

# 420-1G2-HU Logique de programmation

# Bienvenue dans le cours de logique de programmation!

# Bases de la logique de programmation

# Concepts de la programmation

## Où sont les programmes?

- Communications : Internet, courriel, téléphonie mobile.
- Traitement de texte : édition et impression des documents.
- Traitement de données géographiques et statistiques.
- Collecte de données et télédétection.
- Applications de gestion.
- Systèmes d'Information Géographiques.
- Jeux et multimédia.
- Robotique.
- Autres applications.

## Qu'est-ce qu'un programme informatique ?

Il s'agit d'une suite d'instructions précises et ordonnées qui peuvent être exécutées (interprétées) par un ordinateur pour résoudre un problème.

## Quel est le rôle du programmeur ?

La tâche du programmeur consiste essentiellement à rechercher et à écrire une séquence correcte d'instructions pour chaque énoncé qui lui est soumis.

- Il construit les étapes à suivre qui permettent de solutionner un problème;
- Il vérifie la logique de la solution du problème; et
- Il traduit sa logique en une série d'instructions interprétables par l'ordinateur.

## Qu'est-ce que la programmation ?

### ✓ Description vulgarisée

La programmation, c'est de donner des instructions à un ordinateur pour qu'il fasse ce que tu veux. Imagine que tu donnes des étapes précises à un ami pour construire une tour avec des blocs. De même pour le GPS, il te donne l'itinéraire détaillé pour que tu rendes à ta destination. Tu dois être très clair et détaillé pour qu'il suive tes instructions correctement. De la même manière, en programmation, tu écris des lignes de code pour dire à l'ordinateur quoi faire étape par étape. Une fois que tu as écrit le bon code, l'ordinateur exécute ces instructions et accomplit la tâche que tu lui as demandée, que ce soit pour créer un jeu, un site web, ou même contrôler un robot.

Le but de la programmation est de résoudre un problème. Ainsi, la programmation est l'art de commander à un ordinateur de faire exactement ce que l'on veut qu'il fasse en lui fournissant un ensemble d'instructions détaillées, précises et ordonnées servant à résoudre un problème donné.

Le travail d'un ordinateur, quant à lui, consiste essentiellement à *traiter des données* qui lui sont fournies, en suivant les étapes d'un programme préenregistré en mémoire. En fait, un ordinateur n'est rien d'autre qu'une machine qui effectue des opérations simples.



*L'ordinateur exécute le travail qu'on lui fournit et non pas ce qu'on pense lui avoir fourni.*

Les principaux types de traitement dont un ordinateur est capable sont:

- La lecture (entrée) de données
- L'écriture (sortie) de données (ou résultats)
- La gestion de données en mémoire
- Le calcul et la comparaison logique

Quoique les types de traitement de base dont un ordinateur est capable soient peu nombreux, la *programmation est un art difficile*. Les difficultés proviennent principalement de l'ensemble des détails d'une activité complexe dont il faut tenir compte, de les ordonner adéquatement et surtout, de ne pas en oublier.

Il ne doit exister aucune ambiguïté dans la nature et la séquence des instructions du programme, car l'ordinateur n'est qu'une machine peu futée à qui l'on doit fournir à l'avance les moindres détails des tâches que l'on souhaite lui faire exécuter. Le moindre oubli et c'est la panne ou l'erreur.

L'ensemble de ces instructions est appelé un **algorithme**. Un mode d'emploi, une procédure, un itinéraire routier ou encore une recette culinaire, sont des exemples d'algorithme.

## Qu'est-ce qu'un bon programme ?

- **Correct** : S'assurer qu'il fait ce qu'il doit faire.
  - **Robustesse** : S'assurer qu'il s'exécute sans erreur et donne toujours les bons résultats.
  - **Interface facile à utiliser** : S'assurer que l'interface est simple à utiliser.
  - **Simplicité** : Garder le code le plus simple possible et facilement modifiable en fonction de nouvelles exigences.
  - **Présentation et Documentation** : S'assurer que le code est bien documenté.
  - **Efficacité** : S'assurer que l'utilisation du temps et de la mémoire soit efficace.
- 

## Qu'est-ce que Python?

C'est un langage de programmation qui permet aux gens de donner des instructions aux ordinateurs de manière facile et compréhensible. Il s'agit de parler une nouvelle langue avec l'ordinateur pour lui dire quoi faire.

En utilisant Python, tu écris des lignes de code qui ressemblent un peu à des phrases spéciales que l'ordinateur comprend. Ces lignes de code disent à

l'ordinateur comment effectuer certaines actions, comme réaliser des calculs, gérer des données, créer des jeux, etc.

Python utilise des mots et des règles qui sont assez proches du langage humain, ce qui le rend facile à apprendre et à utiliser.

---

## Qu'est-ce que PyCharm ?

PyCharm est un IDE.

Un IDE (Integrated Development Environment) est un environnement de développement intégré. Il s'agit d'une boîte à outils spéciale pour les programmeurs qui leur permet de construire des programmes informatiques. C'est le même principe que pour construire une maison, on utilise des outils comme des marteaux, des clous et des scies.

Une fois l'IDE (PyCharm) ouvert, il donne un espace où tu peux écrire ton code, l'exécuter et te montrer les résultats. Il fournit également des raccourcis et des astuces pour éviter les erreurs et rendre la programmation efficace.

# Les opérateurs

Les opérateurs arithmétiques, logiques et de comparaison dans le langage Python

## Les opérateurs arithmétiques (mathématiques)

Les symboles des opérateurs sont les suivants :

Mathématiques	Syntaxe Python	Description	Exemple
+	+	Addition	5 + 4
-	-	Soustraction	5 - 4
x	*	Multiplication	2 * 8
÷	/	Division	8 / 2
exp	**	Exposant	2 ** 3
mod ou modulo	%	modulo (Reste de la division)	9 % 2
( )	( )	Séparateurs	(6 - 4) * 2

## Ordre de priorité des opérateurs arithmétiques en ordre décroissant

1. ( )
2. \*\*
3. \* et /
4. + et -

# Les opérateurs de comparaison

Les opérateurs de comparaison servent à comparer deux éléments. Comme pour les opérateurs arithmétiques, ils sont représentés par des symboles différents selon le langage de programmation utilisé :

Syntaxe en Python	Description	Exemple	Résultat
==	égal à	5 == 4	False
		8 == 8	True
!=	Différent de	5 != 4	True
		8 != 8	False
>	Supérieur à	5 > 4	True
		4 > 5	False
<	Inférieur à	5 < 4	False
		4 < 5	True
>=	supérieur ou égal à	5 >= 5	True
		4 >= 5	False
<=	inférieur ou égal à	5 <= 5	True
		4 <= 5	True

# Les opérateurs logiques

Les opérateurs de comparaison peuvent être combinés avec des opérateurs logiques qui sont les suivants:

Syntaxe Python	Description
and	Opérateur logique AND

and	ET logique
or	OU logique
not	Négation

Le résultat d'une comparaison par opération logique est représenté par deux états : Vrai ou Faux. L'ordinateur utilise le principe booléen. On représente tout ce qui est vrai par une valeur différente de zéro, généralement le 1.

### Exemple :

$3 > 4$  est faux, donc égal à 0

$6 > 2$  est vrai, donc égal à 1

## Priorité des opérateurs de comparaison et /ou logique (ordre décroissant)

1. ( )
2. Opérateurs arithmétiques
3. Opérateurs de comparaison
4. AND, OR, NOT

## Tables de vérité : le AND (ET) et le OR (OU)

### Le AND (ET)

ET	VRAI	Faux
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

## Le OR (OU)

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

## Combinaison des opérateurs

On peut combiner les opérateurs mathématiques, de comparaison et logiques.

Avant de résoudre les opérateurs de comparaison et/ou logiques, on doit trouver un seul résultat à chaque bloc d'opérations mathématiques. On peut ainsi affirmer que les opérateurs mathématiques ont la priorité sur les opérateurs de comparaison et/ou logiques.

## Exemples

- $5 + 16 * 2 - 8 / 2$
- $6 * (3 + 1) / 4$
- $4 * (5 + 6^{**}2)$
- $3^{**}(6 / 2 + 2)$
- $7 + (10 - (6 * (6 - 3) + (6 + 2 * 6)))$
- $(3 + 2 * (5 - 6)) * (3 + 5 - (5 + 9)) * 2$

# Exercices

## Exercice 1: Expressions arithmétiques et logiques

### But

- Révision sur les opérations arithmétiques, logiques et de comparaison.
- Manipulation des opérateurs avec la syntaxe Python.
- Comprendre comment les opérations sont interprétées en Python.
- Utilisation de la console Python dans l'IDE PyCharm.

### Énoncé

Pour chaque exemple :

- Trouvez d'abord (manuellement) la valeur ou l'état des expressions.
- Remplacez les caractères mathématiques en Python.
- Vérifiez vos résultat dans la console Python dans PyCharm

### Que valent ces expressions ?

- $6 * (2 + 3) - (3 * 23 + 6 * 23 + 8 * 22 - 4 * 2) + 36 * 25 \div 3 * 23 * 2 + 9$
- $8 * (4 * 3 + 12 - 5 * 3) + 8 * 33 \div 2 * 32$
- $(2 * 5 + 3) * (2 * 5 + 1) + ((5 + 7) * (2 * 5 + 1)) \div 2$

### Quel est l'état des expressions suivantes ?

- $3 > 19$  ET  $5 < 4$  OU  $18 < 15$  ET  $7 > 4$
- $3 > 19$  ET  $(5 < 4$  OU  $18 < 15$  ET  $7 > 4)$
- $(3 > 19$  ET  $5 < 4$  OU  $18 < 15)$  ET  $7 > 4$

- $3 > 19 \text{ OU } (13 > 2 \text{ ET } (5 < 4 \text{ OU } 18 < 15 \text{ ET } 7 > 4))$

**Quel est l'état de ces expressions si A vaut 2 et B vaut 9 ?**

- $A + (4 + 9 * 3) - 27 > 2 ^ 3 / 4$
- $3 + B > 19 - A \text{ OU } 5 < 4 \text{ ET } 18 < 15 \text{ OU } 7 > 4$

**Que vaut l'expression suivante si A vaut 4 et B vaut 8 ?**

$$A > 2 * (B - 6)$$

- Pour quelles valeurs de A cette expression changerait d'état ?
- Pour quelles valeurs de B cette expression changerait d'état ?

**Pour quelles valeurs de A l'expression suivante changerait d'état ?**

- $3 + B > 19 - A \text{ OU } 5 < 4 \text{ ET } 18 < 15 \text{ OU } 7 > 4$

# Les commentaires

## Les commentaires de base

Les commentaires sont des notes de texte ajoutées au programme pour fournir des informations explicatives **pertinentes** sur le code source. Ils ne sont pas exécutés, ils sont ignorés pendant la compilation ou l'interprétation du code.

Syntaxe Python : `# Ceci est un commentaire`

**Exemples :**

```
# Ceci est un commentaire expliquant ce que fait la prochaine ligne de code
total = sous_total + tax # Ici, nous ajoutons la taxe au sous-total
```

- ⓘ La compilation (ou l'interprétation) est une étape effectuée par les langages de programmation avant l'exécution du programme.

## Les commentaires multi-lignes

Ces commentaires peuvent couvrir plusieurs lignes. Ils sont entourés par trois guillemets doubles `"""` ou trois apostrophes `'''`. Les commentaires multi-lignes sont couramment utilisés pour documenter des fonctions, des classes et des modules (que vous verrez dans d'autres cours) en fournissant des descriptions détaillées.

**Exemple :** documenter un fichier de code source en python

```
"""
Le code source de ce fichier permet de :
- Lire les notes des 3 examens des étudiants
- Calculer la moyenne
- Afficher la note finale aux étudiants
"""

```

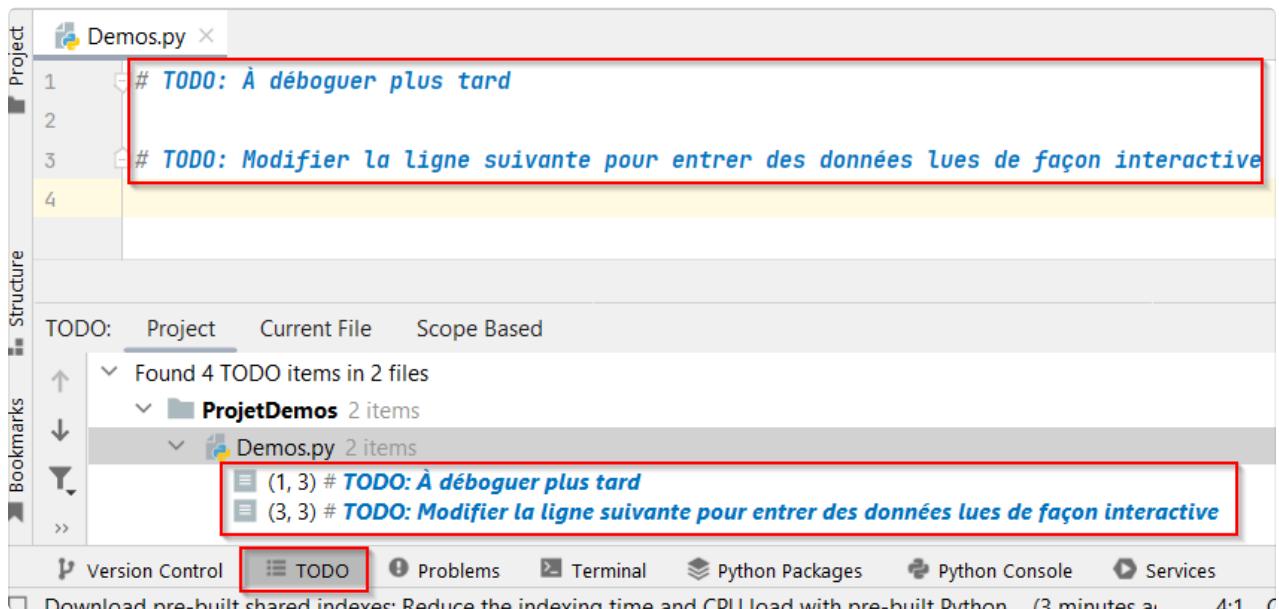
## Les commentaires TODO

Vous pouvez avoir besoin d'écrire des commentaires que vous pourrez retrouver facilement plus tard afin de corriger ou finaliser votre code source.

```
# TODO: À déboguer plus tard
```

```
# TODO: Modifier la ligne suivante pour entrer des données lues de façon
```

Allez dans PyCharm, et cliquez sur l'onglet TODO en bas de la fenêtre



## Les bonnes pratiques lors de la rédaction des commentaires

- Prioriser la clarté du code avant la rédaction de commentaires.

- **Prioriser la création de fonctions si vous devez commenter un bloc de code pour expliquer ce qu'il fait.**
- **Éviter les commentaires évidents. Exemples :**
  - `i = i + 1 # Incrémente i.`
  - `age = input("Veuillez entrer votre age : ") # lire l'âge de la personne.`
- **Exemples de cas où un commentaire serait pertinent**
  - clarifier les parties complexes qui ne sont pas évidentes à comprendre et qu'il n'y a aucun moyen de le mettre en évidence via la clarté du code.
  - justifier un choix non intuitif dans le code.
- **Ne pas oublier de mettre à jour les commentaires si le code source concerné change.**

# Les données

# Les données et types simples

## Introduction aux données

Les noms et adresses de clients, les positions géographiques, les données climatiques (températures, vitesse du vent, quantité de pluie/neige, ), les notes d'un examen, les valeurs des ventes d'un commerce, le numéro d'inventaire d'un produit, sa description, la réponse à une question du type "vrai ou faux?", un taux d'imposition, etc., sont différents exemples de données qui peuvent être traitées dans un programme.

## Les types de données simples

Les différents types de données correspondent à des représentations *internes* différentes, c'est à dire que la façon dont elles sont enregistrées en mémoire varie d'un type à l'autre.

**Exemple:** La chaîne "12345", l'entier 12345 et le réel 12345.0 ont des représentations internes différentes.

Seuls les types de données compatibles peuvent être utilisés dans un même traitement de données.

**Exemple :** il est impossible d'additionner la chaîne "12345" à l'entier 12345

Type de variable	En python	Exemple de valeur
Chaîne de caractère, texte (string)	str	"ceci est une chaîne de caractères"
Nombre entier (integer)	int	45
Nombre réel (float)	float	3.14
Booléen: vrai ou faux (boolean)	bool	True/False

Pour voir le type de la valeur faire : ***type(valeur)***

**Exemples:**

```
type(3.14)
```

```
type("chaine de caractères")
```

## Quelques chaînes de caractères spéciales

Caractère	Description	Exemple	Résultat
" "	Chaîne de caractères vide		
" " "	Caractère espace		
"\n"	Retour à la ligne	"Salut, \nComment ça va"	Salut, Comment ça va?
"\t"	Tabulation	"Salut, \tComment ça va"	"Salut, Comment ça va"

**Exemple avec "\n":**

```
adresse = "Nom : John Smith\nAge : 25\nAdresse : 123 Rue Principale\nVille : Paris"
print(adresse)
```

# Les variables, les constantes et les opérations de base

## Les variables

C'est un identificateur de donnée ou conteneur utilisé pour stocker et conserver de l'information variable.

Un nom de variable peut contenir des lettres et des chiffres ou des soulignés (underscore), mais ne peuvent pas débuter par un chiffre.

### Exemples :

- date\_naissance
- nom\_etudiant
- total\_taxes
- salaire\_brut

## Convention de nommage des variables

 Selon PEP8 (Le guide officiel des conventions de style en Python), les variables (non-constantes) doivent respecter les conditions suivantes :

- **Utilisez des minuscules et des underscores pour les noms de variables (snake\_case):** *identifiant\_etudiant, score\_total\_jeu1*
- **Il est important d'avoir des noms de variables significatifs:** *montant\_total*
- **Évitez les noms de variables d'une seule lettre, à moins qu'ils ne soient très explicites:** *x = 5*
- **Évitez les noms de variables qui sont des mots-clés Python:** *mon\_int*

# Les constantes

Identificateur de donnée de valeur invariable ou conteneur utilisé pour stocker et conserver de l'information qui ne variera pas tout au long de l'exécution du programme.

## Convention de nommage des constantes

- ❶ Semblable aux conventions de nommage des variables, sauf que, par convention,
  - **les mots doivent être en majuscule:** VALEUR\_MAX, TAXE\_PROV, TAUX\_HORAIRE, PI

# Un peu de vocabulaire

## Assignation (ou affectation)

Opération qui assigne une valeur à une variable. En langage Python, l'opérateur de l'assignation est `=`.

La syntaxe de l'affectation en langage Python est :

```
nom_variable = valeur
```

### Exemples :

Dans l'exemple suivant, la variable `code_cours` va contenir la chaîne de caractères "420-405-HU"

```
code_cours = "420-405-HU"
```

Dans l'exemple suivant, la constante `PI` va contenir le float (nombre réel) `3.14`

```
PI = 3.14
```

Dans l'exemple suivant, la variable `rayon` va contenir le int (nombre entier) `20`

```
rayon = 20
```

 **Remarque :** Lors de l'opération d'affectation, c'est d'abord le côté droit qui est évalué en premier, et son résultat sera assigné à la variable placée du côté gauche de l'affectation.

**Exemples :**

```
somme = 2 + 3
```

```
circonference_du_cercle = 2 * PI * rayon
```

## Incrémantation et décrémentation

- L'incrémantation est l'opération qui consiste à augmenter la valeur d'une variable existante de `x` (souvent de 1).
- La décrémentation est l'opération qui consiste à réduire la valeur d'une variable existante de `x` (souvent de 1).

**Exemples :** avec 2 syntaxes Python

- On incrémente de 1 la variable `nombre`

```
nombre = nombre + 1
```

Autre syntaxe Python:

```
nombre += 1
```

- On incrémente de 2 la variable *nombre*

```
nombre = nombre + 2
```

Autre syntaxe Python:

```
nombre += 2
```

- On incrémente de 3 la variable *nombre*

```
nombre = nombre - 3
```

Autre syntaxe Python:

```
nombre -= 3
```

# Les opérations d'écriture et de lecture

## Opération d'écriture

Opération par laquelle le programme transmet une valeur à l'externe, soit à l'écran, dans un fichier, dans une base de données, etc.

Syntaxe Python d'écriture (affichage) de données à l'écran :

```
print(texte_ou_valeur_a_afficher)
```

Exemples :

```
print(5)
print("Bienvenue dans le cours de logique de programmation")
```

## Opération de lecture

Cette opération est effectuée lorsque le programme doit recueillir de l'information. La source d'information peut être l'utilisateur , un fichier de configuration, une base de données, une information système comme la date, etc.

Syntaxe Python pour la lecture de données entrées par l'utilisateur de façon interactive:

```
valeur = input(chaine_de_caractères)
```

- Notez que ce que l'utilisateur va entrer comme valeur sera stocké dans la variable *valeur*.
- *La valeur lue via `input` est un `string`.*
- Si on a besoin que la valeur lue soit un nombre (`int` ou `float`) on procède à la conversion
  - d'un `string` vers un `int` à l'aide de la méthode `int(v_string)` ou

- d'un string vers un float à l'aide de la méthode `float(v_string)`  
avec `v_string` étant la valeur du string lue via `input()`.

**Exemple 1:** code qui demande à l'utilisateur d'entrer son nom, et qui ensuite écrit "Salut, " suivi du nom entré.

```
nom = input("Entrez votre nom:")
print("Salut, " + nom)
```

**Exemple 2 :** code qui demande à l'utilisateur d'entrer un nombre entier.

```
1 note_str = input("Veuillez entrer votre note sur 20 à l'examen :")
2 note = int(note_str)
3
4 note_pourcentage = note * 100 / 20
```

Il est possible de combiner les deux lignes 1 et 2 du code précédent en une seule :

```
note = int(input("Veuillez entrer votre note sur 20 à l'examen :"))
```

# Les opérations sur les chaînes de caractères

## Concaténation de chaînes de caractères

Concaténer deux chaînes de caractères revient à mettre bout à bout au moins deux chaînes de caractères.

**Exemple :**

```
phrase = "le code du cours est :" + "420-1G2-HU"
```

Le contenu de la variable phrase sera `" le code du cours est : 420-405-HU "`

Autre exemple : en utilisant des noms de variables dans la concaténation.

```
nom = "Dalicia"
nom_cours = "logique de prog"
num_groupe = 3
message = "Mon prénom est " + nom + ", j'enseigne " + nom_cours + " au gr
```



`str(num_cours)` sert à transformer le nombre entier dans la variable `num_cours` en chaîne de caractères.

## Les f-strings (Formatted String Literals)

Les f-strings sont préfixées par la lettre 'f' ou 'F', et les expressions Python entre accolades `{}` sont évaluées et insérées dans la chaîne.

```
nom = "Dalicia"
nom_cours = "logique de prog"
num_groupe = 3
message = f"Mon prénom est {nom}, j'enseigne {nom_cours} au groupe {num_g}
```

## La Méthode `str.format()`

Cette méthode vous permet d'insérer des variables dans une chaîne en utilisant des marqueurs de formatage (`{}`) et en appelant la méthode `format()` sur la chaîne.

```
nom = "Dalicia"
nom_cours = "logique de prog"
num_groupe = 3
message = "Mon prénom est {}, j'enseigne {} au groupe {}".format(nom, nom
```

# Laboratoire 1 - données

Les opérateurs et l'écriture à l'écran

## But du laboratoire

- Utiliser l'opération d'écriture à l'écran (`print`).
  - Utiliser les opérateurs arithmétiques, logiques et de comparaison.
  - Utiliser des variables.
  - Utiliser les **différentes méthodes de formatage** d'une chaîne de caractères.
- 

## Question 1

Durée : 1 minute

Afficher à l'écran la phrase "Hello World".

## Question 2

Durée : 5 minute

Afficher à l'écran un entête de programme contenant vos informations.

Résultat attendu :

```
*****  
** Nom: Bouallouche, Dalicia  
** No étudiant: 1234567  
** Date: 01-01-1970  
*****
```

## Question 3

Durée : 5 minute

Afficher à l'écran un entête de programme. Le programme ne fera qu'afficher l'entête et le menu.

Résultat attendu :

```
*****  
** Bienvenue dans mon application **  
*****  
<ligne vide>  
Veuillez entrer un choix:  
1. Hello World  
0. Quitter
```

## Question 4

Durée : 10 minute

Testez les opérateurs mathématiques et logiques et affichez le résultat.

Résultat attendu :

```
5 + 4 = 9  
5 - 4 = 1  
...  
5 // 4 = 1  
5 % 4 = 1  
5 < 4 = False  
5 <= 4 = False
```

Exemple :

Pour afficher `5 + 4 = 9`, le code sera : `print("5 + 4", 5 + 4)`

---

## Question 5

Durée : 10 minute

À l'aide des opérations mathématiques et logiques, répondre aux questions. Vous devez prouver votre réponse à l'aide de code.

- Est-ce que  $5*4$  est égal à  $2*10$  ?
- Est-ce que  $52*20+4$  est plus petit que  $130*8$  ?
- Est-ce que  $52*(20+4)$  est plus grand ou égal à  $12*100+4$  ?

Résultat attendu :

```
5 * 4 est-il égal à 2 * 10 ? True
52 * 20 + 4 est-il plus petit que 130 * 8 ? False
```

Exemple : exécutez le résultat suivant et analysez le.

```
print("Est-ce que 5 + 4 est égal à 8 + 1 ?", 5+4 == 8+1)
```

# Laboratoire 2 - données

Les opérateurs et l'écriture à l'écran

## But du laboratoire

- Utiliser des variables.
  - Écriture de commentaires.
  - Utiliser les opérateurs de lecture et d'écriture ainsi que les opérateurs de base.
  - Utiliser les **différentes méthodes de formatage** d'une chaîne de caractères.
- 

## Question 1

Demander le prénom d'une personne et afficher Bonjour <prénom>.

- Commencez d'abord par identifier la (les) variable(s) ainsi que leurs type(s).

Résultat attendu :

```
Quel est votre nom? Dalicia
Bonjour Dalicia!
```

---

## Question 2

Demander 2 nombres entiers à l'utilisateur et afficher toutes les opérations mathématiques et logiques ainsi que le résultat.

Résultat attendu :

Quel est le premier nombre ? 9

Quel est le second nombre ? 12

```
9 + 12 = 21
9 - 12 = -3
<...>
9 // 12 = 0
9 ** 12 = 282429536481
9 <= 12 = True
9 > 12 = False
```

## Question 3

Écrire un programme qui demande à l'utilisateur le plat principal, la boisson et le dessert et affiche un bon de commande à envoyer au chef. Le bon de commande doit être bien formaté.

- Commencez d'abord par identifier la (les) variable(s) ainsi que leurs types.

**Résultat attendu :**

Quel est le plat principal? Sandwich

Quelle est la boisson? Thé vert

Quel est le dessert? Crème brûlée

```
*****
**      BON DE COMMANDE      **
*****
** Plat principal: Sandwich
** Boisson:          Thé vert
** Dessert:         Crème brûlée
*****
```

## Question 4

Écrire un programme qui permet à l'utilisateur d'entrer une température en degrés Celsius et qui la convertit en une température en degrés Fahrenheit. Le programme doit afficher le résultat de façon professionnelle.

- Commencez d'abord par identifier la (les) variable(s) et les constantes ainsi que leurs types.

Formule de conversion :  $Fahrenheit = Celsius * 9/5 + 32$

---

## Question 5

Créer une application permettant de calculer le montant total d'une facture, incluant la TPS (5%) et la TVQ (9.975%).

- Commencez d'abord par identifier la (les) et les constantes variable(s) ainsi que leurs types.

Référence : [Calculateur de TPS et TVQ 2023](#)

### Calculs

TPS:  $<\text{montant}> * 0.05$

TVQ:  $<\text{montant}> * 0.09975$

TOTAL:  $<\text{montant}> + <\text{tps}> + <\text{tvq}>$

Résultat attendu :

```
Quel est le montant de la facture? 10
Montant Facture:    10.0
Montant TPS:        0.5
Montant TVQ:        0.9975
Montant Total:      11.4975
```

# Les collections de données

# Les listes

## Qu'est-ce qu'une liste?

Une liste est une collection utilisée pour stocker des données de tous types. Les éléments d'une liste sont ordonnés et modifiables (ou mutables).

## Syntaxe d'une liste

Une liste est créée en utilisant des crochets « [ ] » contenant des valeurs séparées par des virgules. Ces valeurs sont appelées ***les éléments de la liste*** et peuvent être de différents types : nombres entiers, chaînes de caractères, nombres flottants, booléens, listes, autres types complexes, etc.

**Exemples :** plusieurs listes avec différents types de données.

```
liste_nombres = [4, 8, 9, 1, 5]  
  
liste_mois = ["janvier", "Février", "Mars"]  
  
liste_bool = [True, True, False, True]  
  
liste_vide = []  
  
liste_mixte = [8, "a", "b", False, 0, "514", 9, True]
```

- ⓘ Les listes [1, 2, 3, 4] et [2, 4, 3, 1] sont différentes.

Dans la console PyCharm, comparez les deux listes en testant

```
[1, 2, 3, 4] == [2, 4, 3, 1]
```

et vérifiez le résultat qui devrait être `False`.

# Longueur d'une liste

La longueur ou la taille d'une liste est le nombre de ses éléments.

**Exemples :**

- La liste `[5, 7, 2]` est de longueur 3.
- La liste `[ ]` est de longueur 0.

En Python, `len` est la méthode qui nous donne la longueur de la liste *liste*.

## Syntaxe

```
len(liste)
```

**Exemple :**

```
longueur_liste = len([5, 7, 2])
print(longueur_liste)
```

# Accès aux éléments de la liste

Afin d'accéder aux éléments individuels de la liste, on utilise des indices (positions) mis entre crochets.

- Le premier élément de la liste est à l'indice 0.
- Le second élément de la liste est à l'indice 1.
- Le troisième élément de la liste est à l'indice 2.
- ...
- Le dernier élément de la liste est à l'indice  $n-1$ , avec  $n$  la longueur de la liste.

## Syntaxe

```
ma_liste[i] avec i l'indice de l'élément de la liste
```

### Exemples :

Pour la liste `liste_mois = ["janvier", "Février", "Mars"]`

- `liste_mois[0]` est: "janvier"
- `liste_mois[1]` est: " Février "
- `liste_mois[2]` est: "Mars"
- `liste_mois[3]` est: *Erreur (indice en dehors de la plage des indices valides)*

### Testez:

```
liste_mois = ["janvier", "Février", "Mars"]

element0 = liste_mois[0]
print(element0)

element1 = liste_mois[1]
print(element1)

print(liste_mois[2])

element_inexistant = liste_mois[3] # Lisez l'erreur
```

## Autre moyen de parcours d'une liste

Il est également possible de parcourir une liste à partir de la fin en utilisant des indices négatifs.

- L'élément à l'indice -1 étant le dernier élément de la liste.
- L'élément à l'indice -2 étant l'avant dernier élément de la liste.
- ...

- L'élément à l'indice  $-n$  (avec  $n$  la longueur de la liste) étant le premier élément de la liste

## Syntaxe

`ma_liste[-i]` avec  $i$  l'indice inversé de l'élément de la liste.

### Exemples :

Pour la liste `liste_mois = ["Janvier", "Février", "Mars"]`

- `liste_mois[-1]` est: `"Mars"`
- `liste_mois[-2]` est: `" Février"`
- `liste_mois[-3]` est: `" Janvier"`
- `liste_mois[-4]` est: *Erreur (indice en dehors de la plage des indices valides)*

### Testez :

```
liste_mois = ["janvier", "Février", "Mars"]

element2 = liste_mois[-1]
print(element2)

element1 = liste_mois[-2]
print(element1)

element0 = liste_mois[-3]
print(element0)

element_inexistant = liste_mois[-4] # Lisez l'erreur
```

## Accès à une succession d'éléments (sous-liste) d'une liste

On peut Accéder à plusieurs éléments successifs de la liste en même temps. Le résultat sera une sous-liste de la liste d'origine.

## Syntaxe

`liste[i:j]` retourne une liste contenant les éléments de la liste aux positions (indices)  $i$  jusqu'à  $j-1$ .  $i$  et  $j$  peuvent être négatifs ou vides.

Ainsi,

- `liste[2:7]` : sélectionne les éléments de la liste aux indices 2, 3, 4,..., jusqu'à 6.
- `liste[:5]` : sélectionne les éléments de la liste aux indices de 0 jusqu'à 4.
- `liste[2:]` : sélectionne les éléments de la liste aux indices de 2 jusqu'à la fin de la liste.

**Exemples :** sur la liste suivante:

```
liste_mois = ["janvier", "février", "mars", "avril", "mai", "juin", "juil
```

- `liste_mois[4:9]` donnera :  
['mai', 'juin', 'juillet', 'août', 'septembre']
- `liste_mois[:6]` et `liste_mois[0:6]` donneront :  
['janvier', 'février', 'mars', 'avril', 'mai', 'juin']
- `liste_mois[8:]` donnera :  
['septembre', 'octobre', 'novembre', 'décembre']
- `liste_mois[-11:-7]` donnera : ['février', 'mars', 'avril', 'mai']
- `liste_mois[-7:-11]` et `liste_mois[5:1]` donneront : [] car le premier indice est supérieur au 2ème.

Testez les exemples ci-dessus dans PyCharm.

# Les opérations sur les listes

## La modification d'un élément dans une liste

Pour modifier un élément dans une liste, on lui assigne une nouvelle valeur à via son indice.

### Syntaxe

```
liste[i] = 5
```

#### Exemple :

```
fruits_preferes = ["Pêche", "Kiwi", "Cerise"]
```

Pour modifier l'élément d'indice 1 qui est "kiwi" par "Ananas", on fait

```
fruits_preferes[1] = "Ananas"
```

## Ajouter un élément à la fin d'une liste

La méthode `append` ajoute un élément à la fin de la liste.

### Syntaxe

```
liste.append(element)
```

#### Exemple :

Ajout de l'élément "Banane" à la liste `fruits_preferes` qui contient

```
['Pêche', 'Ananas', 'Cerise']
```

```
fruits_preferes.append('Banane')
```

Résultat :

```
fruits_preferes aura les éléments ['Pêche', 'Ananas', 'Cerise', 'Banane']
```

## Étendre les éléments d'une liste

La méthode `extend` : concatène deux listes.

### Syntaxe

```
liste1.extend(liste2)
```

## Supprimer un élément d'une liste

### Suppression en utilisant l'indice

La méthode `pop` supprime un élément d'une liste à l'indice donné en paramètre.

### Syntaxe :

`liste1.pop(indice_element)` avec `indice_element` l'indice de l'élément à supprimer. Cette méthode retourne l'élément supprimé.

### Exemple 1 :

Suppression de l'élément d'indice 2 de la liste

```
fruits_preferes = ['Pêche', 'Ananas', 'Cerise', 'Banane']
```

```

fruits_preferes = ['Pêche', 'Ananas', 'Cerise', 'Banane']
fruit_supprime = fruits_preferes.pop(1)
print(fruits_preferes)
print(fruit_supprime)

```

**Résultat :**

`fruits_preferes` aura les éléments `['Pêche', 'Cerise', 'Banane']`.

`fruit_supprime` aura l'élément `'Ananas'`.

**Exemple 2 :**

Suppression de l'élément d'indice 2 puis de l'indice -3 de la liste `liste_nombres` suivante :

```

liste_nombres = [4, 8, 9, 1, 5]
nombre_supprime = liste_nombres.pop(2)
print(liste_nombres)
print(nombre_supprime)

nombre_supprime = liste_nombres.pop(-3)
print(liste_nombres)
print(nombre_supprime)

```

**Résultat :**

À la suppression à l'indice 2 :

- `liste_nombres` aura les éléments `[4, 8, 1, 5]`
- `nombre_supprime` aura l'élément `9`

À la suppression à l'indice -3 :

- `liste_nombres` aura les éléments `[4, 1, 5]`
- `nombre_supprime` aura l'élément `8`

## Suppression en utilisant l'élément lui même

La méthode `remove` supprime l'élément donné en paramètre à la liste. Elle ne retourne aucun élément.

**Syntaxe :**

```
liste.remove(element)
```

**Exemple :**

Suppression de l'élément 8 de la liste `liste_nombres`

```
liste_nombres = [4, 8, 9, 1, 5]
liste_nombres.remove(8)

print(liste_nombres)
```

**Résultat :**

`Liste_nombres` aura les éléments suivants `[4, 9, 1, 5]`

## Autres opérations sur les listes

- `liste.insert(i, e)` : ajoute l'élément `e` à la position (indice) `i` de la liste `liste`.
- `min(liste)` : trouve le plus petit élément de la liste
- `max(liste)` : trouve le plus grand élément de la liste.
- `sum(liste)` : fait la somme des éléments de la liste.
- `liste.sort()` : permet de trier la liste par ordre croissant.
- `e in liste` : retourne `True` si l'élément `e` appartient à la liste `liste`, ou retourne `False` sinon.

- [il existe d'autres méthodes/opérations sur les listes, je les ajouterai au fur et à mesure de nos besoins]

# Laboratoire listes - solution

Les listes

## But du laboratoire

- Manipuler les listes simples et à deux dimensions.
- Utilisation du débogueur pour voir le contenu des variables.

## Exercice 1 : manipulation de listes - échauffement

Créez une liste contenant les éléments suivants : [71, 55, 30, 42, 25, 68, 15].

À chacune des réponses aux questions suivantes, vous devez stocker le résultat dans une variable ayant un nom significatif. Vous devez également afficher le résultat de manière claire et professionnelle.

- Vérifiez si l'élément d'indice 2 de la liste est divisible par 5.
- Vérifiez si le dernier élément de la liste est pair.
- Vérifiez si l'élément 55 de la liste est impair.
- Incrémentez de 100 le premier élément de la liste (à la position 0).
- Échangez (permettez) les positions des valeurs 30 et 68 dans la liste.
- Triez la liste par ordre croissant.
- Inversez la liste. Voici deux façons d'inverser une liste, testez-les sur la console Python pour comprendre leur fonctionnement et choisissez celle qui vous convient :
  - `liste_inversee = liste[::-1]` → Cette méthode retourne la liste inversée.
  - `liste.reverse()` → Cette méthode inverse la liste elle-même.
- Utilisez la méthode `liste.index(element)` pour trouver l'indice de l'élément 68.

- Divisez la liste en deux parties pour créer deux listes distinctes.

## Exercice 2 : calcul moyenne

Écrire un programme en Python qui demande à l'utilisateur de saisir quatre notes d'examens d'un étudiant (en %). Ces notes doivent être stockées dans une liste.

- Calculez la moyenne des notes d'examen de la liste de deux façons : **avec** et **sans utiliser** les fonctions spéciales `sum` et `len` sur les listes.
- Le programme doit afficher le résultat en répondant par vrai ou faux à la question :

Est-ce que l'étudiant a une moyenne supérieure à 60% ? Vrai/Faux

▼ Solution

```
def saisir_notes():
    """
    Demande à l'utilisateur de saisir quatre notes d'examen et les stocke dans une liste.

    Returns:
        list: Une liste contenant les quatre notes d'examens en浮点数.

    """
    notes = []

    for i in range(4):
        try:
            note = float(input(f"Saisir la note {i + 1} (en %):"))
            if 0 <= note <= 100:
                notes.append(note)
            else:
                print("Erreur: La note doit être comprise entre 0 et 100.")
        except ValueError:
            print("Erreur: Veuillez entrer un nombre valide.")

    return notes


def calculer_moyenne_sum_len(notes: list):
    """
    Calcule la moyenne des notes en utilisant les fonctions sum et len.

    :param notes: Liste des notes d'examen.
    :return: La moyenne des notes.
    """

    return sum(notes) / len(notes)


def calculer_moyenne(notes):
    """
    Calcule la moyenne des notes sans utiliser les fonctions sum et len.

    :param notes: Liste des notes d'examen.
    :return: La moyenne des notes.
    """

    total = 0
    compteur = 0

    for note in notes:
        total += note
        compteur += 1

    return total / compteur
```

```
def est_superieure_a_60(moyenne):
    """
    Vérifie si la moyenne est supérieure à 60 %.
    :param moyenne: La moyenne des notes.
    :return: True si la moyenne est supérieure à 60, sinon False
    """

    return moyenne > 60

if __name__ == "__main__":
    notes_examen = saisir_notes()

    moyenne_avec_sum_len = calculer_moyenne_sum_len(notes_examen)
    print(f"Moyenne (avec sum et len): {moyenne_avec_sum_len:.2f}")

    moyenne_sans_sum_len = calculer_moyenne(notes_examen)
    print(f"Moyenne (sans sum ni len): {moyenne_sans_sum_len:.2f}")

    resultat = est_superieure_a_60(moyenne_avec_sum_len)
    print(f"Est-ce que l'étudiant a une moyenne supérieure à 60 ? {resultat}")
```

## Exercice 3 : Palindrome

**Définition:** Un palindrome est un mot, une phrase, un nombre ou une séquence qui se lit de la même manière de gauche à droite et de droite à gauche.

### Exemple :

"Un radar nu" se lit de la même manière de gauche à droite et de droite à gauche.  
C'est cette exemple qui sera utilisé dans votre exercice.

Dans cet exercice, nous allons faire quelques opérations sur une chaîne de caractères "Un radar nu." pour la transformer en "un radar nu" qui est un palindrome. Une vérification sera faite à la fin.

Soit la chaîne de caractères suivante :

```
palindrome = "Un radata na."
```

Ce qui nous intéresse sont les listes. Il est possible de transformer cette chaîne de caractères en une liste de lettres successives. Faites le comme ceci :

```
liste_palindrome = list(palindrome)
```

Appliquez les changements suivantes sur la liste liste\_palindrome:

- Remplacez le premier élément "U" par "u".
- Remplacez le 8ème élément de la liste par "r".
- Remplacez le 4ème "a" de la liste par "u".
- Supprimez le caractère ":".
- Supprimez le 3ème "a" de la liste.
- Supprimez tous les caractères espaces de la liste.
- Créez une nouvelle liste appelée `liste_palindrome_inversee` qui contiendra l'inverse de la liste `liste_palindrome`.
- Comparez les deux listes `liste_palindrome_inversee` et `liste_palindrome`. Si le résultat est vrai, alors vos opérations sur la liste sont correctes et vous avez vérifié que votre phrase est un palindrome. Affichez le résultat de la comparaison de façon claire.

## Exercice 4 : générer des adresses courriels

- ✓ Avant d'écrire votre programme, pensez à la solution de votre choix et notez les étapes sous forme de commentaires `# TODO`. Passez ensuite à l'écriture de votre programme en syntaxe Python.

Créez un programme qui demande à l'utilisateur d'entrer trois fois un nom et un prénom que vous devez mettre dans une liste. Il doit également demander le nom de domaine à utiliser pour les trois personnes (par exemple: `gmail.com`, `cegepoutauais.qc.ca`, `hotmail.com`, etc.).

Le programme doit transformer les noms, prénoms et nom de domaine en trois adresses courriel au format suivant :

[prenom] . [nom]@[domaine]

✓ **Solution**

```

def generer_adresses_courriel(prenoms:list, noms:list, domaine:str):
    """
    Génère des adresses courriel au format [prenom].[nom]@[domaine]
    :param prenoms: Une liste contenant les prénoms.
    :param noms: Une liste contenant les noms.
    :param domaine: Le nom de domaine à utiliser pour les adresses.
    :return: Une liste contenant les adresses courriel générées
    """

    adresses_courriel = []

    for i in range(len(prenoms)):
        prenom = prenoms[i]
        nom = noms[i]

        email = f"{prenom.lower()}.{nom.lower()}@{domaine}"  # I
        adresses_courriel.append(email)

    return adresses_courriel


if __name__ == "__main__":
    prenoms = []
    noms = []

    for i in range(3):
        prenom = input(f"Entrez le prénom de la personne {i + 1}: ")
        nom = input(f"Entrez le nom de la personne {i + 1}: ")

        # Utilisation de strip() pour enlever les espaces superflus
        noms.append(nom.strip())
        prenoms.append(prenom.strip())

    nom_domaine = input("Quel est le nom de domaine à utiliser")

    adresses_courriel = generer_adresses_courriel(prenoms, noms)

    print("\nAdresses courriel générées :")
    for adresse in adresses_courriel:
        print(adresse)

```

## Exercice 5 : listes à deux dimensions

Soit la liste suivante de noms de technologies suivante :

```
technologies = ["Python", "JavaScript", "Java", "C++", "C#",  
                "MySQL", "PostgreSQL", "MongoDB", "Oracle",  
                "HTML", "CSS", "React", "Node.js",  
                "Linux", "Windows", "macOS", "iOS", "Android"]
```

Sachant que :

- Les éléments aux indices 0 à 4 sont des langages de programmation.
- Les éléments aux indices 5 à 8 sont des Bases de données.
- Les éléments aux indices 9 à 12 sont des technologies web.
- Les éléments aux indices 13 à 17 sont des systèmes d'exploitation.

créez une nouvelle liste contenant des éléments qui sont eux-mêmes des listes.  
Ces dernières listes sont, respectivement, celles des langages de programmation,  
des bases de données, des technologies web et des systèmes d'exploitation.

**Liste à deux dimensions attendue :**

```
[["Python", "JavaScript", "Java", "C++", "C#"],  
 ["MySQL", "PostgreSQL", "MongoDB", "Oracle"],  
 ["HTML", "CSS", "React", "Node.js"],  
 ["Linux", "Windows", "macOS", "iOS", "Android"]]
```

Affichez chacune des sous listes de technologies à partir de la liste à deux dimensions (en utilisant des indices).

**Résultat attendu :**

```
Les langages de programmation : ["Python", "JavaScript", "Java", "C++", "  
Les bases de données : ["MySQL", "PostgreSQL", "MongoDB", "Oracle"]  
Les technologies web : ["HTML", "CSS", "React", "Node.js"]  
Les systèmes d'exploitation : ["Linux", "Windows", "macOS", "iOS", "Andro
```

## Exercice 6

Faites la même chose que l'exercice 5 avec la liste suivante qui est légèrement différente de la précédente. À vous de remarquer les différences.

```
technologies = ["Langages de programmation", "Python", "JavaScript", "Java",  
                "Bases de données", "MySQL", "PostgreSQL", "MongoDB", "Oracle",  
                "Web", "HTML", "CSS", "React", "Node.js",  
                "Systèmes d'exploitation", "Linux", "Windows", "macOS", "Android"]
```

# Les dictionnaires

## Qu'est ce qu'un dictionnaire ?

- Collection de clés et de valeurs associées
- Les clés sont uniques
- Est *mutable* → on peut le modifier (clés et valeurs) après la création
- Est *iterable* → on peut utiliser un `for .. in` pour parcourir le dictionnaire
- On accède à une valeur par sa clé et non par sa position  
ex. `voiture["modele"]`

## Syntaxe d'un dictionnaire - déclaration

```
# Sur une ligne avec accolades
voiture = { 'marque' : 'Toyota', 'modele' : 'MR2', 'annee' : 1996 }

# Ou sur plusieurs lignes avec accolades
voiture = {
    'marque' : 'Toyota',
    'modele' : 'MR2',
    'annee' : 1996
}

# Avec dict() et des séquences
voiture = dict([('marque', 'Toyota'), ('modele', 'MR2'), ('annee', 1996)])

# Avec dict() quand les clés sont des chaînes de caractères
voiture = dict(marque='Toyota', modele='MR2', annee=1996)
```

## Ajouter un élément

La clé s'ajoute au dictionnaire lorsqu'on lui assigne une valeur

```
# Déclaration
voiture = {
    'marque' : 'Toyota',
    'modele' : 'MR2',
    'annee' : 1996
}

# Ajout d'une valeur
voiture['couleur'] = 'Bleu ciel'
```

## Manipulation (quelques méthodes)

- Parcourir : `for cle, valeur in dict.items()`
- Supprimer une entrée dans le dictionnaire (paire clé/valeur) : `del dict[cle]`
- Retourner toutes les clés dans l'ordre d'insertion : `list(dict)`
- Vérifier si une clé est dans le dictionnaire : `cle in list(dict)`
- Retourner toutes les valeurs du dictionnaire : `dict.values()`
- Retirer une clé du dictionnaire et retourner sa valeur : `dict.pop(cle)`
- Retirer et retourner la dernière paire clé, valeur du dictionnaire (utile pour vider progressivement un dictionnaire) : `dict.popitem()`
- Vider le dictionnaire : `dict.clear()`

## Ressources

- Tutoriel sur les dictionnaires dans la doc officielle de Python :  
<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>
- Méthodes des dictionnaires dans la doc officielle :  
<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

# Exercices

Dictionnaires, listes

## But du laboratoire

- Réfléchir aux structures de données
- Manipuler les collections de données
- Résolution de problème
- Commencer la révision

## Exercice 1

### Aide

- Utiliser un dictionnaire si vous voulez une structure de donnée pour rechercher de l'information.

### Instructions

1. Pour chaque description, indiquer si on doit utiliser une liste ou un dictionnaire
  - Les articles à acheter à l'épicerie
  - Les informations d'un jeu vidéo (titre, description, prix, etc.)
  - Tous les films en ma possession (seulement les titres)
  - Une liste de vos amis et leur numéro de téléphone principal (un seul numéro)
  - La liste de matériel dans un PC et leurs caractéristiques (ex. Carte graphique: 2 Go, RAM: 32 Go, Carte réseau 1000 Mbps)
2. Écrire un programme python simple qui :

- Permet de créer, de modifier et d'afficher un jeu vidéo (titre, description, prix, etc.)
  - Permet de gérer la liste de matériel dans un PC et leurs caractéristiques (ex. Carte graphique: 2 Go, RAM: 32 Go, Carte réseau 1000 Mbps)
3. Écrire un programme python qui permet de créer, de modifier et d'afficher une collection de jeux vidéos. Pour chaque jeu on doit stocker le titre, la description et le prix.

## Exercice 2

### Instructions

Bob et Ginette ont un enfant et, pendant la grève, ils ne veulent pas que leur enfant oublie les capitales des provinces et territoires du Canada. Comme ils sont des parents *cool* (en tous cas dans leur tête), ils veulent lui programmer un petit jeu questionnaire pour le motiver. Comme ils ne savent pas programmer et que vous êtes un.e ami.e de la famille, c'est à vous que revient la tâche.

Le jeu doit afficher une province ou une capitale au hasard et demander à l'utilisateur d'écrire le nom de la capitale de celle-ci (sans tenir compte de la casse). Si l'utilisateur n'entre aucune réponse, le jeu termine et affiche le pointage.

Selon le pointage, un message d'encouragement sera affiché

- [0-50%] : Continue de pratiquer si tu veux du temps d'écran.
- [50-60%] : Ça s'en vient !
- [60-80%] : Tu y es presque.
- [80-100%] : Super, tu es prêt !

### Code fourni

```
# Les provinces associées à leur capitale.  
provinces = {  
    "Yukon" : "Whitehorse",  
    "TNO" : "Yellowknife",  
    "Nunavut" : "Iqaluit",  
    "Terre-Neuve" : "St-John's",  
    "IPE" : "Charlottetown",  
    "Nouvelle-Écosse" : "Halifax",  
    "Nouveau-Brunswick" : "Fredericton",  
    "Québec" : "Québec",  
    "Ontario" : "Toronto",  
    "Manitoba" : "Winnipeg",  
    "Saskatchewan" : "Regina",  
    "Alberta" : "Edmonton",  
    "Colombie-Britannique" : "Victoria"  
}
```

## Approche suggérée :

Travailler en équipe au moins pour l'analyse (étapes 1 et 2)

1. Analyser l'énoncé
2. Déterminer les fonctions nécessaires, déterminer l'algorithme (pseudo-code et/ou logigramme)
3. Coder la solution (seul.e ou en équipe, avec par exemple code with me)

## Pour aller plus loin

Offrir un choix de réponse (le bon choix doit toujours être présent et les choix ne peuvent pas se répéter pour la même question)

*Note : Les exercices sont tirés et adaptés des exercices de Rémy Corriveau*

# Différents types de traitements

# Structures logiques séquentielle

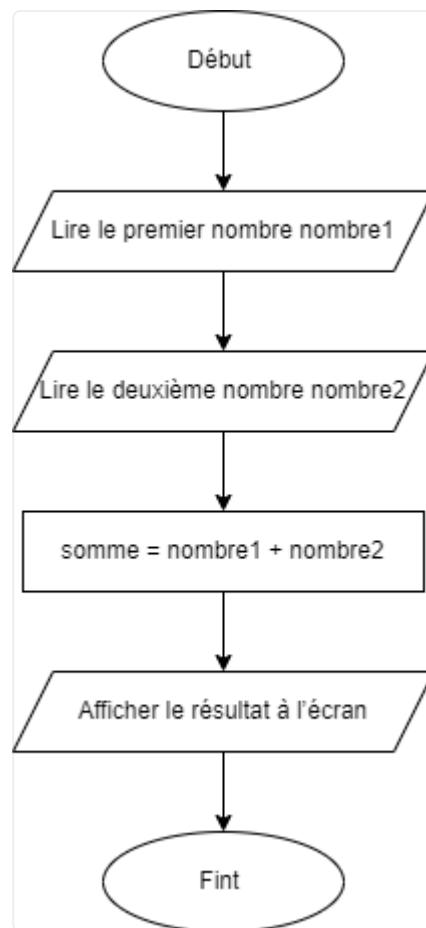
## Définition

Une structure séquentielle consiste simplement en une énumération des traitements et des Entrée/Sortie (E/S ou I/O), sans aucune condition ou répétition de traitements.

### Exemple :

Pour additionner deux nombres, les étapes consistent à :

- Lire le premier nombre.
- Lire le deuxième nombre.
- Faire la somme des deux nombres et les stocker dans une variable.
- Afficher le résultat à l'écran.



# Structures logiques conditionnelles (if/else/elif)

## Définition

Une structure conditionnelle ou décisionnelle est la comparaison d'un, deux ou plusieurs éléments. Cette structure implique l'utilisation des opérateurs logiques. On cherche à connaître l'état (vrai ou faux) de l'expression de comparaison.

Il existe trois types de structure décisionnelle :

### Structure décisionnelle simple

Cette structure permet l'exécution d'une tâche (une ou plusieurs instructions) selon l'état de la condition.

Syntaxe Python :

```
if condition:  
    instructions
```

La structure se lit : si `condition` est vraie alors exécuter les `instructions`.

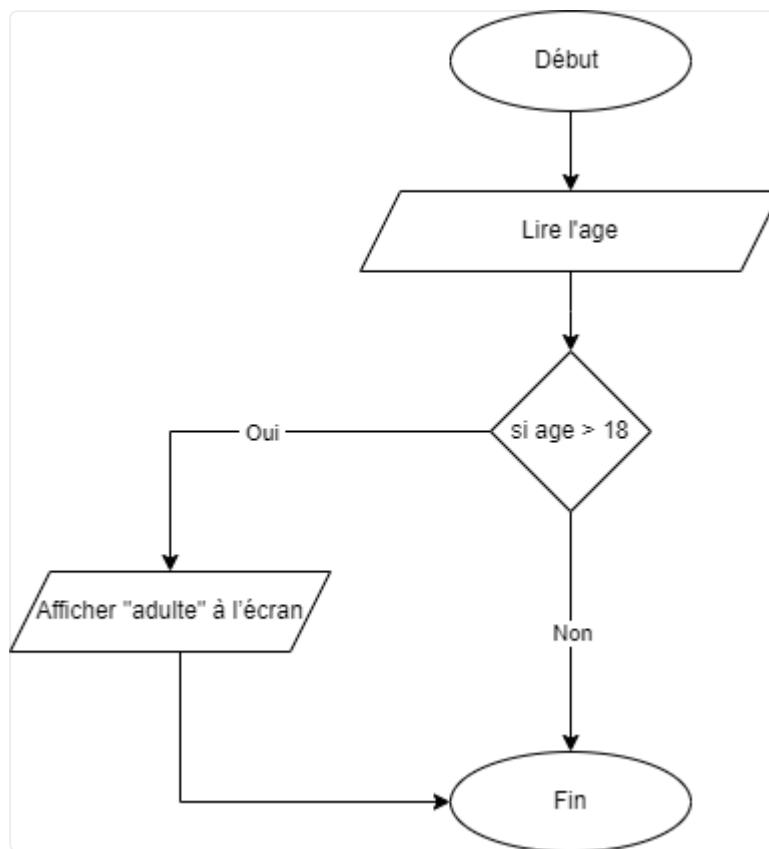
`condition` étant une variable contenant une valeur booléenne ou une expression de comparaison ayant comme résultat vrai ou faux (True ou False en Python).

`instructions` étant un ensemble d'instructions qui vont s'exécuter si la condition est True.



Les indentations sont importantes en langage Python. C'est ce qui donne sa structure au code Python et qui lui permet de s'exécuter correctement.

**Exemple :**



```

age = input("Veuillez entrer l'age de la personne:")
if age > 18:
    print("Adulte")
  
```

## Structure décisionnelle double

Si la condition est vraie, une suite séquentielle d'instructions sera exécutée. Dans le cas où la condition est fausse, c'est une autre suite séquentielle d'instructions (différente de la précédente) qui sera exécutée.

**Syntaxe Python :**

```

if condition:
    instructions1
else:
    instructions2
  
```

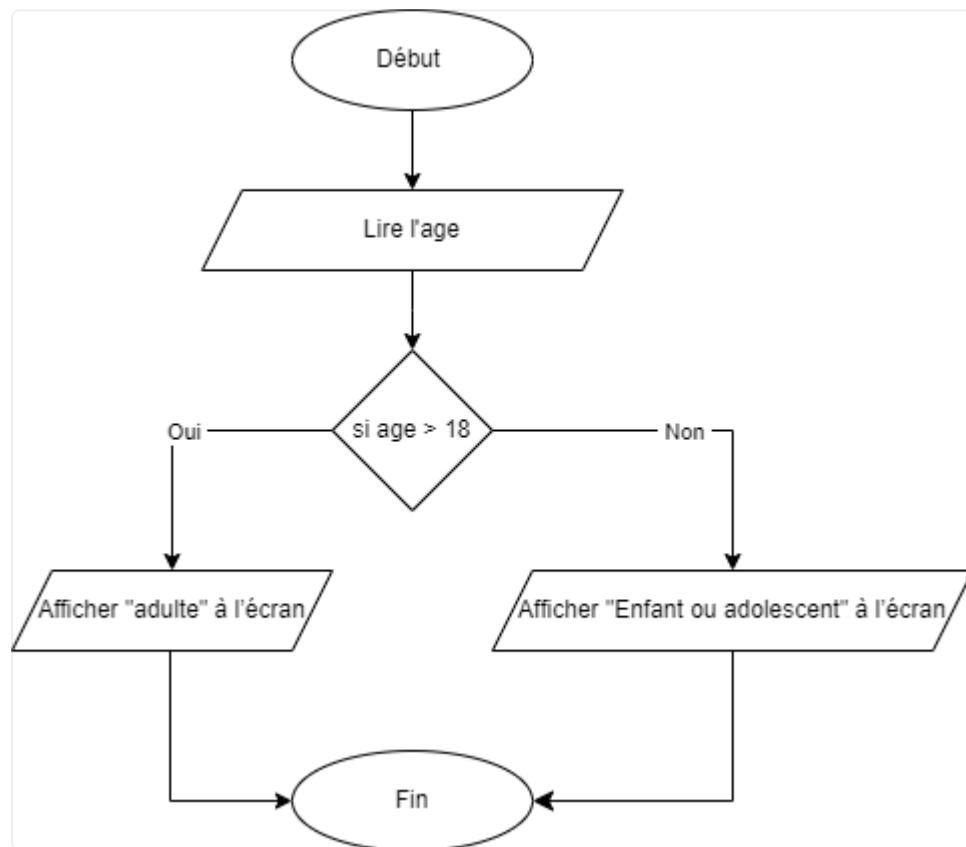
La structure se lit : si `condition` est vraie alors exécuter les `instructions 1` sinon exécuter les `instructions 2`.

`condition` étant une variable contenant une valeur booléenne ou une expression de comparaison ayant comme résultat vrai ou faux (True ou False en Python).

`instructions1` étant un ensemble d'instructions qui vont s'exécuter si la condition est True.

`instructions2` étant un ensemble d'instructions qui vont s'exécuter si la condition est False.

### Exemple : organigramme et code source en Python



```

if age > 18:
    print("Adulte")
else:
    print("Enfant ou adolescent")
  
```

# Structure décisionnelle multiple

Si la condition est vraie, une suite séquentielle d'instructions sera exécutée. Dans le cas où la condition est fausse, c'est une autre condition qui sera évaluée, et si elle est vraie, une autre suite séquentielle d'instructions, ainsi de suite. Si aucune des conditions n'est satisfaite, c'est une autre suite séquentielle d'instructions qui sera exécutée (complètement différente des précédentes).

## Syntaxe Python

```
if condition1:  
    instructions1  
elif condition2:  
    instructions2  
elif condition3:  
    instructions3  
...  
else:  
    instructionsn
```

La structure se lit : si `condition1` est vraie, alors exécuter les `instructions 1`, sinon si `condition2` est vraie, alors exécuter les `instructions 2`, sinon si `condition3` alors exécuter les `instructions 3` ... sinon exécuter les `instructions n`.

`condition1`, `condition2`, ... étant des variables contenant des valeurs booléennes ou des expressions de comparaison ayant comme résultat vrai ou faux (True ou False en Python).

`instructions1`, `instructions2`, ... étant des ensembles d'instructions qui vont s'exécuter si les conditions (resp.) `condition1`, `condition2`, ... sont vraies (True).

`instructionsn` étant un ensemble d'instructions qui vont s'exécuter si aucune condition précédente n'est vraie (True).

## Exemple : avec chaînes de caractères

```
fruit = input("Saisissez un fruit : ")

if fruit == "Banane" or fruit == "Ananas":
    print(f"Le fruit {fruit} est jaune")
elif fruit == "Orange" or fruit == "Abricot":
    print(f"Le fruit {fruit} est orange")
elif fruit == "Pitaya":
    print(f"Le fruit {fruit} est rose")
else:
    print(f"Désolé, ce fruit {fruit} n'existe pas dans ma base de connais")
```

# Structures logiques répétitives (boucles)

## Introduction

La structure répétitive (ou boucle) est une répétition du même bloc de code, jusqu'à l'obtention d'un résultat escompté. L'arrêt de cette boucle se fait selon une condition ou après un certain nombre d'itérations déterminé.

Il existe deux types de boucles en Python : les boucles `while` et les boucles `for`.

## Boucles `while`

Le traitement s'exécute tant que la condition à l'entrée de la structure est vraie, donc le code est exécuté de 0 à plusieurs fois. La structure se lit : "tant que la `condition` est vraie, exécuter les `instructions`".

### Syntaxe Python

```
while <condition>:  
    <instructions à répéter>
```

### Exemple 1

```
compteur = 1  
while compteur < 10:  
    print("compteur = ", compteur)  
    compteur += 1
```

- (i) Notez que pour un arrêt assuré de la boucle, la condition doit évoluer, i.e. elle doit passer de `True` à `False` à un moment donné. Dans l'exemple

précédent, on incrémente le `compteur` à chaque itération, ce qui fait que lorsque le compteur est égal à 10, on sort immédiatement de la boucle.

## Boucle for

Le traitement se répète autant de fois qu'il y a d'éléments dans une collection (un ensemble d'éléments, exemple : une liste). La structure se lit :

pour chaque élément dans la `collection` exécuter les `instructions`.

### Syntaxe Python

```
for <c> in <collection>:  
    <instructions à répéter>
```

## La fonction `range()`

La fonction native `range()` est utilisé lorsqu'on veut itérer à travers une séquence de nombres. Elle est souvent utilisée dans les boucles `for`.

**i** Notez que la fonction `range()` ne génère pas une liste d'entiers, mais plutôt un objet de type "range". Vous pouvez le convertir en liste en utilisant `list()`.

- `range(stop)` : génère une séquence de nombres de `0` jusqu'à `stop-1`.  
Exemple :

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

Essayez ceci :

```
for i in range(10):  
    print(i)
```

- `range(start, stop)` : génère une séquence de nombres de `start` jusqu'à `stop-1`. Exemple :

```
>>> list(range(4, 10))
[4, 5, 6, 7, 8, 9]
```

Essayez ceci :

```
for i in range(4, 10):
    print(i)
```

- `range(start, stop, step)` : génère une séquence de nombres de `start` jusqu'à `stop-1` par pas de `step`. Exemple :

```
>>> list(range(4, 10, 2))
[4, 6, 8]
```

Essayez ceci :

```
for i in range(4, 10, 2):
    print(i)
```

## Les instructions de contrôle des boucles

### `break`

Permet de sortir prématurément de la boucle `for` ou `while` la plus proche (dans le cas de plusieurs boucles imbriquées).

#### Exemple

Dans cet exemple, l'exécution de la boucle s'arrête lorsque `i` est égal à 7. Les valeurs affichées sont 0, 1, 2, 3, 4, 5 et 6.

```
for i in range(10):
    if i == 7:
        break
    print(i)
```

## continue

Permet de passer immédiatement à l'itération suivante d'une boucle `for` ou `while` sans exécuter le reste du code.

### Exemple

Dans cet exemple, lorsque `i` est égal à 3, on passe à l'itération suivante (`i=4`) sans exécuter le `print(i)`. Les valeurs de 0 à 9 sont toutes affichées sauf 3.

```
for i in range(10):
    if i == 3:
        continue  # Passe à l'itération suivante sans exécuter les instru
    print(i)
```

## else

Une clause `else` peut être incluse après une boucle `for` ou `while`. Le bloc `else` contient des instructions qui seront exécutées une fois que la boucle est terminée.

- Dans une boucle `for`, le bloc `else` est exécuté après que la boucle ait atteint sa dernière itération.
- Dans une boucle `while`, le bloc `else` est exécuté après que la condition de la boucle ait atteint la valeur `False`.

Dans les deux cas, le bloc `else` n'est pas exécuté lorsque la boucle a été arrêté via un `break`.

### Exemple 1

Dans cet exemple, la boucle s'exécute jusqu'à la fin (`i==9`), donc le bloc `else` est exécuté et affiche le message "La boucle est terminée."

```
for i in range(10):
    print(i)
else:
    print("La boucle est terminée.")
```

## Exemple 2

Dans cet exemple, la boucle est interrompue à l'itération 7, donc le bloc `else` ne s'exécute pas.

```
for i in range(10):
    if i == 7:
        break # Sort de la boucle
    print(i)
else:
    print("La boucle est terminée.")
```

## Exemple boucles imbriquées

- Il est possible d'avoir deux boucles (ou plus) imbriquées l'une à l'intérieur de l'autre.
- `continue` et `break` s'appliquent à la boucle la plus proche d'eux. i.e.

Dans l'exemple suivant, si `break` s'exécute, on sort de la boucle intérieure (la plus proche), mais on continue l'exécution de la boucle extérieure normalement car `break` ne s'applique pas sur celle-ci.

```

for i in range(1, 4):
    print(f"Boucle extérieure, itération {i}")

    for j in range(1, 6):
        if j == 3:
            continue

        if i == 2 and j == 4:
            break
        print(f"    Boucle intérieure, itération {j}")

    print("-----")

```

## Exemple de boucle while True/break

Parfois, nous devons forcer la création de boucles infinies et utiliser le mot clé `break` pour en sortir sous une condition. Cette pratique doit cependant être limitée à des cas où un `while <condition>` ne s'applique pas.

### Exemple



À la place de cette boucle :

```

while True:
    commande = input("Entrer une commande: ")
    if commande == "exit":
        break

```



Privilégiez cette boucle :

```

commande = ""
while commande != "exit":
    commande = input("Entrer une commande: ")

```

# Exemples avec des listes

## Exemple 1 : parcours d'une liste avec la boucle while

```
fruits = ['pommes', 'poires', 'bleuets', 'raisins', 'bananes', 'pamplemou  
i = 0  
while i < len(fruits):  
    print("fruit = ", fruits[i])  
    i += 1
```

## Exemple 2 : parcours d'une liste par éléments avec la boucle for

```
fruits = ['pommes', 'poires', 'bleuets', 'raisins', 'bananes', 'pamplemou  
for fruit in fruits:  
    print("fruit = ", fruit)
```

## Exemple 3 : parcours d'une liste par les indices avec la boucle for

([Voir la description de la fonction range\(\)](#))

```
fruits = ['pommes', 'poires', 'bleuets', 'raisins', 'bananes', 'pamplemou  
for i in range(len(fruits)):  
    print("fruit = ", fruits[i])
```

- ➊ Notez que contrairement aux boucles *while*, les valeurs de *i* évoluent en prenant le prochain élément de la liste (retournée par `range(len(fruits))`) à chaque fois, nous n'avons pas besoin de le faire nous même.

# Référence



## 4. More Control Flow Tools

Python documentation



# Exercices de base

## But

Se familiariser avec les bases des boucles avant de les appliquer à des énoncés plus complexes.

## Exercice 1

Créer une boucle `while` de 1 à 10 et n'afficher que les nombres pairs.

Faites le même exercice avec la boucle `for`.

### Résultat attendu

```
#2  
#4  
#6
```

## Exercice 2

Créer une boucle `while` de 10 à 1 et afficher les nombres impairs.

Faites le même exercice avec la boucle `for`.

### Résultat attendu

```
#9  
#7  
#5  
...
```

## Exercice 3

Écrire un programme qui affiche :

```
*  
* *  
* * *  
* * * *  
* * * * *
```

## Exercice 4

Créer une boucle `while` qui demande son nom à une personne. Si le nom n'est pas vide, on affiche `Bonjour <prenom>`. Sinon, on sort de la boucle.

Faites le même exercice avec la boucle `for`.

## Exercice 5

Créez une fonction qui force l'usager à entrer un nombre pair au clavier. Le nombre doit être supérieur ou égal à 0.

## Exercice 6

- ✓ Utilisez les légos mis à votre disposition pour vous aider à réfléchir.  
Différentes couleurs/formes de légos pour pour le groupe A et différentes couleurs/formes pour le groupe B.

À l'aide de boucles, affichez toutes les combinaisons possibles entre les groupes A et B.

Faites l'exercice avec les boucles `while` et `for`.

Groupe A	Groupe B
1	5
2	6
3	7
4	8

### Résultat attendu

1-5  
1-6  
1-7  
1-8  
2-5  
2-6  
...  
4-7  
4-8

## Exercice 7

Pour chaque mois, afficher le nombre de jours.

### Résultat attendu :

Le mois 01 possède 31 jours  
Le mois 02 possède 28 ou 29 jours  
Le mois 03 possède 31 jours  
...  
Le mois 11 possède 30 jours  
Le mois 12 possède 31 jours

# Laboratoire boucles - solution

## But

- Maîtriser les **boucles**.
- Consolider la matière vue précédemment.
- Continuer à utiliser les fonctions.

## Énoncé

Pour chaque exercice, vous devez également :

- Créer le pseudo code ou l'organigramme avant de le traduire en code Python.
- Créer un plan de tests et bien tester son programme.
- Utiliser le débogueur au besoin.

## Exercice 1

Il s'agit de faire un jeu à 2 joueurs pour deviner un nombre. Le premier joueur devra entrer le nombre à faire deviner et le second tentera de trouver le nombre. Pour aider, le programme devra indiquer s'il faut monter ou descendre. Le pointage final équivaut au nombre de tentatives.

### Suggestion :

Pour faire "disparaître" le nombre choisi par le premier joueur, effectuer 100 print vide après avoir obtenu le nombre.

▼ Solution

```
##### Pseudo code - fonction deviner_nombre #####
# Paramètres d'entrée de la fonction : nomre_a_deviner
# Sortie : aucun (on affiche le résultat)
#####
# Demander au joueur de proposer un nombre
# Tant que nombre != nombre_a_deviner#
#     Si le nombre proposé == nombre à deviner
#         On a trouvé le nombre, donc on sort de la boucle (break)
#     Sinon
#         Si le nombre proposé est < nombre à deviner
#             Dire à l'utilisateur de proposer un nombre plus grande
#         Sinon
#             Dire à l'utilisateur de proposer un nombre plus petite
#     #
#     Demander au joueur de proposer un nombre
#
# Afficher le Bravo, avec la bonne réponse et le score
#####
```

```
##### Pseudo-code programme principal #####
Demander à l'utilisateur le nombre à faire deviner et le mettre
Laisser 100 lignes vides.
Appeler la fonction : deviner_nombre(nombre_gagnant)
#####
```

```
def deviner_nombre(nombre_a_deviner:int):
    """
        Fonction qui permet de guider l'utilisateur
        à deviner un nombre passé en paramètres.
    :param nombre_a_deviner: Le nombre à deviner.
    :return: Aucun
    """

    score = 0
    trouve = False

    while not trouve:
        score += 1

        nombre_propose = int(input(f"Tentative {score}: Quel nor
            if nombre_propose == nombre_gagnant:
                trouve = True
            else:
                if nombre_propose < nombre_a_deviner:
                    print("Le nombre est plus haut")
                else:
                    print("Le nombre est plus petit")

            print("*****")
            print(f"Bravo! La bonne réponse était {nombre_gagnant}")
            print(f"Votre score est de {score}")

    if __name__ == "__main__":
        nombre_gagnant = int(input("Quel est le nombre à faire deviner?"))
        print("\n" * 100)

        deviner_nombre(nombre_gagnant)
```

## Exercice 2

Imaginez que vous développez un programme pour une application de gestion de prêts. Vous devez développer un programme qui aidera les utilisateurs à suivre le remboursement de leurs dettes. Le remboursement de la dette peut-être fait en plusieurs fois. L'utilisateur doit fournir des montants de remboursement jusqu'à ce

que la dette soit entièrement payée ou que l'utilisateur entre une valeur non numérique.

À la fin du processus, le programme doit afficher le nombre total de paiements effectués.

### Résultat attendu :

```
Quel est le montant de la dette? 4000
Dette: 4000.0$. Combien voulez-vous rembourser? 2500
Dette: 1500.0$. Combien voulez-vous rembourser? 1000
Dette: 500.0$. Combien voulez-vous rembourser? 500
Dernier remboursement de 500.0 $ effectué
*****
Nombre de paiements:      3
Montant total remboursé: 4000.0 $
*****
```

### ▼ Solution

```
##### pseudo-code fonction remboursements_dette
# paramètres d'entrée : dette
# Valeur de retour (sortie) : aucun (affichage)
#####
# nb_remboursement = 0
# total_remboursements = dette
#
# Tant que la dette n'est pas totalement remboursée (dette > 0)
#     Demander montant de remboursement à l'utilisateur : remboursement
#     Si remboursement < 0 alors
#         Afficher qu'il est impossible de faire le remboursement
#         On ignore la suite des instruction et on re-boucle
#     Sinon si remboursement >= dette, alors
#         dette = 0
#         Incrémenter le nombre de remboursement : nb_remboursement
#     Sinon
#         Mettre à jour la dette avec le montant remboursé : dette = dette - remboursement
#         Incrémenter le nombre de remboursement : nb_remboursement = nb_remboursement + 1
#
##### programme principal #####
# Lire le montant de la dette
# Appeler la fonction remboursements_dette(dette)
#####
```

- Code Python

```

def remboursements_dette(dette : float):
    nb_remboursements = 0
    total_remboursements = dette

    while dette > 0:
        remboursement_str = input(f"Dette: {dette}$. Combien voulez-vous rembourser? ")

        if remboursement_str.isalpha():
            break

        remboursement = float(remboursement_str)

        if remboursement < 0 :
            print("Impossible de faire le remboursement, nombre négatif")
            continue

        elif remboursement >= dette:
            print(f"Dernier remboursement de dette ({dette}) effectué")
            dette = 0
            nb_remboursements += 1
        else:
            dette -= remboursement
            nb_remboursements += 1

    total_remboursements = total_remboursements - dette

    print("*" * 20)
    print(f"Nombre de paiements: {nb_remboursements}")
    print(f"Montant total remboursé: {total_remboursements} $")
    print("*" * 20)

if __name__ == "__main__":
    dette = float(input("Quel est le montant de la dette? "))
    remboursements_dette(dette)

```

## Exercice 3

Imaginez que vous travaillez sur un projet de suivi climatique pour une station météorologique. Le programme doit permettre à l'utilisateur d'entrer des

températures successivement. Lorsque l'utilisateur souhaite terminer, il devra entrer le mot "arrêt".

À la fin, le programme affichera la température la plus basse, la température la plus haute et la température moyenne parmi celles qui ont été saisies.

```
Entrez une température (ou 'arrêt' pour finir) : 25
Entrez une température (ou 'arrêt' pour finir) : 18
Entrez une température (ou 'arrêt' pour finir) : 22
Entrez une température (ou 'arrêt' pour finir) : arrêt

La température la plus basse est : 18
La température la plus élevée est : 25
La température moyenne est : 21.67
```

#### ▼ Solution

```
def calculer_min(temp_min, temperature):
    """
    Fonction qui retourne la température minimale entre la nouvelle et la température minimale actuelle.

    :param temp_min: Température minimale actuelle.
    :param temperature: Nouvelle température.
    :return: La température minimale mise à jour.
    """

    if temperature < temp_min:
        temp_min = temperature

    # Notez qu'il est possible de trouver la plus petite température avec la fonction min()
    # temp_min = min(temp_min, temperature)

    return temp_min

def calculer_max(temp_max, temperature):
    """
    Fonction qui retourne la température maximale entre la nouvelle et la température maximale actuelle.

    :param temp_max: Température maximale actuelle.
    :param temperature: Nouvelle température.
    :return: La température maximale mise à jour.
    """

    if temperature > temp_max:
        temp_max = temperature

    # Notez qu'il est possible de trouver la plus grande température avec la fonction max() :
    # temp_max = max(temp_max, temperature)

    return temp_max

def calculer_somme(cumul_temp, temperature):
    """
    Cette fonction met à jour le cumul des températures en ajoutant une valeur de température au cumul existant.

    :param cumul_temp: Le cumul actuel des températures.
    :param temperature: La nouvelle température à ajouter au cumul.
    :return: Le nouveau cumul des températures après l'ajout.
    """

    cumul_temp += temperature
```

```
        return cumul_temp

def calculer_moyenne(total, nb_valeurs):
    """
    Fonction qui calcule et retourne la moyenne des températures.

    :param total: La somme des températures.
    :param count: Le nombre total de températures.
    :return: La moyenne des températures.
    """

    moyenne = total / nb_valeurs
    return moyenne


def afficher_statistiques(temp_min, temp_max, temp_moyenne):
    """
    Fonction qui affiche les statistiques des températures.

    :param temp_min: Température minimale.
    :param temp_max: Température maximale.
    :param temp_moyenne: Température moyenne.
    """

    print(f"Température minimum : {temp_min}")
    print(f"Température maximum : {temp_max}")
    print(f"Température moyenne : {temp_moyenne}")


if __name__ == "__main__":
    temp_min = float('inf')
    temp_max = float('-inf')
    total_temp = 0
    compteur = 0

    temperature_str = input("Veuillez entrer une température : ")

    while temperature_str != "arrêt" :

        temperature = float(temperature_str)

        temp_min = calculer_min(temp_min, temperature)
        temp_max = calculer_max(temp_max, temperature)
        total_temp = calculer_somme(total_temp, temperature)
        compteur += 1

        temperature_str = input("Veuillez entrer une température : ")
```

```
temp_moyenne = calculer_moyenne(total_temp, compteur)
afficher_statistiques(temp_min, temp_max, temp_moyenne)
```

# Modularisation

# Création et appels des fonctions

## Définition d'une fonction

Une fonction est un bloc d'instructions dont le but est de faire une tâche spécifique du programme.

## Quel est le but des fonctions

Le but dans la création de fonctions est de découper le programme en plusieurs parties (modularisation) afin de les réutiliser à plusieurs endroits du programme et améliorer l'organisation et la lisibilité du code.

## Les fonctions natives (ou intrinsèques)

Les fonctions intrinsèques sont des fonctions prédéfinies de Python (*Build-in functions*) que le programmeur peut utiliser.

Exemples de fonction (vous utilisez déjà quelques unes) :

- `print(chaîne)` : affiche la chaîne de caractères passé en paramètre.
- `len(liste)` : retourne la longueur d'une liste.
- `type(x)` : retourne le type de la variable x.
- `int(x)` , `float(x)` , `str(x)` : retournent respectivement les valeurs converties en entier, réel et chaîne de caractère.
- `range(n)` : retourne une séquence de nombres selon ce qu'on lui passe en paramètres.

Vous trouverez plus de fonctions natives dans la documentation officielle de Python suivante :



# Syntaxe

## Déclaration d'une fonction

```
def nom_fonction(parametre_1, parametre_2, ..., parametre_n):
    instruction_1
    instruction_2
    ...
    instruction_n
    return resultat # Résultat étant la valeur de retour de la fonction
```

- `def` : c'est un mot clé pour définir une fonction en Python.
- `nom_fonction` : c'est le nom de la fonction. Celui ci doit être significatif et soigneusement choisi (tout comme le nom d'une variable).
- `parametre_1, parametre_2, ..., parametre_n` (optionnel): ce sont les paramètres de la fonction. On les utilise pour passer des valeurs à la fonction `nom_fonction`. Un paramètre doit être utilisé dans le corps de la fonction et il prend la valeur qu'on lui passe lorsqu'on appelle la fonction. En dehors de cette fonction, le paramètre n'existe pas dans le reste du code.
- `instruction_1, instruction_2, ..., instruction_n` : ce sont des lignes de code formant le corps de la fonction.
- `return resultat` (optionnel): `return` est un mot clé qui permet de retourner une valeur (`resultat`) calculée dans la fonction. `resultat` est appelée valeur de retour.



### Ne pas oublier

- les deux points `:`.
- l'indentation dans le corps de la fonction.

Ils font partie de la syntaxe de la fonction.



Une fonction est exécutée uniquement lorsqu'elle est appelée.

## Appel d'une fonction

Pour appeler une fonction, on doit écrire le nom de la fonction en lui passant les paramètres dont elle a besoin. La fonction doit évidemment être définie avant.



Il est possible d'appeler une fonction

- dans le programme principal.
- dans une autre fonction.

## Exemples

### Exemple 1 : fonction sans paramètres et sans valeur de retour

Déclaration de la fonction print\_hello\_world

```
def print_hello_world():
    print("**** Hello World ****")
```

Appel de la fonction print\_hello\_world

```
print_hello_world()
```

### Exemple 2 : fonction avec paramètres et sans valeur de retour

Déclaration de la fonction affichage

```
def affichage(chaine1, chaine2):
    print("*" * 50)
    concatenation = chaine1 + " " + chaine2
    print(concatenation)
    print("*" * 50)
```

Appel de la fonction affichage

```
affichage("chaine de caractères 1", "chaine de caractères 2")
```

### Exemple 3 : fonction avec paramètres et avec valeur de retour

Déclaration de la fonction addition

```
def addition(x, y, z):
    somme = x + y + z
    return somme
```

Appel de la fonction addition

```
addition(15, 30, 7)
```

# Programme principal

`if __name__ == "__main__":` vient après la déclaration des fonctions et marque le début du programme principal.

#### Définition simplifiée :

`if __name__ == "__main__":` s'assure que le programme en cours d'exécution est le programme principal afin que son contenu (sous le `if __name__ == "__main__":`) soit exécuté.

#### Définition un peu moins évidente pour vous :

`if __name__ == "__main__":` est utilisé pour déterminer si un programme Python (dans un fichier .py) est exécuté en tant que programme principal ou s'il est importé en tant que module dans un autre programme. Si celui est importé, le programme principal (sous le `if __name__ == "__main__":`) n'est pas exécuté.

#### Exemple :

```
def addition(x, y, z):  
    somme = x + y + z  
    return somme  
  
if __name__ == "__main__":  
    addition(15,30,7)
```

## Paramètres par défaut

Il est possible de définir des valeurs par défaut pour les paramètres d'une fonction. Les valeurs par défaut sont utilisées lorsque la fonction est appelée sans fournir une valeur explicite pour un paramètre.

### Syntaxe

```
def nom_fonction(parametre_1, parametre_2 = valeur_par_defaut2, parametre  
# Corp de la fonction
```



Assurez-vous de placer les paramètres avec des valeurs par défaut à la fin de la liste des paramètres de la fonction.

### Exemple

```
def addition(x, y = 5, z = 3)
    somme = x + y + z
    return somme

if __name__ == "__main__":
    somme1 = addition(2)
    somme2 = addition(2, 0)
    somme3 = addition(2, 0, 6)
    somme4 = addition(x=9, y=6, z=0)
    somme5 = addition(z=0, x=9, y=6)

    print(somme1)
    print(somme2)
    print(somme3)
    print(somme4)
    print(somme5)
```

---

## Consolidation sur les fonctions avec des blocs

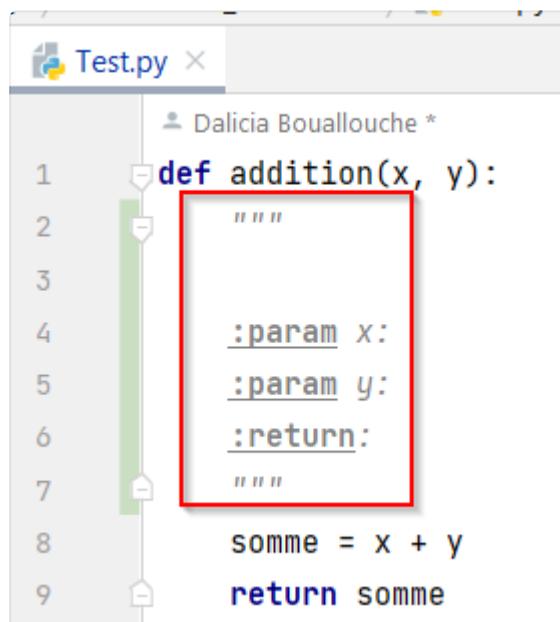
&gt;

# La documentation des fonctions

## Les Docstrings en Python

La documentation des fonctions en Python se fait à l'aide de docstrings. Un docstring est un bloc de commentaires qui se trouve au début du corps de la fonction. Elle sert à définir une fonction, ses paramètres et sa valeur de retour.

Pour générer automatiquement le docstring, entrez « """ », ensuite appuyez sur la touche « Entrée ». Une chaîne de caractères formattée va apparaître. Elle contient des mots clés « :param » pour chaque paramètre de la fonction **que vous devez définir** (s'ils existent) et un mot « :return » si la fonction a une valeur de retour **que vous devez définir**.



The screenshot shows a Python code editor window titled "Test.py". The code defines a function "addition" with parameters "x" and "y". A red box highlights the automatically generated docstring, which includes the triple quotes """" and the parameter definitions :param x: and :param y:. The code editor interface shows line numbers from 1 to 9 on the left and a vertical green bar indicating the current cursor position.

```
def addition(x, y):
    """
    :param x:
    :param y:
    :return:
    """
    somme = x + y
    return somme
```

Docstring généré AVANT la description de la fonction, des paramètres et de la valeur de retour

```

    Dalicia Bouallouche *
def addition(x, y):
    """
    Cette fonction permet d'additionner deux
    nombres x et y passés en paramètre.

    :param x: Le premier nombre à additionner
    :param y: Le deuxième nombre à additionner
    :return: La somme des deux nombres x et y.
    """

    somme = x + y
    return somme

```

Docstring généré APRÈS la description de la fonction, des paramètres et de la valeur de retour

## Exemple

```

def trouver_element(liste, element):
    """
    Cette fonction recherche un élément dans une liste et renvoie True si

    :param liste: La liste dans laquelle rechercher
    :param element: L'élément à rechercher dans la liste.
    :return: True si l'élément est trouvé, False sinon.
    """

    if element in liste:
        return True
    else:
        return False

if __name__ == "__main__":
    ma_liste = [1, 2, 3, 4, 5]

    # Appel de la fonction
    element_est_trouve = trouver_element(ma_liste, 3)
    print(element_est_trouve)

```



**Vous pouvez ajouter un exemple dans la documentation de votre fonction :**

```
def trouver_element(liste, element):
    """
    Cette fonction recherche un élément dans une liste et renvoie

    :param liste: La liste dans laquelle rechercher
    :param element: L'élément à rechercher dans la liste.
    :return: True si l'élément est trouvé, False sinon.

    Example:
        ma_liste = [1, 2, 3, 4, 5]

        trouver_element(ma_liste, 3) --> True

        trouver_element(ma_liste, 6) --> False
    """

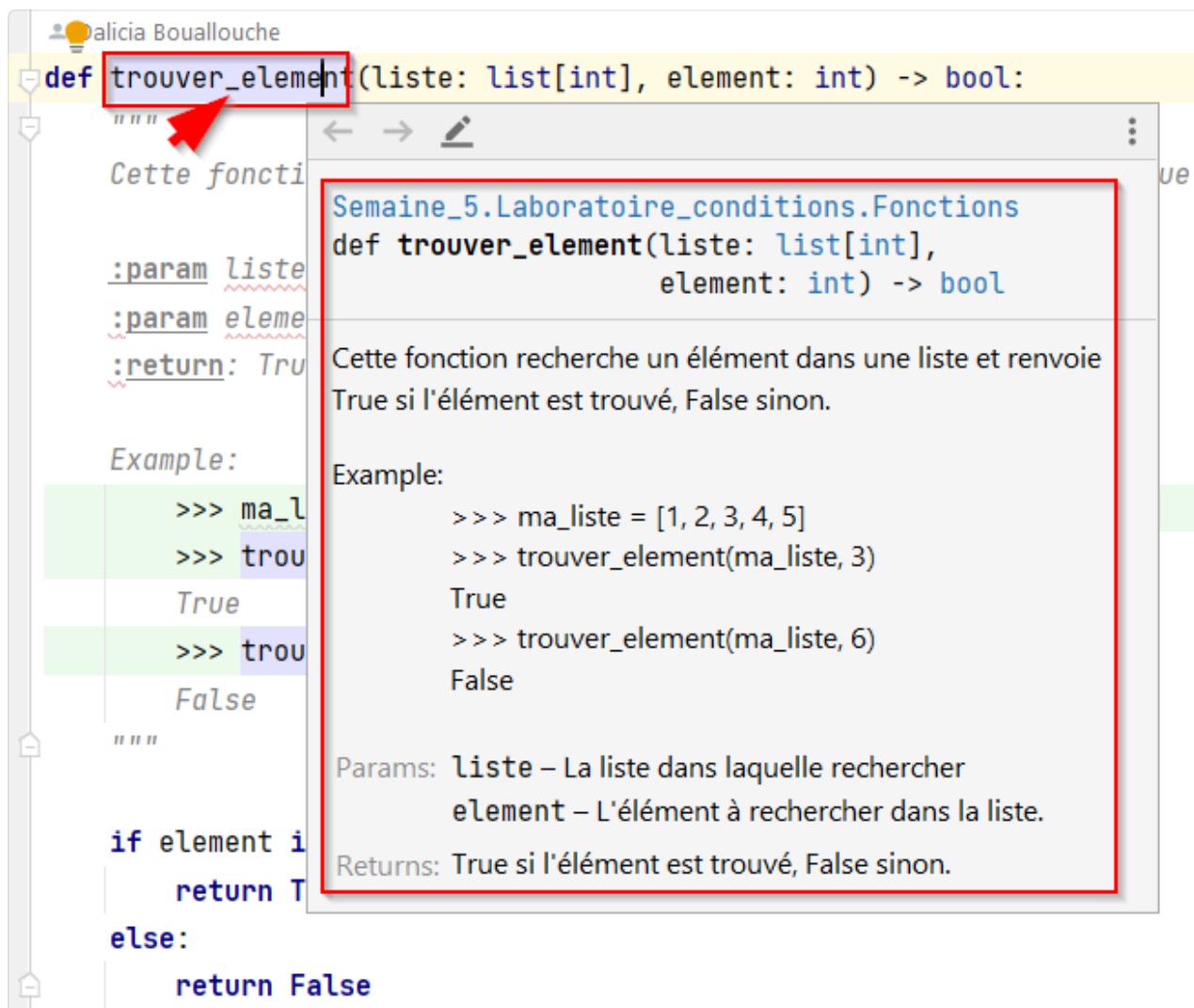
    if element in liste:
        return True
    else:
        return False

if __name__ == "__main__":
    ma_liste = [1, 2, 3, 4, 5]

    # Appel de la fonction
    element_est_trouve = trouver_element(ma_liste, 3)
    print(element_est_trouve)
```

## Comment se servir des docstrings ?

- En lisant le docstring dans le code source
- En positionnant le curseur de votre souris au dessus de l'appel à la fonction.



```
Salicia Bouallouche
def trouver_element(liste: list[int], element: int) -> bool:
    """
    Cette fonction recherche un élément dans une liste et renvoie True si l'élément est trouvé, False sinon.

    :param liste: La liste dans laquelle rechercher
    :param element: L'élément à rechercher dans la liste.
    :return: True si l'élément est trouvé, False sinon.

    Example:
        >>> ma_liste = [1, 2, 3, 4, 5]
        >>> trouver_element(ma_liste, 3)
        True
        >>> trouver_element(ma_liste, 6)
        False
    """
    if element in liste:
        return True
    else:
        return False
```

Semaine\_5.Laboratoire\_conditions.Fonctions

```
def trouver_element(liste: list[int],
                    element: int) -> bool
```

Cette fonction recherche un élément dans une liste et renvoie True si l'élément est trouvé, False sinon.

Example:

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> trouver_element(ma_liste, 3)
True
>>> trouver_element(ma_liste, 6)
False
```

Params: `liste` – La liste dans laquelle rechercher  
          `element` – L'élément à rechercher dans la liste.

Returns: True si l'élément est trouvé, False sinon.

- [Hors cours] En générant automatiquement un fichier de documentation tel que
  - Un README en utilisant la librairie pdoc.
  - Des documents HTML en utilisant des outils de documentation automatique tels que Sphinx, Doxygen ou pydoc intégré à Python.

# Annotation de types dans les fonctions

## Annotation de type (ou *Type hinting*)

Elle sert à indiquer explicitement les types de données attendus pour les paramètres de la fonction, ainsi que le type de la valeur de retour (le cas échéant). L'objectif principal de l'annotation de type est

- d'améliorer la lisibilité du code,
- de rendre plus claire la documentation (via les docstrings) et
- de faciliter la détection d'erreurs.

 Python est un langage de programmation dynamique qui n'impose pas de contraintes strictes sur les types de données. Les annotations de type ne sont pas strictement nécessaires pour que le code fonctionne. Cependant, elles sont une pratique recommandée pour les raisons citées plus haut.

## Syntaxe

```
def nom_fonction(parametre_1: type1, parametre_2: type2, ..., parametre_n
    # Code
    return resultat # Résultat étant la valeur de retour de la fonction
```

## Exemple :

```

def trouver_element(liste: list[int], element: int) -> bool:
    """
    Cette fonction recherche un élément dans une liste et renvoie True si

    :param liste: La liste dans laquelle rechercher
    :param element: L'élément à rechercher dans la liste.
    :return: True si l'élément est trouvé, False sinon.

    Example:
        ma_liste = [1, 2, 3, 4, 5]

        trouver_element(ma_liste, 3) --> True

        trouver_element(ma_liste, 6) --> False
    """

    if element in liste:
        return True
    else:
        return False

if __name__ == "__main__":
    ma_liste = [1, 2, 3, 4, 5]

    # Appel de la fonction
    element_est_trouve = trouver_element(ma_liste, 3)
    print(element_est_trouve)

```

# Appel de la fonction  
element\_est\_trouve = trouver\_element()  
print(element\_est\_trouve)

- (i)** Pour une fonction qui ne retourne rien, elle est annotée avec `-> None`.

**Exemple :**

```

def afficher_message(message: str) -> None:
    print(message)

```

# Laboratoire - fonctions (niveau débutant)

## But

- Se familiariser avec les bases des fonctions.
- Résolution de problèmes.

Notez qu'initialement les traitement conditionnels n'étaient pas demandés dans ce lab. Cependant, les solutions incluent des traitements conditionnels pour plus de pratique.

## Exercice 1

Des étudiants souhaitent avoir une application de mathématiques qui les aide à comprendre les propriétés des triangles, précisément, elle doit les aider à identifier si un triangle est rectangle ou non. Créez un tel programme en utilisant les fonctions.

Voici la formule de vérification qu'un triangle est rectangle :

$$a^2 + b^2 = c^2$$

▼ Solution

Pseudo-code

```
# ***** Les fonctions *****
Nom de la fonction qui vérifie si un triangle est rectangle : t:
# ***** entrées/sorties *****
Paramètres (entrées) : a, b et c
Retour (sorties) : vrai ou faux (boolean, True ou False)
# ***** Début pseudo-code *****
début fonction triangle_est_rectangle :
    si a**2 + b**2 == c**2 alors
        retourner True
    sinon
        retourner False

Fin de la fonction triangle_est_rectangle
```

```
# *****

Début du programme principal :
- Lire la valeur de a (l'utilisateur doit entrer la valeur de a)
- Lire la valeur de b
- Lire la valeur de c

triangle_est_rect = triangle_est_rectangle(a, b, c)

si triangle_est_rect == True, alors
    afficher : "Le triangle dont les valeurs sont " a", " b "et" c " est rectangle."
Sinon,
    afficher "Le triangle dont les valeurs sont " a", " b "et"
Fin du programme principal
# *****
```

## Programme en Python

```

def triangle_est_rectangle(a, b, c):
    """
    Vérifie si un triangle est rectangle.

    :param a: La longueur du premier côté du triangle.
    :param b: La longueur du deuxième côté du triangle.
    :param c: La longueur du troisième côté du triangle (hypoténuse)
    :return: (booléen) True si le triangle est rectangle, False
    """

    if a**2 + b**2 == c**2:
        return True
    else:
        return False

if __name__ == "__main__":
    a = float(input("Entrez la valeur de a: "))
    b = float(input("Entrez la valeur de b: "))
    c = float(input("Entrez la valeur de c: "))

    triangle_est_rect = triangle_est_rectangle(a, b, c)

    if triangle_est_rect:
        print(f"Le triangle dont les valeurs sont {a}, {b} et {c} est un rectangle")
    else:
        print(f"Le triangle dont les valeurs sont {a}, {b} et {c} n'est pas un rectangle")

```

### Plan de test :

a	b	c	Résultat
3	4	5	Le triangle est rectangle
5	12	13	Le triangle n'est pas rectangle
29	420	421	Le triangle est rectangle
24	143	170	Le triangle n'est pas rectangle
52	73	93	Le triangle n'est pas rectangle

## Exercice 2

Écrivez un programme qui fera la conversion d'un angle. L'utilisateur devra fournir un angle en degrés et le programme devra le convertir en radians.

La formule de conversion est la suivante :

$$rad = \pi/180 \times degrés$$

### Plan de tests

Valeurs d'entrée	Résultat observé
Angle en degrés : 0°	
Angle en degrés : 79°	
Angle en degrés : 360°	
Angle en degrés : 58.6°	
Angle en degrés : -45°	

## Exercice 3

Vous travaillez en tant que programmeur pour une application de gestion de profils. Votre tâche est d'écrire un programme qui récupère les informations d'un utilisateur pour lui créer un profil personnalisé (nom, prénom, date de naissance, profession, adresse de résidence et adresse courriel). Ce profil doit ensuite être affiché à l'écran.

L'adresse courriel de l'utilisateur doit être construite de manière automatique à partir des données saisies en respectant le format suivant :

[prénom].[nom]@[domaine]

Le nom de domaine ne change pas et doit être « cegepoutaouais.qc.ca »

Inspirez-vous des laboratoires précédents pour créer un affichage professionnel.

▼ **Solution**

**Pseudo-code**

```

# ***** Les fonctions *****
Fonction qui crée l'adresse courriel : creer_courriel
# ***** entrées/sorties *****
    Paramètres (entrées) : nom et prenom
    Retour (sorties) : l'adresse courriel
# ***** Début pseudo-code *****
début fonction creer_courriel
    Définir la constante NOM_DOMAINE = "cegepoutaouais.qc.ca"
    adresse_courriel = nom + "." + prenom + "@" + NOM_DOMAINE
    retourner adresse_courriel

Fin fonction
# *****

Fonction qui lit les informations du profil et qui affiche le p:
# ***** entrées/sorties *****
    Paramètres (entrées) : aucun
    Retour (sorties) : Aucun (affichage du profil)
# ***** Début pseudo-code *****
Début fonction afficher_profil
    Lire le nom (on demande à l'utilisateur d'entrer le nom)
    Lire le prénom
    Lire la date de naissance (demander un format, ex: JJ/MM/AAA)
    Lire la profession
    Lire l'adresse de résidence
    adresse_courriel = creer_courriel(nom, prenom)

    nom_complet = prenom + " " + nom

    Afficher toutes les informations lues et l'adresse courriel :
    "*****"
        Profil d'Alain:
        Nom complet: Alain Conn
        Date de naissance: 15/03/1860
        Profession: Chercheur introuvable
        Adresse de résidence: 42 Rue des Mystères, 00000 Fictionv:
        Adresse courriel: Alain.Conn@cegepoutaouais.qc.ca
    "*****"

Fin fonction

# *****
Début du programme principal :
    afficher_profil()
Fin du programme principal
# *****

```

## Programme en Python

```
def creer_courriel(nom, prenom):
    """
    Crée une adresse courriel à partir du nom et du prénom.
    :param nom: Le nom d'une personne.
    :param prenom: Le prénom d'une personne.
    :return: L'adresse courriel générée.
    """

    NOM_DOMAINE = "cegepoutaouais.qc.ca"
    adresse_courriel = nom + "." + prenom + "@" + NOM_DOMAINE

    return adresse_courriel

def afficher_profil():
    """
    Lit les informations de l'utilisateur et affiche le profil.
    :return: Aucun (on affiche le profil)
    """

    prenom = input("Entrez le prénom: ")
    nom = input("Entrez le nom: ")
    date_naissance = input("Entrez la date de naissance au format DD/MM/AAAA")
    profession = input("Entrez la profession: ")
    adresse_residence = input("Entrez l'adresse de résidence: ")

    adresse_courriel = creer_courriel(nom, prenom)

    nom_complet = prenom + " " + nom

    print("\n*****")
    print(f"Profil de {prenom}:")
    print(f"Nom complet: {nom_complet}")
    print(f"Date de naissance: {date_naissance}")
    print(f"Profession: {profession}")
    print(f"Adresse de résidence: {adresse_residence}")
    print(f"Adresse courriel: {adresse_courriel}")
    print("*****")

if __name__ == "__main__":
    afficher_profil()
```

# Exercice 4

Vous êtes développeur pour une entreprise de services financiers qui aide ses clients à gérer leurs investissements. Une de vos missions consiste à créer un programme pour vérifier automatiquement si des valeurs spécifiques, comme les prix des actions ou les taux d'intérêt, se situent dans une plage prédéfinie. Cela permet aux analystes financiers de savoir rapidement si les données respectent les seuils de tolérance ou les conditions fixées par les clients.

## ▼ Solution

### Pseudo-code

```
# ***** Les fonctions *****
Fonction qui vérifie si une valeur est dans un intervalle : valeur
# ***** entrées/sorties *****
    Paramètres (entrées) : valeur, seuil_inferieure, seuil_superieur
    Retour (sorties) : vrai ou faux (booléen, True ou False)
# ***** Début pseudo-code *****
début fonction valeur_dans_plage :
    si limite_inferieure <= valeur <= limite_superieure alors
        retourner Vrai
    sinon
        retourner Faux
Fin de la fonction valeur_dans_plage
# *****

Début du programme principal :
    Lire la valeur
    Lire la limite inférieure
    Lire la limite supérieure

    valeur_valide = valeur_dans_intervalle(valeur, limite_inferieure)
    si valeur_valide == True alors
        afficher "La valeur " valeur " est dans la plage prédéfinie"
    sinon
        valeur_valide "La valeur " valeur " n'est pas dans la plage"
Fin du programme principal
# *****
```

### Programme en Python

```

def valeur_dans_intervalle(valeur, seuil_inferieur, seuil_superieur):
    """
    Vérifie si une valeur est dans un intervalle spécifié.
    :param valeur: La valeur à vérifier.
    :param seuil_inferieur: Le seuil inférieur de l'intervalle.
    :param seuil_superieur: Le seuil supérieur de l'intervalle.
    :return: (bool) True si la valeur est dans l'intervalle, si non False.
    """
    return seuil_inferieur <= valeur <= seuil_superieur

if __name__ == "__main__":
    valeur = float(input("Entrez la valeur à vérifier: "))
    limite_inferieur = float(input("Entrez le seuil inférieur: "))
    limite_superieur = float(input("Entrez le seuil supérieur: "))

    valeur_valide = valeur_dans_intervalle(valeur, limite_inferieur, limite_superieur)

    if valeur_valide:
        print(f"La valeur {valeur} est dans la plage pré définie")
    else:
        print(f"La valeur {valeur} n'est pas dans la plage pré définie")

```

## Plan de tests

Valeur à vérifier	Seuil inférieur	Seuil supérieur	Résultat observé
134.73	120	139	La valeur 134.73 est dans la plage pré définie de 120.0 à 139.0.
10.5	9.8	10.5	La valeur 10.5 est dans la plage pré définie de 9.8 à 10.5.
228.67	137.62	228.65	La valeur 228.67 n'est pas dans la plage pré définie de 137.62 à 228.65.
74.22	71.13	77.16	La valeur 74.22 est dans la plage

prédéfinie de 71.13 à 77.16.

57.30

57.30

68.74

La valeur 57.3 est dans la plage prédéfinie de 57.3 à 68.74.

# Révision express fonctions

## But

- Révision rapide sur la création d'une fonction, l'appel d'une fonction et le passage de paramètres.
- Nouvelles notions dans la déclaration de fonctions :
  - Définition de valeurs par défaut aux paramètres de fonctions
  - Annotation de types.

## Révision

Complétez le programme suivant de façon à ce que ça affiche les messages successifs:

- "Bonjour, Alice !"
- "Bonjour, Bob !"

```
def salutation(nom):
    print(f"Bonjour, {nom} !")

if __name__ == "__main__":
    salutation()
    salutation()
```

### ✓ Solution

```
def salutation(nom):
    print(f"Bonjour, {nom} !")

if __name__ == "__main__":
    salutation("Alice")
    salutation("Bob")
```

### Autre solution :

```
def salutation(nom):
    print(f"Bonjour, {nom}!")

if __name__ == "__main__":
    nom_allocuteur = "Alice"
    salutation(nom_allocuteur)

    nom_allocuteur = "Bob"
    salutation(nom_allocuteur)
```

## Valeur par défaut dans les paramètres d'une fonction

[Plus de détails dans les notes de cours.](#)

Il est possible de donner une valeur par défaut à un paramètre d'une fonction.  
Exécutez ce programme, et analysez le résultat. Que remarquez-vous?

```
def salutation(nom="à tous"):
    print(f"Bonjour, {nom}!")

if __name__ == "__main__":
    salutation()
    salutation("Alice")
```

#### ▼ Explication

- Lorsqu'on appelle la fonction `salutation` sans passer de valeur au paramètre `nom` de la fonction `salutation()`, le paramètres `nom` de la fonction `salutation` prend par défaut la valeur "à tous".
- Lorsqu'on appelle la fonction `salutation` en passant une valeur au paramètre `nom` `salutation("Alice")`, le paramètres `nom` de la fonction prend cette valeur passée en paramètre (`"Alice"`).

Il est également possible de spécifier le nom du paramètre lors du passage d'une valeur dans la fonction.

```
def salutation(prenom, nom):
    print(f"Bonjour, {prenom} {nom} !")

if __name__ == "__main__":
    salutation(prenom="Alice", nom="Merveille")
```

[Plus de détails dans les notes de cours.](#)

## Annotation de types

Il est possible de spécifier le type de chaque paramètre dans une fonction (`str`, `int`, `float`, `list`, etc).

Exemple :

```
def salutation(prenom: str, nom: str):
    print(f"Bonjour, {prenom} {nom} !")

if __name__ == "__main__":
    salutation(prenom="Alice", nom="Merveille")
```

# Les modules

## Définition des modules (librairies ou bibliothèques)

Un module est un fichier (ou un dossier) contenant du code Python qui peut être importé et utilisé dans d'autres fichiers Python. Les modules sont une façon de structurer et d'organiser le code en Python, en les séparant en fonctionnalités logiques et en les rendant plus modulaires et réutilisables. Il existe des modules prêts à être utilisés, mais nous pouvons aussi créer des modules personnalisés.



**En bref:** un module est comme une boîte à outil. Elle contient des outils (qui sont les fonctions) qu'on peut réutiliser dans un programme après l'avoir importé.

Exemples de modules existants : `math`, `random`, `datetime`, `statistics`, etc.

## Documentation des modules `math`, `random` et `datetime`



`math` — Mathematical functions

Python documentation



`random` — Generate pseudo-random numbers

Python documentation



`datetime` — Basic date and time types

Python documentation



# L'importation de modules

Vous pouvez importer (i.e. ramener les fonctions utilitaires) des modules pour étendre les fonctionnalités du langage et vous aider à résoudre des problèmes plus facilement. L'importation d'un module en Python se fait à l'aide de l'instruction `import`. Lorsqu'un module est importé, il est possible de faire appel à ses fonctions afin de les utiliser dans notre programme.

## Syntaxes

Il existe plusieurs syntaxes d'importation en Python. Voici les principales :

### Syntaxe `import <nom_module>`

Importer un module complet dans votre programme. Le nom de la fonction à appeler dans le module doit être précédé par le nom du module.

```
import <nom_module>
```

### Exemple

```
import math  
  
print(math.sqrt(25))
```

### Syntaxe `from <nom_module> import <nom_fonction>`

Cette syntaxe permet d'importer une fonction spécifique d'un module plutôt que le module complet. Le reste des fonctions de ce module ne sont pas mises à disposition.

```
from <nom_module> import <nom_fonction>
```

## Exemples

```
from math import sqrt  
  
print(sqrt(25))
```

Importation de plusieurs fonctions et constantes d'un module en même temps :

```
from math import sqrt, pi, radians  
  
print(sqrt(25))  
print(pi)  
print(radians(90))
```

Syntaxe `from <nom_module> import *`

Cette syntaxe permet d'importer toutes les fonctions et constantes d'un module dans votre code. Le nom de la fonction à appeler ne doit pas être précédé par le nom du module.

```
from <nom_module> import *
```



Cette syntaxe n'est pas recommandée car elle peut créer des conflits de noms et rendre le code difficile à comprendre.

## Exemple

```
from math import *  
  
print(sqrt(25))  
print(pi)  
print(radians(90))
```

Syntaxe `import <nom_module> as <alias>`

Cette syntaxe permet d'importer un module sous un **alias** pour faciliter l'utilisation de son nom.

```
import <nom_module> as <alias>
```

## Exemples

```
import math as m
print(m.sqrt(25))
```

```
import datetime as dt

now = dt.datetime.now()
print("Date et heure actuelles : ", now)
```

# Les types de modules

## Les modules intégrés

Python a de nombreux modules intégrés qui peuvent être utilisés directement sans installation supplémentaire. Vous pouvez les importer directement en utilisant le mot clé import.

**Exemples** : math, datetime, statistics, random, etc.

## Les modules tiers (doivent être installés)

Il existe de nombreux modules tiers disponibles pour Python, qui ne sont pas inclus dans l'installation standard de Python. Vous pouvez les installer à l'aide de l'outil **pip** (installateur de packages Python) et les importer en utilisant le mot clé import.

**Exemples** : pytest, numpy, pygame, matplotlib, pandas, GeoPandas, folium, etc.

# Installation de modules tiers

Comme exemple, nous allons installer le module `pytest` dont nous aurons besoin dans les prochains cours.

- **En utilisant la commande pip dans le terminal de PyCharm :**

Allez dans le terminal de PyCharm et entrez la commande suivante :

- `pip install pytest`

```
PS E:\ProjetsPython1G2\projets-python-1-g-2-dalicia> pip install pytest
Collecting pytest
  Downloading pytest-7.4.3-py3-none-any.whl.metadata (7.9 kB)
Requirement already satisfied: iniconfig in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (2.0.0)
Requirement already satisfied: packaging in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (23.2)
Requirement already satisfied: pluggy<2.0,>=0.12 in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (1.3.0)
Requirement already satisfied: exceptiongroup>=1.0.0rc8 in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (1.1.3)
Requirement already satisfied: toml>=1.0.0 in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (2.0.1)
Requirement already satisfied: colorama in c:\users\dalicia\appdata\local\programs\python\python310\lib\site-packages (from pytest) (0.4.6)
  Downloading pytest-7.4.3-py3-none-any.whl (325 kB)
    325.1/325.1 kB 5.1 MB/s eta 0:00:00
  Installing collected packages: pytest
  Successfully installed pytest-7.4.3
PS E:\ProjetsPython1G2\projets-python-1-g-2-dalicia>
```



Il est parfois nécessaire d'exécuter la commande

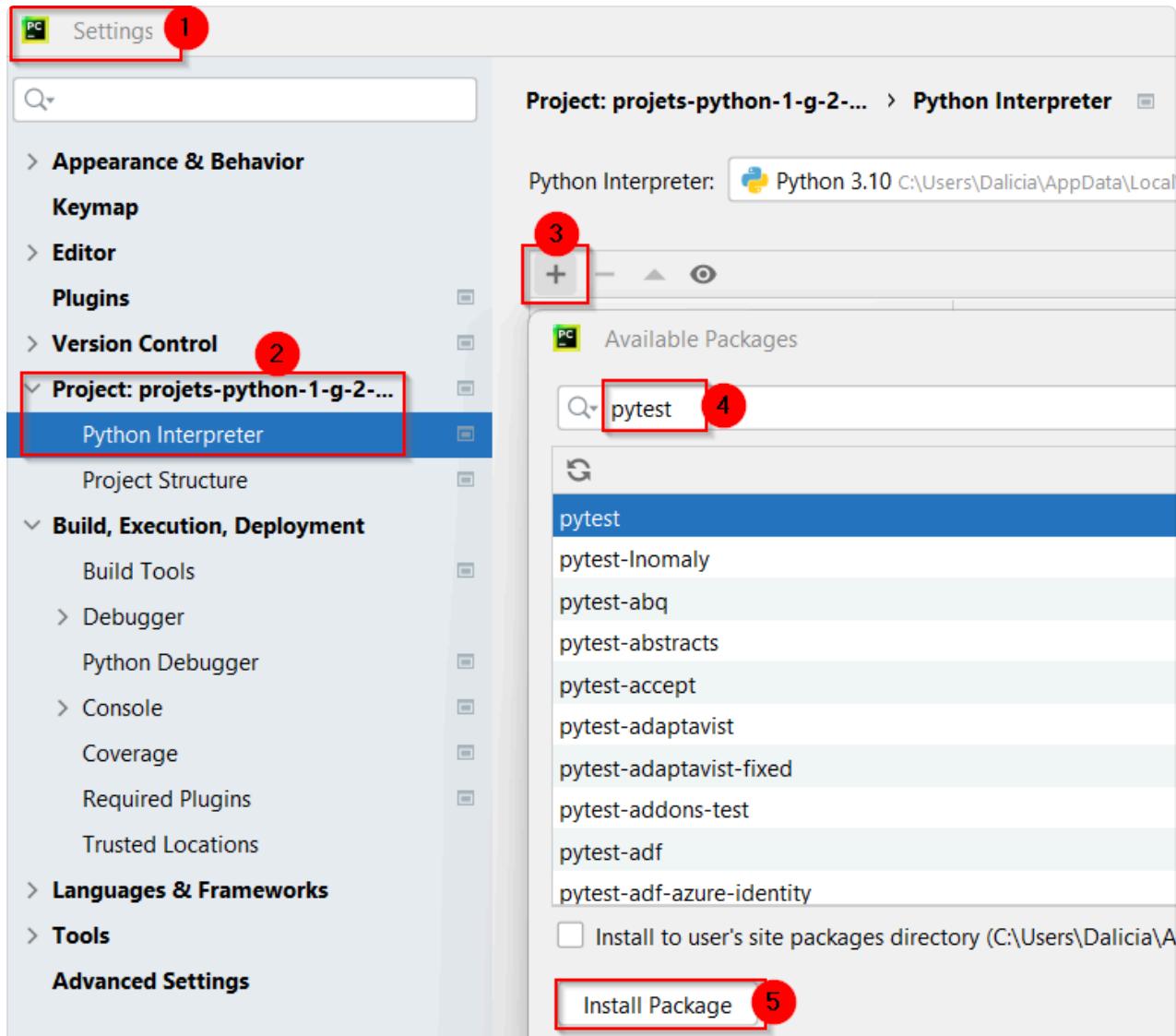
```
python.exe -m pip install --upgrade pip
```

pour mettre à jour l'installateur `pip`. En cas de besoin, le message de suivant apparaît :

```
[notice] A new release of pip is available: 23.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS E:\ProjetsPython1G2\projets-python-1-g-2-dalicia> python.exe -m pip install --upgrade pip
```

- **Dans l'IDE PyCharm :**

Allez dans File→Settings et suivez les numéros de l'image ci-dessous.



## Annexe

Quelques méthodes et constantes en vrac à tester pour les modules math, random, statistics et datetime.

```
math :
```

Méthodes :

- `sqrt`
- `abs`
- `math.radians(30)`, `math.degrees(math.pi/2)`
- `sin`, `cos`, `log`,

Constantes :

- `math.pi`
- `math.e`

Random :

Méthodes :

- `random.random()`
- `random.randint(a, b)`
- `random.choice()`

statistics:

Méthodes :

- `.mean`
- `.median`
- `.stdev`

```
datetime:
```

- `date_heure = datetime.datetime.now()`
- `datetime.datetime.date(date_heure)`
- `x = datetime.datetime(2020, 5, 17)`
  
- `x = datetime.datetime(2018, 6, 1)`  
`x.strftime("%B")`
  
- `date_heure = datetime.datetime(2022, 4, 18, 7, 58, 12)`  
`date_heure.strftime("%m/%d/%Y, %H:%M:%S")`
  
- `from datetime import timedelta`

# Laboratoire

## But

- S'exercer sur l'importation et l'utilisation de modules : exploration des modules `datetime` et `random`.
- Utilisation de la documentation officielle python d'un module.
- Les fonctions.



`datetime` — Basic date and time types

Python documentation



`random` — Generate pseudo-random numbers

Python documentation



## Exercice 1

**Durée :** 10 minutes, incluant recherche dans la documentation du module `datetime`.

Créez une date future de votre choix et affichez dans combien de jours cette date arrivera. Vous devez utiliser une fonction qui fait le calcul et l'affichage.

## Exercice 2

**Durée :** 20 minutes, incluant recherche dans la documentation du module `datetime`.

Créez un programme qui fera les actions suivantes :

- Demander à l'utilisateur une année, un mois et un jour, séparément.
- Créer une fonction qui transforme la date entrée par l'utilisateur au format `jour_semaine jour_mois mois annee`. Exemple : `Thursday 23 November 2023`

- . Pour afficher la date en français, utilisez :

```
import locale  
locale.setlocale(locale.LC_ALL, "fr_CA.utf8")
```

## Exercice 3

**Durée :** 30 minutes, incluant recherche dans la documentation du module `datetime`.

- Créez une fonction qui demande à l'utilisateur sa date de naissance au format `jour/mois/annee` et retourne la date de naissance en objet `datetime`.
- Créez une fonction qui retourne l'âge de la personne en années à partir de la date de naissance précédente (objet `datetime`).
- Créez une fonction qui retourne l'âge de la personne en jours à partir de la date de naissance précédente (objet `datetime`).
- Créez une fonction qui affiche la date de naissance complète (comme dans l'exemple).

**Exemple d'exécution :**

```
Veuillez entrer une date de naissance au format jj/mm/aaaa : 16/04/2015  
Vous avez 8 ans  
Vous êtes vieux de 3122 jours  
Vous êtes né(e) jeudi le 16 avril 2015
```

## Exercice 4

**Durée :** 30 minutes.

Implémentez une fonction qui crée une liste d'étudiants. Les informations de chaque étudiant devraient être son nom, son prénom et son numéro de groupe. Ces informations doivent être entrées par l'utilisateur.

À la fin de la création, la liste doit ressembler à ça :

```
[  
    ['Dupont', 'Jean', 1],  
    ['Martin', 'Alice', 2],  
    ['Tremblay', 'Pierre', 1],  
    ['Lefevre', 'Sophie', 2],  
    ['Dubois', 'Thomas', 2]  
]
```

Nous souhaitons former des binômes d'étudiants du même groupe. Le choix des étudiants doit se faire de façon aléatoire.

- Quelles sont les fonctions à créer, leurs paramètres d'entrées et leurs valeurs de retour ?
- Quel est le nom du module et la fonction du module qui permet de choisir aléatoirement un élément d'une liste.
- Implémentez ces fonctions de façon à créer une liste de binômes d'étudiants du même groupe.

## Exercice 5 : Jeu de Craps

**Durée :** 30 minutes.

### Partie 1 : lancement de dés.

- Commencez par créer une fonction qui simule le lancement de deux dés et qui donne la somme des deux valeurs des dés.
- Testez la et assurez-vous qu'elle fonctionne.

### Partie 2 : Jeu de craps

Créez un programme qui simule le jeu suivant.

**Note :** vous devez utiliser une (ou des) fonction(s).

**Objectif du Jeu :** Le but du jeu est de prédire le résultat d'un lancer de deux dés, en suivant les règles du craps.

### Règles du Jeu :

1. Au premier lancer (appelé le "tir de sortie"):
  - Si le total des dés est 7 ou 11, le joueur gagne.
  - Si le total des dés est 2, 3 ou 12, le joueur perd.
  - Sinon, le total des dés devient le "point" et le jeu passe à la phase suivante.
2. Durant les lancers suivants (phase de point) :
  - Le joueur doit relancer les dés jusqu'à ce qu'il obtienne à nouveau le "point" et gagne.
  - Si le joueur obtient un total de 7 avant d'atteindre à nouveau le "point", il perd.

### Déroulement du Jeu :

1. Le joueur choisit de commencer une partie ou non.
2. Le programme simule le lancer de dés.
  - Le jeu est terminé si le joueur gagne ou perd.
  - S'il s'agit d'une phase de point, le joueur relance les dés jusqu'à ce qu'il gagne ou perde.
  - Le résultat de la partie est affiché, et le joueur a la possibilité de jouer à nouveau.

# La portée des variables

## Dans les fonctions et le programme principal

- Une variable créée dans le programme principal est visible dans les fonctions.
- Une variable créée dans une fonction n'est pas visible.

### Exemple

Dans l'exemple suivant, il y a deux boîtes dans un espace de jeux. Les deux fonctions montrent les respectivement les contenus des deux boîtes. Le programme principal représente l'espace de jeux.

```
def afficher_contenu_boite_1():
    """
    Permet d'afficher le contenu de la boîte 1.
    :return: Aucun
    """
    jeu_boite_1 = "pistolet à eau"

    print("[" + "]")
    print(f"Dans la boîte 1, il y a un {jeu_boite_1}      []")

    print(f"Depuis la boîte 1, on peut voir la {jeu_espace_global}  []")
    print("[" + "]")

def afficher_contenu_boite_2():
    """
    Permet d'afficher le contenu de la boîte 2.
    :return:
    """
    jeu_boite_2 = "qaquettes de tennis"

    print("[" + "]")
    print(f"Dans la boîte 2, il y a des {jeu_boite_2}      []")
    print("[" + "]")

# Tentative d'accès à l'objet dans la boîte 1
print("Depuis la boîte 2, on peut voir un(e)", jeu_boite_1)
print("Erreur : On ne peut pas voir l'objet de la boîte 1 depuis la b")

if __name__ == "__main__":
    # Imaginez que la partie de programme principal est un espace global
    jeu_espace_global = "Jeux de sable"

    print("Explorons les boîtes...")
    afficher_contenu_boite_1()
    afficher_contenu_boite_2()

    # Essayons de voir les objets dans l'espace de jeux, à l'extérieur de
    print(f"Dans l'espace de jeux global, il y a un {jeu_espace_global} e")

    # Tentative d'accès aux objets des boîtes 1 et 2
    print("En dehors des boîtes, on peut voir un ", jeu_boite_1)
    print("En dehors des boîtes, on peut voir des ", jeu_boite_2)
```

# Dans les blocs if/elif/else

```
if __name__ == "__main__":
    # Une variable globale accessible partout dans la maison
    maison = "Vous êtes dans la maison principale."

    # choix = input("Faites un choix (cuisine ou salon) :")
    choix = 'cuisine'

    if choix == "cuisine":
        # Variable locale dans le bloc if (cuisine)
        objet_cuisine = "réfrigérateur"
        print("Vous êtes dans la cuisine et vous trouvez un :", objet_cui

        # On peut voir l'objet de la cuisine et la maison
        print("Depuis la cuisine, on peut voir :", maison)
    else:
        # Variable locale dans le bloc else (salon)
        objet_salon = "Télécommande"
        print("Vous êtes dans le salon et vous trouvez une :", objet_salo

        # On peut voir l'objet du salon et la maison
        print("Depuis le salon, on peut voir :", maison)

    # Tentatives d'accès aux objets après les blocs if/else
    print("Après exploration, on peut voir l'objet de la cuisine :", obje
    print("Après exploration, on peut voir l'objet du salon :", objet_sal

    print("Exploration de la maison...")
    print(maison)  # On peut toujours voir la maison
```

# Débogage

# Le débogueur, c'est ton ami!

# Introductions aux outils de débogage

Le débogage et la prise en main des outils du débogueur.

## Qu'est-ce qu'un bogue (bug)

Il s'agit d'une erreur dans un programme qui peut entraîner un comportement indésirable, des plantages ou des résultats incorrects.

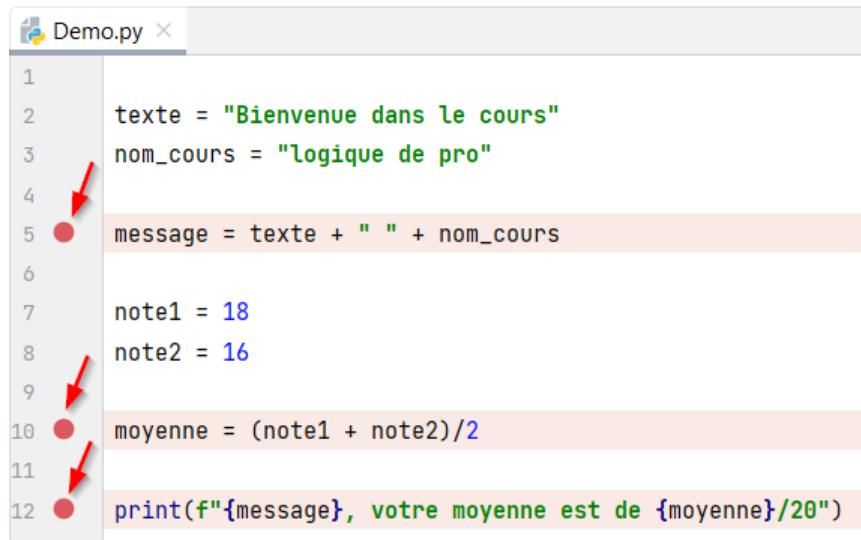
## Qu'est-ce que le débogage?

Le débogage est le processus de recherche, d'identification et de correction des erreurs dans un programme informatique.

Un débogueur est un outil (souvent intégré à l'IDE) qui permet de localiser les bogues de façon efficace et aide à les résoudre. Il contribue grandement à la productivité du programmeur.

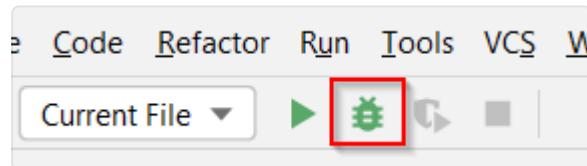
## Utilisation de base du débogueur (c'est ton ami 😊 🤝 )

- Placez les points d'arrêt :** un point d'arrêt est un marqueur qui indique au débogueur à quelle ligne suspendre l'exécution. Pour démarrer le débogage dans PyCharm, placez des points d'arrêt dans votre code en cliquant à gauche de la ligne de code souhaitée.

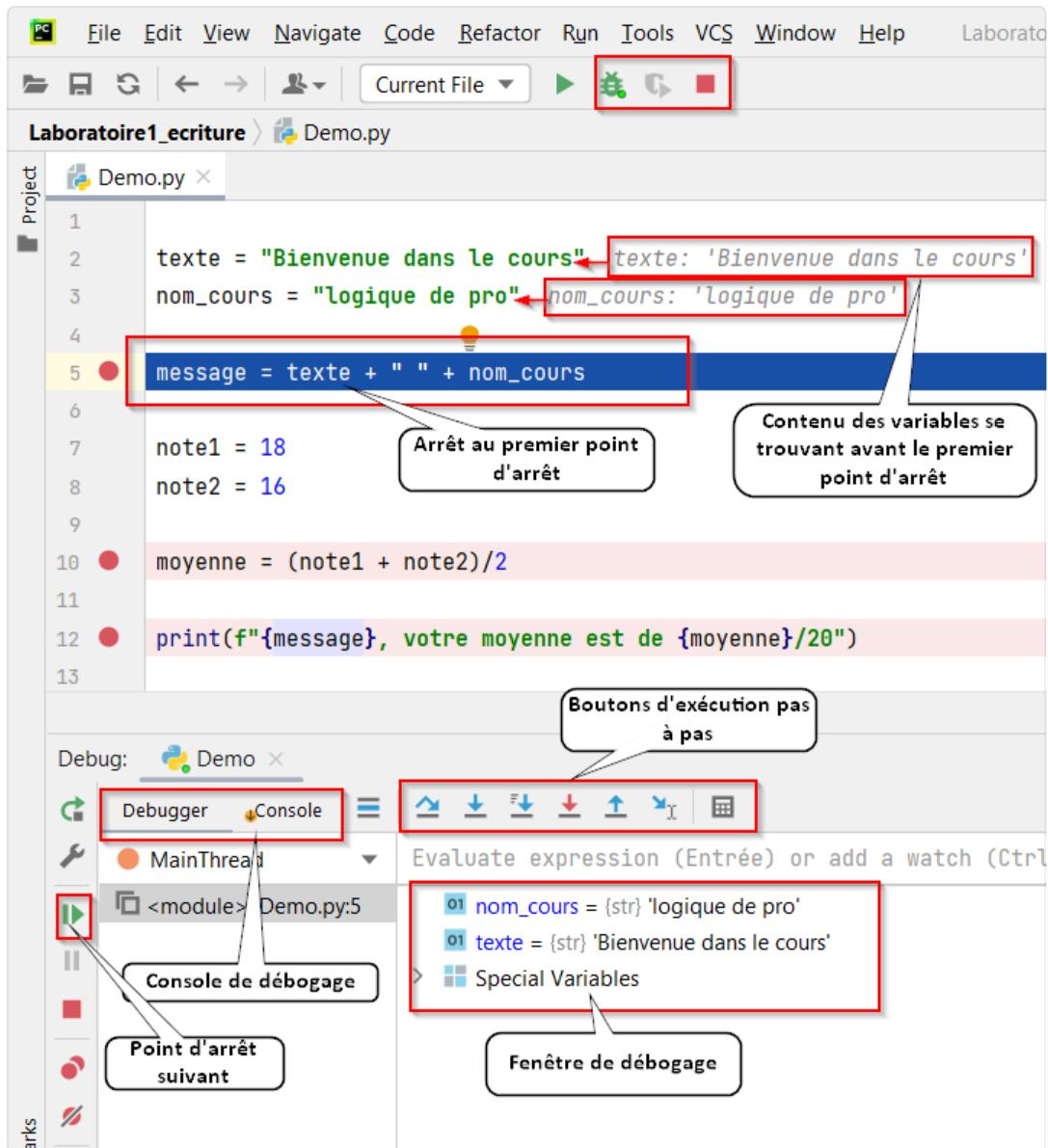


```
Demo.py x
1
2     texte = "Bienvenue dans le cours"
3     nom_cours = "logique de pro"
4
5     message = texte + " " + nom_cours
6
7     note1 = 18
8     note2 = 16
9
10    moyenne = (note1 + note2)/2
11
12    print(f"{message}, votre moyenne est de {moyenne}/20")
```

2. **Lancement du débogueur :** Une fois le(s) point(s) d'arrêt ajouté(s), lancez le débogueur en cliquant sur le bouton de débogage (représenté par un insecte).



3. **Fenêtre de débogage :** Une fois que le débogueur est lancé, l'exécution s'arrête au premier point d'arrêt et une fenêtre de débogage apparaît en bas de l'écran. Cette fenêtre vous permet de naviguer dans le code, de voir les valeurs des variables à chaque étape et de contrôler l'exécution.



4. **Exécution en mode pas à pas :** Pendant l'exécution du débogueur, vous pouvez avancer pas à pas à travers le code en utilisant des boutons ou des raccourcis clavier tels que
  - a. "Step Over" (F8) : Exécute la ligne actuelle et s'arrête à la ligne suivante (ne permet pas de rentrer dans une fonction).
  - b. "Step Into" (F7) : Permet de rentrer dans le code de la fonction si la ligne actuelle est un appel à une fonction. Sinon, exécute la ligne actuelle et s'arrête à la ligne suivante.
  - c. "Step Out" (Shift + F8) : permet de sortir d'une fonction lors du débogage.
5. **Inspecter les variables :** Dans la fenêtre de débogage, vous pouvez voir les valeurs actuelles des variables à chaque point d'arrêt. Vous pouvez également

ajouter des variables à la liste d'observation pour surveiller leurs valeurs tout au long de l'exécution.

## Exemple à déboguer (démo en classe)

```
message_bienvenue = "Bienvenue dans le cours"
nom_cours = "logique de programmation"

message_bienvenue = message_bienvenue + nom_cours

print(message_bienvenue)

note1 = input("Veuillez entrer votre note de l'examen 1 : ")

note2 = input("Veuillez entrer votre note de l'examen 2 : ")

moyenne = (note1 + note2)/2

print(f"Votre moyenne est de {moyenne}")
```

# Techniques de débogage

## Technique de base du débogage

1. Déterminer les variables clés à suivre dans le débogueur.
2. Placer les points d'arrêt à des endroits critiques de votre code où vous voulez examiner les variables.
3. Lancez l'exécution en mode débogage.
4. Exécuter votre programme pas à pas (ou point d'arrêt par point d'arrêt) :
  - a. Pensez au **résultat attendu** à chaque pas.
  - b. Inspectez les variables clés à chaque pas pour suivre le changement de leurs valeurs.
  - c. On cherche l'erreur: on cherche une contradiction entre le résultat/la valeur attendu et le résultat/la valeur trouvé par le programme.
5. **Répétez le processus** en plaçant des points d'arrêt à différents endroits (si nécessaire) jusqu'à ce que vous ayez trouvé l'endroit où l'erreur se trouve.

## Astuces

- Lorsque le code est trop gros, isolez votre problème dans le plus petit programme possible. **Exemples :**
  - Copiez une la petite partie de votre programme où se trouve l'erreur, isolez la dans un autre fichier, *hard codez* des valeurs pour aller plus vite (s'il le faut) et commencez à le déboguer.
  - Prenez la fonction dans laquelle le problème se trouve et exéutez la seule dans un autre fichier sans que ça passe par tout le programme principal et après exécution de plusieurs fonctions.

# Les erreurs et exceptions

# Introduction aux notions d'erreurs

## Introduction

Rien de plus irritant qu'une application qui plante! Le programmeur a le devoir de s'assurer qu'il n'y a pas de bogues dans son programme.



Les entrées inattendues de l'utilisateur peuvent générer des erreurs si le programmeur ne les contrôle pas.

## Les types d'erreurs

Il existe trois types d'erreur :

- Les erreurs de syntaxe
- Les erreurs de logique
- Les erreurs d'exécution

## Erreurs syntaxiques

Ce sont des erreurs dues au non-respect de la syntaxe du langage lors de la saisie du code. Un bon IDE (comme PyCharm) détectera ce type d'erreurs avant même l'exécution et pourrait suggérer un correctif.

Exemples:

- Oublier de fermer une parenthèse.

```
angle = float(input("Veuillez entrer une valeur d'angle en degrés:"))
angle_est_en_degres = angle_en_degres(angle)
```

- Oublier les deux points `:` dans la syntaxe du traitement conditionnel  
`if (condition) : .`

```
if mois in mois_complets_ete
    return True
```

- Ne pas faire d'indentation dans un traitement conditionnel `if`.

```
if (mois in mois_complets_ete) or (mo
    return True
else:
    return False
```

## Erreurs d'exécution

Surviennent lorsque le programme est en cours d'exécution mais rencontre un problème qui l'empêche de fonctionner correctement.

### Exemples :

- Division par zéro (0).

```
14: montant_total_individuel = montant_total_groupe / nb_personnes
Run: SolutionsExamenFacture
C:\Users\Dalicia\AppData\Local\Programs\Python\Python310\python.exe E:/ProjetsPython162/projets-python-1-q-2-dalicia/Semaine_5/Laboratoire_conditions\SolutionsExamenFacture.py
Veuillez entrer le montant du repas avant la taxe : 145
Veuillez entrer le pourcentage de pourboire : 14
Veuillez entrer le nombre de personnes qui doivent se diviser la facture : 0
Traceback (most recent call last):
  File "E:/ProjetsPython162/projets-python-1-q-2-dalicia/Semaine_5/Laboratoire_conditions\SolutionsExamenFacture.py", line 14, in <module>
    montant_total_individuel = montant_total_groupe / nb_personnes
ZeroDivisionError: float division by zero
```

- Utiliser un index qui dépasse la limite d'index d'une liste ou d'une chaîne.

```

2 couleurs = ["orange", "rouge", "jaune"]
3
4 print(couleurs[3])
5
Run: Demo
C:\Users\Dalicia\AppData\Local\Programs\Python\Python310\python.exe E:/ProjetsPython1G2/projets-python-1-g-2
Traceback (most recent call last):
  File "E:/ProjetsPython1G2/projets-python-1-g-2-dalicia/Semaine_6\Fonctions\Demo.py", line 4, in <module>
    print(couleurs[3])
IndexError: list index out of range

```

- Envoyer les mauvais paramètres à une fonction.
- Une erreur d'entrée de l'utilisateur non prise en compte par le programme :
  - On demande à l'utilisateur d'entrer un nombre et il entre un mot.
  - Un fichier que le programme veut ouvrir a été supprimé.
  - L'utilisateur a entré une valeur qui se trouve en dehors des limites demandées.

## Erreurs de logique

Dans ce cas, le programme s'exécute correctement mais l'erreur est que les résultats ne sont pas ceux attendus en raison d'une mauvaise compréhension de la logique du problème et de sa résolution. Elles sont plus subtiles à détecter comparé aux types d'erreurs précédents à cause du manque d'indications (soulignement ou messages d'erreur en rouge).

Exemple d'erreurs courantes:

- Conditions n'incluant pas les cas souhaités.
- Erreur dans le séquencement des opérations.
- Erreur dans l'algorithme en général.
- Etc.

# Les exceptions et leurs gestion

## Introduction

Quelles sont les erreurs possibles dans le code suivant ?

```
nombre = int(input("Entrez un nombre:"))
resultat = 10 / nombre
```

Si on entre :

- Une chaîne de caractères (autre que des nombres), alors nous aurons l'erreur suivante :

```
Entrez un nombre:cinq
Traceback (most recent call last):
  File "E:\ProjetsPython1G2\projets-python-1-q-2-dalicia\Semaine_6\Fonctions\Demo.py", line 1, in <module>
    nombre = int(input("Entrez un nombre:"))❸
ValueError: invalid literal for int() with base 10: 'cinq'❹
```

- Si nous entrons le chiffre 0, alors nous aurons l'erreur suivante :

```
Entrez un nombre:0
Traceback (most recent call last):
  File "E:\ProjetsPython1G2\projets-python-1-q-2-dalicia\Semaine_6\Fonctions\Demo.py", line 2, in <module>
    resultat = 10 / nombre❸
ZeroDivisionError: division by zero❹
```



- 1 : le chemin (incluant le nom) du fichier dans lequel se trouve l'erreur.
- 2 : le numéro de la ligne de code où se trouve l'erreur. **La cause peut provenir d'une ligne précédente (comme dans l'exemple suivant).**
- 3 : La ligne de code où se trouve l'erreur.
- 4 : Le **type de l'exception** et une brève description de l'erreur. Ici, les types d'exceptions dans les deux exemples sont `ValueError` et `ZeroDivisionError`.

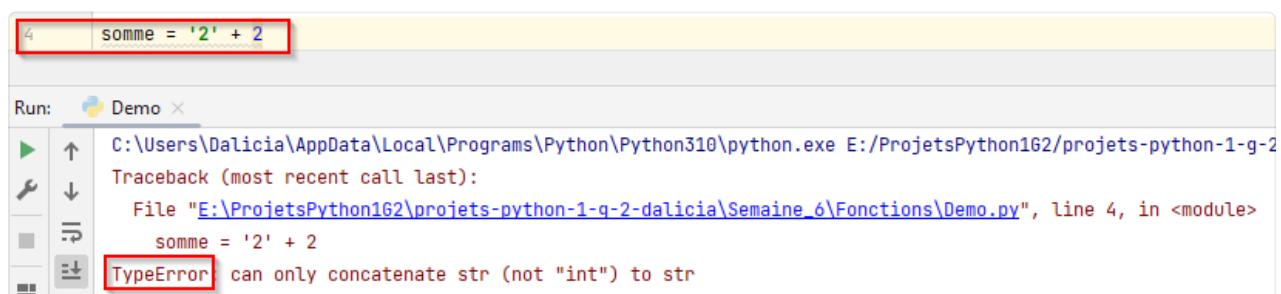
Pour éviter ces erreurs d'exécution, nous pouvons, dans certains cas, valider la valeur entrée à l'aide d'un `if` avant l'instruction dans laquelle il risque d'y avoir une erreur, ou d'utiliser la syntaxe `try ... except`.

## Les exceptions

Les erreurs détectées durant l'exécution sont appelées des *exceptions*. Dans les exemples d'erreurs précédents, les types d'exceptions sont `ValueError` et `ZeroDivisionError` (voir point rouge 4 dans les images précédentes).

Autres exemples :

- `TypeError`



```

4 somme = '2' + 2
Run: Demo ×
C:\Users\Dalicia\AppData\Local\Programs\Python\Python310\python.exe E:/ProjetsPython1G2/projets-python-1-q-2
Traceback (most recent call last):
  File "E:/ProjetsPython1G2/projets-python-1-q-2-dalicia/Semaine_6\Fonctions\Demo.py", line 4, in <module>
    somme = '2' + 2
TypeError: can only concatenate str (not "int") to str

```

**Signification du message d'erreur :** On ne peut pas concaténer une chaîne de caractères (str) et un entier (int).

- `NameError`



```

3 somme = x + 2
4
Run: Demo ×
C:\Users\Dalicia\AppData\Local\Programs\Python\Python310\python.exe E:/ProjetsPython1G2/projets-python-1-q-2
Traceback (most recent call last):
  File "E:/ProjetsPython1G2/projets-python-1-q-2-dalicia/Semaine_6\Fonctions\Demo.py", line 3, in <module>
    somme = x + 2
NameError: name 'x' is not defined

```

**Signification du message d'erreur :** la variable x n'est pas définie.

- `IndexError`

```
list = [1, 2, 3, 4, 5]
print(list[6])
```

Traceback (most recent call last):  
File "C:/Users/Valentia/AppData/Local/Programs/Python/3.10/python.exe", line 2, in <module>  
 print(list[6])  
IndexError: list index out of range

**Signification du message d'erreur :** le numéro de l'indice déborde (l'élément à l'indice 6 n'existe pas dans la liste).

Pour tous les types d'exceptions, voir la documentation officielle de python suivante :



Exceptions natives  
Python documentation



Documentation complète sur les exceptions :



8. Erreurs et exceptions  
Python documentation



## Gestion des exceptions

### Syntaxe try/except

```
try:  
    # Instruction(s) contenant potentiellement une exception.  
except <Type de l'exception>:  
    # Instructions de gestion de l'exception.
```

- Dans le bloc « `try` », on met le code que l'on veut gérer, donc susceptible de lever une exception.
- Dans le bloc « `except` », on met le code à exécuter en cas d'exception. Il est possible de gérer plusieurs exceptions pour la(les) même(s) instruction(s).

**Exemple :**

Gestion des exceptions du code suivant :

```
nombre = int(input("Entrez un nombre:"))
resultat = 10 / nombre
```

- Gestion de l'exception `ZeroDivisionError` : Le code suivant permet de détecter et gérer une exception de type `ZeroDivisionError` dans l'instruction `resultat = 10 / nombre`.

```
nombre = int(input("Entrez un nombre:"))

try:
    resultat = 10 / nombre

except ZeroDivisionError:

    print("La valeur entrée ne doit pas être 0!")
```

- Gestion de l'exception `ValueError` : Le code suivant permet de détecter et gérer une exception de type `ValueError` dans l'instruction `nombre = int(input("Entrez un nombre:"))`.

```
1  try:
2      nombre = int(input("Entrez un nombre:"))
3
4  except ValueError:
5
6      print("La valeur entrée n'est pas un nombre.")
7
8  resultat = 10 / nombre
```

⚠️ Dans l'exemple précédent, si ça rentre dans le `except ValueError`, la variable `nombre` ne sera pas définie à la ligne 8 `resultat = 10 / nombre`, il va y avoir un autre type d'exception.

Lorsqu'une ligne de code dépend directement de la précédente, une solution possible serait de gérer plusieurs exceptions dans le même bloc.

```

try:
    nombre = int(input("Entrez un nombre:"))
    resultat = 10 / nombre
except ValueError:
    print("La valeur entrée doit être un nombre entier.")
except ZeroDivisionError:
    print("On ne peut pas diviser par 0.")

```

## Syntaxe try/except/else/finally

```

try:
    nombre = int(input("Entrez un nombre:"))
    resultat = 10 / nombre
except ValueError:
    print("La valeur entrée n'est pas un nombre")
except ZeroDivisionError:
    print("La valeur entrée ne doit pas être 0!")
except:
    print("Une erreur est survenue")
else:
    print(resultat)
finally:
    print("Fin!")

```

- Dans le bloc « `try` », on met le code que l'on veut gérer, donc susceptible de lever une exception.
- Dans le bloc « `except` », on met le code à exécuter en cas d'exception. Il est possible de gérer plusieurs exceptions pour la(les) même(s) instruction(s).
- Dans le bloc « `else` », on met le code à exécuter lorsqu'aucune exception ne survient.
- Dans le bloc « `finally` » est exécuté peu importe ce qui arrive (qu'il y ait une erreur ou non).

**Exemple :**

```

try:
    nombre = int(input("Entrez un nombre:"))
    resultat = 10 / nombre
except ValueError:
    print("La valeur entrée n'est pas un nombre.")
except ZeroDivisionError:
    print("La valeur entrée ne doit pas être 0!")
except:
    print("Une erreur est survenue.")
else:
    print(resultat)
finally:
    print("Fin!")

```

## Pratiques non conseillées

**⚠️** Gérer plusieurs types d'exceptions de la même façon :

```

try:
    nombre = int(input("Entrez un nombre:"))
    resultat = 10 / nombre
except (ValueError, ZeroDivisionError) :
    print("Une erreur a eu lieu.")

```

**⚠️** Il n'est pas obligatoire de mentionner le type d'exception dans `except` mais ceci **est vivement déconseillé** au risque de gérer à tort tous les types d'exceptions de la même façon.

```

try:
    nombre = int(input("Entrez un nombre:"))
    resultat = 10 / nombre
except :
    print("Une erreur a eu lieu.")

```

# Exercices de base - avec solutions

## But

- Se familiariser avec les bases de la gestion des exceptions avant de les appliquer à des énoncés plus complexes.

## Énoncé

Pour chacun des exercices suivants, vous devez :

- **Tester différentes données/entrées** possibles pour vos fonctions de façon à **faire sortir les exceptions**. Exemples :
  - On entre des valeurs de différents types que ce qui est demandé.
  - On entre des valeurs spécifiques qui risquent de compromettre un calcul.
  - On entre autre chose qu'une liste pour une opération sur les listes.
  - On entre une liste vide.
  - etc.
- **Lisez le message d'erreur** attentivement pour le comprendre et vous familiariser avec.
- **Gérez les exceptions ou corrigez les erreurs** selon les bonnes pratiques.

## Exercice 1

Créez une fonction qui permet de calculer et retourner la division entre deux nombres entiers. Les deux nombres doivent être entrés par l'utilisateur dans votre programme principal.

Arrondissez maintenant le résultat de la division à l'aide de la fonction `round`. Voir la description de `round` [ici](#).



**Clin d'œil :** L'endroit où vous le faites peut vous éviter de devoir gérer une exception de plus. Cela dépend aussi de votre solution.

### ▼ Solution

```
def division(nombre1: int, nombre2: int) -> float:  
    """  
        Cette fonction fait la division entre les deux nombres  
        passés en paramètres et retourne le résultat.  
        :param nombre1: Le 1er nombre entier.  
        :param nombre2: Le 2ème nombre entier.  
        :return: Le résultat de division entre nombre1 et nombre2  
    """  
  
    try:  
        resultat_division = round(nombre1 / nombre2)  
        return resultat_division  
  
    except ZeroDivisionError:  
        return None  
  
    except TypeError:  
        print("Les deux nombres doivent être des entiers")  
        return None  
  
    if __name__ == "__main__":  
  
        try:  
            nb1 = int(input("Veuillez entrer le 1er nombre entier :"))  
            nb2 = int(input("Veuillez entrer le 2ème nombre entier"))  
            res_division = round(division(nb1, nb2))  
            if res_division is None :  
                print("Impossible de faire une division par zéro.")  
            else:  
                print(f"Le résultat de la division entre {nb1} et {nb2} est {res_division}")  
  
        except ValueError:  
            print("Les deux nombres doivent être des entiers")
```

## Exercice 2

Créez une fonction qui permute (échange les positions de) deux éléments d'une liste. La liste et les deux indices doivent être donnés à cette fonction.

### ↙ Solution

```
def permutation_elements_liste(liste: list, i: int, j: int):
    """
    Cette fonction permet de permuter 2 éléments d'une
    liste étant donné leurs indices.
    :param liste: La liste sur laquelle on veut faire la permutation.
    :param i: L'indice de l'élément à permuter.
    :param j: L'indice de l'autre élément à permuter.
    :return: La liste avec les deux éléments permutés.
    """

    try:
        liste[i], liste[j] = liste[j], liste[i]
    except TypeError:
        print("""Aucune permutation effectuée.
        L'indice de la liste doit être un nombre entier.
        le premier paramètre de la fonction n'est pas une liste""")
        return None
    except IndexError:
        print("Aucune permutation effectuée. L'élément à l'indice {} n'existe pas".format(i))
        return None
    return liste

if __name__ == "__main__":
    liste = [1, 2, 3, 4, 5, 6]
    print(f"La liste d'origine est {liste}")

    indice1 = 4
    indice2 = 3

    print(f"Permutation des éléments aux indices {indice1} et {indice2}")
    liste_perm = permutation_elements_liste(liste, i=indice1, j=indice2)

    if liste_perm is not None:
        print(f"La liste avec les éléments permutés est {liste_perm}")
```

# Exercice 3

Soit l'exemple suivant :

```
def trier_liste(liste: list) -> None :
    liste.sorted()

if __name__ == "__main__":
    liste_nums = [6, 3, 5, 2, 4, 1]
    trier_liste(liste_nums)
    print(liste_nums)
```

- Exécutez-le et lisez l'erreur.
- Quel est le type d'exception? Qu'est-ce que cela signifie ?
- Que faut-il faire dans ce cas, gérer l'exception ou corriger l'erreur?

▼ Solution

```

# ***** Réponses aux questions de l'exercice ****
# Exécutez-le et lisez l'erreur.
#         AttributeError: 'list' object has no attribute 'sorted'
# Quel est le type d'exception? Qu'est-ce que cela signifie ?
#         Type d'exception : AttributeError
#         Signification : Une liste n'a pas d'attribut (ici il s'agit
#                           Fonction native proposée : sort()).
# Que faut-il faire dans ce cas, gérer l'exception ou corriger :
#         Mettre liste.sort() à la place de liste.sorted()
# *****

def trier_liste(liste: list) -> None :
    """
    Cette fonction trie la liste passée en paramètres.
    :param liste: liste à trier.
    :return: (Aucun).
    """
    # Correction 2 (en extra) : On gère l'exception où la valeur n'est pas
    try:
        # Correction 1 : sort() à la place de sorted()
        liste.sorted()
        print(liste_nums)
    except AttributeError:
        print(f"{liste_nums} ne peut pas être triée. La valeur n'est pas")
        print("pas une liste.")

    if __name__ == "__main__":
        liste_nums = [8, 7, 1, 5, 4]
        trier_liste(liste_nums)

```

## Exercice 4

```

def multiplication(x, y):
    return COEFICIENT * x * y

if __name__ == "__main__":
    COEFFICIENT = 15
    resultat = multiplication(x = 8, y = 2)
    print(resultat)

```

- Exécutez-le et lisez l'erreur.

- Quel est le type d'exception? Qu'est-ce que cela signifie ?
- Que faut-il faire dans ce cas, gérer l'exception ou corriger l'erreur?

### ▼ Solution

```
# ***** Réponses aux questions de l'exercice ****
# Exécutez-le et lisez l'erreur.
#      NameError: name 'COEFICIENT' is not defined. Did you mean 'COEFFICIENT'?
# Quel est le type d'exception? Qu'est-ce que cela signifie ?
#      Type d'exception : NameError
#      Signification : Le nom COEFICIENT n'est pas connu. Il y a une faute de frappe.
# Que faut-il faire dans ce cas, gérer l'exception ou corriger :
#      Remplacer COEFICIENT par COEFFICIENT
# *****

def multiplication(x: float, y: float) -> float:
    """
    Cette fonction multiplie deux nombres et retourne le résultat.
    :param x: premier nombre à multiplier.
    :param y: deuxième nombre à multiplier.
    :return: le résultat de la multiplication les deux nombres.
    """

    # Correction 2 (en extra) : On gère l'exception où la valeur de x est None
    try:
        # Correction 1 : Correction du nom de la variable COEFFICIENT
        return COEFFICIENT * x * y
    except TypeError:
        print(f"On ne peut pas multiplier {x} par {y}. Les valeurs doivent être des nombres.")

if __name__ == "__main__":
    COEFFICIENT = 15
    resultat = multiplication(x = 8, y = 2)
    if resultat is not None:
        print(resultat)
```

## Exercice 5

Soit le programme suivant, exécutez le et expliquez l'erreur **sans la corriger** (**vous devez utiliser le débogueur pour comprendre l'erreur**).

```
def calcul_moyenne_liste(liste:list[int]=[]) -> float:  
    moyenne = sum(liste) / len(liste)  
    return moyenne  
  
if __name__ == "__main__":  
    valeurs_liste_str = input("Veuillez entrer les valeurs d'une liste sé  
    liste_nombres = valeurs_liste_str.split(",")  
    moyenne = calcul_moyenne_liste(liste_nombres)  
    print(moyenne)
```

Voici la correction du programme précédent (vous n'avez pas besoin de comprendre la correction à la ligne 7 pour faire l'exercice) :

```
1  def calcul_moyenne_liste(liste:list[int]=[]) -> float:  
2      moyenne = sum(liste) / len(liste)  
3      return moyenne  
4  
5  if __name__ == "__main__":  
6      valeurs_liste_str = input("Veuillez entrer les valeurs d'une liste sé  
7      liste_nombres = [int(e) for e in valeurs_liste_str.split(",")]  
8      moyenne = calcul_moyenne_liste(liste_nombres)  
9      print(moyenne)
```

- Gérez les exceptions qui peuvent avoir lieu **dans votre fonction.**

▽ Solution

```
def calcul_moyenne_liste(liste:list[int]) -> float:  
    """  
        Cette fonction calcule la moyenne d'une liste d'entiers.  
        :param liste: liste de nombres entiers.  
        :return: la moyenne des éléments de la liste.  
    """  
  
    try:  
        moyenne = sum(liste) / len(liste)  
  
    except TypeError:  
        print(f"Impossible de calculer la moyenne. La valeur doit être un nombre.")  
        return None  
  
    except ZeroDivisionError:  
        print(f"Impossible de calculer la moyenne. Il doit y avoir au moins un élément dans la liste.")  
  
    else:  
        return moyenne  
  
if __name__ == "__main__":  
    valeurs_liste_str = input("Veuillez entrer des nombres séparés par des virgules : ")  
  
    try:  
        liste_nombres = [int(e) for e in valeurs_liste_str.split(",")]  
  
    except ValueError:  
        print(f"Vous devez entrer des entiers séparés par des virgules.")  
  
    else:  
        moyenne = calcul_moyenne_liste(liste_nombres)  
        if moyenne is not None:  
            print(moyenne)
```

# Laboratoire débogage et gestion des erreurs -solution

## But

- Correction d'erreurs de syntaxe, d'exécution et de logique.
- Utiliser le débogueur et les techniques de débogage.
- Continuer à pratiquer le pseudo-code (algo).

## Énoncé

Vous devez trouver les erreurs dans le code Python de l'enseignant, les réparer et documenter vos découvertes.

Pour chacun des exercices suivants, vous devez :

1. Réfléchir à la logique du programme et écrire un pseudo-code qui décrit les résultats attendus en vous basant sur ce que le programme **est supposé faire** (et non sur le programme plein d'erreurs).
2. Corriger les erreurs de syntaxe.
3. Créer un plan de tests et tester votre programme.

Valeur variable 1	Valeur variable 2	Résultat attendu	Résultat obtenu

4. Corriger le reste des erreurs d'exécution et de logique à l'aide du débogueur.
  - a. Lire les messages d'erreur et corriger les erreurs évidentes sans débogueur.
  - b. Utiliser le débogueur, notamment pour les erreurs de logique ou pour les erreurs d'exécution les moins évidentes. [Voir dans les notes de cours la technique de base et les astuces de débogage.](#)

# Exercice 1 : Retrait d'argent au guichet automatique

Il s'agit d'un programme permettant de retirer de l'argent du guichet automatique. Votre compte débute avec 1000\$ et vous demande d'effectuer 2 retraits.

- Si un retrait est négatif, il ne doit pas avoir lieu.
- Si un retrait est plus grand que le solde du compte, il doit retirer le solde restant seulement.

Le code suivant contient plusieurs erreurs de logique et d'exécution. Vous devez :

- Créer un plan de tests ([selon le tableau dans l'énoncé](#)).
- Identifier les erreurs.
- Corriger les erreurs et gérer les exceptions.

```
def retrait(solde:str = 0, retrait: float = 0) -> bool:
    if retrait > solde:
        retrait = solde
        print(f"* Solde insuffisant, retrait effectué: {retrait} $")

    else:
        solde = solde - retrait
        print(f"* Retrait effectué: {retrait}")

    return solde

if __name__ == "__main__":
    MONTANT_INITIAL = 1000

    solde = MONTANT_INITIAL

    print("*****")
    print("***      GUICHET      ***")
    print("*****")
    print(f"Solde initial: {solde} $")
    print("*****")

    retrait_1 = float(input("Retrait 1: Combien voulez-vous retirer? "))
    solde = retrait(solde, retrait_1)

    retrait_2 = float(input("Retrait 2: Combien voulez-vous retirer? "))
    solde = retrait()

    print("*****")
    print(f"Solde initial: {MONTANT_INITIAL} $")
    print(f"Retrait 1:      {retrait_1} $")
    print(f"Retrait 2:      {retrait_2} $")
    print(f"Solde final:    {solde} $")
    print("*****")
```

## ▽ Solution

### Code Python corrigé :

```
def retrait(solde:float, retrait: float) -> float:
    """
    Fonction qui permet de faire le retrait d'un
    montant du solde.
    :param solde: le solde du compte.
    :param retrait: le montant à retirer.
    :return: le solde restant
    """

    if retrait < 0:
        print("*** Retrait impossible!")

    elif retrait > solde:
        retrait = solde
        solde = 0
        print(f'* Solde insuffisant, retrait effectué: {retrait}')

    else:
        solde = solde - retrait
        print(f'* Retrait effectué: {retrait}')

    return solde

if __name__ == "__main__":
    MONTANT_INITIAL = 1000

    solde = MONTANT_INITIAL

    print("*****")
    print("***      GUICHET      ***")
    print("*****")
    print(f"Solde initial: {solde} $")
    print("*****")

    try:
        retrait_1 = float(input("Retrait 1: Combien voulez-vous"))
        solde = retrait(solde, retrait_1)
    except ValueError:
        print("Erreur. Le montant entré n'est pas une valeur numérique")

    try:
        if solde > 0:
            retrait_2 = float(input("Retrait 2: Combien voulez-vous"))
            solde = retrait(solde, retrait_2)

    except ValueError:
        print("Erreur. Le montant entré n'est pas une valeur numérique")

    print("*****")
```

```
    print(f"Solde initial: {MONTANT_INITIAL} $")\n\n    try:\n        print(f"Retrait 1:      {retrait_1} $")\n    except NameError:\n        pass\n\n    try:\n        print(f"Retrait 2:      {retrait_2} $")\n    except NameError:\n        pass\n\n    print(f"Solde final:     {solde} $")\n    print(f"*****")
```

## Exercice 2 : Roche-Papier-Ciseau

(Inspiré d'un exercice du laboratoire de Rémy Corriveau)

Le code doit afficher un menu et demander aux deux joueurs de choisir leurs armes. Ensuite, le gagnant est affiché ainsi que l'arme victorieuse.

Dans le code suivant, vous trouverez (au moins) :

- 3 erreurs de syntaxe.
- 3 erreurs d'exécution.
- Au moins 4 lignes ayant des erreurs de logique.

Notez que vous aurez besoin d'utiliser la fonction `str.lower()` [qui transforme la chaîne de caractères toute en minuscule](#).

- Une amélioration possible (optionnel).

```

def roche_papier_ciseau(choix_joueur1:str, choix_joueur2:str) -> None:
    """
    Cette fonction compare les choix du jeu Roche-Papier-Ciseaux entre de
    joueurs et affiche le gagnant.

    :param choix_joueur1: Le choix du joueur 1 ("roche", "papier" ou "cis
    :param choix_joueur2: Le choix du joueur 2 ("roche", "papier" ou "cis
    :return: Aucun
    """

    armes = ["roche", "papier", "ciseau"]

    if choix_joueur1 in armes or choix_joueur2 not in armes:
        print("Vous n'avez pas entré roche, papier ou ciseau. :(")
    else:
        if choix_joueur1 == choix_joueur2:
            print("Partie nulle!")
        elif (choix_joueur1 == 'roche' and choix_joueur2 == 'ciseau') or
            joueur_gagnant = "joueur2"
            arme_gagnante = choix_joueur1
        elif choix_joueur1 == 'roche' and choix_joueur2 == 'papier' or ch
            joueur_gagnant = "joueur2"
            arme_gagnante = choix_joueur2

        print(f"\nLe gagnant est {joueur_gagnant} avec {arme_gagnante}, f

if __name__ == "__main__":
    print("""
    Bienvenu au jeu Roche-Papier-Ciseau.
    Veuillez faire un choix parmi les suivants :
    roche
    papier
    ciseau
    """)
    choix1 = input("Choix du joueur 1 : ")
    choix2 = input("Choix du joueur 2 : ")

    roche_papier_ciseau()

```

## Plan de tests (avec 3 exemples)

Choix joueur 1	Choix joueur 2	Résultat attendu	Résultat obtenu
----------------	----------------	------------------	-----------------

roche	ciseau	joueur 1 gagne avec roche.	joueur 2 gagne avec roche
roche	papier	joueur 2 gagne avec papier.	joueur 2 gagne avec papier.
Roche	Papier	joueur 2 gagne avec Papier	Vous n'avez pas entré roche, papier ou ciseau. :(
...			

# Tests unitaires

# Tests unitaires

## Qu'est-ce qu'un test unitaire ?

Les tests unitaires sont des tests automatisés permettant d'assurer le bon fonctionnement de parties spécifiques d'un programme (des fonctions) de manière isolée.

Les tests unitaires peuvent être exécutés automatiquement chaque fois que des modifications sont apportées au code. Cela garantit une validation rapide après chaque modification.

**i** En résumé, un test unitaire est une fonction qui teste une autre fonction. À la place de faire des tests manuellement, on les automatise à l'aide de fonctions à exécuter.

## Qu'est-ce qu'on veut vérifier dans un test unitaire?

Dans ce cours, le minimum à vérifier serait les points suivants :

- Vérifier la valeur et le type de retour d'une fonction.
- Vérifier le contenu d'une liste (valeur, type, nombre d'éléments).
- Vérifier les cas limites (ex. dépassement de capacité, division par zéro, etc.).

D'autres vérifications existent pour assurer le fonctionnement parfait d'une fonction. Vous aurez l'occasion de les découvrir au fil des exercices (si l'occasion se présente) et dans vos prochains cours de programmation.

# Structure d'une fonction de test unitaire : AAA

La méthodologie utilisée pour structurer un test unitaire est AAA pour "Arrange", "Act", et "Assert". Elle est souvent utilisée pour structurer les tests unitaires.

## 1. Arrange (Préparer) :

Mettre en place toutes les conditions préalables nécessaires pour exécuter le test.

- Préparation et l'initialisation des données.
- Configuration de l'environnement.
- Instanciation d'objets.
- etc.

## 2. Act (Agir) :

Exécuter l'action ou le comportement que vous souhaitez tester.

- Appel de la fonction à tester,
- manipulation d'objets,
- ou toute autre opération qu'on veut évaluer.

## 3. Assert (Vérifier) :

Vérifier que le résultat de l'action effectuée dans la phase "Act" est conforme aux attentes.

- Si le résultat est celui attendu, le test passe ; sinon, le test échoue.

# Méthode de test

Les tests unitaires se font à l'aide de modules intégrés ou tiers. Le module que nous allons utiliser est pytest.

## Tests unitaires avec pytest

[Voir les étapes décrites ici pour installer le module pytest.](#)

- Un fichier de test est un fichier python (.py) dont le nom est préfixé par **test\_** suivi du nom du fichier python à tester. Exemple :  
`test_fonctions_a_tester.py` est le fichier dans lequel se trouve les tests unitaires du programme python `fonctions_a_tester.py`.
- Le nom de la fonction de test unitaire doit être préfixée par **test\_**. Par soucis de clarté, on fait suivre le préfixe **test\_** par le nom de la fonction à tester (ou un nom similaire indiquant le test précis).
- Afin d'utiliser certaines fonctionnalités (décorateurs) de pytest, on doit importer pytest: `import pytest`.



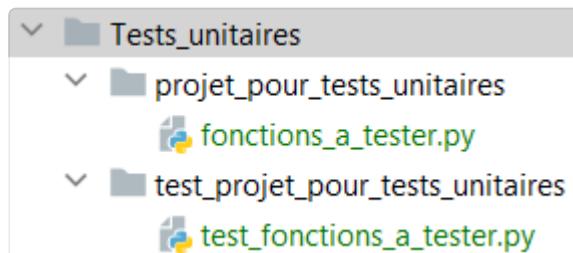
Le nom de la fonction doit obligatoirement commencer par le mot clé `test_`, pour qu'elle soit reconnue comme étant un test unitaire.



#### Bonne pratique :

- On crée un projet de test pour chaque projet python à tester.
- Dans un projet de test, on crée un fichier de test par fichier .py du projet à tester.

Les exemples qui vont suivre sont basés sur la structure suivante contenant le code pour les tests unitaires :



`fonction_a_tester.py` est le fichier du programme dans lequel se trouve les fonctions à tester.

`test_fonction_a_tester.py` est le fichier dans lequel se trouve les fonctions de test du fichier `fonction_a_tester.py`.

## Syntaxe des vérifications assert

```
assert condition[, "message d'erreur optionnel"]
```

- **condition** : la condition qui exprime le résultat attendu est évaluée. Si le résultat est vrai, le test passe ; sinon, le test échoue.
- Le message d'erreur est optionnel.

### Exemples

```
assert resultat == resultat_attendu
assert resultat in liste_resultats_attendus
assert min_resultat_attendu < resultat < max_resultat_attendu
...
```

## Exemple d'un test unitaire

### Fonction à tester :

fonction\_a\_tester.py

```
def division_entiere_1(n, m):
    return n//m
```

### Exemple d'un test unitaire de base pour la fonction précédente :

[Voir la structure AAA pour plus de détails sur # Arrange # Act et # Assert.](#)

test\_fonction\_a\_tester.py

```

import pytest
import Tests_unitaires.projet_pour_tests_unitaires.fonctions_a_tester as

def test_division_entiere_1():
    # Arrange
    n = 6
    m = 2
    resultat_attendu = 3
    # Act
    resultat_division = fonctions_a_tester.division_entiere_1(n, m)

    # Assert
    assert resultat_division == resultat_attendu

```

- *Arrange* : on définit les données de test `n=6` et `m=2`.
- *Act* : on appelle la fonction `division_entiere_1` (qu'on veut tester).
- *Assert* : on vérifie si le résultat de la division de `n=6` sur `m=2` est bien égal à 3.



**Isolation** : une unité spécifique de code doit être testée **seule** pour s'assurer qu'elle fonctionne indépendamment des autres parties du programme.

- On teste une seule fonction à la fois (une fonction de test teste une seule fonction). Il peut y avoir plusieurs fonctions de test pour une seule fonction à tester.
- On n'appelle pas d'autres fonctions qui ont besoin d'être testées (autre que celle qu'on veut tester).



### Précision :

- Les tests unitaires doivent couvrir divers scénarios de données et vérifier que l'unité testée produit les résultats attendus dans toutes les situations.
- Pour une fonction à tester, on regroupe les tests du même type dans la même fonction de test.

# Les données de test

Avant d'écrire un test unitaire, on doit préparer un plan de tests contenant les données à tester. Ce données sont mentionnées au dessus d'un test unitaire :

## Syntaxe :

```
@pytest.mark.parametrize("p1, p2, ..., resultat_attendu", [
    (d1_p1, d1_p2, ..., resultat_attendu1),
    (d2_p1, d2_p2, ..., resultat_attendu2),
    ...,
    (dN_p1, dN_p2, ..., resultat_attenduN),
])
def test_fonction(p1, p2, ..., resultat_attendu):
    # Instructions qui utilisent les données des
    # paramètres dans les sections Arrange/Act/Assert
```

## Exemple :

```
# Arrange
@pytest.mark.parametrize("n, m, resultat_attendu", [
    (10, 2, 5),
    (-10, 2, -5),
    (3, 2, 1),
    (1000000, 10, 100000),
    (0, 5, 0),
    (1, 3, 0)
])
def test_division_entiere_1(n, m, resultat_attendu):
    # Arrange --> c'est le décorateur @pytest.mark.parametrize au dessus

    # Act
    resultat = fonctions_a_tester.division_entiere_1(n, m)

    # Assert
    assert isinstance(resultat, int)
    assert resultat == resultat_attendu
```

# Vérifications (Assert) minimales à faire (selon le type du résultat retourné)

## Vérifications pour les types simples

La vérification	Syntaxe	Exemple
Vérifier que le résultat retourné par la fonction à tester a le bon type.	<code>assert isinstance(&lt;r&gt;, &lt;type&gt;)</code>	<code>assert isinstance(resultat, int)</code>
Vérifier que le résultat est celui attendu.	<code>assert &lt;condition&gt;</code>	<code>assert resultat == 10 assert resultat in [10, 20, 30] assert min_resultat &lt; resultat &lt; max_resultat</code>

## Vérifications pour les résultats de type liste

La vérification	Exemple
Vérifier que le résultat est bien de type list	<code>assert isinstance(&lt;r&gt;, list)</code>
Vérifier que la longueur de la liste est celle attendue	<p>Pour vérifier l'ajout d'un élément dans une liste.</p> <ul style="list-style-type: none"> <li>La variable <code>longueur_init</code> est la longueur initiale de la liste avant l'ajout de l'élément.</li> <li>La variable <code>resultat</code> contient la liste retournée par l'appel à la fonction.</li> </ul> <pre>longueur_resultat = len(resultat) assert longueur_resultat == longueur_init + 1</pre>
Vérifier que chaque élément de la liste a le bon type.	<code>for e in resultat:     assert isinstance(e, int)</code>

**À l'ajout d'un élément à une liste : vérifier que l'élément ajouté est bien à la bonne place.**

```
for e in resultat:  
    assert isinstance(e, int) or
```

```
assert all(isinstance(e, float)
```

**À la suppression d'un élément dans une liste : L'élément supprimé n'existe plus dans la liste ( `not in` ).**

```
assert element_ajoute in resultat
```

```
assert resultat[position] == ele
```

**À la modification d'un élément dans une liste :** - la modification de l'élément a bien été faite correctement.  
- L'ancien élément avant la modification n'existe plus dans la liste.

```
assert element_supprime not in i
```

```
assert resultat[position] == ele
```

```
assert element_avant_modif not in
```

## Vérifications pour les exceptions

La vérification	Syntaxe	Exemple
<b>Vérifier qu'une exception est générée</b>	<code>@pytest.mark.xfail() def test_fonction()     fonctions_a_tes</code>	<code>@pytest.mark.xfail() def test_fonction()     fonctions_a_tes</code>

## Exemples

fonction\_a\_tester.py

```
def division_entiere(n, m):  
    try:  
        return n//m  
    except ZeroDivisionError:  
        return None
```

```

@pytest.mark.parametrize("a, b, resultat_attendu", [
    (10, 2, 5),
    (-10, 2, -5),
    (3, 2, 1),
    (1000000, 10, 100000),
    (0, 5, 0),
    (1, 3, 0)
])
def test_division_entiere(a, b, resultat_attendu):
    # Arrange
    # Act
    resultat = demo.division_entiere(a, b)

    # Assert
    assert isinstance(resultat, int)
    assert resultat == resultat_attendu

# Ici, l'exception ZeroDivisionError est gérée, ce teste va donc passer.
@pytest.mark.xfail(raises=ZeroDivisionError)
def test_exception_zero_division_entiere():
    resultat = demo.division_entiere(5, 0)
    resultat_attendu = 99999999
    assert resultat == resultat_attendu

# Ici, l'exception TypeError n'est pas gérée, ce teste NE va donc PAS pas
@pytest.mark.xfail(raises=TypeError)
def test_exception_TypeError_division_entiere():
    resultat = demo.division_entiere("10", "5")
    resultat_attendu = None
    assert resultat == resultat_attendu

```

## Exercices guidés

Pour chaque fonction :

- Faites le plan de test : les données d'entrée, le résultat attendu
- Listez les vérifications (assert) à faire.
- Créez votre teste unitaire sur la base des deux points précédents.

### Fonction 1 : generer\_adresses\_courriel

```
def generer_adresses_courriel(prenom:str, nom:str, domaine:str) -> str:
    """
    Fonction qui génère une adresse courriel au format
    [nom].[prenom]@[domaine] à partir d'un prenom, nom et domaine.
    :param prenom: Le prenom de la personne.
    :param nom: Le nom de la personne.
    :param domaine: Le nom de domaine pour l'adresse courriel
    :return: L'adresse courriel au format [nom].[prenom]@[domaine]
            tout en lettres minuscules.
    """

    adresse = f"{prenom.lower()}.{nom.lower()}@{domaine}"

    return adresse
```

## Test unitaire de la fonction generer\_adresses\_courriel

Test unitaire de la fonction generer\_adresses\_courriel

```
# Arrange
@pytest.mark.parametrize("prenom, nom, domaine, resultat_attendu", [
    ("Alice", "Smith", "company.org", "alice.smith@company.org"),
    ("bob", "Johnson", "webmail.net", "bob.johnson@webmail.net"),
    ("Eva", "lee", "personal.info", "eva.lee@personal.info")
    # Ajoutez d'autres cas de test si nécessaire
])
def test_generer_adresses_courriel(prenom, nom, domaine, resultat_attendu
    # Act
    resultat = fonctions_a_tester.generer_adresses_courriel(prenom, nom,
        # Assert
        assert isinstance(resultat, str)
        assert resultat == resultat_attendu
```

## Fonction 2 : ajouter\_tache

[...cet exemple n'est pas très pertinent car la méthode .insert(...) sur les listes est écrite correctement et fait déjà des vérifications mais l'idée est de réfléchir à ce qu'il faudrait vérifier dans un test unitaire...]

```
def ajouter_tache(liste_taches, tache, priorite):
    """
    Ajoute une tâche à une liste de tâches selon sa priorité dans la liste
    :param liste_taches: la liste des tâches existante.
    :param tache: la tâche à ajouter dans la liste.
    :param priorite: La position à laquelle on ajoute la tâche.
    :return: retourne la liste incluant la tâche ajoutée.
    """
    liste_taches.insert(priorite, tache)
    return liste_taches
```

## Tests unitaires de la fonction ajouter\_tache

Test unitaire de la fonction ajouter\_tache

```
# Arrange
@pytest.mark.parametrize("liste_taches, tache, priorite, resultat_attendu",
                        ([["Faire les courses"], "Répondre aux courriels", 0, ["Répondre aux c
                        ([["Travailler sur le projet", "Préparer le dîner", "Lire un livre"],
                          [], "Commencer le rapport", 0, ["Commencer le rapport"]),
                        ([["Réviser pour l'examen"], "Prendre une pause", 1, ["Réviser pour l'
                        # Ajoutez d'autres cas de test si nécessaire
])
def test_ajouter_tache(liste_taches, tache, priorite, resultat_attendu):

    # Arrange
    longueur_init = len(liste_taches)

    # Act
    resultat = fonctions_a_tester.ajouter_tache(liste_taches, tache, prio

    # Assert
    assert len(liste_taches) == longueur_init + 1
    assert isinstance(resultat, list)
    assert all(isinstance(tache, str) for tache in resultat)
    assert resultat == resultat_attendu
    assert resultat[priorite] == tache
```



pytest: helps you write better programs — [pytest documentation](#)

>

# Exercice

[inspiré d'un exercice de Rémy Corriveau]

## But

- Se familiariser avec les tests unitaires.
- Compléter un plan de tests.
- Corriger des bogues trouvés par les tests unitaires.
- Faire des commit réguliers à chaque bogue corrigé.

## Énoncé

*Roche, Papier, Ciseau* est un jeu qui se joue entre deux joueurs. Le programme doit d'abord inviter chaque joueur à entrer son choix d'arme parmi les trois options : *roche*, *papier* ou *ciseau*. Une fois les choix saisis, le programme compare les armes en utilisant les règles suivantes :

- **Roche bat Ciseau**
- **Ciseau bat Papier**
- **Papier bat Roche**

Un code rempli de bogues vous a été fourni.



2KB

roche\_papier\_ciseau.zip

archive

Vous devez les trouver à l'aide des **tests unitaires** et **du débogueur**, puis les corriger en suivant ces instructions :

1. Complétez le tableau suivant afin d'avoir un plan de test complet.

arme 1	arme 2	Résultat attendu	Résultat obtenu
roche	ciseau	"joueur 1"	
...			
ROCHE	ciseau	"joueur 1"	
papier	ciseau	"joueur 2"	
...			
papier	CISEAU	"joueur 2"	
ciseau	ciseau	"nulle"	
...			
feuille	...	None	
...	marteau	None	

1. Créez le(s) test(s) unitaires basés sur votre tableau de test :
  - a. Suivez la démarche AAA (Arrange Act Assert) dans chaque fonction de test.
  - b. Vérifiez le type et comparez les résultats attendus et obtenus.
2. Corrigez les bogues trouvés un à la fois. Assurez-vous d'avoir un message de commit pour chaque bogue corrigé.

# Cryptographie

# Introduction à la cryptographie

Auteur : Bruno Charbonneau (adapté par Dalicia).

## Qu'est-ce que la cryptographie

La cryptographie est une façon de protéger des informations et données à l'aide de différentes méthodes. Ces différentes méthodes ont toutes des forces de protection de l'information différentes.

## Le chiffrement

Le chiffrement des données permet de transformer des données pour qu'elles soient déchiffrables seulement par la personne à qui elles sont destinées.

Exemples de formes de chiffrement :

- Utiliser un langage secret.
- Donner un code secret à un ami pour qu'il puisse lire un message qu'on a codifié.

## Le chiffrement de César

Une méthode de chiffrement utilisée depuis longtemps est le chiffrement de César. Cette méthode porte ce nom car elle a été historiquement utilisée par Jules César pour transmettre des informations à ses généraux afin qu'elles ne puissent être lues que par ces derniers.

### Algorithme

La méthode consiste à décaler toutes les lettres d'un mot ou d'un message d'un certain nombre de caractères dans l'alphabet.

César avait comme méthode de décaler tout de 3 lettres, mais ce chiffrement peut être utilisé avec n'importe quelle valeur de rotation vers la droite. Un algorithme populaire, nommé `ROT13`, consiste en fait à utiliser le chiffrement de César avec une valeur de rotation de 13.

-  Le chiffrement de César est considéré comme un chiffrement relativement faible car avec suffisamment d'essais de rotations on peut retrouver le mot ou message initial.

**Exemple :** Chiffrement de César avec un décalage de 3

Mot en claire "bonjour"

Mot chiffré "erqmrxu"

- b décalé de 3 caractères vers la droite devient e
- o décalé de 3 caractères vers la droite devient r
- et ainsi de suite ...

 **Astuce**

Il peut être plus simple de visualiser l'alphabet comme étant un tout circulaire, c'est à dire qu'après "z" on recommence à "a".

## Le hachage (hash)

Le hachage fait également partie de la cryptographie et consiste à générer une séquence de caractères (appelée hash) pour remplacer les données à chiffrer. Il existe plusieurs algorithmes de hachage.

**Exemple**

Voici le résultat de différents algorithmes de hachage sur la phrase "Hello, World!"

Algorithme de hachage	Hash de "Hello, World!"
md5	65a8e27d8879283831b664bd8b7f0ad4
sha256	dffd6021bb2bd5b0af676290809ec3a53191d d81c7f70a4b28688a362182986f
sha512	374d794a95cdcf8b35993185fef9ba368f16 0d8daf432d08ba9f1ed1e5abe6cc69291e0fa 2fe0006a52570ef18c19def4e617c33ce52ef 0a6e5fbe318cb0387

## Caractéristiques du hachage

- **Différents algorithmes** de hachage produisent **différents résultats** pour les **mêmes données**.
- **Un algo de hachage** produira toujours le **même hash** pour la **même donnée** en entrée.
- Chaque **algorithme** créera un **hash de la même longueur peu importe la taille des données initiales**. Par exemple, la taille d'un hash md5 sera toujours de 32 caractères, que ce soit le hash d'un seul mot ou du texte complet d'un livre.
- **Un hash est irréversible** : on ne peut pas retrouver les données initiales de façon algorithmique.



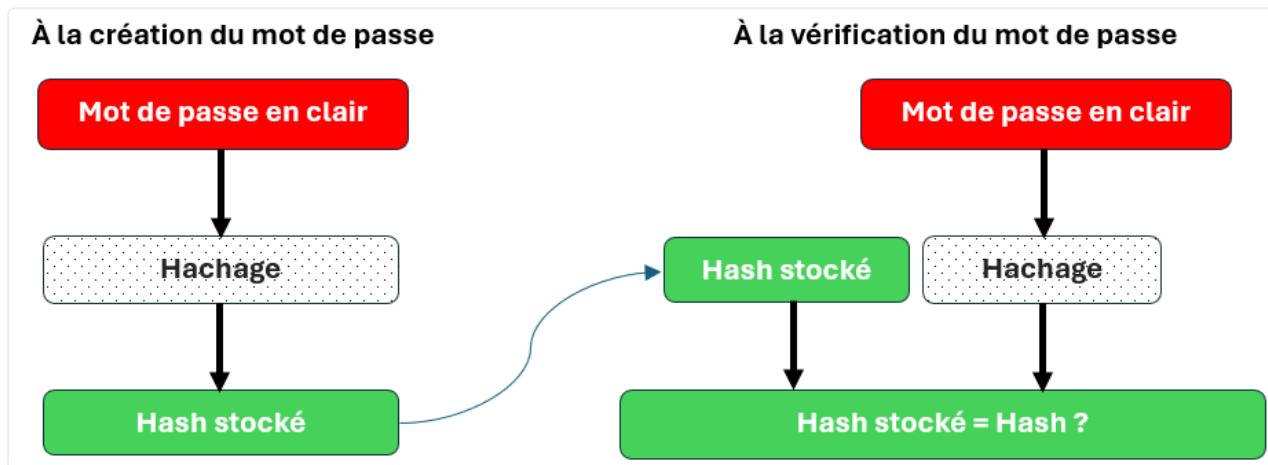
**Collision** : Bien qu'il soit rare, il est possible qu'un algorithme de hachage produise le même hash pour deux entrées différentes. Il s'agit d'une limitation théorique due à la nature finie de l'espace de sortie.

## Exemple d'utilisation : Stocker les mots de passe dans les bases de données

**À la création du mot de passe** : Lorsqu'un utilisateur saisit son mot de passe en créant un compte, ce mot de passe est haché et sauvegardé dans la base de données.

**À la vérification du mot de passe** : Quand l'utilisateur veut ensuite s'authentifier, il saisit à nouveau son mot de passe. Ce dernier est haché puis comparé au hash

stocké dans la base de données. Si les deux hash correspondent, l'utilisateur est alors authentifié.



## Le module hashlib

### [hashlib Python 3.12](#)

Le module `hashlib` est un module intégré au langage python. Il ne requiert pas l'installation d'un module externe et peut être tout simplement importé dans les fichiers python sans manipulation supplémentaire.

Une fois importé, on peut l'utiliser pour faire calculer les hash d'une chaîne de caractères voulue.

**Important :** Le hash se calcule sur les bytes (octet, représentation binaire) de la chaîne de caractères. Il faudra donc en faire la conversion pour que le calcul puisse se faire. Cela peut se faire de différentes façons.

```

# Au moment de la déclaration de la chaîne
chaine_en_bytes = b"Ceci sera converti en bytes"

# Conversion d'une variable
mot = "Bonjour"
mot_en_bytes = mot.encode() # Encodage en bytes (par défaut UTF-8)

```

Pour faire calculer un hash il suffit de spécifier quel type de hash on veut calculer via `hashlib`. Pour obtenir un format relativement lisible pour nous (et non une

chaîne binaire) on utilisera `hexdigest()` pour convertir le tout en un chaîne de caractères hexadécimaux.

```
hash_md5 = hashlib.md5(b"allo").hexdigest()
```

## Lien maths (optionnel)

Si vous souhaitez afficher votre chaîne de caractères en hexadécimal et en binaire (à partir d'un byte), voici le code :

```
chaine_en_bytes = b"Chaine en octets"

# Le format {byte:02X} convertit chaque byte en une chaîne de deux chiffres
print("Représentation hexadécimale : ", ".join(f'{byte:02X}' for byte in

# Le format {byte:08b} convertit chaque byte en une chaîne de 8 bits.
print("Représentation binaire : ", ".join(f'{byte:08b}' for byte in chaine_en_bytes))
```

# Les fichiers

# Accès aux fichiers

## Types de fichiers

Il existe plusieurs types de fichiers. Dans ce cours, nous nous intéresserons aux fichiers textes et CSV.

- **Fichier texte** : Fichier contenant des caractères sans structuration apparente.
- **Fichier CSV** : Fichier contenant des informations séparées par un caractère spécial (point virgule ; , virgule , ou d'autres caractères).

## Étapes d'accès à un fichier

Pour accéder à un fichier, nous avons besoin des trois étapes suivantes :

1. Ouvrir le fichier en mode lecture ou en mode écriture.
2. Lire l'information du fichier ou écrire dans le fichier.
3. Fermer le fichier.

## Chemin d'accès aux fichiers

Séquence de dossiers et sous-dossier pour atteindre un fichier à partir du code.

- **Chemin absolu** : À partir de la racine du disque du système de fichiers (C:\..., D:\..., etc.).  
Exemple : `C:/Application/Data/mon_fichier.txt`
- **Chemin relatif** : À partir de la position du dossier courant.  
Exemple : Si le code qui accède au fichier est dans le répertoire `/home/utilisateur/code.py` et que le fichier à accéder se trouve dans `/home/utilisateur/documents/mon_fichier.txt`, le chemin relatif serait `documents/mon_fichier.txt`

## Modes d'ouverture

- "**r**" : Mode lecture seule (read).
- "**w**" : Mode écriture (write).
- "**a**" : Mode ajout au texte existant (append).

## Ouverture et fermeture d'un fichier : méthode **open/close**

Syntaxe ouverture : `var_objet_fichier = open(nom_fichier, mode_ouverture)`

Syntaxe fermeture : `var_objet_fichier.close()`

Mode d'ouverture : Il existe trois modes d'ouverture pour un fichier :



Peu importe le mode d'ouverture, le fichier doit être fermé après utilisation car elle n'est pas automatique.

Exemple : ouverture fichier en mode lecture

```
mon_fichier = open("Exemple.txt", "r")  
  
# ...Code lecture du fichier ...  
  
mon_fichier.close()
```

## Ouverture et fermeture d'un fichier : méthode **with open**

L'exemple suivant est en mode lecture. Mais la même syntaxe est valable en mode écrire "w" ou en mode ajout "a".

```
with open("Exemple.txt", "r") as fichier:  
    # ... lecture du fichier et autres traitements
```

## Lecture et écriture d'un fichier texte

### Méthodes pour lire un fichier

- `.read()`: Retourne une chaîne contenant tous les caractères du fichier.
- `.read(n)`: Retourne une chaîne contenant les *n* prochains caractères.
- `.readline()`: Retourne une chaîne de tous les caractères jusqu'au prochain retour de ligne (`\n`).
- `.readlines()`: Retourne une liste contenant chacune des lignes du fichier (incluant `\n`) comme éléments de la liste.
- `.read().splitlines()`: Retourne une liste contenant chacune des lignes du fichier (excluant `\n`) comme éléments de la liste.

**i** À chaque lecture, le curseur se déplace. La prochaine lecture se fera à partir de la nouvelle position de ce curseur.

### Méthodes pour écrire dans un fichier

- `.write(<str>)` : Écrit une chaîne de caractères dans le fichier.
- `.writelines(<list>)` : Écrit une liste de chaîne de caractères dans le fichier.

**i**

- Le retour de ligne (`\n`) n'est jamais inclus. Il faut donc prévoir de l'ajouter en cas de besoin.
- En mode écriture, le fichier sera créé s'il n'existe pas. ATTENTION! S'il existe, il sera écrasé.
- En mode ajout, le fichier sera créé s'il n'existe pas. S'il existe, l'ajout se fera à la fin du fichier.

# Exercice

En complément au laboratoire crypto varié, sauvegardez les mots de passes avec leurs hash respectifs dans un fichier txt ou csv (au choix).

Crypto variée

>

# Résolution de problèmes

# Résolution de problèmes : exercice introductif

## Exercice introductif

Nicolas invite Rachel au cinéma. Le film est jeudi prochain à 19h30 et une entrée au prix courant coûte 12 \$. S'il achète les billets en pair, il va faire une économie de 2 \$ sur le total après taxe (15%), mais pour pouvoir faire ça il doit aller les acheter sur place et arriver jeudi avant 19h00 pour s'assurer qu'il reste des billets. Mon boy Nick va évidemment payer pour sa chérie. Par conscience écologique Nicolas n'a pas de voiture. Il peut y aller aujourd'hui en autobus pour s'assurer du meilleur tarif par contre son billet d'autobus lui coûtera 2,50 \$. Nous sommes présentement mardi. Jeudi soir il va prendre un Über pour se rendre au cinéma en passant par la demeure de Rachel. Le taux de Über est de 0,25 \$ plus cher par kilomètre avant 19h00 puisque c'est en période de pointe. Le trajet aller-retour est d'une distance de 12 kilomètres et lui prendra 20 minutes. Quel stratégie doit-il adopter pour payer le plus bas tarif et Nicolas peut-il leur acheter un popcorn ?

## Question

Créez un programme python qui résout ce problème.

## Résolution du problème



Étapes d'analyse et de résolution de problèmes

420-1G2-HU Logique de programmation



# Étapes d'analyse et de résolution de problèmes

## Introduction

Une bonne méthode de travail facilite grandement la tâche de programmation d'une solution informatique efficace et sans erreur, de problèmes moindrement complexes. Nous verrons dans ce qui suit les principales étapes de résolution d'un problème.

### ✓ Exercice

Nicolas invite Rachel au cinéma. Le film est jeudi prochain à 19h30 et une entrée au prix courant coûte 12 \$. S'il achète les billets en pair, il va faire une économie de 2 \$ sur le total après taxe (15%), mais pour pouvoir faire ça il doit aller les acheter sur place et arriver jeudi avant 19h00 pour s'assurer qu'il reste des billets. Mon boy Nick va évidemment payer pour sa chérie. Par conscience écologique Nicolas n'a pas de voiture. Il peut y aller aujourd'hui en autobus pour s'assurer du meilleur tarif par contre son billet d'autobus lui coûtera 2,50 \$. Nous sommes présentement mardi. Jeudi soir il va prendre un Über pour se rendre au cinéma en passant par la demeure de Rachel. Le taux de Über est de 0,25 \$ plus cher par kilomètre avant 19h00 puisque c'est en période de pointe. Le trajet aller-retour est d'une distance de 12 kilomètres et lui prendra 20 minutes. Quel stratégie doit-il adopter pour payer le plus bas tarif et Nicolas peut-il leur acheter un popcorn ?

## Ce qui est demandé

- Lisez chacune des prochaines sections.
- Pour chaque section, répondez aux questions en vert.

# Analyse et compréhension du problème

Dans cette première étape, il est essentiel de bien lire et comprendre la demande ou le problème que vous cherchez à résoudre. Il est important de clarifier les exigences et les objectifs du problème et définir les contraintes.

Cette étape peut nécessiter des discussions avec le client (ou l'enseignant) dans le but de collecter des informations pertinentes.

## ▼ Que comprenez-vous du problème de l'exercice introductif ?

Nicolas va au cinéma avec sa chérie. Il a différentes options pour se rendre au cinéma avant le film afin d'économiser 2\$ sur l'achat de deux billets et il doit prendre la meilleure décision qui lui fasse un maximum d'économies.

## ▼ Quel est l'objectif du problème de l'exercice introductif ?

- Quelle stratégie doit-il adopter pour payer le plus bas tarif?
- Est-ce que Nicolas peut acheter du popcorn?

# Algorithmique : l'art de trouver une solution à un problème

L'algorithmique est une discipline de l'informatique qui se consacre à la **conception de solutions** pour des problèmes donnés. Un algorithme est une séquence d'étapes bien définies pour résoudre un problème ou accomplir une tâche spécifique.

C'est ce que nous allons voir dans ce qui suit.

## Décomposition du problème en sous-problèmes

Cela implique de décomposer le problème en éléments plus petits, d'identifier les relations entre eux et de comprendre les différentes variables ou données qui influent sur le problème. L'analyse vous aidera à avoir une vue d'ensemble de la situation.

Une bonne analyse passe souvent par le découpage du problème principal en plusieurs sous-problèmes, notamment lorsque celui est long ou complexe à résoudre.

Cela simplifie le processus de résolution en le rendant plus modulaire. Chaque sous-problème peut être abordé individuellement, ce qui facilite la résolution du problème global.

✓ Découpez le problème de l'exercice introductif en sous-problèmes.

- **Informations sur le prix du cinéma incluant l'économie :** Nicolas invite Rachel au cinéma. Le film est jeudi prochain à 19h30 et une entrée au prix courant coûte 12 \$. S'il achète les billets en pair, il va faire une économie de 2 \$ sur le total après taxe (15%), mais pour pouvoir faire ça il doit aller les acheter sur place et arriver jeudi avant 19h00 pour s'assurer qu'il reste des billets.
- **Informations sur le prix de l'autobus (s'il y va aujourd'hui ou avant jeudi) :** Mon boy Nick va évidemment payer pour sa chérie. Par conscience écologique Nicolas n'a pas de voiture. Il peut y aller aujourd'hui en autobus pour s'assurer du meilleur tarif par contre son billet d'autobus lui coûtera 2,50 \$.
- **Informations sur le tarif extra du Über (pour un départ jeudi avant 19h00) :** Nous sommes présentement mardi. Jeudi soir il va prendre un Über pour se rendre au cinéma en passant par la demeure de Rachel. Le taux de Über est de 0,25 \$ plus cher par kilomètre avant 19h00 puisque c'est en période de pointe.
- **Informations sur le trajet aller-retour :** Le trajet aller-retour est d'une distance de 12 kilomètres et lui prendra 20 minutes.

## Données d'entrée et de sortie (E/S ou I/O)

Identifiez pour chaque problème ou sous-problème :

- Les données d'entrée nécessaires (les informations ou les données que le programme a besoin d'utiliser).
- Les données de sortie attendues (l'information ou les données qu'on cherche et qui sont produites par le programme).

Il s'agit d'information pertinente qui nous aidera à définir les paramètres de fonctionnement de chaque composant de la solution.

✓ Déterminez les données d'entrée et de sortie pour l'exercice introductif

- **Les entrées :**
  - ~~Nombre de personnes : 2~~
  - ~~Prix du billet : 12\$~~
  - ~~Économie s'il achète deux billets sur place : 2\$~~
  - ~~Taxes: 15%~~
  - ~~Prix de l'autobus : 2.5\$ (avant jeudi)~~
  - ~~Tarif extra du Über: 0.25\$/km (jeudi avant 19h00)~~
  - ~~Distance trajet aller-retour : 12km~~
- **Les sorties :**
  - Quelle stratégie doit-il adopter pour payer le plus bas tarif?  
Est-ce moins cher.
  - Est-ce que Nicolas peut acheter du popcorn?
    - Nous n'avons pas suffisamment d'informations pour répondre à cette question.

## Résolution du problème : étapes des opérations

Déterminez l'ordre dans lequel chaque problème ou sous-problème doit être résolu, en tenant compte des dépendances entre eux.

- Le séquencement des opérations est essentiel pour garantir que la solution globale fonctionne correctement.
- Les étapes doivent être les plus simples possibles.
- La solution ne doit contenir que les étapes importantes qui contribuent à la résolution du problème.

Dans le séquencement des opérations, nous allons retrouver les **traitements séquentiels**, les **traitements conditionnels** et les **traitements répétitifs**.

- ▼ Déterminez les étapes de résolution de l'exercice introductif en utilisant vos propres mots.

Quelle stratégie doit-il adopter pour payer le plus bas tarif?

Est-ce que c'est moins cher :

- d'aller en autobus avant jeudi et économiser 2\$ sur les billets?
  - Prix de l'allée en autobus = 2.5\$
- d'aller en Über le jeudi avant 19h00 et économiser 2\$ sur les billets?
  - Distance aller = distance aller-retour / 2 = 12km / 2 = 6km
  - Prix extra de l'aller avec Über = distance aller \* tarif extra Über = 6km \* 0.25\$ = 1.5\$
- de ne pas aller d'avance pour acheter les billets sur place (ne pas économiser 2\$). Pourquoi ce choix : si le coût du déplacement >= l'économie  $\Rightarrow$  On ne fera pas d'économie.
  - économie sur le billet = 2\$

## Présentation de la solution

Une fois que vous avez trouvé des solutions aux sous-problèmes, vous devez les représenter de manière claire et précise. Cela peut inclure la création de

diagrammes, de schémas, d'organigrammes ou d'autres outils de visualisation pour expliquer la logique derrière chaque solution.

Dans ce cours, nous verrons les deux méthodes suivantes :

## Les organigrammes (flowchart)

Un organigramme est une représentation visuelle et schématique d'un processus ou d'une solution à un problème. Il utilise des symboles graphiques et des connexions pour représenter les étapes, les relations ou les fonctions d'un système de manière claire et concise.

### Définition des symboles

- Début ou fin de la solution

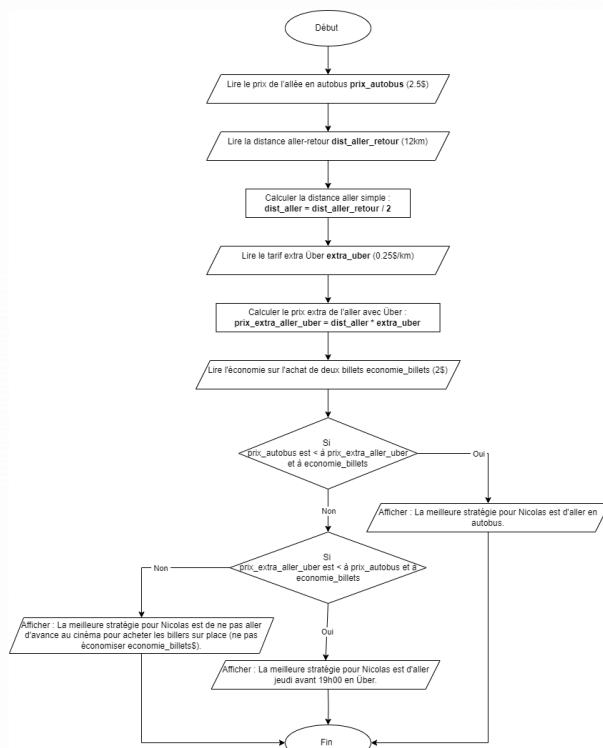
- 
- Une entrée ou une sortie

- 
- Une instruction (une étape du traitement)
- 
- Condition avec branchements selon si la condition est vérifiée ou non (vrai ou faux)

- 
- Connecteur montrant le sens du déroulement de la solution

**Exemple : Faire deviner un nombre à un joueur**

- ✓ Créez l'organigramme correspondant à l'exercice introductif. Vous pouvez utiliser l'outil de votre choix ([suggestion draw.io](#))



## Le pseudo-code

Le pseudo-code est une description d'une solution en langage naturel de haut niveau qui représente la logique de l'algorithme sans se soucier des détails de syntaxe d'un langage de programmation particulier.

### Exemple : Faire deviner un nombre à un joueur

Pseudo code pour faire deviner un nombre à un joueur

```

1  # Choisir un nombre aléatoire à deviner (le programme)
2  # Tant que le joueur n'a pas trouvé le nombre aléatoire
3  #   Le joueur doit entrer une proposition de nombre nb_propose
4  #       Si le nombre aléatoire == nombre proposé alors
5  #           Ecrire le message: Félicitation! Vous avez gagné!
6  #       Sinon
7  #           Retourner à la ligne 2

```

- ✓ Ajoutez le mot clé ***TODO*** à chacune des étapes du pseudo code. À chaque fois que vous avez terminé avec une étape, enlevez le *TODO* (ou tout commentaire quand celui-ci n'est pas pertinent) pour garder un suivi des étapes faites et à faire.

### Exemple avec des TODO :

Pseudo code pour faire deviner un nombre à un joueur (avec TODO)

```

1  # TODO: Choisir un nombre aléatoire à deviner (le programme)
2  # TODO: Tant que le joueur n'a pas trouvé le nombre aléatoire
3  # TODO:   Le joueur doit entrer une proposition de nombre nb_propose
4  # TODO:       Si le nombre aléatoire == nombre proposé alors
5  # TODO:           Ecrire le message: Félicitation! Vous avez gagné!
6  # TODO:       Sinon
7  # TODO:           Retourner à la ligne 2

```

- ✓ Créez le pseudo-code correspondant à l'exercice introductif.

## Implémentation

Une fois que vous avez planifié la solution, vous pouvez passer à l'implémentation. Cela consiste à écrire le code source en utilisant un langage de programmation spécifique.

Vous devez vous assurer de

- suivre les bonnes pratiques de programmation,
- maintenir une structure modulaire et
- documenter correctement votre programme.

✓ Traduisez l'algorithme (organigramme ou pseudo-code) de l'exercice introductif en un programme Python.

## Tests et résultats

Les tests sont une partie cruciale du processus de développement. Ils visent à garantir que la solution fonctionne comme prévu et à identifier les éventuels problèmes ou bogues. Cette étape inclut le débogage du programme avec des outils de débogage.

Il existe plusieurs types de tests, dans ce cours nous verrons les tests suivants :

### Tests manuels

Ce sont des tests réalisés manuellement sur le programme afin de vérifier le bon fonctionnement de la solution. Les tests doivent être variés et complets de façon à s'assurer que le programme fonctionne dans tous les cas possibles. Un plan de tests est requis.

✓ Quels sont les tests que vous devez faire pour tester votre programme ?

### Tests unitaires

Ce sont des tests automatisés qui vérifient le bon fonctionnement des composants individuels de la solution (les fonctions). Les tests doivent être variés et complets

de façon à s'assurer que le programme fonctionne dans tous les cas possibles. Un plan de tests est requis.

Nous verrons les tests unitaires dans quelques semaines.

# Activités avec Légos

# Introduction à la programmation

## Question

Former la lettre/chiffre demandé avec les blocs de construction fournis.

## Instructions

- Équipes de 4 étudiants.
- Une lettre à trouver par équipe.

## Organisation

- 2 étudiants doivent réfléchir à une solution et écrire les instructions en détails sur papier. Ils ont le droit de toucher les pièces individuellement mais pas de construire le chiffre/lettre.
- 1 étudiant va lire les instruction une étape à la fois à l'étudiant qui exécute les étapes. Ne doit pas savoir quel est la lettre/chiffre demandé.
- 1 étudiant va exécuter les étapes et, à la fin, trouver la lettre. Ne doit pas savoir quel est la lettre/chiffre demandé.

# Consolidation sur les fonctions avec des blocs

Cette activité avec des blocs permet de concrétiser le fonctionnement des fonctions.

## Description

Ceci est un exercice à faire à 3 étudiants.

- Un étudiant joue le rôle du programme principal.
- Un 2ème étudiant joue le rôle de la fonction de conversion.
- Un 3ème étudiant joue le rôle de l'utilisateur.

Voici deux bouts de pseudo code qui vous serviront à faire l'exercice. Chacun des étudiants devra remplir son rôle, l'étudiant qui joue le rôle du programme principal doit commencer.

### Rôle: Programme principal

Pseudo code programme principal

Début: programme principal.

Demander à l'utilisateur de fournir un bloc.

Appel à une fonction qui convertit un bloc d'une couleur à une autre.

Récupérer la réponse du serveur.

Fin du programme principal

### Rôle: Fonction de conversion

-----  
Entrées: Le bloc fourni par l'utilisateur  
Sortie: Le bloc de la nouvelle couleur  
-----

Début: fonction qui convertit le bloc d'une couleur à une autre

Si le bloc est rouge alors je renvoie un bloc jaune  
Si le bloc est jaune alors je renvoie un bloc bleu  
Si le bloc est bleu alors je renvoie un bloc vert  
Si le bloc est vert alors je renvoie un bloc rouge

Fin de la fonction

# Pratiquer les boucles avec des blocs

## Instructions

- Mettez-vous en équipes de 3 étudiants.
- Vous aurez besoin d'un crayon et d'une feuille.
- Après avoir créé le pseudo-code, validez le avec le prof, puis programmez le en Python.

### Boucle while

#### Activité 1

Vous avez un ensemble de blocs à votre disposition. On vous demande d'écrire les instructions (sous forme de pseudo-code) pour construire une tour de 5 blocs, un bloc à la fois.

✓ Combien d'instructions avez-vous écrites ? Et si on vous demandait une tour de 100 blocs ?

- Instruction répétitive

Prends un bloc et pose-le  
Prends un bloc et pose-le sur le précédent  
Prends un bloc et pose-le sur le précédent  
Prends un bloc et pose-le sur le précédent  
Prends un bloc et pose-le sur le précédent

- Dans ce cas, on crée une boucle qui répète autant de fois que nécessaire la même instruction (écrite une seule fois).

Répéter 5 fois : Prends un bloc et pose-le sur le précédent

- ✓ Quelle est la question que vous vous posez à chaque fois ?

On compte le nombre de blocs dans la tour et on se demande si on a atteint le nombre de 5 blocs. ⇒ **il s'agit de la condition d'arrêt d'une boucle.**

- ✓ Algorithme (Pseudo-code)

```
Poser le premier bloc.  
Le nombre de blocs est à 1.  
Tant que le nombre de blocs n'a pas atteint 5  
    Incrémenter de 1 le nombre de blocs  
    Poser un bloc par dessus le précédent.
```

En Python, ceci peut-être traduit par la boucle [while](#).

- ✓ Code Python

```
# Le nombre de blocs est à 1 (ceci est appelé une initialisation)  
nombre_blocs = 1  
  
# Poser le premier bloc.  
print(" ____ ")  
print(f" | {nombre_blocs} | ")  
print(" | ____ | ")  
  
# Tant que le nombre de blocs n'a pas atteint 5  
while nombre_blocs < 5:  
    # Incrémenter de 1 le nombre de blocs  
    nombre_blocs += 1  
    # Poser un bloc par dessus le précédent.  
    print(" ____ ")  
    print(f" | {nombre_blocs} | ")  
    print(" | ____ | ")  
  
print("Tour complète avec 5 blocs.")
```

## Activité 2

Faites les même instructions que l'activité 1, mais en alternant les couleurs selon si la position du bloc dans la tour est pair ou impair (choisissez 2 couleurs de votre choix).

### ▼ Pseudo-code

Notez que la solution est un peu différente de la précédente. Maintenant, on initialise le nombre de blocs à 0 et non à 1 (aucun bloc posé à l'extérieur de la boucle).

```
Le nombre de blocs est à 0.  
Tant que le nombre de blocs est < à 5  
    Si la position du bloc est paire, alors  
        Prendre un bloc vert  
    Sinon,  
        Prendre un bloc rouge  
  
    Poser un bloc par dessus le précédent  
    Incrémenter de 1 le nombre de blocs
```

### ▼ Code Python

```
# Séquences ANSI pour les couleurs
RESET = "\033[0m" # Réinitialiser les couleurs
RED = "\033[91m"
GREEN = "\033[92m"

# Solution différente avec le même résultat :
#   On initialise le nombre de blocs à 0
#   Tous les blocs sont ajoutés dans la boucle.
#   Comparez cette solution avec la précédente.

nombre_blocs = 0

while nombre_blocs < 5:
    if nombre_blocs % 2 == 0:
        couleur = GREEN
    else:
        couleur = RED

    print(" ---- ")
    print(f"| {couleur}{nombre_blocs+1} {RESET} |")
    print("| ---- |")

    nombre_blocs += 1

print("La tour est complète avec 5 blocs.")
```

## Boucles `for`

### Activité 3

Vous avez un ensemble de blocs à votre disposition. Disposez les en ligne sur la table (sans les assembler). On vous demande d'écrire les instructions (sous forme de pseudo-code) pour construire une tour, un bloc à la fois.

- ✓ Pseudo-code

Initialiser la variable nombre\_blocs (le nombre total de blocs) :

Pour chaque bloc allant du 1er au dernier (nombre\_blocs) :

- Prendre le prochain bloc de la ligne.
- Ajouter ce bloc au sommet de la tour.

#### ✓ Code en Python

```
nombre_blocs = 10

for i in range(nombre_blocs):
    print(" ____ ")
    print(f" | {i} | ")
    print(" |____| ")

print("La tour est complète avec 10 blocs.")
```

## Activité 4

Faites les même instructions que l'activité 3, mais en alternant les couleurs selon si la position du bloc dans la tour est pair ou impair (choisissez 2 couleurs de votre choix).

#### ✓ Pseudo-code

Initialiser la variable nombre\_blocs (le nombre total de blocs) :

Pour chaque bloc allant du 1er au dernier (nombre\_blocs) :

- Si la position du bloc est paire, alors
  - Prendre un bloc vert
- Sinon,
  - Prendre un bloc bleu

Ajouter le bloc au sommet de la tour.

✓ Code Python

```
# Codes des couleurs à utiliser dans le string
RESET = "\033[0m"
BLEU = "\033[94m"
VERT = "\033[92m"

nombre_blocs = 10

for i in range(nombre_blocs):
    # Déterminer la couleur selon la parité de i
    if i % 2 == 0:
        couleur = VERT # Pour les positions paires
    else:
        couleur = BLEU # Pour les positions impaires

    print(" ____ ")
    print(f" | {couleur}{i} {RESET} | ") # Le numéro du bloc en couleur
    print(" |____| ")

print("La tour est complète avec 10 blocs.")
```

# Laboratoires

# Simulation combat - dragon vs chevalier

## But

- Modifier un code.
- Travailler encore et encore sur les fonctions.
- Utilisation de boucles whiles et for.
- Utilisation de base des listes.
- Déboguer et corriger les erreurs.
- ~~Faire de la gestion d'exceptions.~~
- Introduction aux modules (random).

Durée : 4 heures

## Matériel

Un projet vous a été fourni, il contient :

- Un fichier nommé "***simulation\_combat.py***" dans lequel il y a le **code source que vous devez corriger et compléter**.
- [ ⭐ Optionnel mais conseillé] Un fichier nommé "*test\_simulation\_combat.py*" content quelques tests automatiques pour vérifier vos solutions. **Vous n'avez pas besoin de le modifier, il vous servira à des fins de vérification simplement pour l'instant. On vous montrera comment exécuter les tests unitaires.**
  - Pour utiliser les tests unitaires, vous avez besoin d'installer le module *pytest*. Les étapes sont montrées [ici](#), vous avez le choix entre la méthode en ligne de commande et la méthode dans l'IDE.

Téléchargez et décompressez ce fichier zip, puis copiez-le dans votre dépôt local git *projets-python-1-g-3-[votre\_nom]*.



2KB

Simulation\_combat\_dragon\_chevalier.zip

archive

## Énoncé

On vous demande de programmer une simulation de combat entre un **chevalier** et un **dragon**. Chaque personnage a un certain nombre de **points de vie** (PV) et peut infliger des **dégâts** à son adversaire à chaque tour.



- Un **dégât** représente le nombre de points que la victime (chevalier ou dragon) va perdre.

Voici le programme qui permet de **faire différentes attaques** :

```
# Points de vie initiaux
points_vie_chevalier = 100
points_vie_dragon = 120

# Le chevalier attaque le dragon
degats_chevalier = 20
points_vie_dragon = points_vie_dragon - degats_chevalier
print(f"Le chevalier attaque ! Le dragon perd {degats_chevalier} points de vie")
print(f"Points de vie du dragon : {points_vie_dragon}")

# Le dragon attaque le chevalier
degats_dragon = 25
points_vie_chevalier = points_vie_chevalier - degats_dragon
print(f"Le dragon attaque ! Le chevalier perd {degats_dragon} points de vie")
print(f"Points de vie du chevalier : {points_vie_chevalier}")

# Le chevalier attaque de nouveau le dragon
degats_chevalier = 15
points_vie_dragon = points_vie_dragon - degats_chevalier
print(f"Le chevalier attaque ! Le dragon perd {degats_chevalier} points de vie")
print(f"Points de vie du dragon : {points_vie_dragon}")

# Le dragon attaque de nouveau le chevalier
degats_dragon = 30
points_vie_chevalier = points_vie_chevalier - degats_dragon
print(f"Le dragon attaque ! Le chevalier perd {degats_dragon} points de vie")
print(f"Points de vie du chevalier : {points_vie_chevalier}")
```

## Réutilisabilité

Comme vous avez dû le remarquer, un bloc de code se répète plusieurs fois. Que feriez-vous pour résoudre cette répétition qui rend votre programme disgracieux ?

### ✓ Réponse

Bravo! Il s'agit bien d'une fonction! 🎉

Les fonctions nous permettent la réutilisabilité du code.

Vous devez simplifier ce programme en créant une fonction en suivant le processus suivant :

1. Définissez la signature de la fonction (nom de la fonction et ses paramètres).
2. Complétez le corps de la fonction.
3. Ajoutez le docstring.
4. Exécutez le test unitaire de la fonction avant de la vérifier.

#### ▼ Solution

*[Dans ce qui suit, je vous donne une solution générale au problème. Il est possible que le programme proposé ne soit pas complet, qu'il ne traite pas tous les cas limites, etc. Vous devez vous assurer du bon fonctionnement de votre programme selon tous les critères vus depuis le début de la session (traitement des exceptions, cas limites, validation de données, etc.).]*

- Nom de la fonction "attaquer", voir le docstring pour la description.
- Les paramètres d'entrée de la fonction :
  - **nom\_attaquant** et **nom\_victime** (l'attaquant et la victime peuvent être le dragon ou le chevalier, chacun leurs tous)
  - **degats**.
  - **points\_vie\_victime**.

```
def attaquer(nom_attaquant, nom_victime, degats, points_vie_victime):
    """
    Cette fonction effectue une attaque d'un personnage contre une autre.
    Elle réduit les points de vie de la victime, et affiche le résultat de l'attaque.

    :param nom_attaquant: Le nom de l'attaquant.
    :param nom_victime: Le nom de la victime.
    :param degats: Les points de dégâts infligés à la victime.
    :param points_vie_victime: Les points de vie actuels de la victime.
    :return: Les points de vie de la victime après l'attaque.
    """

    points_vie_victime = points_vie_victime - degats
    print(f"{nom_attaquant} attaque ! {nom_victime} perd {degats} points de vie")
    print(f"Points de vie du {nom_victime} : {points_vie_victime}")
    return points_vie_victime

if __name__ == "__main__":
    # Points de vie initiaux
    points_vie_chevalier = 100
    points_vie_dragon = 120

    points_vie_dragon = attaquer("chevalier", "dragon", 20, points_vie_dragon)
    points_vie_chevalier = attaquer("dragon", "chevalier", 25, points_vie_chevalier)
    points_vie_dragon = attaquer("chevalier", "dragon", 15, points_vie_dragon)
    points_vie_chevalier = attaquer("dragon", "chevalier", 30, points_vie_chevalier)
```

## Fonctionnalité : Plusieurs attaques successives

Nous pourrions aussi imaginer un scénario où un **attaquant réalise plusieurs attaques successives** contre une victime. Lorsque les points de vie sont épuisés, la victime est vaincue.

Quelle est la structure de données que vous utiliserez pour avoir plusieurs attaques successives?

### ▼ Réponse

Une liste de dégâts.

- Modifiez la signature de votre fonction de façon à **prendre en paramètre une liste de dégâts à la place d'un seul dégât.**
  - Vous pouvez aussi la copier, lui changer de nom et faire le reste des ajustements.
- Faites les ajustement nécessaires dans la fonction. Un petit bout de pseudo-code vous a été fourni, complétez le pseudo-code puis implémentez le code Python.

```
#POUR CHAQUE degat DANS liste_degats FAIRE
#    Réduire points_vie_victime de la valeur de degats.
#    <<... Compléter le pseudo-code ....>>
#
##Fin pour
```

- N'oubliez pas d'ajuster la description du docstring de la fonction.

#### ▼ Solution

*[Dans ce qui suit, je vous donne une solution générale au problème. Il est possible que le programme proposé ne soit pas complet, qu'il ne traite pas tous les cas limites, etc. Vous devez vous assurer du bon fonctionnement de votre programme selon tous les critères vus depuis le début de la session (traitement des exceptions, cas limites, validation de données, etc.).]*

```

def attaquer(nom_attaquant:str, nom_victime:str, degats:list, points_vie_victime:int):
    """
    Cette fonction effectue plusieurs attaques successives d'un attaquant sur une victime.
    Elle réduit les points de vie de la victime, et affiche le résultat à chaque attaque.

    :param nom_attaquant: Le nom de l'attaquant.
    :param nom_victime: Le nom de la victime.
    :param degats: Liste de points de dégâts infligés à la victime.
    :param points_vie_victime: Les points de vie actuels de la victime.
    :return: Les points de vie de la victime après l'attaque.
    """

    for degat in degats:
        points_vie_victime -= degat
        print(f"{nom_attaquant} attaque ! {nom_victime} perd {degat} points de vie")
        print(f"Points de vie du {nom_victime} : {points_vie_victime}")
        if points_vie_victime <= 0:
            print(f"{nom_victime} est vaincu !")
            break

    return points_vie_victime

if __name__ == "__main__":
    # Points de vie initiaux
    points_vie_chevalier = 100
    points_vie_dragon = 120

    liste_degats = [40, 80, 20]
    points_vie_dragon = attaquer("chevalier", "dragon", liste_degats)

```

## Fonctionnalité : attaques à tour de rôle

Vous devez mettre à jour votre programme afin d'avoir un **combat à tour de rôle** où l'utilisateur spécifie à chaque tour quel personnage est l'attaquant. La **combat s'arrête lorsque l'un des deux est vaincu** et on doit **afficher qui est vaincu**.

Lorsque l'utilisateur se trompe d'attaquant (ni chevalier, ni dragon), on doit le mettre au courant en lui affichant un message.

ⓘ Pour l'instant, définissez deux listes différentes de dégâts pour chaque attaquant. Nous ajouterons la fonctionnalité de génération aléatoire de la liste de dégâts plus tard.

✓ Il est possible que vous ayez à appeler la fonction qui permet de faire l'attaque (`attaquer`) dans votre nouvelle fonction 😊.

ⓘ Peut vous servir dans votre programme :

- Pour transformer une chaîne de caractères `ma_chaine` entièrement en minuscules `ma_chaine.lower()`
- Pour supprimer les espaces qui entourent une chaîne de caractères `ma_chaine` : `ma_chaine.strip()`
- Pour faire les deux `ma_chaine.strip().lower()`

✓ Questions à se poser (indices)

- Ai-je besoin d'une nouvelle fonction? Si oui, quel serait son nom? ses paramètres ?
- Comment faire pour avoir plusieurs tours durant le combat ? Quelle est la structure logique approprié pour le faire ?
- À quel moment s'arrête le combat ? Comment interpréter "La combat s'arrête lorsqu'un des deux est vaincu" en condition d'arrêt ?

✓ Solution

*[Dans ce qui suit, je vous donne une solution générale au problème. Il est possible que le programme proposé ne soit pas complet, qu'il ne traite pas tous les cas limites, etc. Vous devez vous assurer du bon fonctionnement de votre programme selon tous les critères vus depuis le début de la session (traitement des exceptions, cas limites, validation de données, etc.).]*

Ici, je montre uniquement la nouvelle fonction créée et le programme principal qui a changé :

```
def combat_tour_par_tour(points_vie_chevalier: int, points_vie_dragon: int):
    """
    Fonction qui permet de simuler un combat à tour de rôle entre deux personnages.
    L'utilisateur spécifie qui attaque à chaque tour, et le combat continue jusqu'à ce que l'un des deux personnages soit vaincu.

    :param points_vie_chevalier: Points de vie initiaux du chevalier
    :param points_vie_dragon: Points de vie initiaux du dragon.
    """

    while points_vie_chevalier > 0 and points_vie_dragon > 0:
        attaquant = input("Qui doit attaquer ? (chevalier/dragon) : ")

        if attaquant == "chevalier":
            liste_degats_chevalier = [10, 20, 30]
            points_vie_dragon = attaquer("chevalier", "dragon", liste_degats_chevalier)

        elif attaquant == "dragon":
            liste_degats_dragon = [15, 25, 35]
            points_vie_chevalier = attaquer("dragon", "chevalier", liste_degats_dragon)

        else:
            print("Saisie invalide, veuillez entrer 'chevalier' ou 'dragon'")

        if points_vie_chevalier <= 0:
            print("Le chevalier est vaincu !")
        elif points_vie_dragon <= 0:
            print("Le dragon est vaincu !")

    if __name__ == "__main__":
        # Points de vie initiaux
        pv_chevalier = 100
        pv_dragon = 120

        combat_tour_par_tour(pv_chevalier, pv_dragon)
```

## Débogage

La fonction `evaluer_combat` simule un combat entre un attaquant et une victime en prenant en compte plusieurs facteurs qui influencent l'issue de l'attaque.

- Effectuer des **attaques puissantes** qui infligent à la victime le double des dégâts. Les dégâts sont réduits lorsque la victime à un bouclier (augmentés que de 50%).
- L'attaquant est déclaré trop faible pour continuer lorsque ses points de vie atteignent 30.
- Plusieurs attaques successives sont faites tant que les personnages sont en état de santé.

Votre travail est

- **d'utiliser le débogueur** pour déboguer une fonction qui évalue le résultat d'une attaque,
- d'expliquer brièvement votre raisonnement lorsque vous avez trouvé un bogue à l'aide du débogueur,
- de corriger le bogue.
- *d'archiver et pousser* chacune de vos corrections de bogue dans GitLab avec un beau message de commit.

Points de vie attaquant	Points de vie victime	Attaque puissante	Bouclier victime	Résultat attendu	Résultat obtenu
100	20	True	False	Victoire de l'attaquant	Victoire de l'attaquant
100	20	False	True	Victoire de l'attaquant	Victoire de l'attaquant
20	40	False	False	Défaite de l'attaquant	Défaite de l'attaquant
100	30	True	False	Victoire de l'attaquant	Victoire de l'attaquant

10	40	False	True	Défaite de l'attaquant
----	----	-------	------	------------------------

```

def evaluer_combat(points_vie_attaquant: int, points_vie_victime: int, at
"""
Cette fonction évalue l'issue d'un combat en fonction des points de vie de l'attaquant et de la victime, de l'utilisation d'une attaque puissante et de la présence d'un bouclier.

:param points_vie_attaquant: Les points de vie de l'attaquant.
:param points_vie_victime: Les points de vie de la victime.
:param attaque_puissante: Indique si l'attaquant utilise une attaque puissante.
:param bouclier_victime: Indique si la victime utilise un bouclier pour se protéger.
:return: Un message indiquant le résultat du combat.
"""

while points_vie_attaquant > 0 and points_vie_victime < 0:
    degats = 20

    if attaque_puissante:
        if bouclier_victime:
            degats = degats * 1.5
            print("Attaque puissante mais partiellement bloquée par le bouclier de la victime")
        else:
            degats = degats * 2
            print("Attaque puissante réussie, dégâts doublés !")
    elif bouclier_victime:
        degats = degats // 2
        print("Le bouclier de la victime réduit les dégâts de moitié.")

    if points_vie_attaquant < 10 or points_vie_victime >= 30:
        print("L'attaquant est trop faible pour continuer, le combat est nul")
        return "Défaite de l'attaquant"

    if points_vie_victime <= 0:
        print(f"La victime a {points_vie_victime} points de vie restants")
        return "Victoire de l'attaquant"

    print(f"La victime survit avec {points_vie_victime} points de vie")

```

## Nouvelle fonctionnalité : Potion de soin

[À suivre ...]

## Fonctionnalité : générer aléatoirement la liste de dégâts

Actuellement, dans votre système de combat entre un chevalier et un dragon, les attaques sont réalisées à l'aide de listes de dégâts prédéfinies. Désormais, vous allez ajouter une nouvelle fonctionnalité qui génère automatiquement les dégâts des attaques de manière aléatoire.

[À suivre...]

# Simulateur de bataille - Les anneaux du pouvoir

[Exercice de Rémy Corriveau]

## Énoncé

Vous devez créer un logiciel qui sera utilisé par Galadriel pour estimer ses chances de victoire et les pertes de vies encourues afin de choisir la meilleure stratégie de combat.

Considérant que

- l'armée de Galadriel contient 250 elfes, 150 nains et 500 humains.
- l'armée de Sauron contient 100 trolls, 200 worgs et 2500 orcs.

Une stratégie agressive permet d'éliminer 500 orcs sans rétribution avant même le début de la bataille et une stratégie défensive permet d'obtenir des renforts de 175 nains supplémentaires grâce à une embuscade.

Demandez à Galadriel de choisir sa stratégie et affichez-lui le résultat du combat en termes de victoire ou de défaite.

### Choses à savoir :

- Les elfes et les trolls sont de force équivalente.
- Les nains et les worgs sont de force équivalente.
- Les humains et les orcs sont de force équivalente.
- Un elfe est fort comme 10 humains et un nain est fort comme 5 humains.

### Exigences :

- La création et l'utilisation des fonctions est requise.
- Vous devez utiliser une variable de type liste pour chaque armée.

- Vous devez utiliser une variable de type liste pour les différentes stratégies connues.

## Exemples d'exécution :

### Exemple 1 :

Bienvenu Galadriel, à votre simulateur de combat.

Quelle stratégie voulez-vous adopter [agressive, défensive] ? agressive

Les forces du bien s'inclinent et la terre du milieu sombre dans le chaos

### Exemple 2 :

Bienvenu Galadriel, à votre simulateur de combat.

Quelle stratégie voulez-vous adopter [agressive, défensive] ? défensive

La bataille est remportée et la paix pourra de nouveau régner.

### Plan de tests suggéré

Testez votre programme avec les valeurs suivantes et corrigez si vous constatez des résultats inattendus.

Valeurs d'entrée	Résultat observé
Stratégie : agressive	
Stratégie : défensive	
Stratégie : autre	

# Jeu de cartes

Auteurs : Bruno et Dalicia.

## But

- Boucles imbriquées.
- Listes.
- Recherche et lecture de la documentation officielle (modules random et time).
  - Se servir de la documentation des fonctions dans l'IDE PyCharm.
  - Se servir du site officiel de documentation de Python *Python Docs*.



Les modules | 420-1G2-HU Logique de programmation



## Code source fourni

Programme principal et fonctions à compléter.



2KB

jeu\_cartes.py

Tests unitaires de base pour les fonctions (à exécuter et consulter au besoin).



1KB

test\_jeu\_cartes.py

## Énoncé

Vous devez compléter le code donné ***jeu\_cartes.py*** afin de créer un programme qui simule un jeu de cartes entre deux joueurs.

Le programme principal et les fonctions sont données. Vous devez les compléter en trouvant les bons paramètres aux fonction et en complétant le code source.

Quelques indices sont présents pour vous orienter dans l'utilisation des modules.

### Génération d'un paquet de cartes

On doit générer un paquet de cartes complet à partir d'une ***suite de cartes*** et ***des valeurs de paquets***. Chaque carte sera au format "*valeur de suite*" (ex. : "As de Pique", "2 de Trèfle", ...).

### Sélection de la carte cible

On doit sélectionner aléatoirement une carte du paquet pour définir la ***carte cible*** que les joueurs devront essayer de piocher dans la simulation du jeu.

### Simulation du jeu

Chaque joueur tire des cartes tour à tour d'un paquet mélangé, et le premier qui pioche la carte spécifique désignée comme "***cible***" gagne la partie.

- Le paquet de cartes doit être mélangé avant de commencer la partie.
- La carte pigée ne doit pas être remise dans le paquet de cartes.
- On doit afficher à chaque tour quelle carte a été piochée par quel joueur.
- On souhaite faire des pauses entre les piges pour distinguer les tours des joueurs.

# Lien logique mathématique

# Algèbre de Boole

## But

- Lier les concepts vus en logique mathématique et en logique de programmation et sécurité.
- Renforcer la compréhension des expressions logiques en programmation et des concepts de base de l'Algèbre de Boole.
- Manipuler des fonctions sur les **chaînes de caractères** et d'autres **fonctions intégrées** de Python.
- Utiliser des tuples.

### ✓ Algèbre de Boole vs expressions logique en prog

L'algèbre de Boole fournit la base de la logique conditionnelle en programmation, en utilisant

- des valeurs binaires
- des opérateurs logiques ( ET , OU , NON )

pour structurer les décisions et simplifier les conditions.



Les exemples de ce laboratoire sont tirés d'exercices du cours de logique mathématique fournis par Murray Sylvie.

## Code fourni



2KB

logique\_maths.zip  
archive

# Énoncé

Vous êtes un étudiant et dans votre cours de logique mathématique vous avez vu l'Algèbre de Boole. Vous souhaitez créer un programme qui vous permettra de valider quelques notions comme la tautologie, la contradiction, l'équivalence et l'implication.

# Questions

## Fonctionnalités du programme

- Programmer les différentes opérations (avec des fonctions).
- Utiliser votre programme pour vos vérifications.

Déroulement :

- Deux (2) propositions seront entrées par l'étudiant.
- Transformer les propositions (maths) en expression logique (Python) :
  - On doit d'abord vérifier que l'expression entrée est bien une proposition d'Algèbre de Boole.
- Évaluer l'expression pour toutes les combinaisons de p et q.
  - **Bonus** : afficher la table de vérité de chaque proposition en option dans le programme.
- Pour chaque proposition entrée on dit :
  - S'il s'agit d'une tautologie, une contradiction ou aucun des deux.
- Vérifier si les deux propositions entrées sont équivalentes.
- Vérifier si la première proposition implique la deuxième et inversement.

## Tests unitaires

- Complétez les tests unitaires pour toutes vos fonctions.

- Limitez vous à 2 ou 3 jeux de données (ou plus selon leur pertinence) pour chaque test.

## Contraintes

- Travailler sur le projet fourni.
- On travaillera uniquement avec les propositions ayant p et q.
- Utiliser des tuples pour les valeurs de vérité  $(0, 1)$ .
- Utiliser des fonctions sur les chaînes de caractères pour remplacer des caractères, trouver les caractères alphabétiques, etc. Voir la doc suivante :



Built-in Types — Python 3.10.7 documentation



- Utiliser les fonctions suivantes : `all(liste_bool)` et `eval(exp_str)`. Voir la doc suivante :



Built-in Functions — Python 3.10.2 documentation



## Annexe

### Passage Algèbre de Boole -> Expressions booléennes en prog

	Algèbre de Boole	Expression logiques (booléennes) en prog
Propositions	p, q, r, s	simples variables p, q, r, s
Valeurs de vérité	{0, 1}	valeurs entières 0 et 1 (représentent False et True)

OU	+	or
ET	.	and
Négation	—	not

## Exemples : Proposition (Algèbre Boole) -> Expressions en prog

La barre — sera remplacée par `|` dans le programme.

Proposition (maths)	Proposition dans prog (str)	Expressions booléenne en Python
$\bar{p} \cdot q$	" p . q"	not p and q
$(p + q)$	" (p + q)"	not (p or q)
$(p \cdot q) \cdot (\bar{p} + q)$	"(p . q) .  (p + q)"	(p and q) and not(p ou q)

## Quelques rappels

- **Tautologie** : la proposition **est universellement vraie** dans tous les cas possibles.

$p$	$q$	$p \cdot q$	$\bar{p} \cdot q$	$p + \bar{p} \cdot q$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Peu importe les valeurs de p et q      L'expression est toujours vraie

- **Contradiction** : la proposition ne peut **jamaïs être vraie**, peu importe les valeurs des variables.

$p$	$q$	$\bar{p}$	$p \cdot q$	$\bar{p} \cdot (p \cdot q)$
0	0	1	0	0
0	1	1	0	0
1	0	0	0	0
1	1	0	1	0

Peu importe les valeurs de  $p$  et  $q$

L'expression est toujours fausse

- **Equivalence ( $P \leftrightarrow Q$ )** : Vraie (1) lorsque les deux propositions  $P$  et  $Q$  ont la même valeur de vérité. Faux (0) sinon.

b)

		$p + q$	$\bar{p} + \bar{q}$	$\bar{p}$	$\bar{q}$	$\bar{p} \cdot \bar{q}$	$\bar{p} + \bar{q} \leftrightarrow \bar{p} \cdot \bar{q}$
0	0	0	1	1	1	1	1
0	1	1	0	1	0	0	1
1	0	1	0	0	1	0	1
1	1	1	0	0	0	0	1

Propositions équivalentes

- **Implication ( $P \rightarrow Q$ )** : Tout le temps vraie (1), sauf quand  $P$  est vraie et  $Q$  est fausse.

# Crypto variée

Auteur : Bruno Charbonneau

## But

- Manipuler les algorithmes de hachage avec le module hashlib.
- Manipuler les chaînes de caractères.
- Manipuler les dictionnaires.
- Manipuler du code déjà fourni.

## Code fourni



3KB

cryptographie.zip

archive

## Énoncé

Dans ce laboratoire, vous devrez utiliser vos connaissances en cryptographie afin de découvrir les mots qui se cachent derrière des chaînes de caractères qui ont été chiffrées de différentes façons.

Dans le code déjà fourni vous trouverez une liste de mots `mots_aleatoires` en texte clair. Chacune des chaînes chiffrées correspondent à un de ces mots.

Vous trouverez également les listes `mots_cezar`, `mots_hash`, `mots_cezar_hash`. Ces 3 listes contiennent les mots que vous devrez déchiffrer.

- `mots_cezar` : Cette liste contient les mots chiffrés avec le chiffrement de César.

- `mots_hash` : Cette liste contient des mots qui ont été hachés avec un des algorithmes suivants: md5, sha256, sha512
- `mots_cezar_hash` : Cette liste contient des mots qui ont tout d'abord été chiffrés à l'aide du chiffrement de César, puis hachés avec l'un des 3 algorithmes cités précédemment.

## Tâches

À l'aide du code fourni et de la librairie `hashlib`, vous devrez retrouver les mots qui ont été chiffrés. La fonction **`hasher_mots`** vous permettra de créer un dictionnaire des hash md5, sha256 et sha512 de chacun des mots fourni en texte clair. Ce dictionnaire pourra ensuite vous aider à décoder les mots qui ont été hachés.

La fonction **`chiffrement_cesar`** permet de prendre un mot et de retourner ce mot chiffré avec un certain nombre de rotations selon l'algorithme de César (voir les notes de cours sur la cryptographie).

## Questions

- Vous devez compléter ces deux fonctions permettant de faire le chiffrement.
- Vous devez également créer des fonctions vous permettant de décoder les mots des 3 listes chiffrées.

## Tests unitaires

Vous devez également créer les tests unitaires pour toutes les fonctions créées dans ce programme. Ces tests pourront vous permettre de vous assurer que les différentes méthodes que vous créez pour chiffrer et déchiffrer fonctionnent bien.

Créez un plan de test afin de bien vous assurer de tester différentes possibilités. N'oubliez pas d'inclure des cas qui devraient causer des erreurs afin de vous assurer que votre programme gère bien ces cas également.

# git - système de gestion de versions

# Introduction à Git

## Git

Git est un système de contrôle de version décentralisé. Il permet de suivre et gérer les modifications apportées à un ensemble de dossiers et fichiers au fil du temps. Git conserve un historique complet des modifications, ce qui facilite le suivi des contributions, la résolution de conflits et le retour en arrière en cas de besoin. Les programmeurs peuvent travailler sur des versions différentes d'un projet en parallèle et fusionner leurs modifications de manière cohérente. Il est largement utilisé pour le développement de logiciels, mais peut être utilisé pour tout type de projet nécessitant la gestion de versions.

## GitLab

GitLab est une plateforme de gestion de développement collaboratif basée sur Git.

## Comment vous allez utiliser Git et GitLab

Dans ce cours donné en premier session, il est demandé de travailler sur le même projet Git durant toute la session. Le but étant de vous familiariser avec :

- Le Fork : créer votre propre copie distante d'un dépôt GitLab.
- Le clonage : créer votre copie locale d'un dépôt GitLab distant.
- L'archivage (*commit*) : à faire régulièrement (toutes les 15 à 20 minutes de production de code).
- Pousser les changements (*Push*) : à faire régulièrement juste après chaque commit.
- Au besoin, j'introduirais la commande *Pull* pour récupérer ses changements à distance.

# Étapes de création des dépôts Git

## Création de votre compte GitLab

Allez sur [https://gitlab.com/users/sign\\_in](https://gitlab.com/users/sign_in) et connectez vous à l'aide de votre compte Google.

## Création d'une copie du dépôt

1. Allez dans le dépôt GitLab suivant <https://gitlab.com/420-CO/cours/420-1g3-logique-de-programmation-et-s-curit/projets-python-1g3>
2. Vous devez cliquer sur `Fork` en haut à droite pour créer un nouveau Fork du dépôt. Celui-ci sera votre copie du dépôt.
  - Modifiez le nom du projet par **Projets-Python-1G3- <VotreNom>**.
  - Dans la section "*Project URL*", sélectionnez votre profil.
  - Dans la section "*Branches to include*" (branches à inclure), choisissez "*Only the default branch main*" (uniquement la branche par défaut `main`) .
  - Laissez la visibilité à Privée *Private*.
3. Cliquez sur `Fork Project`.
4. Assurez-vous que votre copie du dépôt **Projets-Python-1G3- <VotreNom>** a bien été créée.

À partir d'ici, vous allez commencer à travailler avec les lignes de commande.

## Installation de Git

- Naviguez à l'endroit dans lequel vous voulez enregistrer votre projet en local via le gestionnaire de fichiers, faites un clic droit et sélectionnez "*Ouvrir dans le terminal*".
- Vérifiez que git est installé en entrant la commande `git --version`.

- Si Git n'est pas installé, téléchargez le ici <https://git-scm.com/downloads> et installez-le. Vérifiez que ça a bien été installé.
- Gardez votre terminal ouvert.

## Clonez un dépôt

1. À partir de votre dépôt **Projets-Python-1G3- <VotreNom>** dans GitLab, cliquez sur **Clone** (bouton bleu à droite) et copiez l'URL (Clone with HTTPS).
2. Dans votre terminal ouvert, entrez la commande  
`git clone <l'URL copiée précédemment> .`
3. Vérifiez via votre explorateur de fichiers que vous avez bien copié localement votre projet Git. C'est dans ce projet que vous allez mettre vos solutions de laboratoires (code source python et autres fichiers en lien avec le code) de façon structurée par semaines et par laboratoire pour le reste de la session.

## Retour dans PyCharm

1. Dans PyCharm, ouvrez le projet que vous venez de cloner.
2. Dans le projet, créez un dossier appelé *Test*.
3. Vous pouvez créer un fichier *test.py* et écrivez dedans `print("Hello world!")`
- .

 **Structure suggérée :**

```
/Semaine_n/
    /Titre_sujet_matiere(optionnel)/
        └── Exercice_n_titre_bref.py
        └── laboratoire_titre_bref.py
```

**Exemple :**

```
/Semaine_06/
    /Retour_sur_quelques_exercices/
        ├── Exercice_02_lab_boucles.py
        ├── Exercice_01_debogage_gestion_exceptions.py
        └── ...
    └── laboratoire_simulation_combat.py
/Semaine_07/
    /Boucles_Conditions/
        ├── Exercice_02.py
        └── Exercice_05.py
/Semaine_08/
    ├── Exercice_01_titre.py
    └── Exercice_02.py
└── ...
```

## Lignes de commandes dans PyCharm

1. Dans PyCharm, ouvrez l'onglet *Terminal* en bas pour ouvrir le terminal (il s'agit de la même chose que le terminal ouvert précédemment, mais ici, on veut travailler dans PyCharm).
2. Les commandes suivantes vont vous permettre de mettre à jour votre projet GitLab distant avec vos modifications dans le projet local :
  - Vérifier l'état des fichiers du projet

```
git status
```
  - Suivi des fichiers (*tracking*)
    - Pour ajouter un fichier ou dossier <nom\_fichier/nom\_dossier>

```
git add <nom_fichier/nom_dossier>
```
    - Pour ajouter tous les fichiers du dossier courant

```
git add .
```
  - Archivage des modifications (*commit*)

```
git commit -m "Message du commit"
```

- Pousser les changements locaux vers le dépôt distant (*push*)

```
git push <remote> <branch>
```

Exemple:

```
git push origin main
```

- Si vous faites d'autres changements dans un fichier et que vous souhaitez les voir (vous pouvez remonter ou descendre à travers les lignes, ou appuyer sur *q* pour quitter le mode diff.)

```
git diff chemin/vers/fichier
```

Pour plus de commandes et plus de détails sur les options des commandes, consultez la documentation Git :



Git - Book



# Classes inversées

# Résolution de problèmes, fonctions et traitements conditionnels

## But

- Papier crayon à l'honneur pour la résolution de problèmes.
- Représentation de la solution avec pseudo-codes et organigrammes.
- Retour sur les fonctions.
- Traitements conditionnels (if/elif/else).

## Énoncé

On essaie de déterminer la catégorie d'une personne et ce que cette personne peut faire dans la société en raison de son âge selon certains critères.

Les critères sont :

- Droit de vote : avoir au moins 18 ans
- Être à l'école : avoir entre 5 et moins de 50 ans
- Pouvoir conduire : avoir 16 ans et plus
- Avoir un prix spécial pour la STO: avoir moins de 18 ans ou 65 ans et plus
- Avoir sa carte de l'âge d'or: avoir 50 ans ou plus
- Notez qu'il est possible d'avoir une fraction d'âge, c'est-à-dire 13.5 ans!

On considère qu'une personne est un ou une :

- adulte, si son âge dépasse 18 ans,
- adolescent.e, si son âge se situe entre 12 et 18 ans et
- enfant, si son âge ne dépasse pas 11 ans.

Votre programme doit donner la possibilité à l'utilisateur d'entrer le nom, le prénom et l'année de naissance d'une personne et, ensuite, de pouvoir afficher, selon son choix, l'âge et la catégorie de cette personne, ce qu'il peut faire en société, ou les deux (3 choix).

Un affichage propre et aligné sera important pour bien voir les réponses.

Voir [l'annexe](#) pour un aperçu du résultat d'exécution d'un tel programme.

---

## Résolution

Avant de commencer à écrire un code source, il est primordial de réfléchir et de concevoir une solution par écrit, via des diagrammes (organigramme) ou un pseudo-code.

[Notes de cours sur la résolution de problèmes](#) (ne faites pas l'exercice qui est dans ces notes de cours).

- Découpage du problème en sous-problèmes → Identification des fonctions
  - **Question à se poser :** Quelles sont les parties distinctes du problème qu'on peut traiter, chacune, indépendamment des autres parties ?
  - Chacune des parties correspond à une fonction.
- Quelles sont les entrées sorties pour chaque sous-problème?
- La résolution de chaque sous-problème se fait dans les étapes suivantes.

### ✓ Solution : Découpage en sous-problèmes (fonctions) et entrées sorties

**Question à se poser :** Quelles sont les parties distinctes du problème qu'on peut traiter, chacune, indépendamment des autres parties ?

- **Fonctions :**
  - Calculer l'âge d'une personne.
  - Trouver la catégorie d'une personne selon son âge.

- Afficher les droits et privilèges d'une personne selon son âge.

- **Programme principal :**

- Lire les entrées de l'utilisateur
- Traiter les choix.
- Afficher les résultats selon le choix.

Quelles sont les entrées (paramètres) sorties (valeurs de retours) pour chaque sous-problème?

- **Calculer l'âge d'une personne :**

**Paramètre :** année de naissance.

**Valeur de retour :** âge .

- **Trouver la catégorie d'une personne selon son âge :**

**Paramètre :** âge.

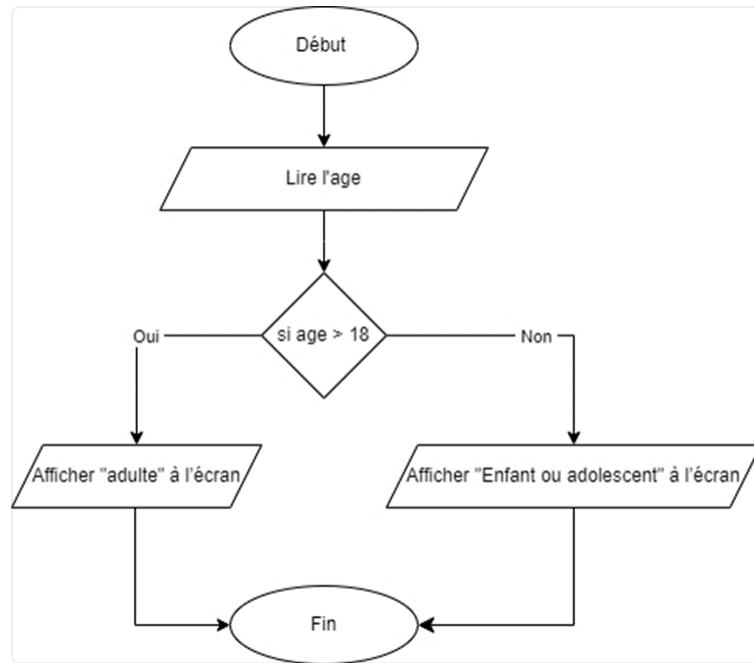
**Valeur de retour :** catégorie.

- **Afficher les droits et privilèges d'une personne selon son âge :**

**Paramètre :** âge.

**Valeur de retour :** aucun, la fonction doit afficher elle-même les droits et privilèges.

Une solution **incomplète** d'un sous-problème vous a été fournie à travers un organigramme et un pseudo-code ainsi qu'un programme en python. Voir les notes de cours pour plus de détails sur les méthodes de [présentation d'une solution](#).



Organigramme pour trouver la catégorie d'une personne selon son âge

Pseudo code pour la fonction qui permet de trouver la catégorie d'une personne selon l'âge.

```

1  # **** Entrées/sorties ****,
2  # Entrées (paramètres) : age
3  # Sorties (retour) : Une chaîne de caractères indiquant la catégorie
4  #           personne "adulte" ou "adolescent"
5  # **** Début pseudo-code ****,
6  # Début : fonction qui détermine la catégorie
7  #     Si l'age est > 18 alors
8  #         categorie = "adulte"
9  #     Sinon,
10 #         categorie = "adolescent"
11 #     retourner la categorie
12 # Fin
    
```

```
1 def categorie_age():
2     """
3         Fonction qui retourne une chaîne de caractères indiquant s'il
4         s'agit d'un adulte ou non selon l'âge entré en paramètre.
5         :param age: l'age de la personne (entier positif)
6         :return: Un string indiquant la catégorie de la personne.
7             Valeurs attendues : "Adulte" ou "Enfant ou Adolescent"
8     """
9     if age > 18:
10         categorie = "adulte"
11     else:
12         categorie = "adolescent"
13
14     return categorie
15
16 if __name__ == "__main__":
17     nom = input("Entrez un prénom : ")
18     prenom = input("Entrez un nom : ")
```

## À vos marques ...

1. La fonction `categorie_age` ainsi que le programme principal contiennent des erreurs de syntaxe et d'exécution. Vous devez les corriger afin de faire fonctionner votre programme. Voir les notes de cours sur [les traitements conditionnels](#) au besoin.

### ▼ Solution

**Erreurs syntaxiques :** Il y a des indentations manquantes aux lignes 10, 12 ainsi que la ligne 17.

```
9     if age > 18:
10    categorie = "adulte"
11 else:
12    categorie = "adolescent"
13
14    return categorie
15
16 ▷ if __name__ == "__main__":
17 nom = input("Entrez un prénom : ")
18 prenom = input("Entrez un nom : ")
19
20    if age > 18:
21        categorie = "adulte"
22    else:
23        categorie = "adolescent"
24
25    return categorie
26
27 ▷ if __name__ == "__main__":
28 nom = input("Entrez un prénom : ")
29 prenom = input("Entrez un nom : ")
```



**Erreur d'exécution :** l'âge doit être un paramètre (une entrée) de la fonction.

### Comment le savoir?

- Mentionné à l'étape de résolution de problèmes. On doit donner l'âge comme entrée à la fonction pour qu'elle puisse trouver la catégorie.
- Mentionné dans la documentation (docstring) de la fonction.

```

1 def categorie_age():
2     """
3         Fonction qui retourne une chaîne de caractères indiquant s'il
4         s'agit d'un adulte ou non selon l'âge entré en paramètre.
5         :param age: l'âge de la personne (entier positif)
6         :return: Un string indiquant la catégorie de la personne.
7             Valeurs attendues : "Adulte" ou "Enfant ou Adolescent"
8     """
9     if age > 18:
10         categorie = "adulte"
11     else:
12         categorie = "adolescent"
13
14     return categorie
15
16 if __name__ == "__main__":
17     nom = input("Entrez un prénom : ")
18     prenom = input("Entrez un nom : ")
19
def categorie_age(age):
    """
        Fonction qui retourne une chaîne de caractères indiquant s'il
        s'agit d'un adulte ou non selon l'âge entré en paramètre.
        :param age: l'âge de la personne (entier positif)
        :return: Un string indiquant la catégorie de la personne.
            Valeurs attendues : "Adulte" ou "Enfant ou Adolescent"
    """
    if age > 18:
        categorie = "adulte"
    else:
        categorie = "adolescent"

    return categorie

if __name__ == "__main__":
    nom = input("Entrez un prénom : ")
    prenom = input("Entrez un nom : ")

```



## 1. Testez votre fonction avec les âges : 30, 15, 8. Que constatez-vous?

### ▼ Solution

- Afin de tester la fonction, nous devons d'abord l'appeler dans le programme principal.
- Cette fonction a besoin du paramètre `age` pour fonctionner. On doit donc lui passer les valeurs d'âge 30, 15, 8, un à la fois.
- Étant donné que la fonction retourne une valeur, qui est la catégorie, on doit "l'attraper" dans le programme principal et la mettre dans une variable. Cette variable sera appelée `categorie`.

On doit donc exécuter le programme avec les valeurs demandées :

```
categorie = categorie_age(30)
```

```
categorie = categorie_age(15)
```

```
categorie = categorie_age(8)
```

```

1 def categorie_age(age):
2     """
3         Fonction qui retourne une chaîne de caractères indiquant s'il
4         s'agit d'un adulte ou non selon l'âge entré en paramètre.
5         :param age: l'âge de la personne (entier positif)
6         :return: Un string indiquant la catégorie de la personne.
7             Valeurs attendues : "Adulte" ou "Enfant ou Adolescent"
8
9     if age > 18:
10         categorie = "adulte"
11     else:
12         categorie = "adolescent"
13
14     return categorie
15
16 D> if __name__ == "__main__":
17     nom = input("Entrez un prénom : ")
18     prenom = input("Entrez un nom : ")
19
20     categorie = categorie_age(30)
21
22     print(f"La catégorie pour 30 ans est : {categorie}")
23
24
25
26
27
28

```

- **Réponse :** Il y a une **erreur de logique**. Lorsqu'on a **8 ans**, on est **enfant** et non pas **adolescent**.

```

16 D> if __name__ == "__main__":
17     nom = input("Entrez un prénom : ")
18     prenom = input("Entrez un nom : ")
19
20     categorie = categorie_age(30)
21     print(f"La catégorie pour 30 ans est : {categorie}")
22
23     categorie = categorie_age(15)
24     print(f"La catégorie pour 15 ans est : {categorie}")
25
26     categorie = categorie_age(8)
27     print(f"La catégorie pour 8 ans est : {categorie}")
28

```

Run scratch

```

↑ "G:\Mon Drive\Cours A24\Logique de programmation\Semaine3\ca
↓ Entrez un prénom : Alain
☰ Entrez un nom : Connu
☱ La catégorie pour 30 ans est : adulte
☱ La catégorie pour 15 ans est : adolescent
☱ La catégorie pour 8 ans est : adolescent

```

age	Résultat attendu	Résultat observé
30	adulte	adulte
15	adolescent	adolescent
8	enfant	adolescent

 **Note** : pour l'instant, vous pouvez tester votre fonction **sans** utiliser l'année de naissance.

3. La fonction ne traite pas le cas où l'âge correspond à celui d'un enfant. Il s'agit d'une erreur de logique que vous allez corriger.

- Avant de corriger l'erreur de logique, vous avez besoin de connaître les [structures décisionnelles multiples](#). Prenez le temps de lire les notes de cours.
- Complétez l'organigramme et le pseudo-code ci-dessus afin de prendre en compte le cas où l'âge correspond à celui d'un enfant. (utilisez le papier et crayon pour schématiser votre organigramme et écrire votre pseudo-code).
- Corrigez ensuite votre fonction et testez la. Assurez-vous d'avoir mis à jour la documentation (docstring) de la fonction.

### ▼ Solution

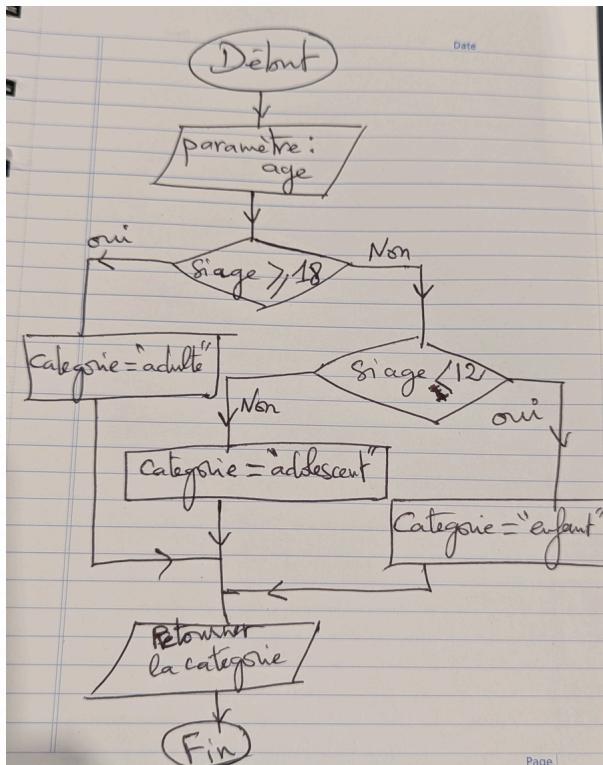
#### Syntaxe structure décisionnelle multiple `if/elif/else`

```
if condition1:  
    # Bloc de code exécuté si condition1 est vraie  
elif condition2:  
    # Bloc de code exécuté si condition1 est fausse et condition2 est vraie  
else:  
    # Bloc de code exécuté si toutes les conditions précédentes sont fausses
```

#### Pseudo-code corrigé

```
# ***** Entrées/sorties *****  
# Entrées (paramètres) : age  
# Sorties (retour) : Une chaîne de caractères indiquant la catégorie  
#                 personne "adulte", "enfant" ou "adolescent"  
# ***** Début pseudo-code *****  
# Début : fonction qui détermine la catégorie  
#         Si l'age est >= 18, alors categorie = "adulte"  
#         Sinon si l'age est < 12, alors categorie = "enfant"  
#         Sinon, categorie = "adolescent"  
#         retourner la catégorie  
# Fin
```

## Organigramme corrigé



### Fonction corrigée :

- Notez que la documentation de la fonction (docstring) a également été corrigée!

```

def categorie_age(age):
    """
    Fonction qui retourne une chaîne de caractères indiquant si
    s'agit d'un adulte, d'un enfant ou d'un adolescent selon l'âge
    entré en paramètre.
    :param age: l'âge de la personne (entier positif).
    :return: Un string indiquant la catégorie de la personne.
    Valeurs attendues : "Adulte", "Enfant" ou "Adolescent"
    """

    if age >= 18:
        categorie = "adulte"
    elif age < 12:
        categorie = "enfant"
    else:
        categorie = "adolescent"

    return categorie
  
```

4. Maintenant, vous allez compléter la résolution du problème. **Pour chacun des sous-problèmes (fonctions) restant(e)s,**
- créez un algorithme (pseudo-code ou organigramme (papier crayon)),
  - traduisez votre algorithme en une fonction Python et
  - testez votre fonction :
    - Créez un plan de test contenant des valeurs variées.
    - Testez votre programme pour confirmer son fonctionnement et relevez les erreurs.

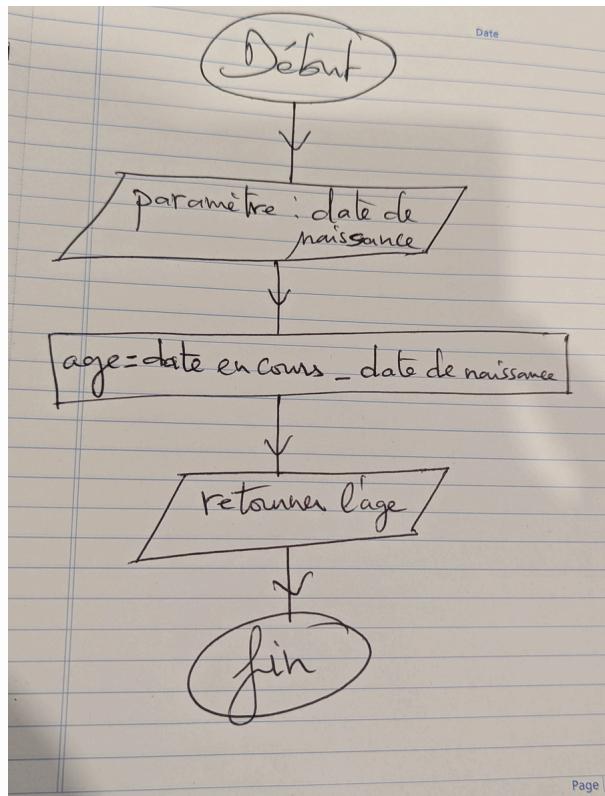
### ▼ Solution

- **Pseudo-code de la fonction `calcul_age` :**

Pseudo-code `calcul_age`

```
***** Entrées/sorties *****
Entrées (paramètres) : année de naissance
Sorties (retour) : l'âge
***** Début pseudo-code *****
Début : fonction qui retourne l'âge
    age = année en cours - année de naissance
    retourner age
Fin
***** Fin pseudo-code *****
```

- **Organigramme de la fonction `calcul_age` :**



- Fonction `calcul_age` en python :

```

def calcul_age(annee_naissance):
    """
    Fonction qui calcule et retourne l'âge.
    :param annee_naissance: année de naissance.
    :return: l'âge.
    """

    annee_courante = 2024
    age = annee_courante - annee_naissance

    return age
  
```

### Appel de la fonction `calcul_age` dans le programme principal

Notez que nous avons besoin de l'année de naissance (variable `annee_naisse`) de la personne afin de calculer son âge.

```

if __name__ == "__main__":
    prenom = input("Entrez un prénom : ")
    nom = input("Entrez un nom : ")

    # Notez que nous avons besoin de l'année de naissance
    # de la personne afin de calculer son age.
    annee_naiss = int(input("Entrez l'année de naissance : "))
    age = calcul_age(annee_naiss)

```

- **Pseudo-code de la fonction `affichage_droits` :**

Pseudo-code `affichage_droits`

```

***** Entrées/sorties *****
Entrées : age
Sorties : Les messages affichants les droits de la personne se
***** Début pseudo-code *****
Début : fonction d'affichage des droits et priviléges
Si age >= 18, afficher "x Droit de vote"
Sinon afficher "- Droit de vote"

Si entre 5 (compris) et 25, afficher "x Être à l'école"
Sinon, afficher "- Être à l'école"

Si age >= 16, afficher "x Pouvoir conduire"
Sinon, afficher "- Pouvoir conduire"

Si age entre 18 et 65 (compris), afficher "x Prix spécial à la
Sinon, afficher "- Prix spécial à la STO"

Si age >= 50, afficher "x Carte de l'âge d'or"
Sinon, afficher "- Carte de l'âge d'or"
Fin
***** Fin pseudo-code *****

```

- **Organigramme de la fonction `affichage_droits` : (ça sera trop grand en organigramme, on se contente du pseudo-code 😊 )**
- **Fonction `affichage_droits` en python :**

```
1 def affichage_droits(age: int):
2     """
3         Fonction qui affiche les droits et les privilèges
4         d'une personne selon son âge.
5         :param age: l'âge de la personne
6         :return: (aucun, le résultat est affiché)
7     """
8     if age >= 18:
9         print("x Droit de vote")
10    else:
11        print("- Droit de vote")
12
13    if 5 <= age < 50:
14        print("x Être à l'école")
15    else:
16        print("- Être à l'école")
17
18    if age >= 16:
19        print("x Pouvoir conduire")
20    else:
21        print("- Pouvoir conduire")
22
23    if age < 18 or age >= 65:
24        print("x Prix spécial à la STO")
25    else:
26        print("- Prix spécial à la STO")
27
28    if age >= 50:
29        print("x Carte de l'âge d'or")
30    else:
31        print("- Carte de l'âge d'or")
```

5. Finalisez le programme principal de façon à permettre de donner à l'utilisateur le choix mentionné dans l'énoncé.

▽ **Solution**

- **Pseudo-code du programme principal**

```

***** Entrées/sorties *****
Entrées : choix de l'utilisateur (A, B, ou C), prénom, nom, an
Sorties : Affichage de l'âge et de la catégorie, des droits de
***** Début pseudo-code *****
Début programme principal

Demander le prénom à l'utilisateur.
Demander le nom à l'utilisateur.
Demander l'année de naissance à l'utilisateur.

Demander (lire) le choix de l'utilisateur (choix) avec le message
"*****"
"Souhaitez-vous :"
    " A. Afficher l'âge et la catégorie de la personne."
    " B. Afficher ce que cette personne peut faire en soc
    " C. Afficher l'âge, la catégorie et ce que cette pers
"*****"

Si choix == 'A' Alors
    Calculer l'âge en appelant la fonction calcul_age
    Appeler la fonction categorie_age pour trouver la catégorie
    Afficher résultat au format "prenom nom est un ou une categ
Sinon Si choix == 'B' Alors
    Afficher résultat au format "prenom nom a les droits suivant
    Appeler la fonction affichage_droits
Sinon Si choix == 'C' Alors
    Appeler la fonction categorie_age pour trouver la catégorie
    Afficher résultat au format "prenom nom est un ou une categ
    Appeler la fonction affichage_droits
Sinon
    Afficher "Cette option (choix) n'existe pas, veuillez choisir
Fin Si
Fin
***** Fin pseudo-code *****

```

- **Programme principal en Python**

Traduisez le pseudo-code en code Python.

## Annexe

Voici un exemple de résultat d'exécution de la totalité du programme :

```
Entrez un prénom : Alain
Entrez un nom : Connu
Entrez l'année de naissance : 1990
*****
Souhaitez-vous :
    A. Afficher l'âge et la catégorie de la personne.
    B. Afficher ce que cette personne peut faire en société.
    C. Afficher l'âge, la catégorie et ce que cette personne peut faire en société.
*****
Réponse (A, B ou C) : C
Alain Connu est un ou une adulte de 34 ans. Il ou elle a les droits suivants :
x Droit de vote
- Être à l'école
x Pouvoir conduire
- Prix spécial à la STO
- Carte de l'âge d'or

Process finished with exit code 0
```

# Révision

# Fonctions, gestion d'erreurs et tests unitaires

## Instructions

Pour chaque énoncé :

1. Créer une fonction qui répond aux exigences
2. Créer 2 tests unitaires pour vérifier le fonctionnement normal
3. Identifier le domaine de valeurs acceptées (ex. quels nombres sont acceptés ?)
4. Ajouter la gestion d'erreur
5. Vérifier à l'aide d'au moins un test unitaire la gestion d'erreur

## Triangle

Écrire une fonction qui prend la longueur des deux cathètes d'un triangle rectangle et qui retourne la longueur de l'hypoténuse d'un triangle rectangle. Voir au besoin :

<https://www.alloprof.qc.ca/fr/eleves/bv/mathematiques/le-theoreme-de-pythagore-m1284>

## Statistiques

Écrire une fonction qui prend une liste de nombres et qui retourne un dictionnaire contenant la somme, la moyenne, le plus grand nombre et le plus petit nombre.

## Étapes suggérées pour l'écriture des fonctions

1. Déterminer les entrées et leur type → deviennent les paramètres de la fonction

2. Déterminer les sorties et leur type → deviennent le retour de la fonction  
(si plusieurs données sont attendues à la sortie, utiliser une collection de données)
3. Écrire la signature de la fonction (nom, paramètres, retour, types)
4. Déterminer l'algorithme de la fonction (pseudo-code et/ou organigramme)
5. Traduire l'algorithme en code et choisir les méthodes appropriées pour ce faire  
(ex. méthodes de Math ou de DateTime)

# Résolution de problème

Temps estimé : environ 4h (sans les parties optionnelles) si vous avez fait l'exercice sur l'inventaire de jeux

## Contexte

Pendant la fin de session, comme vous avez beaucoup de temps libre (!) et un appartement à payer, vous décidez d'acheter et de vendre des voitures miniatures. Comme vous suivez un cours de programmation, vous voulez créer un programme qui vous permet de gérer votre inventaire afin de maximiser votre temps d'étude et vos profits.

## Spécifications du programme

Votre programme doit :

- Utiliser la structure de données utilisée dans `inventaire.json`
- Extraire l'inventaire du fichier `inventaire.json` (voir code fourni) au début du programme
- Permettre l'ajout d'une voiture à l'inventaire (ex. quand vous achetez une nouvelle voiture)
- Permettre la vente d'une voiture et afficher le profit réalisé sur cette vente

Informations à conserver sur les voitures :

- Un numéro unique (voir le format dans fichier .json)
- Modèle
- Couleur
- Année
- Prix d'achat (0 par défaut)
- Prix de vente

- Date d'achat
- Date de vente

## Instructions

1. Séparer l'énoncé en fonctions (nom, responsabilités)
2. Faire un organigramme du programme (les organigrammes sont à l'examen final) - vous pouvez le faire valider en classe
3. Déterminer les structures de données à utiliser
4. Définir les signatures des fonctions
5. Écrire le pseudo-code et/ou faire un organigramme au besoin
6. Transformer le pseudo-code en code

## Exigences supplémentaires

- Utiliser le module DateTime pour la manipulation des dates
- Ajouter de la validation et de la gestion d'erreurs au programme
- Écrire des tests pour au moins 1 fonction comprenant la vérification du fonctionnement normal et la vérification de la gestion d'erreurs

## Pour aller plus loin (optionnel)

- Quand une voiture est vendue, affichez depuis combien de jours elle était dans l'inventaire.
- Ajouter une fonctionnalité de recherche selon [marque, modèle, couleur, prix, vendu ou non, etc] et qui affiche toutes les voitures correspondant au critère désiré.
- Afficher le profit total de toutes les ventes effectuées
- Avant de quitter, mettre à jour l'inventaire dans le fichier inventaire.json

# Code fourni

## Inventaire sous forme de fichier

Créer un nouveau fichier `inventaire.json` dans l'IDE et y copier-coller le contenu

```
{
    "1G6K": {
        "modele": "Ford Econoline E350",
        "couleur": "turquoise",
        "annee": 2021,
        "prix_achat": 22,
        "prix_vente": 65,
        "date_achat": "2023-11-01",
        "date_vente": "2023-11-21"
    },
    "5GAK": {
        "modele": "Tesla Modèle Y",
        "couleur": "bleu ciel",
        "annee": 2020,
        "prix_achat": 0,
        "prix_vente": 17,
        "date_achat": "2022-11-01",
        "date_vente": ""
    },
    "19UU": {
        "modele": "Lotus Esprit",
        "couleur": "jaune néon",
        "annee": 2009,
        "prix_achat": 14,
        "prix_vente": 38,
        "date_achat": "2021-10-20",
        "date_vente": ""
    },
    "5NPE": {
        "modele": "Dodge Charger",
        "couleur": "orange brûlé",
        "annee": 1970,
        "prix_achat": 40,
        "prix_vente": 52,
        "date_achat": "2023-11-01",
        "date_vente": "2023-12-01"
    }
}
```

## Lecture du JSON

```
with open("inventaire.json", "r", encoding="utf-8") as inventaire:
    return json.load(inventaire)
```

## Écriture dans un fichier JSON (partie optionnelle)

```
with open("inventaire.json", "w", encoding="utf-8") as inventaire:  
    json.dump(nouvel_inventaire, inventaire, ensure_ascii=False, indent=4)
```

## Ressources

- Documentation officielle de DateTime:  
<https://docs.python.org/3/library/datetime.html>
- Documentation officielle sur la lecture et l'écriture de fichiers :  
<https://docs.python.org/3.12/library/functions.html#open>
- Documentation officielle sur la manipulation de fichiers JSON:  
<https://docs.python.org/3/library/json.html>

*Note : L'exercice est tiré et adapté des exercices de Rémy Corriveau - merci !*

# Trucs utiles

# Check-list

## Débogage et lecture d'erreurs

- Je sais lire les erreurs (type d'erreur, numéro de ligne, etc.).
- Je sais utiliser le débogueur.

## Structures de données et itérations

- Je sais parcourir une liste simple.
- Je sais parcourir une liste à 2 dimensions (listes imbriquées).
- Je sais parcourir un dictionnaire (clés et valeurs).
- Je sais parcourir une liste de dictionnaires (parcourir la liste, accéder au dictionnaire).

## Manipulation des données

- Je sais ajouter, modifier et supprimer des éléments dans une liste.
- Je sais ajouter, modifier et supprimer des éléments dans un dictionnaire.
- Je sais manipuler une liste dans une liste
- Je sais manipuler un dictionnaire dans une liste

## Conditions et logique

- Je sais créer une instruction : avec des opérateurs, initialiser des variables, lire (input) et écrire (print).
- Je sais comment composer des opérateurs arithmétiques (+, -, \*, %, ...) et logiques (and, or, not) pour former une condition.
- Je sais écrire des conditions simples avec if, elif, else.
- Je sais créer une boucle while
- Je sais créer une boucle for

## Fonctions

- Je sais écrire une fonction avec des paramètres et un retour.

- Je sais appeler des fonctions (programme principal et dans d'autres fonctions)

### **Algorithmes de base**

- Je sais lire un organigramme.
- Je sais écrire un pseudo code ou un organigramme.

### **Tests unitaires**

- Je sais créer un plan de tests qui vérifie une fonction.
- Je sais écrire un teste unitaire.

### **Modules**

- Je connais les fonctions de base vues en classe des modules hashlib, random et datetime.