Python vs. R - The Good, The Bad, and the Ugly

This is a **Jupyter Notebook** with Python3, but... cool hack: we can also run R in cells in the same Notebook!

Some house keeping

Virtual Environments

I use https://docs.python.org/3/library/venv.html (https://docs.python.org/3/library/venv.html) to create virtual environment (python3) on my Mac. (Python 2 is going away...)

Jupyter Notebooks

I don't use the ananconda installation of jupyter because they don't keep up with all the python libraries. In install Jupyter Notebook in my Python3 environment

Use `pip3 install jupyter`

Output Jupyter Notebook to PDF

You'll need a couple of more things:

- http://pandoc.org/installing.html (http://pandoc.org/installing.html)
- https://nbconvert.readthedocs.io/en/latest/install.html#installing-tex (https://nbconvert.readthedocs.io/en/latest/install.html#installing-tex) - need for LaTex
- (and some tweaking of PATH)

The output will go to your Download folder But it might be easier to "print" as PDF

Jupyter Notebook magic

Lots of cool stuff with Jupyter Notebook (old name: iPython) Magic:

- https://ipython.readthedocs.io/en/stable/interactive/magics.html (https://ipython.readthedocs.io/en/stable/interactive/magics.html)

Many languages are suppored in Jupyter Notebooks

- https://github.com/jupyter/jupyter/wiki/Jupyter-kernels (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels)

Running both Python and R in same Notebook

You probably don't want to do this in Real Life, but...

If you want to use R with Python in Jupyter Notebook, need:

- https://rpy2.github.io/doc/v2.9.x/html/index.html (https://rpy2.github.io/doc/v2.9.x/html/index.html)

NOTE: you have to be running jupyter notebook under python3, unless you have previously installed rpy2 under python 2

On your command line:

```
pip3 install rpy2 tzlocal
```

Setup

In [1]:
```
%load_ext rpy2.ipython
# This is allows the us to run R in cells with %%R
```

In [2]:
```
import sys
print(sys.version_info)
print(sys.version)
```

```
sys.version_info(major=3, minor=7, micro=3, releaselevel='final', serial=
0)
3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)]
```

In [3]:
```
%%R
getRversion()
```

```
[1] ‘3.5.3’
```

Every Computer Language is a collection of features and compromises

- Some features are **Good** (depending)
- Some features are **Bad** (depending)
- Some features are **Ugly**

Which language is better (best?)

Depends. Which tool is better: screw driver or hammer?

You're a better handyperson if you know how to use more than one tool!

It's a bit like being bilingual.

- beware of *false friends*: expressions in different languages that look the same but aren't
  - *demand* in French means *ask*; in English it means *ask forcefully*

For Data Science, there's an important consideration

Which language supports the library that I need to do my job?

- R has better support for **statistics** (especially more obscure/sophisticated)
- Python has better support for **Machine Learning** and many other libraries
    - Python is more widely used.
    - Python has a fair amount of statistical support, so R may be less useful.

Underlying Philosophy

- Python was designed and implemented by "Computer Scientists"
- R was designed and implemented by "Statisticians"
    - Joke: The good news about R is it's a language designed by statisticians. The bad news about R is...

R and Python are *Object Oriented* (OO)

Object Oriented Programming (OOP), has many advantages:

- you can package up different groups of functions together
- you create a separate name space, so two different authors can use the same name for a function
- You can use objects instead of using dictionaries or R's "list" to compartmentalize named values.

Good

- Fortunately, OOP in Python was designed and implemented by Computer Scientists.
- Python also has `collections.namedtuple` which gives you OOP syntax but lighter weight

Bad

- **Ugly**
- Unfortunately, OOP in R was not designed and implemented by Computer Scientists. As a result, there are 3 major (and various minor) versions of R's OP. OOP notation not built into language. Gory details here: https://adv-r.hadley.nz/oo.html (https://adv-r.hadley.nz/oo.html)

R and Python are *Free*

- Free is **Good**
- Free is **Bad** not professionally maintained and designed, bugs!
- Free is **Ugly**
    - R has inconsistently named libraries, interfaces, names, error messages, and namespace collisions. Documentation is often lacking.
    - In R, you can specify function from a library with this syntax: *LIBRARY*::*FUNCTION*. Example: `dplyr::intersect(first, second)`
    - with Python, this is less of an issue, because of recommended standards
    - with both, you have to worry about consistency/dependency between libraries but there are ways package up your environment with specific versions of the libraries.

    - with both, some libraries were re-implemented, so picking a "best" library is hard: R has three graphics packages. I'm not sure how many Python has... This is because there's no enforcing standards committee for libraries.

R and Python are *Interpreted*

Two types of computer languages:

- Interpreted
    - Faster to get a resulting program, but it will run slower.
    - Most programs are run only once!
- Compiled
    - Slower to get a resulting program, but it will run faster
    - can be big savings over time

Good - Interpreted means experimentation is fast

- Most programs don't even run once!

Bad - Interpreted means the code could be impractical to run in production

- Often Optimized libraries are reimplemented in C (or other languages) to run faster

Ugly - there are ways to speed up the code but it's not pretty...

- Often critical libraries are rewritten to run fast but they can be clumsy to use
- Python and R weren't designed to run in parallel, but there are hacks!
- Using double FOR loops on matices is SLOW; better to use libraries

Assignment of Python and R

In [4]:
```
x = 1
x
```
Out[4]:
```
1
```

In [5]:
```
%%R
x <- 1  # I prefer this, as it reminds me that I'm using R!
x = 1   # allowed
x
```
```
[1] 1
```

R Variable Naming - Ugly

In R:

- Periods **.** are treated as an allowable character in variable names
- variables can't start with an underscore _

In [6]:
```
%%R
# DON'T BE FOOLED INTO THINKING is IS AN OBJECT
print(is.null(1))
print(is.character("a"))
internal_ <- NULL
print(is.null(internal_)) # trailing underscore is OK


print(is.that)            # that() isn't a member of "is"
```

```
R[write to console]: Error in print(is.that) : object 'is.that' not found
Calls: <Anonymous> -> <Anonymous> -> withVisible -> print


[1] FALSE
[1] TRUE
[1] TRUE
Error in print(is.that) : object 'is.that' not found
Calls: <Anonymous> -> <Anonymous> -> withVisible -> print
```

In [7]:
```
%%R
_internal <- 4   # BAD: leading underscore
```

```
R[write to console]: Error in (function (file = "", n = NULL, text = NUL
L, prompt = "?", keep.source = getOption("keep.source"),  :
  <text>:1:14: unexpected input
1: withVisible({_
                 ^




Error in (function (file = "", n = NULL, text = NULL, prompt = "?", keep.
source = getOption("keep.source"),  :
  <text>:1:14: unexpected input
1: withVisible({_
                 ^
```

What Constitutes the Basic "Operand/Entity" in Python vs R?

In Python it's "Object" or built-in type (integer, string, etc.)

In R, it's a "Vector"

```
In [8]: a = 1
        print(type(a))

        b = [1, 2, 3]
        print(type(b))

        c = (1, 2, 3)
        print(type(c))  # immutable collection

        # And lots of other Python aggregate classes
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
```

```
In [9]: %%R
        a <- 1
        print(class(a))

        b <- c(1, 2, 3)     # R is a bit clumsier creating a "vector"
        print(class(b))

        # R has a type called "list" but it's very different from what Python and C
```

```
[1] "numeric"
[1] "numeric"
```

Operating on Basic Entities in Python vs R

- These operations make sense for CS, but not so much to a Statistician

```
In [10]: # Adding two arrays in Python: concatenation
         v1 = [1,2,3]
         v2 = [4,5,6]
         v1+v2
```

```
Out[10]: [1, 2, 3, 4, 5, 6]
```

```
In [11]: # Multiplying an array in Python: replication
         3*v1
```

```
Out[11]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
In [12]: v1*v2
```

```
---------------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-12-6334bfe1b8b9> in <module>
----> 1 v1*v2

TypeError: can't multiply sequence by non-int of type 'list'
```

```
In [ ]: %%R
        # Adding two arrays in R: parallelization
        v1 = c(1,2,3)
        v2 = c(4,5,6)
        v1+v2
```

```
In [ ]: %%R
        # Multiplying an array in R: parallelization
        3*v1
```

```
In [ ]: %%R
        v1*v2
```

Enter Python numpy

Numpy is Python's Numeric library. It has its own array implementation which is more efficient and allows for more R type syntax.

Drawback, you have to remember if an object is numpy or regular python

```
In [ ]: import numpy as np  # import the numpy library and allow for the nickname '

        w1 = np.array( [1, 2, 3] )
        w2 = np.array( [4, 5, 6] )
        print(w1+w2)
        print(3*w1)
```

```
In [ ]: type(w1)
```

```
In [ ]: assert(type(w1)== list)
```

Accessing an Array/Vector in Python vs. R

- Computer Scientists
    - like to start counting from 0. (Makes sense from Computer Architecture)
    - like to stop counting at N-1
        - BUT: in python code, often N shows up with the understanding the end is N-1

- Everyone else, including statisticians
    - like to start counting from 1. (Make sense from Prehistoric times).
    - like to stop counting at N

```
In [ ]: print(v1[0])
        print(v1[2])
        print(v1[3])
```

```
In [ ]: v1[3] = 4  # Python protects you when you goof, but less "convenient"
```

```
In [ ]: %%R
        print(v1[1])
        print(v1[3])

        #
        print(v1[0])
        print(v1[4])  # oops!
```

```
In [ ]: %%R
        v1[5] <- 5   # Convenient but potentially dangerous
        v1
```

Accessing Multiple Elements from Array/Vector in Python vs. R

Python's numpy array and R's vector can handle a "list" of multiple indices

```
In [ ]: w1[[1,2]]
```

```
In [ ]: %%R
        v1[ c(2,3) ]
```

BUT regular Python arrays works differently

It needs a "slice" or a **List Comprehensions**

```
In [ ]: # non-numpy array
        v1[[1,2]]
```

```
In [ ]: v1[1:3]  # slice does NOT include 3
```

```
In [ ]: # List Comprehensions
        [ v1[i] for i in [1,2]]
```

Accessing an Array/Vector with NEGATIVE indices in Python vs. R

Address an element of an array in assembly code or C is usually a bad idea.

Idea! Why not use negative indices to mean something "useful"?!!

- Computer Scientists are always accessing an element from the end of an array with the algorithms
  - Use negative index to represent the end of the array
  - less ugly than `v1[len(v1) -1]`
- Statisticians are always removing elements of an array

```
In [ ]: print(v1[-1])   # access element 3
        v1[len(v1) -1]
```

```
In [ ]: %%R
        print(v1[-1])  # drops element 1
        v1[length(v1)]  # no minus one, since R counts from 1
```

```
In [ ]: import numpy as np

        a = np.array([1,2,3])
        b = np.array([1,2,3])
        a+b
        3*a
        a[3] = 4
```

Python Dictionary vs. R List

What R calls a "List" is different from what computer science calls a "List"

R's list is more like an ordered dictionary.

BUT: R's list, the keys cannot be numbers

Syntactically:

- Python, a "list" (array) and dictionary looks the same. Python's dictionary can be thought of as an **associative array**, so that syntatically, addressing by key in a dictionary looks the same as addressing a list by index. Advantage: if you change your mind about using a list vs. dictionary (vs. numpy array), you don't have to change the syntax.
  - BUT if **insertion order** is important, use `OrderedDict`
  - BUT if **key order** is important, use `sortedcontainers.SortedDict`

in R, accessing elements in vector and "list" looks different. Accessing an element of a "list" looks ugly.

In [ ]:

```
# Most like R's "list"
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
od[5] = 'five'          # key for Python sd can be an integer
print(od['b'])          # access by key
print(list(od)[1])      # for  iloc to access by index
od
```

In [ ]:

```
# Keeping keys in Sorted Order
# pip install sortedcontainers
# These are similiar
# collections.OrderedDict:
# sortedcontainers.SortedDict but SortedDict has iloc() and requires keys b
from sortedcontainers  import SortedDict
sd = SortedDict()
sd['a'] = 1
sd['b'] = 2
sd['c'] = 3
sd['d'] = 4
sd['5'] = 'five'        # key for Python sd cannot be an int if other keys
print(sd['b'])          # access by key
print(sd.keys()[1])     # for  iloc to access by index
sd
```

In [ ]:

```
%%R
rlist <- list(
    "a" = 1,
    "b" = 2,
    "c" = 3,
    "d" = 4,
    5 = "five"  # ERROR: numerics can't be a key!  This is because ints can
)
rlist
```

In [ ]:

```
%%R
rlist <- list(
    "a" = 1,
    "b" = 2,
    "c" = 3,
    "d" = 4,
    "5" = "five"  # but a number as a string can be a key
)
rlist
```

In [ ]:

```
%%R
print(class(rlist))
print(class( rlist['a'] ))  # with a single bracket, you get back a list an
rlist['a']
```

In [ ]:
```
%%R
rlist[['a']]    ## Access single item
```

In [ ]:
```
%%R
rlist[['a']] = "hi"
rlist
```

IMPORTANT: R list can hold heterogeneous collection of data types

Python arrays and tuples can hold heterogeneuous collection of types. While more flexible, the implementation for this is inefficient.

R vector's can only be the same type. CAUTION: If you try to create a R vector of multiple types, R will silently convert the types.

In [ ]:
```
('a', 0, False)   # tuple
```

In [ ]:
```
['a', 0, False]   # Python list
```

In [ ]:
```
%%R
c(1,   TRUE)   # WITHOUT WARNING: Here, R vector converts everything to NUMBE
```

In [ ]:
```
%%R
c(1, "string", TRUE)   # WITHOUT WARNING: Here, R vector converts everything
```

In [ ]:
```
%%R
# Use R's "list" if you want to have a collection of heterogenuous items
complex <- list(1, list(1, "string", TRUE)) # Nested list, no keys
complex
```

In [ ]:
```
%%R
complex[[2]][2] # get second item from second item (which is a list)
```

What is Truth?

Conditional statements are the work horse of any language but every language implements what is TRUE or FALSE differently!

It's hard to remember the implementation details, be careful if argument isn't explicitly a Boolean in Python or R.

Sometimes, the error message can be obscure.

R has the additional complication of having `NULL` and `NA` as well as `NaN`. If you accidentally test these values for TRUE/FALSE, you'll get a somewhat mysterious error message.

** Use R's built in tests**:

- `is.nan()`
- `is.null()`
- `is.na()`

In [ ]:
```
if 0:
    print("TRUE")
else:
    print("FALSE")
```

In [ ]:
```
if []:
    print("TRUE")
else:
    print("FALSE")
```

In [ ]:
```
if ():
    print("TRUE")
else:
    print("FALSE")
```

In [ ]:
```
if (None):
    print("TRUE")
else:
    print("FALSE")
```

In [ ]:
```
if (float('nan')):
    print("TRUE")
else:
    print("FALSE")
```

In [ ]:
```
%%R
if (0)
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (c())
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
myNA <- NA
if (myNA)
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (is.na(myNA))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
myNULL <- NULL
if (myNULL)
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
myNULL <- NULL
if (is.null(myNULL))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
myNaN <- NaN
if (myNaN)
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (is.nan(myNaN))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
is.null(sqrt(-1))
```

R's treatment of boolean vectors can be tricky but it is powerful

R and Python both have `any()` and `all()` .

In [ ]:
```
%%R
# Looks like R just checks the first element
if (c(0, 1))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (c(1, 0, 0, 0, 0, 1))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (any(c(0, 0, 1, 0, 0, 0)))
    print("TRUE")
else
    print("FALSE")
```

In [ ]:
```
%%R
if (all(c(0, 0, 1, 0, 0, 0)))
    print("TRUE")
else
    print("FALSE")
```

DataFrames

The heart of doing Data Science is putting your data into a **DataFrame.** Superficially, it looks like a matrix or if you are more sophisticated: a table from a spreadsheet.

Key: the main design and implementation consideration of DataFrames is to make Data Science and Machine Learning Operations more efficient.

- A DataFrame is more restricted than a matrix
- IF abused, DF will have unexpected behavior.
- A **column** of a DataFrame has the same data type throughout
    - Homgenuous Data Type has a more efficient layout in memory
- A **row** of a DataFrame represents a heterogenuous collection of data.
    - Allows for heterogenuous data for a training set example

ID	Name	Weight	Smokes	Ethnicity
24	Sue	115.2	TRUE	NativeAmerican
45	Sam	230.8	FALSE	PacificIslander

Columns:

- ID is an integer
- Name is a string, arbitrary number of these
- Weight is a float
- Smokes is a boolean
- Ethnicity: could be a string OR could be a "category"/"factor": a predetermined symbol that represents an **unordered value** represented as a small set of ints (more efficient)
    - Can't (easily) add to existing set of factors

Rows:

- is an instance of one training set example

Good

- R understood the importance of this and has a clean implementation, build into the language and somewhat easy to use syntax for it.

Bad

- Python was designed as a general purpose language. The idea of a DataFrame came later, but the syntax isn't as consistent
  - numpy - homogenuous matrix, not columns - can't(always?) be used for ML
  - Pandas - hacked on top of Python data structure
  - others? (reinventing the wheel!)

  Python and R have similar constructs in constructing a DataFrame, about the same in ease of use. There are MANY different ways to build a DataFrame! Check the documentation

H2O

- I often use http://h20.ai (http://h20.ai) for my Machine Learning, which has it's own DataFrame. H2O has Python and R API's

References

https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c (https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c)

Usually load data from files

- you probably won't do this much...

In [ ]:
```
import pandas as pd

# Create Data per Row
data = [[24, 'Sue',115.2, True, 'NativeAmerican'], [45, 'Sam', 230.8, False

# Create Data per Column
data = {'ID':[24, 45 ],
        'Name':['Sue', 'Sam'],
        'Weight':[115.2, 230.8],
        'Smokes':[True, False],
        'Ethnicity':pd.Series(['NativeAmerican', 'PacificIslander'], dtype=
df = pd.DataFrame(data, columns=['ID', 'Name','Weight', 'Smokes', 'Ethnicit
print(df)
```

```
In [ ]: %%R
        df <- data.frame(
            # insert by columns, with type of column implied by data
            ID = c(24, 45),
            Name = as.character(c('Sue', 'Sam')),
            Weight = c(115.2, 230.8),
            Smokes = c(TRUE, FALSE),
            Ethnicity = as.factor(c('NativeAmerican', 'PacificIslander'))
             , stringsAsFactors=FALSE  # Default is default.stringsAsFactors() whic
        )
        df
```

Note: You can give rows symbolic names, like "First", but this is something I never do in practice, so enough said.

Did we create what we think we did?

Here's how to check the types

```
In [ ]: df.dtypes
```

```
In [ ]: %%R
        str(df)  # Confusing name!  str = structure
```

```
In [ ]: %%R
        # shocking ugliness in R: "mode" has a meaning in statistics, but R ignores
        mode(df)
        # In R, a DF is implemented as a list of vectors.
```

```
In [ ]: %%R
        typeof(df)
```

```
In [ ]: %%R
        class(df)
```

Check Column Names

```
In [ ]: list(df)  # not obvious!
```

```
In [ ]: %%R
        colnames(df)
```

Count of Rows

In [ ]:
```
print(df.shape)
len(df)  # df.shape[0]
```

In [ ]:
```
%%R
print(length(df))  # EASY MISTAKE TO MAKE IN R.  This count of columns!
print(dim(df))  # rows, columns
nrow(df)
```

Summary

In [ ]:
```
df.describe()  # ignores non-numeric columns
```

In [ ]:
```
df["Ethnicity"].describe()
```

In [ ]:
```
%%R
summary(df)
```

In [ ]:
```
%%R
table(df$Ethnicity)
```

Head

Look at the first few rows to see if they look OK

In [ ]:
```
df.head()
```

In [ ]:
```
%%R
head(df)
```

Accessing Data from DataFrames

Good

- R is matrix-like notation

Bad

- Python's Panda feels arbitrary. Consult the documentation! Document your code!

Ugly

- Behavior of Data Frames will surprise you in both languages. It pays to double check to make sure you know what you are actually getting!

In [ ]:
```
print(type(df["Name"]))  # A Column is actually implemented as a "Panda Ser
df["Name"]  # SINGLE square bracket returns a PANDA SERIES
```

In [ ]:
```
print( type(df[["Name"]]) )
df[["Name"]] # DOUBLE square bracket returns a DATAFRAME
```

In [ ]:
```
%%R
print( class(df["Name"]) )
             (df["Name"])
```

In [ ]:
```
%%R
print( class(df[, "Name"]) )
             (df[, "Name"])
```

Panda Accessing Columns by an INDEX: `iloc()`

In [ ]:
```
df.iloc[:, 1]
```

Panda Accessing Columns by a NAME: `loc()`

In [ ]:
```
df.loc[:, "Name"]
```

R uses the same matrix-like syntax for both

In [ ]:
```
%%R
df[, 2]
```

In [ ]:
```
%%R
df[, "Name"]
```

Accessing by Row

In [ ]:
```
df.iloc[0]
```

In [ ]:
```
%%R
df[1, ]
```

Accessing by Row and Column

In [ ]:
```
df.iloc[0]["Smokes"]  # Just a Boolean
```

In [ ]:
```
%%R
df[1, "Smokes"]  # Not a DataFrame!
```

In [ ]:
```
df.iloc[0][["Smokes"]]  # DataFrame of one cell!
```

In [ ]:
```
%%R
df[c(1), c("Smokes")]
```

In [ ]:
```
df.iloc[0:2][["Name", "Smokes"]]
```

In [ ]:
```
%%R
df[c(1:2), c("Name", "Smokes")]
```

In [ ]:
```
bool_vector = df["Weight"] < 200
bool_vector
```

In [ ]:
```
df[bool_vector]
```

In [ ]:
```
%%R
bool_vector <- (df["Weight"] < 200)
print(class(bool_vector))
(bool_vector)
```

In [ ]:
```
%%R
print( class(df[bool_vector, ]))
             df[bool_vector, ]
```

In [ ]:
```
%%R
print( class(df[bool_vector ]))
             df[bool_vector ]  # Probably NOT what you want: mix types get
```

In [ ]:
```
%%R
# Better way is to use "which"
bool_vector <- which(df["Weight"] < 200)
print(class(bool_vector))
(bool_vector)
```

In [ ]:
```
%%R
print( class(df[bool_vector, ]))
             df[bool_vector, ]
```

Summary

- We've gone over a lot of Language Nits in Python vs. R

Bottom line:

- **CHECK YOUR CODE CAREFULLY** especially if you're going back and forth between the two languages

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Graphing

(Make a separate talk some day....)

Good

- R recognized immediately the importance of the Graphics support
- Python added this later

Ugly

- R has three different plotting packages
- Python has multiple ones as well

```
In [ ]:
```