Sujet n°0

Algorithmique & Programmation
Découverte de Python et PyCharm

`Temps de réalisation: 1.5h`

L'enjeu principal du cours est d'apprendre les fondements du langage Python, de l'algorithmique et des structures de données. Nous utiliserons le langage Python, version **3.8+**, comme support d'implémentation et le logiciel PyCharm comme environnement de programmation.

Le TP n°0 a donc pour objectif de :

- se familiariser avec les rudiments du langage Python ;
- prendre en main l'IDE PyCharm ;
- écrire et interpréter un programme ;
- appréhender les *fonctions*, *modules* et *paquets*, leur création et leur utilisation ;
- mettre en œuvre les `docstring` ;
- explorer un outil de test.

L'échelle de progression est la suivante :

● « débutant·e » ● « confirmé·e » ● « avancé·e » ● « expert·e »

Ex.	01	02	03	04	05	06	07	08	09	10	11	12	13
	●	●	●	●	●	●	●	●	●	●	●	●	●

Pour acquérir un niveau de compétence donné, il faut achever **TOUS** les exercices de ce niveau et des niveaux inférieurs. L'objectif de chacun·e est d'atteindre au moins le niveau *confirmé·e* (●) à l'issue du TP.

Votre meilleure amie pour la réalisation des TPs est la documentation officielle de Python :

https://docs.python.org/3/

Choisissez la langue et la version de votre interpréteur, puis utilisez le champ de recherche.

Table des matières

1 Environnement de développement

PyCharm est un environnement de développement (*Integrated Development Environment* - IDE) pour Python, V2 et V3, disponible gratuitement en version *Community*[1]. C'est un IDE professionnel, dédié à la programmation en Python. Il vous permettra rapidement de disposer d'un environnement de développement complet, notamment sur vos machines personnelles.

Bien que PyCharm soit très riche, son utilisation est plutôt intuitive. Outre la fenêtre principale d'édition de fichier, avec coloration syntaxique, complétion, indentation automatique, etc., l'interface se divise en plusieurs fenêtres, accessibles via le menu [`View/Tool Windows`], parmi lesquelles :

- la fenêtre **Project** offre un aperçu des contenus –répertoires, fichiers, bibliothèques externes, etc.– du projet en cours ;
- la fenêtre **Python Console** permet d'exécuter des instructions Python ligne à ligne,
- La fenêtre **Run**, proposée lorsqu'un programme est exécuté, diffuse la sortie standard d'un programme ;
- la fenêtre **Debug** permet de paramétrer et contrôler l'exécution du débogueur.

La documentation officielle (guides, tutoriels, tips, etc.) pour Pycharm se trouve ici :

https://www.jetbrains.com/fr-fr/pycharm/learn/.

Mon premier projet avec PyCharm

Pour débuter avec PyCharm, il est nécessaire de créer un projet. Nous vous invitons à créer un **unique projet pour toute la série des TPs**, à la manière d'un *cahier d'exercices*. Ce sera ensuite bien plus facile pour naviguer dans vos sources et réexploiter des portions de code d'un TP à l'autre.

1. Ouvrir PyCharm :
   - MS-Windows : menu `Démarrer > PyCharm`
   - Linux : exécuter le script `pycharm-community` dans le répertoire `/snap/pycharm-community/current`
2. Créer un nouveau projet dans la fenêtre *Welcome to PyCharm* avec le bouton `New Project`.

1. En tant qu'étudiant·e, vous avez accès à une licence éducative pour tous les produits JetBrains ! Profitez-en : https://www.jetbrains.com/fr-fr/community/education/#students

(a) Sélectionner l'emplacement souhaité pour le projet et nommer le répertoire racine (`algo` ou `exercices` par exemple)
   — MS-Windows : créer le projet dans votre dossier personnel partagé (`Z:`)
   — Linux : créer le projet dans le répertoire partagé `/media/EXXX`

(b) Sélectionner l'interpréteur Python *pré-configuré*, soit Python 3.10 (MS-Windows), soit Python 3.9 (Linux)

(c) Cliquer sur `Create`

Vous êtes fin prêt pour travailler en Python avec PyCharm.

1.1 L'interpréteur Python

La fenêtre de l'interpréteur peut être invoquée à l'aide du menu `View > Tool Windows > Python Console`. Vous pouvez y saisir des commandes interactives.

Cette fenêtre contient une ligne de commande symbolisée par 3 chevrons :

```
>>>
```

Dans cette ligne de commande, vous pouvez écrire un ordre –aussi appelé instruction–, puis envoyer cet ordre à l'ordinateur qui se charge de l'interpréter et de produire le résultat attendu. Par exemple, tapez la commande suivante :

```
>>> 1 + 1
```

Validez l'ordre avec la touche `Entrée` et observez le résultat que vous obtenez. Essayez maintenant d'exécuter les ordres suivants :

```
>>> print("Bonjour !")
>>> 23.2 * (2 + 6.3)
```

Exercice 1: Interpréter des instructions

Question 1. Prévoir le résultat que l'ordinateur livrerait s'il devait interpréter les ordres suivants :

```
>>> 1 > 2
>>> 5.0 / 2
>>> 5 / 2
>>> 5 // 2
>>> 5. // 2
>>> 5 % 2
>>> print("bonjour", 5*2)
>>> True and False
>>> True or False
>>> print( non_defini )
>>> True and non_defini
>>> False and non_defini
>>> non_defini and False
```

Question 2. Confirmer vos prévisions en observant le résultat produit par ces commandes dans l'interpréteur.

1.2 Variables et affectations

Exercice 2: Types et variables

QUESTION 1. Examiner la série de commandes ci-dessous. Prévoir chaque résultat puis le confirmer à l'aide de l'interpréteur Python de PyCharm.

```
>>> i = 2
>>> i = i + 4
>>> i
>>> print(i)
>>> j = 5
>>> i > j
>>> b = i == 9
>>> not b
>>> i != 9
>>> x, y = 1, 2.0
>>> x = x+y
>>> y = x**y
>>> x, y
```

QUESTION 2. Tester la séquence d'ordres suivante :

```
>>> z, t, u = 2, 'True', True or False
>>> type(z), type(t), type(u)
>>> z += 1.0
>>> z, type(z)
>>> z += t
>>> u, type(u)
>>> u -= z
>>> u, type(u)
```

et commenter les résultats.

Nous reviendrons sur la notion de *typage* dans la section suivante.

1.3 Écrire des programmes

Exercice 3: Hello World !

Voici sur un exemple, la procédure pour créer et jouer un programme Python :

1. Sélectionner le répertoire racine du projet dans la fenêtre **Project**
2. Créer un nouveau fichier, nommé `hello.py`, avec `File > New ... > Python File`
3. Écrire ou recopier les 6 lignes suivantes dans le fichier `hello.py` :

```
text = 'Hello'
word = 'le monde'
if text == 'Bonjour':
    word = 0o7   # chiffre 0 (zéro), lettre 'o' et chiffre 7 (sept)
text += ' ' + word
print(text)
```

4. Enregistrer le fichier via le menu `File > Save All`

5. Exécuter le programme `hello.py` à l'aide de la commande `Run 'hello'` du menu `Run`.

Question 1. Reproduire la séquence précédente pour exécuter le programme `hello.py`. Si tout s'est bien déroulé, une fenêtre d'exécution doit indiquer la fin normale du programme (*exit code 0*) et afficher le résultat de l'instruction `print`.

Question 2. Ajouter au programme `hello.py` les lignes suivantes :

```
print(f'Dissection de la chaîne de caractères {repr(text)} :')
for i, c in enumerate(text):
    print(i, c)
```

Enregistrer les modifications dans le fichier `hello.py` via le menu `File > Save All`. Vous pouvez à nouveau exécuter votre programme en suivant le menu `Run > Run 'hello'` ou en cliquant sur le triangle vert (*play*). Observer et décrire ce qu'il se passe.

Remarque: *Pour en savoir plus sur les chaînes de caractères formattés littérales (ou f-strings), veuillez vous reporter à la documentation officielle.*

Question 3. Modifier à nouveau le programme `hello.py` en changeant la valeur de la variable `text` en `text = 'Bonjour'`. Exécuter le programme et proposer une explication à ce nouveau comportement. En particulier, pourquoi l'erreur observée n'a pas été détectée *avant* l'exécution ?

Pour poursuivre l'exemple, modifier la déclaration des variables `text` et `word` comme suit :

```
text: str = 'Bonjour'
word: str = 'le monde'
```

Question 4. Passer la souris sur la valeur surlignée `0o7` et analyser l'indication fournie par PyCharm. Corriger en conséquence –ajouter des guillemets, par exemple–. Il est important de noter que cette correction a lieu *avant* toute exécution.

Remarque: *Notez que ces annotations (`<var>: <type>`) n'ont pas valeur d'obligation et sont purement ignorées par l'interpréteur ! La plupart des IDE génériques (VS Code, Sublime Text, TextMate, Atom, Emacs, vi, etc.) ne proposent pas –encore– de vérification de type, mais disposent de mécanismes d'extension qui peuvent parfois y remédier. À défaut, la vérification de type peut-être réalisée à l'aide de la commande `mypy` du package du même nom.*

1.4 Déboguage avec PyCharm

Pour activer le débogueur, il faut poser préalablement un point d'arrêt (*Breakpoint*) sur une ligne de code du programme. En cliquant à droite du numéro de ligne dans la marge, vous devriez voir apparaître un point rouge qui matérialise ce *Breakpoint*. Ensuite, sélectionnez [`Run/Debug 'xxx'`]. Le programme est exécuté jusqu'au premier point d'arrêt. La fenêtre du débogueur livre alors des informations utiles.

L'exécution se poursuit grâce à quelques actions usuelles à partir d'un point d'arrêt :

— **Resume Program** pour reprendre l'exécution, jusqu'au prochain point d'arrêt ou la fin du programme ;
— **Step Into** pour exécuter l'instruction suivante. Si l'instruction courante est un appel de fonction, le débogueur entre dans la fonction ;
— **Step Over** pour exécuter jusqu'à la ligne suivante du programme ;
— **Step Out** pour exécuter jusqu'à la fin de la fonction courante (très utile pour sortir d'une fonction si on y est entré par erreur).

Le panneau **Frames** affiche la *pile d'appels*, mais dans le cas d'un programme simple il n'y a qu'une seule ligne indiquant « l'appel » au programme principal.

Le panneau **Variables** indique à tout instant l'état des variables déclarées dans le programme. C'est une source d'information très intéressante pour corriger les anomalies.

Exercice 4: Pas-à-pas

QUESTION 1. Copier dans un fichier `debug.py`, puis exécuter le programme suivant :

```
# suite de Collatz
u: int = 27
while u >= 1:
    if u % 2:           # si u est impair:
        u = 3 * u + 1   # alors u est multiplié par 3, + 1
    else:               # si u est pair:
        u //= 2         # alors u est divisé par 2
print('La conjecture de Syracuse est vérifiée !')
```

Quel est le comportement observé ?

QUESTION 2. Relever les dix premières valeurs successives de u à l'aide de la commande *Resume Program* du débogueur, une fois posé un point d'arrêt sur l'instruction `while`.

QUESTION 3. Rendez-vous directement à l'itération de la boucle `while` pour laquelle $u = 1$, grâce à un point d'arrêt conditionnel[2]. Ensuite, une exécution pas à pas (F8) permet de relever les valeurs suivantes de u et détecter l'anomalie.

QUESTION 4. Corriger le programme, en prenant des informations sur la conjecture de SYRACUSE si nécessaire.

2 Premières fonctions, premiers problèmes

Exercice 5: Un peu d'arithmétique

QUESTION 1. Copier dans un fichier `arithmetique.py` et corriger, à l'aide du vérificateur de type de PyCharm, le programme suivant :

```
def somme(n: int) -> int:
    return n*(n+1)/2

print(somme(3.))
```

QUESTION 2. Soit $f(x) = 2x^2 - x + 1$; écrire la fonction f et un programme qui affiche le résultat de $f(1)$, $f(2)$ et $f(3)$ dans un fichier `polynome.py`.

QUESTION 3. Dans un fichier `moyenne.py`, écrire une fonction `moyenne` qui prend en paramètres deux entiers et qui calcule la moyenne de ces deux entiers. Écrire un programme qui affiche la moyenne de 11 et 14, de 18 et 15, de 20 et 15.

QUESTION 4. La suite de l'exercice se déroule dans le fichier `arithmetique.py`.

Écrire une fonction `est_divisible_par` qui prend en paramètres deux entiers n et k et qui renvoie *vrai* si n est divisible par k, *faux* sinon. Écrire un programme qui affiche la divisibilité de 5 par 3, de 6 par 2 et de 9 par 3.

2. Clic droit sur le point d'arrêt pour définir une condition d'arrêt.

Question 5. Écrire une fonction `est_pair` qui prend en paramètre un entier et qui renvoie *vrai* si l'entier est pair, *faux* sinon. La fonction `est_divisible_par` doit servir dans la définition de `est_pair`. Étendre le programme `arithmetique` pour afficher la parité des entiers 2, 4, 3 et 7.

Question 6. Écrire une fonction `est_compris_dans` qui prend en paramètres trois entiers, a, b et c et qui renvoie *vrai* si a est compris entre b et c (inclus). Ajouter au programme 5 tests qui couvrent les cas où $a < b$, $a = b$, $b < a < c$, $a = c$, $a > c$. Qu'arrive-t-il si $b = c$? $b > c$? a n'est pas un entier ? ...

2.1 Les fonctions d'entrée / sortie

Exercice 6: Attraper la saisie !

Question 1. Dans un fichier `util.py`, écrire une fonction de collecte d'un nombre entier à partir du clavier, de signature :

```
def saisir_entier() -> int
```

À la suite de la fonction, écrire un programme qui affiche la valeur saisie au clavier.

Question 2. Que se passe-t-il lorsque l'utilisateur saisit `'bla'` (à tester) plutôt qu'un nombre entier ? Grâce au bloc `try/except`, modifier la fonction pour traiter l'erreur `ValueError` et redemander la saisie tant que la valeur n'est pas entière. Un squelette de code est fourni ici :

```
try:
    # placer ici le code qui génère une ValueError
except ValueError:
    # sans erreur, le programme ne passe pas ici
    continue    # intercepte silencieusement l'erreur et poursuit le programme

# suite du programme...
```

Question 3. Modifier la fonction pour rendre paramétrable l'invite de saisie. La signature devient alors :

```
def saisir_entier(invite: str ='Saisir un nombre entier : ') -> int
```

Question 4. Modifier à nouveau la fonction de telle sorte qu'elle accepte un paramètre supplémentaire –motif d'échappement– optionnel. Le comportement général de la fonction doit demeurer identique :

```
def saisir_entier(invite: str = 'Saisir un nombre entier : ',
                  escape: str | None = None) -> int | None
```

Le type de retour `int | None` indique que l'on attend un entier (`int`) ou `None`. En effet, si la saisie correspond au motif d'échappement (`escape`), alors la fonction termine et retourne `None`, plutôt que de lever une exception et demander une nouvelle saisie. L'usage principal du motif d'échappement est `escape=''` qui permet de sortir silencieusement sur une saisie vide.

Exercice 7: Inventaire à la Prévert

Outre l'entrée (`input`) et la sortie (`print`) standards, les programmes Python peuvent aisément lire et écrire des fichiers de la mémoire externe :

```python
data: str
with open('mes-données.txt', 'r') as infile: # ouverture du fichier (mode r: read)
  data = infile.read()                       # lecture du contenu du fichier

# ici, on réalise un traitement sur les données (data)...

with open('mes-résultats.txt', 'w') as outfile:# ouverture du fichier (w: write)
  outfile.write(data)                        # écriture des données dans le fichier
```

En 1946, Jacques Prévert publie *Paroles*, un recueil de poèmes parmi lesquels figure le fameux *Inventaire*. Nous en avons un exemplaire sous la forme de fichier texte : `data_inventaire_prevert.txt`. Malheureusement, le scribe a totalement brouillé la mise en page !

Les questions de cet exercice seront traitées dans un fichier `prevert.py`.

Question 1. Remettre le poème en forme dans un nouveau fichier `inventaire_prevert.txt`, en numérotant chaque ligne.

La version ainsi obtenue commence par :

```
01. Une pierre
02. deux maisons
03. trois ruines
04. quatre fossoyeurs
05. un jardin
...
```

Astuce: *Il faut changer tous les caractères '*`;`*' en '*`\n`*' (caractère de saut de ligne) grâce à la fonction* `str.replace()`*, puis traiter les chaînes de caractères ligne par ligne* `str.split()` *pour ajouter les numéros au début de chaque ligne (opérateur de concaténation de chaînes '*`+`*') .*

3 Les modules et les paquets

Exercice 8: Importer des modules

Reprenons, par exemple, le module `arithmetique` du fichier `arithmetique.py`. Il expose les symboles `somme`, `est_divisible_par`, `est_pair` et `est_compris_dans`. Le module `util` expose quant à lui le symbole `saisir_entier`.

Copier le programme suivant dans un fichier `main.py` :

```python
import util
import arithmetique as art

n: int = util.saisir_entier()
if not art.est_pair(n):
    n +=1
print(f"Somme de 0 à {n} = {art.somme(n)}")
```

Le fichier `main.py` représente le programme que vous souhaitez exécuter. Ce programme utilise les fonctionnalités des modules `util` et `arithmetique` –abrégé en `art`–. Vous pouvez maintenant exécuter le programme `main` à l'aide du menu `Run > Run 'main'`.

Question 1. Quel effet indésirable peut-on observer dans le résultat de l'exécution ?

Il existe une façon simple de tester les fonctions d'une bibliothèque que vous développez, avant de l'utiliser dans d'autres programmes. Il suffit d'ajouter une condition particulière en fin de fichier et d'y adjoindre le programme de test de votre choix. À titre d'exemple, déplacez le code de test du module `arithmetique` sous la condition suivante :

```
# section exécutée uniquement si le module *est* le programme principal
if __name__ == '__main__':
    # placer les tests (programme) ici
    # attention à l'indentation
    pass
```

Exécutez successivement les programmes `arithmetique` et `main`. Observez que l'exécution du programme `main` n'est plus altérée que par le sous-programme `util`.

Question 2. Faire la modification nécessaire au programme `util` pour gommer définitivement les effets secondaires dans l'exécution de `main`.

Question 3. Modifier le programme `main` pour qu'il continue d'utiliser les fonctions importées, mais sans le préfixe du nom de module.

Dans PyCharm, la création de paquet passe par le menu `File > New ... > Python Package`.

Question 4. Créer un paquet `tp0` dans lequel vous allez verser tous les fichiers déjà créés lors du TP0.

Remarque: *La modification du chemin d'accès aux fichiers source requiert de revoir les instructions d'importation car les modules sont recherchés* à partir de la racine du projet*. Par exemple, le module* `util` *devient* `tp0.util` *partout où il est importé.*

Tous les programmes suivants seront créés dans le paquet `tp0`.

4 Les collections Python : `list`, `tuple`, `set`, `dict`

Ces quatre types natifs, combinés aux types primitifs `bool`, `int`, `float` et `str`, permettent de construire nombre de types dérivés tels que des ensembles de chaînes (`set[str]`), des listes d'entiers (`list[int]`), *etc.* Les chaînes de caractères (`str`) sont en fait des listes de caractères et se manipulent comme telles.

Exercice 9: Lancer de rayon !

L'algorithme de « lancer de rayon » (*ray casting* en anglais) résoud le problème *point-in-polygon* pour déterminer si un point est intérieur à un polygone, convexe ou non.

Le principe est le suivant : à chaque fois qu'on traverse le contour du polygone en se déplaçant, on change systématiquement de côté, on entre ou on sort. En se déplaçant depuis l'infini –forcément à l'extérieur– vers notre point, soit on sort autant de fois qu'on entre –nombre pair d'intersections– soit on entre une fois de plus –nombre impair–, *cf.* Figure 9. Finalement, puisque l'ordre dans lequel on trouve les intersections ne change pas le résultat, il n'est pas nécessaire de trier les intersections et il suffit de parcourir et tester les arêtes du polygone dans l'ordre dans lequel elles sont données.

Question 1. En utilisant la fonction fournie qui teste l'intersection entre une demi-droite horizontale d'origine donnée O et un segment $[AB]$, écrire un programme qui détermine si un point est à l'intérieur d'un polygone.

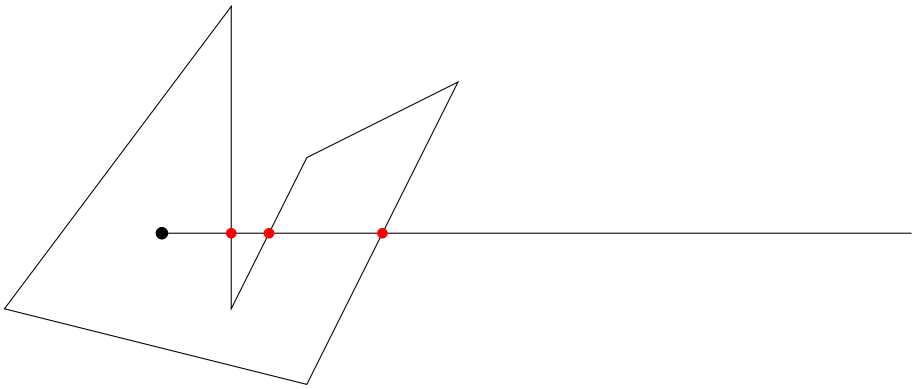

FIGURE 1 – Illustration du principe de l'algorithme de lancer de rayon
. Le point est intérieur car la demi-droite dont il est issu traverse le polygone un nombre impair de fois.

```python
Point = tuple[int, int]      # alias de type

def intersect(O: Point, A: Point, B: Point) -> bool:
    (xO, yO), (xA, yA), (xB, yB) = O, A, B
    return (
        (yO <= yA) == (yO > yB) and          # ordonnée dans l'intervalle
        xO < (xB - xA) * (yO - yA) / (yB - yA) + xA  # point du bon côté
    )
```

Exercice 10: Analyse fréquentielle

QUESTION 1. Créer un programme qui construit l'histogramme de fréquences des caractères apparaissant dans l'*Inventaire* de J. Prévert. Vous prendrez soin de supprimer les caractères de ponctuation et les formes accentuées pour ne retenir que 26 entrées au plus dans un tableau associatif (`dict`).

Remarque: *La bibliothèque de fonctions dédiées au type natif* `str` *de Python est riche et peut sans doute offrir quelqu'aide pour réaliser ce traitement.*

Astuce: *Pour constituer la liste des caractères à supprimer, il est facile de chercher dans le texte* tous les caractères sauf *les 26 entrées attendues !*

QUESTION 2. Afficher l'histogramme trié selon la fréquence d'apparition des caractères.

QUESTION 3. Normaliser les valeurs produites en pourcentage de la quantité de caractères du texte.

QUESTION 4. Produire l'histogramme normalisé et ordonné de fréquences d'un texte littéraire en français, tiré du projet Gutenberg : http://gutenberg.org.

QUESTION 5. Relever les 10 plus grands écarts de valeur entre les deux histogrammes précédents.

5 Pour aller plus loin...

Exercice 11: Les docstrings

Documenter le code Python devient rapidement une nécessité, en particulier pour des projets de développement collaboratifs. Les docstring permettent de produire de la documentation à partir d'annotations –à l'instar des commentaires– présentes dans le code source.

QUESTION 1. Écrire les *doctrings* du module `tp0.arithmetique` et de ses fonctions.

> **Remarque:** *Vous pourrez interroger les* docstrings *dans l'interpréteur Python à l'aide de la commande* `help(.)`.

QUESTION 2. Produire une documentation complète du module `tp0.arithmetique` au format HTML à l'aide de l'outil Sphinx.

Exercice 12: Les tests unitaires

Pytest est un outil externe à la librairie standard et requiert d'installer le module idoine. Dans la fenêtre de **Terminal**, reproduire la commande suivante :

```
prompt$ pip install -U pytest
```

La suite de tests Pytest est essentiellement fondée sur les *assertions* en Python, qui permettent dans le cas général de déclarer des invariants, des pré- et post-conditions, comme dans l'exemple suivant :

```python
def hello_world(name: str) -> str:
    assert name != '' and name[0].isupper()  # pré-condition
    return f'Bonjour {name}!'
```

Dans Pytest, un test correspond à l'écriture d'une fonction préfixée par `test_`, et munie d'une ou plusieurs assertion(s) :

```python
def inc(n: int) -> int:   # fonction à tester
    return n + 1

def test_inc():           # fonction de test
    assert inc(3) == 5    # assertion évaluée par pytest
```

Les fonctions de test peuvent être regroupées dans un module séparé, également préfixé par `test_`.

Pour plus d'information sur les fonctionnalités de Pytest, se reporter à la documentation officielle, et les nombreuses ressources en ligne.

QUESTION 1. Écrire une première suite de tests unitaires dans `test_arithmetique.py` pour la fonction `est_compris_dans` de la bibliothèque `arithmetique`.

QUESTION 2. Introduire une « fixture »[3] à même de produire un ou plusieurs cas de test.

QUESTION 3. Ajouter du paramétrage à l'aide de l'annotation `@pytest.mark.parametrize` pour couvrir tous les cas sans répéter les fonctions de test.

QUESTION 4. Étendre la série de tests à toutes les fonctions du module `arithmetique`.

3. Un contexte préparé pour l'exécution du test.

5.1 Vers l'infini et au-delà...

Si vous êtes arrivé·e· s jusqu'ici en ayant fait contrôler votre production par un· e chargé·e de TP, vous pouvez poursuivre.

Exercice 13: Quelques outils supplémentaires

Explorer successivement les thèmes suivants en élaborant de petits cas d'usage à partir du code source des exercices précédents :

1. le guide PEP-8 des bonnes pratiques d'écriture de code et les outils de vérification (`flake8` ou `pycodestyle` , par exemple) ;
2. la création de paquets Python « prêts à l'emploi » à partir du code source ;
3. la librairie de *profiling* de code `cProfile` .