Curso de extensão em jurimetria

Alexandre Chibebe Nicolella
José de Jesus Filho
Ricardo Feliz Okamoto

Objetivos da aula

1. Desenvolver autonomia nos seus projetos próprios de programação em R
    a. Aprender a organizar projetos no R em formato de pacote
    b. Aprender a fazer funções e instanciá-las
    c. Aprender a fazer relatórios automatizados no R

Pacotes em R

Percurso

1. projetos x pacotes
2. inicializando um pacote
3. a estrutura de pastas dos pacotes
    a. data-raw
    b. data
    c. R
    d. inst

projetos x pacotes

console > scripts > projetos > pacotes

- Podemos programar em R usando apenas o Console
- Mas preferimos usar scripts, pois eles salvam nossos códigos, permitindo que a gente o reutilize
- Mas podemos querer salvar um conjunto de scripts, então usamos projetos
- Mas podemos querer reaproveitar projetos inteiros e usar partes deles em outros projetos, então transformamos os projetos em pacotes
    - Eu pessoalmente só faço as coisas em R em formato de pacote

reprodutibilidade

- no fim, criar pacotes é um grande esforço de criar análises *reprodutíveis*

paths (caminhos)

- um grande problema para a reprodutibilidade são os *caminhos*
- caminhos absolutos x caminhos relativos

inicializando um pacote

inicializando um pacote

```
usethis::create_package(“pasta/onde/salvar/o/pacote/nome_pacote”)

usethis::use_git()

usethis::use_github()
```

sistema git

O git é um sistema de versionamento de códigos. Vamos usar o git através do github.

Como instalar?

1. Baixar o git no seu computador
2. Criar uma conta no github
3. Definir usuário e senha

   ```
   git config --global user.name "usuario"
   git config --global user.email "email@email"
   ```

4. Criar chaves de acesso:
   a. SSH
      i. Tools > Global options > git/SVN > SSH Key > Create SSH Key
      ii. View public key → copia e cola
      iii. Registar SSH no github: github > settings > SSH and GPG keys > New SSH key
   b. https
      i. usethis::create_github_token()
      ii. usethis::edir_r_environ()
      iii. GITHUB_PAT=[token]

inicializando um pacote

```
usethis::create_package(“pasta/onde/salvar/o/pacote/nome_pacote”)

usethis::use_git()

usethis::use_github()
```

Comandos importantes

Clonar

$ cd [path onde deseja colocar o repositório]

$ git clone git@github.com:[organizacao]/[repositorio].git

Pull, add, commit, push

$ git pull

$ git add .

$ git commit -m “subindo arquivos”

$ git push

Exercício

1. Formem duplas
2. Cada pessoa da dupla deverá criar um pacote no seu próprio github, chamado "{nome}101". Não use letras maiúsculas no início.

   Por exemplo, na dupla de Alberto com Renata, o Alberto vai criar um pacote chamado "alberto101" e a Renata "renata101". Siga os três passos:

   usethis::create_package("{pasta}/{nome}101")
   usethis::use_git()
   usethis::use_github()
3. Com o pacote criado, cada pessoa deverá criar um arquivo do tipo R e subir no data-raw, e dar push (1- git add . 2- git commit -m "sobe arq" 3- git push)
4. A outra pessoa da dupla deverá então clonar o repositório da outra. Por exemplo, o Alberto deverá dar "git clone git@github.com:renatinha12768/renata101.git" e a Renata deverá dar "git clone git@github.com:betaoDoR/alberto101.git"
   a. Tome cuidado para não clonar o repositório da outra pessoa dentro do seu repositório.

estrutura de pacote

Estrutura geral

- data-raw : códigos que geram bases de dados que vão parar no *data*
- data : dados que podem ser usados de pronto
- R : funções exportáveis ou não
- inst : “installable”, coisas que vão ser instaladas no computador de outras pessoas quando elas baixarem o seu pacote

(exemplo: https://github.com/tidyverse/dplyr)

Automatizações

Percurso

1. Funções
2. Relatórios

Funções

O que são funções?

São algoritmos que a gente decidiu empacotar de forma mais simples. Eles são úteis para repetir um mesmo “passo a passo” várias vezes.

```
gosto_de <- function(argumento1 = NULL) {
    glue::glue(“Gosto de {argumento1}”)
}
```

Pergunta - PARAMOS AQUI

Qual vai ser o resultado de “gosto_de(x)”?

A) “Gosto de pipoca”
B) “Gosto de x”
C) Erro: a função exigia “argumento1”, mas foi dada como parâmetro “x”
D) “Gosto de {argumento1}”
E) “Gosto de NULL”

```
gosto_de <- function(argumento1 = NULL) {
        glue::glue(“Gosto de {argumento1}”)
}


x <- “pipoca”
gosto_de(x)
```

Retomando a aula passada (caminho absoluto x relativo)

1. Se vocês baixarem o meu pacote *ricardo101*, qual será o caminho relativo que vocês deverão seguir para abrir o meu arquivo *funcoes_basicas.R*?

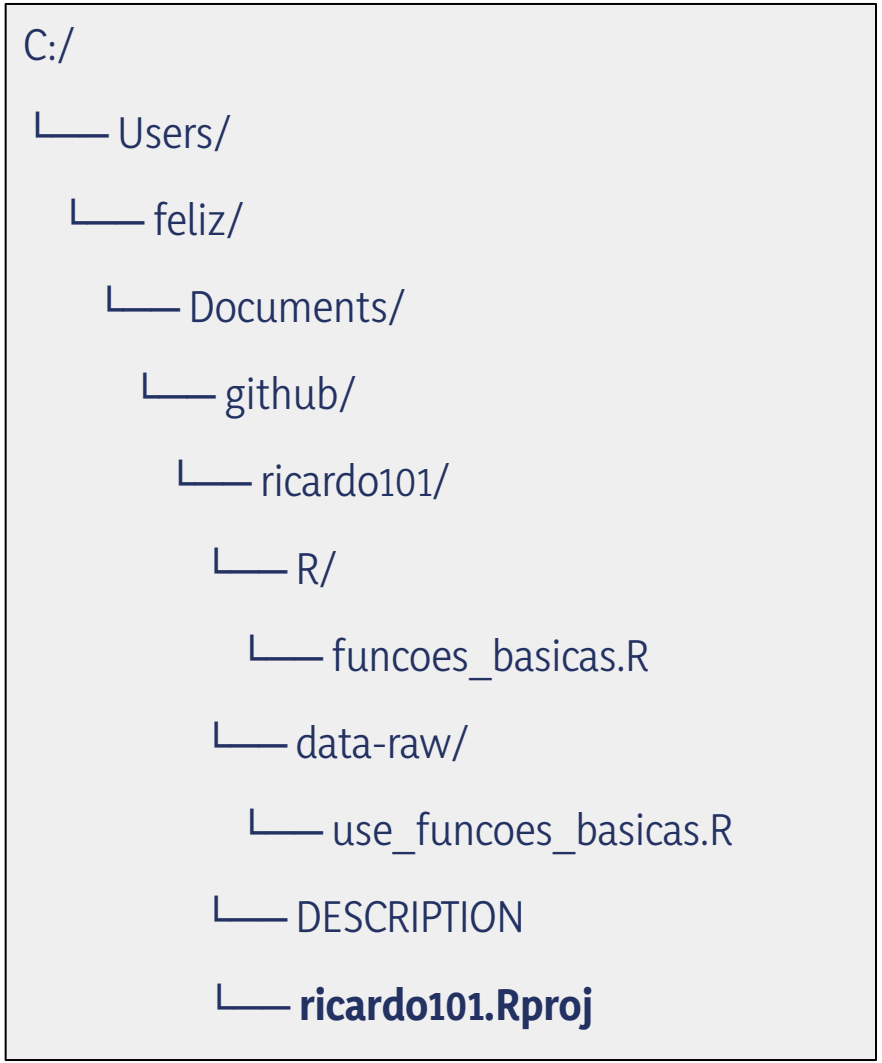
R/funcoes_basicas.R

2. E para o arquivo *use_funcoes_basicas.R*?

data-raw/use_funcoes_basicas.R

3. E para o arquivo *DESCRIPTION* ?

read.file(DESCRIPTION)



Retomando a aula passada (estrutura de pacote)

1. Eu quero criar uma função para fazer alguns cálculos complexos. A função se chama *calcula()*. Onde devo colocar essa função? **R**
2. Eu tenho uma base de dados externa em formato .csv, chamada *base_ruim.csv*. Eu vou processar essa base e transformá-la em uma versão mais utilizável, chamada *base_limpa.rda*.
   a. Onde eu salvo a base original? **data-raw**
   b. Onde e como eu salvo a nova base? **data // usethis::use_data(base_limpa)**
   c. Onde eu salvo o script que contém as funções que arrumam a base original? **data-raw**
3. Eu quero criar uma cópia da *base_limpa.rda* em formato Excel (.xlsx) e exportar para o usuário, criando então a base_limpa.xlsx. **inst/extdata**

Retomando a aula passada (função)

O que irá acontecer com a chamada *gosto_de(“pipoca”)*?

A) Ele vai imprimir “Gosto de pipoca”
B) Ele vai imprimir “Gosto de {coisa}”
C) Ele vai dar erro

```
gosto_de <- function(argumento1 = NULL) {
        glue::glue(“Gosto de {coisa}”)
}

gosto_de(“pipoca”)
```

Retomando a aula passada (função)

O que irá acontecer com a chamada *gosto_de(“pipoca”)*?

A) Ele vai imprimir “Gosto de pipoca”
B) Ele vai imprimir “Gosto de {coisa}”
C) Ele vai imprimir “Gosto de café”
D) Ele vai dar erro

```
gosto_de <- function(argumento1 = “café”) {
        glue::glue(“Gosto de {coisa}”)
}

gosto_de(“pipoca”)
```

Retomando a aula passada (função)

O que irá acontecer com a chamada *gosto_de(“pipoca”)*?

A) Ele vai imprimir “Gosto de pipoca”
B) Ele vai imprimir “Gosto de {coisa}”
C) Ele vai imprimir “Gosto de café”
D) Ele vai imprimir “Gosto de abelha”
E) Ele vai dar erro

```
gosto_de <- function(argumento1 = “café”) {
        glue::glue(“Gosto de {coisa}”)
}
coisa <- “abelha”
gosto_de(argumento1 = coisa)
```

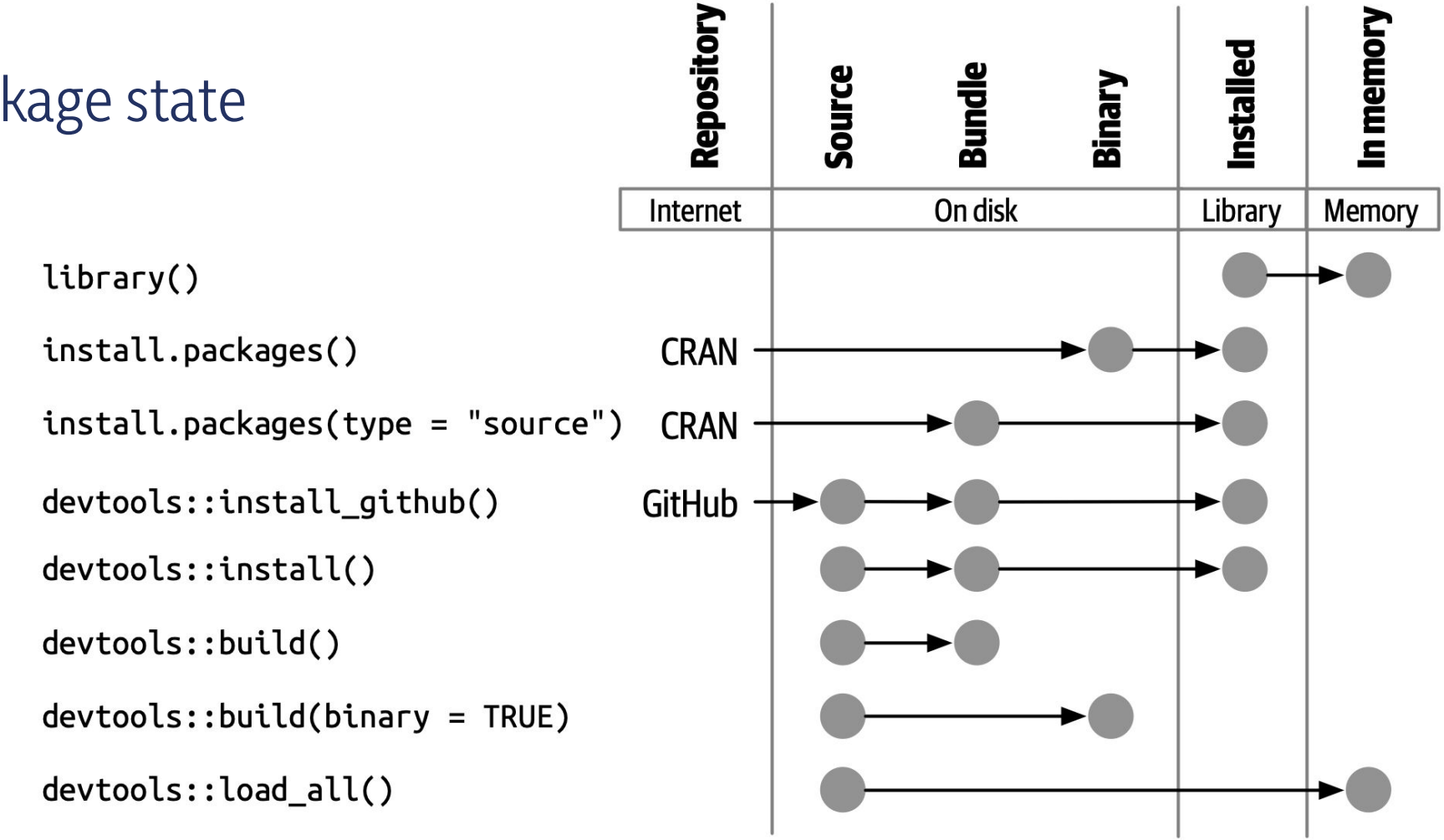
Retomando a aula passada (função)

O que irá acontecer com a chamada *gosto_de(“pipoca”)*?

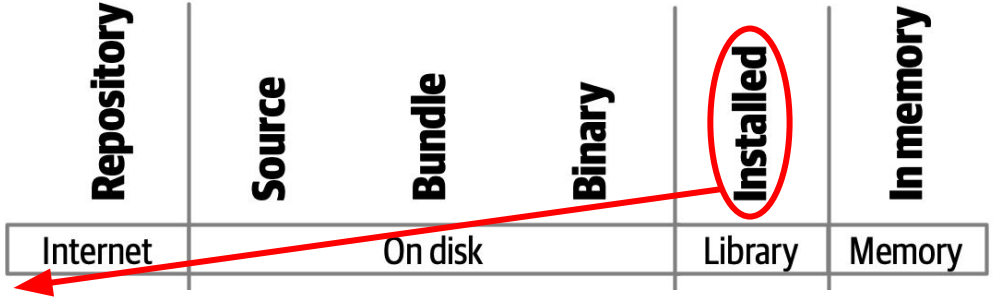
A) Ele vai imprimir “Gosto de pipoca”
B) Ele vai imprimir “Gosto de {coisa}”
C) Ele vai imprimir “Gosto de café”
D) Ele vai imprimir “Gosto de abelha”
E) Ele vai dar erro

```
gosto_de <- function(argumento1 = “café”) {
        glue::glue(“Gosto de {coisa}”)
}
coisa <- “abelha”
gosto_de(“pipoca”)
```

Package state



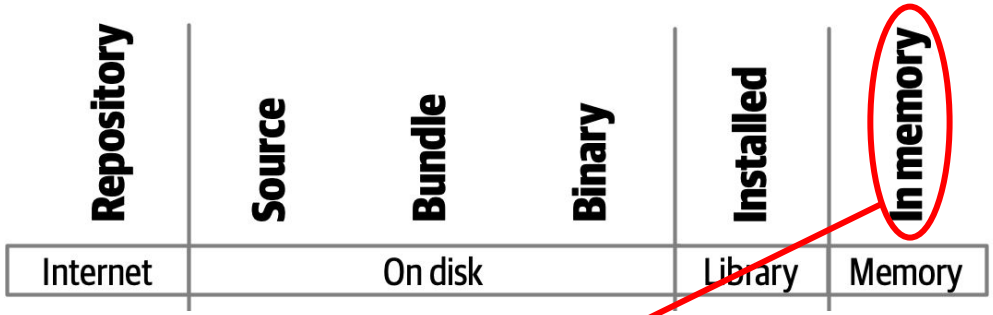
Package state



Installed: O que significa que o pacote está instalado?

- Vocês acham que o pacote que vocês criaram (“{nome}_101”) está instalado no computador de vocês?
- *.libPaths()*
- Consequência? Podemos “chamar” os pacotes quando eles estão instalados.

Package state



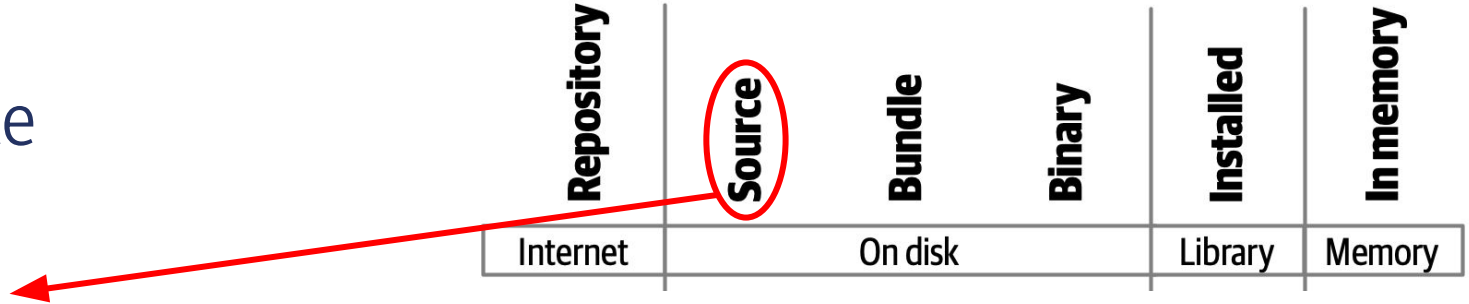
Installed: O que significa que o pacote está instalado?

- Vocês acham que o pacote que vocês criaram ("{nome}_101") está instalado no computador de vocês?
- *.libPaths()*
- Consequência? Podemos "chamar" os pacotes quando eles estão instalados.

In memory: É o estado do pacote depois de chamarmos o pacote! Chamamos usando *library*(). Com isso, todos os "exportáveis" do pacote se tornam disponíveis para nós, isto é, todas as bases de dados (aquilo que está na pasta *data*) e todas as funções (aquilo que está na pasta *R*)

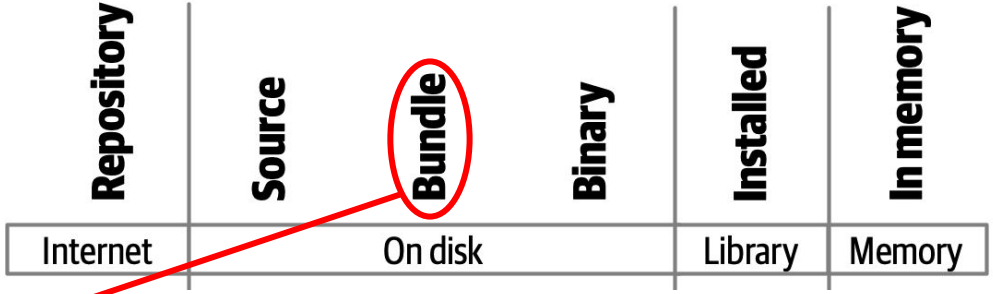
- Mas podemos usar funções e bases de dados sem ocupar espaço na memória, usando a estrutura *pacote::funcao()* ou *pacote::dados*.

Package state



Source: É só uma pasta com uma estrutura específica

Package state

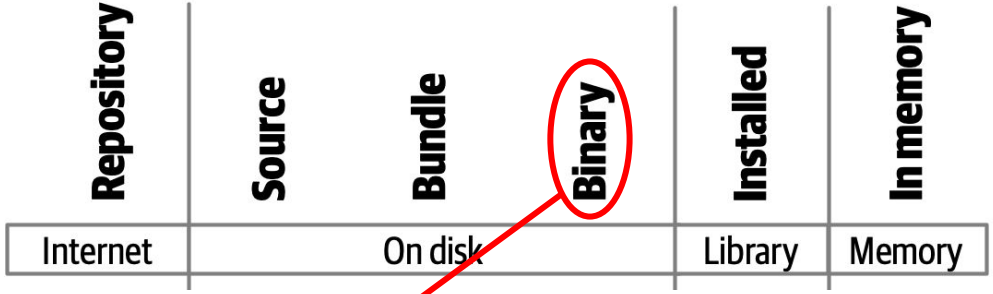


Source: É só uma pasta com uma estrutura específica

Bundle: A pasta inteira foi comprimida para um único arquivo do tipo tarballs (.tar ou .gz). Arquivos do tipo tarballs são agnósticos (não dependem de nenhum sistema operacional ou linguagem)

- devtools::build() → pkgbuild::build() → R CMD build
- .Rbuildignore

Package state



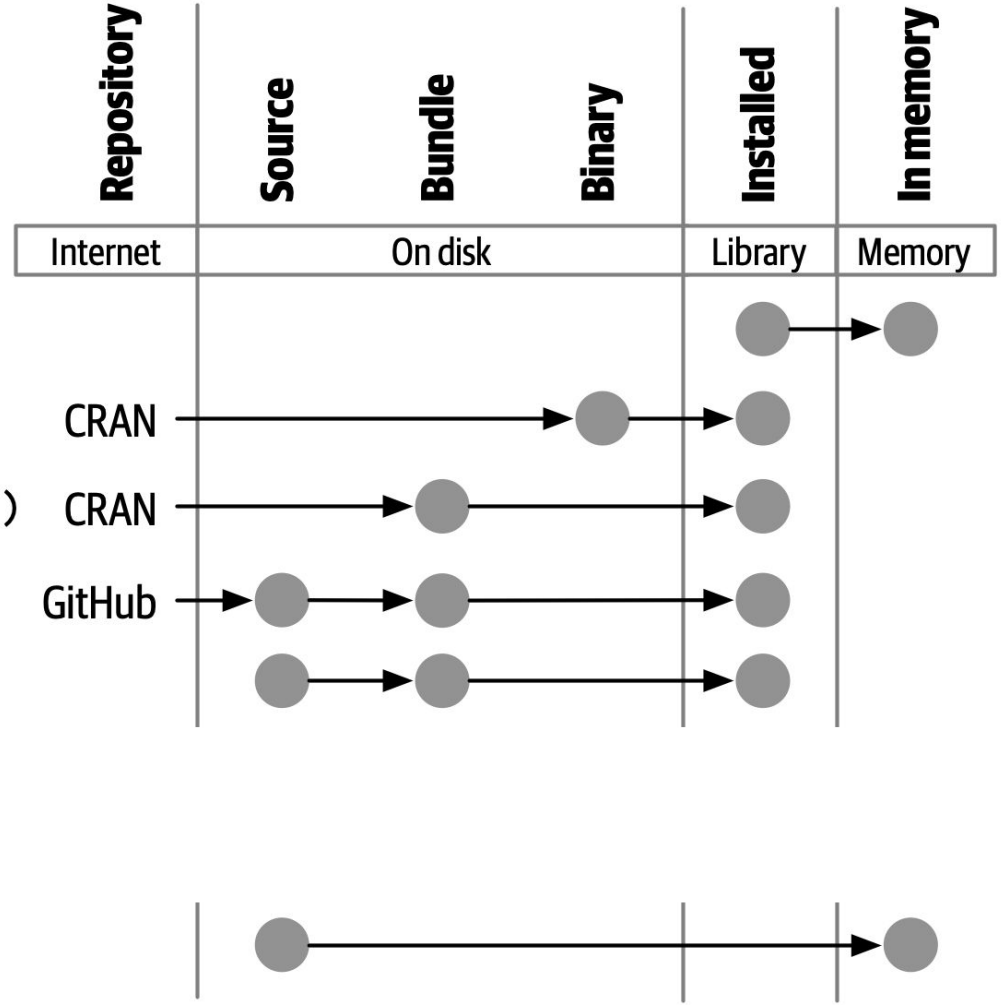
Source: É só uma pasta com uma estrutura específica

Bundle: A pasta inteira foi comprimida para um único arquivo do tipo tarballs (.tar ou .gz). Arquivos do tipo tarballs são agnósticos (não dependem de nenhum sistema operacional ou linguagem)

- devtools::build() → pkgbuild::build() → R CMD build
- .Rbuildignore

Binary: É a mesma coisa que o bundle (uma pasta comprimida em um arquivo único), mas funciona só para Windows (.zip) e MacOS (.tgz). Quando baixamos do CRAN diretamente ele traz um arquivo binário para o nosso computador e o descomprime.

Package state



Exercícios

1. Vasculhem o CRAN e baixem qualquer pacote que vocês queiram. Qual função vocês devem usar?
2. Para baixar o pacote de uma pessoa aqui da sala, qual função devemos usar?
3. Com instalar o próprio pacote de vocês?
4. A pessoa A tem o pacote *pessoaA/a101* na versão 1. A pessoa B instalou o pacote *a101* na versão 1. Então, depois de um tempo, a pessoa A fez mudanças locais no seu pacote, criando a versão 2 do pacote. Sobre isso responda:
    a. Qual versão do pacote (1 ou 2) a pessoa A vai carregar quando usar a função *library*(a101)?
    b. Se a pessoa B der *remotes::install_github("pessoaA/a101")* qual versão será instalada (1 ou 2)?
    c. Se a pessoa A usar a função *devtools::load_all()* qual versão do pacote (1 ou 2) ela vai carregar?
    d. O que a pessoa A deve fazer para ter o seu pacote *a101* atualizado?
    e. O que a pessoa A deve fazer para que o pacote *a101* da pessoa B fique atualizado?

Relatórios

markdown, Rmarkdown, quarto

- markdown é uma linguagem de anotação, que “anota” um texto com marcações de programação, indicando onde deve ter negrito, itálico, título 1, título 2, título 3... etc.
- Rmarkdown foi um pacote desenvolvido por Yihui Xie originalmente e que depois foi incorporado oficialmente pela equipe do RStudio. Ele acoplava ao markdown a possibilidade de utilizar código R
- e agora temos o *quarto*, que é nativo do RStudio, que simplifica ainda mais a utilização de markdown no R, e permite ainda a integração com outras linguagens de programação, tais como python.

iniciando um documento quarto

- mostrar no R...

estruturas básicas de *quarto*

- mostrar no R...

Muito obrigado!

ric.feliz@gmail.com