# Reticulate

# What is Reticulate ?

The R**eticulate** package provides an **R interface to Python** modules, classes, and functions.

-> It is very helpful when you code in Python but suddenly one task would be much easier to do in R
-> you use Reticulate to upload you Python code/object and use them in R

How to use it ?

In your R terminal:

```
> library(reticulate)
```

If you want to specify a Python version to be used you can do:

```
> use_python(« /usr/local/bin/python »)
```

Or use the functions use_virtualenv() or use_condaenv()

# Import Python modules

You import your Python **module** using import() function than you can use the Python functions related to this modules using **$**:

```
> os <- import("os")
> os$listdir(« .")
 [1] ".config"                "Music"                    ".condarc"
"Codes_python"           ".vim"                     ".DS_Store"
 [7] ".nedit"                 ".CFUserTextEncoding"   ".hgignore_global"
"Untitled.ipynb"         "vpn"                      ".local"
```

Works with useful and well known Python packages:

```
> numpy <- import("numpy")
> Z = np.arange(10,50)
Error in np.arange(10, 50) :
  impossible de trouver la fonction « np.arange"
> Z = numpy$arange(10,50)
> print(Z)
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49
```

Another very useful function: the Python help function:

```
> os <- import("os")
> py_help(os$chdir)
```

# Use Python scripts

Very useful if you have a whole **script** written in Python and you want to use it in your R code !

Example: you have a basic Python script « add.py » on your desktop:

```
def add(x, y):
  return x + y
```

In your R terminal:

```
> source_python('/Users/camille/Desktop/add.py')
> add(5, 10)
```

You can also execute Python code within the main module using the py_run_file and py_run_string functions. You can then access any objects created using the py object exported by reticulate. For example my script script.py creates the x variable:

```
> py_run_file(« script.py »)
> py$x
```

# Use Python context

The R with generic function can be used to interact with Python context manager objects. For example:

```
> py <- import_builtins()
> with(py$open("output.txt", "w") %as% file, {
>   file$write("Hello, there!")
> })
```

This example **opens** a file and ensures that it is **automatically closed** at the end of the with block. Note the use of the %as%operator to alias the object created by the context manager.

# Functions

By default R functions are converted to Python with a generic signature, where there's neither keyword argument nor default values for arguments. However, by applying the r_to_py() one can see that the signature of the wrapped function looks different than the original R function's signature. Besides, some Python libraries have strict checking on the function signatures of user provided callbacks. In these cases the generic function(...) signature will fail this checking.

For these cases you can use py_func() to wrap the R function so that the wrapped function has exactly the same signature as that of the original R function, e.g. one argument a without default value and another argument b with default value 1.5.

```
> wrapped_func <- py_func(function(a, b = 1.5) {})
> inspect$getargspec(wrapped_func)
ArgSpec(args=['a', 'b'], varargs=None, keywords=None, defaults=(1.5,))
```

Note that the signature of the R function must not contain esoteric Python-incompatible constructs. For example, we cannot have R function with signature like function(a = 1, b) since Python function requires that arguments without default values appear before arguments with default values.

# Interactive Python

If you want to work with Python interactively you can call the repl_python() function, which provides a Python REPL embedded within your R session. Objects created within the Python REPL can be accessed from R using the py object exported from reticulate. For example:

```
> repl_python()
> import pandas
> flights = pandas.read_csv(« file.csv »)
> flights = flights[flights['dest'] == "ORD"]
> flights = flights[['carrier', 'dep_delay', 'arr_delay']]
> flights = flights.dropna()
> exit
```

Note that Python code can also access objects from within the R session using the r object (e.g. r.flights).

# Be careful

Some R <-> Python issues might overcome, for example:

- If a Python API requires a list and you pass a single element R vector it will be converted to a Python scalar. To overcome this simply use the R list function explicitly:

```
> foo$bar(indexes = list(42L))`
```

R and Python have different default numeric types. If you write 42 in R it is considered a floating point number whereas 42 in Python is considered an integer. This means that when a Python API expects an integer, you need to be sure to use the L suffix within R. For example, if the foo function requires an integer as it's index argument you would do this:

```
> foo$bar(index = 42L)
```

- Python collections are addressed using 0-based indices rather than the 1-based indices you might be familiar with from R. So to address the first item of an array in R you would write items[[1]]

Whereas if you are calling a method in Python via reticulate that takes an index you would write this to address the first item:

```
> items$get(0L)
```

Note the use of the 0-based index as well as the L to indicate t that the value is an integer.

- To check whether a particular package is installed:

```
> py_module_available("pandas")
```