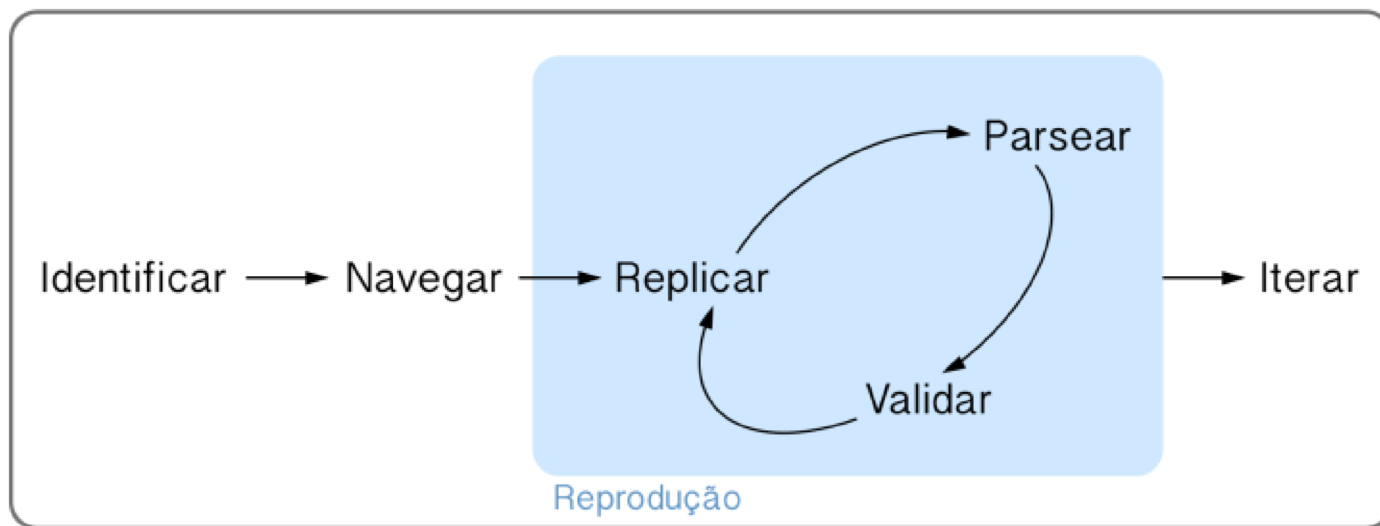
Web Scraping

Iteração

O fluxo do web scraping

- Sempre que fazemos um web scraper é bom seguir um fluxo definido
- Por enquanto já foram apresentados elementos da maior parte do passo-a-passo, mas nada foi dito sobre a iteração



Por que iterar?

- Dificilmente queremos fazer uma tarefa de web scraping uma vez só (senão bastaria baixar a página uma vez e raspá-la)

- Podemos querer baixar muitas páginas de uma vez ou uma página a cada certo tempo

- Iteração, tratamento de erros e automatização passam a ser relevantes

  - O pacote `purrr` nos ajudará a iterar

  - O pacote `purrr` retornará para tratar qualquer erro que possa aparecer

  - Falaremos de Github Actions na última aula

- Se você estiver interesse em aprender mais, veja nosso curso de Deploy!

Elementos comuns

- **Rodar em paralelo**. Quanto mais rápido, melhor!
- **Rodar com tratamento de erros**. Coisas dão errado no web scraping.
- **Utilizar barras de progresso**. Remédio para ansiedade.

Introdução

- Iteração é um padrão de programação extremamente comum que pode ser altamente abreviado

```
nums <- 1:10
resp <- c()
for (i in seq_along(nums)) {
  resp <- c(resp, nums[i] + 1)
}
resp
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
library(purrr)
map_dbl(nums, ~.x + 1)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

A função map

- A função `map()` recebe um vetor ou uma lista de entrada e aplica uma função em cada elemento do mesmo
- Podemos especificar o formato da saída com a família de funções `map_***()`
- A função pode ser declarada externamente, internamente ou através de um *lambda*

```
soma_um <- function(x) {
  x + 1
}
map(nums, soma_um)
map(nums, function(x) x + 1)
map(nums, ~.x + 1)
```

Utilidade do map

- Se tivermos uma lista de URLs, podemos iterar facilmente em todos sem abrir mão da sintaxe maravilhosa do Tidyverse

```
urls <- c(
  "https://en.wikipedia.org/wiki/R_language",
  "https://en.wikipedia.org/wiki/Python_(programming_language)
)
urls %>%
  map(read_html) %>%
  map(xml_find_first, "//h1") %>%
  map_chr(xml_text)
```

```
## [1] "R (programming language)"      "Python (programming language)
```

Tratando problemas

- Ao repetir uma tarefa múltiplas vezes, não podemos garantir que toda execução funcione
- O R já possui o `try()` e o `tryCatch()`, mas o `purrr` facilita ainda mais o trabalho

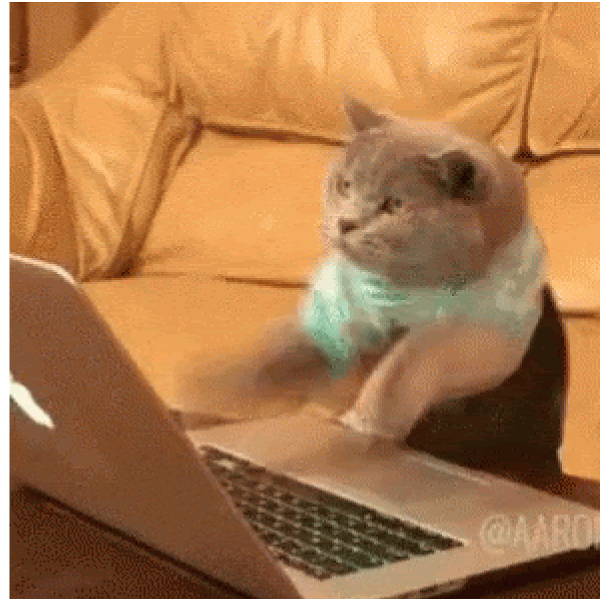
```
read_html("https://errado.que")
```

```
## Error in open.connection(x, "rb"): Could not resolve host: errado.
```

```
maybe_read_html <- possibly(read_html, NULL)
maybe_read_html("https://errado.que")
```

```
## NULL
```

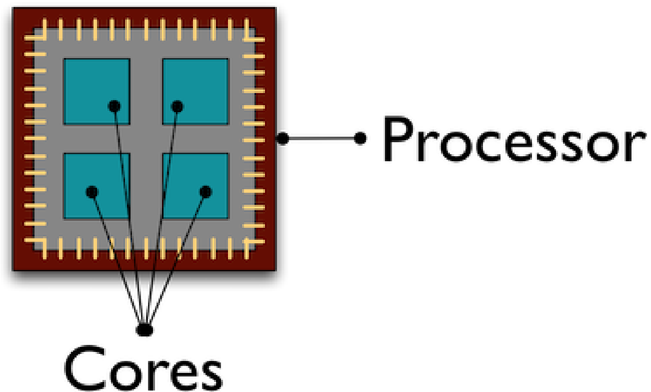
Vamos ao R!



Paralelismo

O que isso significa?

- Antigamente, computadores eram capazes de executar apenas uma sequência de comandos por vez
- Avanços tecnológicos permitiram que o processador fosse capaz de fazer "malabarismo" com diversos processos
- Paralelismo (ou multiprocessamento) chegou apenas com os primeiros *dual-core*



Em mais detalhes

- A unidade de processamento central pode ter mais de um **núcleo** (*multicore*)
- Um **processo** é composto por uma sequência de comandos ou tarefas
- Cada núcleo consegue executar apenas um **comando** por vez
- Os comandos de um processo podem ser interrompidos para que sejam executados os de outro (*multitasking*)
- O computador pode executar várias tarefas simultaneamente escalonando os comandos para seus diferentes núcleos (*multithreading*)
- Muitos computadores possuem **núcleos virtuais**, permitindo dois comandos por vez em cada núcleo (*hyperthreading*)

Exemplo mínimo

O pacote `parallel` já vem instalado junto com o R e consegue rodar comandos paralelamente tanto no Windows quanto em outros sistemas. Por padrão, ele quebra a tarefa em 2.

```
library(parallel)
library(tictoc)

tic()
res <- map(1:4, function(x) Sys.sleep(1))
toc()
```

```
## 4.011 sec elapsed
```

```
tic()
res <- mclapply(1:4, function(x) Sys.sleep(1))
toc()
```

```
## 2.024 sec elapsed
```

Futuros

- O pacote `{future}` expande o pacote `{parallel}`, permitindo o descolamento de tarefas da sessão principal
  - Ele pode operar em 2 níveis: *multicore* e *multissession*
- Em cima do `{future}`, foi construído o `{furrr}` com o objetivo de emular a sintaxe do `{purrr}` para processamento paralelo
- Diferentemente do `{parallel}`, o `{future}` é capaz de descobrir sozinho o número de núcleos virtuais do computador

```r
library(future)
availableCores()
```

```
## system
##      8
```

Barras de progresso

- Com o pacote `{progressr}` (recente!), é possível adicionar barras de progresso à suas chamadas, mesmo se a chamada for em paralelo.

```
# coloca o script no contexto
progressr::with_progress({

  # cria a barra de progresso
  p <- progressr::progressor(4)

  purrr::walk(1:4, ~{
    # dá o passo
    p()
    Sys.sleep(1)
  })

})
```

Como faz?

Vamos estabelecer um plano de execução paralela com a função `plan()`. Entender a diferença entre todos os planos disponíveis.

```
plan(multisession)
```

- `sequential`: não executa em paralelo, útil para testes
- `multicore`: mais eficiente, não funciona no Windows nem dentro do RStudio
- `multisession`: abre novas sessões do R, mais pesado para o computador

Como faz?

Agora vamos criar uma função que retorna o primeiro parágrafo de uma página da Wikipédia dado o fim de seu URL (como "/wiki/R_language"). Dicas: textos são denotados pela *tag* `<p>` em HTML; pule o elemento de classe "mw-empty-elt".

```
download_wiki <- function(url) {
  url %>%
    paste0("https://en.wikipedia.org", .) %>%
    read_html() %>%
    xml_find_first("//p[not(@class='mw-empty-elt')]") %>%
    xml_text()
}
```

Como faz?

Executar a função anterior em paralelo para todas as páginas baixadas no exercício de iteração. Dicas: utilize `future_map()` do pacote `furrr`; não se esqueça do `possibly()`"!

```
library(furrr)
prgs <- "https://en.wikipedia.org/wiki/R_language" %>%
  read_html() %>%
  xml_find_all("//table[@class='infobox vevent']//a") %>%
  xml_attr("href") %>%
  future_map(possibly(download_wiki, ""))
prgs[[3]]
```

```
## [1] "Programming paradigms are a way to classify programming langu
```

Vamos ao R!