



UE121 - Introduction à la programmation

HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG

Technologie de l'informatique - bloc 1

Sécurité des systèmes - bloc 1

Atelier 1 : Variables, opérateurs, entrées et sorties

Objectifs

- Se familiariser avec l'environnement de travail
- Découvrir la structure d'un script en Python
- Premiers scripts Python
- Découverte des éléments de base : variable, type, littéral, instruction et expression, entrées et sorties, opérateurs

INTRODUCTION	2
A. VARIABLE.....	2
B. ENTRÉES ET SORTIES.....	11
C. INSTRUCTIONS ET EXPRESSIONS	14
D. OPÉRATEURS	16
E. OPÉRATEURS PARTICULIERS	22

Introduction

Cet atelier est prévu pour deux séances d'atelier.

Dans ce document, plusieurs conventions sont utilisées :

- les mots gras désignent des termes de vocabulaire liés à l'**informatique en général**.
- les mots soulignés et gras désignent des termes de vocabulaire directement liés aux cours de **programmation**.
- le logo 🧠 signifie que vous avez quelque chose à réaliser.
- le logo 📋 est associé aux cadres présentant certaines conventions.
- le logo 🧹 est associé aux cadres présentant les éléments liés à la propreté/lisibilité du code (*clean code*).

Dans cet atelier, vous allez découvrir les notions de base de Python...

La première notion qu'il est nécessaire de maîtriser est celle de variable. Une variable en informatique n'a pas la même signification que la variable utilisée en mathématique. De plus, en Python, elles ont certaines particularités qu'il est utile de préciser. Dans cette partie seront abordées les notions de types, d'adresse... et les bons usages quant au choix de leur nom.

Les instructions d'entrée/sortie permettent de dialoguer avec l'utilisateur. Les instructions permettant d'afficher quelque chose à l'écran ou de l'envoyer à l'imprimante ou tout autre périphérique de sortie, s'appellent « instructions de sortie ». Celles permettant de récupérer une information provenant d'un clavier, de la souris ou d'un autre périphérique d'entrée, s'appellent « instructions d'entrée ». Vous aurez l'occasion de mieux en comprendre l'utilisation dans ce même atelier.

Ensuite, les notions d'instruction et d'expression sont vues. La nuance entre ces deux concepts est importante à comprendre !

Enfin, ce sont les différents opérateurs qui sont vus ainsi que les notions qui les accompagnent, telle que la réaffectation, le caractère muable ou non des variables, la priorité des opérateurs...

A. Variable

Une **variable** associe un **nom** à une zone mémoire, localisée par une **adresse**, où l'on peut mémoriser une **valeur**. La Figure 1 en est une représentation simplifiée.

Quel que soit le langage, elle est caractérisée par

- son **nom**, il est spécifié lors de sa première affectation,
- sa **valeur** dont la représentation mémoire dépend du type de la variable,
- son **adresse**, l'endroit où cette valeur est stockée en mémoire, définie par le système,
- son **type**, permet à l'ordinateur de réserver la zone mémoire adéquate (abordé plus loin),
- sa **portée**, la portion de code dans laquelle la variable est accessible.

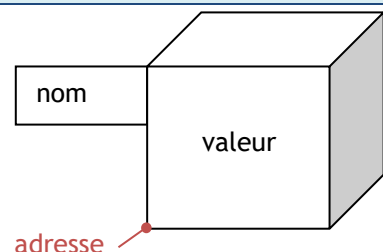


Figure 1 - Variable

L'instruction ci-dessous est une affectation ou assignation.

```
nb_torches_sac = 20
```

L'**affectation** a pour effet de réserver un espace mémoire et d'assigner une valeur à la variable, c'est-à-dire placer la valeur en question dans la zone mémoire.

Dans cet exemple, elle a pour effet de réserver un espace mémoire, de l'associer au nom de variable `nb_torches_sac` et d'y mettre la valeur `20`.

Déclaration


Dans certains langages, tels que le C et le Java, les variables doivent être déclarées avant de se voir affecter une valeur. La déclaration est l'instruction qui permet de préciser le type de la variable, c'est-à-dire le type des valeurs qui pourront lui être affectées par la suite. C'est à l'exécution de cette instruction qu'un espace mémoire est alloué et associé au nom de la variable.

En Python, une variable est déclarée lors de sa première affectation.

L'affectation est donc une instruction qui a pour effet de réaliser plusieurs opérations dans la mémoire de l'ordinateur. En effet, elle permet de...

- créer, s'il n'existe pas dans la portée, un nom de variable,
- mémoriser ce nom dans un espace de la mémoire prévu à cet effet,
- créer et mémoriser la valeur dans un autre espace mémoire prévu à cet effet,
- établir un lien entre le nom de la variable et l'emplacement mémoire de la valeur correspondante,
- attribuer un type à la variable en fonction de la valeur qui lui est affectée.

Une valeur peut prendre différentes formes : un entier, un réel, un caractère, une chaîne de caractères, une adresse, voire même une expression (qui doit être évaluée avant d'être utilisée) ...

 Tapez les quelques instructions suivantes dans le Shell et vérifiez que les valeurs des variables sont bien mémorisées en tapant ensuite le nom de la variable.

```
bloc = 1           # entier
message = "Hello world !" # chaîne de caractères
pi = 3.14159       # réel
majeur = True      # booléen
```

Pour la première instruction, vous devriez avoir le même affichage qu'à la Figure 2.

```
>>> bloc = 1
>>> bloc
1
>>> |
```

Figure 2 - Valeur de variable

Commentaires

Le caractère # permet de mettre des commentaires dans le code.

Dès que l'interpréteur le rencontre, il sait que les caractères qui suivent ne font pas partie du script, mais sont juste là afin d'ajouter des informations SI NÉCESSAIRE !

Le nom de la variable

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Ils font partie de ce qu'on appelle des identificateurs. Un identificateur ou identifiant est un mot qui désigne un élément du programme : variable, constante, procédure, fonction, type, etc.

En Python, les noms de variables doivent cependant obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres ($a \rightarrow z$, $A \rightarrow Z$) et de chiffres ($0 \rightarrow 9$), qui doit toujours commencer par une lettre.
- La casse est significative (les caractères majuscules et minuscules sont distingués) : "Guido", "guido", et "GUIDO" sont des variables différentes.
- Les 29 « mots réservés » du langage sont interdits ! (ex : "print").

Les règles de nommage ont été assouplies depuis Python 2, car il devient possible d'utiliser n'importe quel caractère Unicode¹ (si ce ne sont pas des opérateurs) dans les noms.

Il faut également noter que les variables dont le nom commence par le caractère _ ont une signification particulière :

- les noms commençant par un _ ne sont pas exportés lorsqu'ils se trouvent dans un module ;
- les noms commençant par deux _ et finissant par deux _ sont réservés par le langage lui-même, notamment pour la programmation orientée objet.

De plus, il est recommandé de commencer les noms de variables avec une minuscule, car l'usage veut qu'on réserve plutôt la majuscule aux noms de classes.

Si le nom d'une variable doit comporter plusieurs mots, la norme Python recommande le "snake_case", c'est à dire en séparant les mots par le caractère "_".

Exemple : `adresse_ip`.

Sinon, d'autres notations existent ou peuvent être imposées en fonction d'un projet, des habitudes d'une entreprise, etc.

Bien choisir les noms de variables



Efforcez-vous de bien choisir les noms de vos variables : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir.

¹ Plus d'informations sur <https://home.unicode.org/basic-info/faq/>

Par exemple, des noms de variables tels que `longitude` et `latitude` conviennent mieux que `x` et `y`, car un bon programmeur doit veiller à ce que ses scripts soient faciles à lire. Il convient aussi d'éviter autant que possible l'énumération de variables (`toto1`, `toto2`, `toto3...`), cela rend le programme parfaitement incompréhensible et sujet à des erreurs.



Voici quelques informations qui pourraient être nécessaires dans un script. Choisissez pour chacune d'elles un nom de variable adéquat.

Informations	Noms de variable
Date au format AAAAMMJJ	
Surface en m ² du sol d'une chambre de kot	
Nombre de jours sans jouer sur son PC	
Première lettre de votre nom de famille	
Avoir un permis de conduire	
Âge de l'étudiant(e) le(la) plus jeune	

La valeur d'une variable

La valeur assignée à une variable peut se présenter sous la forme d'un littéral ou d'une expression. Le concept d'expression est abordé plus loin dans cet atelier.

Les **littéraux** sont des notations permettant d'écrire des valeurs littéralement dans le code d'un script. Le format d'écriture d'un littéral dépend de son type...

- **Entier** : nombre entier signé, par défaut, de type `int`.
On peut modifier la base et le signe. Ils n'ont comme limite de taille que la mémoire disponible...

Exemples

- décimale 430 -79 0
- octale 0o14 (12)₁₀ -0o765 (-501)₁₀
- hexadécimale 0xD (13)₁₀ -0x2B (-43)₁₀

- **Réel** : nombre exprimé en base 10, contenant un point décimal et éventuellement un exposant, par défaut, de type `float`.
La précision des réels en Python correspond à la double précision de la norme IEEE754.

Exemples

0. 1.57 3.14e+5 34.6E-3

- **Chaîne de caractères**² : suite de caractères entourée de "guillemets" ou 'apostrophes', de type `str`.

Exemples

- 'la musique.'
 - '"Oui", répondit-il,'
 - 'j'aime bien'
 - "e"
- OU
- "la musique."
 - "\"Oui\", répondit-il,"
 - 'j\'aime bien'
 - 'e'

Il n'y a pas de type pour un caractère tout seul.

² Plus d'informations sur <https://docs.python.org/fr/3.7/tutorial/introduction.html#strings>

- **Booléen** : vous avez déjà l'habitude de manipuler des valeurs entières, réelles et même les caractères, mais les booléens vous sont peut-être un peu moins familiers. En effet, le type `bool` correspond aux valeurs suivantes `True` et `False`.

Concrètement, dans un formulaire, si vous désirez savoir si un étudiant est dans le bloc 1, vous pouvez proposer un choix sous la forme d'un petit carré à sélectionner comme à la Figure 3.

J'ai des cours dans les blocs suivants :

- ☒ Bloc 1
- ☐ Bloc 2
- ☐ Bloc 3

Figure 3 - Exemple de booléens

Si le carré associé au bloc 1 n'est pas sélectionné, c'est que l'étudiant n'est pas dans le bloc 1, l'affirmation « J'ai des cours dans le bloc 1 » est fausse ; si le carré est sélectionné, l'affirmation « J'ai des cours dans le bloc 1 » est vraie.

Pour mémoriser cette information, le plus simple est donc d'utiliser une valeur booléenne. Dans le cas où l'étudiant a des cours dans ce bloc, la valeur est `True`, et si non la valeur est `False`. Il n'y a pas d'autres valeurs possibles.

L'adresse de la zone mémoire

Comme vous l'avez vu précédemment, chaque affectation correspond à la réservation d'une zone mémoire. On parle d'allocation de mémoire.

Sur base des quatre instructions d'affectation ci-dessous,

```
age = 18           # entier
message = "Hello world !" # chaîne de caractères
pi = 3.14159       # réel
majeur = True      # booléen
```

et en schématisant chacune des valeurs au moyen d'une forme géométrique, on obtient la représentation proposée à la Figure 4.

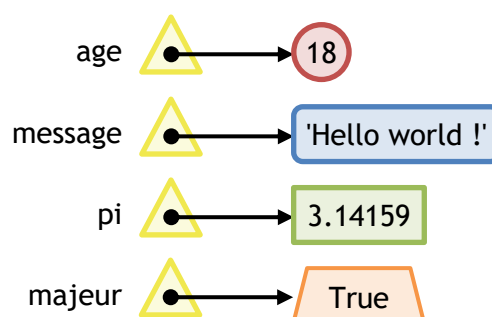


Figure 4 - Formes de valeurs

Quelle que soit la forme de la valeur, l'allocation de mémoire (réalisée lors de la première affectation) associe toujours l'adresse de la zone allouée au nom de la variable.

Une fonction est fournie pour permettre de connaître l'adresse contenue dans une variable : `id()`.

🧠 Dans le Shell, tapez l'instruction suivante et demandez son adresse au moyen de l'appel à `id` sur cette variable.

```
x = 5
id(x)
```

Vous devriez obtenir un résultat similaire à la Figure 5. Bien sûr, l'adresse affichée n'est pas la même puisqu'elle dépend de la machine sur laquelle le script est exécuté et est déterminée par le système d'exploitation !

```
>>> x = 5
>>> id(x)
140723744531408
```

Figure 5 - Adresse de variable

Cela signifie que le nom de la variable, ici `x`, est associé à l'adresse en question. La valeur `5` est donc à cette adresse dans la mémoire.

🧠 Pour vous familiariser avec cette outil, procédez de la même façon pour les variables proposées ci-dessus.

Le type de la variable

Le type de la variable est celui de la valeur qu'elle contient.

Selon le schéma ci-dessus, le type de la variable correspond à la forme de la valeur qu'elle référence, comme le montre la Figure 6.

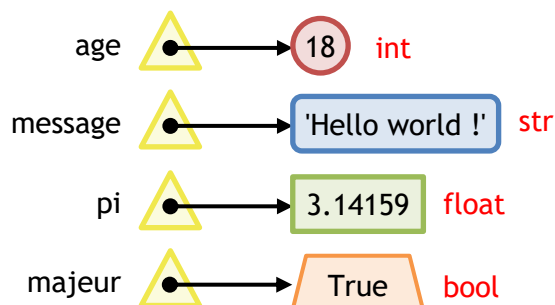


Figure 6 - Type de valeurs

Une fonction est fournie pour permettre de connaître le type d'une variable/valeur : `type()`.

🧠 Dans le Shell, tapez l'instruction suivante.

```
type(x)
```

Vous devriez obtenir le résultat présenté à la Figure 7.

```
>>> x = 5
>>> id(x)
140723744531408
>>> type(x)
<class 'int'>
```

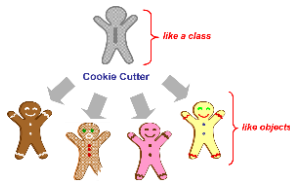
Figure 7 - Type de variable

🧠 À nouveau, procédez de la même façon pour les variables proposées ci-dessus.

Classe et objet

Devant le nom du type (ici `'int'`) est inscrit le mot `'class'`, qui signifie classe.

Comme vous l'avez vu lors de la présentation du cours, il existe plusieurs paradigmes de programmation (procédural, fonctionnel, logique...). Il existe aussi plusieurs façons d'analyser un problème et donc d'écrire le programme qui permet d'y apporter une solution. L'une d'entre elles est la programmation orientée objet. Cette façon de rédiger un programme repose sur les concepts de classe et d'objet. Vous aborderez les nombreux autres concepts liés à cette façon de faire dans la suite du cours voire même de votre formation.



Bien que, dans les scripts que vous allez écrire dans le cadre de ce cours, vous n'allez jamais écrire vos propres classes, il semble utile de comprendre ce que ces termes cachent. En effet, en Python, toutes les valeurs que vous allez manipuler sont en fait des objets.

On pourrait voir la classe comme un moule à gâteau et les objets comme les gâteaux fait à partir de ce moule.

Une **classe** correspond à un « type ». Les **objets** sont les « valeurs » produites lorsqu'on instancie une classe. Les objets générés sur base de la classe sont appelés les instances de la classe.

Python offre plusieurs types natifs. Ces types sont en effet définis dans la syntaxe du langage (sous la forme de classes). En voici la liste :

- None
- **Booléens** (`bool`)
- Nombres
 - Entiers (`int`)
 - Réels (`float`)
 - Complexes (`complex`)
- Séquences
 - Lists (`list`)
 - **String** (`str`)
 - Tuples (`tuple`)
 - Ranges (`range`)
 - Bytes
 - Bytearray
- Ensembles
 - Sets
 - Frozenset
- Dictionnaires

Dans un premier temps, vous n'allez travailler qu'avec les **booléens**, les **nombres entiers**, les **nombres réels** et les **chaines de caractères**.

Typage dynamique et implicite

Afin de déterminer le type d'une variable, l'interpréteur doit pouvoir déterminer le type de la valeur qui lui est affectée.

On parle de **typage dynamique**, dans le sens où il est réalisé à l'exécution de l'instruction d'affectation, et de **typage implicite** puisqu'il repose sur le travail de déduction de l'interpréteur.

Le type d'une valeur est défini en fonction de la manière dont la valeur est écrite. En effet, lorsqu'on écrit une valeur explicitement/**littéralement** dans un script, il est nécessaire de respecter un certain format.

🧠 Déterminez le type de chaque littéral proposé dans le tableau ci-dessous.

Littéral	Type
True	
False	
430	
-79	
0o15	
0x2D	
6.5	
3.14e+5	
"musique"	
'musique'	

Vous pouvez bien entendu corriger vos réponses via le Shell grâce à la fonction `type()`.

🧠 Affectez quelques-unes de ces valeurs à la variable `x` et vérifiez son type à chaque fois.

C'est lors de la dernière affectation d'une valeur à la variable que son type est déterminé. En effet, à chaque fois qu'on affecte une nouvelle valeur à la variable, son type peut changer.

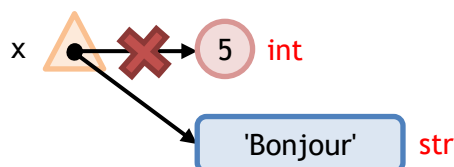
Le fait de faire une affectation à une variable qui avait déjà été affectée s'appelle la **réaffectation**.

Comme vous l'avez compris, la première affectation d'une variable permet de déclarer cette variable et de lui donner un type sur base de la valeur qui lui est associée. Lorsque vous faite une affectation par la suite, la valeur référencée par cette variable change et donc son type peut également changer puisque le typage est dynamique et implicite.

Les instructions de la Figure 8 en sont un exemple.

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> x = "Bonjour"
>>> type(x)
<class 'str'>
```

Figure 8 - Réaffectation - type



9 - Schéma de réaffectation

Après la seconde affectation, la variable `x` ne contient plus l'adresse d'un entier, mais celle d'une chaîne de caractère. La Figure 9 montre le schéma de ce qui se passe en mémoire.

Muable et immuable

Certains types sont dit **immuables**, c'est-à-dire qu'on ne peut pas changer leur valeur au cours du script. Si vous réaffectez une variable immuable, c'est une nouvelle référence qui est associée au même nom de variable, comme le montre la Figure 10.

```
>>> x = 5
>>> id(x)
140723484550096
>>> x = "Bonjour"
>>> id(x)
2471399827976
```

Figure 10 - Réaffectation - référence

D'autres types sont **muables**, c'est-à-dire qu'on peut changer leur valeur au cours du script. Ces types seront abordés plus tard dans le cours.

Voici un tableau reprenant certains de ceux-ci selon qu'ils sont muables ou pas.

Immuable	Muable
int	list
float	dict
complex	set
bool	...
string	
tuple	
range	
...	



Pour vous permettre de bien comprendre cette histoire de type immuable et de référence, voici quelques instructions à entrer dans le Shell. Pour chacune, observez l'adresse associée à la variable affectée ou réaffectée au moyen de la fonction `id()`.

- `a = 7`, notez son adresse :
- `a = 5`, notez son adresse :

Que constatez-vous ?

Normalement, l'adresse associée à la variable a changé. En effet, les deux valeurs étant différentes, il s'agit de deux objets différents, ayant chacun leur propre adresse. N'hésitez pas à accompagner votre démarche par quelques dessins comme vous en avez vu précédemment.

- `b = 500`, notez son adresse :
- `c = 500`, notez son adresse :

Que constatez-vous ?

Chacune des variables a une adresse différente, bien que ce soit la même valeur... En effet, si on veut modifier `b` en lui affectant maintenant 700, la valeur de `c` n'est pas supposée changer.

- `b = 700`, notez son adresse :

Vérifiez les valeurs de `b` et de `c`. Il suffit de taper leur nom à l'invite du Shell...


Observez maintenant une autre particularité de Python :

- `d = 5`, notez son adresse :

Comparez l'adresse associée à `a` et celle associée à `d`. Selon ce que vous venez de constater avec `b` et `c`, bien qu'il s'agisse de la même valeur, l'adresse devrait être différente !

Pourquoi `a` et `d` ont-elles la même référence, vers 5, mais `b` et `c` ne contenaient pas la même référence, vers 500 ?

En fait, il est intéressant de savoir que les entiers entre -5 et 256 sont mis en **cache** (https://docs.python.org/3/c-api/long.html#c.PyLong_FromLong) : « The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object... ». Le but est de limiter les allocations de mémoire répétées pour des nombres qui sont très souvent utilisés.

 Procédez de la même façon avec les instructions ci-dessous. Pour chacune, observez l'adresse associée à la variable affectée ou réaffectée au moyen de la fonction `id()` et notez cette adresse dans le tableau ci-dessous.


Instruction	Référence
<code>s1 = "bonjour"</code>	
<code>s2 = "bonjour"</code>	
<code>s1 = "au revoir"</code>	

B. Entrées et sorties

Les instructions d'entrée/sortie sont des appels de fonctions natives qui permettent de dialoguer avec l'utilisateur.

Sorties

Comme vous l'avez découvert dans vos premiers scripts, la fonction qui permet d'afficher un message est `print()`.

 Sur base de ce que vous avez pu tester précédemment, écrivez les instructions qui permettent de réaliser les scripts suivants :

1. Affecter une valeur de type numérique (entière ou réelle) à la variable `nbr` et afficher cette valeur.
2. Affecter une valeur de type chaîne de caractères à la variable `message` et afficher cette valeur.
3. Affecter une valeur aux variables `nb1` et `nb2` et afficher le produit de ces nombres, comme dans l'exemple suivant, en effectuant directement le produit dans l'instruction d'affichage au moyen de l'opérateur `*` :

Le produit de 5 et 8 vaut 40.

Entrées

L'instruction qui permet de récupérer les valeurs entrées au clavier est `input()`.



Entrez l'instruction suivante dans le Shell et exécutez-la (appuyer sur ENTER) :

```
|| input()
```

Que se passe-t-il ?

Appuyez à nouveau sur ENTER. Que se passe-t-il ? Comment interprétez-vous cela ?

Recommencez mais au lieu d'appuyer directement sur ENTER après avoir exécuté l'instruction, entrez votre prénom et ensuite appuyez sur ENTER. Que se passe-t-il ? Cela confirme-t-il ce que vous aviez déduit ci-avant ?



Soyez un peu plus convivial avec l'utilisateur ! Commencez par préciser ce qu'il est supposé entrer comme information, en l'occurrence le message « Prénom : ».

Quelle solution avez-vous trouvée ?

Si vous ajoutez l'instruction suivante juste avant `input()`, cela donne-t-il l'affichage attendu ?

```
|| print("Prénom : ")
```

Vous devriez avoir quelque chose de semblable à ce qui est proposé à la Figure 11.

```
>>> print("Prénom : ")
Prénom :
>>> |
```

Figure 11 - Affichage

Il n'est donc pas possible de rentrer la réponse à la « question » directement... L'invite de commande primaire est en effet à nouveau proposée et une nouvelle instruction doit être introduite pour poursuivre le script. L'objectif est d'avoir un affichage semblable à celui de la Figure 12.

```
Prénom : |
```

Figure 12 - Invitation à répondre

Essayez de trouver une manière de résoudre ce problème en vous référant au livre en ligne : https://fr.wikibooks.org/wiki/Programmation_Python/Afficher_un_texte.

Cette étape réalisée, il vous reste à trouver la solution pour faire en sorte que le prénom soit mémorisé afin d'être réutilisé dans l'instruction suivante :

```
|| print("Bien joué, ", prénom, " !")
```

La réponse se trouve au même endroit...



Maintenant que vous savez récupérer des informations entrées par l'utilisateur, exercez-vous un peu :

1. Écrivez l'instruction qui permet de récupérer votre âge dans la variable `age`.
2. Affichez un message contenant votre prénom et votre âge, comme dans l'exemple suivant, les deux éléments soulignés en pointillés étant les valeurs récupérées précédemment :

```
Guido a 62 ans
```

3. Demandez à l'utilisateur les valeurs des variables `nb1` et `nb2` (en deux instructions).
4. Affichez le produit de ces deux nombres, en calculant directement le produit dans l'instruction d'affichage. Le message produit doit ressembler à ceci :

```
Le produit de 5 et 8 vaut 40
```

Cette dernière instruction devrait poser un problème. Quel est le message d'erreur ? Pourquoi se produit-il ? La solution à ce problème est proposée plus loin dans cet atelier... Vous y reviendrez plus tard !

Input...

Pour préciser quelle information est attendue, vous pouvez donner un message sous la forme d'une chaîne de caractères à la fonction. Afin de pouvoir utiliser l'information récupérée, il faut l'affecter à une variable. L'instruction ci-dessous est une instruction qui demande à l'utilisateur d'entrer un nombre et elle le mémorise dans la variable `nombre`.

```
nombre = input("Nombre : ")
```

Son exécution commence par l'affichage du message comme proposé à la Figure 13.

```
>>> nombre = input("Nombre : ")
Nombre : |
```

Figure 13 - Input

Le Shell attend que l'utilisateur entre une réponse afin de continuer l'interprétation du script. Après que l'utilisateur ait entré la réponse au clavier et appuyé sur la touche ENTER, le script se poursuit par l'affectation de la valeur entrée à la variable prévue à cet effet, celle qui est à gauche du `=`.

```
>>> nombre = input("Nombre : ")
Nombre : 5
>>> nombre
'5'
```

Figure 14 - Résultat de l'input

Lorsqu'on demande au Shell la valeur de la variable `nombre`, on remarque les apostrophes autour de la réponse, comme présenté à la Figure 14. En effet, les valeurs fournies par l'utilisateur lors d'un appel à la fonction `input` sont transmises au script sous la forme d'une chaîne de caractères.

C. Instructions et expressions

Ces deux termes représentent des notions distinctes, bien que liées l'une à l'autre.

Instruction

Une **instruction** est un bout de code qui correspond à une **action** : afficher un message, mettre une valeur dans une variable...

Lorsque vous tapez une instruction à l'invite du Shell et que vous l'exécutez, l'interpréteur n'affiche rien... sauf dans le cas d'un **print** dont c'est le but. Il se contente de l'exécuter !



Testez-le avec l'instruction suivante :

```
|| somme = 2 + 3
```



Créez un nouveau fichier source, nommé **somme.py**. Écrivez-y le code source proposé ci-dessous.

```
|| print("somme vaut", somme)
|| somme = 2 + 3
```

Exécutez le script (F5). Que constatez-vous ?

Sans modifier les instructions, remettez de l'ordre dans le script afin qu'il affiche le message suivant :

```
| Somme vaut 5
```

Comme vous l'avez compris, un programme est une suite d'instructions. Les instructions sont exécutées les unes à la suite des autres et forment ce qu'on appelle une **séquence d'instructions**.

L'ordre des instructions est primordial lors de l'exécution de vos scripts. Par exemple, une instruction affichant la valeur d'une variable, si elle est exécutée avant d'avoir affecté une valeur à celle-ci, n'affiche pas la valeur espérée mais celle qui est à ce moment-là dans la zone mémoire correspondante ! Elle provoque même une erreur si la variable n'existe pas !

Une séquence d'instructions est une des façons d'agencer les instructions les unes par rapport aux autres. Vous pourrez bientôt modifier l'ordre dans lequel les instructions sont exécutées, appelé le **flux des instructions**, au moyen d'instructions de **contrôle de flux** et ainsi faire un choix entre deux actions, répéter une ou plusieurs actions...



Recopiez le code proposé ci-dessous à l'invite du Shell.

```
|| nb_torches_sac = 20 print("Il y a", nb_torches_sac, "torches dans
|| votre sac.\n")
```

Exécutez le script. Que se passe-t-il ?

Recommencez en écrivant ces instructions chacune sur une ligne.

```
|| nb_torches_sac = 20
|| print("Il y a", nb_torches_sac, "torches dans votre sac.\n")
```

Exécutez le script. Que se passe-t-il ?

À votre avis, qu'est ce qui sépare 2 instructions en Python ?

Voici un script dont le but est de déterminer le nombre de torches nécessaires pour avoir un tas (*stack*) de torches, c'est-à-dire 64 torches.

```
nb_torches_sac = 20
print("Il y a", nb_torches_sac, "torches dans votre sac.\n")
nb_torches_a_faire = 64 - nb_torches_sac
print("Il faut faire ", nb_torches_a_faire, " torches !\n")
```



Recopiez ce script en respectant l'indentation dans un nouveau fichier, nommé `torches.py`. Il est conseillé d'écrire vous-même les instructions plutôt que de faire un copier/coller. En effet, c'est en rédigeant du code que vous vous familiarisez avec l'IDLE. De plus, le Shell supporte très mal le copier/coller à cause de l'indentation...

Exécutez le script (F5). Quel est le message affiché ? Que signifie-t-il ?

Corrigez les erreurs et exécutez à nouveau le script...

Vous en concluez que toutes les instructions d'un même groupe d'instructions, appelé **bloc d'instructions**, doivent être au même niveau d'indentation. En Python, ce sont donc les niveaux d'indentation qui définissent les blocs d'instructions.

Indentation



Dans d'autre langage, les blocs d'instructions sont englobés dans des accolades et l'indentation n'est donc pas une priorité... Cependant, le fait que le code soit bien indenté permet d'en faciliter la lecture. Cette notion est donc intégrée dans la syntaxe du langage !

Les instructions « principales » du script, celles qui sont exécutées en premier, sont celles qui se trouvent à la colonne 0 comme montré à la Figure 15. La première de ces instructions est ce qu'on appelle le **point d'entrée** du script.

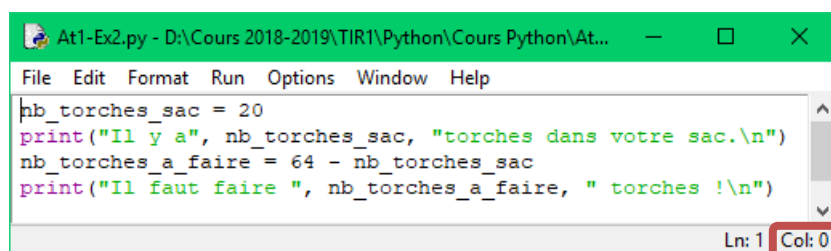


Figure 15 - Instructions principales

Expression

Dans une instruction, une valeur peut être un **littéral** (42, 'Z', 3.1415) mais elle peut aussi être le résultat de l'**évaluation d'une expression**.

Une **expression** est une combinaison de littéraux, de variables, d'opérateurs, etc. qui, lorsqu'on la calcule/l'évalue, produit une nouvelle **valeur** :

`5 * 8` `nombre_lu + 3` `lettre == 'a'`

Lorsque vous tapez une expression à l'invite du Shell et que vous l'exécutez, elle est évaluée et le résultat est affiché !



Exécutez quelques-unes des expressions ci-dessus afin de le constater vous-même !

Il existe plusieurs opérateurs permettant de former des expressions. Le point suivant permet d'en aborder quelques-uns.

D. Opérateurs

Les notions de variable et de littéral ont déjà été abordées ci-avant. Il vous reste donc à comprendre ce qu'est un opérateur et comment les utiliser.

Les **opérateurs** que vous connaissez pour le moment sont les opérateurs arithmétiques (+, -, *, /, %) et ceux de comparaison (<, ≤, ≥, >, =, ≠). À ceux-ci s'ajoute l'opérateur d'affectation qui a déjà été abordé ci-avant.

Opérateur d'affectation

Dans la plupart des langages de programmation, l'affectation se fait au moyen de l'opérateur `=`.

 Recopiez l'instruction proposée ci-dessous à l'invite du Shell.

```
| print(nb = 1)
```

Exécutez le script. Que se passe-t-il ?

Recommencez avec l'instruction suivante.

```
| x = (y = 1)
```

Exécutez le script. Que se passe-t-il ?

En Python, l'affectation est une instruction, et ne peut être considérée comme une expression, d'où la réaction de l'interpréteur concernant ces deux instructions.

Quelques informations complémentaires peuvent être apportées à la notion d'affectation.

 Modifiez le fichier source `somme.py` en remplaçant l'instruction

```
| somme = 2 + 3  
par
```

```
| 2 + 3 = somme
```

Exécutez le script et observez ce qui se produit.

Vous devriez avoir le message d'erreur `SyntaxError: can't assign to operator`.

L'opérateur d'affectation `=` suit une règle bien précise et générale à la plupart des langages de programmation : c'est la valeur située à la droite du `=`, appelée **expression de droite**, qui est mémorisée dans la zone mémoire située à la gauche du `=`, appelée **expression de gauche**.

Le message d'erreur ci-dessus sous-entend donc que l'opérande à gauche de l'opérateur `=` doit être une expression de gauche, c'est-à-dire une zone mémoire !

 Corrigez le code source pour que l'instruction d'affectation soit correctement écrite. Exécutez à nouveau le code source et vérifiez que tout se passe comme vous l'espérez.

On peut donc préciser la syntaxe de l'instruction d'affectation comme ceci :

```
|| <l-value> = <r-value>
```

Il faut y remplacer

- <l-value> par une expression de gauche, une zone mémoire,
- <r-value> par une expression de droite, une valeur.



Écrivez les affectations correspondant aux affirmations suivantes :

1. Le nombre de points de vie de votre personnage est de 45
2. Le taux de gaucher dans le monde est de 12,7 %
3. Le prénom le plus donné en Belgique en 2018 est Arthur
4. La lettre commune à vos deux sections est le i
5. Le personnage est en pleine forme si ses points de vie sont supérieurs à 53

Avez-vous choisi des noms de variables permettant de savoir à quoi correspond la valeur en question ?

Dans le dernier cas, vous pouvez constater que les expressions de gauche peuvent être utilisées en tant qu'expressions de droite. En effet comme une expression de gauche est une zone mémoire contenant une valeur, elle peut être utilisée dans une expression de droite.

Par exemple, une variable est à la fois

- une expression de gauche lorsqu'on la considère comme étant la zone mémoire dans laquelle on désire mémoriser une valeur, par exemple `nb_torches_sac = 42`,
- une expression de droite lorsqu'on la considère comme étant la valeur que l'on désire utiliser comme opérande d'une expression, par exemple `nb_torches_a_faire = 64 - nb_torches_sac`.



Reprenez le code source du fichier `torches.py` et intégrez-y convenablement les instructions suivantes.

```
|| nb_torches_max = 64
|| nb_torches_sac = 42
|| nb_torches_a_faire = nb_torches_max - nb_torches_sac
```

Dans ce même fichier source, affichez les résultats de ces instructions, en passant

- soit par l'expression, comme vous l'avez déjà fait précédemment avec la somme :

```
|| print("Somme : ", 2 + 3)
```
- soit par la variable contenant la valeur résultant d'évaluation de cette expression :

```
|| print("Somme : ", somme)
```

L'affichage doit être celui présenté ci-dessous.

```
Vous devez refaire 22 torches.
```

À nouveau, ces instructions d'affichage font intervenir une expression de droite, c'est-à-dire une valeur, soit sous la forme d'une expression, soit sous la forme d'une variable.

Opérateurs arithmétiques

Petite précision sur l'opérateur `-` (et le `+`) car il peut être envisagé de deux façons :

- s'il porte sur un seul opérande, on parle d'**opérateur unaire** : `-x` `-(8)`
Remarque : Dans `-8` et `-5`, le `-` fait partie de l'écriture du littéral.
- s'il est utilisé entre deux opérandes, on parle d'**opérateur binaire** : `x - y`

Les autres opérateurs, listés plus haut, sont des opérateurs binaires puisqu'ils font intervenir deux opérandes : `5 != 8` `3.5 * 9` `7 % 2`

Le Shell peut être utilisé comme une calculatrice. En effet, comme vous l'avez vu précédemment, lorsque vous y tapez une expression arithmétique, il affiche la valeur qui en résulte.



Entrez les expressions suivantes les unes à la suite des autres et complétez le tableau avec leur résultat ainsi que leur type :

Expression	Résultat	Type	Expression	Résultat	Type
<code>6 + 4</code>			<code>6 * 4</code>		
<code>6.0 + 4</code>			<code>6.0 * 4</code>		
<code>4 + 6.0</code>			<code>4 * 6.0</code>		
<code>4. + 6.0</code>			<code>4.0 * 6.0</code>		
<code>6 - 4</code>			<code>6 / 4</code>		
<code>6.0 - 4</code>			<code>6.0 / 4</code>		
<code>4 - 6.0</code>			<code>4 / 6.0</code>		
<code>4.0 - 6.0</code>			<code>4.0 / 6.0</code>		

Vous pouvez constater que le résultat de l'évaluation d'une expression n'a pas nécessairement le même type que les opérandes de celle-ci. De plus, que vous tapiez `6.0` ou `6.`, le résultat est le même. Ces deux littéraux sont équivalents.



Créez un nouveau fichier source `vacances.py` dans lequel vous allez recopier le code source proposé ci-dessous. Complétez les instructions d'affichage en remplaçant chaque `...` par une expression, utilisant les variables, afin d'afficher les valeurs demandées !

```
nb_jours_vacances = 27
nb_jours_vacances_enseleilles = 11
prix_vacances_par_jour = 7.50
budget_vacances = 300.00
depenses_par_jour_enseleille_en_vacances = 8.25
print("Nombre de jours de vacances non enseleillés :", ...)
print("Prix total des vacances :", ...)
print("Nombre de jours de vacances autorisés par le budget :", ...)
print("Prix correspondant aux jours de vacances enseleillés :", ...)
print("Dépenses totales pour les jours de vacances enseleillés :", ...)
print("Pourcentage de jours enseleillés sur les vacances :", ...)
```



Ajoutez les instructions ci-dessous. Complétez-les avec les expressions nécessaires à l'affichage des valeurs demandées ainsi que le format d'affichage adéquat.

```
print("Nombre de semaines entières passées en vacances :", ...)
print("Nombre de jours de vacances en plus des semaines entières :", ...)
```

Avez-vous trouvé la façon la plus efficace de répondre à ces deux dernières demandes ?

Il arrive fréquemment de ne pas vouloir avoir un résultat réel, mais bien entier. En Python, l'opérateur qui permet de faire une division entière est `//`.

Un opérateur que certains ne connaissent pas et qui est pourtant très utile est le modulo, `%`. Il permet de récupérer le reste de la division entière d'un nombre par un autre.

Dans la division euclidienne, que vous devriez avoir vu en primaire, voici les différents éléments de la division :

$$\begin{array}{r}
 \text{dividende} \dots 23 \overline{) 4} \dots \text{diviseur} \\
 \underline{- 20} \dots \text{quotient} \\
 \text{reste} \dots 3
 \end{array}$$

Dans le cas du `//`, on récupère le quotient, alors que dans le cas du `%`, on récupère le reste.

Un autre opérateur intéressant est le `**`. À quoi sert-il ?



Entrez les expressions suivantes les unes à la suite des autres et complétez le tableau avec ces résultats ainsi que leur type :

Expression	Résultat	Type	Expression	Résultat	Type
<code>6 ** 4</code>			<code>6 % 4</code>		
<code>6. ** 4</code>			<code>6. % 4</code>		
<code>4 ** 6.</code>			<code>4 % 6.</code>		
<code>4. ** 6.</code>			<code>6. % 4.</code>		
<code>6 // 4</code>			<code>6 / 0</code>		
<code>6. // 4</code>			<code>6. / 0</code>		
<code>4 // 6.</code>			<code>6 // 0</code>		
<code>4. // 6.</code>			<code>6 % 0</code>		



Afin de vous assurer que vous ayez bien compris quand et comment utiliser les opérateurs `//` et `%`, ajoutez les initialisations ci-dessous dans le code source de `vacances.py`, ainsi que les instructions proposées. Complétez à nouveau ces instructions comme précédemment.

```

heure_depart_HHMM = 1427
duree_voyage_en_minutes = 134

print("Heure de l'heure de départ :", ...)
print("Minutes de l'heure de départ :", ...)
print("Temps de trajet :", ... , "h et", ... , "min")
print("Nombre d'heures entières dans la durée du voyage :", ...)
print("Nombre de minutes de voyage en plus des heures entières :", ...)
print("Heure de l'heure d'arrivée :", ...)
print("Minutes de l'heure d'arrivée :", ...)

```



Recommencez cet exercice, mais avec une heure de départ initialisée à `1143` et une durée de voyage en minutes initialisée à `92`. N'oubliez pas de vérifier que votre réponse est plausible : les minutes ne peuvent pas excéder 59 !

Opérateurs de comparaison

Les opérateurs de comparaison servent à comparer des valeurs, et ont pour résultat une valeur booléenne, c'est-à-dire vrai ou faux. En Python, les littéraux correspondant aux valeurs booléennes sont `True` et `False`.

Deux précisions doivent être apportées concernant la syntaxe des opérateurs de comparaison :

- dans le cadre de la rédaction du code source d'un programme, vous ne pouvez pas utiliser les symboles \leq , \geq et \neq . Ceux-ci sont respectivement remplacés par `<=`, `>=` et `!=` (le « ! » signifiant « *not* »/« pas », le symbole « != » signifiant ainsi « différent » ou « *not equals* »/« pas égal »).
- l'opérateur de comparaison `=` ne peut être écrit tel quel car il pourrait être confondu avec l'opérateur d'affectation. Or, l'interpréteur n'a pas à faire de choix quant à la signification d'un opérateur... La syntaxe de l'opérateur d'égalité est donc `==`.



Toujours dans ce même fichier `vacances.py`, ajoutez et complétez les instructions ci-dessous avec les expressions de comparaisons adéquates.

```
print("Il n'a jamais fait ensoleillé durant les vacances :", ...)  
print("Les vacances ont dure moins de 15 jours :", ...)  
print("Il a fait ensoleillé tous les jours des vacances :", ...)  
print("Le budget n'est pas suffisant pour couvrir le prix des jours de  
      vacances :", ...)  
print("Le voyage dure moins de 2 heures :", ...)  
print("Le depart est prévu apres midi :", ...)  
print("Le cout total des vacances (prix par jour + dépenses des jours  
      ensoleillés) dépasse le budget :", ...)  
print("Le coût moyen d'un jour de vacances n'atteint pas les 12  
      euros :", ...)
```

Priorités des opérateurs

Les expressions peuvent être plus complexes en associant différents opérateurs. Afin de pouvoir évaluer ces expressions, il est nécessaire de donner des priorités aux opérateurs.

Vous connaissez probablement déjà certaines de ces priorités comme celle concernant les opérateurs `+` et `*`. En effet, dans l'expression `3 + 4 * 5`, vous devez d'abord multiplier 4 par 5, et ensuite ajouter ce résultat à 3, obtenant ainsi le résultat de 23.

Ce mécanisme qui amène certains opérateurs à être pris en compte avant d'autres s'appelle la **priorité des opérateurs/opérations**.

Comme dans tout langage, la norme du Python définit précisément les priorités de chacun des opérateurs et suppose que les opérateurs d'une priorité plus grande (plus élevée) sont effectués avant les opérateurs d'une priorité inférieure.

Dans l'expression ci-dessus, le `*` a une priorité plus élevée que le `+`.



Évaluez les expressions suivantes sans passer par un programme. Faites-le également en ajoutant des parenthèses autour de la première sous-expression puis autour de la seconde sous-expression. Dans l'exemple ci-dessus, vous auriez eu les trois expressions suivantes : $3 + 4 * 5$, $(3 + 4) * 5$ et $3 + (4 * 5)$.

```
15 + 3 * 2
3 * 4 % 5
15 % 4 / 3
3 * 4 % 5
15 % 4 % 3
13.5 % 5 % 1.5
2 ** 3 ** 2
```

Vérifiez vos solutions avec le Shell. Avez-vous obtenu les mêmes résultats ? Si non, à votre avis, quelle en est la raison ?

Voici le tableau des priorités des opérateurs arithmétiques et de comparaison :

<div style="display: inline-block; transform: rotate(-45deg); transform-origin: left top; width: 50px; height: 50px; border-left: 1px solid black; border-bottom: 1px solid black; position: relative;"> Priorité </div>	**	←
	-(unaire)	→
	* / // %	→
	+ -(binaire)	→
	< <= > >= != ==	→

Vous remarquez que certains opérateurs ont la même priorité, ils sont sur la même ligne. Cependant, il y a également un ordre à respecter dans l'évaluation d'expressions composées d'opérateurs ayant la même priorité. Dans le cas d'une expression telle que $5 + 3 - 2$, l'évaluation des expressions se fait de gauche à droite (→) $5 + 3$, qui donne 8, pour ensuite évaluer $8 - 2$. En Python, le seul opérateur dont l'évaluation a lieu de la droite vers la gauche (←) est l'exposant ******.



Sans les exécuter, donnez la valeur des variables et précisez le type du résultat après les instructions suivantes :

```
val1 = 10 + 6 / 2
val2 = 3 * 2 + 4 * 3
val3 = 2 <= 1 + 4
val4 = 2 + 1 != 3
val5 = 24 + 36 % 12
```

Vérifiez vos solutions avec le Shell. Avez-vous obtenu les mêmes résultats ?

Cette notion semble futile dans certains cas, mais vous avez pu voir que cela peut engendrer des erreurs inattendues... pour ceux qui n'ont pas pris la peine de bien comprendre ces propriétés.



Dans les instructions suivantes, ajoutez des parenthèses uniquement là où elles sont absolument nécessaires pour que la valeur de l'expression soit celle de la valeur attendue.

```
print("Expression :", 10 + 6 / 2, "- Attendue : 8")
print("Expression :", 3 * 2 + 4 * 3, "- Attendue : 54")
print("Expression :", 24 + 36 % 12, "- Attendue : 0")
```

N'hésitez pas à les tester dans le Shell.



Enlevez les parenthèses des expressions suivantes lorsqu'elles peuvent être retirées.
Sur base des affectations, vérifiez que la valeur de l'expression correspond toujours à la valeur attendue.

```
a = 3
b = 7
print("Expression :", (a * 12) + (b - 5), "- Attendue : 38")
print("Expression :", (5 * a) + 2 * ((3 * b) + 4), "- Attendue : 65")
print("Expression :", (5 * (a + 2) * 3) * (b + 4), "- Attendue : 825")
```

E. Opérateurs particuliers

Opérateur « deux en un »

Vous savez que des variables peuvent être utilisées de part et d'autre de l'opérateur d'affectation =.

Il est fréquent de devoir modifier la valeur d'une variable sans pour autant vouloir la mémoriser sous un autre nom. Dans ce cas, on peut écrire

```
|| nb_torches_sac = nb_torches_sac + 22
```

ou, pour gagner quelques caractères

```
|| nb_torches_sac += 22
```

Ces deux instructions ont le même effet sur la variable `nb_torches_sac`, elle l'augmente de 22.

Bien sûr cette façon de faire est applicable aux autres opérateurs. Vous aurez de nouveaux opérateurs à considérer : `+=`, `-=`, `*=`, `**=`, `/=`, `//=` et `%=`.

Ces opérateurs ont une priorité moindre que les autres et seront donc à prendre en compte en dernier lieu.



Prévoyez ce que va afficher le programme suivant et vérifiez ensuite vos réponses...

```
nb_exercices = 7
print("nbExercices =", nbExercices)
nb_exercices += 3
print("nbExercices =", nbExercices)
nb_exercices *= 4
print("nbExercices =", nbExercices)
nb_exercices /= 5
print("nbExercices =", nbExercices)
nb_exercices %= 3
print("nbExercices =", nbExercices)
nb_exercices **= 5
print("nbExercices =", nbExercices)
nb_exercices //= 10
print("nbExercices =", nbExercices)
```

Conversion de type ou transtypage



Dans le fichier source `torches.py`, remplacez l'affectation de la valeur 42 à la variable `nb_torches_sac` par une instruction de lecture.

```
nb_torches_max = 64
nb_torches_sac = input("Nombre de torches dans ton inventaire : ")
nb_torches_a_faire = nb_torches_max - nb_torches_sac
```

Exécutez le script (F5). Quel est le résultat ?

L'interpréteur Python a parmi ses nombreux rôles celui de vérifier que les types utilisés dans les instructions sont compatibles entre eux.

Avant d'utiliser le résultat d'une expression, il faut en déterminer le type afin de s'assurer que le résultat de l'instruction corresponde à l'utilisation que vous en faites.

Dans l'exemple ci-dessus, la variable `nb_torches_sac` est garnie avec une chaîne de caractères. Comme cela a déjà été expliqué plus haut, la fonction native `input` permet de récupérer une valeur entrée par l'utilisateur, mais elle la considère comme étant une chaîne de caractères. L'addition d'une valeur entière et d'une chaîne de caractères ne fait pas partie de la définition du langage, d'où le message d'erreur.

Afin de faire en sorte que la variable `nb_torches_sac` contienne une valeur entière, il faut passer par un opérateur particulier, l'opérateur de **conversion de type** ou ***cast***. Cet opérateur est un opérateur unaire puisqu'il porte sur un seul opérande.

La syntaxe est la suivante :

```
<var_type>(<r_value>)
```

Il faut y remplacer

- `<var_type>` par un type, et
- `<r_value>` par une valeur.

Cet opérateur a une priorité plus importante que tous les précédents.



Corrigez et exécutez à nouveau ce script.

Afin d'avoir le résultat attendu, il faut forcer la valeur reçue de l'utilisateur à être mémorisée sous la forme d'un entier, comme montré ci-dessous.

```
nb_torches_max = 64
nb_torches_sac = int(input("Nombre de torches dans ton inventaire : "))
nb_torches_a_faire = nb_torches_max - nb_torches_sac
```

Les fonctions de conversion de type dont vous risquez d'avoir besoin sont les suivantes :

- `int()` permet de transformer une valeur en entier
- `str()` permet de transformer la plupart des valeurs d'un autre type en chaînes de caractère
- `float()` permet la transformation en réel



Modifiez les instructions (p13, points 3 et 4) permettant l’affichage du produit de deux nombres afin que cette fois le script affiche le résultat attendu !

1. Demandez à l'utilisateur les valeurs des variables `nb1` et `nb2` (en deux instructions).
2. Affichez le produit de ces deux nombres, en calculant directement le produit dans l’instruction d’affichage. Le message produit doit ressembler à ceci :

```
Le produit de 5 et 8 vaut 40
```

Concaténation

La concaténation est le fait de « coller » plusieurs chaînes de caractères les unes aux autres.



Tapez les chaînes de caractères, telles qu’elles sont proposées, l’une à la suite de l’autre à l’invite du Shell, sans appuyer sur ENTER !

- `"Oui", répondit-il,`
- `"j'aime bien "`
- `"la musique."`

Exécutez le script. Que se passe-t-il ?

Plusieurs chaînes de caractères, écrites littéralement, côte à côte, sont automatiquement concaténées. Vous pouvez remarquer que certains caractères sont « échappés » par un `\`. Ceci permet en effet de faire en sorte que le caractère soit considéré en tant que tel et non comme la fin de la chaîne en question.

Cependant, il se peut que ces chaînes de caractères aient été mémorisées dans des variables, comme ci-dessous.

```
oui = "Oui", répondit-il,
aime = "j'aime bien "
musique = "la musique."
```

Comment faire pour afficher les valeurs de ces trois variables côte à côte ?

L’opérateur de concaténation est le `+` et il ne prend en compte que les valeurs de type `str`.



Pour répondre à la question ci-dessus, vous devez écrire l’expression suivante :

```
oui + aime + musique
```

Il est cependant utile de remarquer que seules ces deux façons de faire sont supportées par le langage.



Testez les différents scripts proposés ci-dessous et tirez-en les conclusions qui s’imposent.

1. `prefix = "Py"`
`prefix "thon"`
2. `prenom = "Guido"`
`nom = "van Rossum"`
`prenom + " " + nom`
3. `"Python " + 3`

Typage « fort »

Dans la dernière expression de l'exercice ci-avant, `"Python " + 3`, l'interpréteur vérifie que les deux opérandes sont du même type :

- si la première valeur est un `int`, la seconde doit aussi être un `int` et l'opérateur est considéré comme étant l'opérateur d'addition de deux entiers (idem si ce sont des réels),
- si la première valeur est un `str`, la seconde doit aussi être un `str` et l'opérateur est considéré comme étant l'opérateur de concaténation de deux chaînes de caractères.

Ce phénomène est dû au fait que bien que les types n'apparaissent pas explicitement dans le script, de nombreuses vérifications sont effectuées lors de l'interprétation du code.

On parle de typage fort...

Cependant, la définition de typage fort et de typage faible est très floue³ et on ne va pas s'étendre là-dessus... Veuillez cependant à utiliser les bons opérateurs avec les bonnes valeurs.

La solution à notre problème, dans cet exemple est de forcer le 3 à être considéré comme étant une chaîne de caractères.



Quelle est-donc l'expression permettant d'afficher « Python 3 » au moyen de l'opérateur de concaténation ?

³ Voir notamment les articles : https://fr.wikipedia.org/wiki/Typage_fort et https://en.wikipedia.org/wiki/Strong_and_weak_typing