



UE121 - Introduction à la programmation

HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG

Technologie de l'informatique - bloc 1

Sécurité des systèmes - bloc 1

Atelier 2 : Structure de contrôle

Objectifs




- Découvrir la notion de fonction en Python
- Manipuler les structures de contrôles : alternatives, répétitives et fonctions
- Être capable de découper un script en fonctions
- Être capable de gérer la portée des variables

INTRODUCTION	2
A. QU'EST-CE QU'UNE CONDITION ?	2
B. ALTERNATIVES.....	4
C. RÉPÉTITIVES.....	9
D. POURQUOI DES FONCTIONS ?.....	11
E. FONCTION	15
F. PORTÉE DES VARIABLES	16

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*).

Cet atelier a pour objectif de vous permettre de comprendre les notations liées aux instructions de **contrôle de flux**.

Comme vous l'avez déjà observé précédemment, un programme impératif est une suite d'instructions qui sont exécutées les unes à la suite des autres. Ces instructions forment une **séquence d'instruction**.

L'ordre dans lequel les instructions s'exécutent est le **flux des instructions**. Comme vous allez le voir, ce flux peut être modifié grâce à certaines instructions qui sont donc appelées instructions de contrôle de flux.

Vous allez commencer par découvrir les alternatives et la façon de les écrire en Python. Ensuite vous pourrez vous aventurer dans les notions liées aux répétitives. Il est bien sûr important de bien comprendre le rôle de chacune de ces structures de contrôle afin de choisir celle qui est la plus adéquate quant au flux des instructions. Enfin, vous allez voir les définitions et les appels de fonction.

Pour pouvoir utiliser des alternatives et des répétitives, il est nécessaire de vous assurer d'avoir compris la notion de condition.

A. Qu'est-ce qu'une condition ?

Une condition est une **expression booléenne**, c'est-à-dire une expression dont la valeur est vraie ou fausse.



En Python, quels sont les littéraux qui correspondent à ces valeurs ? Attention à les écrire correctement !

Si vous ne savez plus de quoi il s'agit, retournez à l'atelier 1.

Ce type de valeur est le résultat d'une opération de comparaison. Pour pouvoir comparer des valeurs, qu'elles soient littérales, mémorisées dans des variables, le résultat de l'appel d'une fonction... on utilise les **opérateurs de comparaison**.

Les valeurs comparées peuvent être à priori de n'importe quel type (nombre ou chaîne de caractères). Cependant, en Python, il faut que les deux valeurs de la comparaison soient du même type ! La priorité de ces opérateurs a déjà été abordée dans l'atelier 1.

On peut bien entendu comparer la valeur d'une variable à un littéral ou les valeurs de deux variables... La comparaison de deux littéraux n'a pas vraiment de sens dans un script puisque vous devriez en connaître le résultat directement. Il est donc inutile de demander à la machine de le calculer pour vous !

Voici quelques exemples :

```
age > 18
nb_echecs == 0
nbr1 <= nbr2
```

Il s'agit donc d'évaluer la condition afin de savoir si le résultat est `True` ou `False`. Dans le premier exemple, si la valeur de la variable `age` est plus grande que `18`, la condition vaut `True`, sinon, la condition vaut `False`.



Testez les conditions ci-dessus après avoir affecté une valeur aux variables utilisées.

La Figure 1 montre un exemple de ce que l'évaluation de la première condition donne comme résultat selon la valeur donnée à `âge`, ici 8.

```
>>> âge = 8
>>> âge > 18
False
```

Figure 1 - Évaluation d'une condition



Trouvez les conditions qui correspondent aux propositions suivantes, en choisissant des noms de variables explicites et qui respectent les conventions proposées par le PEP8 (<https://www.python.org/dev/peps/pep-0008/>).

1. La température est plus grande que 15.
2. La somme est égale à 120.
3. Le nom est plus petit que « van Rossum » (selon l'ordre lexicographique).
4. Le nombre est différent de 0
5. Le nombre est pair

Pour ne pas répéter une même condition plusieurs fois dans un script, on peut passer par une variable de manière à mémoriser le résultat de l'évaluation de la condition.

```
est_masculin = sexe == 'M'
```

Cette instruction déclare la variable `estMasculin` et lui affecte le résultat de l'évaluation de la condition, c'est-à-dire `True` si le `sexe` vaut `'M'` et `False` sinon.

Enfin, certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées grâce à une simple comparaison. En effet, dans certaines conditions, se cachent plusieurs conditions. On parle de conditions composées. Les conditions sont alors reliées par des **opérateurs logiques** : ET, OU et NON, en Python, `and`, `or` et `not`.

Voici le tableau des priorités des opérateurs complété avec les opérateurs logiques.

**	←
-(unaire)	→
* / // %	→
+ -(binaire)	→
< <= > >= != ==	→
not	→
and	→
or	→



Trouvez les conditions qui correspondent aux propositions suivantes.

1. L'âge est positif et inférieur à 18
2. Le sexe est soit 'M', soit 'F'
3. L'année est bissextile (c'est-à-dire divisible par 4 et pas par 100, ou divisible par 400)

Afin de pouvoir utiliser ces informations dans la suite de votre script, affecter le résultat de ces deux conditions composées respectivement à la variable `age_valide`, `sexe_valide` et `annee_bissextile`.



Écrivez un script `infos.py` qui permet de demander et de récupérer l'âge et le sexe de l'utilisateur, et qui indique si ces informations sont valides au moyen des deux premières instructions que vous avez écrites ci-avant.



Écrivez un script `bissextile.py` qui permet de donner une année et qui indique si c'est une année bissextile en exécutant la bonne instruction.

B. Alternatives

Maintenant que vous savez écrire des conditions, vous allez pouvoir les utiliser. En effet, une alternative est une instruction qui permet de choisir quelles instructions seront exécutées après l'évaluation d'une condition.

Il existe plusieurs formes d'alternatives : les alternatives simples, les alternatives doubles et les alternatives multiples.

Alternatives simples

La forme la plus simple d'alternative est :

SI condition ALORS bloc d'instructions

Cela signifie que, après l'évaluation de la condition, si la valeur est `True`, le bloc d'instructions qui suit le **ALORS** est exécuté. En revanche, dans le cas où cette valeur est `False`, le bloc d'instruction est ignoré.

Dans les deux cas, les instructions situées juste après l'alternative seront exécutées.

En Python, l'alternative simple est composée d'une **instruction d'en-tête**, celle qui contient `if` suivi de la condition et qui se termine par `:`. Le bloc d'instructions à exécuter dans le cas d'une évaluation de la condition à `True` doit se trouver sur la ligne suivante et être indenté d'une tabulation vers la droite.

```
if condition :  
    bloc d'instructions si True
```

Indentation

Il est maintenant temps de reparler de la notion d'**indentation** rapidement abordée dans l'atelier 1. En effet, vous y avez vu que toutes les instructions d'un même bloc d'instructions doivent être au même niveau d'indentation. Ce sont donc les niveaux d'indentation qui définissent les blocs d'instructions.

« Il y a deux solutions pour indenter : utiliser **quatre espaces** ou un seul caractère **tabulation**, mais jamais un mélange des deux sous peine d'erreurs *IndentationError: unindent does not match any outer indentation level*. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents.

La tabulation est notamment la convention utilisée pour la librairie standard de Python [...]

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau*. Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. »¹

Selon le PEP8, il est recommandé de mettre 4 espaces. D'ailleurs, dans le Shell de Python, il est possible de définir le nombre d'espaces par lequel une tabulation est remplacée. Pour ce faire, allez dans le menu « Option > Configure IDLE », dans l'onglet « Fonts/Tabs », comme indiqué dans la Figure 2.

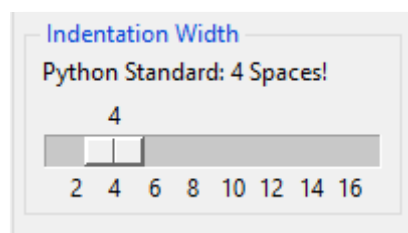


Figure 2 - Indentation

Les instructions qui suivent ont pour but de demander une cote à l'utilisateur et de voir s'il s'agit d'un échec. Si c'est le cas, un message est affiché, sinon rien n'est affiché.

```
cote = input("Cote : ")  
if cote < 10 :  
    print("Aïe, c'est un échec !")
```

¹ Extrait de https://fr.wikibooks.org/wiki/Programmation_Python/Structure_d%27un_programme



Prenez le temps de recopier ces instructions dans un fichier source `cote.py`. Exécutez ce script.

Que constatez-vous ?

Selon les opérateurs que vous avez vus dans l'atelier 1, essayez de trouver une solution ! Si vous ne trouvez pas après avoir CHERCHÉ, la suite de l'atelier devrait vous aiguiller...



Assurez-vous d'avoir compris la notion de bloc d'instructions et donc d'indentation en analysant l'exemple suivant.

```
prix_total = nb_repas * prix_repas
if nb_repas > 10 :
    print("Réduction pour plus de 10 repas !")
prix_total *= 0.9
print("À payer : ", prix_total, "euros")
```

La réduction est-elle appliquée à bon escient ?

Ajoutez les instructions permettant de demander les valeurs de `nb_repas` et `prix_repas` à l'utilisateur et écrivez le tout dans un fichier source.

Avez-vous bien tenu compte de tous les cas ?

Remarque

Tant que vous êtes dans l'aspect « test de programme », il semble important de préciser quelques éléments à ce sujet !

Des tests ne vont jamais remplacer une spécification précise. Si c'est vrai qu'on peut utiliser des tests pour expliciter ce qu'on attend du code, seuls ils ne suffiront jamais à décrire précisément ce que le code doit faire. Il pourra toujours rester un doute. Par exemple, supposez qu'on vous demande d'écrire les instructions qui sur base de deux entiers nb1 et nb2 garnissent la variable `résultat` de façon à correspondre aux cas de test suivants.

Cas testé	Résultat attendu
2 et 7	7
9 et 4	9
1 et -6	1

Les deux implémentations suivantes passent avec succès ces trois tests.

Implémentation #1 : donne le maximum parmi nb1 et nb2

```
if nb1 > nb2 :
    resultat = nb1
else :
    resultat = nb2
```

Implémentation #2 : donne nb1 s'il est impair, sinon nb2

```

if nb1 % 2 == 1 :
    resultat = nb1
else :
    resultat = nb2

```

Cet exemple montre bien que les tests seuls ne suffisent pas pour déterminer précisément ce qu'on veut que la fonction fasse. Seul un énoncé clair et précis permet de lever toute ambiguïté.

Ce n'est pas parce que tous les tests réussissent que le code est correct. Si les tests révèlent un problème, c'est qu'il y a un bug quelque part dans le code... Cependant, l'inverse n'est pas vrai : un programme qui passe avec succès tous les tests pourrait quand même avoir des bugs. Les tests peuvent révéler certains bugs, mais ils ne permettent pas de garantir que le code est correct !

Par exemple, on vous demande d'écrire les instructions qui déterminent la valeur absolue d'un entier x et la mettent dans `resultat`.

Un jeu de tests acceptables pourrait être le suivant (on teste un cas positif, un cas négatif, et on n'oublie pas le cas du zéro).

Cas testé	Résultat attendu
-1	1
1	1
0	0

Seulement, les trois implémentations suivantes passent avec succès tous ces tests... Or, seule l'une d'entre elles remplit bien le contrat de départ, à savoir déterminer la valeur absolue d'un entier !

Implémentation #1 : calcule $|x|$

```

if x < 0 :
    resultat = x * -1
else :
    resultat = x

```

Implémentation #2 : calcule x^2

```

résultat = x * x

```

Implémentation #3 : indique si x est non nul (0 pour faux, 1 pour vrai)

```

if x == 0 :
    resultat = 0
else :
    resultat = 1

```

La principale leçon à retenir est que, même si les tests peuvent être utiles et déceler des bugs, ils ne remplacent pas la réflexion qui doit avoir lieu lors de la conception du programme. Bien penser la structure du programme pour que tous les cas y soient traités correctement est la meilleure manière d'arriver à un code correct !

Alternatives doubles

La deuxième forme d'alternative est :

*SI condition **ALORS** bloc d'instructions **SINON** bloc d'instructions*

Cela signifie que, après l'évaluation de la condition, si la valeur est **True**, le bloc d'instructions qui suit le **ALORS** est exécuté. En revanche, dans le cas où cette valeur est **False**, c'est le bloc d'instructions qui suit le **SINON** qui est exécuté.

Dans les deux cas, les instructions situées juste après l'alternative seront exécutées.

En Python, l'alternative double est composée de deux parties :

- la première partie est introduite par l'**instruction d'en-tête**, celle qui commence par le mot réservé **if** suivi de la condition et qui se termine par **:**. Le bloc d'instructions à exécuter dans le cas d'une évaluation de la condition à **True** doit se trouver sur la ligne suivante et être indenté d'une tabulation vers la droite.
- la deuxième partie est introduite par le mot réservé **else** suivie de **:**. Le bloc d'instructions à exécuter dans le cas d'une évaluation de la condition à **False** doit se trouver sur la ligne suivante et être indenté d'une tabulation vers la droite.

```
if condition :  
    bloc d'instructions si True  
else :  
    bloc d'instructions si False
```

Les instructions qui suivent ont pour but de demander une cote à l'utilisateur et de vérifier qu'il s'agit d'une réussite. Si c'est le cas, un message précisant qu'il s'agit d'une réussite est affiché, sinon un message précisant qu'il s'agit d'un échec est affiché.

```
cote = int(input("Cote : "))  
if cote >= 10 :  
    print("Super ! Tu as réussi !")  
else :  
    print("Aïe, c'est un échec...")
```



Prenez le temps de modifier le fichier source **cote.py** afin d'y intégrer cette modification. Exécutez ce script.

Alternatives multiples



Reprenez le script précédent et sauvez sous un autre nom. Ajoutez les instructions permettant de préciser « Ouf ! De justesse... » lorsque la cote est de 10.

Une solution est d'utiliser des *alternatives imbriquées* :

```
cote = int(input("Cote : "))  
if cote > 10 :  
    print("Super ! Tu as réussi !")  
else :  
    if cote < 10 :  
        print("Aïe, c'est un échec...")  
    else :  
        print("Ouf ! De justesse...")
```


Le deuxième `if` correspond au bloc d'instructions à réaliser lorsque la première condition (`cote > 10`) est fausse... Faites bien attention à l'indentation !

Une autre solution est d'utiliser la notation `elif`, qui est la contraction de `else if` :

```
cote = int(input("Cote : "))
if cote > 10 :
    print("Super ! Tu as réussi !")
elif cote < 10 :
    print("Aïe, c'est un échec...")
else :
    print("Ouf ! De justesse...")
```

Il s'agit juste d'une autre façon d'écrire des alternatives imbriquées qui a l'avantage de faire en sorte que tous les cas traités soient au même niveau d'indentation.

C. Répétitives

Une répétitive est une instruction de contrôle de flux, au même titre que les alternatives. Elle permet de **répéter** un certain nombre de fois (voire indéfiniment) une séquence d'instructions. Dans le langage commun, vous entendrez également parler de « boucles ».

Python propose deux instructions particulières pour construire des répétitives : l'instruction `for ... in`, que vous découvrirez lors d'un atelier futur, et l'instruction `while` qui est présentée dans la suite de cet atelier.



Écrivez un script contenant les instructions ci-dessous et sauvez-le dans `affiche_i.py`.

```
i = 1
while i <= 10 :
    print(i)
    i += 1
print("Fin")
```

Mais i, ce n'est pas un bon nom de variable !?

Le nom de variable `i` est parfois critiqué à juste titre, mais ce nom se trouve être approprié quand il s'agit d'une boucle constituée de **seulement quelques instructions** et dans laquelle le `i` ne sert qu'à passer à l'itération suivante. En effet, dans ce cas, la portée de la variable est suffisamment limitée que pour ne pas induire de doute quant à sa signification. Pour information, le `i` vient des mathématiques, où la lettre `i` est utilisée comme **indice** du symbole de sommation Σ .

Exécutez-le ! Que se passe-t-il ? Avant de lire les explications qui suivent, prenez le temps de décrire ce qui se passe de la manière la plus détaillée possible en FRANÇAIS.

Le mot `while` signifie « tant que » en anglais. Cette instruction indique qu'il lui faut répéter le bloc d'instructions qui suit tant que la condition, ici `i <= 10`, est vraie. Dans ce cas, on recommence tant que le contenu de la variable `i` est inférieur ou égale à 10.

Comme l'instruction `if` vue précédemment, le mot `while` fait partie de l'**en-tête** de l'instruction. Cette ligne contient également la condition et se termine par `:`.

Ces instructions sont décalées d'une tabulation afin de bien visualiser les instructions qui sont à répéter. Pour rappel, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau.

L'instruction `while` provoque la réalisation des étapes suivantes :

- la condition est évaluée.
- si la valeur qui en résulte est `False`, alors le bloc d'instructions qui suit est ignoré et l'exécution du programme se poursuit à la première instruction qui suit le bloc en question ; dans cet exemple `print("Fin")`.
- si la valeur qui en résulte est `True`, alors le bloc d'instructions aussi appelé le **corps de la boucle**, est exécuté :
 - l'appel de la fonction `print()` pour afficher la valeur courante de la variable `i`.
 - l'instruction `i += 1` qui incrémente/augmente de 1 le contenu de la variable `i`.

L'exécution de ces deux instructions consiste en la réalisation d'une première **itération**. Ensuite le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction `while`. La condition est à nouveau évaluée, et ainsi de suite tant que l'évaluation de cette condition donne `True`.

On parle donc aussi de structure **itérative**.

Boucle qui ne commence/fini jamais

La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur). Si ce n'est pas le cas, la condition ne peut être évaluée !

Si la condition est évaluée à `False` dès le départ, le corps de la boucle n'est jamais exécuté. Cette boucle n'a donc pas de commencement.

Il faut donc veiller à ce que la condition puisse être évaluée à `True` au moins une fois, sans quoi la boucle n'a aucun sens !



Testez le script suivant pour bien visualiser ce type d'erreur :

```
i = 3
while i < 3 :
    print("Hello !")
    i += 1
```

Si la condition est toujours évaluée à `True`, le corps de la boucle est répété indéfiniment. C'est une boucle sans fin ou boucle infinie.

Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable (il pourrait y en avoir plusieurs) intervenant dans la condition, de manière que cette condition soit évaluée à `False` et la boucle se termine.



Le script suivant est une boucle sans fin ! Testez-le pour bien visualiser ce type d'erreur :

```
i = 3
while i < 5 :
    print("Hello !")
```

Comme vous l'avez vu dans le point C des Exercices 0, vous pouvez interrompre l'exécution d'un script via les touches CTRL + C.



Afin de vous assurer d'avoir bien compris le concept de répétitive, écrivez le script qui permet d'afficher la table de multiplication de 7 dans `table_de_7.py`.



Sauvez ce script dans `tables_de_1_a_10.py` et ajoutez les instructions permettant d'afficher les tables de multiplication de 1 à 10.

D. Pourquoi des fonctions ?

Une autre façon de modifier le flux des instructions est de définir une fonction et d'y faire appel. Mais pourquoi définir une fonction ?

Un problème complexe nécessite souvent un algorithme qui l'est tout autant pour obtenir une solution. C'est pourquoi une stratégie de **modularité** est souvent mise en place lors de l'écriture d'un programme : les instructions sont rassemblées en différentes parties distinctes et indépendantes. Celle-ci fournit entre autres les avantages suivants : répétition, réutilisation, simplification... Ceux-ci sont développés par la suite.

Il existe de nombreux termes et définitions permettant de décrire les entités à la base de cette modularité : fonctions, procédures, routines... Ils traduisent tous le concept de **sous-programmes**, des séries d'instructions qui peuvent être exécutées à partir d'un programme parent.

Jusqu'à maintenant, vous avez eu l'occasion d'utiliser diverses fonctions telles que `id()`, `print()`, `input()`... Ces fonctions sont des fonctions natives, c'est-à-dire qu'elles sont fournies avec l'interpréteur de Python. D'autres fonctions sont également disponibles en important des modules déjà écrits par d'autres programmeurs.

Module



En Python, il s'agit d'un fichier qui peut être importé dans un autre fichier afin de profiter des fonctions qui y sont définies.

À quoi ça sert de découper un script en petits morceaux de code alors qu'on pourrait simplement tout écrire « en une fois » ?

La première façon de présenter ce concept est complètement indépendant de la programmation. La seconde, en revanche, y est directement liée.

Dans la vie réelle

Dans la gestion de l'organisation de votre journée, vous avez la possibilité de faire diverses activités :

- vous préparer le matin,
- aller promener votre chien,
- aller faire des courses,
- aller au cours,
- revoir vos cours,
- aller à la salle de sport,
- ...

Chacune de ces activités est elle-même constituée de tâches plus précises. Par exemple, pour l'activité vous préparer le matin, vous devez réaliser les tâches suivantes :

- vous lever,
- aller à la toilette,
- préparer vos vêtements,
- prendre votre douche,
- prendre votre petit déjeuner,
- vous laver les dents.

Chaque tâche peut encore une fois être divisée en actions. Par exemple, lorsque vous prenez votre douche, vous devez

- entrer dans la douche,
- ouvrir le robinet,
- vous laver le corps,
- vous laver les cheveux,
- vous rincer,
- sortir de la douche,
- vous essuyer.

Dans le cas où vous allez à la salle de sport, la dernière tâche que vous allez réaliser est probablement de prendre votre douche. Il s'agit de la même tâche que celle réalisée lorsque vous vous préparez le matin.

Au lieu de faire la liste des actions associées au fait de prendre votre douche, vous dites juste « je vais prendre ma douche ». Il semble donc logique de regrouper les actions liées à une tâche sous un même nom afin d'y faire référence au moyen de ce nom, plutôt que de lister toutes les actions à chaque fois.

De plus, pensez au cas où des actions sont ajoutées à la tâche prendre sa douche, comme par exemple prendre le savon et/ou prendre le champoing. Il faut ajouter ces actions à chaque fois qu'on rencontre la tâche prendre sa douche... et si on ne les modifie pas toutes ?

En programmation

Comme dans l'exemple ci-dessus, un script, surtout s'il est long, réalise souvent des traitements similaires, voire identiques, à plusieurs moments de son exécution. Par exemple, la validation d'une adresse mail, le calcul d'un masque de sous-réseau, l'affichage d'un titre... pourraient être répétés à des moments différents d'un même script.

Dans le cas d'un script qui présente les résultats de l'analyse du trafic sur un réseau, plusieurs informations sont affichées et sont regroupées sous divers titres. Par exemple, les informations liées à l'analyse des paquets du protocole IP sont regroupées sous le titre « IP ». Afin de faciliter la lecture des résultats et de retrouver les informations recherchées d'un coup d'œil, vous devez écrire les titres comme montré ci-dessous.

```
#####  
### TITRE  
#####
```

Un bout de code qui permet de faire cela est

```
print("#####")
print("### IP")
print("#####")
```

Dans le cas où il faut afficher plusieurs titres, la manière la plus « évidente », mais aussi la moins habile, de le faire, est bien entendu de répéter le code autant de fois que nécessaire, c'est-à-dire en recopiant presque mot à mot les lignes de codes.

```
print("#####")
print("### IP")
print("#####")
#...
print("#####")
print("### TCP")
print("#####")
```

La rédaction d'un tel script paraît simple puisque ce ne sont que des copier/coller avec quelques modifications. Cependant, le script est inutilement long et il contient des répétitions.

En procédant de cette manière, vous prenez des risques, aussi bien en termes de **lisibilité** qu'en terme de **maintenabilité**.

En effet, toute modification du code répété, par exemple, remplacer les # par des * impose la recherche des différentes apparitions de ce code.

Dans ce cas-ci il ne s'agit que d'un affichage de titre, mais qu'en est-il si le bout de code répété contient des instructions influençant le résultat du script ? En cas d'oubli d'une modification, le résultat du script peut devenir incertain.

Or, le fait d'être lisible et facilement modifiable est un critère essentiel pour l'élaboration d'un programme informatique, et donc d'un script en Python. Cela devient même une nécessité incontournable lorsqu'on travaille en équipe et que le programme doit être mis à jour par des personnes qui n'en sont pas les concepteurs.

Il faut donc opter pour une autre stratégie, qui consiste à

- séparer ces instructions du corps du script,
- leur donner un nom (ici `afficher_titre`),
- faire appeler à ces instructions (qui ne figurent donc plus qu'en un seul exemplaire) à chaque fois qu'on en a besoin via le nom qu'on leur a donné.

```
def affiche_titre() :
    print("#####")
    print("###", titre)
    print("#####")
titre = "IP"
affiche_titre()
#...
titre = "TCP"
affiche_titre()
```



Créez un nouveau fichier source, nommé `titres.py` et copiez-y le code source proposé ci-dessous. Exécutez-le afin de voir le résultat.

Grâce à cette façon de faire, la lisibilité est assurée et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité du script. Il s'agit du concept de **point de modification unique**.

Vous allez ainsi apprendre à décomposer votre programme en une série de **sous-programmes**, appelés fonctions. Chaque fonction doit correspondre à une tâche algorithmique simple/élémentaire/**unitaire**.

Quelques questions à se poser afin de s'assurer que c'est le cas :

- Le sous-programme permet-il de réaliser **une seule tâche** ?
- Puis-je lui associer un nom **court** et **pertinent** ?
- Puis-je décrire cette tâche en une **phrase simple** ?
- Est-ce que sa définition favorise la **réutilisabilité** ?
- Reste-il, dans le code, des tâches qui pourraient être extraites sous la forme de sous-programmes (soit parce qu'elles sont répétées et contreviennent au concept de **point de modification unique**, soit parce qu'il s'agit d'une tâche dite **unitaire**) ?

Attention à ne pas exagérer dans la découpe en fonctions au risque d'obtenir un code illisible et des fonctions d'une ligne ou deux qui ne sont appelées qu'une seule fois... Le script qui en résulte doit être plus lisible, plus facilement modifiable et permettre d'avoir une meilleure vue d'ensemble sur le script.

Avantages des fonctions

Comme dit dans l'introduction et expliqué ci-dessus, il y a plusieurs avantages à découper le code en fonctions. En voici un résumé...

Si au cours d'un algorithme, une série d'instructions se répète, celle-ci peut être isolée sous forme de fonction et exécutée dès que c'est nécessaire. En plus d'**éviter les répétitions**, si le code venait à changer, on ne devrait le modifier qu'à un seul endroit.

Une série d'instruction qui forme une tâche à part entière et potentiellement courante peut être **réutilisée** dans une autre partie du programme, voire dans un autre programme. On évite donc d'écrire plusieurs fois le même code.

Écrire un ensemble de séries d'instructions, chacune résolvant un petit problème simple est plus facile que d'écrire une longue série d'instructions résolvant un problème complexe. C'est ce qu'on appelle la division en sous-problèmes. Plutôt que de tout de suite se lancer dans la résolution du problème général, il est souvent plus facile et plus efficace de le scinder en plusieurs **problèmes plus petits et plus faciles à résoudre**.

Dans cette optique, une règle générale en programmation est de faire une fonction de toute portion de code qui est répétée au moins 3 fois dans le script/programme.

E. Fonction

Une fonction est une séquence d'instructions pouvant être exécutée indépendamment. La fonction, et donc la série d'instructions qui la composent, peut être exécutée à n'importe quel moment dans un algorithme comme s'il s'agissait d'une seule instruction.

L'utilisation des fonctions nécessite deux étapes :

- la définition de la fonction, et
- l'appel de cette fonction.

La **définition d'une fonction** commence par le mot clé `def` (un raccourci de « définition ») suivi du **nom** de la fonction et de `()`, et se termine par `:`.

```
def nom_de_la_fonction () :  
    corps_de_la_fonction
```



Dans un nouveau fichier source, définissez une fonction qui affiche le message "Bonjour le monde !" dans la console du Shell. Donnez un nom adéquat à la fonction ! Exécutez le script. Que se passe-t-il ?

À votre avis, pourquoi ?

Que faut-il faire pour résoudre ce souci ?

La réponse est la suivante : lorsque la fonction est définie, et pour que son code soit exécuté, il faut faire un **appel** à la fonction. Dans ce cas, son appel se limite à citer le nom de la fonction suivi de `()`.

```
nom_de_la_fonction ()
```

Dans ce cas, l'appel doit se trouver dans la partie principale du script, c'est-à-dire en dehors de toute définition de fonctions.



Écrivez cet appel en première ligne du fichier source contenant la fonction. Que se passe-t-il ? À votre avis, pourquoi ? Que faut-il faire pour régler ce problème ?

Voici une piste de réponse : Python interprète les instructions les unes à la suite des autres.

Voici maintenant l'explication de ce qui s'est produit : lorsque vous avez mis l'appel de la fonction avant sa définition, comme il n'avait pas encore pris connaissance de la définition de la fonction, le Shell n'a pas pu l'exécuter... Si vous placez ce même appel après la définition de la fonction, alors le Shell connaît la fonction au moment de l'appel et il peut exécuter le code qui lui est associé.



Maintenant que vous avez écrit votre première fonction et que vous savez y faire appel, organisez votre script comme proposé ci-dessous :

- En premières lignes, placez la définition de la fonction.
- Après cette définition et dans la partie principale du script, ajoutez l'instruction `print("Avant l'appel de la fonction.")`.
- Toujours dans la partie principale, faites appel à la fonction.
- Et enfin, ajoutez l'instruction `print("Après l'appel de la fonction.")`.

Exécutez le script.

Lors de l'exécution de ce script, vous remarquez que l'ordre d'exécution des instructions ne suit pas l'ordre dans lequel elles sont écrites dans le fichier source. En effet, lors de l'appel de la fonction, c'est l'instruction qui se trouve dans la définition de la fonction qui est exécutée alors qu'elle se trouve avant l'appel en question (dans ce cas, il n'y en a qu'une, mais on peut bien sûr y mettre une séquence d'instruction). Lorsque la fonction est finie, que toutes les instructions qu'elle contient ont été exécutées, le script reprend à l'instruction qui suit cet appel.

Voici quelques précisions, en commentaire, sur le code proposé pour l'affichage des titres.

```
# définition de la fonction
def affiche_titre() :
    print("#####")
    print("###", titre)
    print("#####")
titre = "IP"
affiche_titre() # appel de la fonction
#...
titre = "TCP"
affiche_titre() # appel de la fonction
```

Fonction ou procédure ?

En algorithmique, on distingue la notion de fonction de celle de procédure. Une **fonction** joue le rôle d'une **expression** évaluable alors qu'une **procédure** joue le rôle d'une **instruction**.

Python ne fait pas la distinction, à strictement parler, entre le concept de procédure et celui de fonction. Cependant, le programmeur peut réaliser une procédure à l'aide d'une fonction qui ne renvoie aucune valeur.

Dans cet atelier et dans les exercices qui y sont associés, vous ne définirez que des procédures. Une des façons de distinguer une procédure est le nom qu'elle porte : on utilise un **verbe** qui décrit l'action accomplie.

Dans l'exemple ci-dessus, la « fonction » définie est une procédure puisqu'elle ne fait qu'afficher un titre. Son nom est donc `affiche_titre`.

F. Portée des variables

Une variable n'est pas nécessairement accessible partout dans le script. De plus, les variables ne sont pas toutes accessibles en même temps et pour la même durée. La partie du script dans laquelle la variable est accessible est appelée sa **portée**, et la durée pendant laquelle la variable existe sa **durée de vie**.

Pour comprendre ce phénomène, il est utile de comprendre la notion d'**espace de noms**. Chaque entité (variable, fonction...) a un nom ou **identificateur**. En Python, toute variable est une référence vers un objet. Un identificateur est donc associé à la référence d'un objet. Un espace de noms est le « répertoire » qui associe à chaque identificateur (nom de variable, nom de fonction...) la référence vers la zone mémoire où est stocké l'objet.

En Python, il y a plusieurs niveaux d'espace de noms organisés selon une hiérarchie connue :

- **built-in** : il reprend les noms générés au lancement de l'interpréteur. C'est la raison pour laquelle les fonctions natives sont accessibles n'importe où dans un script.
- **global** : il reprend les noms générés au niveau de chaque module. Ces espaces de noms sont isolés les uns des autres. De ce fait, si un même identificateur est utilisé dans deux modules distincts, ils n'interfèrent pas...
- **local** : il reprend les noms générés au niveau de chaque fonction. À chaque exécution d'une fonction, c'est-à-dire à chaque appel, un espace de noms lui est associé.

C'est au moment de son assignation qu'une variable est liée à un espace de noms particulier. Sa portée et sa durée de vie dépendent donc de l'endroit où la variable est définie.



Lisez le bout de code suivant et essayez de déterminer ce qui sera affiché.

```
# Exemple_1
def affiche_a() :
    a = 10
    print("Dans la fonction :", a)

affiche_a()
print("Hors de la fonction :", a)
```

Exécutez le code de l'`exemple_1`, après l'avoir copié dans un fichier source. Que se passe-t-il ? À votre avis, pourquoi ?



Lisez le bout de code suivant et essayez de déterminer ce qui sera affiché.

```
# Exemple_2
a = 10

def affiche_a() :
    print("Dans la fonction :", a)

affiche_a()
print("Hors de la fonction :", a)
```

Exécutez le code de l'`exemple_2`. Que se passe-t-il ?

Pouvez-vous expliquer ce qui diffère entre l'`exemple_1` et l'`exemple_2` ?



Maintenant que vous avez compris, déterminez ce que va produire ce bout de code.

```
# Exemple_3
a = 10

def affiche_a() :
    a = 20
    print("Dans la fonction :", a)

affiche_a()
print("Hors de la fonction :", a)
```

Exécutez le code de l'`exemple_3`. Que se passe-t-il ? Que pouvez-vous en conclure ?

Local et global

Lorsqu'une variable est définie à l'intérieur du corps d'une fonction, elle est accessible à partir de sa définition et jusqu'à la fin de la fonction. Elle existe aussi longtemps que la fonction est en cours d'exécution, l'espace de nom associé à la fonction étant libéré dès la fin de celle-ci.

Les variables qui sont définies dans une fonction sont appelées **variables locales** à la fonction. Ainsi, les variables locales à une fonction ne peuvent pas être utilisées en dehors de cette fonction. De plus, les variables définies au sein d'une fonction n'entrent pas en conflit avec celles définies en dehors, même si elles portent le même nom.

Lorsque vous écrivez des instructions en dehors de toute fonction, vous écrivez dans le bloc d'instructions principal. Les variables qui sont définies à ce niveau sont appelées **variables globales**. Dans ce cas, la variable est accessible partout dans le script, et aussi à l'intérieur de tout script qui l'importe.

Il existe encore une autre portée, appelée **nonlocal**, mais elle est liée à la définition de fonctions au sein d'une fonction, ce qui ne sera pas abordé dans ce cours introductif.



Pour conclure cette notion de portée de variable, déterminez ce que produit ce code.

```
# Exemple_4
a = 10

def affiche_a() :
    print("Dans la fonction :", a)
    a = 20

affiche_a()
```

Exécutez le code de l'**exemple_4**. Que se passe-t-il ? À votre avis, pourquoi ?

Piste de réponse : toute assignation de variable dans une fonction en fait une variable locale de la fonction !



Enfin, exécutez ce dernier exemple.

```
# Exemple_5
a = 10

def affiche_a() :
    global a
    a = 20
    print("Dans la fonction :", a)

affiche_a()
print("Hors de la fonction :", a)
```

Ainsi, pour que la fonction puisse modifier la variable globale **a**, il faut ajouter le mot-clé **global**. Il signale que la variable **a** à utiliser dans la fonction est la variable globale et non une variable locale à créer. Pour information, il est préférable dans tous les cas de minimiser le nombre de variables globales et par conséquent l'utilisation du mot-clé **global**. Vous verrez dans la suite du cours comment procéder pour l'éviter.