test python in Rstudio

mz

In a massive fit of procrastination I decided it was imperative that I learned linear algebra for deep learning... Namely I'd come accross this expanded tutorial walking through the Deep Learning book by Goodfellow et al. that added illustrations and python code for the Linear Algebra chapter.

Since I'd not even looked at python since I wrote a Space Incaders game for a coursera course on Interactive Programming with Python over 6 years ago, I thought it was about time to return to the scene of the crime.

But not without Rstudio, surely? I mean RStudio is like my emacs or vim i.e. I am irrationally attached to a working environment and will go to great lengths to stay within my comfort zone. Yes, I'm getting old.

So how does one run python in Rstudio? Well, a bit of research and it turns out there are a couple of options:

1. you can execute lines of code with python running in the terminal using `Ctrl+Alt+Enter`
2. you can call python via `knitr`: python is one of several language engines that `knitr` supports in addition to R, allowing you to write chunks in python that get executed during knitting.
3. from R you can use the `reticulate` package to call python from R where you can either execute code line by line or source `.py` scripts, or include them in knitr chunks.[1]

So option 1. is minimally invasive–all it's really doing is letting you run python in the terminal inside Rstudio, so you don't have to switch between windows.

Option 2. allows you to use both R and python in the Rstudio environment, but they are not necessarily communicating just yet. This could be used as an RStudio alternative to jupyter notebooks, but that's probably a very bad case of reinventing the wheel.

Option 3 is more very specifically dedicated to the interoperability of R and python, i.e. creating a shared environment for objects that can be accessd by both engines.

Option 2 can easily be upgraded to option 3 by simply loading the `reticulate` package, which allows you to work with a hybrid notebook environment with both modules accessible to each other.

So let's see if we can make all of these options work in practice.

Prerequisites

But let's start at the beginning. For this tutorial you will need:

- a working installation of python (go to https://www.python.org/ and click on Downloads and select your platform. Go down this rabbithole to decide if you want python 2 or python 3.
- `RStudio` version 1.1 or newer
- `knitr` package in R, version 1.18 or newer
- `reticulate` package in R

Running python in the RStudio terminal

Since RStudio 1.1 you can access the system shell directly as a tab in the RStudio IDE, next to the console tab. Switch between the terminal and console tabs using `Shift+Alt+T` and get back to the console with `Ctrl+2`.

[1] In fact, if I understand this correctly `knitr` is using the `reticulate` package anyway, but I don't really get the behind the scenes, so don't take my word for it.

If you have python installed on your system simply move to the terminal and type in `python` to start the python interpreter.[2]

You can now enter python code directly into the terminal (not the reproducible way of doing things) or send code to the terminal from a script.

So you can open a new script file (`Ctrl+Shift+N`) and type a simple python assignment:

```
x = [1, 2]
```

```
x
```

You can run the code by highlighting both lines and pressing `Ctrl+Alt+Enter`, or by placing your cursor in the first line and stepping through the code using the same keyboard shortcut. The output will of course be printed in the terminal window:

```
>>> x = [1, 2]
>>> x
[1, 2]
```

Knitting python and text (and R)

So option two is knitting text and code, in this case python code, by simply declaring the engine at the begining of the chunk declaration. So start a new `.Rmd` file and try out a `python` chunk, this one checks the version of python we are using:

````
```{python}
import sys
print(sys.version)
```
````

```
## 2.7.15 (v2.7.15:ca079a3ea3, Apr 30 2018, 16:30:26) [MSC v.1500 64 bit (AMD64)]
```

So far so good. Now let's add an R chunk into the same file, creating the vector `x` in R:

````
```{r}
x <- 4:1
x
```
````

```
## [1] 4 3 2 1
```

Now of course the python interpreter doesn't know antyhing about the x we just defined in R:

````
```{python,  error=TRUE}
print(x)
```
````

```
## NameError: name 'x' is not defined
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

And the same would apply if we created an object in python and tried to access it in R.

````
```{python}
y = [1,2,3,4]
y
```
````

[2] Actually in the near future you might be able to skip this step altogether as sending python code to the terminal will automatically start an instance of python as per this feature request

```
## [1, 2, 3, 4]
```

And if we look for it in an R chunk it is of course not there:

```
```{r,  error=TRUE}
print(y)
```
```

```
## Error in print(y): object 'y' not found
```

That's where option 3 comes in, we need the python and R modules to be able to communicate with eachoter.

Reticulating python[3]

So after the usual package installation you can now load the `reticulate` package in an R chunk, and try again to access the python object `y`:

```
```{r}
library(reticulate)
print(py$y)
```
```

```
## [1] 1 2 3 4
```

And voila! Using the `py$` prefix, R now has access to all the objects created in the instance of python we are running. One issue to note is that both languages have different data types, although the conversion is pretty straightforward. In a list in python got converted into a vector in R.

And the opposite will happen if we try accessing the R environment from python, which we do using the `r.` prefix:

```
```{python}
print(r.x)
print(type(r.x))
```
```

```
## [4, 3, 2, 1]
```

```
## <type 'list'>
```

Here we also ckecked to see that x is now a list in python. Although they are mainly logical, you can look up 'type conversions' on this site to make sure.

Now we can of course preform an operation using both objects in whichever environment we wish:

```
```{r}
py$y  + x
```
```

```
## [1] 5 5 5 5
```

What about adding the vectors up in python? Well, they are not vectors here, they are lists, and summation of lists is of course a very different beast:

```
```{python}
print(y + r.x)
print(map(sum, zip(y , r.x)))
```
```

[3] I had to look this up: reticulated, comes from latin reticulum, meaning network, net-like structure. Due to it's complex networked pattern a species of python from South east Asia is called the reticulated python. Ergo `reticulate` became the name for the package that 'networks' or interfaces between python and R.

```
## [1, 2, 3, 4, 4, 3, 2, 1]
```

```
## [5, 5, 5, 5]
```

Now this might seem totally mundane, but it is actually pretty neat. In fact, up until terrifically recently knitr chunks using python would not even persist the same session over chunks, meaning each new chunk would forget anything you did in a previous one.

As we have seen they are now all operating in one session, one that shares objects with the R session as well, if you use the `reticulate` package of course. The only issue that you might come up against is that this only works if you knit the whole document together–not if you try to run individual chunks, although that might also change in the future.

Reticulating with python sans knitr

We eased into `reticulate` via `knitr`, but of course you can use it directly in an R script as well. The main tools from the `reticulate` package that you will require are:

- `import()` to import python packages
- `py_run_string()` to run lines of code
- `py_run_file()` to run whole files
- `source_python()` to run whole files and make outputs R objects

So in order to import a module, e.g. numpy, we can do the following:

```
np <- import("numpy")
mat <- matrix(c(1,2,3,4), c(2,2))
np$shape(mat)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 2
```

So here I imported the NumPy package into R and then used the `shape` command from that package to get the 'dimensions of the array `mat`', which is equivalent to the `dim()` function in R.

In order to run a line of python code I can use the `py_run_string()` command, which again exports any objects thus created into the `py` object:

```
py_run_string("x = r.mat.shape")
py$x
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 2
```

So step through that one carefully: inside the python module I used an r object, hence the `r.` in front of the `mat` matrix name, and applied a numpy fucntion, `shape`. This was assigned to x inside the python module, so in order to access it in R I used the `py$` prefix. My brain hurts just a little bit now.

OK, so next let's try sourcing a script. There are two ways of doing it with reticulate, we'll try both to see what the difference is.

So let's first create a python script, I'll do one direct form R for sake of completeness:

```
file.conn <- file("my_script.py")
writeLines(c("my_var = 42"), file.conn)
close(file.conn)
```

Now let's try the two different source commands:

```
py_run_file("my_script.py")
print(my_var)
```

```
## Error in print(my_var): object 'my_var' not found
```

```
print(py$my_var)
```

```
## [1] 42
```

So using the `reticulate::py_run_file()` function exports the `my_var` object created by the script into the R environment into the `py` object, so it can be accessed using the `py$` prefix.

And what happens if we instead source it with the `reticulate::source_python()` command?

```
rm(list = ls())
source_python("my_script.py")
print(my_var)
```

```
## [1] 42
```

```
print(py$my_var)
```

```
## [1] 42
```

Now it actually exports it directly into the R environment so we can access it using its name directly.