Practical 9: Presenting Data

Visualisation and Tables

Table of contents

Complete	Part 1: Foundations	Part 2: Data	Part 3: Analysis	
90%	▯▯▯▯▯▯▯▯▯	▯▯▯▯▯▯▯	▯▯▯▯▯▯▯	9/10

> **Important**
>
> This practical focusses on the final topic we want to cover in *Foundations*: visualisation! You will have seen quite a bit of this across the preceding three to four weeks, but it was done in an *ad-hoc* way, here we try to systematise things a bit.

> **▯ Connections**
>
> Here we're trying to tidy up the loose ends. You've already worked with basic data visualisations in Seaborn and Matplotlib (including (geo)panda's plot function), but we want you to have a better sense of how that *works* as part of a coherent – if altogether rather complex and overwhelming – approach to managing a data visualisation.

1 Preamble

```
import os
import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns
```

```
import matplotlib.cm as cm
import matplotlib.pyplot as plt

import os
from requests import get
from urllib.parse import urlparse

def cache_data(src:str, dest:str) -> str:
    """Downloads and caches a remote file locally.

    The function sits between the 'read' step of a pandas or geopandas
    data frame and downloading the file from a remote location. The idea
    is that it will save it locally so that you don't need to remember to
    do so yourself. Subsequent re-reads of the file will return instantly
    rather than downloading the entire file for a second or n-th itme.

    Parameters
    ----------
    src : str
        The remote *source* for the file, any valid URL should work.
    dest : str
        The *destination* location to save the downloaded file.

    Returns
    -------
    str
        A string representing the local location of the file.
    """

    url = urlparse(src) # We assume that this is some kind of valid URL
    fn  = os.path.split(url.path)[-1] # Extract the filename
    dfn = os.path.join(dest,fn) # Destination filename

    # Check if dest+filename does *not* exist --
    # that would mean we have to download it!
    if not os.path.isfile(dfn) or os.path.getsize(dfn) < 1:

        print(f"{dfn} not found, downloading!")

        # Convert the path back into a list (without)
        # the filename -- we need to check that directories
        # exist first.
        path = os.path.split(dest)

        # Create any missing directories in dest(ination) path
        # -- os.path.join is the reverse of split (as you saw above)
        # but it doesn't work with lists... so I had to google how
        # to use the 'splat' operator! os.makedirs creates missing
        # directories in a path automatically.
        if len(path) >= 1 and path[0] != '':
            os.makedirs(os.path.join(*path), exist_ok=True)
```

```
    # Download and write the file
    with open(dfn, "wb") as file:
        response = get(src)
        file.write(response.content)

    print('Done downloading...')

else:
    print(f"Found {dfn} locally!")

return dfn
```

2 Using Maplotlib

2.1 Anatomy of a Figure

> **Tip**
>
> You might want to bookmark the 'Anatomy of a Figure' image so that you can easily find and refer to it in the future. This structure is why matplotlib is so much nastier than ggplot, but it does also give you greater *control* over the output if you really dig into the guts of things.

One of the reasons that Matplotlib is so much more complex than ggplot is that it can actually *do* a plot more than ggplot, including image manipulation, axis translation, and even 3D. You can get a sense of this by looking at the tutorials since the Users guide can be a bit overwhelming.

Nevertheless, the core components of all matplotlib figures can be seen here:

2.2 Finding Fonts

I find matplotlib's use of fonts to be *profoundly* weird. If you use conda and install directly on to the computer then you *might* have access to all of your computer's fonts (though there are different *types* of fonts as well, not all of which will show up), but for most users it will be those that were installed into Docker.

2.2.1 Using Fontconfig

fontconfig is the base Linux utility for managing fonts. We can list font using fc-list and then a set of 'switches' determining the kind of information we want back. Since fontconfig doesn't exist on OSX or Windows, you'll need to do some more investigating and poking around to get these details on a conda install (I'll show an option further down)...

Here we ask fontconfig to format the output so that we only get the first part of the family name, and then we pipe (recall | sends output from one utility to another!) the output of that to sort, which sorts the output, and uniq which removes duplicates (which there will be because there are **bold**, *italic*, small-caps, etc. versions of each font). To

make better sense of this you can always try playing around with all three steps in the output below!

```
fonts = ! fc-list --format="%{family[0]}\n" | sort | uniq
print(fonts)
```

```
['.Al Bayan PUA', '.Al Nile PUA', '.Al Tarikh PUA', '.Apple Color Emoji UI', '.Apple SD Gothic NeoI', '.Aqua Kana', '.Arial
```

> **Capturing output**
>
> Notice that we're able to capture the output of an external application (called via the Terminal) with `fonts = ! ....` This can be useful when something is easy to do on the command line but hard to do in Python.

The below option also pipes output from fonctconfig, but to the grep utility which checks each line for the character sequence Liberation. Now we're asking fontconfig to include style details which will relate to both weight (regular, bold, extra bold, light, etc.) and italic, bold, small caps, etc.

```
fonts = ! fc-list : family style | grep "Liberation"
print(sorted(fonts))
```

```
['Liberation Mono:style=Bold', 'Liberation Mono:style=Bold Italic', 'Liberation Mono:style=Italic', 'Liberation Mono:sty
```

You can find more examples here, a more detailed set of instructions here, and even information about (for example) supported languages based on RFC 3066.

Here are the languages supported by the Ubuntu Light font:

```
! fc-list "Liberation Mono" : lang
```

```
:lang=aa|af|av|ay|be|bg|bi|br|bs|ca|ce|ch|co|cs|cy|da|de|el|en|eo|es|et|eu|fi|fj|fo|fr|fur|fy|gd|gl|gv|ho|hr|hu|ia|id|ie|i
```

Here are the monospace fonts installed:

```
! fc-list :spacing=mono : family | sort | uniq
```

```
.Apple Color Emoji UI
.LastResort
.SF NS Mono
.Times LT MM
Adobe Garamond
Adobe Jenson MM
Adobe Wood Type
Amethyst
Andale Mono
Apple Color Emoji
Courier New
Fira Mono
Fira Mono,Fira Mono Medium
GB18030 Bitmap
```

Inconsolata
Inconsolata Condensed
Inconsolata Condensed,Inconsolata Condensed Black
Inconsolata Condensed,Inconsolata Condensed ExtraBold
Inconsolata Condensed,Inconsolata Condensed ExtraLight
Inconsolata Condensed,Inconsolata Condensed Light
Inconsolata Condensed,Inconsolata Condensed Medium
Inconsolata Condensed,Inconsolata Condensed SemiBold
Inconsolata Expanded
Inconsolata Expanded,Inconsolata Expanded Black
Inconsolata Expanded,Inconsolata Expanded ExtraBold
Inconsolata Expanded,Inconsolata Expanded ExtraLight
Inconsolata Expanded,Inconsolata Expanded Light
Inconsolata Expanded,Inconsolata Expanded Medium
Inconsolata Expanded,Inconsolata Expanded SemiBold
Inconsolata ExtraCondensed
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed Black
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed ExtraBold
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed ExtraLight
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed Light
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed Medium
Inconsolata ExtraCondensed,Inconsolata ExtraCondensed SemiBold
Inconsolata ExtraExpanded
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded Black
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded ExtraBold
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded ExtraLight
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded Light
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded Medium
Inconsolata ExtraExpanded,Inconsolata ExtraExpanded SemiBold
Inconsolata SemiCondensed
Inconsolata SemiCondensed,Inconsolata SemiCondensed Black
Inconsolata SemiCondensed,Inconsolata SemiCondensed ExtraBold
Inconsolata SemiCondensed,Inconsolata SemiCondensed ExtraLight
Inconsolata SemiCondensed,Inconsolata SemiCondensed Light
Inconsolata SemiCondensed,Inconsolata SemiCondensed Medium
Inconsolata SemiCondensed,Inconsolata SemiCondensed SemiBold
Inconsolata SemiExpanded
Inconsolata SemiExpanded,Inconsolata SemiExpanded Black
Inconsolata SemiExpanded,Inconsolata SemiExpanded ExtraBold
Inconsolata SemiExpanded,Inconsolata SemiExpanded ExtraLight
Inconsolata SemiExpanded,Inconsolata SemiExpanded Light
Inconsolata SemiExpanded,Inconsolata SemiExpanded Medium
Inconsolata SemiExpanded,Inconsolata SemiExpanded SemiBold
Inconsolata UltraCondensed
Inconsolata UltraCondensed,Inconsolata UltraCondensed Black
Inconsolata UltraCondensed,Inconsolata UltraCondensed ExtraBold
Inconsolata UltraCondensed,Inconsolata UltraCondensed ExtraLight
Inconsolata UltraCondensed,Inconsolata UltraCondensed Light
Inconsolata UltraCondensed,Inconsolata UltraCondensed Medium
Inconsolata UltraCondensed,Inconsolata UltraCondensed SemiBold
Inconsolata UltraExpanded
Inconsolata UltraExpanded,Inconsolata UltraExpanded Black

```
Inconsolata UltraExpanded,Inconsolata UltraExpanded ExtraBold
Inconsolata UltraExpanded,Inconsolata UltraExpanded ExtraLight
Inconsolata UltraExpanded,Inconsolata UltraExpanded Light
Inconsolata UltraExpanded,Inconsolata UltraExpanded Medium
Inconsolata UltraExpanded,Inconsolata UltraExpanded SemiBold
Inconsolata,Inconsolata Black
Inconsolata,Inconsolata ExtraBold
Inconsolata,Inconsolata ExtraLight
Inconsolata,Inconsolata Light
Inconsolata,Inconsolata Medium
Inconsolata,Inconsolata SemiBold
Input Mono
Input Mono Compressed
Input Mono Compressed,InputMonoCompressed Black
Input Mono Compressed,InputMonoCompressed ExLight
Input Mono Compressed,InputMonoCompressed Light
Input Mono Compressed,InputMonoCompressed Medium
Input Mono Compressed,InputMonoCompressed Thin
Input Mono Condensed
Input Mono Condensed,InputMonoCondensed Black
Input Mono Condensed,InputMonoCondensed ExLight
Input Mono Condensed,InputMonoCondensed Light
Input Mono Condensed,InputMonoCondensed Medium
Input Mono Condensed,InputMonoCondensed Thin
Input Mono Narrow
Input Mono Narrow,InputMonoNarrow Black
Input Mono Narrow,InputMonoNarrow ExLight
Input Mono Narrow,InputMonoNarrow Light
Input Mono Narrow,InputMonoNarrow Medium
Input Mono Narrow,InputMonoNarrow Thin
Input Mono,InputMono Black
Input Mono,InputMono ExLight
Input Mono,InputMono Light
Input Mono,InputMono Medium
Input Mono,InputMono Thin
Liberation Mono
Menlo
Miguta
PT Mono
Roboto Mono
Source Code Pro,Source Code Pro ExtraLight
```

2.2.2 Using Python+Terminal

Another way to get at this information is to try asking matplotlib what fonts it *already* knows about in its cache:

```
import matplotlib
loc = matplotlib.get_cachedir()
!ls {loc}
```

```
fontlist-v330.json fontlist-v390.json
```

Hopefully you will see a list of installed fonts when you run this next block of code. See if you can make sense of what this code does!

```
fonts = !cat {loc + '/fontlist-v330.json'}
fonts = set(list(filter(lambda x:'"name"' in x, fonts)))
fonts = [x.replace('     "name": "','') for x in fonts]
print(fonts)
```

```
['STIX Two Math', 'Lucida Grande', 'Apple Chancery', 'DejaVu Sans Mono', 'Mukta Mahee', 'OS OpenMap Local', '.SF
```

```
fonts = ! fc-list : family style | grep "Ubuntu"
print(sorted(fonts))
```

```
[]
```

2.2.3 Fontdicts

Now that we know what's available, the next step is to set up some useful defaults that we can re-use across multiple plots to ensure consistency of output. The format for specifying fonts on a per-figure basis is a dictionary, so where you see fontdict in the matplotlib documentation the following should work:

Here's the example:

```
font = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 16,
        }
```

```
tfont = {'fontname':'Liberation Sans Narrow', 'weight':'bold'}
bfont = {'fontname':'Liberation Sans Narrow', 'weight':'normal', 'horizontalalignment':'left'}
afont = {'fontname':'Liberation Sans Narrow', 'weight':'light'}
```

I am setting the 'title font' (tfont) and 'body copy font' (bfont) and 'axis font' (afont) here to use in the output below. You can pick another font and see what happens.

2.2.4 2.3: Using Fonts

At this point we're going to work towards a kind of 'atlas' that would make it easy to compare some features for different London boroughs. I basically implemented a the basic matplotlib version of QGIS' Atlas functionality.

```
# This will be wahtever LA you processed last week
LA = 'Waltham Forest'
```

```
msoa_gdf = gpd.read_parquet(os.path.join('data','geo',f'{LA}-MSOA_data.geoparquet'))
```

```
median_gdf  = msoa_gdf[['MSOA11CD','median_price','geometry']]
listing_gdf = msoa_gdf[['MSOA11CD','listing_count','geometry']]
```

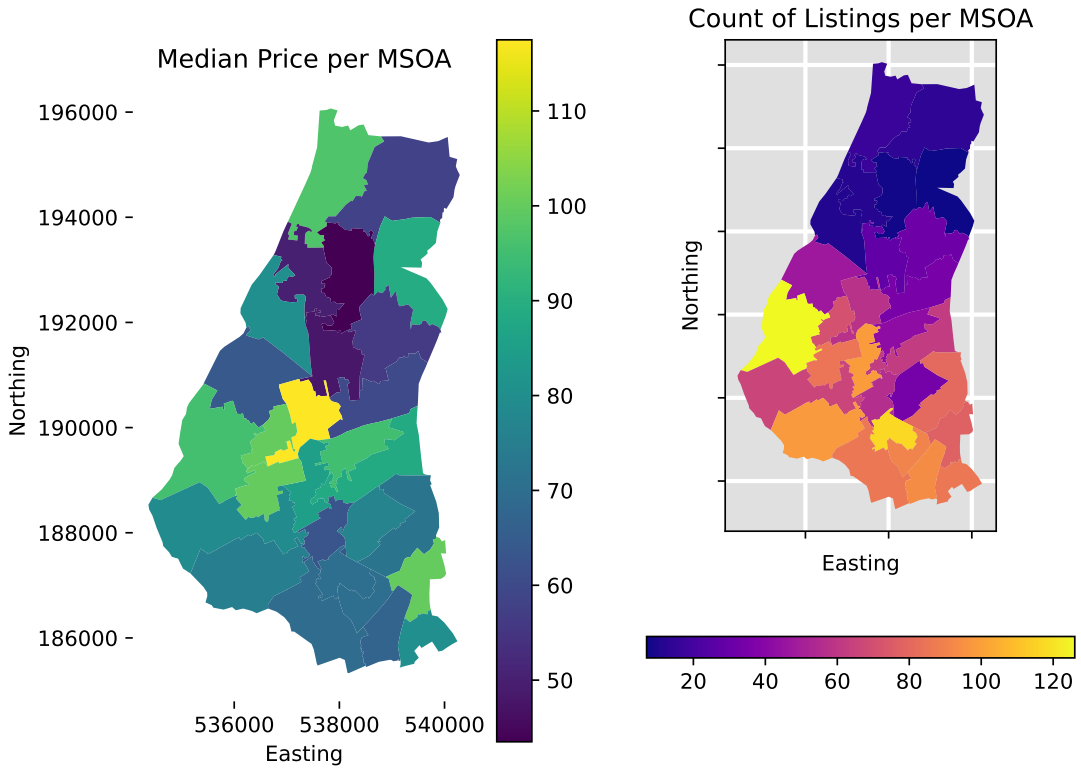
```
import matplotlib.pyplot as plt
```

2.2.5 The Defaults

Here is a demonstration of some of the ways you can adjust features in a Python matplotlib plot. I'm not suggesting either of these is a *good* output, but that's not the point! The idea is to see the various ways you can tweak a plot… And notice that we've not yet changed any fonts. And it shows.

```
# Set up a 1 x 2 plot (you can also leave off the nrows= and ncols=)
f,axes = plt.subplots(nrows=1, ncols=2, figsize=(8,6))
# ax1 will be the first plot on the left, ax2 will be on the right;
# a 2 (or more) *row* plot will return a list of lists... 1 list/row.
ax1 = axes[0]
ax2 = axes[1]

# Left plot is the median price
median_gdf.plot(column='median_price', ax=ax1, legend=True, cmap='viridis')
ax1.set_title("Median Price per MSOA");
# Turn off the frame, one side of the plat at a time
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)
ax1.spines['bottom'].set_visible(False)
ax1.spines['left'].set_visible(False)
# Set the labels
ax1.set_xlabel("Easting");
ax1.set_ylabel("Northing");

# Right plot is the number of listings; note
# here the use of both zorder (which is the
# 'stacking order' of elements on the plot, and
# the legend_kwds (keywords) to change the
# orientation of the plot to horizontal
listing_gdf.plot(column='listing_count', ax=ax2, legend=True, cmap='plasma', zorder=1,
            legend_kwds={"orientation": "horizontal"})
ax2.set_title("Count of Listings per MSOA");
# Set a background colour for the plot
ax2.set_facecolor((.4, .4, .4, .2))
# Add grid lines and set their zorder to
# below that of the data on the plot
plt.grid(visible=True, which='major', axis='both', color='w', linestyle='-', linewidth=2, zorder=0)
ax2.set_axisbelow(True)

# This is equivalent to the ax1.spines...
# above, but if you use it here you lose
# the background to the plot as well!
#plt.gca().set(frame_on=False)

# Remove the labels on the ticks of the
# axes (meaning: remove the numbers on
# x- and y-axes).
ax2.set_xticklabels([])
ax2.set_yticklabels([])
```

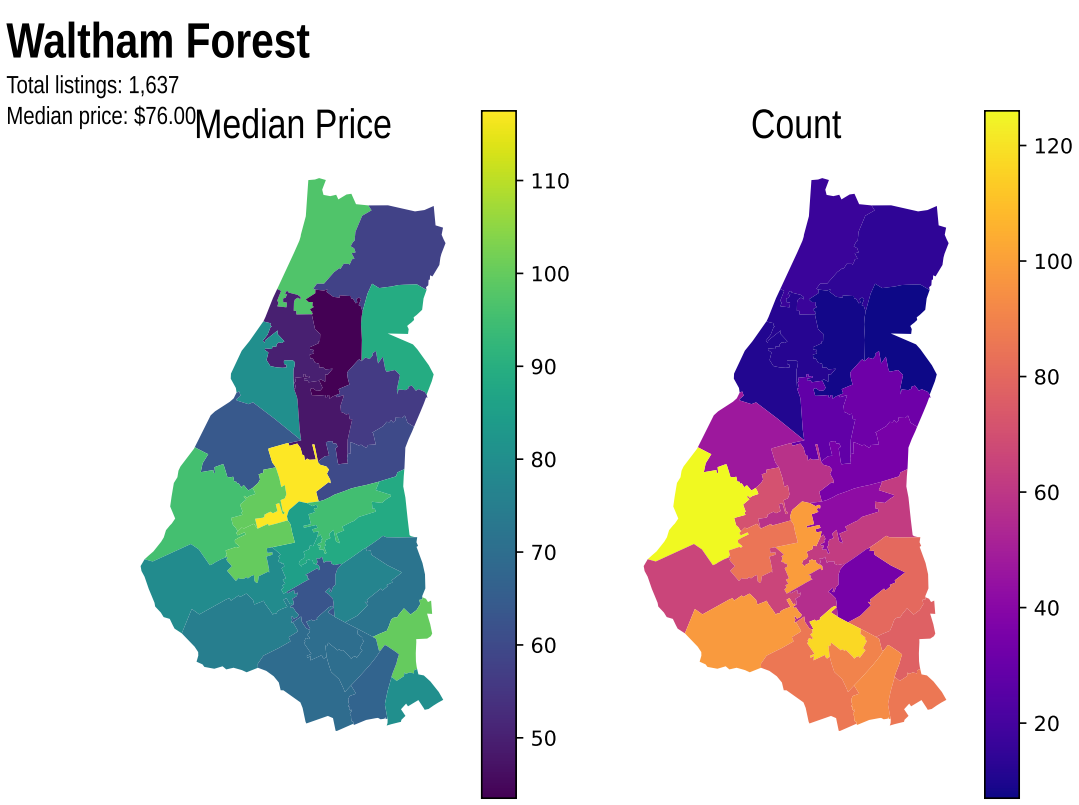
```
# Set the labels
ax2.set_xlabel("Easting");
ax2.set_ylabel("Northing");
```



2.2.6 Improving on Defaults

```
f,axes = plt.subplots(1,2,figsize=(8,6))

# Set up the plots
median_gdf.plot(column='median_price', ax=axes[0], legend=True, cmap='viridis')
listing_gdf.plot(column='listing_count', ax=axes[1], legend=True, cmap='plasma')
for ax in axes:
    ax.axis('off')
    # Note that here, set_facebolor doesn't work,
    # presumably because the axis is 'off'
    ax.set_facecolor((.4, .4, .4, .2))

# Add the 'super-title', but notice that it is not
# longer either centered (x=0.025) or centre-aligned
# (horizonal alignment=left). We also see **tfont, which
# is a way of expading the 'tfont' dictionary into a
# set of parameters to a function call. We do the same
# for the titles on each figure, but passing a different
# fontdict.
f.suptitle(LA, x=0.025, ha='left', size=24, **tfont)
axes[0].set_title('Median Price', size=20, **afont)
axes[1].set_title('Count', size=20, **afont)
```

```
# And add a short piece of text below the borough
plt.figtext(x=0.025, y=0.92, linespacing=1.4, va='top', size=12,
        s=f"Total listings: {listing_gdf['listing_count'].sum():,.0f}\nMedian price: ${median_gdf['median_price'].medi
```



3 Create an Atlas

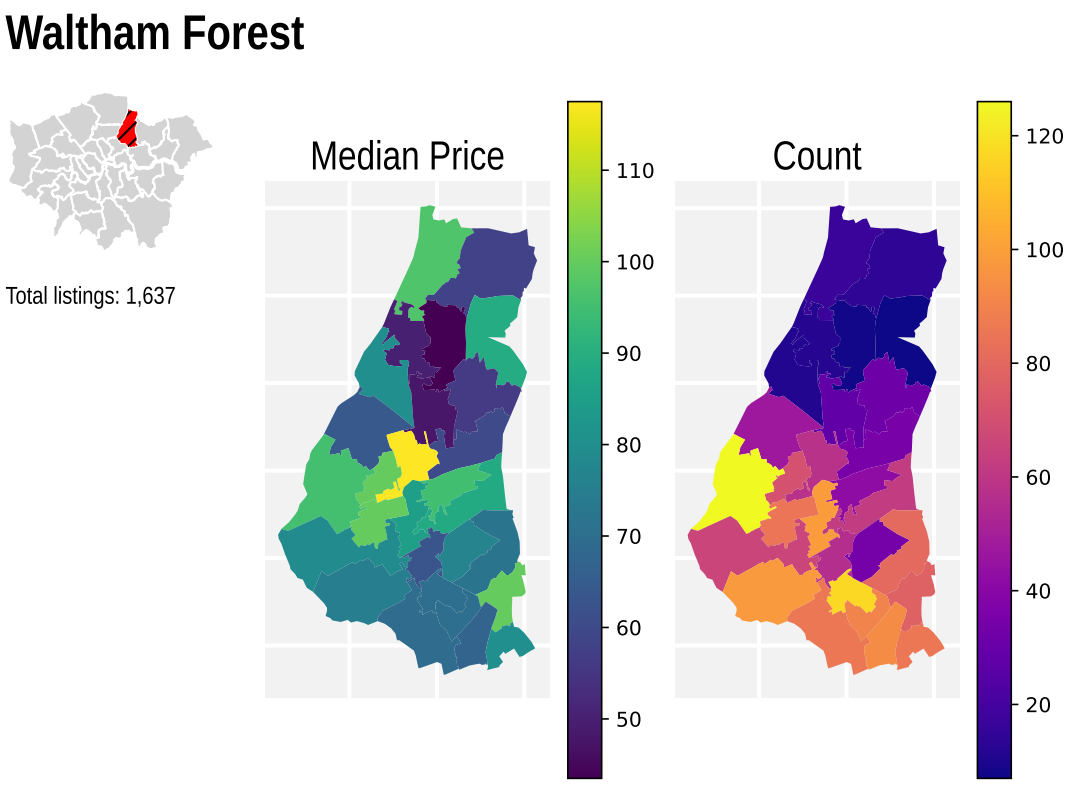
3.1 Adding Picture-in-Picture

We're now going to emulate a *bit* of QGIS' Atlas function by creating two subplots and then adding a *third* plot afterwards that shows where the borough is.

```
f,axes = plt.subplots(1,3,gridspec_kw={'width_ratios':[1,4,4]}, figsize=(8,6))

# Plot 0 is basically being used as a 'spacer'
# as you'll see below
axes[0].axis('off')

# Plot 1 is the median price
median_gdf.plot(column='median_price', ax=axes[1], legend=True, cmap='viridis')
axes[1].set_title('Median Price', size=20, **afont)

# Plot 2 is the count of listings
listing_gdf.plot(column='listing_count', ax=axes[2], legend=True, cmap='plasma')
axes[2].set_title('Count', size=20, **afont)

# For plots 1 and 2... if you were doing this a lot it could be a function!
for ax in axes[1:]:
```

```
    ax.set_facecolor((.9, .9, .9, .5))
    ax.grid(visible=True, which='major', axis='both', color='w', linestyle='-', linewidth=2, zorder=0)
    ax.set_axisbelow(True)
    ax.spines['top'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.tick_params(axis='both', which='both', length=0)

# Add a *third* chart that we use as a kind of 'PiP'
# to show which borough we're talking about. The
# add_axes call is here taking information about the
# positioning and size of the additional figure.
# Disable ax2.axis('off') if you want to see the
# figure in full.
ax3 = f.add_axes([0.015, 0.7, 0.2, 0.2])
spath = 'https://github.com/jreades/fsds/blob/master/data/src/' # source path
ddir  = os.path.join('data','geo') # destination directory
boros = gpd.read_file( cache_data(spath+'Boroughs.gpkg?raw=true', ddir) )
boros.plot(facecolor='lightgrey', edgecolor='w', linewidth=1, ax=ax3)
boros[boros.NAME==LA].plot(facecolor='r', edgecolor='none', hatch='///', ax=ax3)
ax3.axis('off')

# Add the 'super-title', but notice that it is not
# longer either centered (x=0.025) or centre-aligned
# (horizonal alignment=left). We also see **tfont, which
# is a way of expanding the 'tfont' dictionary into a
# set of parameters to a function call. We do the same
# for the titles on each figure, but passing a different
# fontdict.
f.suptitle(LA, x=0.025, ha='left', size=24, **tfont)

# And add a short piece of text below the borough
plt.figtext(x=0.025, y=0.65, s=f"Total listings: {listing_gdf['listing_count'].sum():,.0f}", size=12, **bfont);
```

```
Found data/geo/Boroughs.gpkg locally!
```



3.2 Bonus Achievement Unlocked!

If you have the time and inclination, see if you can convert the above to an *actual* atlas output:

1. You'll want to turn this plot into a function so as to be able to produce (and save) the map for *every* borough.
2. You'll even need to parameterise the filename so that you save to *different* PNG files as well as going back to see how we generated the listing and pricing data frames for the Local Authority...
3. And you'll *also* need to make sure that you ensure a consistent colour bar (for all of London, because the median price and number of listings will vary rather a lot by LA)
4. Then there's the placement of the PiP for some boroughs with long names
5. And finally, you might consider adding some more text to atlas–maybe pull some content from Wikipedia using Beautiful Soup (bs4)?

4 Think Text!

I also wanted to draw your attention to this outstanding piece on using text effectively in data visualisation: we often add labels as afterthoughts without too much regard for where they go or how they look; however, getting the content, positioning, size, and even font/font-weight 'right' can make all the difference to the effectiveness of your chart! The illustrations are top-notch.

And see the bibliography at the end!

> **Bookmark Me!**
>
> Basically, bookmark this blog post and refer to it every time you are making a map or chart.

5 Using Bokeh

Bokeh can do a *lot* more than this, but I just wanted to give you a flavour of the other visualisation tools supported by Python. This obviously works *very* differently in setup and use.

```
gdf_la = gpd.read_parquet(os.path.join('data','geo','Listings_with_LA.geoparquet'))
msoas  = gpd.read_parquet(os.path.join('data','geo','London_MSOA_Names.geoparquet'))
```

5.1 For a Chart

Group the listings by Borough and Room Type, and aggregate by median price, also producing a count variable for the number of listings of each type in each Borough.

```
la_tots = gdf_la[gdf_la.NAME==LA].groupby(by='room_type', observed=False).agg(
                    {'price':'median', 'listing_url':'count'}
                ).reset_index().rename(columns={'listing_url':'count'})
la_tots
```

	room_type	price	count
0	Entire home/apt	117.0	946
1	Hotel room	NaN	0
2	Private room	47.0	687
3	Shared room	24.5	6

```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure
from bokeh.models import ColumnDataSource, HoverTool
from bokeh.palettes import Spectral4
from bokeh.models import CustomJS, Dropdown

output_notebook()

room_types = la_tots.room_type.to_list()
prices     = la_tots.price.to_list()
counts     = la_tots['count'].to_list()

# Add hover tooltip
source = ColumnDataSource(data=dict(
   rt=room_types,
   count=counts,
   price=prices,
))
```

```
TOOLTIPS = [
   ("Room Type", "@rt"),
   ("Number of Listings", "@count{,}"),
   ("Median Price", "$@price{,}/night")
]

p = figure(x_range=room_types, height=300, tooltips=TOOLTIPS,
        title=f"Median Price by Room Type in {LA}",
        toolbar_location=None, tools="")

p.vbar(x='rt', top='count', width=0.9, source=source)
p.xgrid.grid_line_color = None
p.y_range.start = 0

show(p)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_load.v0+json
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_exec.v0+json
```

5.2 For a Map

This is not the prettiest code, but it should work…

```
from bokeh.plotting import figure

from bokeh.io import output_file, show, output_notebook, push_notebook, export_png
from bokeh.models import ColumnDataSource, GeoJSONDataSource, LinearColorMapper, ColorBar, HoverTool
from bokeh.plotting import figure
from bokeh.palettes import brewer

#output_notebook()

msoadf = gpd.sjoin(
        gdf_la[gdf_la.NAME==LA].reset_index(),
        msoas[msoas.Borough==LA].drop(columns=['index_right']), predicate='within')

msoagrdf = msoadf.groupby('MSOA11NM').agg({'price':['median','count']}).reset_index()
msoagrdf.columns=['msoa11nm','median','count']
```

I cobbled the mapping functions below together from two tutorials I found online (this one and this one). As you can see, this is a very different approach to mapping data, but it has clear benefits for exploratory purposes and produces fast, interactive maps… and I've not even added selection and filtering tools!

```
import json

def get_geodatasource(gdf):
    """Get getjsondatasource from geopandas object"""
    json_data = json.dumps(json.loads(gdf.to_json()))
    return GeoJSONDataSource(geojson = json_data)

def bokeh_plot_map(gdf, column=None, title=''):
    """Plot bokeh map from GeoJSONDataSource """

    geosource = get_geodatasource(gdf)
    palette = brewer['OrRd'][8]
    palette = palette[::-1]
    vals = gdf[column]

    #Instantiate LinearColorMapper that linearly maps numbers in a range, into a sequence of colors.
    color_mapper = LinearColorMapper(palette=palette, low=vals.min(), high=vals.max())
    color_bar = ColorBar(color_mapper=color_mapper, label_standoff=8, width=500, height=10,
                         location=(0,0), orientation='horizontal')

    tools = 'wheel_zoom,pan,reset,hover'

    p = figure(title = title, height=700, width=850, toolbar_location='right', tools=tools)
    p.add_tile("CartoDB Positron", retina=True)
    p.xgrid.grid_line_color = None
    p.ygrid.grid_line_color = None

    # Add patch renderer to figure
    p.patches('xs','ys', source=geosource, fill_alpha=0.5, line_width=0.5, line_color='white',
              fill_color={'field' :column , 'transform': color_mapper})

    # Specify figure layout.
    p.add_layout(color_bar, 'below')

    # Add hover
    hover = p.select_one(HoverTool)
    hover.point_policy = "follow_mouse"
    hover.tooltips = [("Borough", "@Borough"),
                      ("Neighbourhood", "@msoa11hclnm"),
                      ("Count of Listings", "@count"),
                      ("Median Price", "$@median")]

    return p
```

Reproject to Web Mercator:

```
msoa_gdf = pd.merge(msoagrdf, msoas, left_on='msoa11nm', right_on='MSOA11NM', how='inner')
msoa_gdf = msoa_gdf.set_geometry('geometry').set_crs('epsg:27700')

msoageo = msoa_gdf.to_crs('epsg:3785')
msoageo.total_bounds

array([-6.74542047e+03,  6.71906611e+06,  3.04361304e+03,  6.73637453e+06])
```

And map it!

```
p = bokeh_plot_map(msoageo, 'median', title=f'MSOA-Level Activity in {LA}')

handle = show(p, notebook_handle=True)
push_notebook(handle=handle)
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_exec.v0+json

> ℹ Connections
>
> And that's it. That's all she wrote! You've now covered in <10 weeks what many people might take 10 *months* to cover. So do not feel like either: 1) you know it all; or 2) you know nothing. You have learned a *lot*, but it's probably just enough to see how much you *don't* know. And *that* is the start of wisdom. Good luck, young Python-master!