PROMISES in R

```r
library(future)
library(promises)
library(dplyr)

promise = future(iris)

promise %...>%
    filter(Sepal.Length > 6) %...T>%
    nrow(.) %...>%
    group_by(Species) %...T>%
    print(.) %...T>%
    arrange(desc(Sepal.Girth)) %...!%
    {NULL}
```

```
library(future)
library(promises)
library(dplyr)

promise = future(iris)

promise %...>%
    filter(Sepal.Length > 6) %...T>%
    nrow(.) %...>%
    group_by(Species) %...T>%
    print(.) %...T>%
    arrange(desc(Sepal.Girth)) %...!%
    {NULL}
```

NO.

```r
library(future)
library(promises)
library(dplyr)

promise = future(iris)

promise %...>%
    filter(Sepal.Length > 6) %...T>%
    nrow(.) %...>%
    group_by(Species) %...T>%
    print(.) %...T>%
    arrange(desc(Sepal.Girth)) %...!%
    {NULL}
```

```r
promise = future(iris)

promise %...>%
  filter(Sepal.Length > 6) %...T>%
  nrow(.) %...>%
  group_by(Species) %...T>%
  print(.) %...T>%
  arrange(desc(Sepal.Girth)) %...!%
  {NULL}
```

```r
promise = future(iris)

promise %...>%
  filter(Sepal.Length > 6) %...T>%
  nrow(.) %...>%
  group_by(Species) %...T>%
  print(.) %...T>%
  arrange(desc(Sepal.Girth)) %...!%
  {NULL}
```

```
promise = future(iris)

promise %...>%
  filter(Sepal.Length > 6) %...T>%
  nrow(.) %...>%
  group_by(Species) %...T>%
  print(.) %...T>%
  arrange(desc(Sepal.Girth)) %...!%
  {NULL}
```

```
promise %...>%
    filter(Sepal.Length > 6)

then(
    promise,
    onFulfilled = function(x)
    filter(x, x$Sepal.Length > 6)
)
```

```
promise %...>%
  filter(Sepal.Length > 6)

then(
  promise,
  onFulfilled = function(x)
  filter(x, x$Sepal.Length > 6),
  onRejected = panicFunction
)
```

```
then(promise,
     onFulfilled = NULL,
     onRejected = NULL)

catch(promise, onRejected, tee = FALSE)

finally(promise, onFinally)
```

```
promise = future(iris)

promise %...>%
  filter(Sepal.Length > 6) %...T>%
  nrow(.) %...>%
  group_by(Species) %...T>%
  print(.) %...T>%
  arrange(desc(Sepal.Girth)) %...!%
  {NULL}
```

```
promise = future(iris)

promise %...>%
   filter(Sepal.Length > 6) %...T>%
   nrow(.) %...>%
   group_by(Species) %...T>%
   print(.) %...T>%
   arrange(desc(Sepal.Girth)) %...!%
   {NULL}
```

```
promise = future(iris)

promise %...>%
   filter(Sepal.Length > 6) %...T>%
   nrow(.) %...>%
   group_by(Species) %...T>%
   print(.) %...T>%
   arrange(desc(Sepal.Girth)) %...!%
   {NULL}
```

```
promise = future(iris)

promise %...>%
  filter(Sepal.Length > 6) %...T>%
  nrow(.) %...>%
  group_by(Species) %...T>%
  print(.) %...T>%
  arrange(desc(Sepal.Girth)) %...!%
  {NULL}
```

Promises - the R package

For those who ***need a while to finish***

Async in R

Async functions

They start things, and give you back a special object called a promise. If it doesn't return a promise, it's not an async function.

"Promises" same as in the NSE chapter of Hadley Wickham's Advanced R book?

No. The promises we're talking about are directly inspired by a central abstraction in modern **JavaScript**, and the JS folks named them "promises"

Many small or a few big?

It's mostly helpful for apps that have ***a few specific operations that take a long time, rather than lots of little*** **operations that are all a bit slow** on their own and add up to one big slow mess. We're looking for watermelons, not blueberries.

Shiny

Async programming is a major new addition to Shiny that can make certain classes of apps ***dramatically more responsive under load****.*
Because R is ***single threaded*** *(i.e. it can only do one thing at a time), a given Shiny app process can also only do one thing at a time: if it is fitting a linear model for one client, it can't simultaneously serve up a CSV download for another client.*

You can ***use promises with Shiny outputs****. If you're using an async-compatible version of* ***Shiny (version >=1.1)****, all of the* ***built-in renderXXX*** *functions can deal with either regular values or promises.*

Always keep your word!

`future(expr, …)`	Creates a **`promise object`**. **`expr`** - **R `expression`** (e.g. function call)
`then(promise, onFulfilled = NULL, onRejected = NULL)`	**`promise`** - A promise object **`onFulfilled`** - A function that will be invoked if the `promise` value successfully resolves. **`onRejected`** - A function taking the argument `error`. **`tee`** - if `TRUE`, ignore the return value of the callback, and use the original value instead. **`onFinally`** - A function with no arguments, to be called when the async operation either succeeds or fails.
`catch(promise, onRejected, tee = FALSE)`	
`finally(promise, onFinally)`	

Pipe	Usage	Equivalent with regular pipe	Description
`%...>%`	`promise %...>% func()`	`promise %>% then(func).%>% catch(func).`	**Promise pipe operators** Promise-aware pipe operators, in the style of **magrittr**. Like `magrittr` pipes, these operators can be used to chain together pipelines of promise-transforming operations. Unlike magrittr pipes, these pipes wait for promise resolution and pass the unwrapped value (or error) to the rhs function call.
`%...T>%`	`promise %...T>% func()`	`is equivalent to promise %T>% then(func).`	
`%...!%`	`promise %...!% func()`	`promise %>% catch(func)`	

`%...T!%`	`promise %...T!% func()`	`promise %T>% catch(func)`
		`promise %>% catch(func, tee = TRUE)`	

`promise_all(..., .list = NULL)`	**waits for multiple promise** objects to **all be successfully fulfill**	… - promise objects. **`.list`** - A list of promise objects--an alternative to . . .
`promise_race(..., .list = NULL)`	**waits for the first of multiple promise** objects to be either fulfilled or rejected.	

```
output$table <- renderTable({
    read.csv.async("https://rstudio.github.io/promises/data.csv") %...>% filter(state == "NY")
 })
```

```
Example with Shiny
```

Based on https://rstudio.github.io/promises/. Template by SlidesCarnival

CREDITS

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash

Graphics

- https://emojipedia.org/microsoft/