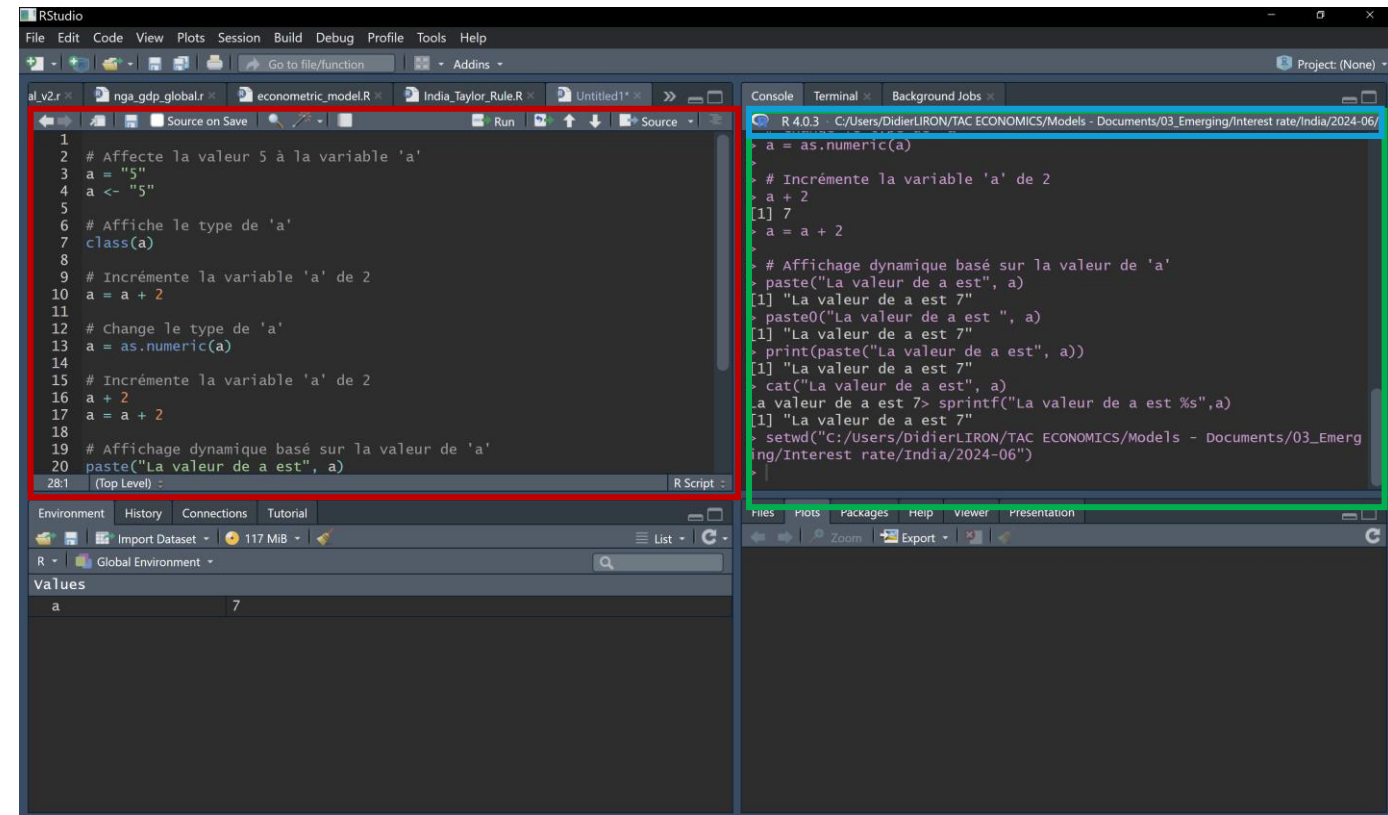


# Introduction à R et Python

M1 IEF - 2024



1. Qu'est ce que R ?
2. Qu'est ce que Rstudio ?
3. Différence **script** vs **console**
4. Créer un environnement sain de développement pour un projet
5. Qu'est ce qu'un **repertoire de travail** ?
6. Qu'est ce qu'une librairie (et le CRAN) ?



The screenshot displays the RStudio environment with several panes. The top-left pane shows an R script with the following code:

```
1  
2 # Affecte la valeur 5 à la variable 'a'  
3 a = "5"  
4 a <- "5"  
5  
6 # Affiche le type de 'a'  
7 class(a)  
8  
9 # Incrémente la variable 'a' de 2  
10 a = a + 2  
11  
12 # Change le type de 'a'  
13 a = as.numeric(a)  
14  
15 # Incrémente la variable 'a' de 2  
16 a + 2  
17 a = a + 2  
18  
19 # Affichage dynamique basé sur la valeur de 'a'  
20 paste("La valeur de a est", a)
```

The top-right pane shows the R console with the following output:

```
R 4.0.3 C:/Users/DidierLIRON/TAC ECONOMICS/Models - Documents/03_Emerging/Interest rate/India/2024-06/  
a = as.numeric(a)  
# Incrémente la variable 'a' de 2  
a + 2  
[1] 7  
a = a + 2  
# Affichage dynamique basé sur la valeur de 'a'  
paste("La valeur de a est", a)  
[1] "La valeur de a est 7"  
paste0("La valeur de a est ", a)  
[1] "La valeur de a est 7"  
print(paste("La valeur de a est", a))  
[1] "La valeur de a est 7"  
cat("La valeur de a est", a)  
La valeur de a est 7> sprintf("La valeur de a est %s",a)  
[1] "La valeur de a est 7"  
setwd("C:/Users/DidierLIRON/TAC ECONOMICS/Models - Documents/03_Emerging/Interest rate/India/2024-06")
```

The bottom-left pane shows the Environment pane with the following values:

Values
a
7

# Plan

01

Les variables

02

Les structures de  
contrôle

03

Les fonctions

04

Manipuler les  
dataframes

05

Les séries temporelles

06

Les APIs

## 1. Les types

- Chaines de caractères: **character**
- Numériques: **numeric**
- Booleens: **logical**
- Vide: **NULL**

## 2. Les vecteurs: `c(1, 2, « Hello »)`

## 3. Afficher un résultat: `paste`, `paste0`, `print`, `cat`, `sprintf`

## 4. Importer un fichier

- Chemin relatif vs absolue (`./`, `../`)
- `csv`, `xlsx`
- Vérifier les types des colonnes

```
# Affecte la valeur 5 à la variable 'a'
a = "5"
a <- "5"

# Affiche le type de 'a'
class(a)

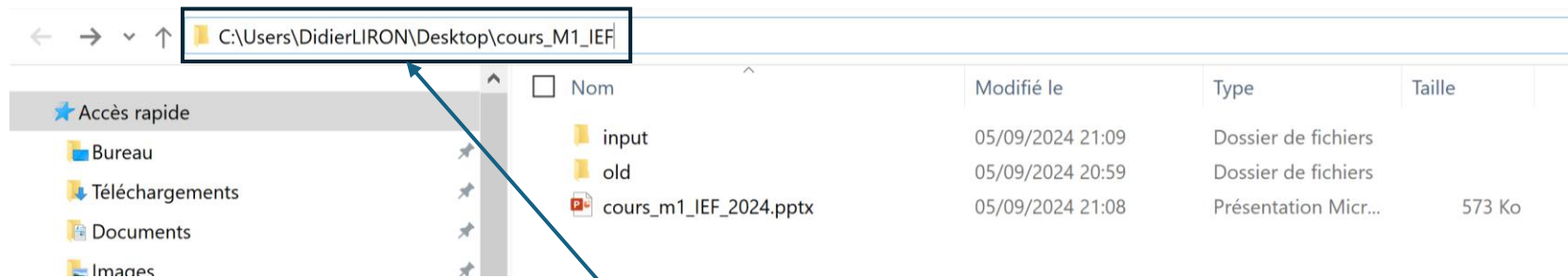
# Incrémente la variable 'a' de 2
a = a + 2

# Change le type de 'a'
a = as.numeric(a)

# Incrémente la variable 'a' de 2
a + 2
a = a + 2

# Affichage dynamique basé sur la valeur de 'a'
paste("La valeur de a est", a)
paste0("La valeur de a est ", a)
print(paste("La valeur de a est", a))
cat("La valeur de a est", a)
sprintf("La valeur de a est %s", a)
```

Etant donné un repertoire de travail fixé à cet endroit :



**Chemin absolu :** « C:\Users\DidierLIRON\Desktop\cours\_M1\_IEF »

**Chemin relatif :** « . »

Pour aller chercher un fichier dans le repertoire input :

`./input/[monFichier]`

→ Pointe sur le repertoire **courant**

Pour aller chercher un fichier dans le repertoire précédent (Desktop):

`../[monFichier]`

→ Pointe sur le repertoire **précédent**

```
# Fixe le repertoire de travail
setwd(« C:\Users\DidierLIRON\Desktop\cours_M1_IEF »)

# Lire le fichier data.csv situé dans le repertoire input
# Chemin absolu
read.csv(« C:\Users\DidierLIRON\Desktop\cours_M1_IEF
/input/data.csv »)
# Chemin relatif
read.csv(« ./input/data.csv »)

# Lire le fichier dataprev.csv situé dans le repertoire
Desktop
# Chemin absolu
read.csv(« C:\Users\DidierLIRON\Desktop\dataPrev.csv »)
# Chemin relatif
read.csv(« ../dataPrev.csv »)
```

Fichier data.csv:

Separateur de colonnes « , »  
Separateur de décimales « . »

```
# Importer le fichier csv
read.csv(« data.csv », sep=« , », dec= « . »)
```

Notepad++

Excel

```
"date","usgdp","sp500","usbond10","uscpi","tusgdp","usinfl"
1947-01-01,2034.45,NA,NA,21.7,NA,NA
1947-04-01,2029.024,NA,NA,22.01,NA,NA
1947-07-01,2024.834,NA,NA,22.49,NA,NA
1947-10-01,2056.508,NA,NA,23.1266666666667,NA,NA
1948-01-01,2087.442,NA,NA,23.6166666666667,2.60473346604733,8.83256528417819
1948-04-01,2121.899,NA,NA,23.9933333333333,4.57732387591276,9.01105558079662
1948-07-01,2134.056,NA,NA,24.3966666666667,5.39412119709566,8.47784200385355
1948-10-01,2136.44,NA,NA,24.1733333333333,3.88678283770354,4.5257999423465
1949-01-01,2107.001,NA,NA,23.9433333333333,0.936984117403039,1.3832039520113
1949-04-01,2099.814,NA,NA,23.9166666666667,-1.04081296989159,-0.319533203667677
1949-07-01,2121.493,NA,NA,23.7166666666667,-0.588691205854019,-2.78726601994808
1949-10-01,2103.688,NA,NA,23.66,-1.53301754320271,-2.12355212355212
1950-01-01,2186.365,NA,NA,23.5866666666667,3.76668069924977,-1.48962828901574
1950-04-01,2253.045,NA,NA,23.7666666666667,7.29736062336952,-0.627177700348436
1950-07-01,2340.112,NA,NA,24.2033333333333,10.3049597618281,2.05200281096276
1950-10-01,2384.92,NA,NA,24.6933333333333,13.3685223284061,4.36742744435052
```

date	usgdp	sp500	usbond10	uscpi	tusgdp	usinfl
01/01/1947	2034.45	NA	NA	21.7	NA	NA
01/04/1947	2029.024	NA	NA	22.01	NA	NA
01/07/1947	2024.834	NA	NA	22.49	NA	NA
01/10/1947	2056.508	NA	NA	23.1266667	NA	NA
01/01/1948	2087.442	NA	NA	23.6166667	2.60473347	8.83256528
01/04/1948	2121.899	NA	NA	23.9933333	4.57732388	9.01105558
01/07/1948	2134.056	NA	NA	24.3966667	5.3941212	8.477842
01/10/1948	2136.44	NA	NA	24.1733333	3.88678284	4.52579994
01/01/1949	2107.001	NA	NA	23.9433333	0.93698412	1.38320395
01/04/1949	2099.814	NA	NA	23.9166667	-1.040813	-0.3195332
01/07/1949	2121.493	NA	NA	23.7166667	-0.5886912	-2.787266
01/10/1949	2103.688	NA	NA	23.66	-1.5330175	-2.1235521
01/01/1950	2186.365	NA	NA	23.5866667	3.7666807	-1.4896283
01/04/1950	2253.045	NA	NA	23.7666667	7.29736062	-0.6271777
01/07/1950	2340.112	NA	NA	24.2033333	10.3049598	2.05200281
01/10/1950	2384.92	NA	NA	24.6933333	13.3685223	4.36742744

### 1. Les boucles

- for : parcourt tous les éléments d'un vecteur et exécute autant de fois un code donné

```
for(i in vecteur) {  
    code  
}
```

- while : Exécute un code donné tant qu'une condition n'est pas remplie

```
while(condition){  
    code  
}
```

### 2. Les conditions: exécute ou non un code en fonction d'une condition

```
if (condition) {  
    code  
} else if {  
    code  
} else {  
    code  
}
```

```
# parcourt le vecteur c(1,2,3,4) et affiche les valeurs  
for(i in c(1,2,3,4)) {  
    print(i)  
}
```

```
# parcourt un autre vecteur et affiche les valeurs  
for(i in c("Bonjour","tout","le","monde","!")) {  
    print(i)  
}
```

```
# Incrémente i de 1 jusqu'à ce que i vaille 5
```

```
i = 1  
while( i < 5) {  
    print(i)  
    i = i + 1  
}
```

```
# Affichage des valeurs de a sous certaines conditions
```

```
a = 7
```

```
if(a == 7) {  
    print("a = 7")  
} else if(a == 10) {  
    print("a = 10")  
} else {  
    print("a n'est ni égal à 7 ni à 10.")  
}
```

2 - Les combinaisons de booléens

a	b	a ET b
F	F	F
F	V	F
V	F	F
V	V	V

a	b	a OU b
F	F	F
F	V	V
V	F	V
V	V	V

A	B	C	A ET B	A ET C	(A ET B) OU (A ET C)
F	F	F	F	F	F
F	F	V	F	F	F
F	V	F	F	F	F
F	V	V	F	F	F
V	F	F	F	F	F
V	F	V	F	V	V
V	V	F	V	F	V
V	V	V	V	V	V

# Quelques exemples R des combinaisons de booléens

```
> TRUE & TRUE
[1] TRUE

> TRUE & FALSE
[1] FALSE

> TRUE | TRUE
[1] TRUE

> TRUE | FALSE
[1] TRUE

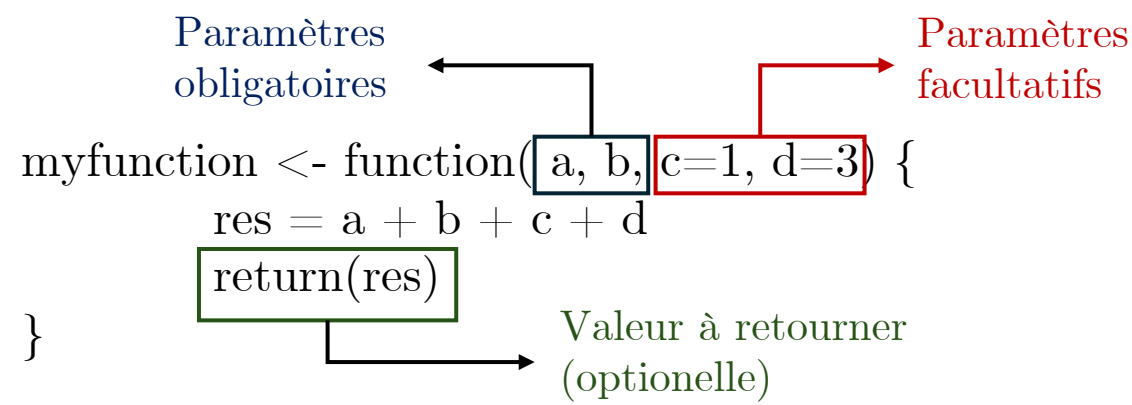
> (TRUE & FALSE) | FALSE
[1] TRUE
```



1. Vérifier si un nombre est premier ou pas. Pour savoir si un nombre est divisible par un autre, utilisez le modulo (`%%` en R)
2. Les deux premiers termes de la suite de Fibonacci sont tous deux un. Les termes suivants de la séquence sont trouvés en additionnant les deux termes immédiatement précédents. Écrivez un code qui écrit les  $n$  termes de la séquence de Fibonacci pour tout  $n \geq 3$
3. Écrivez un code qui simule le lancer d'une pièce avec une probabilité  $p$  associée aux piles, jusqu'à obtenir face, et renvoie le nombre de piles obtenues. Ensuite, reproduisez l'exécution de ce code 100 fois et afficher le nombre maximum de piles obtenus lors d'une même séquence. Pour générer une probabilité entre 0 et 1, utilisez `runif(1)`.

### 3 - Les fonctions

Une fonction informatique est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande (l'appel de la fonction).



Toutes les variables contenues dans une fonction (paramètres inclus) sont 'locales' et ne sont accessibles que dans cette fonction. Elles sont créées à l'appel de la fonction et détruites à la fin de son exécution. Par opposition aux variables déclarées partout ailleurs dans le code, dites 'globales' et accessibles partout (y compris dans une fonction).

```
# Déclaration d'une fonction
myfunction <- function(a, b, c = 1, d = 3) {
  res = a + b + c + d
  return(res)
}

# Différents appels à ma fonction
myfunction(a = 1, b = 2)
myfunction(1, 2)
xx = myfunction(a = 1, c = 2, b = 3)

# Illustrations variables locales/globales
res = 2
myfunction(1, 2)
print(res)

myfunction <- function(a, b, c = 1, d = 3) {
  res <- a + b + c + d
  return(res)
}
res = 2
myfunction(1, 2)
print(res)
```

1. Vérifier si un nombre est premier ou pas. Pour savoir si un nombre est divisible par un autre, utilisez le modulo (`%%` en R)  
=> Transformer votre réponse en utilisant une fonction avec un seul paramètre **obligatoire** définissant le nombre à vérifier.
2. Les deux premiers termes de la suite de Fibonacci sont tous deux un. Les termes suivants de la séquence sont trouvés en additionnant les deux termes immédiatement précédents. Écrivez un code qui écrit les  $n$  termes de la séquence de Fibonacci pour tout  $n \geq 3$   
=> Transformer votre réponse en utilisant une fonction avec un seul paramètre **obligatoire**  $n$  définissant le nombre de termes à ajouter à la suite de Fibonacci.
3. Écrivez un code qui simule le lancer d'une pièce avec une probabilité  $p$  associée aux piles, jusqu'à obtenir face, et renvoie le nombre de piles obtenues. Ensuite, reproduisez l'exécution de ce code 100 fois et afficher le nombre maximum de piles obtenus lors d'une même séquence. Pour générer une probabilité entre 0 et 1, utilisez `runif(1)`.  
=> Transformer votre réponse en utilisant une fonction avec deux paramètres **facultatifs**. Un premier pour définir la probabilité associée aux piles (par défaut 0.5), et un second pour le nombre de lancers (par défaut 100).

Format matriciel avec des colonnes nommées, et des individus (lignes) pouvant l'être également (par défaut 1:n). Il a l'avantage de pouvoir contenir des données de types mixtes.

Fonctionne comme les matrices pour accéder aux données, avec en plus des noms de colonnes (obligatoire, par défaut R nomme lui-même la colonne si besoin).

mydf[ , ]

Sélectionner des lignes ←      Sélectionner des colonnes →

Il faut savoir que bien qu'un data frame se présente sous forme d'une matrice, en réalité il s'agit d'une liste particulière dont les éléments ne peuvent être que des vecteurs de même longueur.

### # Création d'un dataframe

```
taille = c(121, 156, 194)
poids = c(67, 78, 235)
sexe = c("Homme","Femme","Homme")
mydata = data.frame(taille = taille, poids = poids, sexe = sexe)
```

### # Operations de base

```
length(mydata)
dim(mydata)
summary(mydata)
colnames(mydata)
rownames(mydata)
nrow(mydata)
ncol(mydata)
```

```
mydata[c(2, 3), ] # les lignes 2 et 3
mydata[-c(2, 3), ] # toutes les lignes sauf 2 et 3
mydata[, c(2, 3) ] # les colonnes 2 et 3
mydata[, -c(2, 3)] # toutes les colonnes sauf 2 et 3
mydata[c(2, 3), -c(2, 3)] # les lignes 2 et 3, toutes les colonnes
sauf 2 et 3
mydata$poids # colonne "poids" (resultat = vecteur)
mydata["poids"] # colonne "poids" (resultat = dataframe)
```

### # Sous-selection conditionnelle

```
mydata[mydata$sexe == "Homme", ]
```

- Afficher les dimensions d'un dataframe: `dim(df)`
- Concatener deux dataframes côtes à côtes: `cbind`, comme Colonne bind
- Concatener deux dataframes l'un sous l'autre: `rbind`, comme Row bind
- Afficher les noms de colonnes: `colnames(df)`
- Afficher les noms de lignes: `rownames(df)`
- Renommer les colonnes: `colnames(df) = c(...)`
- Renommer les lignes: `rownames(df) = c(...)`
- Afficher les N premières lignes: `head(df, N)`
- Afficher les N dernières lignes: `tail(df, N)`

### Exercice:

1. Importer les fichiers « `eia_data.csv` » et « `brent.csv` » à l'aide de la fonction `read_table` (en faisant attention aux séparateurs de colonnes, et avec la première ligne comme noms de colonnes). Faites de même avec `read.csv`
2. Vérifier qu'ils ont les bonnes dimensions pour les fusionner
3. Concatener les deux dataframes (ne prendre qu'une seule colonne « date »)
4. Renommer les colonnes en :  
`c("date", "cocpopec", "copsoppec", "paprnonopec", "papropec", "paprrus", "paprur", "pascoecd", "patcchn", "patcnonoecd", "patcoecd", "patcwld", "brent")`

```
# Afficher les dimensions (lignes, colonnes)
dim(mydata)

# Ajouter une colonne "test" de deux manières
mydata = cbind(mydata, test = c(1,2,3))
Mydata$test = c(1,2,3)

# renommer les lignes
rownames(mydata) = c("ind1", "ind2", "ind3")

# Afficher la première et la dernière ligne
head(mydata, 10)
tail(mydata, 10)
```

- Transformer une colonne en facteur: `as.factor(df["..."])`  
Une colonne en facteur ne peut contenir que des données définies dans ce facteur, sinon cela crée un NA
- Afficher les valeurs distinctes d'une colonne: `table(df["..."])`
- Afficher les statistiques d'un dataframe: `summary(df)`
- Trier un df par rapport à une colonne: `df[order(df["..."]), ]`
- Faire une opération matricielle en ligne ou en colonne (ici la moyenne):
  - En ligne : `apply(df, 1, function(x) mean(x) )`
  - En colonne: `apply(df, 2, function(x) mean(x) )`
- Faire une opération glissante sur N périodes (ici moyenne mobile):
  - `rollapply(df, N, function(x) mean(x) )`
- Passer d'un format long à wide et inversement:
  - Long => wide: `cast(df, ...)`
  - Wide => long: `melt(df, ...)`

```
# Créer un DF et transformer une colonne en facteur
taille = c(121, 156, 194)
poids = c(67, 78, 235)
sexe = c("Homme","Femme","Homme")
mydata = data.frame(taille = taille, poids = poids, sexe = sexe)

mydata$sexe = as.factor(mydata$sexe)

# Importer un fichier csv
df = read.csv("input/eia_data.csv")

# Faire la moyenne des lignes
apply(df, 1, mean)
apply(df, 1, function(x) mean(x) )
# Ajouter une nouvelle colonne au DF avec cette moyenne
df["meanrow"] = apply(df, 1, function(x) mean(x) )
# Calculer une moyenne mobile 10 jours
library(zoo)
rollapply(df, 10, function(x) mean(x))

# Passage à un format long
library(reshape2)
df_melt = melt(df, id.vars = "date", variable.name="symbol")

# Repasser à un format wide
dcast(df_melt, formula = "date ~ symbol", fun.aggregate = mean)
```

Une série temporelle est un format matriciel ayant un index au **format date**. Il est ensuite possible d'appliquer un grand nombre d'opération sur ces séries, comme les lags.

- Format natif de R: le **ts**
  - Connait le temps, dans le sens ou il sait créer des dates
  - Ne permet de traiter que les fréquence Annual, Quarterly et Monthly
  - Sous-selection temporelle à l'aide de la fonction window
- Format importable : **xts**
  - Ne connait pas le temps, il faut lui spécifier les dates. Il n'est pas capable d'en créer lui-même
  - Permet de traiter toutes les fréquences
  - Fonctionne exactement comme un Data Frame
  - Sous-selection temporelle avec les crochets
    - `mon_xts["2021-01-01"]` : selectionne le 1<sup>er</sup> janvier 2021
    - `mon_xts["2021-01-01/"]` : selectionne depuis le 1<sup>er</sup> janvier 2021
    - `mon_xts[« /2021-01-01"]` : selectionne jusqu'au 1<sup>er</sup> janvier 2021
    - `mon_xts["2020-01-01/2021-01-01"]` : selectionne du 1<sup>er</sup> janvier 2020 jusqu'au 1<sup>er</sup> janvier 2021

```
# Créer un ts avec des données mensuelles
my_ts = ts(1:100, start = 2021, frequency = 12)
# Sélectionner l'année 2022
window(my_ts, start = 2022, end = c(2022,12))

# Créer un xts avec des données mensuelles
library(xts)
dates = as.Date(c('2024-01-01', '2024-02-01',
'2024-03-01', '2024-04-01', '2024-05-01'))
my_xts <- xts(1:5, order.by = dates)
# Sélectionner le premier trimestre 2024
my_xts["2024-01/2024-03"]
```

- Créer un lag (**attention, ils n'ont pas été implémentés dans le même sens !**)
  - `ts: lag(my_ts, -1)` -> un nouveau point est crée car le `ts` est capable de créer le temps
  - `xts: lag(my_xts, 1)` -> aucun nouveau point n'est créé car le `xts` ne sait pas créer le temps, on perd donc une valeur
- Agréger les données pour changer la fréquence
  - `ts : aggregate(my_ts, nfrequency = 4, FUN = mean)`
  - `xts : apply.quarterly(my_xts, mean)`
- Gérer les données manquantes :
  - `na.locf(df, fromLast = False)` : complète les données manquantes par la dernière valeur connue (si `fromLast = T`, alors on utilise la prochaine valeur connue)
  - `na.fill(df, fill=0)` : complète les données manquantes par un 0 (ou pour toute autre valeur renseignée au paramètre `fill`)
  - `na.approx(df)` : effectue une interpolation linéaire sur toutes les valeurs manquantes
  - `na.spline(df)` : effectue une spline sur toutes les valeurs manquantes



1. Importer les fichiers `brent.csv` et `eia_data.csv` depuis le repertoire `input` et renommer les colonnes
2. Transformer ces deux structures en série temporelle avec un format `xts`
3. Fusionner les données dans une variable « base »
4. Formater « base » en trimestrielle
5. Créer une variable « `base_ts` » qui est la même que « base » mais au format `ts`
6. Effectuer un lag de 5 trimestres sur la colonne « `brent` » de « base » et l'ajouter dans la base
7. Calculer une moyenne mobile 6 trimestres sur cette même colonne et l'ajouter dans la base
8. Calculer la croissance annuelle du `brent` et l'ajouter dans la base
9. Créer un indicateur qui vaut 1 si la croissance du `brent` est positive ou nulle, 0 sinon, et l'ajouter dans la base
10. Sauvegarder la base dans un fichier « `baseFinale.csv` » situé dans un repertoire « `output` »