Data Analytics with HPC

Practical – Data Cleaning with Python

Reusing this material

Overview

- Practical Aim:
  - To practice some common techniques for cleaning and preparing data directly in Python
- Practical based on Section 2 of "An introduction to data cleaning with R" from Statistics Netherlands
  - Available on CRAN at http://cran.r-project.org/doc/contrib/de_Jonge+van_der_Loo-Introduction_to_data_cleaning_with_R.pdf

epcc

THE UNIVERSITY OF EDINBURGH

Practical Contents

- Part 1 – using pandas read_csv() to read csv data into a data frame, this illustrates
  - Header row
  - Setting column names
  - Using column classes
  - Coercion
- Part 2 – dealing with unstructured text data. Artificial example that illustrates various techniques
  - Pattern matching and regular expressions
  - Python lists and functions
  - More coercion

Part 1

- Reading data into a data frame

Logging in and getting started on your own laptop

Prerequisites

- Python 2.7 and Conda.

Command line install:

- conda create --name pythonData
- conda install -n pythonData Jupyter pandas
- source activate pythonData
- jupyter notebook

- Open http://localhost:8888 in browser.

Logging in and getting started on RDF

- Open a terminal window and run the following commands:
- # Login
- > ssh username@login.rdf.ac.uk
- # Load python modules
- > module load python
- > module load anaconda
- # Create working directory
- > mkdir dataCleaning
- > cd dataCleaning
- # create the input data files
- nano unnamed.txt
- nano daltons.txt
- ipython notebook --no-browser --port=8889 (or any no. 8000-9000)

On Mac or Unix connection to RDF

- Open another terminal window (mac or Unix) and run this command:
- ssh -N -f -L localhost:8888:localhost:8889 username@login.rdf.ac.uk

- Go to http://localhost:8888 in your browser. Open the dataCleaning directory and create a new notebook to work in.

Windows connection to RDF

- Open putty

set ssh connection:

- Host Name: user@IP
- port: 22
- set putty/connections/SSH/tunnels
- source: local port:8888
- Destination: remote server: localhost:8889

Setting up our data files

- Create a text file called unnamed.txt.
  - `> nano unnamed.txt`
- Put the following into this file:

```
21,6.0
42,5.9
18,5.7*
21,NA
```

Setting up our data files

- Create another text file called daltons.txt
  - `> nano daltons.txt`
- Put the following into this file:

```
%% Data on the Dalton Brothers
Gratt,1861,1892
Bob,1892
1871,Emmet,1937
% Names, birth and death dates
```

read_csv using pandas

- Pandas is the Python Data Analysis Library

```
import pandas as pd
pd.read_csv("unnamed.txt")
```

- Import the pandas module as pd
- Read this with pd.read_csv()
  - What has happened to the first row?
    - now a header

	21	6.0
0	42	5.9
1	18	5.7*
2	21	NaN

- Read this again with
- header=None as an argument
  - What has happened now?

```
pd.read_csv("unnamed.txt", header=None)
```

	0	1
0	21	6.0
1	42	5.9
2	18	5.7*
3	21	NaN

Setting the column names (1)

- Let's read the data into a Python object this time and also set the column names.

```
person = pd.read_csv("unnamed.txt", header=None, names=('age','height'))
person
```

	age	height
0	21	6.0
1	42	5.9
2	18	5.7*
3	21	NaN

Setting the data type

- Let’s convert the height column into numeric values

```
person.height = pd.to_numeric(person.height)
person
```

```
ValueError: Unable to parse string "5.7*" at position 2
```

Setting the data type

- Let's convert the height column into numeric values

```
person.height = pd.to_numeric(person.height, errors='coerce')
person
```

	age	height
0	21	6.0
1	42	5.9
2	18	NaN
3	21	NaN

Structure of the Data Frame

- Let's check the structure
  - It's a data frame containing:
    - an age column of ints
    - a height columns of floats.

```
person.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 0 to 3
Data columns (total 2 columns):
age          4 non-null int64
height       2 non-null float64
dtypes: float64(1), int64(1)
memory usage: 96.0 bytes
```

PART 2

Dealing with unstructured text data

Dealing with unstructured data

Step 1 – Read the file

Step 2 – Select only lines containing data

Step 3 – Split each line into its separate fields

Step 4 – Standardise the rows

Step 5 – Transform to a data frame

Step 6 – Normalise or coerce to the correct type

```
%% Data on the Dalton Brothers
Gratt,1861,1892
Bob,1892
1871,Emmet,1937
% Names, birth and death dates
```

```
daltons
```

	name	birth	death
0	Gratt	1861.0	1892
1	Bob	NaN	1892
2	Emmet	1871.0	1937

Step 1 - readlines()

- readLines reads a file and returns a character vector, where each element is one line from the file

- Use readlines() to read this into Python

```
with open("daltons.txt") as f:
    txt = f.readlines()
```

```
txt
```

```
['%% Data on the Dalton Brothers\r\n',
 'Gratt,1861,1892\r\n',
 'Bob,1892\r\n',
 '1871,Emmet,1937\r\n',
 '% Names, birth and death dates\r\n']
```

Step 2 – Selecting lines only with data

- In our example a % at the beginning of the line indicates a comment. Let’s remove those lines.
- To do this we first need to learn about patterns and regular expressions

- Using a sample data set – iris

iris = pd.read_csv('https://github.com/pandas-dev/pandas/raw/master/pandas/tests/data/iris.csv')

```
iris = pd.read_csv('https://github.com/pandas-dev/pandas/raw/master/pandas/tests/data/iris.csv')
```

epcc

THE UNIVERSITY OF EDINBURGH

Step 2 – Selecting lines only with data

```
iris = pd.read_csv('https://github.com/pandas-dev/pandas/raw/master/pandas/tests/data/iris.csv')
```

```
iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
SepalLength    150 non-null float64
SepalWidth     150 non-null float64
PetalLength    150 non-null float64
PetalWidth     150 non-null float64
Name           150 non-null object
dtypes: float64(4), object(1)
memory usage: 5.9+ KB
```

```
names = iris.columns.tolist()  # Alternatively list(iris)
```

```
names
```

```
['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Name']
```

Using List Comprehension

- Python's list comprehension applies a function to each element in a list.

```python
numbers = [4,5,6]
[x*2 for x in numbers]
```

```
[8, 10, 12]
```

- A simple pattern match in Python

```python
'Petal' in 'PetalLength'
```

```
True
```

- Use list comprehension to match the pattern in every item in the list

```python
["Petal" in name for name in names]
```

```
[False, False, True, True, False]
```

- Put matches into new list

```python
[name for name in names if 'Petal' in name]
```

```
['PetalLength', 'PetalWidth']
```

Regular Expressions in Python

- As before, using regular expressions

```
import re
[name for name in names if re.search("Petal", name)]
```

```
['PetalLength', 'PetalWidth']
```

- ^ matches pattern at start
-

```
[name for name in names if re.search("^P", name)]
```

```
['PetalLength', 'PetalWidth']
```

- $ matches pattern at end

```
[name for name in names if re.search("th$", name)]
```

```
['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']
```

- [] character class, match characters enclosed in [ ]

```
[name for name in names if re.search("[g][t][h]", name)]
```

```
['SepalLength', 'PetalLength']
```

- For more see help(re) for full explanation

An aside on Simple string matching alternatives

```
[name for name in names if name.startswith("P")]
```

```
['PetalLength', 'PetalWidth']
```

```
[name.startswith("P") for name in names]
```

```
[False, False, True, True, False]
```

```
[name for name in names if name.endswith("th")]
```

```
['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']
```

```
[name for name in names if "gth" in name]
```

```
['SepalLength', 'PetalLength']
```

Subsetting and Logicals

- Logical and &

```
iris[(iris.Name == "Iris-versicolor") & (iris.PetalWidth >= 1.7)]
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
70	5.9	3.2	4.8	1.8	Iris-versicolor
77	6.7	3.0	5.0	1.7	Iris-versicolor

- Logical or |

```
iris[(iris.SepalLength == 4.3) | (iris.SepalLength == 7.9)]
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
13	4.3	3.0	1.1	0.1	Iris-setosa
131	7.9	3.8	6.4	2.0	Iris-virginica

- Logical not ~

```
iris[~(iris.SepalLength > 4.3)]
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
13	4.3	3.0	1.1	0.1	Iris-setosa

- Note difference in behaviour between == and =

Selecting rows and columns

- Pandas filter() command selects columns
- Can filter by regular expression

```
iris.filter(regex='^P').columns
```

```
Index([u'PetalLength', u'PetalWidth'], dtype='object')
```

- Select columns and rows at the same time

```
iris.filter(regex='^P')[~(iris.SepalLength > 4.3)]
```

	PetalLength	PetalWidth
13	1.1	0.1

Step 2 (cont) Selecting lines only with data

- Find lines starting with a % sign

```
[name for name in txt if re.search("^%", name)]
```

```
['%% Data on the Dalton Brothers\r\n', '% Names, birth and death dates\r\n']
```

- Remove those lines starting with a % sign

```
dat = [name for name in txt if not re.search("^%", name)]
dat
```

```
['Gratt,1861,1892\r\n', 'Bob,1892\r\n', '1871,Emmet,1937\r\n']
```

Step 3 – split lines into fields

- For each line, we now want to extract the content for each field
- We now need to know about splitting lines and learn about lists in Python

Python Lists

- In a Python a list can contain objects of different types, including others lists

```
L = [1,2, "three", [3,3]]
```

- [] retrieves and object from the list. Indexing starts at zero.

```
L[0:3]
[1, 2, 'three']
```

- Can select a range of values

```
L[0]
1
```

- Use – to count from end

```
L[-2]
'three'
```

- From second last to end`

```
L[-2:]
['three', [3, 3]]
```

epcc

THE UNIVERSITY OF EDINBURGH

split

- split() – splits a string into a list of substrings at the point indicated by the split pattern

```
x = "Split the words in a sentence\n"
x.split(" ")
```

```
['Split', 'the', 'words', 'in', 'a', 'sentence\n']
```

Step 3 (cont) split lines into fields

- Use split() to split each line into data chunks
- Use strip() to remove whitespace characters such as \n

```
x.strip().split(" ")
```

```
['Split', 'the', 'words', 'in', 'a', 'sentence']
```

- Do this for each line in dat

```
field_list = [ln.strip().split(",") for ln in dat]
field_list
```

```
[['Gratt', '1861', '1892'], ['Bob', '1892'], ['1871', 'Emmet', '1937']]
```

Step 4 – Standardise Rows

- Now we want to make sure each row has the same number of fields and in the same order
- Let's write a function to process each row.

User-defined functions in Python

```
def my_function (arg1, arg2, ... ):
    statements
    return(object)

code not in my_function
```

- Objects in the function are local to the function
- The object returned can be any data type
- Functions are stored as objects
- An explicit return statement is required
- : marks the start of the body of the function. The body must be indented, the end of the indentation marks the end of the function.

assign_fields function

- So let's write a function that takes the list representing each line, extracts the person's name, their birth and death dates and re-orders them accordingly.

- Let's call this function assign_fields and store it in a file called assign_fields.py

- Open a text file with: `nano assign_fields.py`

assign_fields function

```
import pandas as pd
def assign_fields(x):
    # x is a list of words from a line.

    # create a list to hold the extracted fields, initialised to 'NA' by default.
    out = ['NA'] * 3

    for word in x:
        # extract the name value (alphabetical) and insert in the first position.
        if word.isalpha():
            out[0] = word
        else:
            # extract birth date (if any)
            # based on knowledge that all Dalton brothers were born before 1890
            # and died after 1890
            if (int(word) < 1890):
                out[1] = word
            elif (int(word) > 1890):
                out[2] = word
    # Returns a list format: [name, born, died]
    return out
```

Step 4 (cont)

- Save the assign_fields.py file and restart ipython
- Read the file in again after re-starting ipython

```
import pandas as pd
import re
with open("daltons.txt") as f:
    txt = f.readlines()
dat = [name for name in txt if not re.search("^%", name)]
field_list = [ln.strip().split(",") for ln in dat]
```

- Let's run the assign fields function on the elements of field_list

```
from assign_fields import assign_fields
standard_fields = [assign_fields(ln) for ln in field_list]
standard_fields
```

```
[['Gratt', '1861', '1892'], ['Bob', 'NA', '1892'], ['Emmet', '1871', '1937']]
```

Step 5 – Transform to a data frame

- Let's convert list of standardised rows into a data frame.

```
daltons = pd.DataFrame(standard_fields)
```

```
daltons
```

	0	1	2
0	Gratt	1861	1892
1	Bob	NA	1892
2	Emmet	1871	1937

```
daltons = pd.DataFrame(standard_fields, columns=['name','birth','death'])
```

```
daltons
```

	name	birth	death
0	Gratt	1861	1892
1	Bob	NA	1892
2	Emmet	1871	1937

Step 6 – Normalise & coerce to correct type

- Now need to coerce our columns to the correct types eg. numerics, characters, categories, …. In this case birth and death. need to be numerics

```
daltons.birth = pd.to_numeric(daltons.birth, errors='coerce')
```

```
daltons.death =  pd.to_numeric(daltons.death, errors='coerce')
```

```
daltons
```

	name	birth	death
0	Gratt	1861	1892
1	Bob	NA	1892
2	Emmet	1871	1937

Step 6 – Normalise & coerce to correct type

- The birth column contains floats instead of integers because you can't mix int and NaN data types in pandas.

```
daltons.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
name      3 non-null object
birth     2 non-null float64
death     3 non-null int64
dtypes: float64(1), int64(1), object(1)
memory usage: 144.0+ bytes
```

Repeatability

- Storing the instructions in a file along **with comments** enables repeatability
- Ipython notebooks allow nicely formatted comments, code, and output to be mixed.

```
import pandas as pd
import re
with open("daltons.txt") as f:
    txt = f.readlines()
dat = [name for name in txt if not re.search("^%", name)]
field_list = [ln.strip().split(",") for ln in dat]
from assign_fields import assign_fields
standard_fields = [assign_fields(ln) for ln in field_list]
colnames = ['name', 'birth', 'death']
daltons = pd.DataFrame(standard_fields, columns=colnames)
daltons.birth = pd.to_numeric(daltons.birth, errors='coerce')
daltons.death = pd.to_numeric(daltons.death)
print("Daltons")
print(daltons)
print('\nInfo')
print(daltons.info())
```

epcc

Fixing character vectors – re.sub

- sub() - replaces a pattern

```
import re
string = "Replace the spaces in this text"
re.sub(" ", "-", string)
```

```
'Replace-the-spaces-in-this-text'
```

- Can choose how many occurrences to replace

```
string = "Replace first space in this text"
re.sub(" ", "-", string, count=1)
```

```
'Replace-first space in this text'
```

Fixing character vectors – re.sub

- Apply a substitution across every string in a list

```
names
```

```
['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Name']
```

```
[re.sub("e", '-', name) for name in names]
```

```
['S-palL-ngth', 'S-palWidth', 'P-talL-ngth', 'P-talWidth', 'Nam-']
```

Parallel processing in Python

- Can use the ‘multiprocessing’ module to run code across more than one processor
- Serial version:

```
standard_fields = [assign_fields(ln) for ln in field_list]
```

- Parallel version:

```
import multiprocessing
from multiprocessing import Pool

try:
    cpus = multiprocessing.cpu_count()
except NotImplementedError:
    cpus = 2   # arbitrary default

pool = Pool(processes=cpus)
pool.map(assign_fields, field_list)
```

```
[['Gratt', '1861', '1892'], ['Bob', 'NA', '1892'], ['Emmet', '1871', '1937']]
```