On découvre

- ce que sont les interactions avec l'utilisateur et les entrées/sorties
- comment écrire des programmes Python et les exécuter dans un terminal

1 Pourquoi des interactions avec l'utilisateur ?

Jusqu'à présent, nous avons écrit des bibliothèques, c'est à dire des suites de fonctions écrites dans un fichier.

Nous avons exécuté les tests de ces fonctions en cliquant sur le bouton `Run Test` de Thonny-l1test, et avons réalisé des appels de fonction en interagissant avec l'interpréteur dans la console de Thonny.

Nous avons donc agi en *programmeur ou programmeuse Python* à l'intérieur de l'environnement de développement Thonny.

Cette semaine nous allons adopter le point de vue de l'*utilisateur ou l'utilisatrice* d'un programme Python.

Thonny est un programme Python, ainsi que son extension L1test. Vous n'en êtes pas les programmeurs. Vous en êtes les utilisateurs :

- vous lancez le programme Thonny-l1test en cliquant sur son icône,
- ou en exécutant la commande thonny-l1test dans un terminal
- puis vous exécutez l1test en cliquant sur le bouton associé.

Vous *interagissez* donc avec ces programmes en cliquant dans une interface graphique.

Cette semaine nous allons apprendre comment programmer des interactions plus rudimentaires :

- afficher un texte à l'écran
- récupérer une valeur entrée au clavier.

2 Entrées et sorties

Quand on s'intéresse aux interactions à base de texte écrit (et non de clic), on utilise les termes suivants :

- le clavier, ou l'*entrée standard*, permet de saisir les entrées du programme (les données d'entrée)
- le terminal ou la console de Thonny permettent d'afficher à l'écran un texte indiquant les *sorties* du programme (le résultat de ses calculs). On parle de *sortie standard*.

Pour afficher un texte, on utilise en Python la fonction `print`.

Pour saisir une valeur au clavier, on utilise en Python la fonction `input`.

Ces instructions seront détaillées plus loin.

3 Programme principal

On souhaite programmer le comportement suivant :

```
Entrer un numéro de jour : 16
Entrer un numéro de mois : 7
Le 16/7 est en été.
```

On identifie :

- des affichages : `Le 16/7 est en été.` (et aussi `Entrer un numéro de jour :`)
- des saisies : du numéro de jour, du numéro de mois
- un calcul : il a bien fallu calculer quelque part que le 16 juillet est en été.

On aura donc un programme `saison.py` qui aura cette structure :

```
# Calcul de la saison

def calcule_saison(num_jour:int, num_mois:int)->str:
    """Renvoie la saison associée au jour dont le numéro de jour
    et de mois sont passés en paramètre.
    [...]
    """
    [...]

def affiche_saison(num_jour:int, num_mois:int, saison:str) -> None:
    """Affiche un message indiquant que le jour et le mois passés
    en paramètre appartiennent à la saison passée en paramètre.
    [...]
    """
    [...]

def saisie_jour() -> int:
    """ Renvoie le numéro d'un jour saisi au clavier après avoir
    affiché le message `Entrer un numéro de jour :`
    [...]
    """
    [...]

def saisie_mois() -> int:
    """ Renvoie le numéro d'un mois saisi au clavier après avoir
    affiché le message `Entrer un numéro de mois :`
    [...]
    """
    [...]
```

On voit bien comment coder la fonction `calcule_saison()`, on verra plus tard comment coder les fonctions `saisie_jour()`, `saisie_mois()` et `affiche_saison()`.

Mais il faut écrire quelque part que lors de l'exécution du programme on veut :

- d'abord appeler la fonction `saisie_jour()` pour saisir un numéro de jour,
- puis appeler la fonction `saisie_mois()` pour saisir un numéro de mois,
- ensuite appeler la fonction `calcule_saison` avec le numéro de jour et le numéro de mois saisis pour calculer la saison,
- enfin appeler la fonction `affiche_saison` pour afficher le résultat du calcul.

Une bonne pratique est de définir, plutôt en fin de ce fichier, une fonction particulière qui fait le liant entre les différentes fonctions du programme. Cette fonction sera par convention nommée `main()` (*fonction principale*), ou `mon_programme_main()`.

Dans notre cas on écrira en fin de fichier :

```
def main() -> None:
    jour = saisie_jour()
    mois = saisie_mois()
    la_saison = calcule_saison(jour, mois)
    affiche_saison(jour, mois, la_saison)
```

Enfin, il reste à indiquer que lors de l'exécution du programme la fonction `main()` doit être appelée. On écrira en toute fin de fichier les lignes suivantes qui sont une convention Python :

```
if __name__ == '__main__':
    main()
```

L'outil `Thonny_l1test` vous permet de générer ces lignes sans avoir à les écrire ni les apprendre par cœur.

Département Informatique de la Faculté des Sciences et Technologies

4 Exécuter un programme depuis Thonny

Pour exécuter un programme dans Thonny, c'est-à-dire exécuter le `main` contenu dans le fichier, on a 2 possibilités.

On peut cliquer sur le bouton `Run Program`[1] (flèche blanche dans un rond vert) et observer le programme se dérouler dans la console de Thonny. Ensuite, une fois le programme terminé, il est possible d'interagir avec l'interpréteur Python comme d'habitude.

On peut aussi ouvrir un terminal en choisissant « *Exécuter le script courant dans un terminal* » du menu « *Exécuter* » de Thonny (ou « *Run current script in terminal* » du menu « *Run* »). Dans ce cas le programme se déroule mais ensuite l'invite est celle du terminal et non celle de Python (voir cours ODI). Vous pouvez donc exécuter une commande type `ls` mais pas interagir avec Python sans lancer un interpréteur.

Dans le cas de l'exemple `saison.py` ci-dessus, avec les 2 méthodes on verra pour commencer s'afficher l'invitation à entrer un numéro de jour.

Dans les 2 cas, les tests contenus dans les fonctions de `saison.py` ne sont pas exécutés. Rien ne se produira dans la fenêtre L1Test.

Inversement, l'exécution des tests via le bouton `Run Test` n'exécute pas le `main`.

5 Memento

- on distingue les rôles de *programmeur* et d'*utilisateur* d'un programme
- les entrées/sorties permettent à un programme d'interagir avec l'utilisateur
- cette interaction textuelle s'effectue lors de l'exécution d'un programme dans un terminal
- un programme principal comporte une fonction «***main***», point d'entrée de l'exécution du programme

[1] Dans la version française ce bouton est nommé "exécuter le script courant" - même si dans l'UE PROG nous n'écrivons pas des scripts mais ds bibliothèques et des programmes.

On découvre

- comment écrire du texte sur le terminal
- que les valeurs de tout type peuvent être écrites sur le terminal

La fonction Python prédéfinie `print()` permet d'afficher du texte, c'est à dire d'écrire sur le terminal.

1 Usage de la fonction `print()`

Vous pouvez lire `help(print)` pour plus de détails !

La fonction s'utilise ainsi :

`print(<val_1>, <val_2>, ..., <val_n>)`

où les paramètres `<val_i>` sont des objets de n'importe quel type.

Les paramètres sont séparés par des virgules.

Le nombre des paramètres est au choix de l'appelant, on peut aussi appeler la fonction `print` sans paramètre :

```
>>> print()
```

2 Écriture selon les types

Dans le cas d'une chaîne de caractères de longueur n, les n caractères sont écrits les uns à la suite des autres. Les caractères `"` ou `'` qui balisent la chaîne ne sont pas écrits.

```
>>> print('Salut à toi !')
Salut à toi !
```

Dans le cas d'un entier les chiffres sont écrits les uns à la suite des autres :

```
>>> print(52)
52
>>> print(-12)
-12
```

Dans le cas d'un flottant Python utilisera la notation simple ou la notation scientifique :

```
>>> from math import pi
>>> print(pi)
3.141592653589793
>>> val = pi * 42
>>> print(val)
7.590924172052281e+20
```

Dans le cas d'un booléen les caractères des littéraux `True` ou `False` sont écrits :

```
>>> print(True)
True
```

3 Caractères spéciaux

Les caractères spéciaux permettent entre autres de représenter dans un littéral de type chaîne des caractères d'espacement comme le retour à la ligne ou la tabulation.

On trouve par exemple les caractères spéciaux suivants :

- `\n` qui permet d'insérer un passage à la ligne sur le terminal
- `\t` qui permet d'insérer une tabulation
- `\"` et `\'` qui permettent d'écrire les guillemets
- `\\` qui permet d'écrire le `\` lui-même.

On notera qu'un caractère spécial est noté sous la forme du caractère \ – antislash ou barre oblique inverse – suivi d'un autre caractère, par exemple `\n`.

Cette barre oblique inverse n'est pas comprise par Python comme un caractère, la barre ne sera pas écrite comme telle. De même pour le caractère qui suit la barre oblique.

Voilà l'affichage qu'en fait la fonction `print` :

```
>>> print("un\ndeux\ntrois\nsoleil")
un
deux
trois
soleil
>>> print("\"entre guillemets\"")
"entre guillemets"
>>> print("\\o/")
\o/
>>> print('***\t***\n***\t***')
***  ***
***  ***
```

4 Évaluation et affichage des paramètres

Comme pour tout appel de fonction, un appel à la fonction `print` commence par l'évaluation de ses paramètres.

Par exemple, pour exécuter `print(5+3)`, Python évalue `5+3` à la valeur `8`, puis affiche `8`.

Quand on utilise plusieurs paramètres, les valeurs des paramètres de `print()` sont écrites les unes à la suite des autres, *séparées par une espace*. De plus un *retour à la ligne* est ajouté à la fin.

Par exemple :

```
>>> la_saison = "été"
>>> print("On est en", la_saison)
On est en été
```

Un appel à `print()` sans paramètre affiche juste un saut de ligne.

Pour produire l'affichage de `Le 16/7 est en été.`, on peut utiliser les chaînes formatées vues en semaine 2.

```
>>> num_jour = 16
>>> num_mois = 7
>>> la_saison = 'été'
>>> print(f"Le {num_jour}/{num_mois} est en {la_saison}.")
Le 16/7 est en été.
```

5 Fonction et procédure

La fonction `print()` est particulière : elle ne renvoie pas de valeur, elle écrit sur le terminal.

Une telle fonction est appelée *procédure*.

En Python, une procédure qui ne renvoie rien renvoie par convention la valeur `None`. On indiquera `None` dans la signature des procédures que nous écrirons.

Le corps de ces procédures ne contient pas d'instruction `return`.

Nous pouvons maintenant écrire la procédure d'affichage du chapitre précédent.

```
def affiche_saison(num_jour:int, num_mois:int, saison:str) -> None:
    """Affiche un message indiquant que le jour et le mois passés
    en paramètre appartiennent à la saison passée en paramètre. Par ex :
    Le 16/7 est en été.
    precondition: num_jour compris entre 1 et 31
                  num_mois compris entre 1 et 12
    """
    print(f"Le {num_jour}/{num_mois} est en {saison}.")
```

Noter l'absence de `return` et la présence de `None` dans la signature.

Les fonctions `main()`, point d'entrée des programmes, sont elles aussi des procédures d'affichage.

6 (Ne pas) tester les affichages avec L1test

L1Test est un outil de test unitaire pour débutant · es. D'une manière générale le test des interactions utilisateur requiert des techniques de test avancées.

6.1 Programme principal

`L1test` ne permet pas de tester un programme principal réalisant des interactions avec l'utilisateur. On se limitera à exécuter le programme à la main dans l'interpréteur de `Thonny-l1test` ou dans un terminal et on vérifiera visuellement que le comportement correspond au comportement attendu.

6.2 Fonctions de calcul

Par contre il est impératif que les fonctions *de calcul* soient testées de manière automatique avec `L1Test`, c'est ce qu'on pratique depuis la semaine 1.

6.3 Procédures d'affichage

`L1test` n'est pas adapté pour tester les procédures d'affichage. La docstring de ces procédures ne contient donc pas de test.

Dans l'exemple précédent la procédure `affiche_saison` n'a pas de test et il est tout à fait normal que l'affichage soit à l'orange dans la fenêtre de test.

On pourrait être tenté d'écrire ce genre de test incorrect :

```
$$$ affiche_saison(16, 7, "été")
Le 16/7 est en été.
```

Ce test est en erreur à cause d'une erreur de syntaxe. En effet `Le 16/7 est en été.` n'est **pas** une expression Python.

On pourrait être tenté d'écrire cet autre test incorrect :

```
$$$ affiche_saison(16, 7, "été")
"Le 16/7 est en été."
```

Cette fois le test est en échec. `L1test` fournit le message suivant : `Expected: "Le 16/7 est en été.", Got : None`. `None` la valeur particulière Python qui est systématiquement renvoyée par les fonctions ne possédant pas d'instruction `return`.

Le seul test qu'on peut écrire avec `L1test` pour une procédure d'affichage sera du type :

```
$$$ affiche_saison(16, 7, "été")
None
```

Il n'est pas très informatif ! C'est pourquoi nous n'écrirons pas de test pour les procédures.

6.4 Test du calcul de la chaîne à afficher

Mais alors, comment faire quand la chaîne de caractères à afficher est issue d'un calcul non trivial ? Par exemple quand on souhaite afficher la grille à 2 dimensions d'un jeu type Sudoku ? D'un côté L1Test ne permet pas de tester les procédures d'affichage, de l'autre il est nécessaire de tester les calculs.

Dans ce cas la solution consiste à écrire une fonction de calcul qui calcule et renvoie une chaîne (par exemple la chaîne qui représente la grille de jeu). Cette chaîne contiendra des caractères spéciaux comme le retour à la ligne. Cette fonction est parfaitement testable.

Ensuite il restera à écrire une fonction d'affichage (non testable) qui se contente d'afficher la chaîne fournie en paramètre, ou d'appeler la fonction précédente.

Dans l'exemple précédent la chaîne `"Le 16/7 est en été."` n'est pas très compliquée.

On aurait néanmoins pu écrire 2 fonctions :

- l'une de calcul, testable et testée qui produit la chaîne,
- l'autre d'affichage qui appelle la précédente.

```
def calcule_chaine_saison(num_jour:int, num_mois:int, saison:str) -> str:
    """Calcule une chaîne indiquant que le jour et le mois passés
    en paramètre appartiennent à la saison passée en paramètre.

    precondition: num_jour compris entre 1 et 31
                  num_mois compris entre 1 et 12

    $$$ calcule_chaine_saison(16, 7, "été")
    "Le 16/7 est en été."
    """
    return f"Le {num_jour}/{num_mois} est en {saison}."

def affiche_saison(num_jour:int, num_mois:int, saison:str) -> None:
    """Affiche un message indiquant que le jour et le mois passés
    en paramètre appartiennent à la saison passée en paramètre. Par ex :
    Le 16/7 est en été.

    precondition: num_jour compris entre 1 et 31
                  num_mois compris entre 1 et 12
    """
    print(calcule_chaine_saison(num_jour, num_mois))
```

6.5 Tester avec des caractères spéciaux

Quand on veut utiliser en donnée de test une chaîne de caractères contenant un caractère spécial comme `\n` (retour à la ligne) ou `\t` (tabulation), on s'aperçoit que leur utilisation déclenche une erreur. Par exemple :

```
$$$ "a" + "\n"
'a\n'
```

déclenche une erreur assez incompréhensible : `raise ValueError('line %r of the docstring for %s has '` `ValueError: line 10 of the docstring for <nom_fichier>`.

Pour une raison difficile à expliquer en L1, il est nécessaire de doubler le caractère \. On dit qu'on l'*échappe*.[1]

On écrira par exemple :

```
def formatte_etoiles() -> str:
    """Renvoie une chaîne contenant des étoiles formattées.
    ***   ***
    ***   ***

    Précondition : /
    Exemple(s) :
    $$$ formatte_etoiles()
    '***\\t***\\n***\\t***'
    """
    etoiles = '***'
    ligne = etoiles + '\t' + etoiles
    return ligne + '\n' + ligne
```

[1] La nécessité d'échapper le caractère \ n'est pas propre à L1Test, vous verrez par exemple en LInfo comment fonctionnent les expressions régulières.

Université de Lille
Écrire sur le terminal
2025-2026
UE PROG, L1S1 MI

On notera bien que les tests contiennent des \ d'échappement (dans `\\t` et `\\n`) mais que le code n'en contient pas (`\t` et `\n`).

7 Bonnes pratiques

7.1 Écrire des fonctions de calcul et des procédures d'affichage

Un programme principal classique effectue un calcul et réalise un affichage pour porter à la connaissance de l'utilisateur le résultat du calcul.

Dès que le code devient un peu long ou complexe, ou a vocation à être maintenu ou partagé, une bonne pratique consiste à isoler ce code dans :

- des fonctions qui réalisent un calcul à partir des valeurs de leurs paramètres d'une part ;
- des procédures qui réalisent un affichage d'autre part.

Le programme principal réalise des appels à ces fonctions et procédures.

Une autre bonne pratique consiste à identifier clairement les procédures qui réalisent un affichage en choisissant un nom qui commence par exemple par `affiche`.

L'exemple du chapitre précédent respecte ces bonnes pratiques.

NB : Comprendre la différence entre une fonction qui renvoie une valeur (`return`) et une procédure qui réalise un affichage (`print`) est essentiel.

7.2 Tester le calcul de la chaîne à afficher

Quand la chaîne à afficher est issue d'un calcul complexe dont le test est nécessaire, il est souhaitable d'utiliser 2 fonctions :

- l'une qui calcule la chaîne à afficher et est testée,
- l'autre qui réalise juste l'affichage, non testée.

8 Memento

- on écrit des valeurs de tout type sur un terminal avec la procédure prédéfinie `print()`
- `print()` écrit sur une ligne les valeurs de ses paramètres séparées par une espace
- une procédure est une fonction qui ne renvoie pas de valeur.
- les procédures d'affichage ne se testent pas avec `L1test`.

Département Informatique de la Faculté des Sciences et Technologies

On découvre :

- comment lire un texte depuis le terminal
- que des données de tout type peuvent être lues depuis le terminal

La fonction Python prédéfinie `input()` permet de lire des caractères saisis par l'utilisateur via le terminal.

Elle renvoie la chaîne de caractères lue.

1 Usage de la fonction `input()`

La fonction s'utilise ainsi :

`input(<message>)`

où `<message>` est une chaîne de caractères, appelée aussi *prompt* ou *invite*.

L'exécution de l'appel a l'effet suivant :

- affichage de la valeur de `<message>` sur le terminal
- tant que la touche `Entrée` n'a pas été frappée : attente que l'utilisateur frappe sur des touches au clavier et affichage immédiat des caractères associés
- renvoi de la chaîne de caractères contenant dans l'ordre les caractères correspondants aux frappes (sans prendre en compte la touche `Entrée`)

La fonction `input` renvoie systématiquement une chaîne de caractères, même si l'intention est de saisir, par exemple, un entier.

Exemple

Pour demander la saisie d'un numéro de jour en commençant par afficher le message `Entrer un numéro de jour :` on pourra écrire :

```
>>> saisie = input('Entrer un numéro de jour : ')
```

L'exécution de cette instruction déclenche l'affichage du message et l'attente d'une frappe au clavier :

```
Entrer un numéro de jour :
```

Si l'utilisateur souhaite saisir l'entier `16`, il appuie sur la touche `1` puis sur la touche `6` puis sur la touche `Entrée`.

```
Entrer un numéro de jour : 16
```

Si on consulte la valeur de `saisie`, on constate qu'elle contient bien la *chaîne de caractères* attendue :

```
>>> saisie
'16'
```

NB On peut aussi utiliser la fonction `input()` sans paramètre. Mais alors aucune invite ne s'affichera pour signifier à l'utilisateur qu'il doit frapper des touches !

NB La fonction `input()` est difficile à comprendre car elle effectue un affichage **et** renvoie la chaîne de caractères saisie (toujours avec le type str).

2 Fonctions de saisie

Nous pouvons maintenant écrire les fonctions de saisie du chapitre précédent. Par exemple :

```
def saisie_jour() -> int:
    """ Renvoie le numéro d'un jour saisi au clavier après avoir
    affiché le message `Entrer un numéro de jour :`
    """
    return int(input("Entrer un numéro de jour : "))
```

Plusieurs remarques sur la fonction `saisie_jour()`:

- elle renvoie un entier et non une chaîne de caractères, grâce à l'appel de la fonction de conversion `int()` ;
- si l'utilisateur saisit autre chose qu'un entier, la fonction produira une erreur ;
- elle n'a *pas de paramètre* ;
- elle contient une instruction `return` : ce n'est pas une procédure.

NB Une fonction de saisie peut avoir un ou plusieurs paramètres, par exemple pour modifier l'invite.

```
def saisie_jour_ulterieur(jour:int) -> int:
    """ Renvoie le numéro d'un jour saisi au clavier ultérieur
    au numéro de jour passé en paramètre.
    """
    saisie = input(f"Entrer un numéro de jour ultérieur à {jour} :")
    return int(res)
```

NB Nous verrons plus tard comment vérifier les saisies effectuées par l'utilisateur.

3 Fonctions calculant une donnée aléatoire

La fonction de saisie sans paramètre peut surprendre. La donnée saisie vient "de l'intérieur" de la fonction grâce à l'appel de `input`, puis est retournée. On trouve ce même principe pour les fonctions qui utilisent l'aléatoire.

On souhaite écrire une fonction `lancer_de` qui simule le lancer d'un dé à 6 faces. Cette fonction renvoie une valeur entière entre 1 et 6, et n'a pas de paramètre. La donnée calculée vient de l'intérieur de la fonction.

```
from random import randint
def lancer_de() -> int:
    """ Simule le lancer d'un dé à 6 faces.

    Précondition : /
    [...]
    """
    return randint(1, 6)
```

De même que pour les fonctions de saisie, une fonction qui produit une donnée aléatoire peut avoir un ou plusieurs paramètres.

On souhaite écrire une fonction `tirer_caractere` qui renvoie un caractère tiré au hasard dans une chaîne de caractères non vide. Cette chaîne est passée en paramètre.

```
from random import choice
def tirer_caractere(chaine : str) -> str:
    """ Tire un caractère pris au hasard dans `chaine`

    Précondition : chaine non vide
    [...]
    """
    return choice(chaine)
```

4 Bonnes pratiques

4.1 Écrire des fonctions de saisie

On veillera à isoler les saisies dans des fonctions dont le nom commence par `saisie`.

Il est important de bien comprendre la différence entre :

- une fonction de calcul qui possède des paramètres (les données d'entrées de son calcul) et renvoie le résultat du calcul
- une fonction de saisie sans paramètre

Exemple

La fonction `calcule_saison()` calcule la saison associée à un numéro de jour et à un numéro de mois. Les données du calcul sont donc un numéro de jour et un numéro de mois, ce sont aussi les paramètres de la fonction.

Cette fonction ne réalise *pas* de saisie. On ne saisit pas au clavier le numéro de jour ni de mois.

Le travail de saisie se fait dans `saisie_jour()`, qui n'a pas de paramètre, et certainement *pas* le jour saisi !

4.2 (Ne pas) Tester avec L1test

On ne peut pas tester grand chose sur la fonction `saisie_jour()` ! La valeur renvoyée dépend du comportement de l'utilisateur, qui pour le moment peut entrer un peu n'importe quoi. Les fonctions de saisie sont à ce stade non testables et on ne met pas de test dans leur docstring.

Par contre, des tests doivent être écrits dans la fonction `calcule_saison()` puisqu'elle a des paramètres.
Nous voyons ici l'intérêt de séparer les saisies des calculs dans des fonctions séparées.

4.3 Cas des fonctions avec aléatoire

De même on ne peut pas tester grand chose pour les fonctions `lancer_de` et `tirer_caractere`, puis que le résultat n'est pas prévisible. Le plus simple est de ne pas les tester. Néanmoins il est possible de tester les propriétés de la valeur renvoyée :

- elle est comprise entre 1 et 6
- elle appartient à la chaîne

La forme des tests est alors un peu particulière car les tests peuvent utilisent une variable locale au test qui permet de mémoriser le résultat de la fonction. Par exemple :

```
$$$ de = lancer_de()
$$$ 1 <= de
True
$$$ de <= 6
True
```

La ligne `de = lancer_de()` ne constitue pas un test à elle toute seule, elle ne contient pas de résultat attendu. Elle permet de préparer le test qui la suit. La variable `de` a la portée des tests écrits dans la docstring de la fonction.

On aura ainsi :

```
from random import randint
def lancer_de() -> int:
    """ Simule le lancer d'un dé à 6 faces.

    Précondition : /
    $$$ de = lancer_de()
    $$$ 1 <= de
    True
    $$$ de <= 6
    True
    """
    return randint(1, 6)

from random import choice
def tirer_caractere(chaine : str) -> str:
    """ Tire un caractère pris au hasard dans `chaine`

    Précondition : chaine non vide
    $$$ car = tirer_caractere("aeiou")
    $$$ car in "aeiou"
    True
    """
    return choice(chaine)
```

5 Memento

- on lit des chaînes de caractères depuis le terminal avec la fonction prédéfinie `input()`
- les fonctions prédéfinies de conversions telles `int()` ou `float()` permettent de lire des données de tout type

Université de Lille
Lire depuis le terminal
2025-2026
UE PROG, L1S1 MI
Département Informatique de la Faculté des Sciences et Technologies