Genome data analysis in Python

A brief tutorial on the use of *jupyter notebooks* and the python data analysis library *pandas* for genomic data analysis.

Workshop on Population and Speciation Genomics, Český Krumlov, January 2018.
By Hannes Svardal (hs669@cam.ac.uk)

This is a jupyter notebook running a Python 2 kernel. The Jupyter Notebook App (formerly IPython Notebook) is an application running inside the browser. Jupyter notebooks can run different kernels: Python 2/3, R, Julia, bash, ...

Further resources about jupyter notebooks can be found here:
- https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/
- https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook

Jupyter notebooks can run locally or on a server. You access them in your browser.

To start the jupyter server - Log into your amazon cloud instance: `ssh wpsg@my-ip-here.compute-1.amazonaws.com` (replace my-ip-here with your instance's address) - Navigate into the tutorial directory: `cd ~/workshop_materials/03a_jupyter_notebooks/` - Start the notebook server: `jupyter notebook --no-browser --port=7000 --ip=0.0.0.0` - In your local browser, navigate to the web address: http://my-ip-here.compute-1.amazonaws.com:7000 - On the web page, type in the password *evomics2018*

Now you should have this notebook in front of you. - At the top of the webpage, the notebook environment has a **header** and a **toolbar**, which can be used to change settings, formatting, and interrupt or restart the kernel that interprets the notebook cells. - The body of the notebook is built up of cells of two has two major types: markdown cells and code cells. You can set the type for each cell either using the toolbar or with keyboard commands. The right-most button in the toolbar shows all keyboard shortcuts. - **Markdown cells** (this cell and all above) contain text that can be formatted using html-like syntax http://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Working%20With%20Markdown%20Cells.html Double-klick into a markdown cell (like this one) to get into *edit mode* - **Code cells** contain computer code (in our case written in python 2). Code cells have an **intput field** in which you type code. Cells are evaluated by pressing *shift + return* with the cursor being in the cell. This produces an **output field** with the result of the evaluation that would be returned to std-out in a normal python (or R) session. Below are a few examples of input cells and the output. Note that by default only the result of the last operation will be output, and that only if it is not asigned to a variable, but all lines will be evaluated.

Here are some very basic operations. Evaluate the cells below and check the results.

```
# This is a code cell.
# Evaluate it by moving the cursor in the cell an pressing <shift + return>.
1+1
```

```
# This is anoter code cell.
# There is no output because the last operation is assigned to a variable.
# However, the operations are performed and c is now assigned a value.
# Evaluate this cell!
a = 5
b = 3
c = a * b
```

```
# The variables should now be assigned. Evaluate.
print 'a is', a
print 'b is', b
print 'c is a*b, which is', c
```

Try to create more cells using either the "plus" button in the toolbar above or the keyboard combination (Ctrl + M) + B (First Ctrl + M together, then B). Try to define variables and do calculations in these cells.

Python basics

This is very basic python stuff. People who are farmiliar with python can skip this part.

loading modules

```
# Load some packages that we will need below
# by evaluating this cell
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
# Refer to objects (e.g. functions) from these packages by with module.object
print np.sqrt(10)
print np.pi
```

lists, list comprehension, and numpy arrays

Lists are a very basic data type in python. Elements can be accessed by index. **Attention:** Different from R, Python data structures are generally zero indexed. The first element has index 0.

```
list0 = [1, 2, 3, 4]
print list0
print list0[1]
```

```
# the last element
print list0[-1]
# elements 2 to 3
print list0[1:3]
```

List comprehensions are a very useful feature in Python. It is an in-line way of iteratin through a list.

```
# This is the syntax of a so-called list comprehension.
# A very useful feature to create a new list by iterating through other list(s).
squares = [i*i for i in list0]
print squares

# Doing this in conventional syntax would be more verbose:
squares = []
for i in list0:
    squares.append(i*i)
print squares
```

A numpy array is a vector-like object of arbitrary dimensions. Operations on numpy arrays are generally faster than iterating through lists.

```
array0 = np.array(list0)
print array0
```

```
# Operations on an array are usually element-wise.
# Square the array elements.
print array0**2
```

```
# Instantiate array 0 .. 19
x = np.arange(20)
print x
```

```
# 2D array
array2d = np.array(
                [[1,2,3],
                [4,5,6]]
                )
print array2d
print array2d*2
print 'number of rows and columns:', array2d.shape
```

anonymous (lambda) function

A lambda function is a function that is not bound to a name at creation time.

```
# This is a regular function
def square(x):
    """
    This is a regular function
    definition. Defined in evomics2018.

    This function takes a number (int or float)
    and returns the square of it.

    """
    return x*x

print 'This is a regular function:', square
print square(5)
```

```
# A lambda function is defined in-line; here it is bound to a name,
# but that is not necessary
square2 = lambda x: x*x
print 'This is an anonymous function:', square2
print square2(5)
```

The advantage of an anonymous function is that you can define it on the go.

```
#For this you must pre-define the function 'square'
map(square, list0)
```

```
#Here the same but defining the function on the go.
#This is very useful when we apply functions to data frames below
map(lambda x:x*x, list0)
```

Ipython

Ipython is an interactive interface for python. Jupyter notebooks that run a python kernel use Ipython. It basically is a wrapper around python that adds some useful features and commands. A tutorial can be found here: https://ipython.org/ipython-doc/3/interactive/tutorial.html

The four most helpful commands (type in a code cell and evaluate)

command	description
?	Introduction and overview of IPython's features.
%quickref	Quick reference.
help	Python's own help system.
object?	Details about 'object', use 'object??' for extra details.

```
# Evaluate this to get the documentation of the function **map** as a popup below.

map?
```

```
# Get the docstring of your own function defined above.
square?
```

Ipython magic

IPython *magic commands*, are tools that simplify various tasks. They are prefixed by the % character. Magic commands come in two flavors: line magics, which are denoted by a single % prefix and operate on a single line of input, and cell magics, which are denoted by a double %% prefix and operate on multiple lines of input.

Examples

```
# Time a command with %timeit
%timeit 1+1
```

```
%%timeit
#Time a cell operation
x = range(10000)
max(x)
```

```
# This is a very useful magic that allows us to create plots inside the jupyter no
tebook
# EVALUATE THIS CELL!!!
%matplotlib inline
```

```
# Make a basic plot
plt.plot(np.random.randn(10))
```

running shell commands

You can use ipython magic to run a command using the system shell and return the output. Simply prepend a command with "!" or start a cell with %%bash for a multi line command.

```
!ls
```

```
files = !ls
print files
```

```
!msmc2
```

```
%%bash
cd ~
ls
echo ----------------
echo $PATH
```

Pandas

https://pandas.pydata.org/ *pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

List of tutorials
- https://pandas.pydata.org/pandas-docs/stable/tutorials.html

10 minutes quick start guide - https://pandas.pydata.org/pandas-docs/stable/10min.html

Installation (just like other python modules)
`pip install pandas`

```
import pandas as pd
```

The two most important data structures in pandas are **Series** and **DataFrames**.

pandas Series

A Series is a 1D-array-like object where each element has an index.

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

In this case the index of s are integers 0,1,... but it could be strings, floats, ...

```
s
```

```
# for operations like addtions, elements are matched by index
s + s
```

```
# elements are matched by index, even if they arein a different order)
s1 = pd.Series([1, 3, 5, np.nan, 6, 8],index=['F','E','D','C','B','A'])
s2 = pd.Series([1, 3, 5, np.nan, 6, 8],
               index=['A','B','C','D','E','F'])
print s1
print '-------------'
print s2
```

```
# what do you expect the result of this to be?
s1 + s2
```

Do you understand the result above?

```
# Access an element using the index
s.loc[2]
```

```
# Access an element using the psition
s.iloc[2]
```

In the above case, the two are trivially the same, but for s1 and s2 it is very different. Try both ways of accessing elements on s1 and s2.

pandas DataFrame

Pandas data frames are similar to R data frames. A DataFrame is a 2D-array-like object where each element has a row index and a column index. The row index is called 'index', the column index is called 'columns'.

In the following, create a simple data frame and inspect its elements. Try to modify the code in this section.

```
df = pd.DataFrame([[1,2,3],
                   [4,5,6]],
                  index=[100,200],
                  columns=['A','B','C'])
```

```
df
```

```
df.index
```

```
df.columns
```

```
df.index.values
```

```
# access by position
df.iloc[1, 2]
```

```
# access an element by index
df.loc[200, 'C']
```

```
#access a row
df.loc[200,:]
```

```
#access a column
df.loc[:,'C']
```

```
#logical indexing
df.loc[df['A']>2,]
```

```
df.loc[:, df.loc[200]>4]
```

```
df + df
```

```
# mean of rows
print df.mean()
```

```
# mean of columns
print df.mean(axis=1)
```

```
# If a function on a data frame returns a 1D object, the results is a pd.Series
print type(df.loc[100,:])
```

Apply operations

Data Frames have many handy methods built in. For applying functions, grouping elements, plotting. We will see several of them below. Here The simples apply opperations.

```
df.apply(square)
```

```
# apply along rows (column-wise)
df.apply(np.sum, axis=0)
```

```
# apply along columns (row-wise)
df.apply(np.sum, axis=1)
```

For the above there exists a shortcut. You can directly use df.sum(axis=...)

```
# apply element-wise
df.applymap(lambda i:'ABCDEFGHIJKLMN'[i])
```

What does the above code cell do? Play around with it to understand what is happening.

Tipp: It helps to look at each of the part in turn.

```
'ABCDEFGHIJKLMN'[2]
```

```
df.applymap?
```

Working with SNP calls

Here it gets interesting. How can we use pandas to analyse genome data. Note that some of the below is a bit simplified and you would do things slightly differently in a production pipeline.

The combination of jupyter notebooks and pandas is great for quick exploration of data. But using ipython parallel one can also handle demaning analyses.

We will be using a cichlid fish VCF file with bi-allelic SNP calls.

```
#check which files are in the folder
!ls
```

```
vcf_fn = 'cichlid_data_outgroup.vcf.gz'
```

```
# Use bash magic to take a look at the file contents
```

```
%%bash
gzip -dc "cichlid_data_outgroup.vcf.gz" | head -n 18
```

Parse in the header line of the file

```
## parse the header line starting with "#CHROM"
import gzip
with gzip.open(vcf_fn) as f:
    for line in f:
        if line[:6] == '#CHROM':
            vcf_header = line.strip().split('\t')
            vcf_header[0] = vcf_header[0][1:]
            break
```

```
#The vcf
print vcf_header[:20]
```

```
# Read a tsv, csv, into a data frame
pd.read_csv?
```

```
# Here we read in the vcf file, which basically is tab-separated value file.
gen_df = pd.read_csv(vcf_fn,
                     sep='\t',
                     comment='#',
                     header=None,
                     names=vcf_header,
                     index_col=['CHROM','POS'])
```

```
gen_df.head()
```

```
# Convert the GT=string into data frames with integer for first and second haploty
pe
first_haplotype = gen_df.iloc[:, 9:].applymap(lambda s: int(s.split('|')[0]))
second_haplotype = gen_df.iloc[:, 9:].applymap(lambda s: int(s.split('|')[1]))
```

```
first_haplotype.head()
```

```
# Create a second level in the column index that specifies the haplotype
first_haplotype.columns = pd.MultiIndex.from_product([first_haplotype.columns, [0]
])
second_haplotype.columns = pd.MultiIndex.from_product([second_haplotype.columns, [
1]])
```

```
first_haplotype.head()
```

```
# Creat a haplotype dataframe with all the data
hap_df = pd.concat([first_haplotype, second_haplotype], axis=1).sort_index(axis=1)
```

```
hap_df.head()
```

```
import subprocess

def read_hap_df(vcf_fn, chrom=None, start=None, end=None, samples=None, **kwa):
    """
    A slightly more advanced vcf parser.
    Reads in haplotypes from a vcf file.
    Basically does the same as done in the
    cells above, but allows the used to
    specify the range of the genome that
    should be read in. Also allows to specify
    which samples should be used.

    Parameters:
    vcf_fn : file path of the VCF to be read
    chrom : specify which chromosome (or scaffold)
            to read from the file
            (only works on bgzipped, tabix-indexed files)
            default ... read whole file
    start: specify the start nucleotide position
            (only works if chrom given on bgzipped,
            tabix-indexed files); default=1
    end: specify the ebd nucleotide position
            (only works if chrom given on bgzipped,
            tabix-indexed files); default=chrom_end
    samples: list of sample names to read;
              default ... all samples

    returns:
    Pandas dataframe of index (chrom, pos)
    and columns (sample, haplotype). Values
    are 0 for first and 1 for second allele.

    """
    # parse header
    with gzip.open(vcf_fn) as f:
        for line in f:
            if line[:6] == '#CHROM':
                vcf_header = line.strip().split('\t')
                vcf_header[0] = vcf_header[0][1:]
                break

    # determine genomic region to read in
    if chrom is not None:
        assert vcf_fn[-3:] == ".gz", "Only supply chrom if vcf is bgzipped and tab
ix indexed"
```

```
        region = chrom
        if end is not None and start is None:
            start = 0
        if start is not None:
            region += ':' + str(start)
            if end is not None:
                region += '-' + str(end)
    else:
        region = None

    # If no specific samples given, use all samples in the VCF
    if samples is None:
        samples = vcf_header[9:]

    # Either use regional input or input whole VCF
    if region is None:
        stdin = vcf_fn
    else:
        tabix_stream = subprocess.Popen(['tabix', vcf_fn, region],
                                        stdout=subprocess.PIPE,
                                        stderr=subprocess.PIPE)
        stdin = tabix_stream.stdout


    gen_df = pd.read_csv(stdin,
                         sep='\t',
                         comment='#',
                         names=vcf_header,
                         usecols=['CHROM','POS']+samples,
                         index_col=['CHROM','POS'], **kwa)
    first_haplotype = gen_df.applymap(lambda s: int(s.split('|')[0]))
    second_haplotype = gen_df.applymap(lambda s: int(s.split('|')[1]))

    first_haplotype.columns = pd.MultiIndex.from_product([first_haplotype.columns,
 [0]])
    second_haplotype.columns = pd.MultiIndex.from_product([second_haplotype.column
s, [1]])

    hap_df = pd.concat([first_haplotype, second_haplotype], axis=1).sort_index(axi
s=1)

    return hap_df
```

```
small_hap_df = read_hap_df(vcf_fn,
                     chrom='Contig237',
                     start=3000,
                     end=5000,
                     samples=['VirSWA1', 'VirSWA2', 'VirSWA3',
                              'VirSWA4', 'VirSWA5', 'VirSWA6'])
```

```
small_hap_df
```

indexing

```
# access a cell by index
# gt_df.loc['row index', 'column index']
print 'Get the second haplotye of individual',
print 'VirSWA6 for position 3203 on Contig237:',
print hap_df.loc[('Contig237', 3203), ('VirSWA6', 1)]
```

```
hap_df.loc['Contig237'].loc[ 3000:3400, 'VirSWA6']
```

investigate haplotype data frame

```
# get the number of SNPs and number of samples
hap_df.shape
```

```
# get the name of sequenced in this data frame
hap_df.index.droplevel(1).unique()
```

load sample metadata

```
meta_df = pd.read_csv('cichlid_sample_metadata.csv', index_col=0)
```

```
meta_df.head()
```

Group a data frame using groupby

Groupby groups a data frame into sub data frames. You can group on values in a specific column (or row) or by applying a function to column or row indices. This is very handy.

```
# group individuals by sampling location
place_groups = meta_df.groupby('place')
```

```
# iterate through groups
for group_name, group_df in place_groups:
    print group_name
    print group_df
    print '----------------------------------------'
```

You can apply functions to the groups. These are applied to each group data frame. Pandas will try to give a series or data frame as result where the index contains the group names.

```
place_groups.apply(len)
```

```
# here individuals are grouped by the columns genus and species
meta_df.groupby(['genus', 'species']).apply(len)
```

```
# This is a Series with the same index as meta_df.
# The values are True/False depending on whether the species name is virginalis.
is_virginalis = (meta_df['species']=='virginalis')
```

What length do you expect is_virginalis to be? How many True and False entries?

The above can be used for logical indexing.

```
# Logical indexing. Select viriginalis samples only.
meta_df[is_virginalis].groupby('place').apply(len)
```

apply operations

Apply functions to our haplotype data frame.

```
allele_frequency = hap_df.mean(axis=1)
```

```
allele_frequency.head()
```

Plot the site frequency spectrum.

```
allele_frequency.hist(bins=20)
```

What is on the x and y axis? Does this spectrum look neutral to you?

restrict to samples of species Copadichromis virginalis

```
virginalis_samples = meta_df[meta_df['species']=='virginalis'].index.values
```

```
# only virginalis samples
hap_df_vir = hap_df.loc[:, list(virginalis_samples)]
# the list conversion above is not needed in newer pandas versions
```

```
af_virginalis = hap_df_vir.mean(axis=1)
```

```
af_virginalis_variable = af_virginalis[(af_virginalis>0)&(af_virginalis<1)]
```

What does the above line of code do?

```
# restrict haplotype data frame to alleles variable in virginalis
hap_df_vir = hap_df_vir.loc[af_virginalis_variable.index, :]
# or equivalently
#hap_df_vir = hap_df_vir[(af_virginalis>0)&(af_virginalis<1)]
```

Check how the number of SNPs was reduced by removing non-variable sites

```
print af_virginalis.shape
print af_virginalis_variable.shape
```

```
af_virginalis_variable.hist(bins=20)
```

remove low frequency variants

```
n_alleles = hap_df.notnull().sum(axis=1)
```

```
min_allele_count = 4
hap_min_ac = hap_df[(allele_count >= min_allele_count) & (allele_count <= n_allele
s - min_allele_count)]
```

```
print hap_df.shape
print hap_min_ac.shape
```

```
(hap_min_ac.mean(axis=1)).hist(bins=20)
```

grouping by sample

```
# grouping can be done by a dictionary that is applied to index or columns
sample_groups = {'VirMAL1':'Malombe',
                 'VirMAL2':'Malombe',
                 'VirMAL3':'Malombe',
                 'VirSWA1':'South West Arm',
                 'VirSWA2':'South West Arm',
                 'VirSWA3':'South West Arm'}
```

```
sample_groups0 = hap_df_vir.groupby(sample_groups, axis=1, level=0)
```

```
sample_groups0.mean()
```

```
# group using a function
def get_location(sample_id):
    return meta_df.loc[sample_id, 'place']

location_groups = hap_df_vir.groupby(get_location, axis=1, level=0)
```

```
# equivalent to above but using a lambda function
location_groups = hap_df_vir.groupby(lambda id: meta_df.loc[id, 'place'], axis=1,
level=0)
```

Calculate the allele frequency for each local population.

```
population_af = location_groups.mean()
```

```
fig = plt.figure(figsize=(16,10))
ax = plt.gca()
axes = population_af.hist(bins=20, ax=ax)
```

Calculate nucleotide diversity π and divergence d_{xy}

```
# pi = 2p(1-p)
# dxy = pq
```

```
# apply a function in rolling windows of 100 SNPs
window_size = 100
rolling_window_df = population_af.loc['Contig237', 'Malombe'].rolling(window_size,


                                                                      center=True,
 axis=0)
pi_rolling = rolling_window_df.apply(lambda s:(2*s*(1-s)).mean())
```

```
pi_rolling.plot(style='.')
```

```
def get_dxy(af):
    """
    Get dxy between Malombe and
    South East Arm
    """
    dxy = af['Malombe']*(1-af['South East Arm']) + (1-af['Malombe'])*af['South Eas
t Arm']
    return dxy.mean()

# apply function in non-overlapping 100 bp windows
window_size = 100
dxy = population_af.loc['Contig237'].groupby(lambda ix: ix // window_size).apply(g
et_dxy)
```

```
dxy.plot(style='.')
```

Vary the parameters of the above functions. Try to plot pi for the different chromosomes and the different populations.

A more general function to calculate dxy across muliple populations

```
def get_divergence(af):
    """
    Takes a allele frequency df
    returns nucleotide diversity (diagonal)
    and dxy (off-diagonal).

    Attention! The estimator for pi
    is biased because it does not take
    resampling of the same allele into account.
    For small populations pi will be downward biased.
    """

    # This looks complicated. If basically
    # uses tensor muliplication to efficiently
    # calculate all pairwise comparisons.
    divergence = np.einsum('ij,ik->jk',af, 1-af) \
            + np.einsum('ij,ik->jk',1-af, af)
    # the above results in a numpy array
    # put it into a data frame
    divergence = pd.DataFrame(divergence,
                              index=af.columns,
                             columns=af.columns)
    return divergence
```

```
get_divergence(population_af)
```

```
individual_af = hap_df_vir.groupby(axis=1, level=0).mean()
```

```
individual_dxy = get_divergence(individual_af)
```

```
#Be aware of the biased single-individual pi estimated on the diagonal.
individual_dxy
```

The above could be used to construct a neighbour-joining tree.

Ipython parallel

Ipython parallel is very handy to use multiple local or remote cores to do calculations. It is surprisingly easy to set up, even on a compute cluster. (However, the ipyparallel package is not installed for python 2.7 on this amazon cloud instance and I realised it too late to fix it.)

Here are more resource for the parallel setup:
- https://ipython.org/ipython-doc/3/parallel/index.html - https://ipython.org/ipython-doc/3/parallel/parallel_process.html

A minimal example (that would work for me):

In a terminal execute `ipcluster start -n 4` to start a ipython cluster with 4 engines

```
from ipyparallel import Client
```

```
rc = Client(profile="default")
lv = rc.load_balanced_view()
```

```
map_obj = lv.map_async(lambda x: x*x, range(20))
```

The above is the parallel equivalent of `map(lambda x: x*x, range(20))` but using the 4 engines started above.

```
# retrieve the result
result = map_obj.result()
```

This is very useful to quickly analyse multiple chromosomes or chunks of chromosomes in parallel.