Course Analytics - Wrangling FutureLearn Data With Python and R

Tony Hirst

Course Analytics - Wrangling FutureLearn Data With Python and R

Tony Hirst

This book is for sale at
http://leanpub.com/courseanalytics-wranglingfuturelearndatawithpythonandr

This version was published on 2016-04-30

Disclaimer

Contents

Introduction

This book contains recipes for wrangling FutureLearn course data using Python and R. It is presented as a work in progress, published via the Leanpub publishing platform.

As with all learning environments operated by schools, colleges, universities and other learning providers, the FutureLearn platfrom collects data about a wide range of learner interactions with the platfrom. The burgeoning field of learning analytics aims to harness this data in order to provide a range of tools for monitoring and reporting on learning activity, often with the aim of predicting learning behaviour, identifying at risk learners, or personalising the the offering provided to invididual learners.

My own preference is to see the data collected as a reflection of the performance of the course materials themselves. That is, I see the data as being useful to *course authors* and *course designers* who want to better understand how learners interact with and pass through the materials *en masse* so that the performance of the materials themselves, rather than that of individal learners, can be assessed.

The origins of this book lay in the Open University / FutureLearn course Learn to code for Data analysis[1], first presented in Autumn 2015. The course taught learners how to use the Python programming language, and more specifically the `pandas` data analysis Python package, to perform a range of data analysis and visualisation tasks. The data used in the course included freely available opendata from the World Bank and the United Nations. The programming environment used in the course was a browser based environment known as Jupyter notebooks. The notebook style programming interface differs from traditional programming environments (though it will be familiar to users of mathematical computing enviornments such as Mathematica), where code is written in often austere and intimidating text editor environment. Instead, the notebook environment blends editiable text areas with executable code cells whose output can be displayed inline in the notebook document once the code is run. The notebooks themselves are edited via a web browser, and served in the background either by a desktop application or an online service such as SageMathCloud.

As the course was being delivered, the course team was provided with occasional data downloads detailing some of the interactions occurring on the platform: enrolment and unenrolment figures, data about comment or quiz activity, and data describing when learners first accessed, or last marked as complete, each step of the course. It seemed to me in-keeping with a philosophy of my original faculty at the Open University, the Faculty of Technology, as was, that we should use the content and context of the course reflexively in the analysis of the data produced from the course.

I had also pushed for using the notebooks as the production environment of the course - the notebooks are authored in the markdown text format which is the format used to create FutureLearn courses.

[1]https://www.futurelearn.com/courses/learn-to-code

As it was, our course materials went from markdown, into Microsoft Word for editing, then back to markdown for entry into the FutureLearn platform. And then we had to work out how fix the errors. This book, in part, is an attempt to reclaim the use of the notebooks as an authoring environment. The Leanpub platform constructs books from one or more markdown manuscript files, which I have created using the notebooks and, more directly, by editing files direclty in Github, which acts the "source code repository" for the manuscript files used to create the book. The RStudio environment too can publish to markdown (I technique I used to create the Wrangling F1 Data With R[2] Leanpub book) and I fully expect to use that route when it comes to adding in some R recipes.

And so to the book itself. The original Python/`pandas`recipes themselves started out life as recipes that used only the techniques that had been taught in the *Learn to Code for Data Analysis* course, although they were updated and slightly expanded for the "first commit" of the book. In particular, I added in some interactive features that allow interactive widgets to be embedded in the notebook to allow you to manipular the data directly through drop down selection lists and numerical sliders selectors.

The book as presented is a *static* beast, rendered in HTML and incapable of executing the code fragments. The book is also skeletal and unillustrated. Whilst FutureLearn data *can* be analysed and the results of the data *can* be shared publicly, the data is not openly licensed and permission is required before any analyses can first be shared. And I don't do permission. The ethics around what I consider fair game for analysis, and what I don't, are also my own.â'¶

But that is not to say the code cannot be run. The book is a rendering of hybrid text and code notebooks (original *.ipynb* files for the Jupyter notebooks, *Rmd* for content authored in RStudio), the original versions of which can be found in the Github repository where all the source materials for this book can be found: psychemedia/futurelearnStatsSketches[3].

There's one final innovation(?), invention(?), experiment(?) I'm pursuing here, and that focuses on sustainability. The majority of effort associated with producing this work was done in my own time, on the one day of the working week (as well as evenings and weekends) that my 0.8FTE Open University contract does not pay me for. It's my own learning time, my own play time, though it often feeds back into my OU activities, and whilst I'd originally hoped to spend the the time engaged in profitable consulting activities, it hasn't worked out like that. So here's the rub: this book hasn't been funded by a research project, indeed hasn't really been funded at all. It's not proper research and I make no real claims of it (it's the output of my playing with and learning how to use a range data analysis tools; it *is* my *learning diary* and a record of my learning journey). But it is something I'm happy to share, and something I may spend time on exploring further, as well as perhaps trekking around the country to talk about - if it can cover its own expenses.

So if the contents of this work save you time, make a donation. Leanpub supports a flexible pricing policy and doesn't take a huge cut (though the tax man does take his fair share). The price of this book starts at *free* and goes up from there. You can pay as much or as little as you want, and pay as many times as you want. Once you've got a copy of the book, you get all future updates to it *for*

[2]https://leanpub.com/wranglingf1datawithr/

[3]https://github.com/psychemedia/futurelearnStatsSketches/

free, unless you want to pay again. If this was an article you wanted to read in an academic journal, and your library didn't have a copy, it would cost you what? Up to $50 dollars or so? If this was a training session provided from a technical trainer, how much would the quarter-day, half-day or full-day seat cost?

But WTF?, you may think, *isn't this moonlighting, isn't this double pay on top of your OU job?*. So here's the deal: any monies raised from the sale of this book will go solely on covering expenses that *I* deem relevant to exploring the ideas contained in this work (and that's the caveat: *I, me, my opinion*). This may include things like web hosting (so I can host online interactive versions of the code), or paying Github for a paid for plan, or traveling to relevant events. (Sustainability cuts several ways: I'm feeding Leanpub from Github, so if Github goes away, I have more work to do in finding an alternative publication route; such as using Dropbox; which also has a paid plan (although income from my F1 data book currently pays for that…) But so you can see any the money goes, I'll publish information about any and all receipts and expenses.

Tony Hirst, April 2016

Part 1 - Jupyter Notebook Recipes (Python / pandas)

FutureLearn Stats Recipes - Jupyter Notebooks

Tony Hirst, The Open University / @psychemedia / OUseful.info blog

This work has been independently produced as a recreational data activity. The author is not affiliated with FutureLearn Ltd or any Open University learning analytics projects.

This notebook contains a few recipes, sketches and doodles around FutureLearn course stats made available to partners. It has been tested using the `psychemedia/ou-tm351-pystack` Docker container on Docker Hub but should work with other Python 3 notebook kernels with `pandas` and `seaborn` installed. (*The notebook has not been tested with Python 2.7 but I think it should probably work, though you might need to add* `from __future__ import division`.)

Note that the notebooks may not be rendered or not work properly in Internet Explorer: try Chrome, Firefox or Safari instead.

This notebook will analyse data from four data files made available by FutureLearn (and updated on a daily basis) for each course:

- *COURSE_enrolments.csv*: data about role, enrolments, unenrolments and full participation dates;
- *COURSE_step-activity.csv*: record of when each step was first visited by each user and if/when it was completed;
- *COURSE_comments.csv*: record of each comment along with the time it was posted, who it was posted by, how many likes it has, and whether it was part of a thread;
- *COURSE_question-response.csv*: each question attempt by each learner is recorded, along with the answer submitted and whether it was the correct answer.

To analyse the data, click in this area and then press the play button in the toolbar (or hit *shift-Return*) to run each cell in turn.

You can tell when a code cell has been run because the `In[]` indicator will be populated by a number showing the order in which the cells were run.

A `[*]` indicator shows the current cell (or preceding cells if multiple cells are run from the `Cell` menu) is currently being executed.

As well as interactively executable code cells, Jupyter notebooks also support a range of interactive widgets that can be used to create interactive dashboard like displays.

Once you have run the notebook, you can save an HTML version of it from the `File` menu.

Initialisation

We need to load in some libraries and utility functions that we will use to analyse and manipulate the data.

```
%matplotlib inline

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from matplotlib.patches import Rectangle
import random
from ipywidgets import widgets, interact
from datetime import date, timedelta

def offsetDays(date_str,offset):
    ''' Return a date a specified number of days before (offset<0) or after (offset>0) a p\
articular date'''
    if offset<0:
        return (pd.to_datetime(date_str)-timedelta(days=abs(offset))).date().strftime('%Y-\
%m-%d')
    return (pd.to_datetime(date_str)+timedelta(days=offset)).date().strftime('%Y-%m-%d')
```

If you get any warnings about packages not being found, you will need to install them. The following code cell gives examples of how to install the `seaborn` package. Which package manager you use depends on how your system is set up. If you use the wrong one, you may get an error message but it shouldn't break anything. Uncomment the appropriate line (remove the #), run the code cell, and and then rerun the code cell above to see if you can now import the package.

```
##The ! says: run this command on your computer command line
#!pip install seaborn
#!pip3 install seaborn
#!conda install seaborn
```

USER SETTINGS

The notebook is self-contained and should be able to generate reports from standard FutureLearn data files.

There are however a few settings that are required.

Loading the Data

Download and unzip the data files into a folder contained in the same directory as this notebook file. Also make note of the course name stub used as part of each filename.

FutureLearn provide a CSV each day, generated at midnight, so you may want to grab the data frequently during the course, or just analyse it all at the end of the course.

Note that it make take some time to load the data in. Also note that the data is held in memory so a low powered machine may struggle for large datasets.

For example, if your filenames are of the form `learn-to-code-1_enrolments.csv` in the folder `learn-to-code-1` use the settings:

```
#CHANGE THESE SETTINGS FOR YOUR OWN DATA FILE
#The settings point to files in a location reataive to the notebook: PATH/STUB_comments.cs\
v etc
#For example: learn-to-code-1/learn-to-code-1_comments.csv
PATH='learn-to-code-1'
STUB='learn-to-code-1'
```

Notable Dates and Steps

Some reports can be highlighted around notable dates (such as the course registration opening date, or start date) or steps (for example, steps where exercises or activities are defined or social steps with particular calls for comment).

Identify any notable dates or steps here in order for these to be highlighted automatically.

```
#CHANGE THESE SETTINGS FOR YOUR OWN DATA FILE
#To help generate useful reports, identify key dates for your course in the form: YYYY-MM-\
DD
COURSE_OPEN_REGISTRATION='2015-08-11'
COURSE_START_DATE='2015-10-26'
COURSE_END_DATE='2015-11-23'

#We can also define relative timestamps, for example, five days prior to the start date
COURSE_PRE_START_MAILINGDATE=offsetDays(COURSE_START_DATE,-5)


#Add notable dates to the SPECIAL_DATES list. If there none, set: SPECIAL_DATES=[]
SPECIAL_DATES=[COURSE_OPEN_REGISTRATION,
               COURSE_START_DATE,
               COURSE_END_DATE,
               COURSE_PRE_START_MAILINGDATE ]

NUMBER_OF_STUDY_WEEKS=4

#The SESSION_GAP specifies the minumim time period (in minutes) that distinguishes between\
 two study sessions
SESSION_GAP=65


#CHANGE THESE SETTINGS FOR YOUR OWN DATA FILE
#If you have any exercise or activity steps, they can be highlighted in some step reports
#The steps are defined in the form WSS, where W is the week number and SS the step number \
(with leading 0)
#For example, week 1 step 2 is written: 102; week 3 step 12 is written: 312
#To highlight those steps, set: EXERCISE_STEPS=[102, 312]
#If you do not want to distinguish any steps, use an empty list: EXERCISE_STEPS=[]
EXERCISE_STEPS=[108,207,212,306,315,407,413]

#SOCIAL_STEPS may be used to define steps where a social activity is prompted
#If you do not want to define any steps, use an empty list: SOCIAL_STEPS=[]
#The steps can then be used to highlight particular steps in comment reports
SOCIAL_STEPS=[101,105,108,207,212,217,306,322,405,407,413]
```

Most of the code should “just work” without further modification, although you can edit and add code cells, as well as markdown/text commentary cells to the notebook, as you see fit.

YOU SHOULD NOT NEED TO MAKE ANY FURTHER CHANGES TO THE NOTEBOOK BELOW THIS LINE (UNLESS YOU WANT TO!)

We can now automatically generate some more dates of interest.

```
STUDY_WEEK_DATES=[]
for i in range(NUMBER_OF_STUDY_WEEKS):
    STUDY_WEEK_DATES.append(offsetDays(COURSE_START_DATE,7*i))
```

Load in the Data

Use the specified data file locations to load the data in.

```
#Load in the data - enrolments file
enrolments=pd.read_csv('{}/{}_enrolments.csv'.format(PATH.strip('/'),STUB),parse_dates=[1,\
2,4])

#Preview the data
enrolments.head(3)
```

```
#Load in the data - steps file - THIS MAY TAKE SOME TIME
steps=pd.read_csv('{}/{}_step-activity.csv'.format(PATH.strip('/'),STUB),
                  parse_dates=[4,5],
                  dtype={'step':object})

#Let's generate an ordered numerical key for the week/step combinations that lets us easil\
y distinguish weeks
steps['flstep']=steps['week_number']*100+steps['step_number']

#Preview the data
steps.head(3)
```

```
#Load in the data - comments file
comments=pd.read_csv('{}/{}_comments.csv'.format(PATH.strip('/'),STUB),parse_dates=[7],dty\
pe={'step': object})

#The step as defined is not brilliant for sorting - create a new step indexing scheme
comments['flstep']=comments['week_number']*100+comments['step_number']

#Preview the data
comments.head(3)
```

```
#Load in the quiz response data - THIS MAY TAKE SOME TIME
qnresp=pd.read_csv('{}/{}_question-response.csv'.format(PATH.strip('/'),STUB),parse_dates=\
[6])

#Create a simple sortable numeric id for each quiz question number
qnresp['qid']=(100*qnresp['week_number'])+qnresp['step_number']+qnresp['question_number']/\
100

#Create a key for a particular response
qnresp['qidr']=qnresp['quiz_question']+'.'+qnresp['response'].apply(str)

#Create the week/step lookup
qnresp['flstep']=qnresp['week_number']*100+qnresp['step_number']

#Preview the data
qnresp.head(3)
```

Simple Reporting

Let's start with some simple reports that describe the overall participation state of the course. It will be convenient to create several lists of user IDs that represent different populatations, such as learners who have fully participated, learners who have completed at least one step, or learners who have commented.

```
#Roles available
print("Assigned roles: {}.".format(', '.join(([r for r in enrolments['role'].unique()]))))
```

A simple function will pull out the unique identifiers of people with a particular role:

```
def usersByRole(role):
    return set(enrolments[enrolments['role']==role]['learner_id'])
```

We can also create some canned groups that may be useful later.

```
#Create a range of lists of user IDs for users with different states of enrolment
enrolled_learners=usersByRole('learner')

unenrolled_learners=set(enrolments[(enrolments['unenrolled_at'].notnull()) &
                               (enrolments['role']=='learner')]['learner_id'])

fullypart_learners=set(enrolments[(enrolments['fully_participated_at'].notnull()) &
                              (enrolments['role']=='learner')]['learner_id'])

#Create a list of learners who have visited at least one step
stepstart_learners=set(steps[steps['first_visited_at'].notnull()]['learner_id'])

#Create a list of learners who have completed at least one step
stepcomplete_learners=set(steps[steps['last_completed_at'].notnull()]['learner_id'])

#Create a list of learners who have commented at least once
commenting_learners=set(comments[comments['author_id'].isin(enrolled_learners)]['author_id\
'])


#Create a list of learners who have completed at least one quiz element
quiz_learners=set(qnresp[qnresp['learner_id'].isin(enrolled_learners)]['learner_id'])


#Define a simple dict to record the groups and a way of retrieving the members of each gro\
up by name
```

```
#These groups can be automatically referenced from some interactive chart controls
grouplist=['enrolled_learners','unenrolled_learners','fullypart_learners',
           'stepstart_learners','stepcomplete_learners','commenting_learners','quiz_learne\
rs']
groupLookup=lambda x: globals()[x]
```

We can now do some simple counting around those sets of individuals.

```
print('Enrolled learners: {:,}'.format(len(enrolled_learners)))
print('Fully participating learners: {:,}'.format(len(fullypart_learners)))
print('Unenrolled learners: {:,}'.format(len(unenrolled_learners)))

print('Enrolled users who completed at least one step: {:,}'.format(len(enrolled_learners.\
intersection(stepcomplete_learners))))
print('Enrolled users who visited at least one step: {:,}'.format(len(enrolled_learners.in\
tersection(stepstart_learners))))
print('Enrolled users who did not visit at least one step: {:,}'.format(len(enrolled_learn\
ers.difference(stepstart_learners))))

print('Unenrolled users who completed at least one step: {:,}'.format(len(unenrolled_learn\
ers.intersection(stepcomplete_learners))))
print('Unenrolled users who visited at least one step: {:,}'.format(len(unenrolled_learner\
s.intersection(stepstart_learners))))
print('Unenrolled users who did not visit at least one step: {:,}'.format(len(unenrolled_l\
earners.difference(stepstart_learners))))

print('Enrolled learners who commented: {:,}'.format(len(commenting_learners)))
print('Enrolled learners who did not comment: {:,}'.format(len(enrolled_learners.differenc\
e(commenting_learners))))
print('Fully participating learners who commented: {:,}'.format(len(commenting_learners.in\
tersection(fullypart_learners))))
print('Fully participating learners who did not comment: {:,}'.format(len(commenting_learn\
ers.difference(fullypart_learners))))
print('Unenrolled learners who commented: {:,}'.format(len(commenting_learners.intersectio\
n(unenrolled_learners))))
print('Unenrolled learners who did not comment: {:,}'.format(len(commenting_learners.diffe\
rence(unenrolled_learners))))
```

These sorts of report can of course be combined into slightly more flowing summaries. For example:

```
report='''
Of the {:,} enrolled learners, {} ({:.2f}%) fully participated and {:,} ({:.2f}%) commente\
d.
However, {:,} enrolled learners ({:.2f}%) did not even visit a single step.
'''.format(len(enrolled_learners),
          len(fullypart_learners),
          100*len(fullypart_learners)/len(enrolled_learners),
          len(commenting_learners),
          100*len(commenting_learners)//len(enrolled_learners),
          len(enrolled_learners.difference(stepstart_learners)),
          100*len(enrolled_learners.difference(stepcomplete_learners))/len(enrolled_learne\
rs))

print(report)
```

Creating a wider range of textualised reports is left as a recreational data activity for the interested reader.

Time Filters

Another way of partitioning the data is to limit it to activity occurring within a particular time of day range. The following functions allow you to limit data sets to data contained within a particular date (or datetime) range or time of day range.

```
#Display the enrolment/unenrollment/fully participating counts
#If dates are specified, use the form YYYY-MM-DD
def date_limiter(df, start=None, end=None, index=None):
    if index is not None:
        df=df.set_index(index)
    df.sort_index(inplace=True)
    if start is not None and end is not None:
        df=df[start:end]
    elif start is not None:
        df=df[start:]
    elif end is not None:
        df=df[:end]
    return df

#http://stackoverflow.com/a/10567298/454773
def timeOfDay_limiter(df, start=None, end=None, index=None):
    if index is not None:
        df=df.set_index(index)
        df.sort_index(inplace=True)
    hour = df.index.hour
    if start is not None and end is not None:
        selector = ((start <= hour) & (hour <= end))
        df = df[selector]
    elif start is not None:
        selector = ((start <= hour) & (hour <= 23))
        df = df[selector]
    elif end is not None:
        selector = ((0 <= hour) & (hour <= end))
        df = df[selector]
    return df


#Let's see if there are any early starters working around 1am on the first day...
date_limiter(timeOfDay_limiter(steps,start=1,end=1,index='first_visited_at'),
             COURSE_START_DATE,
             COURSE_START_DATE ).head(3)


#We can also slice into hour/minute blocks using the date_limiter() function
tmp=date_limiter(steps, start=COURSE_START_DATE+' 01:10:50',
                 end=COURSE_START_DATE+' 02',
                 index='first_visited_at')
print(tmp.head(1),tmp.tail(1))
steps.reset_index()
tmp=None
```

Enrolment Summaries

Enrolment summary reports are the first reports we can start to generate, in advance of the course opening date.

It's easy enough to plot a chart that shpws the accumulated count of learners in the data to date:

```
def plot_cumulativeCount(df,group,groupset,index,title):
    df=df[df[group].isin(groupset)]
    df=df.reset_index().set_index(index)
    df.sort_index(inplace=True)
    df['enaccnum']=range(len(df))
    ax=df['enaccnum'].plot(legend=False, title=title)
    ax.set_xlabel('')



plot_cumulativeCount(enrolments,'learner_id',enrolled_learners,'enrolled_at',
                     'Enrolment count onto the course')
```

We can also try to be a bit cleverer, and display returns for learners who *registered* within a particular time period to track their unenrolment and full participation behaviour as a group.

An additional helper function overplots notable date lines onto the chart, such as course open registration date and course start date.

```
def notableDatePlot(ax):
    for sd in SPECIAL_DATES:
        ax.axvline(pd.to_datetime(sd), color='grey', linestyle='--', lw=2)

def studyWeekDatePlot(ax):
    for sd in STUDY_WEEK_DATES:
        ax.axvline(pd.to_datetime(sd), color='grey', linestyle='--', lw=2)

def enrolInWindow(en,start=None,end=None,edge=False,notableDates=True):
    ''' Normally the start and end dates apply to enrolment date.
        Any full participation or unenrolment dates are relative to those individuals so m\
ay extend
        outside the start/end date range.
        To hard limit the chart start/end dates, apply the edge=True setting. '''
    enw=en.set_index('enrolled_at')
    enw.sort_index(inplace=True)
    enw=date_limiter(enw, start, end)
    enw['enaccnum']=range(len(enw))
    ax=enw['enaccnum'].plot(title='Enrolment count onto the course',
                              figsize=(15,10),legend=True,label='Enrolments',color='b')
    unenw=enw[enw['unenrolled_at'].notnull()].copy()
    unenw=unenw.set_index('unenrolled_at')
    unenw.sort_index(inplace=True)
    if edge:
        unenw=date_limiter(unenw, start, end)
        if len(unenw)>0:
            unenw['unenaccnum']=range(len(unenw))
            ax=unenw['unenaccnum'].plot(ax=ax,legend=True,label='Unenrolments',color='r')

    fullp=enw[enw['fully_participated_at'].notnull()].copy()
    fullp=fullp.set_index('fully_participated_at')
    fullp.sort_index(inplace=True)
    if edge:
        fullp=date_limiter(fullp, start, end)
        if len(fullp)>0:
            fullp['fullpaccnum']=range(len(fullp))
            ax=fullp['fullpaccnum'].plot(ax=ax,legend=True,label='Full participation',colo\
r='g')
    ax.set_xlabel("Date")

    if notableDates:
        notableDatePlot(ax)
```

In the following view, we plot enrolment stats for all enrolled learners over all time.

```
enrolInWindow(enrolments[enrolments['learner_id'].isin(enrolled_learners)])
```

Things to look for include bursts around notable dates. Additional notable dates (such as mailings or campaign dates), can be added to the charting function.

We can also look at the behaviour of learners who enrolled before, after or between particular dates. Note that by default, the date limiting applies to when learners *enrolled* - unenrolment or full completion dates may occur in the time period after any specified enrolment period.

```
#Look at the behaviour of folk who enrolled on or after the course start date
enrolInWindow(enrolments[enrolments['learner_id'].isin(enrolled_learners)],
              start=COURSE_START_DATE)
```

If we apply the `edge=True` argument, the chart display is hard limited to the specified date range. However, the unenrolment and fully participated counts will still be relative to the population who enrolled in the specified period.

If you want a chart generated over all data but displaying within just a limited time window, set the range via the actual chart parameters (*method not described*).

```
#Look at the behaviour of learners in the period up to the course start date
enrolInWindow(enrolments[enrolments['learner_id'].isin(enrolled_learners)],
              end=COURSE_START_DATE, edge=True)
```

Enrolment Behaviour

Used as a standalone dataset, enrolment data allows us to ask a variety of questions about when learners enrol and unenrol, the duration between enrolling and unenrolling, and so on.

Enrolment dates also allow us to identify several distinct groupings of learners:

- learners who enrolled when the course was first announced;
- learners who enrolled between the course first being announced and actual start date;
- learners who enrolled around on or shortly after the course start date;
- learners who enrolled some time after the start date, eg after the first week of the course.

These groupings can be retrieved using the `date_limiter()` function, assuming that the index is set to the appropriate timestamp:

```
#Find learners who registered on the course start date
startdatelearners=date_limiter(enrolments[enrolments['learner_id'].isin(enrolled_learners)\
],
                               start=COURSE_START_DATE,
                               end=COURSE_START_DATE,
                               index='enrolled_at')
print('{} learners enrolled on the course start date, {}.'.format(len(startdatelearners),
                                                                  COURSE_START_DATE))
```

As well as using enrolment returns to monitor the effectiveness of promotional, call-to-action registration campaigns through monitoring enrolment numbers, we can also use the enrolment data to explore unenrolments, for example asking how long learner who unenrol from the course are enrolled on to it.

```
#The difference between enrolment time and unenrolment times gives us the period of enrolm\
ent.
enrolmentLasted=enrolments[enrolments['learner_id'].isin(unenrolled_learners)][:]
enrolmentLasted['entime']=(enrolmentLasted['unenrolled_at']-enrolmentLasted['enrolled_at']\
).astype('timedelta64[D]')

#We can also generate the number of enrolled "study days":
#Calculate relative to enrolment or course start date, whichever is the later
enrolmentLasted['studyentime']= np.where(enrolmentLasted['enrolled_at'] >= pd.to_datetime(\
COURSE_START_DATE), \
                                              (enrolmentLasted['unenrolled_at']-enrolmentL\
asted['enrolled_at']).astype('timedelta64[D]'), \
                                              (enrolmentLasted['unenrolled_at']-pd.to_date\
time(COURSE_START_DATE)).astype('timedelta64[D]'))
```

For learners who enrol before the course start date, how many days do they remain enrolled?

```
WHICH_TIME='entime'
#WHICH_TIME can be: entime, studyentime; title/axis labels will need updating correspondin\
gly
tmp=date_limiter(enrolmentLasted,
                end=COURSE_START_DATE,
                index='enrolled_at').groupby(WHICH_TIME).size().reset_index()
ax=tmp.plot(kind='scatter',
            x=WHICH_TIME, y=0, figsize=(15, 10),
            title='Enrolment duration for unenrolled learners enrolling before {}'.format(\
COURSE_START_DATE))
ax.set_xlabel('Enrolment duration (days)')
ax.set_ylabel('Number of unenrolments')
```

Things to look out for:

- enrolment durations for early enrollers that correspond to the number of days between the course being announced and the start of the course.

Looking at the number of available study days a learner was enrolled before unenrolling gives us a view on the extent to which unenrolment occurred prior to the course start date (‘negative’ study days are days prior to the start of the course). We can plot this as a scatterplot, (as above) or as a histogram.

```
enrolmentLasted['studyentime'].plot(kind='hist',
                                    title='Time to full unenrolment (available study days)\
')
```

```
#Are there any popular enrolment dates?
tmp=date_limiter(enrolmentLasted,
                index='enrolled_at')

ax=tmp.groupby(tmp.index.date).size().plot( figsize=(15, 10),
          title='Number of enrolments per day' )
ax.axvline(pd.to_datetime(COURSE_START_DATE), color='grey', linestyle='--', lw=2)
```

```
#Popular enrolment dates
tmp.groupby(tmp.index.date).size().sort_values(ascending=False).head()
```

```
#Are there any popular unenrolment dates?
tmp=date_limiter(enrolmentLasted,
                index='unenrolled_at')

ax=tmp.groupby(tmp.index.date).size().plot( figsize=(15, 10),
          title='Number of unenrolments per day' )
notableDatePlot(ax)
```

```
#Popular unenrolment dates
tmp.groupby(tmp.index.date).size().sort_values(ascending=False).head(10)
```

Possible notable dates for unenrolments:

- date course is first announced (people enrol, then unenrol on same day);
- date a week before course starts when a reminder mailing goes out;
- start date of course when folk realise it's not what they expected;
- date reminders go out at start of each study week.

A similar calculation to the live-course / ‘study days’ enrolment period can also be calculated for fully participating learners.

```
#Calculate the enrolment time before full-participation
enrolmentFullyPart=enrolments[enrolments['learner_id'].isin(fullypart_learners)][:]
enrolmentFullyPart['fullyparttime']= np.where(enrolmentFullyPart['enrolled_at'] >= pd.to_d\
atetime(COURSE_START_DATE), \
                                              (enrolmentFullyPart['fully_participated_at']\
-enrolmentFullyPart['enrolled_at']).astype('timedelta64[D]'), \
                                              (enrolmentFullyPart['fully_participated_at']\
-pd.to_datetime(COURSE_START_DATE)).astype('timedelta64[D]'))


#Show time to full participation relative to enrolment date or course start date, whicheve\
r is later
enrolmentFullyPart['fullyparttime'].plot(kind='hist',title='Time to full participation (av\
ailable study days)')
```

If you've paced the course for an N week period, you're probably looking for evidence by the end of the course of a peak around 7N study days for your fully participating learners.

Adding in Some Step Intelligence

If we combine the `enrolment` data with the `steps` data, we can start to explore whether there are any particular steps that are associated with learners unenrolling or otherwise withdrawing / disengaging from the course.

Looking at Last *First Visited* Data

For learners who unenroll, are there any particular steps that are commonly the last visited step?

One way of estimating this is to assume that learners follow steps in order and then find the largest step number visitied for each unenrolled learner.

```
#merge enrolment data with the largest step number recorded for each unenrolled learner
tmp=pd.merge(enrolmentLasted,
             steps.sort_values('flstep',ascending=False).groupby('learner_id')['flstep'].f\
irst().reset_index(),
             on='learner_id')
```

The following table identifies steps that are commonly the last step encountered by unenrolled learners. In other words, if learners unenroll, these may be the steps that may have been the one to finally drive them away.

(The step numbers are coded `WSS`, where `W` is the week number and `SS` the step number; for example, *week 1 step 2* is identified as `102`.

```
tmp.groupby('flstep').size().sort_values(ascending=False).head(10)
```

We can also visualise the full table as a bar chart. The peaks are the most common steps last encountered by learners who unenroll.

I can't figure out how to addd dashed lines to the chart to show "distinguished" steps such as exercise or activity steps, but we can highlight particular bars - so let's create a simple function to highlight the exercise steps.

```
#The steps highlighter can use thing like:
#exercise (for EXERCISE_STEPS)
#social (for SOCIAL_STEPS)

#?Promote colour settings up to (optional) user defined values?
EXERCISE_COLOUR='#aa3333'
SOCIAL_COLOUR='#33aa33'

def highlightSteps(ax,df,typ='exercise'):
    if typ=='exercise':
        vals=EXERCISE_STEPS
        col=EXERCISE_COLOUR
    elif typ=='social':
        vals=SOCIAL_STEPS
        col=SOCIAL_COLOUR
    else: return
    if df.index.name!='flstep':df.set_index('flstep',inplace=True)
    for highlight in vals:
        if highlight in df.index:
            pos = df.index.get_loc(highlight)
            ax.patches[pos].set_facecolor(col)
```

Now we can decorate the chart to highlight steps associated with exercises or social activities.

```
#Chart of number of people who unenrolled on a particular step
#Note that we need to bring in the full range of steps from the original steps dataframe
tmp2=pd.merge(pd.DataFrame({'flstep':steps['flstep'].unique()}),
              tmp.groupby('flstep').size().reset_index(),
              on='flstep', how='left')
#Make sure the steps are ordered correctly
tmp2.sort_index(inplace=True)
ax=tmp2.plot(kind='bar',x='flstep',y=0,legend=False,figsize=(15,10),
             title='Number of learners unenrolling on a particular step.')


#Now we can highlight any distinguised bars
#EXERCISE_STEPS defines a list of distinguished steps encoded as WSS (Week*100)+Step; step\
 1->01)
#http://stackoverflow.com/a/20395817/454773
highlightSteps(ax,tmp2,'exercise')
```

Possible things to look for:

- people unenrolling on the last step of a week (gave it a try, didn't work out, or got what they needed?);

- people enrolling on the first step of a week (looking forward, doesn't seem to offer what's expected? Style not to liking?);
- people enrolling on an early mid-week step (some activity / exercise that is not what was expected and/or otherwise puts people off?)

More generally, and to take into account learners who withdraw silently and without unenrolling, it may be worth looking at the last steps that learners in general (not just ones who unenrolled) either first visit or mark as complete.

Let's start with a count of how many times each step is the last 'first visited' step across all learners; we can do this for all learners, or ignore learners who unenrolled:

```
sample=steps[steps['learner_id'].isin(enrolled_learners)]
#Other samples include:
#- exclude unenrolled learners
##sample=steps[~steps['learner_id'].isin(unenrolled_learners)]
#-include all learners
##sample=steps[steps['learner_id'].isin(enrolled_learners)]
#-include all learners who didn't fully participate
##sample=steps[~steps['learner_id'].isin(fullypart_learners)]
#We could also sample based on learners who enrolled before the course start date, after t\
he course start date, etc
```

For this sample, grab the learners' last `first visited` steps. This partly assumes that learners only visit a page once, which may or may not be true. (FutureLearn doesn't make 'last visited' data available as part of the default data.)

The table records the step number followed by the number of learners for whom this was their last first visited step.

```
#Sort on the basis of first_visited time, with most recent (largest) time first
tmp=sample.sort_values('first_visited_at',ascending=False).groupby('learner_id')
#Now take the first (most recent) record for each learner
tmp=tmp[['first_visited_at','flstep']].first().reset_index()
#The date_limiter() function also sets the index
tmp=date_limiter(tmp, index='first_visited_at')
tmp.groupby('flstep').size().sort_values(ascending=False).head(10)
```

If we chart the result, the peaks show us the steps which were the last first visited steps for a large number of users. These steps might be worth looking at to see if there is any indication as to why learners may not go on to visit any further steps.

Note that we identified the last first visited step by date, rather than step. This means that a learner might visit a higher step number at some point in time *before* a lower step number. In such a case, a lower step number may be recorded as a learner's last step (i.e. last in terms of time first visited) even though they had previously visited steps further into the course.

```
tmp.groupby('flstep').size().sort_index().plot(kind='bar',x='flstep',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Count of times each step is the las\
t "first visited" step')
```

These last "first visited" step counts identify those steps last encountered by learners for the first time in terms of time. We can also run an accumulated count of 'last seen' learners over the steps. The chart counts how many people who have 'last first seen' a particular step number, which is to say, the number of people who did not later progress to another step for the first time.

An interpretation of this chart is that the vertical y-axis provides an indication of how many people have been "lost" from the course by any particular point (i.e. people who know longer see any steps for the first time).

```
tmp.groupby('flstep').size().sort_index().cumsum().plot(kind='bar',x='flstep',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Accumulated number of people who ha\
ve had their last "first visited" step.')
```

If we subtract the accumulated count from the overall number of learners in the sample, this gives us an indication of how many people visited at least one more step (that is, an indication of the number of people who do visit at least one more step for the first time).

```
tmp.groupby('flstep').size().sort_index().cumsum().rsub(len(tmp)).plot(kind='bar',x='flste\
p',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Number of people who "first visited\
" at least one more step after first visiting a particular step.')
```

Looking at Last *Last Completed* Data

We can also identify those steps that learners last marked as complete. This allows us to identify steps where learners were last motivated to tell us they'd completed the step.

```
#Sort against time to find the last steps (in time) complete by each learner
tmp=sample.sort_values('last_completed_at',ascending=False).groupby('learner_id')
#Take the most recent record for each learner
tmp=tmp[['last_completed_at','flstep']].first().reset_index()
tmp=date_limiter(tmp, index='last_completed_at')
tmp.groupby('flstep').size().sort_values(ascending=False).head(10)
tmp.groupby('flstep').size().sort_index().plot(kind='bar',x='flstep',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Count of times each step is the las\
t "last completed" step')
```

As before, we can chart the accumulated count to get a feel for how far people did - or did not - get into the course in terms of steps marked as complete. Once again, the y-axis charts how many people have stopped updating step completion checkboxes by a particular step.

```
tmp.groupby('flstep').size().sort_index().cumsum().plot(kind='bar',x='flstep',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Accumulated count of number of peop\
le who have posted their last "last completed" step')
```

As before, we can subtract the accumulated count from the sample size to generate a bar chart in which the height of the bars gives us an indication of how many people went on fron the current step to mark at least one more step as complete.

```
#The rsub() method subtracts the calling column value from the value passed into the method
tmp.groupby('flstep').size().sort_index().cumsum().rsub(len(tmp)).plot(kind='bar',x='flste\
p',y=0,
                                                legend=False,figsize=(15,10),
                                                title='Count of times someone leave each s\
tep to go on to complete at least one more "last completed" step')
```

Step Intelligence - Activity Measures

For each step, we can also count the number of learners who first visited or last completed each step, who commented on each step, or who engaged in quiz activity on a particular step.

A basic formulation of this sort of report is something like the following:

```
steps[(steps['first_visited_at'].notnull()) & (steps['learner_id'].isin(enrolled_learners)\
)]['flstep'].value_counts().sort_index().head()
#Alternative formulation:
#steps[(steps['first_visited_at'].notnull()) & (steps['learner_id'].isin(enrolled_learners\
))].groupby('flstep').size().head()
```

It's easy enough to create a simple function bound to an interactive widget that lets us chart these counts for different sorts of activity for different groups of learners.

To configure the charts, we need to parse the relevant timestamps, so we'll define a dict to perform the relevant mapping.

```
tsDict={'comments': 'timestamp',
        'steps_first': 'first_visited_at',
        'steps_last':'last_completed_at',
        'questions': 'submitted_at',
        'enrolments': 'enrolled_at',
        'unenrolments': 'unenrolled_at'
        }

tsData={'comments': comments,
        'steps_first': steps,
        'steps_last':steps,
        'questions': qnresp,
        'enrolments': enrolments,
        'unenrolments': enrolments
        }
```

The interactive function is defined to allow us to display different forms of activity report for different groups of learner.

With a bit of jiggery pokery, we can define highlighting rules within the function, for example to highlight bars as social steps in `comments` reports.

```
def i_coreLearnerStepActivityCounts(df,activity,group,igroup,enhancements=True):
    if df.index.name is not None: df.reset_index(inplace=True)
    pid='author_id' if activity=='comments' else 'learner_id'
    if 'flstep' in df.columns:
        _activity=tsDict[activity]
        tmp=df[(df[_activity].notnull()) & (df[pid].isin(igroup))]['flstep'].value_counts(\
).sort_index()
        ax=tmp.plot(kind='bar',figsize=(15,10),
                    title='Learner interaction count for {} of their {} activity at each s\
tep.'.format(group,_activity))
    else: print('Chart not available for that activity.')

    if not enhancements: return

    tmp.index.names=['flstep']
    if activity=='comments': highlightSteps(ax,tmp,'social')

def i_learnerStepActivityCounts(activity='steps_first',group='enrolled_learners'):
    igroup=groupLookup(group)
    df=tsData[activity]
    i_coreLearnerStepActivityCounts(df,activity,group,igroup)

gk=grouplist
gk.sort()
ak=list(tsData.keys())
ak.sort()
interact(i_learnerStepActivityCounts,
         activity=ak,
         group=gk)
```

We can also use a date filter to tunnel down on activity within a particular period - a quick sketch shows how to produce a chart that displays step activity on a single day.

```
tmp=date_limiter(steps,'2015-11-04','2015-11-04',index='first_visited_at')
tmp['flstep'].value_counts().sort_index().plot(kind='bar',figsize=(15,10))
```

We can generalise this to produce the first fumblings of interactive that allows us to explore data filtered according to a particular time range.

A utility function helps us generate dates relative to dates of interest, declared by their name *as a string*.

```
def startEndRange(start,end,start_offset=0,end_offset=0):
    start=globals()[start]
    end=globals()[end]
    start= offsetDays(start,start_offset)
    end=offsetDays(end,end_offset)
    return (start,end)
```

The interactive function itself will construct the date range relative to dates of interest.

Note that if additional dates of interest are declared appropriately at the start of the notebook, they will be available as options in these reports.

```
def periodBasedActivityReport(activity='steps_first',
                              group='enrolled_learners',
                              start='COURSE_START_DATE',
                              start_offset=0,
                              end='COURSE_START_DATE',
                              end_offset=7):
    #tmp=date_limiter(df,'2015-11-04','2015-11-04',index='first_visited_at')
    #tmp['flstep'].value_counts().sort_index().plot(kind='bar',figsize=(15,10))
    (start,end) = startEndRange(start,end,start_offset,end_offset)
    igroup=groupLookup(group)
    df=tsData[activity]
    df=date_limiter(df,start,end,index=tsDict[activity])
    i_coreLearnerStepActivityCounts(df,activity,group,igroup)

activities=list(tsDict.keys())
activities.sort()

interact(periodBasedActivityReport,
         report=['hourOfDay','dayOfWeek'],
         group=gk,
         activity=activities,
         start=['COURSE_OPEN_REGISTRATION','COURSE_START_DATE',
                'COURSE_END_DATE','COURSE_PRE_START_MAILINGDATE'],
         end=['COURSE_OPEN_REGISTRATION','COURSE_START_DATE',
              'COURSE_END_DATE','COURSE_PRE_START_MAILINGDATE'],
         start_offset=(0,35,1),
         end_offset=(0,35,1),
        )
```

Step Intelligence - Social Activity in More Detail

The *Step Intelligence Activity Reports* allow us to explore the general commenting behaviour around particular steps (select the `comments` actvity), but we can also probe a little further. For example, we can explore the steps on which learners *first* commented.

```
#Sort the comments by timestamp
if comments.index.name!='timestamp':comments.set_index('timestamp',inplace=True)
#One way of identifying first comments by individual is to remove each duplicate author_id\
 and just True the first
tmp=comments.sort_index()[~(comments.duplicated('author_id'))].groupby(['flstep'])['author\
_id'].nunique()
ax=tmp.plot(kind='bar',title='Count of first comment by step')
highlightSteps(ax,tmp,'social')
```

The *first comment* report identifies those steps where learners make their first comment. This can be used to check the performance of “welcome” steps or first major social activity steps.

Reporting on Social Activity

Although not really any more useful than as a simple indicator that there is some commenting activity taking place on a course, many people like to see activity indication charts that go up and to the right, such as the growth in number of comments over time.

These sorts of charts are largely meaningless (are a large number of learners commenting infrequently, or are a small number of learners commenting regularly, or some combination thereof?), except perhaps when you look to the gradient for information about *rates* of commenting, but pointless charts such as these dominate many dashboards and reports, so here's how to create them.

To begin with, we can sort the comments by timestamp and then generate an accumulated count number for each one plotting the result over time.

```
if comments.index.name!='timestamp':comments.set_index('timestamp',inplace=True)
comments.sort_index(inplace=True)
comments['accnum']=range(len(comments))

comments.plot(y='accnum',legend=False, title='Accession plot of comments')
```

We can also spy on the educators to see how their comments grow as a group.

```
educator_comments=comments[comments['author_id'].isin(usersByRole('educator'))][:]
if not len(educator_comments.index):
    print('No comments by educators in that period? Further educator_comment cells may thr\
ow an error.')
```

(It would perhaps make sense to just support a generic role/comment explorer here, using a widget to select role.)

```
educator_comments['accnum']=range(len(educator_comments))
educator_comments.plot(y='accnum',legend=False,
                       title='Accession plot of comments by educators')
```

A staircase style of chart shows short bursts of activity.

The charts start to become more useful if we start to segment the populations around which the charts are generated, so we can get a feel for how the different populations behave.

For example, if we're going to pry into learner activity, we should fully expect folk to pry into educator behaviour too. (If we're going to analyse the heck out of learners, educators should be fully prepared for personnel/HR folk to start analysing the heck out of them in return… After all, why should they miss out on all this analytics hype…?! *Learning anaytics* -> *work analytics*. You have been warned.)

To start with, let's create a simple utility function to plot around groups in a tidy way.

```
def accessionPlotByGroup(df,group,attr,title='',enhancements=True):
    #Plot the growth in comments for each week
    fig, ax = plt.subplots()
    grouped = df.groupby(group)
    for key, group in grouped:
        group.plot(ax=ax, y=attr,label=key,title=title)
    if not enhancements: return

    ax=studyWeekDatePlot(ax)
```

Now let's pick into the indvidual educator activity.

```
educator_comments['educator_accnum']=''
educator_comments['educator_accnum']=educator_comments.groupby('author_id')['educator_accn\
um'].transform(lambda x: range(len(x.sort_index())))

ax=accessionPlotByGroup(educator_comments,'author_id','educator_accnum',
                        title='Accession plot of comments by educator with week start date\
s marked')
```

A similar grouping technique can be used to split out the growth in comment numbers according the number of comments made on steps associated with a particular week.

```
if comments.index.name!='timestamp': comments=comments.set_index('timestamp').sort_index()
comments['week_accnum']=''
comments['week_accnum']=comments.groupby('week_number')['week_accnum'].cumcount()

accessionPlotByGroup(comments,'week_number','week_accnum',
                     title='Accession plot of comments by week')
```

A key thing to look for here are different waves of activity associated with learners starting to comment on steps associated with each study week.

We can generalise that sort of function as the basis for an interactive report generator:

```
def i_coreLearnerStepAccessionCounts(df,activity,group,igroup,enhancements=True):
    pid='author_id' if activity=='comments' else 'learner_id'
    df=df[df[pid].isin(igroup)][:]

    if df.index.name!=tsDict[activity]: df=df.set_index(tsDict[activity]).sort_index()
    df['week_accnum']=''
    df['week_accnum']=df.groupby('week_number')['week_accnum'].cumcount()
    accessionPlotByGroup(df,'week_number','week_accnum',
                         title='Accession plot of {} by week'.format(activity),
                         enhancements=enhancements)

def i_learnerStepAccessionCounts(activity='steps_first',group='enrolled_learners'):
    igroup=groupLookup(group)
    df=tsData[activity]
    i_coreLearnerStepAccessionCounts(df,activity,group,igroup)

interact(i_learnerStepAccessionCounts,
         activity=['comments','questions'],
         group=gk)
```

Exploring Study Sessions and the Time Spent in a Study Session

With data being provided at the individual level, we can try to identify the structure of individual study sessions completed by each learner, along with the amount of time they are spending on their studies. This information can then be aggregated, perhaps according to different user groups (fully participating learners, for example, or learners who unenrolled). We can also start to search for any common patterns in study behaviour that we might be able to draw on to help us improve the design and pacing of the materials.

The original inspiration for these sketches was based around the idea that we should be able to tell various sorts of story about how folk progress through the course based on snapshots of indivisal learner behvaiour.

Note the the idea is *not* to try to build models of learners so we can then make direct interventions against individual learners. The motivation is to understand learner progress so we can improve the way that static course materials are designed and how learners use them. ("Personalised" courses can also be thought of as interventions made to course materials that are otherwise failing learners in general.)

To start with, let's see how an indvidual learner, selected at random from the set of fully participating learners studied the course.

Create a couple of utility functions to help us select a random (but by default, fully participating) learner.

```python
#Select a random, fully participating learner
def getRandomLearner(typ=fullypart_learners):
    return random.choice(list(typ))

def getRandomLearnerSteps(lid):
    return steps[steps['learner_id']==lid].copy()
```

When charting learner step activity, time will go along the horizontal x-axis, and encoded step number (WSS) on the vertical y-axis. The spacing of the y-axis will allow us to clearly see the activity associated with seperate weeks, but we can reinforce it by adding horizontal lines to segment the study week steps further.

```
def stepHlines(plt):
    for w in range(NUMBER_OF_STUDY_WEEKS):
        plt.axhline(y=150+(100*w), color='grey', linestyle='--', lw=2)

def learnerPlot(learner):
    if learner.index.name!='first_visited_at': learner.set_index('first_visited_at',inplac\
e=True)
    plt.plot_date(learner.index,y=learner['flstep'],markersize=4)
    stepHlines(plt)
```

Select a random fully participating learner and generate the activity plot.

```
learnerId=getRandomLearner()
learner=getRandomLearnerSteps(learnerId)
learnerPlot(learner)
```

In most cases, you should see clear groupings of activity reading horizontally across the chart, representing different study periods.

A proxy for the time *between* steps is the time taken on a step, measured as the difference between firest visited step times. (This assumes a learner working linearly through the course.)

As a heuristic, if the (proxy) time spent on a single step is greater than a specified period (greater than or equal to M minutes or H hours,for example, we might assume that there is a break between study sessions; I'm going to opt for a period of at least 65 minutes (defined via the ealier set `SESSION_GAP`). (If a practical activity takes over an hour, this means it will be counted as two study sessions).

(There are probably far better segmenters available, or even better quick approximators (eg based on z-score?); one for the to do list…)

```
def learnerSessionSteps(learner,period=SESSION_GAP):
    #This isn't really optimised at all...
    #Should probably sort the index once rather than on each pass for each learner
    if learner.index.name is not 'first_visited_at':
        learner.set_index('first_visited_at',inplace=True)
    learner.sort_index(inplace=True)
    learner.reset_index(inplace=True)
    learner['stepdiff']=learner['first_visited_at'].diff()
    learner['stepdiff_m']=learner['stepdiff'].astype('timedelta64[m]')
    learner['stepdiff_h']=learner['stepdiff'].astype('timedelta64[h]')
    h,m=minutes, seconds= divmod(period, 60)
    learner['newSession']= (learner['stepdiff_m']>=m) & (learner['stepdiff_h']>=h)
    learner['sessionNum']=learner['newSession'].cumsum()
    return learner
```

```
def learnerPlotColoured(learner,period=SESSION_GAP):
    #Belt and braces in case the perios was changed
    learner=learnerSessionSteps(learner,period)
    if learner.index.name!='first_visited_at': learner.set_index('first_visited_at',inplac\
e=True)
    learner.groupby('sessionNum').apply(lambda x: plt.plot_date(x.index,y=x['flstep'],mark\
ersize=4))

learner=learnerSessionSteps(learner)
learnerPlotColoured(learner)
```

For the same learner, summarise how many steps were in each study session.

```
print('This learner appears to have engaged in {} study sessions.'.format(len(learner.grou\
pby('sessionNum'))))
learner.groupby('sessionNum').size().plot(kind='bar')
```

We can also chart the number of steps carried out within each study session and course week for the same learner.

```
learner.groupby(['sessionNum','week_number']).size().plot(kind='barh')
```

Based on the intuitions and crude approximations used above, we can split out the separate sessions for a particular learner and rebase them against the start time for the session.

The timebase provided in the datasets is the datetime of the recorded events. We can also rebase the time to a normalised/relative starting points, most naturally:

- the earliest `first_visited_at time` within the steps data for a course by a learner, which allows us to estimate when a learner first started to look at the learning materials associated with the course;
- the earliest `first_visited_at time` within a particular study session, which allows us to see how learners use a study session.

Let’s start by rebasing learners to give us a measure on individual elapsed time in hours since starting the course.

```
def rebase_learners(steps):
    #This should only be run once
    if 'learnertime_h' in steps.columns: return steps

    if steps.index.name=='first_visited_at': steps.reset_index(inplace=True)
    #Set a rebase origin for each learner based on their first step time
    rebase=steps.groupby('learner_id')['first_visited_at'].aggregate(min)
    rebase.name='first_seen'
    rebase=rebase.reset_index()
    #Merge the rebase time back in to whatever you want to rebase against, and calculate t\
he difference
    steps=pd.merge(steps,rebase,on='learner_id')
    steps['learnertime']=steps['first_visited_at']-steps['first_seen']

    #Set up some basic difference measures - eg time from first sighting in hours
    steps['learnertime_h']=steps['learnertime'].astype('timedelta64[h]')
    return steps

steps=rebase_learners(steps)
```

This rebasing allows us to compare the study patterns of two or more learners over the course. The horizontal x-axis should probably be labeled in days rather hours to make it a bit more readable.

```
ax=steps[steps['learner_id']==getRandomLearner()].plot('learnertime_h','flstep',kind='scat\
ter',color='orange')
steps[steps['learner_id']==getRandomLearner()].plot('learnertime_h','flstep',kind='scatter\
',ax=ax, color='purple')
```

An alternative rebasing sets the original time to the start of each individual study session. T

```
def learner_rebaseSession(learner,period=SESSION_GAP):
    if learner.index.name is not None: learner.reset_index(inplace=True)
    learner=learnerSessionSteps(learner,period)
    #Set the start time for each session as the earliest seen first_visited time in the se\
ssion
    learner_rebase=learner.groupby('sessionNum')['first_visited_at'].aggregate(min)
    learner_rebase.name='first_seen_session'

    #Generate an increasing session time across steps
    learner=pd.merge(learner,learner_rebase.reset_index(),on='sessionNum')
    learner['learnersessiontime']=learner['first_visited_at']-learner['first_seen_session']
    learner['learnersessiontime_h']=learner['learnersessiontime'].astype('timedelta64[h]')
    learner['learnersessiontime_m']=learner['learnersessiontime'].astype('timedelta64[m]')
    return learner
```

```
learnerId=getRandomLearner()
learner=getRandomLearnerSteps(learnerId)

test=learner_rebaseSession(learner.copy())
fig, ax = plt.subplots()

#Filter out groups less than a particular length
#test=test.groupby('sessionNum').filter(lambda x: len(x) > 5)

for key, group in test.groupby('sessionNum'):
    group.plot('learnersessiontime_m','flstep',kind='scatter',ax=ax)
#I'm not sure why I can't get this to display different colours for each session?
```

If we can generate reports for an individual, we can run them over all learners. The following function will calculate the session times over all learners.

```
#THIS CELL MAY TAKE QUITE A LONG TIME TO RUN...
learnerSessionTimes=steps.groupby('learner_id').apply(learner_rebaseSession)
```

We can now start to generate various reports over the data regarding the amount of time learners spent engaged with the course as well as the amount of time actually spent studying it.

```
tmp_fullLearners=learnerSessionTimes[learnerSessionTimes['learner_id'].isin(fullypart_lear\
ners)]
tmp_unenrolledLearners=learnerSessionTimes[learnerSessionTimes['learner_id'].isin(unenroll\
ed_learners)]

#How many sessions on average do fully participating learners engage in?
tmp_val=tmp_fullLearners[['learner_id','sessionNum']].groupby('learner_id').aggregate(lamb\
da x: 1+max(x['sessionNum']))
print('Number of study sessions fully participating learners engage in: mean {:.2f}, media\
n {:.2f}'.format(tmp_val.mean()[0],tmp_val.median()[0]))

#How many sessions on average do unenrolled learners engage in?
tmp_val=tmp_unenrolledLearners[['learner_id','sessionNum']].groupby('learner_id').aggregat\
e(lambda x: 1+max(x['sessionNum']))
print('Mean number of study sessions unenrolled learners engage in: mean {:.2f}, median {:\
.2f}'.format(tmp_val.mean()[0],tmp_val.median()[0]))

tmp_val=tmp_fullLearners[['learner_id','sessionNum','learnersessiontime']].groupby('learne\
r_id').aggregate(lambda x: np.mean(x['learnersessiontime']))['learnersessiontime']
print('Duration of study sessions fully participating learners engage in: mean {}, median \
{}'.format(tmp_val.mean(),tmp_val.median()))
```

```
tmp_val=tmp_unenrolledLearners[['learner_id','sessionNum','learnersessiontime']].groupby('\
learner_id').aggregate(lambda x: np.mean(x['learnersessiontime']))['learnersessiontime']
print('Duration of study sessions unenrolled learners engage in: mean {}, median {}'.forma\
t(tmp_val.mean(),tmp_val.median()))

#The following is an approximation based on summed step difference times within a study se\
ssion
tmp_val=learnerSessionTimes[(learnerSessionTimes['learner_id'].isin(fullypart_learners)) &\
 (learnerSessionTimes['newSession']==False)][['learner_id','sessionNum','stepdiff']].group\
by('learner_id').aggregate(lambda x: np.sum(x['stepdiff']))['stepdiff']
print('Overall duration (ish) of study for fully participating learners: mean {}, median {\
}'.format(tmp_val.mean(),tmp_val.median()))

tmp_val=learnerSessionTimes[(learnerSessionTimes['learner_id'].isin(unenrolled_learners)) \
& (learnerSessionTimes['newSession']==False)][['learner_id','sessionNum','stepdiff']].grou\
pby('learner_id').aggregate(lambda x: np.sum(x['stepdiff']))['stepdiff']
print('Overall duration (ish) of study for unenrolled learners: mean {}, median {}'.format\
(tmp_val.mean(),tmp_val.median()))

#The following is based on the difference between first and last visited steps;
tmp_val=tmp_fullLearners[['learner_id','first_visited_at']].groupby('learner_id').aggregat\
e(lambda x: np.max(x['first_visited_at'])-np.min(x['first_visited_at']))['first_visited_at\
']
print('Overall time engaged in studying the course by fully participating learners: mean {\
}, median {}'.format(tmp_val.mean(),tmp_val.median()))

tmp_val=tmp_unenrolledLearners[['learner_id','first_visited_at']].groupby('learner_id').ag\
gregate(lambda x: np.max(x['first_visited_at'])-np.min(x['first_visited_at']))['first_visi\
ted_at']
print('Overall time engaged in studying the course by unenrolled learners: mean {}, median\
 {}'.format(tmp_val.mean(),tmp_val.median()))
```

The Full-Learner View

We can start to tighten things up around the time spent on a course by generating a single time view over the course that includes all timestamped activity in a single list.

TO DO

```
#colour chart showing different activities
```

Scheduling Support

One of the most direct ways of supporting learners on the FutureLearn platfrom is through comment moderation, in which moderators organised by the publisher institutions keep an eye on comment threads and respond where necessary.

By investigating times of day and days of the week when there appears to be significant amounts of platform activity associated with a course, course publishers may be better able to schedule moderator support.

In particular, the `steps`, `comments` and `question` logs provide timestamp information that can be used to identify busy (or quiet) periods. A detailed study might look for correlations between the different sorts of activity, and explore differnt policies about when moderators should be online (for example, should a moderator try to encourage activity if comments are quiet but there is a lot of stop activity? Should moderators follow busy commenting periods or busy quiz periods?).

All I am interested in here is in producing a few quick tools to help us look at the data in order to spot busy and quiet periods.

Activity Reports by Day of Week and Hour of Day

To try to get a feel for when there is most activity ongoing around a course, we can generate two different sorts of macroscopic chart that use day-of-week and hour-of-day dimensions to display activity counts: *bubble matrix charts* and *heatmaps*.

The bubble matrix chart aligns a "bubble" display against day of week and hour of day with bubble size representing the activity count in that hourly period summed over all weeks contained in the dataset.

```
def activityBubbleMatrix(df,typ='comments',title=''):
    tmp=df.reset_index().set_index(tsDict[typ])
    tmp=tmp.groupby([tmp.index.dayofweek,tmp.index.hour]).size().reset_index()
    tmp.rename(columns={0:'count'},inplace=True)
    ax=tmp.plot(kind='scatter',x='level_0',y='level_1',s=tmp['count'],figsize=(15,10))
    ax.set_xlabel('')
    ax.set_ylabel('Hour of day')
    ax.set_title(title)
    ax.set_xticklabels(['','Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturd\
ay', 'Sunday'])
```

We can pass a range of activity data types into the charting function to display activity counts for each of the data types collected.

```
activityBubbleMatrix(comments,title='Number of comments by hour of day and day of week')
```

The second chart type is a heatmap where the intensity of the cell colour reflects the level of activity within the period associated with each cell.

```
def activityHeatmap(df,typ='comments',title='',label=True):
    tmp=df.reset_index().set_index(tsDict[typ])
    tmp['dow']=tmp.index.dayofweek
    tmp['hod']=tmp.index.hour
    val='id' if typ=='comments' else 'learner_id'
    tmp=tmp.pivot_table(index=['dow'], columns='hod',values=val,aggfunc='count')

    plt.rc("figure", figsize=(15, 10))
    cmap=sns.light_palette("purple", reverse=False,as_cmap=True)
    ax=sns.heatmap(tmp, annot=label, linewidths=.5,cmap=cmap,fmt='g')
    ax.set_ylabel('')
    ax.set_xlabel('Hour of Day')
    ax.set_yticklabels(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'\
, 'Sunday'][::-1])
    ax.set_title(title)
```

Again, we can pass in different activity data, such as step activity data.

```
activityHeatmap(steps,typ='steps_first',
                title='Heatmap showing comment activity by hour of day and day of week',
                label=False)
```

We can combine the separate charting functions within a single interactive function that allows us to generate each sort of activity chart for each sort of timestamped data.

```
def activityCharter(activity='comments',chartType='heatmap'):
    df=tsData[activity]
    title='{} showing activity around {}.'.format(chartType.capitalize(),activity)
    activityHeatmap(df,activity,title) if chartType=='heatmap' else activityBubbleMatrix(d\
f,activity,title)
```

The `interact()` function constructs the controls around activity and chart type selectors.

The heatmap is also configured by default to display the count values.

```
interact(activityCharter,activity=activities,chartType=['bubbleMatrix','heatmap'])
```

Note that there are problems associated with both these charts where activity levels drop of over several weeks - high levels of activity from early in the course are likely to swamp out different patterns of activity that emerge later in the course.

We could attempt to limit the date range of the data passed into the chart functions, but the day-of-week labels may be incorrect if we have a sample time period of less than a week.

As well as the macroscopic view over time of day and day of week, we can just limit the views to show summary hour-of-day or day-of-week charts. This charts provide a more glanceable take on peak days/hours, though again they are biased by hight volumne of activity days.

```
def activityHourOfDayScatter(df,typ='comments',title=''):
    df=df.reset_index().set_index(tsDict[typ])
    #Count comments by hour of day
    ax=df.groupby([df.index.hour]).size().reset_index().plot(kind='scatter',x='index',y=0,\
title=title)
    ax.set_xlabel("Hour of day")
    ax.set_ylabel("Number of {}".format(typ))
    #There must be a neater way than this?1!
    ax.set_xlim(-0.3,23.3)

def activityDayOfWeekScatter(df,typ='comments',title=''):
    df=df.reset_index().set_index(tsDict[typ])
    #Count comments by day of week
    ax=df.groupby([df.index.dayofweek]).size().reset_index().plot(kind='scatter',x='index'\
,y=0,title=title)
    ax.set_xlabel("Day of week")
    ax.set_ylabel("Number of {}".format(typ))
    #There must be a neater way than this?1!
    ax.set_xlim(-0.3,6.3)
```

To look in closer relief at the hourly or daily breakdown of a particular sort of activity, it may often be convenient to filter within a particular date range.

For example, daily activity (*Monday* is day `0`).

```
activityDayOfWeekScatter(date_limiter(steps,offsetDays(COURSE_START_DATE,3),
                                      offsetDays(COURSE_START_DATE,5),
                                      index=tsDict['steps_first']),
                         typ='steps_first', title='First visited step activity in days 3 t\
o 5 after the start of the course')
```

Alternatively, we can look at hourly breakdowns, even down to the level of within a single day, such as the first day, or across a specified date range.

For example, was there any question answering activity on the first day of the course?

```
activityHourOfDayScatter(date_limiter(qnresp,COURSE_START_DATE,COURSE_START_DATE,
                                      index=tsDict['questions']),
                         typ='questions',title='Count of question activity on first day of\
 course')
```

We can start to sketch an interactive display that helps us limit the data to a crudely specified period more easily.

```
def periodHourDayActivityChart(report,activity='enrolments',
                               start='COURSE_OPEN_REGISTRATION',
                               start_offset=0,
                               end='COURSE_OPEN_REGISTRATION',
                               end_offset=7):

    (start,end) = startEndRange(start,end,start_offset,end_offset)

    title='Count of {} by {} from {} to {}'.format(activity,report,start,end)

    df=tsData[activity]
    if report=='hourOfDay':
        ax=activityHourOfDayScatter(date_limiter(df,start,end,index=tsDict[activity]),
                                    typ=activity,title=title)
    elif report=='dayOfWeek':
        ax=activityDayOfWeekScatter(date_limiter(df,start,end,index=tsDict[activity]),
                                    typ=activity,title=title)

interact(periodHourDayActivityChart,
         report=['hourOfDay','dayOfWeek'],
         activity=activities,
         start=['COURSE_OPEN_REGISTRATION','COURSE_START_DATE','COURSE_END_DATE','COURSE_P\
RE_START_MAILINGDATE'],
         end=['COURSE_OPEN_REGISTRATION','COURSE_START_DATE','COURSE_END_DATE','COURSE_PRE\
_START_MAILINGDATE'],
         start_offset=(0,35,1),
         end_offset=(0,35,1),
        )
```

It is left to further work to normalise and aggregate the event time data from each of the data files into a single, long, tidy formatted dataframe with at the very least timestamp and event type columns.

Quiz Diagnostics

The FutureLearn question response data identifies each answer selection made by a learner, recording the quiz question number, the answer option selected, and whether this was the correct answer.

```
print('Enrolled learners who answered at least one quiz question: {:,}.'.format(len(quiz_l\
earners)))

print('Number of questions: {}.'.format(len(qnresp['qid'].unique())))
```

As a headline report, one thing we might be interested in is the typical behaviour of learners in terms of the average number of questions answered.

```
def questionCountsByLearner(group):
    return qnresp[qnresp['learner_id'].isin(group)].groupby('learner_id').apply(lambda x: \
x['qid'].nunique())

print('Mean number of questions attempted by fully particpating learners: {}'.format(np.me\
an(questionCountsByLearner(fullypart_learners))))
print('Median number of questions attempted by fully participating learners: {}'.format(np\
.median(questionCountsByLearner(fullypart_learners))))

print('Mean number of questions attempted by unenrolled learners: {}'.format(np.mean(quest\
ionCountsByLearner(unenrolled_learners))))
print('Median number of questions attempted by unenrolled learners: {}'.format(np.median(q\
uestionCountsByLearner(unenrolled_learners))))
```

Question/Answer Diagnostics

Quiz question/answer diagnostics allow us to explore how each question appears to be performing based on mass learner direct interactions with each question.

It will be useful to identify the true responses for each answer, identified using our custom question identifier, `qid`, so we can use them to highlight further reports.

```
correct_ans=sorted(qnresp[qnresp['correct']]['qidr'].unique().tolist())
```

To begin with, we might look at how many attempts are made by learners, on average, at answering each question.

Let’s start by grouping the rows according to question and learner.

```
qnresp_qgroup=qnresp.groupby(['quiz_question','learner_id']).size().reset_index().groupby(\
'quiz_question')
```

We can now chart this information using a bar plot.

```
qnresp_qgroup[0].aggregate([np.mean,np.median]).plot(kind='bar')
```

If the average number of attempts far exceeds a single attempt, we may have a problem with that question.

Another way of reviewing questions is to probe a little bit more deeply into the behaviour around each responses provided to each quiz question. For example, we can display a count of how many times each answer option was selected for each question, along with whether the answer selection was the correct answer or not:

```
qnresp.groupby(['week_number','step_number','question_number','response','correct']).size()
```

A possibly more informative report counts the total number of responses per question and then reports the percentage of this total corresponding to responses made in favour of each particular answer.

```
def quiz_pivot(df, percent=True):
    tmp = df.pivot_table(index=['week_number','step_number','question_number'],
                         columns='response',
                         values='correct',
                         aggfunc='count')
    if percent:
        return tmp.apply(lambda x : 100*(x / x.sum()), axis=1)
    return tmp

qnresp_pivot=quiz_pivot(qnresp)
qnresp_pivot
```

Summaries such as this can often be usefully visualisied using a heat map style chart. If we additionally highlight the cell corresponding to the correct answer for each question, we can get an immediate report on how well a question is performing in terms of the percentage of correct anwser options submitted as a fraction of the total number of the answers (inclduing multiple attempt answers) submitted for each question.

To make the chart easier to read, the percentage values can be dsiplayed within each cell of the heat map, along with a gradient based coloured background displaying the “heat” of each cell.

```
#Define a function to find the co-rodinates in the grid of the correct answer
def calcCoords(ans):
    x=int(ans.split('.')[-1])-1
    y=len(correct_ans)-correct_ans.index(ans)-1
    return (x,y)


def quizHeatmap(df,title='Heatmap showing percentage of answer option selection by questio\
n (correct answer highlighted)'):
    plt.rc("figure", figsize=(15, 10))
    cmap=sns.light_palette("purple", reverse=False,as_cmap=True)
    #The fmt option suppresses default label display using scientific notation
    ax=sns.heatmap(df, annot=True, linewidths=.5,cmap=cmap,fmt='g')
    ax.set_xlabel('Answer')
    ax.set_ylabel('Question')
    ax.set_title(title)
    #Highlight correct answer cells
    #http://stackoverflow.com/a/31291200/454773
    for ans in correct_ans:
        ax.add_patch(Rectangle(calcCoords(ans), 1, 1, fill=False, edgecolor='red', lw=3))
```

(I need to figure out how to limit the percent label to show at most 1 decimal place…)

```
quizHeatmap(qnresp_pivot)
```

In terms of diagnostics, look for highly selected answers that are *not* the highlighted correct answer, or questions where there are significant numbers of learners selecting across a wide range of ansswer options on a particualr question.

If learners almost universally select the same, correct answer, that question may be a bit too easy!

We can also look at quiz interaction behaviour for different groups of learners, such as counts of all question attempts by unenrolled learners:

```
quizHeatmap(quiz_pivot(qnresp[qnresp['learner_id'].isin(unenrolled_learners)],percent=Fals\
e))
```

As well we using heat maps to display the pivot table data, we can also use dodged bar charts. If bars for a particular question are of a similar height it suggests there may be some confusion as to which is the correct answer. If there are two bars of a similar height, it might suggest that the main distractor is too distracting, or alternatively that the question is acting as a strong distinguisher.

```
#For a bar chart based on percentages, simply use:
qnresp_pivot.plot(kind='bar')
```

It might be worth automating a simple text report to flag questions where the first offered answer tends to be incorrect.

Displaying Reports for `n`'th Attempted Answers on Each Question

The above reports summarise *all* responses to the quiz questions.

It might also be useful - indeed, it may be *more* useful - to report on the first, second or even third attempt quiz answers reported by each learner by filtering the data to contain just the first, second or third attempted answers by each learner before performing the count.

For convenience, let's define a couple of functions to make this sort of report a bit easier to display.

```
def quizNthAttempt(df,n=0):
    return df.sort_values('submitted_at').groupby(['learner_id','week_number','step_number\
','question_number']).nth(n).reset_index()

def quizHeatMapNthVisit(df,n=0,percent=True):
    ''' Chart the propotion of answers submitted to each option at the n'th attempt. First\
 attempt: n=0 '''
    tmp=quizNthAttempt(df,n)
    tmp=quiz_pivot(tmp, percent)

    #Need to use s/thing like https://pypi.python.org/pypi/inflect for eg 1st, 2nd, 3rd ch\
oice etc
    quizHeatmap(tmp,"Heatmap of counts for learner attempt number {} at a question (correc\
t answer highlighted)".format(n+1))
```

Now we can generate a heat map showing the percentage of submissions for each answer option on a learner's `n`'th attempt at a question. (The first attempt has index `n=0`.)

The final Boolean argument of the `quizHeatMapNthVisit()` function specifies whether we want to diaply raw count or percentage information as the cell label.

```
quizHeatMapNthVisit(qnresp,1,False)
```

Charting for higher values of `n` will display only those questions where multiple attempts were made at the question.

As well as reporting percentages, we can also use a dodged bar chart to display raw counts. This has the secondary benefit of showing volumes of attempts at each question, particularly when filtered relative to the number of `n`'th attempt answers.

Unfortunately, the dodged bar charts do not identify the correct answer option for each question, so it's hard to see whether dominant anwsers were for the correct option.

```
tmp=quizNthAttempt(qnresp,0).pivot_table(index=['week_number','step_number','question_numb\
er'],
                                     columns='response',
                                     values='correct',
                                     aggfunc='count')
tmp.plot(kind='bar')
```

A more direct visual report on the volume of activity for each step comes in the form of a stacked bar chart:

```
tmp.plot(kind='bar',stacked=True)
```

The stacked bar chart also provides us with a means of easily exploring the volume of *n*'th attempts at each particular question in the form of a graphic that identifies questions where multiple attempts are made (for whatever reason).

We could go on to investigate whether learners try incorrect answers having got the correct answer, for example, and also display counts of these attempts as a bar chart (*not explored here*).

The following chart shows how many third attempts were made across the quiz questions.

```
#For first attempt stats, set n=1; for second attempt, n=2; for M'th attempt, n=M
def i_quizNthAttemptCounts(attempt=1):
    tmp=quizNthAttempt(qnresp,attempt-1).pivot_table(index=['week_number','step_number','q\
uestion_number'],
                                                     columns='response',
                                                     values='correct',
                                                     aggfunc='count')
    tmp.plot(kind='bar',stacked=True,title='Count of attempt number {} responses to quiz q\
uestions'.format(attempt))
interact(i_quizNthAttemptCounts,attempt=(1,5,1))
```

Probing Individual Questions

Our initial headline reports gave some indication of the general behaviour around each question. If a particular question seems to be a cause of concern, how might we probe it in a little more detail?

Let's grab a sample question at random (or specify your own).

```
#Get a random sample question
sampleq=random.choice(qnresp['quiz_question'].unique())
print('Using question: {}'.format(sampleq))
#Alternatively, identify your own sample question.
#For example,for week 1, step 12, question 3, set: sampleq='1.12.3'
```

How many attempts were made at the question overall?

We have already grouped rows by question and learner as `qnresp_qgroup`, so let's build on that by selecting out the rows for a particular question.

```
ax=qnresp_qgroup.get_group(sampleq)[0].value_counts().sort_index().plot(kind='bar',
                                                                      title='Count of numb\
er of attempts for question {}'.format(sampleq))
ax.set_xlabel('Number of attempts')
ax.set_ylabel('')
```

The list of questions isn't too long, so we can create an interactive drop down list identifying each question type and bind it to a chart display to show counts against the varying number of attempts made with respect to the selected question.

```
def i_questionAttempts(sampleq):
    ax=qnresp_qgroup.get_group(sampleq)[0].value_counts().sort_index().plot(kind='bar',
                                                                      title='Count of numb\
er of attempts for question {}'.format(sampleq))
    ax.set_xlabel('Number of attempts')
    ax.set_ylabel('')
interact(i_questionAttempts, sampleq=qnresp.sort('quiz_question')['quiz_question'].unique(\
).tolist())
```

Summary

This notebook contains a variety of sketches and doodles showing how we can start to have a conversation with data from FutureLearn courses. The aim has been not to model learner behaviour or provide learner level diagnostics, preferring instead to review the mass action behaviour of learners in order to better help us understand how the course materials themselves are performing.

The notebook was put together as a recereational data activity, largely in my own time. In part, I've also used it to teach myself about how to use Jupyter notebooks' interactive widgets (I've never really used them before…)