Python Files

./introduction/ex1.py

```
n = int(input("Entrer un entier: "))

for i in range(1, n+1):
    print("".join([str(j) for j in range(1, i+1)]))
```

./introduction/ex2.py

```
NBR_ELEVES = 3

class Etudiant():
    def __init__(self, nom, age, notes):
        self.nom = nom
        self.age = age
        self.notes = notes
        self.moyenne = 0

eleves = []

for _ in range(NBR_ELEVES):
    nom = input("Entrer le nom de l'élève: ")
    age = int(input("Entrer l'age de l'élève: "))
    notes = [int(input("Entrer une note : ")) for _ in range(3)]
    eleves.append(Etudiant(nom, age, notes))

# Calcule de la moyenne, de la meilleure note et de la moins bonne note

moyenne = 0
note_min = 21
note_max = 0

for eleve in eleves:

    # Scrap de toutes les notes pour la pire et la meilleure
    for note in eleve.notes:
        # Meilleure note
        if note > note_max :
            note_max = note

        # Pire note
        if note < note_min:
            note_min = note
```

```
    moyenne += sum(eleve.notes)

print(f"La moyenne est de : {moyenne/len(eleves*3)} et les notes vont de
{note_min} à {note_max}")
```

./introduction/ex3.py

```
"""
On veut récupérer depuis le fichier donné en ligne de commande:
    - Le nombre de caractères
    - Le nombre de lignes
    - Le nombre de mots
    - Tous les mots distincts
    - Les vingts premiers mots du fichier
"""

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("file", help="Le fichier à analyser")

args = parser.parse_args()

# On ouvre le fichier
with open(args.file, "r") as f:
    # On récupère le contenu
    contenu = f.read()

    # On compte le nombre de caractères
    nbr_caracteres = len(contenu)

    # On compte le nombre de lignes
    nbr_lignes = len(contenu.splitlines())

    # On compte le nombre de mots
    nbr_mots = len(contenu.split())

    # On récupère tous les mots distincts
    mots_distincts = set(contenu.split())

    # On récupère les vingts premiers mots
    vingts_premiers_mots = contenu.split()[:20]

    # On affiche les résultats
    print(f"Nombre de caractères: {nbr_caracteres}")
    print(f"Nombre de lignes: {nbr_lignes}")
    print(f"Nombre de mots: {nbr_mots}")
    print(f"Mots distincts: {mots_distincts}")
    print(f"Vingts premiers mots: {vingts_premiers_mots}")
```

./introduction/ex4.py

```
import os
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("path", help="Le dossier à scanner")

args = parser.parse_args()

path = args.path

liste_fichiers = []

for root, dirs, fichiers in os.walk(path):

    for fichier in fichiers:
        liste_fichiers.append(os.path.join(root, fichier))

for nom in liste_fichiers :
    print(nom)
```

./introduction/ex5.py

```
"""
    Grâce aux données présentes dans le fichier /data/population.csv, on veut :
        - La population la plus élevée et l'année associée
        - La population la plus faible et l'année associée
        - La population moyenne
"""

import csv

data_path = "./data/population.csv"

population_totale = 0

p_max = []
p_min = []

with open(data_path) as fichier :

    reader = csv.reader(fichier)

    cpt = 0

    for row in reader:
```

```
        if row[0].isdecimal():

            cpt += 1

            y, p = int(row[0]), int(row[1])

            # Init
            if p_max == []:
                p_max = [y, p]
            if p_min == []:
                p_min = [y, p]

            # Récupération des données
            population_totale += p

            if p > p_max[1] :
                p_max = [y, p]
            if p < p_min[1] :
                p_min = [y, p]

moyenne = population_totale/(cpt)

print(f"Pmin : {p_min}, Pmax : {p_max}, Moyenne : {moyenne}")
```

./Projet/alexis.py

```
import multiprocessing as mp
import numpy as np
import signal
import os
import time
import math
import random
import sys
import ctypes
GRID_W = 20
GRID_H = 20

def effacer_ecran(): print(CLEARSCR, end='')
def erase_line_from_beg_to_curs(): print("\033[1K", end='')
def curseur_invisible(): print(CURSOFF, end='')
def curseur_visible(): print(CURSON, end='')
def move_to(lig, col): print("\033[" + str(lig) + ";" + str(col) + "f", end='')


def neighbors(X, row_number, column_number):
    return [[X[i][j] if i >= 0 and i < len(X) and j >= 0 and j < len(X[0]) else 0
             for j in range(column_number-1, column_number+2)]
            for i in range(row_number-1, row_number+2)]
```

```
def lifeStep(X, x, y):
    return np.sum(X[x-1:x+2, y-1:y+2]) - X[x, y]


isRunning = mp.Value(ctypes.c_bool, True)


def handler(signum, frame):
    isRunning.value = False
    exit(0)


def a_life(x, TickSem, UnTickSem, UiSem, isRunning, state1, state2):
    s1 = np.frombuffer(state1.get_obj(), ctypes.c_int)
    arr1 = s1.reshape((GRID_W, GRID_H))

    s2 = np.frombuffer(state2.get_obj(), ctypes.c_int)
    arr2 = s2.reshape((GRID_W, GRID_H))
    UnTickSem.release()

    while (isRunning.value):
        TickSem[x].acquire()

        for y in range(GRID_W):

            bit = arr1[y][x]

            nbn = lifeStep(arr1, y, x)

            arr2[y][x] = arr1[y][x]
            if arr1[y][x] == 1:
                if (nbn < 2) or (nbn > 3):
                    arr2[y][x] = 0
            else:
                if nbn == 3:
                    arr2[y][x] = 1
            bit = arr2[y][x]

            UiSem.acquire()
            move_to(y+1, x*2+1)
            # print('' if bit else ' ')
            print('X' if bit else f"  ")

            s1 = f"{CL_RED}{nbn}{CL_GRAY}"
            s2 = f"{CL_BLUE}{nbn}{CL_GRAY}"
            UiSem.release()
    return


def addGlider(x, y, a):
    """adds a glider with top left cell at (i, j)"""
    glider = np.array([[0,   0, 1],
```

```
                        [1,  0, 1],
                        [0,  1, 1]])
    a[x:x+3, y:y+3] = glider


def addStatic1(x, y, a):
    static = np.array([[0, 1, 0],
                       [0, 1, 0],
                       [0, 1, 0]])
    a[x:x+3, y:y+3] = static


if __name__ == "__main__":

    signal.signal(signal.SIGINT, handler)

    state1 = mp.Array(ctypes.c_int, GRID_H*GRID_W)
    state2 = mp.Array(ctypes.c_int, GRID_H*GRID_W)
    s1 = np.frombuffer(state1.get_obj(), ctypes.c_int)
    a1 = s1.reshape((GRID_W, GRID_H))
    s2 = np.frombuffer(state2.get_obj(), ctypes.c_int)
    # for i in range(GRID_H*GRID_W):
    #     grid[i] = i

    print(a1)

    addGlider(2, 1, a1)
    addStatic1(10, 10, a1)

    print(a1)

    process = [0 for y in range(GRID_H)]
    # TickSem = mp.Semaphore(0)
    TickSem = [mp.Semaphore(0) for x in range(GRID_H)]
    UnTickSem = mp.Semaphore(0)
    UiSem = mp.Semaphore(1)

    effacer_ecran()
    curseur_invisible()

    UiSem.acquire()
    move_to(GRID_H+2, 1)
    print("Loading...")
    UiSem.release()
    for x in range(0, GRID_H):
        process[x] = mp.Process(target=a_life, args=(
            x, TickSem, UnTickSem, UiSem, isRunning, state1, state2))
        process[x].start()
    UiSem.acquire()
    move_to(GRID_H+2, 1)
    print("Waiting...")
    UiSem.release()
    for i in range(GRID_H):
        UnTickSem.acquire()
```

```
        move_to(GRID_H+2, 1)
        print(f"Waiting... ({i+1}/{GRID_H})")
    UiSem.acquire()
    move_to(GRID_H+2, 1)
    print("Go !                        ")
    UiSem.release()
    while (isRunning.value):
        for x in range(0, GRID_H):
            TickSem[x].release()

        time.sleep(0.05)

        # Copy nextstate to the current state

        for i in range(GRID_H*GRID_W):
            s1[i] = s2[i]
    for x in range(0, GRID_H):
        process[x].kill()
    print("Fini")
```

./Projet/course-LALOI-BATTU-BALAGUER.py

```
"""

    4IRC
    Exercice course hippique
    Groupe :
        - Maxime BATTU
        - Eileen BALAGUER
        - Batiste LALOI

"""
# Cours hippique
import multiprocessing as mp

import os, time, math, random, sys, ctypes, numpy as np

# Constantes
NB_PROCESS = 20
OFFSET = 5

# Variables partagées
leaderboard = mp.Array(ctypes.c_int, NB_PROCESS)
resultats = mp.Array(ctypes.c_int, NB_PROCESS)
finished = mp.Value('i', 0)
rang = mp.Value(ctypes.c_int, 1)
gagnant = ""
perdant = ""

#-----------------------------------------------------
```

```
# Une liste de couleurs à affecter aléatoirement aux chevaux
lyst_colors=[CL_WHITE, CL_RED, CL_GREEN, CL_BROWN , CL_BLUE, CL_MAGENTA, CL_CYAN,
CL_GRAY,
             CL_DARKGRAY, CL_LIGHTRED, CL_LIGHTGREEN,  CL_LIGHTBLU, CL_YELLOW,
CL_LIGHTMAGENTA, CL_LIGHTCYAN]

def effacer_ecran() : print(CLEARSCR,end='')
def erase_line_from_beg_to_curs() : print("\033[1K",end='')
def curseur_invisible() : print(CURSOFF,end='')
def curseur_visible() : print(CURSON,end='')
def move_to(lig, col) : print("\033[" + str(lig) + ";" + str(col) + "f",end='')

def en_couleur(Coul) : print(Coul,end='')
def en_rouge() : print(CL_RED,end='') # Un exemple !

# Referee
def arbitre(keep_running, leaderboard) :
    while keep_running.value :
        move_to(NB_PROCESS+OFFSET+2, 10)
        erase_line_from_beg_to_curs()
        en_couleur(CL_WHITE)

        if finished.value == NB_PROCESS :
            move_to(NB_PROCESS+OFFSET+5+NB_PROCESS, 0)
            keep_running.value = False
        else :
            print(f"Leader : {chr(ord('A') + np.argmax(leaderboard))} - Dernier :
{chr(ord('A') + np.argmin(leaderboard))}")

        time.sleep(0.1)

# La tache d'un cheval
def un_cheval(ma_ligne : int, keep_running, leaderboard) : # ma_ligne commence à 0

    col=1

    while col < LONGUEUR_COURSE and keep_running.value :
        move_to(ma_ligne+1,col)         # pour effacer toute ma ligne
        erase_line_from_beg_to_curs()
        en_couleur(lyst_colors[ma_ligne%len(lyst_colors)])
        if col %2 :
            print(f"+|__{chr(ord('A')+ma_ligne)}__/{' '*(LONGUEUR_COURSE-col-
1)}|")
        else :
            print(f"x|__{chr(ord('A')+ma_ligne)}__/{' '*(LONGUEUR_COURSE-col-
1)}|")


        leaderboard[ma_ligne] = col

        col += 1
        time.sleep(0.1 * random.randint(1,5))
```

```
        if col == LONGUEUR_COURSE :
            finished.value += 1
            with resultats.get_lock():
                resultats[ma_ligne] = rang.value
                rang.value += 1
                move_to(ma_ligne + 1, LONGUEUR_COURSE+1 + 10)
                en_couleur(CL_WHITE)
                print(f"Finis ! {resultats[ma_ligne]}e")

# -----------------------------------------------------
# La partie principale :
if __name__ == "__main__" :

    LONGUEUR_COURSE = 100 # Tout le monde aura la même copie (donc no need to have
a 'value')
    keep_running=mp.Value(ctypes.c_bool, True)
    lettres = [chr(ord('A')+i) for i in range(NB_PROCESS)]
    predicton = input(f"Entrer une lettre entre A et {lettres[-1]} : ")

    while predicton not in lettres :
        predicton = input(f"Erreur ! Entrer une lettre entre A et {lettres[-1]} :
")

    mes_process = [0 for i in range(NB_PROCESS)]

    effacer_ecran()
    curseur_invisible()

    # Referee that looks fot the horse in first place and the one in last place
    arbitre = mp.Process(target=arbitre, args=(keep_running, leaderboard))
    arbitre.start()

    for i in range(NB_PROCESS):  # Lancer Nb_process processus
        mes_process[i] = mp.Process(target=un_cheval, args= (i,keep_running,
leaderboard))
        mes_process[i].start()

    for i in range(NB_PROCESS): mes_process[i].join()

    move_to(NB_PROCESS + OFFSET + 2, 1)
    print(CLEARELN)
    classement_final = [chr(ord('A') + i) for i in np.argsort(resultats)]
    print("Clasement : ")
    for i in range(NB_PROCESS):
        print(f"{i+1} : {classement_final[i]}")

    if classement_final[0] == predicton.upper() :
        print(f"Vous avez gagné ! Le gagnant est {classement_final[0]}")
    else :
        print(f"Vous avez perdu ! Le gagnant est : {classement_final[0]}")

    curseur_visible()
```

./Projet/fractal.py

```
# -*- coding: utf-8 -*-
import multiprocessing as mp
import os
import sys
import math
import time

size = 1000
zoo = pow(0.5, 13.0 * 0.7)
percentCompletion = 1

print(f"Image de taille {size}x{size}")
NB_PROCESS = int(input(f"Entrer un nombre de processus (Entre 1 et
{mp.cpu_count()}): "))
print(f"Rendu de l'image en cours avec {NB_PROCESS} processus...")

# Cette fonction permet de calculer la couleur d'un pixel en utilisant
def calcul(x, y, image, pixel_index):
    pass
"""
Maintenant créé une fonction qui peut créer 5 processus, chacun gérant le rendu
d'une ligne
Quand une ligne est terminée, le processus envoie le résultat au processus
principal, qui l'écrit dans l'image, et demande ensuite la ligne suivante à rendre
"""
def render_line(line, image):
    for x in range(size):
        calcul(x, line, image, 3*(line * size + x))

# Init the l'image dans un tableau partagé
image = mp.Array('B', size*size*3)

# monoProcess()
def monoProcess():
    for y in range(size):
        for x in range(size):
            calcul(x, y, image, 3*(y * size + x))

# multiProcess()
def multiProcess():
    #  Créer un pool de processus
    for pack in range(size//NB_PROCESS):
        # On les lance chacun sur la prochaine ligne à render
        for line in range(NB_PROCESS):
            p = mp.Process(target=render_line, args=(pack*NB_PROCESS+line, image))
            p.start()
        for line in range(NB_PROCESS):
            p.join()
```

```
        # Afficher la progression du rendu
        percentCompletion = round(pack * 100 / (size//NB_PROCESS))
        print("[" + "=" * (percentCompletion//2) + " " * (50 -
percentCompletion//2) + "] " + str(percentCompletion) + "%", end="\r")


# Commencer le timer
start_time = time.time()
multiProcess()
end_time = time.time()

# Afficher le temps d'exécution avec 2 chiffres après la virgule
print(f"\nTemps d'exécution: {end_time - start_time:.2f} secondes")

fd = os.open("image1.ppm", os.O_CREAT | os.O_WRONLY, 0o644)
os.write(fd, "P6\n{} {}\n255\n".format(size, size).encode())
os.write(fd, bytes(image))
os.close(fd)
```

./Projet/gameoflife.py

```
import time
import random
import os
import ctypes

import multiprocessing as mp
import numpy as np

TAILLE = 30
NB_PROCESS = 5

ITERATION = 50

def clear(): os.system("clear")
def move_to(lig, col) : print("\033[" + str(lig) + ";" + str(col) + "f",end='')

# Create a shared two dimensionnal array
shared_current_grid = mp.Array(ctypes.c_int, TAILLE * TAILLE)
shared_next_grid = mp.Array(ctypes.c_int, TAILLE * TAILLE)

# create a numpy array from the shared memory block
current_grid = np.frombuffer(shared_current_grid.get_obj(), ctypes.c_int)
next_grid = np.frombuffer(shared_next_grid.get_obj(), ctypes.c_int)

# reshape the array into a two dimensional array
current_grid = current_grid.reshape((TAILLE, TAILLE))
next_grid = next_grid.reshape((TAILLE, TAILLE))

def display_grid(grid):
```

```
    print("_" * (2 * TAILLE + 1))
    for line in grid:
        print("|", end="")
        for cell in line:
            if cell == 0:
                print("  ", end="")
            else:
                print("██", end="")
        print("|")
    print("¯" * (2*TAILLE + 1))

def addGlider(x, y, grid):
    glider = np.array([[0,   0, 1],
                       [1,  0, 1],
                       [0,  1, 1]])
    grid[x:x+3, y:y+3] = glider

# Fonction qui renvoie le nombre de voisins vivants d'une cellule donnée
def living_neighbours_count(x, y, grid):
    count = 0
    for i in range(-1, 2):
        for j in range(-1, 2):
            if i == 0 and j == 0:
                continue
            elif x + i < 0 or x + i >= TAILLE or y + j < 0 or y + j >= TAILLE:
                continue
            elif grid[x + i][y + j] == 1:
                count += 1
    return count

# Fonction qui traite une cellule
def render_cell(x, y, current_grid, next_grid):
    living_neighbours = living_neighbours_count(x, y, current_grid)
    if current_grid[x][y] == 1:
        if living_neighbours == 2 or living_neighbours == 3:
            next_grid[x][y] = 1
        else :
            next_grid[x][y] = 0
    else:
        if living_neighbours == 3:
            next_grid[x][y] = 1


# Fonction qui traite une ligne
def render_line(line, current_grid, next_grid):
    for cell in range(TAILLE):
        render_cell(line, cell, current_grid, next_grid)

# Ajout d'une glider dans la grille de départ
# addGlider(1, 1, current_grid)

# On remplis de manière aléatoire la grille, avec 20% de chance d'avoir une
cellule vivante
for i in range(TAILLE):
```

```
    for j in range(len(current_grid[i])):
        if random.randint(0, 4) == 0:
            current_grid[i][j] = 1

clear()

cpt = 0
while True:

    # Effacer l'écran
    move_to(0, 0)

    # Pack de processus
    for pack in range(TAILLE//NB_PROCESS):
        # print(f"Pack #{pack}")
        for line in range(NB_PROCESS) :
            p = mp.Process(target=render_line, args=(pack*NB_PROCESS + line,
current_grid, next_grid))
            p.start()

        for _ in range(NB_PROCESS):
            p.join()

    display_grid(next_grid)

    for i in range(TAILLE):
        for j in range(len(next_grid[i])):
            current_grid[i][j] = next_grid[i][j]

    # time.sleep(0)
    cpt += 1
    print(f"Génération numéro : {cpt}")
```

./Projet/main.py

```
import random
import random

N = 100

def formule(array):
    somme = 0

    formule = "math.sqrt(1 - math.pow(x, 2))"

    for nombre in array:
        f = formule.replace("x", str(nombre))
        somme += eval(f)

    return somme
```

```
# Tableau de N valeurs aléatoires entre 0 et 1
tableau = [random.random() for _ in range(N)]

print(f"Valeur de pi approchés{4*formule(tableau)/len(tableau)}")
```

./Projet/message.py

```
import time
import os
import random
import math

import multiprocessing as mp
import numpy as np

GRID_SIZE = 10

def clear() :
    os.system('cls' if os.name == 'nt' else 'clear')

# 1 :
def init_grid() :
    grid = np.random.randint(2, size=(GRID_SIZE, GRID_SIZE))
    return grid

# 2 :
def update_grid(grid) :
    new_grid = np.zeros((GRID_SIZE, GRID_SIZE))
    for x in range(GRID_SIZE) :
        for y in range(GRID_SIZE) :
            # Check if the current cell is outside of the grid bounds
            if x < 0 or x >= GRID_SIZE or y < 0 or y >= GRID_SIZE :
                continue

            live_neighbors = 0
            for i in range(-1, 2) :
                for j in range(-1, 2) :
                    if i == 0 and j == 0 :
                        continue
                    elif x + i < 0 or x + i >= GRID_SIZE or y + j < 0 or y + j >=
GRID_SIZE :
                        continue
                    elif grid[x + i][y + j] == 1 :
                        live_neighbors += 1
            if grid[x][y] == 1 :
                if live_neighbors == 2 or live_neighbors == 3 :
                    new_grid[x][y] = 1
            else :
                if live_neighbors == 3 :
                    new_grid[x][y] = 1
```

```
    return new_grid


# 3 :
def update_grid_parallel(grid) :
    pool = mp.Pool(5)
    new_grid = pool.map(update_grid, np.array_split(grid, 5))
    pool.close()
    return np.concatenate(new_grid)

# 4 :
def display_grid(grid) :
    for x in range(GRID_SIZE) :
        for y in range(GRID_SIZE) :
            if grid[x][y] == 1 :
                print('X', end='')
            else :
                print(' ', end='')
        print()

def main() :
    grid = init_grid()
    while True :
        clear()
        display_grid(grid)
        grid = update_grid_parallel(grid)
        time.sleep(0.5)

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

./tp1/moyenne.py

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("notes", help="Notes", nargs='*')

args = parser.parse_args()

notes = list(map(int, args.notes))

if notes == []:
    print("Aucune moyenne à calculer")
else :

    valide = True

    for note in notes :
```

```
        if note < 0 or note > 20 :
            valide = False
            print(f"Note(s) non valide(s) : {note}")

    if valide :
        moyenne = sum(notes)/len(notes)

        print(f"Moyenne = {moyenne:.2f}")
```

./tp1/processus.py

```
import os
import random

liste = [random.randint(0, 100) for i in range(10)]
print(liste, sum(liste))

def sommeImpairs():
    somme = 0
    for i in range(1, len(liste), 2):
        somme += liste[i]
    return somme

def sommePairs():
    somme = 0
    for i in range(0, len(liste), 2):
        somme += liste[i]
    return somme

def main():
    tube = os.pipe()
    pid = os.fork()

    if pid == 0:
        # Processus fils
        os.close(tube[0])
        # On écrit dans le tube la somme des éléments pairs
        os.write(tube[1], str(sommeImpairs()).encode())
        os.close(tube[1])
    else:
        # Processus "sous père"
        pid = os.fork()
        if pid == 0:
            # Processus fils
            os.close(tube[0])
            # On écrit la somme des pairs dans le tube
            os.write(tube[1], str(sommePairs()).encode())
            os.close(tube[1])
        else:
            # Processus père
            os.close(tube[1])
```

```
            somme = 0
            # Récupération des résultats
            for _ in range(2):
                somme += int(os.read(tube[0], 100).decode())
            print(somme)
            os.close(tube[0])

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

./tp1/compilation/compilation.py

```
import os
import argparse

# On récupère les noms des fichiers passés en paramètre
parser = argparse.ArgumentParser()
parser.add_argument("fichiers", nargs="*", help="Les fichiers à compiler")
args = parser.parse_args()

fichiers = args.fichiers

path = "./src/"

# Pour chaque fichier, on lance un processus fils
for fichier in fichiers:
    pid = os.fork()
    if pid == 0:
        # Processus fils
        # Compilation du fichier source et dépot dans le dossier obj
        os.system("gcc " + path + fichier + " -o obj/" + fichier[:-2] + ".out")

# On exécute les fichiers compilés
for fichier in fichiers:
    os.system("./obj/" + fichier[:-2] + ".out")
```