette ligne va afficher `à l'ensimag` sur la sortie standard (`puts`).

Ensuite, l'exécution de la ligne 12 va commencer par l'appel à la fonction `f` avec `a` comme argument. En effet, pour pouvoir affecter la nouvelle valeur à `a` il faut connaître le résultat de `f(a)`.

L'exécution se poursuit donc dans la fonction `f` et juste avant d'exécuter la ligne 6, les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	2010	Globale
p	int	2010	Fonction f

L'exécution de la ligne 6 va afficher `bienvenue` sur la sortie standard.

L'exécution de la ligne 7 va d'abord calculer la valeur de `p + 11` et ensuite renvoyer cette valeur, c'est à dire `2021` , du côté de l'appelant de la fonction. Autrement dit, le `return` renvoie l'exécution à la ligne 12 avec la valeur `2021`. L'affectation de la ligne 12 peut donc maintenant avoir lieu, et une fois celle-ci effectuée les variables sont les suivantes :

Nom	Type	Valeur	Portée
a	int	2021	Globale

L'exécution de la ligne 13 va d'abord effectuer la concaténation puis afficher le résultat, à savoir `en 2021` sur la sortie standard. On notera ici que ruby est également un langage fortement typé car la conversion explicite, via `String(a)`, de `a` en chaîne de caractères est nécessaire pour pouvoir faire la concaténation.

Enfin, l'exécution de la ligne 14 va afficher `(signéYoda)` sur la sortie standard.

Exercice 6 : et encore dans un autre langage, ça donne quoi ? (pour aller plus loin)

Question 1

Réécrire le programme de l'exercice précédent dans le langage impératif de votre choix (autre que Python et Ruby, en C par exemple).

Question 2

Identifier les différences fondamentales avec Python.