Chapitre # (ALGO) 1 Fondamentaux

1 Introduction à Python .....
2 Fonctions & Procédures.....
3 Tests logiques & Boucles .....

Si debugger, c'est supprimer des bugs, alors programmer ne peut être que les ajouter.
— **Edsger Dijkstra**

Résumé & Plan

L'objectif de ce chapitre est de redécouvrir les principales structures en Python : les boucles, les tests, et de savoir les manipuler. Nous travaillerons pour le moment principalement sur des objets de type `int` (des entiers), `float` (flottants, des nombres à virgule). Les autres types seront approfondis dans de prochains chapitres.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices.
- Traitez les exercices présents dans la liste donnée au début de séance.
- Exercices 🌓 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

1. INTRODUCTION À PYTHON

1.1. Environnement de travail

Le réseau du lycée. Vous disposez d'un accès au réseau pédagogique du lycée qui vous fournit un espace disque accessible depuis n'importe quel ordinateur du lycée.

- Saisissez votre identifiant et votre mot de passe personnels pour vous connecter au réseau.
- Naviguez dans l'arborescence réseau et identifiez votre dossier personnel (dossier de travail). Vos fichiers doivent être enregistrés à cet emplacement, en créant au besoin des sous-dossiers (vous pouvez d'ores et déjà créer un dossier «Informatique», puis un sous-dossier TP1).
- Identifiez également le dossier partagé, qui est accessibles à tous les élèves de la classe ainsi qu'aux enseignants. Vous y trouverez pour certaines séances des fichiers déposés pour certains TPs.

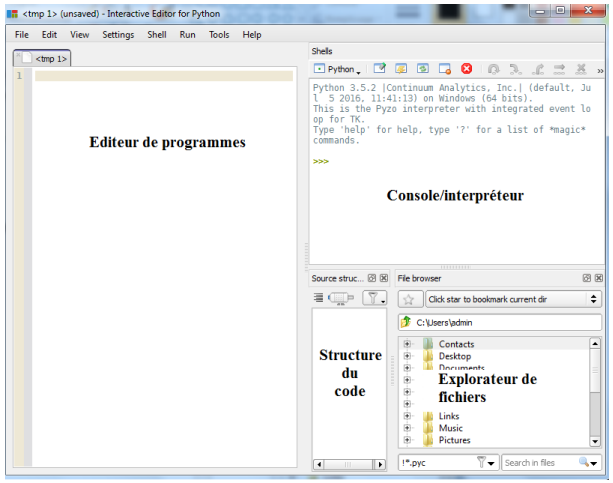
L'IDE *Pyzo*. L'environnement de développement *Pyzo* est choisi pour sa simplicité de mise en oeuvre et sa licence open source. Il est constitué :

- d'un **éditeur** (fenêtre de gauche par défaut) qui permet de saisir le programme (les mots-clé du langage sont colorés, ce qui permet une relecture facile, et l'indentation est automatique),
- d'une **console** (fenêtre de droite par défaut, appelée parfois aussi **Shell**), qui permet d'exécuter des instructions en ligne de commande (en tapant à la suite de `>>>` et de suivre le déroulement de son programme). Pour vous familiariser avec la console, taper par exemple les commandes `1+1`, `x = 3` puis `x+1`.
- D'un **explorateur de variables** (menu «Workspace»), qui permet de connaître les valeurs contenues dans les variables en cours d'utilisation. Cet fenêtre est généralement moins utile sauf pour les longs codes.
- La **structure du code** liste les différentes fonctions ainsi que leur dépendance.

Afin de prolonger le travail fait en classe, il est recommandé que vous l'installiez sur votre ordinateur personnel à partir du site `http://www.pyzo.org`.

Attention Différence entre éditeur et console

Il faut enregistrer son travail sous la forme de fichiers `.py` de façon régulière, dans un répertoire créé à cet effet (par exemple `TP1`) et on exécutera le programme en cours à l'aide de la commande «démarrer comme un script» ou de la touche F5. Une différence importante est que l'ensemble des instructions du fichier sera exécuté en une seule fois, contrairement à la console.



Attention

Les noms de fichier ne doivent ni contenir d'espaces, ni d'accents, ni de numéro au début.

En `Python`, toute ligne commençant par un `#` est un commentaire : son contenu sera ignoré lors de l'exécution du programme. Les commentaires servent donc à améliorer la compréhension du code, mais il ne faut pas non plus en abuser. Par exemple écrire `# augment x de 1` à côté de l'instruction `x += 1` est complètement inutile.

De plus, l'indentation (décalage du début de ligne pour aligner verticalement les instructions) est non seulement essentielle pour relire son programme (quelles sont les instructions exécutées après un **`if`** ou dans une boucle **`for`**?) mais aussi obligatoire pour le bon fonctionnement du programme. L'indentation ne se fait pas n'importe comment : on utilise la touche de tabulation ⇥ du clavier.

Dans un premier temps, nous allons travailler uniquement dans la console.

Résumé Privilégier l'éditeur

Il y a trois raisons pour lesquelles il faut travailler dans l'éditeur :
- On peut écrire plusieurs instructions à la suite.
- On peut sauvegarder son travail et donc le retrouver au prochain démarrage de `pyzo`.
- On peut y écrire des commentaires afin de clarifier son code.

1.2. Console

Il suffit pour exécuter une instruction saisie dans la console d'appuyer sur la touche ↵.

Exercice 1 | Calculs avec la console Solution Exécutez les instructions suivantes dans la console, et comprendre le résultat. *Vous pourrez annoter directement le sujet du TP.*

- `2+3`
- `2-3`
- `3**2`
- `10//3`
- `2+3.0`
- `3*2`
- `10/3`
- `10%3`

La console peut donc aussi servir de calculatrice, afin d'afficher des résultats «à la volée». Résumons les différentes opérations rencontrées dans un tableau.

OPÉRATIONS NUMÉRIQUES USUELLES

Opérateur Python	Opération mathématique
`x + y`	Somme de x et y
`x - y`	Différence de x et y
`x*y`	Produit de x et y
`x/y`	Quotient de x par y
`x**y`	x à la puissance y
`abs(x)`	Valeur absolue de x
`x//y`	Division entière de x par y
`x%y`	Reste de la division entière de x par y

1.3. Types

PANORAMA DES DIFFÉRENTS TYPES. En Python, les objets manipulés ont un type et le typage est automatique. C'est-à-dire que lorsque vous créez un objet en Python (par exemple *via* les commandes précédentes exécutées dans la console), Python lui affecte un type de manière automatique. Pour connaître le type d'une expression, on peut utiliser la commande type. Par exemple :

```
>>> type(12) # entier
<class 'int'>
>>> type(12.0) # flottant
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type("douze")
<class 'str'>
>>> type("12") # chaîne de caractère
<class 'str'>
>>> type(["d", "o", "u", "z", "e"]) # liste
<class 'list'>
>>> type({"1":"2", "2 eme":[1,2,3]}) # dictionnaire
<class 'dict'>
```

Fixons un peu de vocabulaire. Un objet informatique est dit *immuable*, si son état ne peut pas être modifié après sa création. À l'inverse il est dit *mutable*. Voici un panorama des types que nous allons rencontrer tout au long de l'année.

Type	Exemple	Quelques opérations	Mutable ?
Entier `int`	`42`	`+ , - , *, **`	Oui
Flottant `float`	`0.3`	`+ , - , *, ** , /`	Oui
Booléen `bool`	`1 == 2`	**`and`, `or`, `not`**	Oui
Chaînes `str`	`"blablabla"`	`+ , len , in`	Non
Liste `list`	`[0,1,2]`	`+ , len , in`	Oui
Tableaux `numpy`	`np.array([0, 1, 2])`	`+ , len , in`	Oui
Tuple `tuple`	`(0,1,2)`	`+ , len , in`	Non
Dictionnaire `dict`	`{'clef': 'val', ...}`	`.Keys, .Values`	Oui

Dans ce TP, nous utilisons principalement les types ci-après :

- le type `int` : les entiers relatifs.
- Le type `float` : les nombres flottants.
- Le type `str` : les chaînes de caractères («string» en Anglais).
- Le type `bool` : les booléens **`True`**, **`False`**.

Types numériques. En Python, les nombres peuvent avoir deux types, `int` ou `float`, suivant qu'il s'agit d'un entier relatif ou d'un nombre flottant. On dispose de plus des opérations suivantes :

Opérations numériques usuelles & Type

Commande	Effet
+	Additionne deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.
-	Même chose, mais pour la soustraction.
*	Multiplie deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.
/	Divise deux quantités. Le résultat est **toujours** un flottant. Par exemple, `2/2` retourne `1.0`.
**	Élève à la puissance deux quantités. Si les deux sont des entiers, le résultat est un entier. Sinon un flottant.

Dans le cas des entiers (type `int`), on possède deux commandes supplémentaires pour l'arithmétique.

```
>>> 5//2 # quotient de la division euclidienne
2
>>> 5%2 # reste de la division euclidienne
1
```

Attention aux flottants!

En Python, les entiers ont une taille arbitraire, limitée seulement par les capacités de la machine, les calculs se font donc en **valeurs exactes**.

```
>>> 2**100
1267650600228229401496703205376
>>> (10**100 +1) - 10**100
1
```

Les flottants en revanche ont un nombre de décimales limité[1] et il peut y avoir des erreurs d'arrondi, les calculs se font donc en **valeurs approchées**.

```
>>> 2.0**100
1.2676506002282294e+30
>>> (10**100 +1.)-10**100 # curieux, non ?
0.0
```

[1] Le concept «d'infini» est incompatible avec la notion même d'ordinateur. Par exemple, $\frac{1}{3}$ possède

Type booléen & Tests logiques. Ce type est adapté aux tests logiques (nous en reparlerons longuement quand nous ferons les tests `if` plus tard dans ce TP). Les booléens sont `True` et `False`.

Attention

On n'écrit **pas** `"True"` et `"False"` qui sont des chaînes et non des booléens.

```
>>> type("True")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Comment obtient-on concrètement un booléen ? Par exemple, par un test logique d'égalité. La syntaxe est la suivante `variable_1 == variable_2`, le résultat est alors un booléen. On peut aussi comparer des objets entre eux (si toutefois cela a un sens) à l'aide des symboles mathématiques usuels <, >, <= *etc.*.

Exercice 2 | **Tests logiques** Solution Prédire et observer les résultats des tests logiques ci-après.

```
1 == 2
1 == 1.0
1 == 1**2
"abc" = "a bc"
int(1.0) == 0+1
int(1.0) == 0+1.0
1 == 1/1
2 < 2
2 <= 2
2 <= 2.0
```

Tests logiques usuels

Opérateur	Signification
==	Est égal
!=	Est différent
<	Est strictement inférieur
>	Est strictement supérieur
>=	Est supérieur ou égal
<=	Est inférieur ou égal

Attention

Comme en témoigne l'exemple précédent, il faut donc rester **très méfiant** à propos des test ==. Par exemple, le résultat de `1 == 1.0` n'est pas du tout intuitif, Python opère implicitement à une conversion en flottant de l'entier `1` avant de réaliser le test. En cas de doute, mieux vaut effectuer quelques tests rapides dans la console avant.

Nous avons, comme dans le cours de Mathématiques, des opérations « ou » et « et » sur les booléens.

Opérateur « et »

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

Opérateur « or »

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

On dispose aussi d'un opérateur **`not`** pour obtenir la négation d'une proposition logique.

(Négation)

```
>>> not True
False
```

```
>>> not False
True
```

Remarque 1 **(Conversion de types)** Il est possible de convertir, par exemple, une expression de type `int` en `float`.

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> float(a)
1.0
>>> b = float(a) # conversion en flottant
>>> type(b)
```

un nombre infini de 3 après la virgule, Python est incapable de tous les prendre en compte donc tronquera la série de 3

```
<class 'float'>
>>> c = int(b)
>>> c # retour aux entiers
1
```

Il existe beaucoup d'autres possibilités de conversion, toutes sont intuitives ! Par exemple :

```
>>> int(True)
1
```

1.4. Variables

En Python, une variable est une façon de nommer un objet, ou plus précisément, de faire référence à un objet présent dans la mémoire. On dit que la variable *pointe vers l'objet*. Lorsque l'on fait pointer la variable vers un objet, on dit que l'on affecte cette variable. En Python, l'affectation se fait avec le signe égal =.

DÉCLARATION/AFFECTATION & MODIFICATION D'UNE VARIABLE. La syntaxe d'une déclaration de variable est la suivante :

```
>>> x = 1 # création de la variable x pointant vers l'entier 1
```

De manière générale, on écrit : `nomdelavariable = expression`. On peut aussi affecter à la chaîne plusieurs variables.

Affectations simultanées

```
>>> x, y, z = 1, 2, "coucou"
```

On peut afficher le contenu d'une variable en tapant dans la console le nom de la variable :

```
>>> x
1
>>> y
2
>>> z
'coucou'
```

On peut ensuite modifier le contenu des variables, ou encore créer de nouvelles variables à partir d'anciennes.

```
>>> x = x + 1
>>> y = x/2
>>> z = z + " au revoir"
```

On constate alors les modifications opérées.

```
>>> x
2
>>> y
1.0
>>> z
'coucou au revoir'
```

ÉCHANGER LES VALEURS DE DEUX VARIABLES. On suppose créees deux variables `a` et `b`, et on souhaiterait échanger leur valeur. Une idée naïve serait la suivante.

Exemple 1 (Échange naïf)

```
>>> a = 1
>>> b = 2.0
>>> b = a
>>> a = b
>>> a
1
>>> b
1
```

Analyser les valeurs de `a` et `b`. L'échange a-t-il été fait correctement ? Pourquoi ?

Dans ce problème, la bonne façon de voir les variables c'est comme des tasses que l'on remplit à l'aide de certains objets (entiers, flottants *etc.*). La question est donc : comment échanger le contenu de chaque tasse ? Réponse :

il faut une troisième tasse !

Exemple 2 (Échange correct) Compléter (avec une ou plusieurs lignes) la version ci-après pour que l'échange soit correct.

```
a = 1
b = 2.0
...
```

Dans la plupart des langages, il est nécessaire d'opérer comme ci-dessus (à l'aide d'une troisième variable). Cependant, Python possède une syntaxe efficace permettant d'éviter cela :

■■ Échanger les valeurs de deux variables

```
a, b = b, a
```

ADDITIONNER, SOUSTRAIRE, MULTIPLIER. Il existe des syntaxes plus « ramassées » afin d'additionner, soustraire ou encore multiplier une même variable.

■■ Privilégier

```
>>> n = 1
>>> n += 1
>>> n
2
>>> n -= 1
>>> n
1
>>> n *= 2
>>> n
2
```

■■ Éviter

```
>>> n = 1
>>> n = n+1
>>> n
2
>>> n = n-1
>>> n
1
>>> n = 2*n
>>> n
2
```

ACCÈS AUX OBJETS DÉFINIS. On peut par ailleurs accéder à l'ensemble des variables déclarées *via* la commande `globals()`, elles sont également à retrouver dans la fenêtre « explorateur d'objets » de Pyzo.

1.5. Importation de modules

Des modules supplémentaires ont été crées pour le traitement spécifique de certaines tâches. Nous pouvons citer les principaux que nous utiliserons :

- `math` : qui comme son nom l'indique, est dédié aux Mathématiques (toutes les fonctions usuelles principales par exemple).
- `numpy` : qui est dédié à tout le calcul numérique (résolution de systèmes linéaires, opérations diverses sur les matrices *etc.*). Nous utiliserons largement ce module dans le **??**.
- `matplotlib` : qui est dédié à l'affichage de graphiques divers.
- dans une moindre mesure vous utiliserez le module `scipy` pour les statistiques de deuxième année par exemple. Nous utiliserons largement ce module dans le **??**.

Afin d'utiliser une bibliothèque, on commence par l'importer. Pour cela, on procède de la manière suivante.

```
import nom_module as prefixe_a_choisir
```

Pour les modules cités précédemment nous ferons toujours :

```
import math as ma
import numpy as np
import matplotlib.pyplot as plt # sous-module pyplot de \
↪ matplotlib
```

Pour accéder à une fonction on utilisera

```
prefixe_a_choisir.nom_de_la_fonction
```

Par exemple :

```
>>> ma.sqrt(0)
0.0
```

Remarque 2 (Méthodes alternatives d'importation à éviter) On peut aussi importer les modules sans préfixe, c'est-à-dire que lorsque nous ferons appel aux fonctions dudit module, on peut éviter de taper le nom du préfixe. Par exemple,

```
>>> from math import *
>>> sqrt(2)
1.4142135623730951
```

Mais cela est dangereux : en effet, les modules `math` et `numpy` possèdent par exemple tous les deux des fonctions `cos`, `sin` *etc.*. Donc en important `math` puis `numpy`, on « écrase » les fonctions `math` par celles de `numpy` lors de la seconde affectation. Utiliser un préfixe permet donc d'avoir une garantie sur quelle fonction on utilise quand y fait appel.[a]

[a] Et il existe des différences entre le `cos` de `math` et le `cos` de `numpy` par exemple, nous y reviendrons.

2. FONCTIONS & PROCÉDURES

Nous créerons la plupart du temps des « groupes » d'instructions Python que l'on appelle « fonctions » ou « procédures ».

- On appellera *fonction* toute série d'instructions informatiques retournant un résultat (présence d'un `return` dans la suite).
- On appellera *procédure* toute série d'instructions informatiques ne retournant pas de résultat (par exemple, une modification des variables données en argument).

Attention

Un comportement analogue aux fonctions, que l'on trouve parfois dans littérature, consiste à utiliser des commandes du type `input` / `print`, mais nous nous l'interdirons.

Jusque maintenant, nous avons essentiellement travaillé dans la console. À présent, on part plutôt du côté de l'éditeur (fenêtre de gauche).

2.1. Généralités

Dans le langage Python, une fonction est une suite d'instructions dépendant de paramètres. Rappelons la syntaxe pour la définir en langage Python.

Structure d'une fonction

```
def f(x1, x2, ..., xn):
    """
    Documentation (docstring)
    """
    # instructions créant y1, \
    ↪ ..., yp
    return y1, y2, ... , yp
```

Structure d'une procédure

```
def f(x1, x2, ..., xn):
    """
    Documentation (docstring)
    """
    # instructions utilisant \
    ↪ x1, ..., xn
```

Les paramètres `x1, x2, ..., xn` sont ici obligatoires, il existe aussi des paramètres dits optionnels (*i.e.* possédant une valeur par défaut), mais nous ne les utiliserons pas. La seule différence entre les deux structures de code est que l'une ne renvoie rien (procédure).

```
f(val1, val2, ..., valn) # exécution de fonction
help(f) # affichage de la documentation
res_1, ..., res_p = f(val1, val2, ..., valn) # stockage du \
↪ résultat de la fonction dans des variables
```

Une fonction est dite *itérative* si elle ne fait pas appel à elle-même (généralement construite à partir de structures simples comme **`if`**, **`while`**, **`for`** *etc.*). Elle est dite *récursive* dans le cas contraire, nous ne travaillerons pour le moment pas avec des fonctions récursives, cela sera fait dans le **??**.

Exemple 3 Observons cet exemple qui aide à mieux comprendre la gestion des paramètres d'une fonction et de ce qu'elle retourne.

```
def f(x):
    return x+1

def g(x):
    y = 1
```

On peut alors envisager différents types d'exécution.

```
>>> x = 1.0
>>> f(x) # retourne un résultat
2.0
>>> x    # variable x NON modifiée
1.0
>>> g(x) # ne renvoie rien
```

Enfin, notons qu'il est bien sûr possible d'utiliser dans le corps d'une fonction une autre fonction définie au préalable. Par exemple l'exécution du programme suivant entré dans l'éditeur :

```
def f(x):
    return x+1

def g(x):
    return x**2

def h(x):
    return g(f(x))
```

```
>>> h(2)
9
```

VARIABLES LOCALES ET GLOBALES. Toute grandeur définie à l'intérieur d'une fonction ne peut être utilisée à l'extérieur, sauf exception. Les variables déclarées à l'intérieur d'une fonction sont donc appelées *variables locales*.[2]

Cependant il est aussi possible de décréter une variable comme étant «globale» et que l'on pourra utiliser ensuite à l'extérieur.

Exemple 4

1. Dans ce premier exemple, la variable `a` est locale.
```
def g(x):
    a = x**2
    return a**2
```
En effet, une exécution dans la console renvoie une erreur.
```
>>> g(1)
1
>>> a
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'a' is not defined
```
2. Mais dans le second,
```
def h(x):
    global a
    a = x**2
    return a**2
```
cette fois-ci, la variable `a` est bien définie même en dehors de la fonction. On parle alors de *variable globale*. Une fois la fonction `h` exécutée, un appel à `a` dans la console donnera la valeur de `a`.
```
>>> h(1)
1
>>> a
1
```

Définition 1 | Effet de bord
Une fonction est dite à *effet de bord* si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

Par exemple, une fonction modifiant une liste passée en argument, nous en rencontrerons plus tard.

[2] Sur le même principe que les variables de sommes ou d'intégrales en Mathématiques.

LES FONCTIONS `print` ET `input`. Deux fonctions importantes de Python — présente nativement sans importation de module — sont `print` et `input` et qui s'utilisent comme suit :

- `print(x)` : prend en argument une variable `x` ou un objet, et affiche l'objet. Cette fonction sert la plupart du temps à des fins de débogage, en faisant afficher des quantités interessantes au milieu d'un algorithme qui renverrait une erreur.
- `x = input("message d'information")` : prend en argument une chaîne qui est un message d'information, et stocke dans `x` la valeur entrée. Cette fonction sera très rarement utilisée, elle est seulement faite pour interagir avec un utilisateur (par exemple si vous programmez un jeu et que l'utilisateur doit faire un choix). Mais, en TP, la plupart du temps l'utilisateur ce sera vous-même donc on privilégiera l'usage de fonctions (voir plus bas).

On peut également afficher plusieurs objets à la suite qui doivent être séparés par des virgules.

```
>>> print("2+3=5")
2+3=5
>>> print(2+3, "=", 5)
5 = 5
>>> print("2+3", "=", 5)
2+3 = 5
```

Attention `print` ou pas `print` dans l'éditeur ?
En exécutant le code de l'éditeur, les résultats sont affichés dans la console, mais seulement ceux des instructions dont l'affichage est explicitement demandé par la commande `print`. Ainsi, si l'éditeur est
```
x = 1
```
vous ne verrez aucun affichage (mais la variable `x` sera créee). En revanche, un affichage aura lieu si l'éditeur contient :
```
x = 1
print(x)
```
En revanche, dans la console, tout est affiché par défaut.

Remarque 3 (L'instruction `lambda` : définir en «mode *inline*» des fonctions)
Pour définir des fonctions d'une variable réelle, une syntaxe relativement commode est la suivante
```
>>> g = lambda x: x**2
>>> g(2)
4
```

```
>>> def g_prim(x):
...     return x**2
...
>>> g_prim(2)
4
```

Les deux commandes définissent dans le cas présent la fonction $x \longmapsto x^2$.
Vous voyez ici que le mot **lambda** est reconnu par Python. Il est donc interdit d'utiliser **lambda** en nom de variable.[a]

Remarque 4 (Méthode ≠ Fonction) On appelle *méthode* sur un objet `O` (liste, tuple, entiers, *etc.*), toute instruction du type

`O.nom_methode()`.

Par exemple, l'instruction `.append(x)` rajoute `x` dans `L` est une méthode sur les listes. Nous la verrons dans le Chapitre (ALGO) 2. Il faut distinguer les méthodes des fonctions. Par exemple, `sum` est une fonction Python qui retourne la somme des éléments d'une liste (la notion de fonction sera étudiée dans la prochaine section).

2.2. Fonctions de modules

Python dispose de plusieurs bibliothèques de fonctions destinées à un usage spécifique. Il convient de les charger grâce à la commande **import** avant de pouvoir les utiliser. Détaillons celles qui nous seront les plus utiles pendant l'année.

Le module math. Ls fonctions numériques usuelles (racine carrée, logarithme, *etc.*...) ne sont pas disponibles de base. Il faudra pour les utiliser importer des bibliothèques de calcul numérique comme `math` ou `numpy`. Prenons l'exemple de la bibliothèque `math` : elle contient notamment les fonctions suivantes.

Voir toutes les fonctions disponibles dans un module

```
>>> import math
>>> dir(math)
```

[a] Privilégier alors `lamba` par exemple

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', \
↪ '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', \
↪ 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', \
↪ 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', \
↪ 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', \
↪ 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', \
↪ 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', \
↪ 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', \
↪ 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', \
↪ 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

On retiendra que la fonction `factorial` peut désormais être utilisée. Les fonctions usuelles peuvent aussi être importées depuis la bibliothèque `numpy` : ces dernières sont plus adaptées aux calculs numériques matriciels, par exemple pour les tracés de courbe que nous reverrons plus tard.

Les modules numpy, scipy, random et matplotlib Les bibliothèques `numpy` et `scipy` fournissent des outils pour le calcul scientifique. Nous utiliserons surtout `random` (simulations), `numpy` (calculs numériques), `matplotlib` (graphiques), plus rarement `scipy` (en Statistiques essentiellement). Tout autre module n'est pas un attendu du programme.

3. TESTS LOGIQUES & BOUCLES

On présente dans cette section les principales structures en Python.

3.1. Tests logiques

Test if simple

```
if test:
    instructions
else:
    instructions
```

Test if avec plusieurs conditions

```
if test:
    instructions
elif test:
```

```
    instructions
else condition:
    instructions
```

Remarque 5 S'il y a plus que deux cas nécessitant un traitement différencié, on peut utiliser l'instruction `elif` (contraction de else et if). L'instruction `else` finale sera traitée seulement si les cas précédents n'ont pas été rencontrés.

Exercice 3 | Fonctions mathématiques définies par morceaux Solution Coder en Python les deux fonctions mathématiques ci-après :

$$f_1 : x \mapsto \begin{cases} -x & \text{si } x \leqslant -1, \\ x^2 & \text{si } x > -1 \end{cases}, \quad f_2 : x \mapsto \begin{cases} -x & \text{si } x \leqslant -1, \\ x^2 & \text{si } -1 < x < 2 \\ x+2 & \text{si } x \geqslant 2. \end{cases}$$

Exercice 4 | Maximum Solution

1. Voici une fonction qui donne le maximum de deux nombres (sans utiliser la fonction max).
```
def maximum1(a, b):
    if a > b:
        return a
    else:
        return b
```
Essayez de comprendre pourquoi la deuxième fonction `maximum2` marche aussi.
```
def maximum2(a, b):
    if a > b:
        return a
    return b
```
2. En utilisant l'une des fonctions précédentes, faites la même chose pour le maximum de trois nombres.

Attention Comportement de la commande `return`

Nous avions donc vu que lorsque Python rencontre `return`, il renvoie le résultat et sort de la fonction, quelque soit ce qui est écrit après.

3.2. Boucles

3.2.1. Inconditionnelle : boucle `for`

Ce sont les boucles dont on connait à l'avance le nombre d'étapes nécessaires pour arriver au résultat.

(Boucle for)

```
for k in string/list/etc:
    instructions
```

```
for _ in string/list/etc:
    instructions
    # répétition des \
    ↪ instructions
```

Lorsque le corps de boucle de dépend d'aucun indice, il s'agit alors d'une répétition d'actions, on peut alors utiliser la version de droite avec underscore.

La plupart du temps, nous utilisons l'itérateur `range` pour faire avancer l'indice d'une boucle `for`. Le mieux est de comprendre cela sur un exemple.

Exemple 5
```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for i in range(1, 3):
...     print(i)
...
1
2
>>> for i in range(3, 1):
...     print(i)
...
```
Commentez ces deux exemples.

Dans ce TP nos boucles `for` parcourront des `range`, mais on peut aussi parcourir des listes, des chaînes de caractères et même des dictionnaires (*cf.* Chapitre (ALGO) 2 et **??**). Lorsque l'indice d'une boucle `for` n'est pas utilisé dans un bloc, on peut omettre de lui donner un nom, voici un exemple.

Répétition d'instructions

```
>>> for _ in range(1, 3):
...     x = 1
...
```

Résumé

- `range(a, b)` : parcourt tous les entiers entre `a` et `b-1` si $b > a$. Dans le cas contraire, ne fait rien.
- `range(a, b, h)` : parcourt tous les entiers entre `a` et `b-1` en avançant avec un pas de `h` si $b > a$. Dans le cas contraire, ne fait rien.
- `range(b)` correspond à `range(0, b)`.

En d'autres termes,

«Pour i allant de 0 à $n-1$»

```
for k in range(n):
    ...
```

«Pour k allant de 1 à n»

```
for k in range(1, n+1):
    ...
```

«Pour k allant de 5 à 13 de 2 en 2»

```
for k in range(5, 13, 2):
    ...
```

Exercice 5 | Fonctions mystères Pour chaque fonction ci-après, écrire ce qu'elle renvoie en interprétant son résultat.

```
def mystere_1(N):
    S = 0
    for k in range(1, N+1):
        S += k
    return S
```

```
def mystere_2():
    S = 2
    for k in range(3):
        S = S**2
    return S
```

Exercice 6 | Calculer un produit : la factorielle Solution Si $n \in \mathbf{N}$ est un entier, on appelle *factorielle de n* la quantité définie par :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ 1 \times 2 \times \cdots \times n & \text{sinon.} \end{cases}$$

1. Pour calculer $n!$ on propose le programme suivant :

```
def factoriel(n):
    P = 1
    for k in range(n+1):
        P *= k
    return P
```

2. Écrire une fonction d'en-tête `factorielle(n)`, où `n` est un entier, qui renvoie la valeur de $n!$. *On prévoira tous les cas possibles sur n.*

Exercice 7 | Calculer une somme Solution

1. Écrire une fonction d'en-tête `harmo(n)`, `n` étant un entier strictement positif, et qui renvoie la valeur de
$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \cdots + \frac{1}{n}.$$
2. Écrire un script qui affiche la valeur de $\sum_{k=1}^{10^p} \frac{1}{k} - \ln(10^p)$ pour $p \in [\![1, 8]\!]$. Qu'observe-t-on ?

Attention La présence d'un `return` arrête une boucle `for` !

Un point très important est à constater : dès que, au sein d'une fonction, on arrive sur une instruction `return` dans une boucle `for`, alors la boucle `for` est arrêtée. Voyez par exemple :

```
def f():
    n = 0
    for _ in range(3):
        n += 1
        return n
>>> f()
1
```

L'entier `n` n'a été augmenté qu'une seule fois de 1, le premier `return` a arrêté complètement la boucle `for`. De manière générale, le `return` arrête toute la fonction.

Exercice 8 | Vérification de formules Solution

1. Écrire une fonction d'en-tête `somme_puissance(p, n)` prenant en argument deux entiers, et renvoyant la valeur de $\sum_{k=0}^{n} k^p$.
2. Vérifier, à l'aide de Python, les assertions mathématiques ci-après :

$$\forall n \in [\![0, 10]\!], \quad \sum_{k=0}^{n} k = \frac{n(n+1)}{2}, \quad \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}.$$

Exercice 9 | **Dessins** Solution Écrire des fonctions prenant en argument le nombre de lignes `n` (ici `n = 5`) et permettant l'affichage des figures suivantes.

1.
```
    *
   * *
  * * *
 * * * *
_________
```

2.
```
. . . .
*. . .
* *. .
* * *.
* * * *
_________
```

3.
```
* * * *
. . . *
. . . *
. . . *
* * * *
_________
```

Indication : *On pourra observer le résultat de l'instruction* `"*"*3+"."*2` *dans la console*

CALCULER LES TERMES D'UNE SUITE. Commençons avec une suite récurrente à 1 pas, c'est-à-dire le terme suivant ne dépend que du terme précédent.

$$u_0 = 1, \quad \forall n \in \mathbf{N}, \quad u_{n+1} = 2u_n + 3.$$

On souhaite créer une fonction qui va retourner la valeur de u_n pour tout $n \in \mathbf{N}$. L'idée est la suivante :

Méthode **Calculer informatiquement le terme u_n d'une suite récurrente $u_{n+1} = f(u_n)$**

- On stocke la valeur initiale de la suite dans une variable `u` $\leftarrow u_0$.
- On itère la relation de récurrence (en faisant ici `u = 2*u + 3`) autant de fois que nécessaire à l'aide d'une boucle `for` de longueur `n`. Le plus simple est de faire coïncider l'indice des mathématiques avec celui de la boucle `for`.

Exemple 6 Pour notre suite précédente, cela donnerait :

```
def suite_U_exemple(n):
    u = 1
    for _ in range(1, n+1):
        u = 2*u + 3
    return u
```

On peut alors tester.

```
>>> suite_U_exemple(3)
29
>>> suite_U_exemple(0)
1
```

On constate que le script fonctionne également pour `n = 0`. En effet, dans ce cas la boucle `for` ne s'exécute pas puisque les bornes du `range` ne sont pas dans le bon sens. On retourne alors simplement le `u` initial.

À vous de jouer.

Exercice 10 | **Suite récurrente à un pas** Solution On considère la suite « arithmético-géométrique »[3] définie par :

$$u_0 = 4, \quad \forall n \geq 0, \quad u_{n+1} = 2 - \frac{u_n}{2}.$$

Écrire un programme itératif, *i.e.* faisant appel à une boucle `for`, prenant en argument un entier n et qui calcule u_n.

COMPTER. Python peut également servir à compter des choses au moyen d'une variable, appelée *compteur*, vouée à augmenter de 1 à chaque occurence de comptage. La structure type est donc la suivante.

Comptage
```
def comptage():
    N = 0 # compteur
    for _ in .....:
        if evenement_a_compter:
            N += 1
    return N
```

Exercice 11 | **Nombre d'entiers divisibles par 4** Solution Écrire une fonction d'en-tête `compte_divisibles_par_quatre(n)`, `n` étant un entier, retournant le nombre de nombres divisibles par 4 entre 0 et n. *Indication :* *On rappelle que le symbole* `%` *permet de retourner le reste de la division euclidienne.*

CHRONOMÉTRER. Il est important en Informatique de pouvoir quantifier l'efficacité d'un programme plutôt qu'un autre, cela peut se faire à l'aide du module `time`, que l'on importe de la manière suivante.

```
>>> import time as ti
>>> ti.time()
1692708480.652765
```

La commande `ti.time()` retourne le nombre de secondes écoulées depuis le 1er janvier 1970 à 0 :00. On peut ensuite s'« amuser » à calculer le nombre d'années écoulées entre cette date et la création de ce TP : `ti.time()/(3600*24*365)` qui retourne 53.67543381111923. En calculant des différences de temps, on peut alors estimer le temps d'exécution d'instructions Python. Voyons un exemple.

[3] Nous étudierons ces suites dans le cours de Mathématiques

■■ Chronométrage naïf

```
>>> import time as ti
>>> t_deb = ti.time()
>>> x = 3
>>> y = x**4
>>> t_fin = ti.time()
>>> t_fin - t_deb (temps mis pour l'exécution des deux instructions)
3.1948089599609375e-05
```

Exercice 12 | En s'inspirant du code précédent, donner une estimation du temps nécessaire pour l'exécution de la question **2.** de l'exercice 7.

Pour être plus robuste, mieux vaut réaliser ce chronométrage un certain nombre de fois et retourner la moyenne ; en effet, la rapidité d'exécution dépendra des conditions d'utilisation de la machine, et peut donc sensiblement différer d'une exécution à l'autre.

■■ Chronométrage robuste

```
>>> import time as ti
>>> nb_ex = 1000 # nb d'exécutions
>>>
>>> temps_moy = 0
>>> for _ in range(nb_ex):
...     t_deb = ti.time()
...     x = 3
...     y = x**4
...     t_fin = ti.time()
...     temps_moy += t_fin - t_deb
...
>>> temps_moy/nb_ex # temps moyen sur nb_ex essais
7.128715515136718e-08
```

OPTIMISER : TROUVER LE MINIMUM/MAXIMUM D'UNE FONCTION SUR UNE LISTE D'ENTIERS.
Cet exercice n'est à traiter qu'en fin de TP, s'il reste du temps. Les techniques mises en jeu seront largement revues dans le Chapitre (ALGO) 2 sur les listes.

Exercice 13 | **Trouver un maximum / minimum** Solution Dans cet exercice, on suppose créee une fonction `f` qui prend en argument un entier et renvoie un flottant. On pourra définir pour tout l'exercice :

```
def f(x):
    return x*(10-x)
```

puis en tester d'autres à la fin.

1. On commence par essayer de chercher le maximum.
   - **1.1)** Compléter la fonction `maximum` ci-après, prenant en argument un entier $N \in \mathbf{N}$ et retournant la plus grande valeur parmi `f(0), ..., f(N)`.

```
def maxi_f(N):
    """
    N : int -> maximum de f(0), ..., f(N)
    """
    maxi = f(0)
    for k in range(......):
        if .... > maxi :
            maxi = .....
    return maxi
```

   Tester pour `N = 10`, cela vous semble-t-il cohérent ?
   - **1.2)** Dans la fonction précédente, peut-on remplacer le symbole > par >= ?
   - **1.3)** Adapter la fonction précédente, en une fonction `maxi_ind_f`, et retournant en plus du maximum un indice $i \in [\![0, N]\!]$ où $f(i)$ est égal audit maximum.
2. Faire le même travail que dans la première question, mais pour trouver le minimum.

■ 3.2.2. Conditionnelle : boucle `while`

L'arrêt de la boucle dépend ici d'une condition. On se sait pas *a priori* lorsqu'elle va se terminer. Il convient donc de faire attention au fait que cette boucle se termine bien avant de lancer le code python.

■■ Boucle while

```
while test:
    instructions
```

Exercice 14 | **Suite récurrente à 1 pas divergente, le retour. Algorithme de seuil.** Solution On considère (u_n) la suite définie par :

$$u_0 = 2, \quad \forall n \geqslant 0, \quad u_{n+1} = 3u_n - 1.$$

On admet que pour tout $A > 0$, il existe $n_0 \in \mathbf{N}$ tel que $u_n > A$ pour tout $n \geqslant n_0$. Autrement dit, u_n est aussi grand que l'on veut pourvu que n soit assez grand.

Écrire une fonction d'en-tête `cherche_ndiv(A)` prenant en argument `A` un réel strictement positif, et qui retourne le premier entier n_0 de sorte que $u_{n_0} > A$. Testez cette fonction pour $A = 10^3$.

Exercice 15 | **Suite récurrente à 1 pas convergente. Algorithme de seuil.** Solution
On reprend l'exercice 10. On admet (en attendant le cours de Mathématiques) que cette suite « converge vers $\frac{4}{3}$ », c'est-à-dire que u_n est aussi proche que l'on veut de $\frac{4}{3}$ pourvu que n soit assez grand. Autrement dit, u_n est une bonne approximation de $\frac{4}{3}$ lorsque n est grand.

Écrire une fonction d'en-tête `cherche_n(eps)` prenant en argument `eps` un réel strictement positif, et qui retourne le premier entier n de sorte que u_n soit assez proche de $\frac{4}{3}$ au sens suivant :

$$\left|u_n - \frac{4}{3}\right| < \varepsilon.$$

Testez cette fonction pour $\varepsilon = 10^{-3}$.

Exercice 16 | **Suite de Heron. Algorithme de seuil.** Solution Soit $a \in \mathbf{R}^{+\star}$ et (u_n) la suite récurrente définie par :

$$a_0 = a, \quad \forall n \in \mathbf{N}, \quad u_{n+1} = \frac{u_n^2 + a}{2u_n}.$$

On admet (en attendant le cours de Mathématiques) que cette suite « converge vers $\sqrt{a}$ », c'est-à-dire que u_n est aussi proche que l'on veut de $\sqrt{a}$ pourvu que n soit assez grand. Autrement dit, u_n est une bonne approximation de $\sqrt{a}$ lorsque n est grand.

1. Écrire un programme itératif, *i.e.* faisant appel à une boucle `for`, prenant en argument un entier n et qui calcule u_n.
2. Écrire une fonction d'en-tête `heron_approx(a, eps)` prenant en argument a et `eps` un réel strictement positif, et qui retourne le premier entier n de sorte que u_n soit assez proche de $\sqrt{a}$ au sens suivant :
   $$\left|u_n - \sqrt{a}\right| < \varepsilon.$$
   En combien d'étapes la suite de Heron donne-t-elle une valeur approchée de $\sqrt{167}$ à 10^{-4} près ? à 10^{-8} près ?

Exercice 17 | **Suite & Conjecture de Syracuse** Solution Soit $a > 0$. On définit alors la suite suivante

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair,} \end{cases} \qquad u_0 = a.$$

La conjecture de Syracuse est la suivante : « pour tout entier $a > 0$, il existe un indice n tel que $u_n = 1$ ».

1. Créer une fonction d'en-tête `Syracuse(a, n)` et qui retourne la valeur de u_n, pour n un entier positif.
2. On appelle *temps de vol* le plus petit indice n tel que $u_n = 1$. Créer une fonction d'en-tête `temps_vol(a)` et qui retourne le temps de vol de la suite. *On prévoit que si l'on a pas atteint* 1 *en* 10^3 *coups, on retourne* `False`.
3. On appelle *altitude maximale de vol* la plus grande valeur de u_n jusqu'à son temps de vol (c'est-à-dire d'atteinte de 1). Adapter la fonction précédente pour qu'elle retourne en plus l'altitude maximale de vol.

SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé

```
>>> 2+3
5
>>> 2+3.0
5.0
>>> 2-3
-1
>>> 3*2
6
>>> 3**2
9
>>> 10/3
3.3333333333333335
>>> 10//3
3
>>> 10%3
1
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == 1
True
```

Solution (exercice 2) Énoncé

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == 1**2
True
>>> "abc" == "a bc"
False
>>> int(1.0) == 0+1
True
>>> int(1.0) == 0+1.0
True
>>> 1 == 1/1
True
>>> 2 < 2
False
>>> 2 <= 2
True
>>> 2 <= 2.0
True
```

Solution (exercice 3) Énoncé

```
def f_1(x):
    if x <= -1:
        return -x
    else:
        return x**2

def f_2(x):
    if x <= -1:
        return -x
    elif -1 < x < 2:
        return x**2
    else:
        return x+2
```

Solution (exercice 4) Énoncé

1. Lorsque Python rencontre `return`, il renvoie le résultat et sort de la fonction, quelque soit ce qui est écrit après. Dans notre cas,
   - si `a > b`, il réalise l'instruction `return a` puis sort de la fonction,
   - sinon, il sort de la structure conditionnelle et suit l'instruction suivante : `return b`.

   Cette fonction renvoie donc bien le maximum.
2.
```
def maximum3(a, b, c):
    return maximum1(a, maximum1(b, c))
>>> maximum3(-1, -2, 4)
4
```

Solution (exercice 6) Énoncé

1. k varie entre 0 et n et à la première boucle, on multiplie `P` par 0, `P` vaudra alors 0, et sa valeur ne changera plus jusqu'à la fin de la boucle.

2.
```
def factorielle(n):
    """
    retourne la factorielle de n de manière itérative
    """
    P = 1
    for k in range(1, n+1):
        P *= k
    return P
>>> factorielle(1)
1
>>> factorielle(4)
24
```

Solution (exercice 7) Énoncé

```
def harmo(n):
    S = 0
    for k in range(1, n+1):
        S += 1/k # multiplication de P par k
    return S
>>> for p in range(1, 9):
...     print(harmo(10**p)-ma.log(10**p))
...
0.6263831609742079
0.5822073316515288
0.5777155815682065
0.5772656640681646
0.5772206648931064
0.5772161649007153
0.5772157148989514
0.5772156699001876
```

On constate que lorsque p grandit, la suite se rapproche d'une certaine valeur. *La valeur en question s'appelle la constante d'Euler, elle sera étudiée dans le cours de Mathématiques en 2ème année.*

Solution (exercice 8) Énoncé

```
def somme_puissance(p, n):
    S = 0
    for k in range(0, n+1):
        S += k**p
    return S

def verifications_formules(n):
    verif_1 = (somme_puissance(1, n) == n*(n+1)/2)
    verif_2 = (somme_puissance(1, n) == n*(n+1)/2)
    verif_3 = (somme_puissance(1, n) == n*(n+1)/2)
    return verif_1 and verif_2 and verif_3

def verifications():
    for n in range(0, 11):
        if verifications_formules(n) == False:
            return False
    # si on arrive ici, c'est que toutes les formules sont \
    ↪ vérifiées
    return True
>>> verifications()
True
```

Solution (exercice 9) Énoncé

```
def dessin_1(n):
    for i in range(n):
        print('*'*i)

def dessin_2(n):
    for i in range(n):
        print('*'*i+'.'*(n-i-1))

def dessin_3(n):
    print('*'*(n-1))
    for i in range(1,n-1):
        print('.'*(n-2)+'*'*(1))
    print('*'*(n-1))
>>> dessin_1(5)

*
**
***
****
>>> dessin_2(5)
....
```

```
*...
**..
***.
****
```

```
>>> dessin_3(5)
****
...*
...*
...*
****
```

Solution (exercice 10) Énoncé

```
def suiteU_arithmgeo(n):
    u = 4
    for _ in range(1, n+1):
        u = 2 - u/2
    return u
```

```
>>> suiteU_arithmgeo(3)
1.0
```

Solution (exercice 11) Énoncé

```
def compte_divisibles_par_quatre(n):
    N = 0
    for i in range(n+1):
        if i % 4 == 0:
            N += 1
    return N
```

```
>>> compte_divisibles_par_quatre(10)
3
```

Solution (exercice 13) Énoncé

```
def f(x):
    return x*(10-x)

def maxi_f(N):
    """
    N : int -> maximum de f(0), ..., f(N)
    """
    maxi = f(0)
    for k in range(N+1):
        if f(k) > maxi :
            maxi = f(k)
    return maxi
```

```
>>> maxi_f(10)
25
```

Le résultat attendu est bien cohérent, puisque d'après le cours de mathématiques la fonction f est minimale en $\frac{10}{2} = 5$, et de valeur associée $f(5) = 25$.

Dans la fonction précédente, il est possible de remplacer le symbole > par >=. En effet, cela ne changera pas la valeur du maximum, en revanche, cela changera la valeur de l'indice trouvé dans la question qui suit.

```
def maxi_ind_f(N):
    """
    N : int -> maximum de f(0), ..., f(N), et un indice en \
    ↪ lequel il est atteint
    """
    maxi = f(0)
    ind = 0
    for k in range(N+1):
        if f(k) > maxi :
            maxi = f(k)
            ind = k
    return maxi, ind

def mini_ind_f(N):
    """
    N : int -> maximum de f(0), ..., f(N), et un indice en \
    ↪ lequel il est atteint
    """
    mini = f(0)
    ind = 0
    for k in range(N+1):
        if f(k) < mini :
            mini = f(k)
            ind = k
    return mini, ind
```

Pour le minimum c'est exactement la même fonction, il suffit de changer le symbole > en <.

```
>>> maxi_ind_f(10)
```

```
(25, 5)
>>> mini_ind_f(10)
(0, 0)
```

Solution (exercice 14) Énoncé

```
def cherche_ndiv(A):
    u = 2
    n = 0
    while u <= A:
        u = 3*u-1
        n += 1
    return n
>>> cherche_ndiv(10**(-3))
0
```

Solution (exercice 15) Énoncé

```
def cherche_ncv(eps):
    u = 4
    n = 0
    while abs(u-4/3) >= eps:
        u = 2 - u/2
        n += 1
    return n
>>> cherche_ncv(10**(-3))
12
```

Solution (exercice 16) Énoncé

```
def suiteU_Heron(n, a):
    u = a
    for _ in range(1, n+1):
        u = (u**2+a)/(2*u)
    return u
>>> suiteU_Heron(10**2, 2)
1.414213562373095
>>> ma.sqrt(2)
1.4142135623730951
```

```
def heron_approx(a, eps):
    u = a
    n = 0
    while abs(u-ma.sqrt(a)) >= eps:
        u = (u**2+a)/(2*u)
        n += 1
    return n
>>> heron_approx(167, 10**(-4))
7
>>> heron_approx(167, 10**(-8))
8
```

Solution (exercice 17) Énoncé

```
def syracuse(a, n):
    """
    retourne la liste des n premiers termes de syracuse
    """
    u = a
    for _ in range(1, n+1):
        if u % 2 == 0:
            u = u/2
        else:
            u = 3*u+1
    return u
>>> a = 10
>>> syracuse(a, 2)
16.0
>>> syracuse(a, 10)
4.0
```

Ensuite, on peut s'occuper du temps de vol.

```
def temps_vol(a):
    """
    retourne le temps du vol au-dessus de 1 de Syracuse
    """
    temps_vol = 0
    n = 0
    while syracuse(a, temps_vol) != 1:
        temps_vol += 1
    return temps_vol

def altitude_temps_vol(a):
```

```
    """
    retourne le temps du vol au-dessus de 1 de Syracuse
    """
    temps_vol = 0
    altitude_max = syracuse(a, temps_vol)
    n = 0
    while syracuse(a, temps_vol) != 1:
        temps_vol += 1
        valeur = syracuse(a, temps_vol)
        if valeur > altitude_max:
            altitude_max = valeur
    return temps_vol, altitude_max
```

Cette fonction n'est pas optimale puisqu'il est inutile de recalculer tous les termes de la liste à chaque fois.

```
>>> altitude_temps_vol(10)
(6, 16.0)
```