Lycée Louis-Le-Grand, Paris 2014/2015
MPSI 4 – Informatique pour tous
D. Cazor, A. Troesch

TP n° 1 : À la découverte de Python (exercices 1 à 7)

Correction de l'exercice 1 – Utilisation de la console comme calculatrice

1. La console peut s'utiliser comme une calculatrice. Il suffit de taper l'opération après l'invite de commande :

```
>>> 54 + 23
77
>>> 24 - 7
17
>>> 123 *87
10701
>>> 34 / 5
6.8
```

2. Quelques essais montrent que :
   - `//` donne le quotient de la division euclidienne
   - `%` donne le reste de la division euclidienne
   - `**` est l'exponentiation (puissance)

```
>>> 12 // 5
2
>>> 36 // 7
5
>>> 12 % 5
2
>>> -24 % 5
1
>>> -5 // 5
-1
>>> -24 // 5
-5
>>> 2 ** 6
64
>>> 3 ** 4
81
```

3. On ne peut pas se contenter de juxtaposer des calculs sur une même ligne (erreur de syntaxe). On peut les séparer par une virgule, ce qui fournit un résultat sous forme de couple (de la classe `tuple`, représentant les n-uplets), ou par un point-virgule. Dans ce cas, les deux résultats sont donnés successivement.

```
>>> 12 * 3 24 * 4
  File "<stdin>", line 1
    12 * 3 24 * 4
           ^
SyntaxError: invalid syntax
>>> 12 * 3 , 24 * 4
(36, 96)
>>> 12 * 3 ; 24 * 4
36
96
```

Il serait faux de penser que ces deux façons de faire sont équivalentes. Il y a une différence fondamentale entre les deux :

- Séparées par une virgule, les deux expressions sont calculées simultanément. En fait, l'expression entrée est interprétée comme le `tuple (12*3,24*4)` (avec parenthèses implicites)
- Séparées par un point-virgule, il s'agit de la succession de deux instructions.

La première syntaxe n'est possible qu'avec des expressions (calculs), alors que la seconde est valide pour toute succession d'instructions. Voyons sur un exemple la subtilité de la situation :

```
>>> x = 3
>>> x = x + 2, x + 3
>>> x
(5, 6)
>>> x = 3
>>> x = x + 2; x + 3
8
>>> x
5
>>>
```

La première série montre que l'affectation `x = x + 2, x + 3` est comprise comme une affectation de tuple, x prenant la nouvelle valeur donnée par les deux calculs (simultanés) de $x + 2$ et $x + 3$ (avec l'ancienne valeur de x). Lors de la deuxième série, on effectue d'abord l'affectation `x = x + 2`, remplaçant l'ancienne valeur 3 de x par 5, puis on effectue l'opération `x + 3`, qui renvoie la valeur 5. Cette dernière opération s'effectuant sans affectation, elle ne modifie pas la valeur de x, toujours égale à 5 en fin de calcul.

Correction de l'exercice 2 – À la découverte des Variables

Une variable est la donnée d'une place en mémoire, destinée à stocker une valeur, et d'un nom permettant l'accès à cette place en mémoire. La valeur d'une variable peut changer au cours du temps. L'action de donner une valeur à une variable s'appelle « l'affectation », et s'effectue avec le signe =.

1. L'opération d'affectation n'est pas symétrique : la variable se trouve à gauche, la valeur à droite :

```
>>> x
3
>>> 3 = x
  File "<stdin>", line 1
SyntaxError: can't␣assign␣to␣literal
```

2. Une variable est un contenu. Définir une variable y par une expression en fonction d'une autre variable x se fait par le calcul de l'expression en x, avec le contenu de la variable x au moment de l'affectation. En aucun cas, une modification ultérieure de la valeur de x ne peut avoir d'effet sur y : le contenu de la variable y est une valeur et non une expression en x.

```
>>> y = 7 * x
>>> y
21
>>> x = 1
>>> y
21
```

3. • `help(id)` amène la page d'aide suivante :

```
Help on built-in function id in module builtins:

id(...)
    id(object) -> integer

    Return the identity of an object. This is guaranteed to be unique among
    simultaneously existing objects. (Hint: it's␣the␣object's memory address.)
(END)
```

On y apprend que `id()` renvoie l'identifiant d'un objet (correspondant d'une façon ou d'une autreà l'adresse mémoire assignée à la variable)

- De même help(type) nous apprend sans surprise que la fonction `type()` renvoie le type d'un objet. On y apprend aussi qu'on peut créer un nouveau type, mais cela ne nous concernera pas vraiment.

```
>>> id(x)
211273705440
>>> type(x)
<class 'int'>
```

4. Modifier la valeur de x modifie son identifiant, car une telle modification est en fait vue comme une nouvelle affectation (on redéfinit une nouvelle variable x). Toute opération sur une variable passant par une nouvelle affectation modifie son identifiant. On peut cependant modifier des variables sans affectation, en utilisant des méthodes associées à certains types. Ces méthodes ne modifient pas alors l'identifiant. Nous en verrons des exemples lorsque nous manipulerons des listes ou des chaînes de caractères.

```
>>> x += 1
>>> id(x)
211273705472
>>> type(x)
<class 'int'>
>>> x += 0.5
>>> id(x)
140638118502688
>>> type(x)
<class 'float'>
```

5. L'instruction `x += y` remplace le contenu de la variable x par la valeur de $x + y$. Remarquez que cela modifie comme précédemment l'identifiant de x (il s'agit d'une nouvelle affectation). Cette instruction est donc équivalente à $x = x + y$

```
>>> x = 3
>>> id(x)
211273705504
>>> x += 2
>>> id(x)
211273705568
>>> x
5
```

   De même pour les autres opérations :

```
>>> x = 3
>>> x -= 1
>>> x
2
>>> x *= 6
>>> x
12
>>> x /= 4
>>> x
3.0
```

6. On pourrait penser dans un premier temps faire :

```
>>> x = 2
>>> y = 5
>>> x = y
>>> y = x
>>> x;y
5
5
```

C'est un échec : on n'a pas du tout fait l'échange des contenus de x et y ; les deux variables ont pris la valeur de y. En inversant les deux affectations, elles prennent toutes deux la valeur de x. En y réfléchisant un peu plus, on se rend facilement compte que c'est assez logique, les affectations se faisant successivement !

Une méthode classique consiste à introduire une nouvelle variable, stockant provisoirement la valeur d'une des deux variables.

```
>>> x = 2
>>> y = 5
>>> z = x
>>> x = y
>>> y = z
>>> x ; y
5
2
```

On peut aussi chercher à effectuer les deux affectations simultanément. On pourrait penser à :

```
>>> x = y ; y = x
```

mais cela pose le même problème que précédemment, les deux instructions disposées sur la même ligne et séparées par un point-virgule étant effectuées successivement. On peut alors penser à :

```
x, y = y, x
```

et ça marche ! En fait, cette instruction est interprétée comme une unique affectation portant sur le couple (x, y).

Correction de l'exercice 3 – Premiers affichages

On répond en ligne de commande.

1. L'affectation d'une valeur à une variable se fait avec l'égalité simple =, en indiquant à gauche du signe le nom de la variable, et à droite la valeur (numérique, ou tout autre type disponible) :

   ```
   >>> Bonjour = 0
   ```

   Si on veut visualiser la valeur d'une variable, on tape son nom. L'ordinateur nous renvoie sa valeur :

   ```
   >>> Bonjour
   0
   ```

2. La fonction `print` convertit en chaînes de caractère les valeurs de ses différents paramètre afin faire un affichage à l'écran. L'instruction `print(Bonjour)` renvoie :

   ```
   >>> print(Bonjour)
   0
   ```

   En effet, ici, `Bonjour` désigne non pas la chaîne de caractère Bonjour (qui permettrait d'afficher le texte « Bonjour »), mais la variable définie précédemment. Une succession de caractère ne désigne une chaîne de caractères que si elle est entourée des délimiteurs convenables. Sinon, elle désigne le nom d'un objet ; ainsi, si la variable `Bonjour` n'avait pas été définie avant, nous aurions eu une erreur :

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

   Cette erreur possède un nom : `NameError` (le fait de connaître, et de pouvoir distinguer les noms d'erreur peut avoir un intérêt pour la gestion des exceptions, par la structure `try... except...`)

   Les délimiteurs de chaînes de caractère sont au nombre de 2 :
   - les apostrophes `'texte'`
   - les guillemets `"texte"`

L'intérêt d'avoir plusieurs délimiteurs est de pouvoir plus facilement utiliser ces délimiteurs comme caractère du texte. Ainsi, `'Aujourd'hui'` n'est pas une syntaxe possible ; en revanche, on peut définir la chaîne `"Aujourd'hui"`. De même, on ne peut pas définir `"Il dit : "Bonjour"."`, mais `'Il dit : "Bonjour".'` convient.

Un troisième délimiteur permet de gérer par exemple le cas du texte :

`Il dit: "Il fait beau aujourd'hui".`

Il s'agit du triple-guillemet `"""`. Ces triples-guillemets (définissant les raw-string) acceptent aussi les retours à la ligne par la touche « Entrée » au lieu du caractère spécial `\n`.

3. On obtient le type d'un objet par la fonction `type` :

```
>>> type(Bonjour)
<class 'int'>
```

Ainsi, la variable « Bonjour » est de type `'int'`, à savoir de type entier. La reconnaissance de type de la variable s'est fait automatiquement lors de l'affectation. Le type entier n'est en `Python3`, pas borné en taille, et comprend donc également le type entier long anciennement dans `Python2`. Pour la chaîne de caractère, on obtient

```
>>> type('Bonjour')
<class 'str'>
```

Ceci nous indique que le type d'une chaîne de caractères est `'str'`, pour `string`.

4. Nous avons déjà répondu plus haut :

```
>>> print("Aujourd'hui")
Aujourd'hui
```

```
>>> print('Il␣dit:␣"Bonjour"')
Il dit: "Bonjour"
>>> print("\"Aujourd'hui\"")
"Aujourd'hui"
```

Dans le dernier affichage, on distingue le caractère " du délimiteur de chaînes de caractères, en le faisant précéder du *backslash* \. On aurait pu s'en sortir avec une raw string, mais cela poserait en fait un problème d'espace, le premier et le dernier caractère d'une raw string ne pouvant pas être un guillemet normal.

5. Le caractère spécial `\n` est un retour à la ligne :

```
>>> print("Bonjour.\nAu␣revoir.")
Bonjour.
Au revoir.
```

6. Pour consulter l'aide associée à une fonction, on utilise la fonction `help`. L'instrution `help(print)`, renvoie par exemple :

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep='␣', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

On y apprend les différents paramètres utilisables avec `print`, ainsi que la syntaxe adaptée, et les valeurs par défaut (espace pour `sep`, retour à la ligne pour `end`).

```
>>> x = 1
>>> y = 2
>>> z = 3
>>> print(x,y,z,sep = ' ; ')
1 ; 2 ; 3
>>> print(x,y,z,sep = ' et ')
1 et 2 et 3
>>> print(x,y,z, end = '.')
1 2 3.>>> print(x,y,z, end = '.\n')
1 2 3.
>>> print(x,y,z, sep = ' ;\n', end = ' ;\nCQFD\n')
1 ;
2 ;
3 ;
CQFD
```

Remarquez que sans retour à la ligne final, l'invite de commande se met à la suite du texte affiché, ce qui est un peu désagrable.

Correction de l'exercice 4 – Lecture de données rentrées par l'utilisateur

1. La fonction est la fonction permettant une interface d'entrée de données par l'utilisateur. Cette instuction prend en paramètre un texte. Lors de son exécution, le texte s'affiche, puis l'ordinateur attend que l'utilisateur entre un texte (validée par la touche « Entrée »). Le format de cette entrée est une chaîne de caractère : classe `str` (*string*).

   Cette chaîne de caractères peut être utilisée comme n'importe quelle chaîne de caractères, par exemple stockée dans une variable, en vue d'une utilisation ultérieure. Le contenu de cette variable peut être affiché. On peut aussi afficher la chaîne de caractère avec `print`.

   ```
   >>> texte = input('Entrez un texte : ')
   Entrez un texte : Bonjour
   >>> texte
   'Bonjour'
   >>> print(texte)
   Bonjour
   >>>
   ```

   Dans cet exemple, le texte entré par l'utilisateur est « Bonjour ».

2. On pourrait penser à faire comme plus haut :

   ```
   >>> x = input('Entrez une valeur : ')
   Entrez une valeur : 5
   >>> x+3
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: Can't convert 'int' object to str implicitly
   >>> x
   '5'
   >>> type(x)
   <class 'str'>
   ```

   Le message d'erreur lors du calcul de $x+3$ nous indique qu'il y a une erreur de type (`TypeError`). Plus précisément, l'ordinateur ne peut pas convertir implicitement (c'est-à-dire sans l'ordre explicite de l'utilisateur ou du programme) une chaîne de caractères en une valeur numérique, ou réciproquement (l'opération + étant définie aussi sur les chaînes de caractères, elle correspond dans ce cas à la concaténation des chaînes). Cette erreur suggère que la variable x, contrairement à ce que l'on souhaite, mais conformément à ce qu'on a dit dans la question 1, n'est pas de type numérique : il s'agit d'une chaîne de caractères. Cela se confirme par la présence des apostrophes autour de la valeur donnée à x lorsqu'on demande l'affichage du contenu de la variable x. La dernière instruction nous confirme que x est bien de type `str`.

3. La fonction `eval` permet d'évaluer une chaîne de caractères, dans le cas où cette chaîne représente une valeur numérique (ou un objet de certains autres types, comme les listes ou les ensembles) :

```
>>> eval(x)
5
```

Souvent, lorsqu'on attend de l'utilisateur une valeur numérique, on couple la fonction `eval` et la fonction `input` dans une même instruction :

```
>>> x = eval(input('Entrez␣une␣valeur␣:␣'))
Entrez une valeur : 5
>>> x
5
>>> type(x)
<class 'int'>
>>> x + 3
8
```

Cette fois, la variable x est bien de type numérique (entier en l'occurrence).

Correction de l'exercice 5 – Comment importer un module

1. Essayons :

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

Eh ben flûte ! C'est quoi ce langage qui ne connaît même pas les fonctions mathématiques les plus élémentaires ? ! En fait, très peu d'outils mathématiques sont disponibles de façon directe en Python. Ces outils sont définis dans des modules spécialisés qu'il faut importer. Cela permet une plus grande légèreté du logiciel (on n'importe que les fonctions qui nous sont utiles, inutile de s'encombrer de la définition de celles qui ne nous servent pas). Cela permet, sans alourdir une utilisation standard, de définir dans certains modules spécialisés des outils très pointus et très performants. Nous aurons par exemple l'occasion d'utiliser en cours d'année les modules `numpy` et `scipy`, définissant un certain nombre d'outils de calcul numérique et scientifique. Nous utiliserons également le module `matplotlib`, et plus précisément le sous-module `matplotlib.pyplot` pour le tracé de graphes.

2. Importons le module math et réessayons :

```
>>> import math
>>> sin(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(1)
0.8414709848078965
```

Il ne connaît toujours pas la fonction sin si on lui demande directement. En revanche, en précisant où la chercher, ça marche. On peut obtenir la liste des fonctions définies dans un module grâce à l'aide : `help(math)` nous renvoie la liste (et la description) des fonctions définies dans le module `math`. On peut alors calculer l'expression demandée :

```
>>> math.sqrt(1 + math.log(math.pi)**2)
1.5200021419579581
```

On peut importer un module en lui donnant un autre nom (alias) :

```
>>> import math as ma
>>> ma.cos(ma.pi / 3)
0.5000000000000001
```

C'est surtout utile pour les modules dont le nom est long, ou pour les sous-modules (nécessitant la mention du nom du module et du sous-module), comme textttmatplotlib.pylot, qu'on peut importer par exemple uniquement sous le nom `plt`, ce qui est plus pratique.

Un usage courant veut par exemple que le module `numpy` soit souvent importé sous le nom `np`.

3. On peut aussi importer juste une fonction du module. On n'aura plus besoin alors de spécifier où la trouver (mention du module en préfixe) :

```
>>> from math import sin
>>> sin(1) + sin(2) + sin(3) + sin(4) + sin(5) + sin(6)
-0.10325384847654717
>>> from math import sqrt
>>> sqrt(1+sqrt(1+sqrt(1+sqrt(1+sqrt(1+sqrt(2))))))
1.6174427985273905
```

4. Enfin, si on fait appel à un grand nombre de fonctions du module, on peut les importer toutes, en écrivant `from math import *`

Quelques essais suggèrent que $\Gamma(n) = (n-1)!$ pour tout $n \in \mathbb{N}^*$. Cependant Γ renvoie un résultat de type `float`, contrairement à la factorielle. De plus, Γ est défini sur les réels positifs non nécessairement entiers, contrairement à la factorielle (mais pas sur les réels négatifs).

```
>>> factorial(7)
5040
>>> gamma(7)
720.0
>>> gamma(8)
5040.0
>>> gamma(6.7)
413.4075167652709
>>> gamma(-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>> factorial(5.7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: factorial() only accepts integral values
```

En fait, comme vous le verrez l'année prochaine, la fonction Γ est définie par une intégrale impropre :

$$\forall x > 0, \quad \Gamma(x) = \int_0^{+\infty} t^{x-1}\mathrm{e}^{-t}\,\mathrm{d}t = \lim_{a\to 0}\lim_{A\to+\infty}\int_a^A t^{x-1}\mathrm{e}^{-t}\,\mathrm{d}t.$$

Des techniques de comparaison que vous étudierez l'an prochain permettent de montrer que cette limite est bien définie. En faisant d'abord une intégration par parties sur l'intégrale entre a et A, puis en faisant tendre A vers ∞ et a vers 0, on peut montrer que pour tout $x > 0$,

$$\Gamma(x+1) = x\Gamma(x).$$

De plus, il est facile de calculer $\Gamma(1) = 1$. Une récurrence immédiate amène alors, pour tout $n \in \mathbb{N}^*$, la relation conjecturée $\Gamma(n) = (n-1)!$.

Correction de l'exercice 6 – Premières fonctions

1.

```
>>> def f(x,n):
...     print('1 + {}^{} = {:g}'.format(x, n, 1 + x ** n))
...
>>> f(4,3)
1 + 4^3 = 65
>>> f(12,9)
1 + 12^9 = 5.15978e+09
```

2. Essayons de calculer $f(3,3)+f(4,4)$:

```
>>> f(3,3) + f(4,4)
1 + 3^3 = 28
1 + 4^4 = 257
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
>>> type(f)
<class 'function'>
>>> type(f(3,3))
1 + 3^3 = 28
<class 'NoneType'>
```

Cela nous renvoie une erreur de type. En étudiant de plus près, on se rend compte que si f possède bien un type (`function`), ce n'est pas le cas de $f(3,3)$. En fait, il n'existe pas d'objet $f(3,3)$. En effet, l'évaluation de f en $(3,3)$ ne fournit pas un objet, mais effectue une certaine action (l'affichage)

Pour définir une valeur de sortie de la fonction f, il faut utiliser l'instruction `return`. De cette manière, on définit l'objet $f(x,y)$, sur lequel on peut employer toute opération ou toute méthode disponible pour le type correspondant.

3. Rajoutons la ligne correspondant :

```
>>> def f(x,n):
...     print('1 + {}^{} = {:g}'.format(x, n, 1 + x ** n))
...     return 1 + x ** n
...
>>> f(3,3)
1 + 3^3 = 28
28
>>> f(3,3)+f(4,4)
1 + 3^3 = 28
1 + 4^4 = 257
285
```

Remarquez qu'on a toujours l'affichage demandé pour chacun des deux couples, avant le résultat final du calcul. Observez ce qui se passe si on échange les deux lignes :

```
>>> def f(x,n):
...     return 1 + x ** n
...     print('1 + {}^{} = {:g}'.format(x, n, 1 + x ** n))
...
>>> f(3,3)
28
>>> f(3,3)+f(4,4)
285
```

La seconde ligne (affichage) n'est pas exécutée ! L'instruction `return` provoque la sortie de la fonction. Tout ce qui suit n'est pas exécuté (on parle de code mort)

Cela dit, il n'est pas très pertinent de faire des affichages intermédiaires pour une fonction numérique. Imaginez ce que cela ferait si on calculait à l'aide d'une boucle la somme des $f(k,2)$ pour k allant de 1 à 1000 : on aurait l'affichage de 1000 lignes indésirables avant d'obtenie le résultat qui nous intéresse !

De ce fait, on mélange assez rarement des fonctions d'affichage et des fonctions de calcul. Une fonction de calcul se contente de calculer, sans afficher le résultat. On épure alors notre fonction :

```
>>> def f(x,n):
...     return 1 + x ** n
...
>>> f(3,3)
28
```

```
>>> f(3,3) + f(4,4)
285
```

Remarquez au passage l'importance de l'indentation :

```
>>> def f(x,n):
... return 1 + x ** n
  File "<stdin>", line 2
    return 1 + x ** n
        ^
IndentationError: expected an indented block
```

Attention, l'indentation doit être stricte s'il y a plusieurs lignes. Voici ce qu'il se passe si les deux lignes ne sont pas bien alignées :

```
>>> def f(x,n):
...     x = x ** n
...    return 1 + x
  File "<stdin>", line 3
    return 1 + x
               ^
IndentationError: unindent does not match any outer indentation level
```

L'importance de l'alignement en Python, vient du fait que c'est l'indentation qui délimite les différents blocs de la structure d'un programme en Python, contrairement à la plupart des autres langages de programmation qui utilisent souvent plutôt des délimiteurs de blocs explicites du type `begin` et `end`.

Correction de l'exercice 7 – On écrit un programme avec un éditeur de texte adapté (emacs, vi...), ou dans un environnement de programmation (Pyzo, Eclipse...) On peut dans le premier cas lancer l'exécution du programme en ligne de commande (par exemple `python3 essai.py` pour lancer l'exécution du programme `essai.py`), ou depuis l'environnement de programmation (depuis l'onglet `Run`). Il est recommandé, comme pour tout document informatique, de préciser quel est le langage utilisé en commentaire dans la première ligne. En Python, les commentaires se font en faisant précéder le texte du symbole ♯. Nous omettrons systématiquement cette étape. Depuis la version 3 de Python, il n'est plus nécessaire de spécifier l'encodage des caractères.

1. Nous avons là un programme très simple, d'une seule ligne :

   ```
   print('Bonjour')
   ```

   L'exécution de ce programme (que j'ai sauvegardé sous le nom `py011-1.py`) depuis un terminal donne :

   ```
   $ python3 py011-1.py
   Bonjour
   ```

2. Prenez l'habitude de commenter vos programmes. Cela peut faciliter une reprise ultérieure de votre code (par vous ou par quelqu'un d'autre). Un programme non commenté peut très vite devenir assez incompréhensible. Évidemment, ici, j'exagère un peu dans l'autre sens.

   ```
   # Définition de la fonction f
   def f(x):
       return 1 / (1 + x ** 2)

   # Demande d'une valeur x à l'utilisateur
   x = eval(input('Entrez␣une␣valeur␣de␣x␣:␣'))
   # Ne pas oublier de convertir cette entrée en valeur numérique grâce à eval

   #Affichage du résultat
   print('f({:g})␣=␣{:g}'.format(x, f(x)))
   ```

   À nouveau, une exécution dans un terminal donne :

   ```
   $ python3 py011-2.py
   Entrez une valeur de x : 3
   f(3) = 0.1
   ```