Reproductibilité

Tsoldour

2025-02-20

body { text-align : justify} header { text-align : center}

Introduction

Depuis que la science moderne existe, chaque résultat est d'abord considéré du point de vue de sa **reproductibilité**. Une expérience, aussi élégante soit-elle, n'est considérée par la communauté scientifique que dans la mesure où elle est *reproductible*. Karl Popper (1902-1994) en avait d'ailleurs fait l'un des critères fondamentaux de la scientificité.

Mais qu'entendons-nous par-là ?

La reproductibilité se décline a différents niveaux de la pratique scientifique. Il peut s'agir de la reproductibilité des *méthodes*, des *données* ainsi que des *résultats*. Concrètement, cela se traduisait jusqu'à il y a peu de temps encore par l'élaboration de design expérimentaux répondant à des normes établies sur des considérations essentiellement statistiques (nombre de réplicats, randomisation, etc.), des pratiques de laboratoire strictes et standardisées, ou encore la tenue rigoureuse d'un cahier de laboratoire où le chercheur/ingénieur/technicien notait le plus fidèlement possible chacune des actions réalisées dans le cadre du projet de recherche. Ces *bonnes pratiques* sont toujours valables et font l'objet d'une évaluation stricte par les pairs dans le cadre des publications scientifiques - objectif ultime de chaque chercheur - qui doivent fournir suffisamment d'informations pour que l'ensemble du projet ayant abouti à la rédaction de tel article puisse être reproduit à l'identique par quiconque s'en donne les moyens.

Oui, mais voilà !

Depuis plusieurs années maintenant, la science en général et les sciences de l'environnement en particulier souffrent d'une **crise de la reproductibilité** (Ioannidis, 2005; Harris & Sumpter, 2015). Cette dernière résulte pour une part de l'affaiblissement des bonnes pratiques admises dans les sciences concernées, mais aussi de l'émergence de l'informatique et du Big Data qui impliquent l'utilisation de jeux de données souvent complexes et multiples au sein d'un même projet, ainsi que l'utilisation d'outils d'analyse évoluant à un rythme effréné. Cela ouvre une nouvelle dimension dans le monde de la reproductibilité, à laquelle il faut faire face, humblement, si l'on veut produire une science qui, de ce point de vue là au moins, puisse prétendre à une certaine valeur.

Dans ce contexte, nous proposons ici d'exposer les bonnes pratiques de base permettant la reproductibilité d'un travail de recherche reprosant sur l'utilisation d'outils bioinformatiques. **Le language de programmation R sera notre fil conducteur.**

Ces bonnes pratiques se déclinent en 4 axes principaux :

- **L'architecture du projet :** Elle correspond à la manière d'organiser son répertoire de travail, qui doit être efficace (chemins d'accès simplifiés, noms de fichiers facilement intelligibles) et normée, en s'atisfaisant à la convention du **Research compendium**.

- **La mise en oeuvre du projet :** Aussi étonnant que cela puisse paraître, un simple clic dans une interface n'est pas reproductible ! Il est donc impératif de privilégier l'interaction en **ligne de commande** avec sa machine, quand bien même cela demande un long et difficile apprentissage. De plus, l'utilisation et la génération de *scripts* doit se faire selon certaines règles. De même que l'enchaînement des actions réalisées sur les données (**workflow**) doit être automatisé pour éviter toute erreur/variation dûe à l'opérateur. Enfin, il est important de garder à l'esprit que toute action réalisée sur un ordinateur, qu'il s'agisse d'une opération en ligne de commande ou de l'utilisation d'une interface clic-bouton donnera un résultat intimement lié à **l'environnement système**, qu'il faut donc prendre en compte.

  Pour tout cela, deux principaux packages R sont à connaître : *targets* et *renv*

- **Le suivi du projet :** De même qu'au laboratoire toute modification d'un protocole doit être discutée avec ses collègues, validée collectivement et notifiée dans le cahier de laboratoire, de même tout changement opéré au sein d'un workflow doit être discuté, validé, recensé ET réversible. Pour cela, des outils de **suivi des changements** (**versioning**) existent, qui devraient être systématiquement utilisés.

  Git (en local) et *Github* (en distancié) sont les outils de versioning les plus communément utilisés par la communauté.

- **Le partage du projet :** Le partage du projet doit se faire à l'aide d'outils limitant l'intégration manuelle d'objets divers au sein du manuscrit (rapport, thèse, article, etc.) dans le but, toujours, de réduire au maximum le risque d'erreur individuelle. Ces outils reposent sur le principe de **programmation lettrée** (**literate programming**), qui consiste en l'intégration de balises au sein même du texte, dont le rendu visuel n'est accessible qu'après exportation du document au format pdf (ou word, html, LateX, etc.).

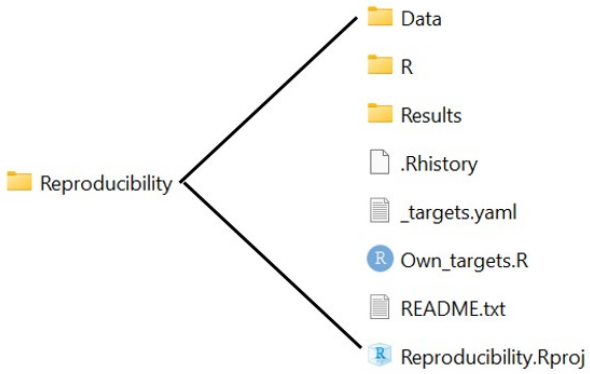
  Sur Rstudio, l'outils de référence est *Rmarkdown*.

1. L'architecture du projet

L'architecture du projet doit répondre aux normes du **Research compendium**. Ce dernier est un ensemble de règles simples à suivre pour organiser de manière **standardisée** et efficace l'ensemble numérique de votre projet de recherche. En adoptant cette convention, nimporte quelle personne familière du **Research compendium** sera en mesure d'appréhender rapidement le projet, de l'élargir, de le diffuser, de le reproduire. Le **Research compendium** constitue ainsi l'architecture de base d'un projet reproductible !

1.1 La structure fondamentale

L'ensemble des éléments numériques d'un projet de recherche doivent être contenus dans un unique répertoire de travail, ici *Reproducibility*. Ils doivent en revanche être clairement séparés au sein du répertoire de travail.



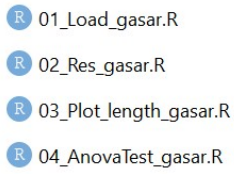
Les éléments minimums devant être présents dans le répertoire de travail de votre projet sont:

- **Data :**

Ce dossier ne doit contenir que les **données brutes** du projet, programmées de préférence en **lecture seules**. Puisque tout le projet repose sur ces données, elles ne doivent jamais être manipulées ! Par ailleurs, les données nettoyées doivent déjà être considérées comme des résultats. Elles ne doivent donc pas se trouver dans ce dossier.

- **R :**

Ce dossier contient les **scripts du projet**. Ces scripts contiennent les actions réalisées sur les données. Il est préférable de se limiter à des scripts courts correspondant à des actions bien précises (nettoyage, analyse statistique, représentation graphique). Il est important que chaque script soit facilement compréhensible par chacun. Pour cela, il est pertinent de nommer le script avec le nom de la fonction qu'il contient et de les numéroter par ordre d'utilisation. On visualise ainsi facilement l'ensemble du workflow.



Pour en faciliter encore la compréhension, il est aussi conseillé d'attribuer un en-tête standardisé à chaque script. Il n'y a pas de norme établi, il faut simplement que cet en-tête permette de comprendre rapidement à quel projet appartient le script, quel est son rôle au sein du projet et, éventuellement, de contacter son auteur.

```
############################
# NOM_DU_PROJET
# NOM_DU_SCRIPT
# OBJECTIF DU SCRIPT
# CONTACT
############################
```

- **Results**

Ce dossier contient l'ensemble des résultats : données néttoyées, graphiques, résultats statistiques, etc. Les résultats doivent *a minima* porter le nom du script les ayant produits. Si plusieurs résultats sont émis pour un même script, ils doivent alors être rangées dans des dossiers portant le nom de leur script parent.

Contrairement au dossier **Data** dont le contenu est sacré, et au dossier **R** qui doit faire l'objet d'un suivi particulier, le contenu de **_output** n'a aucune valeur dans la mesure où il peut être produit à l'identique et *ad libitum*. Il ne faut donc pas avoir peur de manipuler ses résultats, les effacer, les déplacer, etc.

- **R_project**

Tout projet de recherche qui se respècte, s'il utilise R comme language de programmation pour ses analyses, doit être réalisé dans le cadre d'un **R_project**. Situé à la base du répertoire de travail, le R_project associé à votre projet est ce qui vous permet d'utiliser des **chemins d'accès relatifs** pour l'ensemble des fichiers contenus dans votre répertoire de travail. Ainsi en diffusant votre projet avec le fichier *.Rproj* associé, il ne sera jamais nécessaire de toucher aux chemin d'accès utilisés dans les scripts.

Si vous ne travaillez pas déjà de manière systématique au sein de R_projects, il est grand temps de s'y mettre ! Un tutoriel pour créer des R_projects est accessible ici: **LIEN**. Ainsi, vous n'aurez plus jamais besoin de spécifier votre environnement de travail à l'aide de la fonction **setwd()**, qui utilise des chemins d'accès absolus valables uniquement sur votre ordinateur (et encore !).

1.2 Les éléments supplémentaires

Quelques éléments peuvent être ajoutés au répertoire de travail qui, s'ils ne sont pas obligatoires sont quand même les bienvenus.

- **README.txt**

Le fichier **README** est un fichier qui a pour vocation d'expliquer le projet. C'est généralement le premier fichier qu'on ouvre lorsqu'on découvre un projet pour la première fois. On peut y faire mention de toutes les choses pertinentes à savoir pour comprendre l'origine, le contexte, l'objectif et les modalités de réalisation d'un projet.
Le fichier **README** est aussi le bon endroit pour lister l'ensemble des informations système propres au projet : le système d'exploitation, la version de R, les packages et les versions de package utilisés, etc.

- **Docs**

Un dossier **Docs** peut éventuellement venir compléter le répertoire de travail en contenant par exemple la documentation associée aux packages spécifiques utilisés.

2. La mise en oeuvre du projet - le package *targets*

targets est un package servant à la gestion de **workflow**. Il sera votre allié idéal pour organiser votre projet, de le faire évoluer, mais aussi d'avoir un oeil sur **l'architecture du projet** - très pratique lorsque le projet contient de multiples jeux de données, de nombreux scripts et énormément de résultats et que ces derniers sont en plus inter-dépendants ! - et de visualiser le degré d'actualisation de vos différents objets.
Pour comprendre comment fonctionne *targets*, rien de tel qu'un exemple concret !
A noter qu'il est recommander d'écrire ses scripts sous forme de fonction. Cela permet (i) de limiter le nombre d'objets créés lorsque les scripts tournent et (ii) de simplifier au maximum l'utilisation du script de gestion *targets*

Pour cet exemple, nous utiliserons le jeu de données **Allo.csv** (disponible ici : https://github.com/TSolDour/Reproducibility/tree/master/Data) que nous allons charger à l'aide d'une fonction dédiée:

```r
load_gasar <- function(file){
  read_table(file=file, locale = locale(decimal_mark = ",")) %>%
    data.frame() %>%
    filter(Espece == "Crassotrea_gasar")
}
```

Pour ce projet, nous souhaitons réaliser une comparaison succinte de la longueur des coquilles de *Crassostrea gasar* en fonction de la station d'échantillonnage. Pour cela, nous voulons d'abord :

- (i) établir les statistiques descriptives pour chaque espèce ;
- (ii) représenter graphiquement les données ;
- (iii) réaliser un test de comparaison multiple (ANOVA)

Ces trois étapes correspondent aux trois fonctions ci-dessous:

```r
#-----------------------------------------------------------------
Res_gasar <- function(data) {
  data %>%
    group_by(Station, Lot) %>%
    summarise(length=mean(Longueurs) )
}
#-----------------------------------------------------------------

#-----------------------------------------------------------------
Plot_length <- function(Resume){
  ggplot(data = Resume, aes(x=Station, y=length, color=Lot)) +
    geom_point()
}
#-----------------------------------------------------------------

#-----------------------------------------------------------------
AnovaTest_gasar <- function(data, a, b){
  library(tibble)
  library(dplyr)
  library(car)
  Obj <- aov(a ~ b, data=data)
  Shap <-  shapiro.test(Obj$residuals)
  if(Shap["p.value"] > 0.05){
    Lev <- leveneTest(Obj$residuals, data$Station)
  } else(return("No residual normality"))
  if(Lev["group","Pr(>F)"] > 0.05){
    Res <- anova(Obj)
  } else(return(paste0("No homoscedasticity, p-value = ", (Lev["group","Pr(>F)"]))))
  if(Res["b", "Pr(>F)"] < 0.05){
    Post <- TukeyHSD(Obj)
    return(list(data.frame(Post$b) %>%
                  rownames_to_column(var="b") %>%
                  filter(p.adj<0.05),Res["b", "Pr(>F)"]))
  } else (return("null"))
}
#-----------------------------------------------------------------
```

Le projet étant très simple (seulement 4 fonctions), il est possible de tout enregistrer dans un seul script que vous appellerez *FONCTIONS.R* afin de simplifier les choses. Mais dans un projet plus conséquent il est recommandé de **correctement séparer les fonctions en différents scripts**.

Maintenant que votre script est prêt et correctement enregistré dans le dossier **R** de votre projet, vous allez pouvoir utiliser le package *targets*.
Placez-vous à la racine de votre projet et créez le document *_targets.yalm* contenant ceci:

```
Own:
  script: Own_targets.R
  store: Results/Own
```

Ensuite, si ce n'est pas déjà fait, vous devez installer et charger le package *targets*. Vous pouvez maintenant commencer à utiliser *targets* en utilisant, toujours dans votre console, la fonction :

```
use_targets(script = "Own_targets.R")
```

Cette fonction crée le script suivant que vous avez nommé du nom de votre projet, ici *Own_targets.R*. Ce script est **le tableau de bord de votre workflow**. C'est toujours par lui que vous lancerez votre workflow après avoir édité vos scripts existants ou en avoir créé de nouveaux.

Voyons maintenant comment il fonctionne.

- Avant tout, je vous conseille de **nettoyer l'espace de travail**:

```
rm(list=ls())
```

- Il faut ensuite **charger l'ensemble des packages nécessaires à la réalisation du workflow**. Dans ce cas, ils sont assez peu nombreux:

```
tar_option_set(
  packages = c("readr", "dplyr", "ggplot2", "tibble", "car"))
```

- Il faut maintenant **charger tous les scripts du workflow** (dans le bon ordre !). Ici, nous avons regroupé toutes nos fonctions en un seul script ce qui simplifie l'affaire:

```
tar_source("R/FUNCTIONS.R")
```

- C'est ici la partie délicate : vous devez lister de manière exhaustive et sans erreur l'ensemble des fonctions du package et leur *output* associé. La fonction pour le faire se résume ainsi : *tar_target(OUTPUT, FONCTION)*

```
list(
  tar_target(file, "Data/Allo.csv", format="file"),
  tar_target(data, load_gasar(file)),
  tar_target(Resume, Res_gasar(data)),
  tar_target(Plot, Plot_length(Resume)),
  tar_target(Anova_gasar_Length_Station, AnovaTest_gasar(data, data$Longueurs, data$Station)))
```

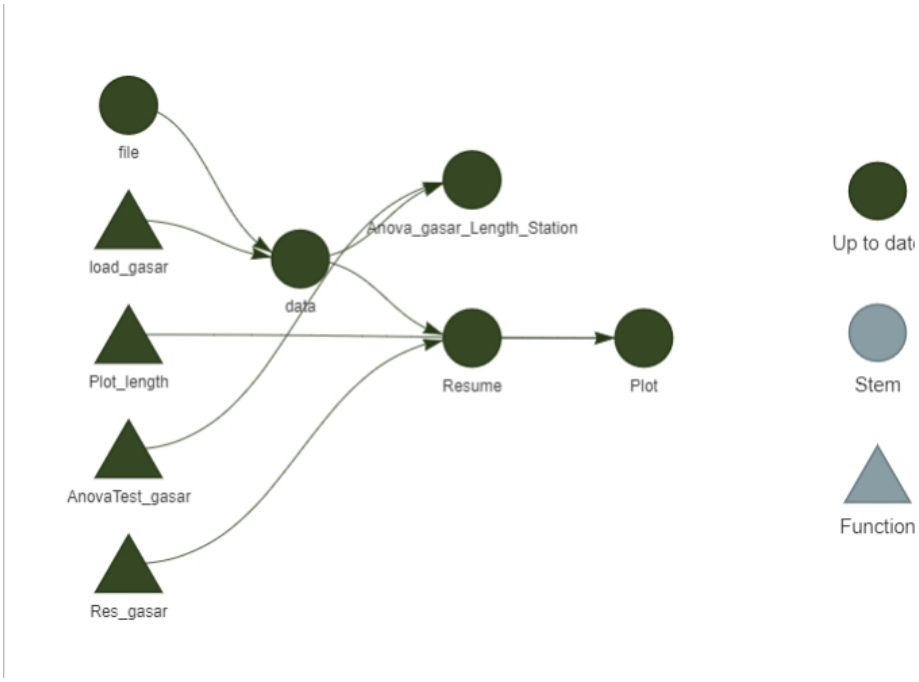
Voilà, votre projet est maintenant prêt à être géré par *targets* ! Votre script **projet_targets.R** ne doit rien contenir d'autre que les éléments déjà présents. Vous pourrez évidemment le mettre à jour en ajoutant de nouveaux packages, de nouveaux scripts et de nouvelles *targets* à mesure que vous avancez dans votre projet.

Plusieurs fonctions de base que vous devez utiliser dans votre console vous permettent d'interagir avec votre workflow:

```
Sys.setenv(TAR_PROJECT = "Own") # Pour spécifier à target que vous vous trouvez dans la partie
# "Own" de votre projet
tar_manifest(fields = command) # Pour vous assurer qu'il n'y a pas d'erreur dans
# le listing des targets
tar_visnetwork() # Pour visualiser l'architecture du projet
tar_make() # Pour faire tourner le workflow
tar_read() # Pour afficher les résultats du workflow
```

La fonction `tar_visnetwork()` est particulièrement pratique puisqu'elle vous permet d'afficher l'ensemble de votre projet et d'en percevoir ainsi l'architecture, les dépendences entre les objets et leur niveau d'actualisation.

Dans le cas de cet exemple, la figure produite est la suivante:



Par ailleurs, la fonction `Sys.setenv()` est à utiliser surtout lorsque votre répertoire de travail contient plusieurs sous projets. L'argument qu'on lui indique renvoie au contenu du fichier *PROJET.yalm* dans lequel sont listés tous les sous projets du répertoire et qui définit les dossiers dans lesquels ranger les *outputs* associés.

3. La reproductibilité de l'environnement de travail - le package *renv*

L'intérêt de *renv* pour la gestion et la reproductibilité de vos projets de recherche est triple :

- (i) *renv* permet d'**isoler** chaque projet en lui fournissant sa propre **librairie de packages**. Ainsi, en installant de nouveaux packages ou en mettant à jour des packages existants dans le cadre d'un projet, vous n'impacterez pas les autres projets utilisant éventuellement les versions antérieures.

- (ii) *renv* facilite le **transport** des projets entre ordinateurs en automatisant l'installation des packages et de leurs versions dont dépend le projet.
- (iii) *renv* améliore la **reproductibilité** du projet en assurant l'utilisation constante d'une même librairie de packages quel que soit l'utilisation, l'ordinateur et l'époque d'utilisation.

Point sémantique : Le terme "librairie" désigne ici un dossier contenant l'ensemble des *packages* utilisés. Habituellement, avec la fonction `library()`, on charge les packages depuis une **librairie système** partagée par tous les projets R. Avec *renv*, ce sont des librairies spécifiques qui sont créées pour chaque projet.

Une fois le package installé grâce à la fonction `install.packages("renv")`, vous pouvez convertir votre projet par la fonction suivante `renv::init()`. Ce-faisant, trois nouvelles entitées sont créées à la racine de votre projet:

- La **librairie du projet** : *renv/library*. Ici seront stockés tous les packages du projet.
- Le **fichier verrou** : *renv.lock*. Ce fichier contient toutes les métadonnées nécessaires pour pouvoir réinstaller tous les packages sur n'importe quel nouvel ordinateur.
- Le **R profile** : *.Rprofile*. Ce fichier est utilisé par *renv* pour configurer chaque nouvelle session de R de sorte qu'il soit possible d'utiliser la librairie du projet.

Au cours de l'élaboration de votre projet, vous serez très probablement amenés à charger, installer, mettre à jour différents packages. Pour que renv mette correctement à jour le fichier *renv.lock*, il suffit de faire tourner la fonction `renv::snapshot()`.

Enfin, la fonction `renv::restore()` permet, lorsque vous partagez votre projet avec une tierce personne de réinstaller automatiquement l'ensemble de la librairie projet sur son ordinateur à partir des métadonnées de *renv.lock*.

Ici sont les trois fonctions `init()`, `snapshot()`, `restore()` qui permettent une utilisation basique du package *renv* pour améliorer la reproductibilité de votre projet. Pour une utilisation avancée, vous trouverez toutes les informations nécessaires ici : https://rstudio.github.io/renv/articles/renv.html.

3. Le suivi du projet - Git et Github

3.1 De quoi parle-t-on ?

Git est un système de *contrôle de version*, pensé initialement pour faciliter le *travail collaboratif* pour les développeurs travaillant sur des projets de grande taille et complexes. Ainsi, git permet de gérer l'évolution d'un ensemble de documents au sein d'un projet - **repository** dans le jargon.

Attention: Git peut être difficile à appréhender. Ce système est tout sauf intuitif... Si vous ne travaillez pas en *mode collaboratif* sur votre projet, il y a d'autres manières plus accessibles d'assurer un suivi de version en local. Autrement dit, vous immerger dans l'univers du **Git** n'a de sens que si vous vous destinez à travailler sur un projet avec d'autres collègues qui seront amenés à intervenir sur les scripts, les manuscripts et les autres objets dudit projet (ce qui est, en fait, le cas de la presque totalité des chercheurs !). Et dans ce cas, ce n'est pas seulement de **Git** dont vous aurez besoin mais de sa version *délocalisée*. Il en existe un certain nombre mais la plus généralement utilisée est **Github**.

Dans cette partie, vous allez apprendre à **utiliser Rstudio et contrôle de version à l'aide de Git et Github**. Il faut pour cela avoir installé R et Rstudio sur votre ordinateur, ainsi qu'avoir créé un compte gratuit sur **Github**.

3.2 Installation de Git

Il y a différentes procédures d'installation de Git mais toutes ne se valent pas.

- Sur une machine **windows**, il est recommandé de suivre cette **procédure**, qui installe automatiquement le **Git bash** ainsi qu'un ensemble d'autres outils nécessaires. Par ailleurs, cette procédure place le fichier executable dans un dossier conventionnel qui en facilite l'utilisation par Rstudio.
- Sur une marchine **MacOS**, la meilleure façon de faire et d'aller directement dans le shell et de taper ces deux commandes :

```
git --version
git config
```

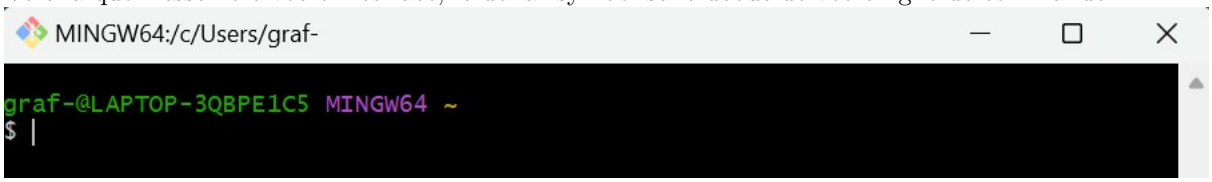
Un *pop-up* devrait alors proposer une installation, qu'il faut accepter !

3.3 Configuration de Git

Avant de passer par R, il est important de se familiariser avec les commandes de base de Git. Pour cela, il faut ouvrir l'application **Git bash**, accessible depuis le menu "Rechercher" de votre ordinateur.

Une fois ouverte, cette application n'est rien d'autre qu'un **shell**, une interface utilisateur permettant de recevoir des commandes émises en langage informatique et de les transmettre à un **programme**, ici Git. Et comme son nom l'indique, ce shell comprend préférentiellement le langage **bash**. Une liste assez complète des commandes bash élémentaires est disponible **ici**.

Voici à quoi ressemble votre interface, le dollar symbolise le début de votre ligne de commande :



A partir de là, il est possible de naviguer dans votre machine et de réaliser un ensemble d'actions basiques :

```
cd PATH           # Naviguer jusqu'à un répertoire choisi;
ls -l             # Afficher le contenu du répertoire;
touch NAME        # Créer un nouveau fichier;
rm NAME           # Effacer un fichier;
mkdir NAME        # Créer un nouveau dossier;
rm -r NAME        # Effacer un dossier
```

Une fois que vous maîtrisez ces quelques fonctions, il est temps de **programmer** votre Git !

Avec Git, la synthaxe est toujours la même `Git COMMANDE --ARGUMENT`, pour la configuration nous utiliserons `git config`:

```
git config --global user.name "votre nom"
git config --global user.email "votre email"
git config --global core.editor "emacs" #Ou tout autre éditeur de votre choix
git config --global init.defaultBranch "master" #Nom de la branche principale de vos projets
```

De nombreuses autres options de configuration sont disponibles que vous devrez découvrir par vous-même. Ici sont les principales options pour bien commencer. Pour voir l'ensemble de vos options :

```
git config --global --list
```

3.4 Connecter Git et Github

Maintenant que Git vous connaît, il faut lui présenter votre Github ! Là encore plusieurs façons de faire existe, c'est la plus simple qui vous est présentée ici.

Il faut d'abord générer un **PAT** (*Personal Access Token*). Le PAT est une sorte de mot de passe qui atteste de notre appartenance au Git et définit ce que nous pouvons ou pas faire.
Deux manières de générer un **PAT**:

- Directement *via* Github, une fois connectés, cliquez sur *settings>Developer settings>Personal access tokens>Generate new token*
- Sur Rstudio, installez le package *usethis* et utilisez la fonction `usethis::create_github_token()`. Cela ouvre un *pop-up* vous permettant de **(i) décrire** le PAT (à quoi est-il destiné), **(ii) définir une date d'expiration** (recommandé par Github) et **(iii) définir le scope** du PAT. Une fois ces paramètres renseignés, cliquez sur *Generate token.* | | **AVANT DE FERMER LA FENÊTRE** assurez-vous de **copier** correctement le **PAT** car une fois la fenêtre fermée vous ne pourrez plus y avoir accès ! Un bon moyen de **copier le PAT** dans un endroit sûr facilement accessible pour Rstudio, git et Github est d'utiliser la fonction `usethis::gitcreds_set()`. R vous demandera alors de renseigner votre PAT pour le stocker en lieu sûr:

```
gitcreds::gitcreds_set()

? Enter password or token: ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-> Adding new credentials...
-> Removing credentials from cache...
-> Done.
```

C'est bon, vous avez vos papiers, vous pouvez vous connecter à Github !

- Dans Github, cliquez sur le gros bouton vert *new* à côté de **repositories**. Remplissez les champs nécessaires et laissez les paramètres par défaut. De préférence, donnez à votre repository le même nom que le projet local que vous voulez mettre en partage sur Github, par exemple le projet *Reproducibility*, puis cliquez sur *Create repository.*
- Cliquez ensuite sur le bouton <> *Code* et copiez l'**URL HTTPS**
- Dans le Git bash, placez-vous dans le dossier *Reproducibility* qui contient l'ensemble du R_project avec son gestionnaire *targets* et sa librairie propre *renv, ainsi que les scripts, données et résultats du projet.
- Faites tourner la fonction suivante avec l'URL HTTPS que vous avez copié :

```
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY.git
```

- Votre Github repository est maintenant connecté à votre projet local mais il est vide. Pour transférer l'ensemble de votre projet sur le *repository* github :

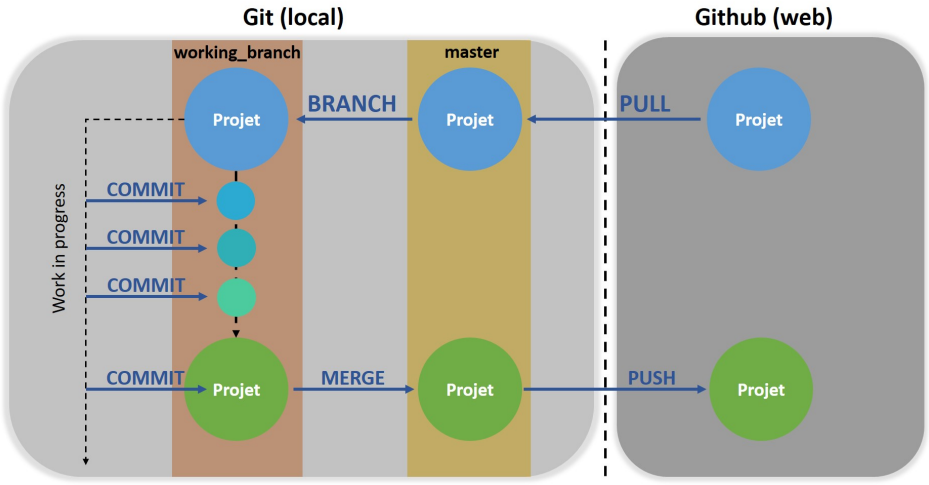
```
git push --set-upstream origin master
```

En rafraichissant la page de Github, vous devriez voir apparaître l'ensemble de votre projet. Il faut dorénavant considérer que la **localisation première** (la seule qui doit être systématiquement à jour) de votre projet n'est plus votre ordinateur mais votre **repository Github**. L'ordinateur n'est plus qu'un moyen d'agir sur les composants de votre projet.

Remarque : Avec `Git clone URL HTTPS` vous pouvez faire l'inverse, à savoir importer un *repository* depuis Github vers votre propre ordinateur.

3.5 Exemple d'utilisation en routine

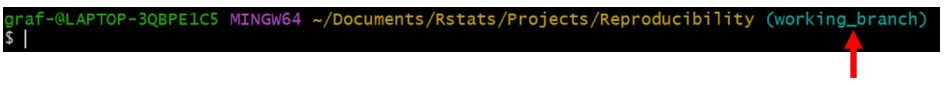
Vous vous devez maintenant de connaître l'architecture de base de Git et Github ainsi que les fonctions associées à certaines actions pour commencer à utiliser cet outil correctement. Une routine d'utilisation basique de Git et Github est synthétisée dans le schéma suivant :



En considérant toujours que **la dernière version de votre projet se trouve sur Github**, il faut, au début de votre journée de travail, vous rendre en local à la racine du répertoire de votre projet *via* le **git bash** : `cd PATH`. Une fois localisé au bon endroit, utilisez la commande `git pull` de manière à rapatrier la dernière version du projet sur votre machine. Ce faisant, vous avez maintenant votre projet *up-to-date* dans la branche *master* de votre git.

Pour travailler serainement sur le projet, je vous conseille de créer immédiatement une branche parallèle en local, sur laquelle vous ferez toutes vos modifications avant de les réintégrer à la branche principale. Il est consillé de toujours donner des **noms explicites** aux différentes branches que vous créerez ! Ici, par exemple : `git branch working_branch`. Une fois votre branche créée, il faut vous placer sur celle-ci à l'aide de la fonction `git checkout working_branch`.

La branche sur laquelle vous vous trouvez est indiquée à la fin du chemin d'accès de votre localisation.



A ce stade, vous vous trouvez sur une branche parallèle (*working_branch*) qui contient la même version du projet que la branche principale (*master*). Vous pouvez commencer à travailler !

Par exemple, vous pouvez créer - si ce n'est pas déjà fait - le fichier README.txt à la racine de votre projet, ou le modifier s'il existe déjà. Cette modification n'existe que dans votre *working_branch* mais elle n'est pas encore prise en compte par Git. Il faut pour cela faire deux choses :

- Signifier à Git de prendre en compte la création ou la modification du fichier : `git add NomDuFicier` ou bien `git add --all` dans le cas où plusieurs actions sont à prendre en compte en même temps (création d'un fichier, modification d'un autre, etc.) ;
- Dire à Git de capturer ces modifications. On parle alors de **commit** qui est une sorte de capture d'écran, d'enregistrement de l'état actuel du projet. Un commit requiert un commentaire de sorte qu'il vous sera ultérieurement possible de savoir explicitement à quel ajout/modification du projet correspond chaque commit. La fonction est la suivante : `git commit -m "Votre message"`.

Remerque : Si vous êtes un peu perdus, vous pouvez demander à Git le status des différents objets du projet : sont-ils tous pris en compte ? combien d'actions doivent être *commitées* ? Il suffit de taper : `git status`.

Ainsi, après chaque commit, c'est une nouvelle version du projet qui est enregistrée sur votre branche. Ce n'est pas la peine de *commiter* à chaque modification apportée. Il est conseillé de le faire à des moments stratégiques, lorsqu'une modification significative a été apportée au projet : la rédaction d'un nouveau script, l'ajout de nouvelles données, la production d'un nouveau résultat... En réalisant plusieurs commits au cours de votre journée de travail, le projet se modifie en conservant la trace de chaque étape clé : les versions du projet, qui fonctionnent comme les cailloux laissés derrière lui par le Petit Poucet. C'est là tout l'intérêt du *versionning*, en cas de problème, d'erreur ou d'impasse, il vous est possible de revenir à la version précédente en un claquement de doigt : `git reset HEAD^` !

A la fin de votre session de travail, le moment est venu de renvoyer la dernière version de votre projet vers Github pour qu'elle soit accessible par chacun et partout. Au moins deux possibilités s'offrent à vous : (i) Vous êtes satisfaits du travail accompli, vous être sûr de vos modifications et personne d'autre que vous ne doit les valider ou (ii) Vous devez rendre vos modifications accessibles à un collègue avant de les intégrer pleinement au projet.

Dans le premier cas, il vous faut réintégrer la dernière version de votre projet sur votre branche principale en local. Pour cela, rendez-vous sur votre branche *master* : `git checkout master`, puis fusionnez le contenu de votre branche parallèle à la branche principale : `git merge working_branch`. Ca y est, la branche *master* contient la dernière version de votre projet, que vous pouvez renvoyer sur la *master* de Github par la fonction `git push`.

Dans le second cas, vous pouvez envoyer directement votre *working_branch* vers Github avec la même fonction `git push`. En actualisant la page de github, vous verrez alors apparaître cette nouvelle branche qui pourra être consultée, modifiée ou fusionnée à la branche principale à tout moment.

3.5 Connecter Rstudio à Git et Github