Developing internal tools for multi-lingual teams

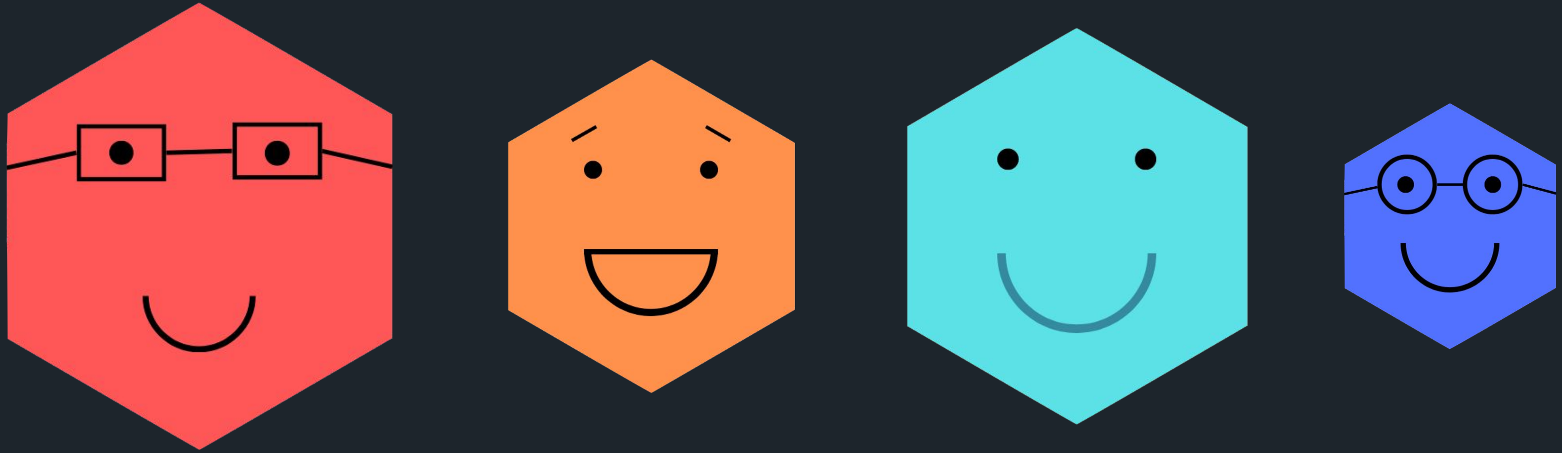
Jamie Ralph

rstudio::conf
July 2022

bumble

Photo by Klara Kulikova | Unsplash

18:05
badoo
18:05
bumble
18:05
FRUITZ



What strategies can we use to make developing tools simultaneously with Python and R easier?

Idea #1:
Building identical generic functions

print()

print()

summary()

print()

summary()

broom::tidy()

A simple generic function in R

```r
say_hello <- function(x) {
  UseMethod("say_hello")
}
```

```r
say_hello.data.frame <- function(x) {
  print("Hello dataframe!")
}
```

```r
say_hello.data.frame <- function(x) {
  print("Hello dataframe!")
}
say_hello.default <- function(x) {
  print("Hello there!")
}
```

Can we do this in Python?

```python
from functools import singledispatch
from functools import singledispatch


def say_hello(x):
    return "Hello there!"
```

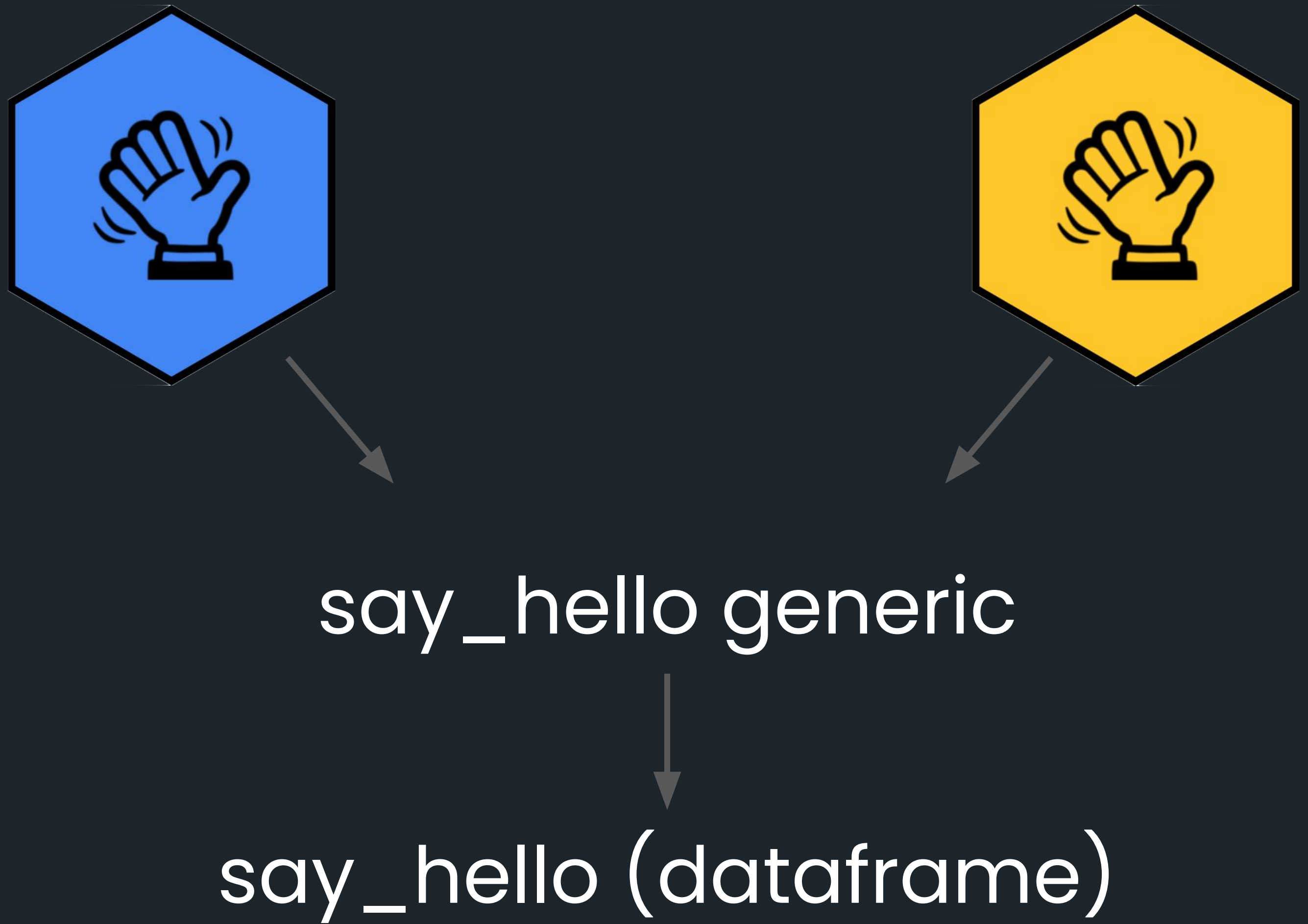
```python
from functools import singledispatch

@singledispatch
def say_hello(x):
    return "Hello there!"
```

```python
import pandas as pd

@say_hello.register(pd.DataFrame)
def greet_df(x):
    print("Hello dataframe!")
```

say_hello generic


say_hello generic
say_hello (dataframe)

```r
# funcs.R
# Method 1


# Method 2


# Method 3
```

```python
# funcs.py
# Method 1


# Method 2


# Method 3
```

Idea #2:
Identical error handling with classes

FileNotFoundError

ZeroDivisionError

KeyError

```python
class MyInternalError(Exception):
    pass
```

```python
raise MyInternalError("Better let IT know.")
```

```
try:
    # Some code goes here
except FileNotFoundError:
    # Code to handle FileNotFoundError
```

Can we do this in R?

```
stop("This throws an error!")
#> Error: This throws an error!

rlang::abort("This throws an error!")
#> Error:
#> ! This throws an error!
```

```
tryCatch(
  error = function(cnd) {
    # Code to run if error is thrown
  },
  # Code running with active error
    # handler
)
```

```r
abort_credentials_missing <- function() {
  rlang::abort(
    class = "error_credentials_missing",
    message = "Credentials not found!")
}
```

```r
abort_credentials_missing <- function() {
  rlang::abort(
    class = "error_credentials_missing",
    message = "Credentials not found!")
}
```

```
abort_credentials_missing()
#> Error in `abort_credentials_missing()`:
#> ! Credentials not found!
```

```r
tryCatch(
  error_credentials_missing = function(cnd) {
    # Code to handle missing credentials
  },
  # Code running with active credentials
  # error handler
)
```

What about error chaining?

```python
try:
    1 / 0
except ZeroDivisionError as e:
    raise MyInternalError("Error!") from e
```

```
try:
    1 / 0
except ZeroDivisionError as e:
    raise MyInternalError("Error!") from e
```

```
ZeroDivisionError: division by zero

The above exception was the direct
cause of the following exception:

MyInternalError
```

```r
rlang::try_fetch(
  error = function(cnd) {
    rlang::abort("An error!", parent = cnd)
  },
  df$x
)
```

```r
rlang::try_fetch(
  error = function(cnd) {
    rlang::abort("An error!", parent = cnd)
  },
  df$x
)
```

```
#> Error:
#> ! An error!
#> Caused by error in `df$x`:
#> ! object of type 'closure' is not subsettable
```

Idea #3: Creating your own internal wrappers

Calling Python from R

Importing Python modules into R

```
pymod <- reticulate::import("py_module")
```

```r
pymod <- reticulate::import("py_module")

# Call internal_function from py_module
pymod$internal_function()
```

```r
# R/zzz.R
pymod <- NULL
```

```r
# R/zzz.R
pymod <- NULL

.onLoad <- function(libname, pkgname) {
    pymod <<- reticulate::import(
              "py_module",
              delay_load = TRUE
    )
}
```

```r
# R/zzz.R
pymod <- NULL

.onLoad <- function(libname, pkgname) {
    pymod <<- reticulate::import(
                         "py_module",
                         delay_load = TRUE
    )
}
```

Sourcing Python files from R

```python
# inst/python/add.py
def add_two(x, y):
  return x + y
```

```r
source_python("inst/python/add.py")
add_two(5, 10)
```

Calling R from Python

rpy2

Importing R libraries into Python

```
from rpy2.robjects.packages import importr
from rpy2.robjects.packages import importr


base = importr("base")
```

```
from rpy2.robjects.packages import importr


base = importr("base")

r_sum = base.sum
```

```
from rpy2.robjects.packages import importr
from rpy2.robjects import IntVector

base = importr("base")

r_sum = base.sum

r_sum(IntVector([10, 10]))
```

Accessing functions from the R environment

```
import rpy2.robjects as ro
import rpy2.robjects as ro

r_code = "my_func <- function() {1}"
```

```python
import rpy2.robjects as ro

r_code = "my_func <- function() {1}"

ro.r(r_code)
```

```python
import rpy2.robjects as ro

r_code = "my_func <- function() {1}"

ro.r(r_code)

py_func = ro.globalenv["my_func"]
```

Defining a temporary package structure

```
from rpy2.robjects.packages import STAP
from rpy2.robjects.packages import STAP

r_code = "my_function <- function() {1}"
```

```
from rpy2.robjects.packages import STAP

r_code = "my_function <- function() {1}"

pack = STAP(r_code, "pack")
```

```
from rpy2.robjects.packages import STAP

r_code = "my_function <- function() {1}"

pack = STAP(r_code, "pack")

pack.my_function()
```

Now for the bad news...

Now for the bad news…

Two environments

Now for the bad news...

Two environments

Object conversion

Now for the bad news...

Two environments
Object conversion
Error messages

Generic functions

Errors as classes

Internal wrappers

Thank You