# R308 : Consolidation de la programmation

# TPD4 : Les Threads en Python

## 1 Synchronisation des Threads :

### 1.1 La classe Print est donnée :

```
class Print(Thread):
	def __init__(self, msg:str):
		Thread.__init__(self)
		self.__msg=msg
	def run(self)-> None:
		for car in self.__msg:
			print(car, end = "")
			time.sleep(0.01)
		print()
```

La méthode run() affiche le message passé en paramètre caractère par caractère avec un pause de 10 ms entre chaque.

### 1.2 Le programme principal :

Le programme principal est donnée

```
if __name__ == '__main__':
	# déclaraton des variables
	P1:Print = None
	P2:Print = None
	# instanciation des objets
	P1 = Print("bien le bonjour mes amis")
	P2 = Print("il est grand temps de se synchroniser …")
	# initialisation : effacer l'écran
	os.system("clear")
	# execution des threads
	P1.start()
	P2.start()
```

### 1.3 Travail à réaliser :

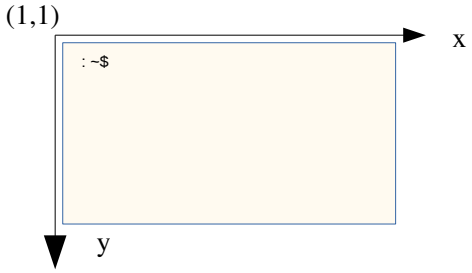
1. Vérifier que l'affichage proposé est incohérent.
2. Proposer trois solutions qui permettent d'afficher convenablement les messages

- utilisation d'un verrou en paramètre du constructeur de la classe Print
- utilisation d'un verrou comme variable de la classe Print
- utilisation de la méthode joint() de la classe Thread.

## 2 Héritage multiple :

### 2.1 Affichage d'un point à un endroit quelconque de l'écran :

Précision sur l'orientation de la console :




Testez la fonction suivante :

```
def affiche_car(x:int, y:int, car:str)-> None:
	# méthode générique
	# utilisation des codes d'échappement ANSI pour l'affichage dans la console
	# print("\033[y;xHchaine") Using ANSI escape sequence, moves curser to row y, col x:
	chaine:str = f"\033[{y};{x}H{car}"
	print(chaine)
```

à l'aide du programme suivant :

Remarque : vous utiliserez un terminal pour lancer votre programme.

```
import os
LARGEUR:int = 80
```

```
HAUTEUR:int = 22

if __name__ == "__main__":
	os.system("clear")
	affiche_car(1,1,"x")
	affiche_car(1,HAUTEUR,"e")
	affiche_car(LARGEUR,1,'°')
	affiche_car(LARGEUR,HAUTEUR,"_")
```

## 2.2 La classe Point « pseudo graphique » :

La classe Point est donnée. En utilisant les caractères d'échappement ANSI, elle permet d'afficher ou d'effacer un caractère à un endroit donné de la console.

```
class Point:
	def __init__(self, x: int=0, y:int= HAUTEUR, car: str="")-> None:
		self.__x = x
		self.__y = y
		self.__car = car
	def affiche_car(self,x: int, y: int, car: str)-> None:
		# méthode générique
		# utilisation des codes d'échappement ANSI pour l'affichage dans la console
		# print("\033[y;xHchaine") Using ANSI escape sequence, moves curser to row y, col x:
		chaine: str = f"\033[{y};{x}H{car}"
		print(chaine)
	def affiche(self)-> None:
		# affiche le caractère à la position (x,y)
		self.affiche_car(self.__x, self.__y, self.__car)
		self.derniere_ligne()
	def efface(self)-> None:
		# affiche un espace à la position (x,y)
		self.affiche_car(self.__x, self.__y, " ")
		self.derniere_ligne()
	def derniere_ligne(self)-> None:
		# positionne le prompt en derniere ligne
		self.affiche_car(1,HAUTEUR,"")
```

## 2.3 Premier travail :

Écrivez un programme qui affiche et efface quelques points dans une liste

Remarque : pour la suite du programme, vous pourrez utiliser la fonction suivante pour afficher un message sur la dernière ligne

```
def affiche_derniere_ligne(msg:str)->None:
	# compléter la ligne avec des espaces
	i:int = len(str(msg))+1
	while i < LARGEUR:
		msg += " "
		i += 1
	print(f"\033[{HAUTEUR+1};1H{msg}")
```

## 2.4 La classe EtoileEphemere :

Une étoile éphémère est un point qui possédera également deux temps :

- un temps de sommeil avant son apparition.
- un temps durant lequel elle est visible.

Chaque étoile sera représentée par le caractère *

De plus, chaque étoile sera totalement indépendante (un thread spécifique)

1. la représentation UML de la classe EtoileEphemere est la suivante :

| Classe EtoileEphemere (Thread, Point) | Héritage multiple de Thread et de Point |
|---|---|
| Attributs :<br>__t_sommeil:float<br>__t_visible:float<br>__visible:bool | __t_sommeil: temps avant affichage<br>__t_visible: temps d'affichage de l'étoile<br>__visible: booléen qui indique si l'étoile est visible ou pas |
| Méthodes : | |
| __init__(self,x:int,y:int,t_sommeil:float, t_visible:float): | Sauvegarde des attributs<br>Appels aux constructeurs des classes mères |
| run(self)-> None: | Attend avant de rendre visible l'étoile<br>Affiche l'étoile pendant le temps de visibilité<br>Efface l'étoile |
| get_visible(self)->bool: | Observateur qui retourne le booléen __visible |

2. Ecrivez la définition de la classe EtoileEphemère

### 2.5 Le programme principal :

1. Le programme principal partiel est donné :

```
if __name__ == '__main__':
	# declaration des variables
	liste_etoiles:List[Etoile_ephemere]
	nb_etoiles:int
	nb:int
	x:int # compris entre 1 et LARGEUR
	y:int # compris entre 1 et HAUTEUR
	duree:float # duree = temps maxi
	temps_sommeil:float
	tempo_visible:float
	# initialisation et instanciation
	liste_etoiles = []
	nb_etoiles = 100
	duree = 5.0
	os.system("clear")
	# boucle pour générer les etoiles et les ajouter à la liste
	nb = 0
	while nb < nb_etoiles:
		# generer aléatoirement x, y, temps_sommeil, temps_visible
		...
		# ajouter une nouvelle étoile à la liste
		...
	# boucle pour lancer les threads
	for etoile in liste_etoiles:
		...
	# compter le nombre d'étoiles toutes les 30 ms
	...
		affiche_derniere_ligne(f"nb etoiles:{nb_etoiles}")
```

2. Compléter le programme principal aux endroits indiqués
3. Compléter le programme avec la fonction "compte_etoiles" qui permettra de compter le nombre d'étoiles visibles.

Le prototype de la fonction sera :

```
def compte_etoiles(arg:Tuple)-> int:
```

La liste des étoiles sera passée sous la forme d'un tuple … pour préparer une question de l'exercice suivant (utilisation d'un timer)

# 3 Création d'un timer :

https://docs.python.org/fr/3.6/library/threading.html

### 3.1 Timer Objects¶

This class represents an action that should be run only after a certain amount of time has passed — a timer. `Timer` is a subclass of `Thread` and as such also functions as an example of creating custom threads.
Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

```
def hello():
    print("hello, world")

t = Timer(interval=3, function=hello)
t.start()  # after 3 seconds, "hello, world" will be printed
```

```
class threading.Timer(interval, function, args=None, kwargs=None)
```

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.
Modifié dans la version 3.3: changed from a factory function to a class.

```
cancel()
```

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

### 3.2 Ecrivez une fonction fonction_timer qui affiche un message toutes les 2 secondes :

Indication : la fonction se relancera elle-même toutes les 2 secondes

… comment mettre fin au programme ?

### 3.3 Evolution de la fonction : un arrêt propre :

Votre fonction devra maintenant avoir le prototype suivant :

```
def fonction_timer(msg:str, tempo:int, nb_appels:int)->None:
```

Complétez la fonction de telle sorte que l'affichage ne se produise que 5 fois.

Remarque : le nombre d'appels est un paramètre de la fonction et pourra être incrémenté avant chaque nouvel appel.

### 3.4 La classe MonTimerLanceUneFonction :

Votre classe MonTimerLanceUneFonction va permettre d'exécuter une fonction à intervalles réguliers sans toutefois connaître le corps de la fonction. Les paramètres éventuels seront précisés dans un tuple.

Le diagramme UML de la classe est le suivant :

| **class MonTimerLanceUneFonction:** | |
|---|---|
| self.__tempo = tempo<br>self.__fonction:function = fonction<br>self.__args:tuple = args<br>self.__timer:Timer | <br>#(1)<br>#(2)<br> |
| def __init__(self, tempo:float, fonction:object, args:Tuple ):<br>def start(self)-> None:<br>def stop(self)-> None: | <br>#(3)<br>#(4) |

(1) La fonction est un paramètre du constructeur
(2) args est un tuple qui contient les paramètres de la fonction.
(3) La méthode start() permettra de lancer le timer
(4) La méthode stop() permettra d'arrêter le timer.

1. Ecrivez la définition de la classe **MonTimerLanceUneFonction**
2. Reprendre l'exercice précédent afin de calculer et afficher un nombre d'étoiles visibles en utilisant votre timer.

# 4 Exercice complémentaire ... un problème de baignoire:

On souhaite modéliser le comportement du remplissage d'une baignoire qui a une fuite.

## 4.1 La classe Baignoire est donnée :

Une Baignoire possède un volume et un maximum (en litres d'eau)
La méthode remplir() permet d'augmenter le volume d'eau de la baignoire.
La méthode fuite() diminue le volume d'eau de la baignoire.
L' attribut __tempo_remplissage permet de simuler le débit de remplissage.
L' attribut __tempo_fuite permet de simuler le débit de fuite.
L'attribut __fin permet de stopper de savoir si la baignoire est pleine ou vide.

```
class Baignoire:
def __init__(self, vol_max:int, vol_init:int, tempo_remplissage:float, tempo_fuite:float):
	self.__volume_maxi:int = vol_max
	self.__volume:int = vol_init
	self.__tempo_remplissage:float = tempo_remplissage
	self.__tempo_fuite:float = tempo_fuite
	self.__fin:bool = None

def remplir(self, qantite:int )->None:
	self.__fin = False
	while not(self.__fin) and (self.__volume + qantite < self.__volume_maxi):
		self.__volume += qantite
		time.sleep(self.__tempo_remplissage)
		print(f" remplissage:volume de la baigoire:{self.__volume:.2f}")
	if not(self.__fin):
		self.__volume = self.__volume_maxi
		print(f" remplissage:volume de la baigoire:{self.__volume:.2f}")
		self.__fin = True

def fuite(self, qantite:int )->None:
	self.__fin = False
	while not(self.__fin) and (self.__volume - qantite > 0):
		self.__volume -= qantite
		time.sleep(self.__tempo_fuite)
		print(f" fuite:volume de la baigoire:{self.__volume:.2f}")
	if not(self.__fin):
		self.__volume = 0
		print(f" fuite:volume de la baigoire:{self.__volume:.2f}")
		self.__fin = True

def get_volume(self)-> float:
	return self.__volume
```

**Écrire un programme de test qui met en œuvre la baignoire qui se remplit puis fuit.**

## 4.2 Modifier votre programme de test afin de lancer dans les méthodes remplir() et vider() dans des threads.

Vous pourrez faire plusieurs essais notamment avec les paramètres ci dessous :

| Volume maxi (litre) | Volume initial (litre) | Tempo remplissage (seconde) | Tempo fuite (seconde) | Quantité remplissage (litre) | Quantité fuite (litre) |
|---|---|---|---|---|---|
| 100 | 50 | 0,5 | 0,3 | 10 | 8 |
| 100 | 50 | 0,5 | 0,3 | 10 | 5 |

Vérifiez que dans le premier cas la baignoire se vide et que dans le second cas elle se remplie.

## 4.3 Complétez votre classe pour qu'elle puisse mesurer le temps nécessaire pour rempli ou vider la baignoire.

Vous pourrez complétez votre classe Baignoire avec 3 attributs :

```
	__t0:float16		# qui mémorisera l'instant initial
	__t_fin:float16		# qui mémorisera l'instant final
```

Vous ajouterez également un méthode get_temps() qui retournera le temps écoulé.

Remarque : La fonction time() du module time retourne l'heure actuelle en secondes sous la forme d'un float.