

Python for Data Analysis

Tutorial by Iza Romanowska, University of Southampton

1. Installing Python

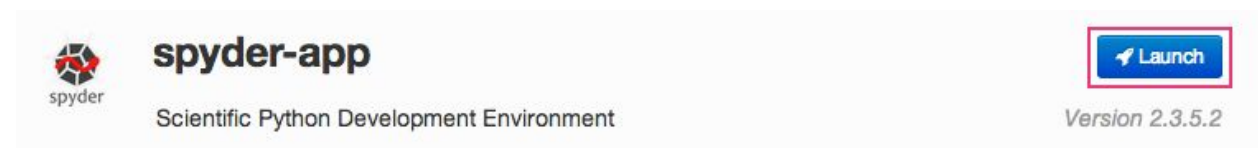
Installing Python and its libraries 'by hand' can be tedious. However, you can easily bypass the difficulties by installing one of the scientific distributions of Python, which contain most of the packages you will need. The two most popular ones are [Anaconda](#) and [Enthought Canopy](#). The former is free and open source, the latter is a paid service but you can get a free academic licence if you have a university email (it also gives you access to a large library of video tutorials). We recommend to install Anaconda (follow these [instructions](#)) but if you are working on Enthought Canopy, go to the last page for a list of workarounds.

If you prefer to custom install Python (keep in mind that it comes preinstalled on Mac OS X) you can download the latest version from [here](#) or follow this user-friendly [tutorial](#) on how to install all the necessary tools from the command line. If you are working on Windows, follow these [instructions](#).

Not sure where to find the 'command line'?

- Windows:
 - a. open the windows menu,
 - b. type 'cmd' in the search box
 - c. click on the top result
- Mac:
 - a. hit cmd+shift
 - b. type 'terminal'
 - c. click on the top result

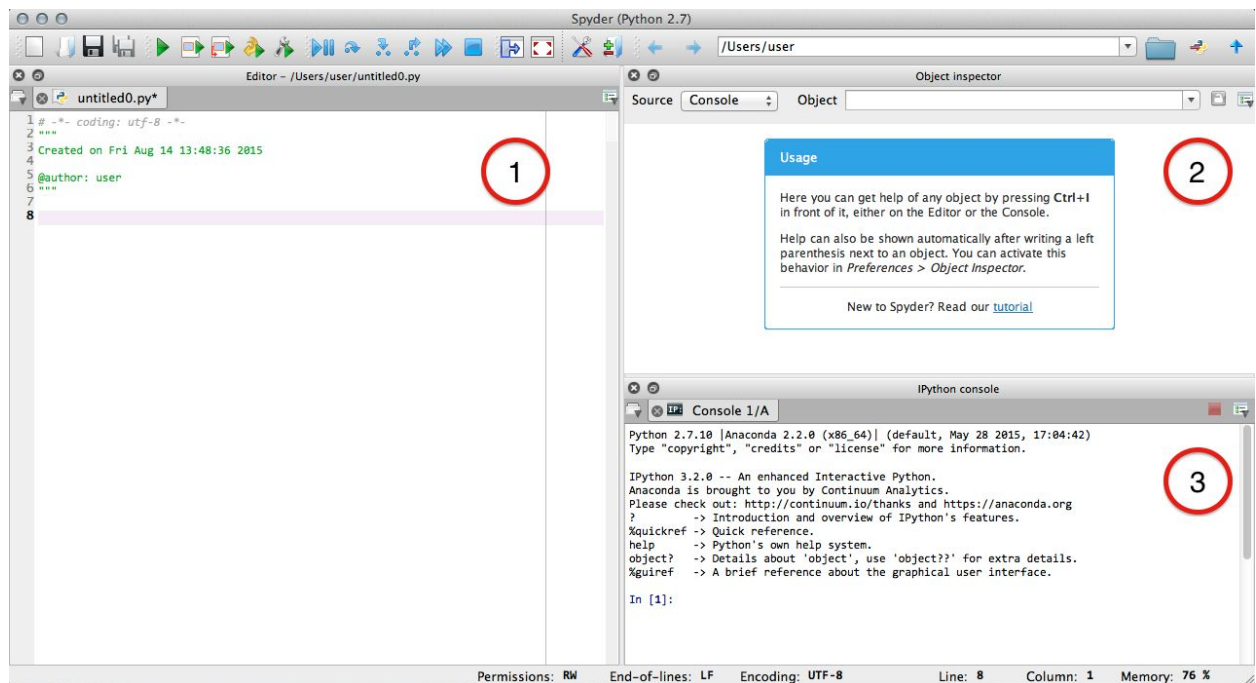
The examples in this tutorial were developed using Spyder (**S**cientific **P**ython **D**evelopment **E**nvi**R**onment), i.e., Anaconda's IDE (Interactive Development Environment - jargon for: 'user-friendly interface'). To open Spyder, either type: `>>> spyder` in your command prompt/terminal or find the [Launcher](#) app. On Windows, it is in the start menu. On Mac it is in the anaconda folder (if you are not sure where the folder is: cmd+shift -> type: 'anaconda' and choose the top result). Click the [Launch](#) button in the spyder-app box (Fig. 1).



The folder you were given contains all the datasets used in the tutorial. We recommend you set it as your working directory (instructions below). Complete Python scripts of each section can also be found in this folder. However, the tutorial has been developed for archaeologists with very limited or no previous coding experience, so you should be able to follow it without the need to check the scripts. We recommend using them only as a backup option if you are really stuck.

2. The Python Interactive Development Environment - Spyder

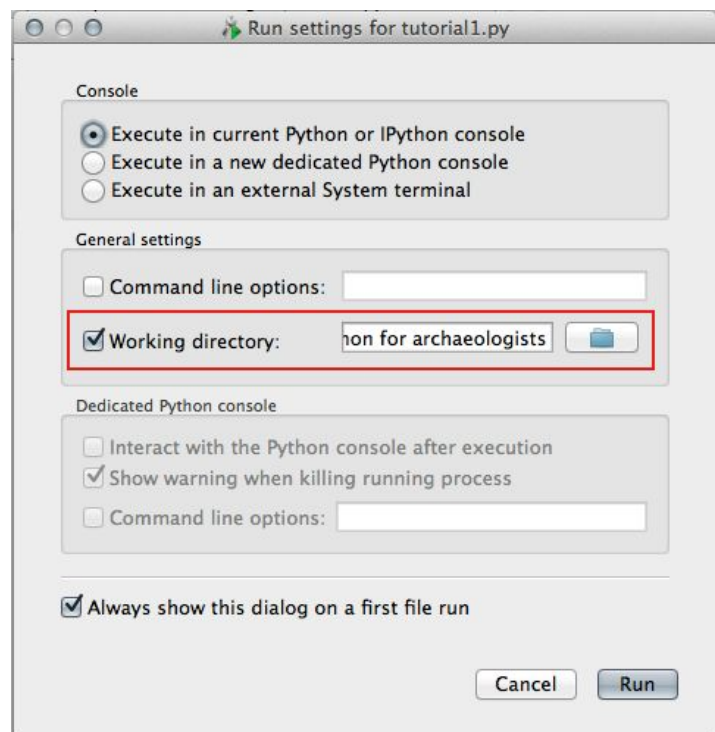
This exercise will allow you to familiarise yourself with the Python IDE - Spyder and the basic workflow of developing scripts. The default screen of Spyder consists of the **editor** (1), the **object inspector** (2), and the IPython **console** (3) (Figure 2).



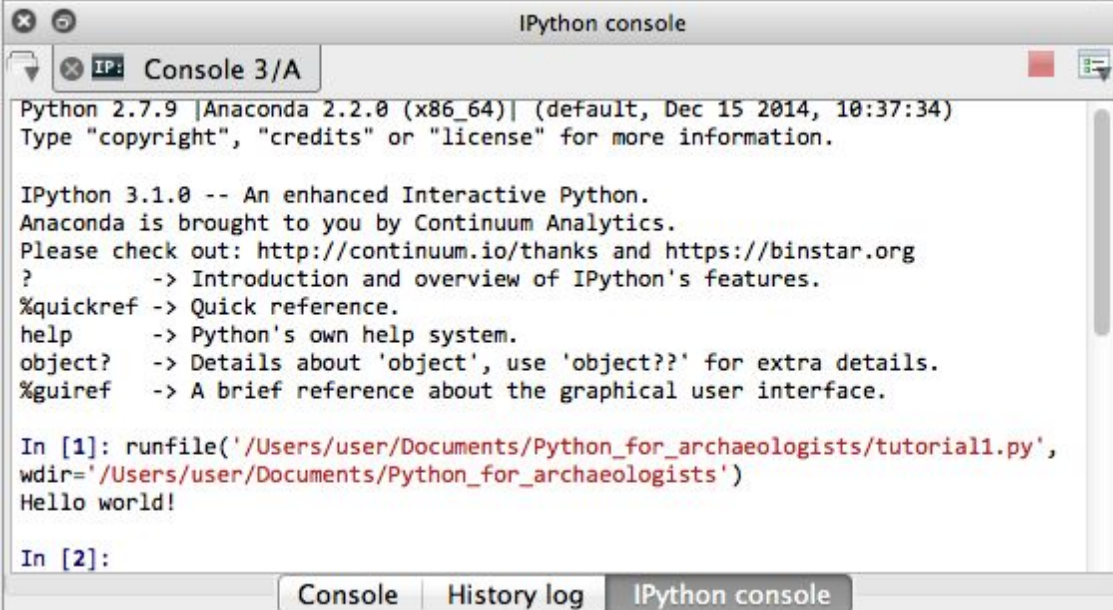
The editor pane (1) is where you type the Python script. Click anywhere within the editor (1) and type:

```
print ("Hello world!")
```

and hit F5 (windows) or fn+F5 (mac). Spyder will prompt you to save the script. It is best practice to keep each data analysis project in a separate folder and to always give it a meaningful name, e.g, 'SiteXXX_Lithic'. Create a 'Python_DataAnalysis' folder or navigate to an already existing one and save the script there. A prompt will appear (Figure 3) asking to specify the working directory. A working directory is a folder on your computer, which Python uses for all input/output related processes - such as saving figures or looking for data files. Python script will only search for input and save output files in the same folder where it is saved (otherwise you will have to specify their path each time).



Check that the 'Working directory:' box shows the project folder. If that is the case click on the 'Run' button. The Spyder console (Figure 2, circle 3) should show the same output as in Figure 4.



```
Python 2.7.9 [Anaconda 2.2.0 (x86_64)] (default, Dec 15 2014, 10:37:34)
Type "copyright", "credits" or "license" for more information.

IPython 3.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%guieref -> A brief reference about the graphical user interface.

In [1]: runfile('/Users/user/Documents/Python_for_archaeologists/tutorial1.py',
wdir='/Users/user/Documents/Python_for_archaeologists')
Hello world!

In [2]:
```

The first two lines tell you that a python script called 'tutorial1.py' was executed from the specified working directory - `wdir`. The final line is the output of the executed script, in this case the phrase 'Hello world!' was printed on the screen.

Congratulations you have successfully run your first Python script!

Now we will make this script better. Type in the editor (pay attention to the indentation):

```
def printer ():
    """ Prints the words 'Hello world!' """
    print ("Hello world!")          # printing function
printer ()
```

In the first line we define (using the keyword `def`) the function and give it the arguments (we will come back to this in a moment). The second line, delimited with triple quotation marks, is used to annotate the code with a short summary of the function. Note that in Python the indentation is important and the program will not run correctly if you do not indent the lines inside the function. The next line is the core of the function - this is the print statement that Python will read and execute. You can see that we have inserted a comment following the hash symbol (`# printing function`). This may look redundant in such a simple example, but documenting your code is important and, in the long run, will save you and others time so it is worth getting used to doing it no matter how simple the script. Finally, the last line calls Python to execute the code. Run the code (F5 or `fn+F5`) and see what happens.

Now that we have defined a function we can also run it directly from the console. Type `printer()` next to the `In [4]:` in the console window (the number may differ depending

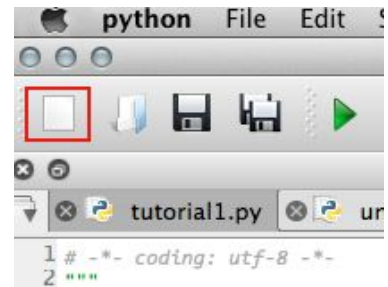
how many times you have run the script previously) and hit enter. The result should look like this:

```
In [4]: printer()  
Hello world!
```

This may not look terribly exciting or much different to the previous output but once you have defined a function you enter a new realm of possibilities.

Open a new script window by clicking on the white square icon (Figure 5) or hit ctrl/cmd+N and type:

```
def print_sum(x, y):  
    """prints a sum of two arguments"""  
    sum = x + y  
    print(sum)    # printing function
```



Save and run the script. You can now pass arguments ('x' and 'y') to the 'printer' command (1st line) which sums them (3rd line) and prints the result on the screen (4th line). Type in the console `print_sum(2, 3)` and hit enter. If the result is 5 then, well done, you have successfully turned your computer into a simple calculator.

Two final notes. The 'help' function is a useful tool to quickly check any object. Type in the console window `help(printer)` and you will see your own documentation. Second, try removing one of the parentheses in your script. A red warning sign appears in the left hand side panel. This is the 'debugger' software, which looks for mistakes in the code. If you hover your mouse over the sign, it will tell you what type of error it has detected.

3. Data formats and reading the data

Most large-scale data storage is done in .txt or .csv files¹. To read the contents of a file, we open it and create a copy assigned to a variable. At this stage it is a good idea to take a peek into the data to see how it is structured. Download the file 'Acheulean.csv' and place it in the project folder. Create a new script and run it (make sure you save it in the same working folder) or type in the console:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
Acheul_Afr = pd.read_csv('Acheulean.csv')
print Acheul_Afr.head()
```

In the first line we import the Pandas, Numpy and Matplotlib libraries using the 'pd', 'np' and 'plt'² acronyms respectively. They will be used as suffixes at the beginning of all functions that come from the libraries, see for example: `pd.read_csv()` a few lines later. The csv reader function assigns the contents of the file to a variable called `Acheul_Afr` (it does not matter how you name the variable, but we strongly recommend to use a short but meaningful word). The following lines print the top few lines of the data using the `head()` function. If you run the script you should see the following output (Figure 6).

```
In [7]: runfile('/Users/user/Documents/Python_for_archaeologists/reading_file.py',
wdir='/Users/user/Documents/Python_for_archaeologists')
   Unnamed: 0   Lat   Long   HA   CL   KN   FS   D   CS   P   CH   SP   OLT   SS   \
0  Olorgesailie -1.58  36.45  197   96  58  17   5   11   3  32  52   6  213
1      Isimila -7.90  35.61  246  208  30  28   6   30  16  62  17  15   98
2  Kalambo Falls -8.60  31.24  337  264  59  96   8  124  18  69   6  17  303
```

You may have noticed that as you were typing, a yellow window popped up (Figure 7) with a list of arguments the function takes and their default values (if you have not - try typing the `Acheul_Afr = pd.read_csv('Acheulean.csv')` line again, the yellow window will appear as you reach the opening parenthesis).

A few of them are quite useful to know when reading in the file. For example, the argument `header` is used to specify that one of the rows of the data consists of the column labels and should be omitted from any calculations. Setting it to `header = 0` will ensure that the first row of data is used as column names³. In some cases the data was saved with a different delimiter. This can be rectified using the `sep` argument. For example, if a semicolon was used instead of a comma in a .csv file `sep = ';'` will allow the function to read the file correctly. Data files often start with extra information, metadata, or simply the name of the dataset - none of which will be used in the analysis. For these situations `read_csv()` has a really useful argument - `skiprows`. Thus, `skiprows = 5`, will omit the first 5 rows allowing you to only load the actual data. To change the arguments' default values, simply

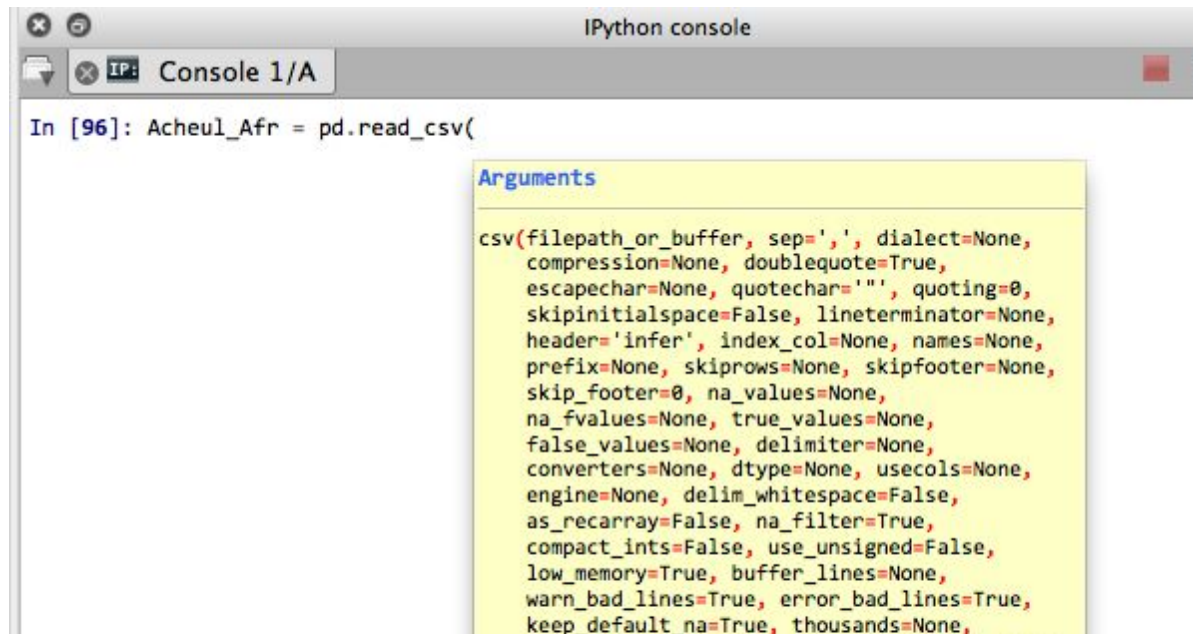
¹ The difference between the two is the use of the space (.txt) or the comma (.csv) as a delimiter.

² Although you can pick any acronym, these are conventionally used.

³ Why '0' and not '1'? Python start counting from 0 hence the first line is the line number 0, the second - number 1, etc.

list them in the parentheses following the function. For example, we can specify that the first line of our data consists of labels.

```
Acheul_Afr = pd.read_csv('Acheulean.csv', header = 0)
```



Finally, once tidy it is always good to get a general idea of what the data looks like. Add the following to your script and run it again:

```
plt.style.use('ggplot')
Acheul_Afr.hist(figsize = (10,10))
```

You should get a compiled figure of all the data represented as histograms⁴. This is a quick way of checking what type of values you are dealing with and whether they are normally distributed. After we have manipulated the data we may want to save it into a file. Type in the console:

```
Location_output = 'Acheul_Afr.csv'
Acheul_Afr.to_csv(Location_output)
```

A new file should appear in your working directory folder. Use `Location_output = '/Users/.../Acheul_Afr.csv'` (where '...' indicate the path of the destination folder) if you want the file to be saved in another folder that is not the working directory.

⁴ If you are using Enthought Canopy add the following line at the end: `plt.show()`.

4. Tidying data

The data imported into Python is stored in data frames. Data frames consist of a type of object specific to Pandas called series⁵. The data frame is like a whole Excel spreadsheet containing all the data read into Python. A series, on the other hand, represents a column of data and the associated index: 0, 1, 2, etc. (Figure 8).

Index	Data
0	-1.58
1	-7.90
2	-8.60
3	-19.92
4	-0.45
5	-14.43

This means you can access any data point by referring to its column and row. However, this also means that keeping the data tidy is crucial to ensure that statistical operations are done on the correct subset of data. The golden rule is that each column should represent a variable (e.g., latitude, number of handaxes on a site, presence or absence of a hearth) and each row contains one observation (e.g., a site, an excavation square, one feature).

The following few exercises will allow you to familiarise yourself with tidying and subsetting data. Download 'Ologresailie.csv' and read it in as `Ologresailie`. The file contains counts of different types of stone artefacts recovered from 19 Palaeolithic localities in the Ologresailie Basin, Kenya.

The data in the file is rather untidy - it could be converted into a more readable format (from long to wide), some of the values are clearly erroneous and some of the fields contain non numerical values. In the next few steps we will tidy this messy data up.

4.1 Reshaping and pivoting data frames

Let's start with reshaping the data from long to wide. You can type all of the following code snippets either in the script or in the console:

```
Ologresailie = Ologresailie.pivot(index = 'Locality', columns = 'attribute', values = 'value')
```

Use the `head()` method to check the data. You can see that each column represents a variable and each row consists of one observation. The columns include information on the stratigraphic units, a general count of stone tool categories (e.g. 'Large cutting tools') and the count of particular types (e.g., 'HA' for Handaxes).

Because in the 'long' version the columns contained both numeric values and strings, python converted everything into 'objects'. Now that we have everything in columns, we need to convert all numeric values into numeric values: integers or floats; otherwise it will be impossible to do any calculations.

⁵ You can check the type of object you are dealing with by using the `type` function. Type into the console: `type(Acheul_Afr)` and `type(Acheul_Afr["Lat"])` to see the different types of objects we were manipulating in the previous section.

```
Olorgesailie = Olorgesailie.convert_objects(convert_numeric=True)
```

To check if it worked you can use a very handy function `info()` which tells you what type of data is present in each column. Type in the console: `Olorgesailie.info()`

At the moment the columns are in alphabetical order but we can tidy it up so that the strata information will come first followed by the categories and then tool types. Let's get the names of the columns first, type:

```
list(Olorgesailie.columns.values)
```

Let's reorder the list.

```
cols = ['Level', 'Strata', 'Heavy.duty.tools', 'Large.cutting.tools', 'Large.scrapers',
        'Other.large.tools', 'Small.tools', 'Spheroids', 'BLCT', 'CB', 'CH', 'CHA', 'CL', 'CS',
        'HA', 'KN', 'LFS', 'OLT', 'OST', 'PAT', 'PHA', 'SP', 'SSNP', 'SSS']
```

```
Olorgesailie = Olorgesailie[cols]
```

You can check how the data looks now with the `head()` function.

attribute	
Level	0
Strata	0
Heavy.duty.tools	0
Large.cutting.tools	0
Large.scrapers	0
Other.large.tools	1
Small.tools	0
Spheroids	0
BLCT	0
CB	0
CH	0
CHA	0
CL	0
CS	0
HA	1
KN	0
LFS	0
OLT	0
OST	0
PAT	0
PHA	0
SP	0
SSNP	0
SSS	2
dtype:	int64

4.2 Missing data

Most of datasets have some missing data. These are usually represented as NaN (Not a Number) value. To check if the data contains any missing data, type:

```
print Olorgesailie.isnull().sum()
```

This should produce an output similar to Figure 9. The number of NaNs is given per column. Apparently we have four missing values in the in the 'Other.large.tools', 'HA' and 'SSS' columns. If you want to check which rows in the column contain NaNs, type:

```
print Olorgesailie[np.isnan(Olorgesailie['HA'])]
```

There are several ways of dealing with missing values. The easiest (but also the most problematic) is to remove observations which contain missing values. Type:

```
Olorgesailie_noNaNs = Olorgesailie.dropna(axis = 0, how =
'any')
```

and run:

```
print Olorgesailie_noNaNs.isnull().sum()
```

There should be no more missing values in the table, but we have also removed four rows of data with perfectly fine values that could be used in other tests. An alternative to removing whole rows is to replace all missing values with 0 using the `fillna()` function. Run the check again, but saving it as `Olorgesailie_NaN0`:


```
Olongesailie_NaN0 = Olongesailie.fillna(0)
```

However, this method is not ideal either - in practice we now have a fake value in fields where the real value was not known. This can skew some results. For example, try:

```
Olongesailie['Other.large.tools'].dropna().mean()
```

and

```
Olongesailie.fillna(0.0)['Other.large.tools'].mean()
```

The results are different. Why? This is because to calculate the mean all values are summed and then divided by the number of values. In the first method the NaNs are omitted, so the sum of all values is divided by the number of values excluding the NaNs. If you replace them with zero, like in the second method, they are summed up with the rest so the sum of values is divided by a higher number - as it includes the NaNs. Hence the difference in results.

The most common statistical tests and methods are usually flexible enough to deal with missing values (they treat them as zero or ignore them depending on the type of test) so use this function only when the method you need to use cannot handle NaNs and only on copies of the data files. Also make sure that any replacement of NaNs to zeros was noted in the metadata.

4.3 Errors in the data

Datasets contain errors more often than we would like it to happen. In many cases these are untraceable, but sometimes some of them can be detected. We know that our dataset consists mostly of non-numerical fields and artefact counts. Thus any negative numbers must be errors. Let's see all the unique values we have in our data frame, type in the console:

```
pd.unique(Olongesailie.values.ravel())
```

We seem to have two negative values: -1 and -5. Let's investigate where they come from. The '-1' is actually easy to spot, someone marked the presence/absence of spheroids using '-1' and '1'. This is a rather unfortunate approach as the value will skew the results of any statistical test. Instead, we can simply replace these with a much more suitable '0'.

```
Olongesailie['Spheroids'] = Olongesailie['Spheroids'].replace(-1, 0)
```

We can make it even better now, by turning it into a boolean (True/False) variable - the most appropriate data type for marking presence/absence.

```
Olongesailie['Spheroids'] = Olongesailie['Spheroids'].astype(bool)
```

However, there still seem to be some negative numbers in the dataframe! These are clear errors so let's convert all negative values to NaNs.

```
Olongesailie = Olongesailie.replace('-5', np.nan)
```

Run the `pd.unique(Olongesailie.values.ravel())` function again; all of the remaining values look plausible so we will leave it at this.

4.4 Subsetting the data

Once the data is tidy we may start running some statistics on it. However, most of the time we are interested only in a specific subset of the data - for example, only one type of artefacts or just the lower strata. Subsetting data in Python is surprisingly easy.

Let's start with a few basics. Type the following commands:

To access a column:

```
Olongesailie['HA']          # note the square brackets
```

To access a row:

```
Olongesailie.iloc[1]        # first row of the data  
Olongesailie[:5]            # first five rows
```

To access a specific data point, first specify the column and then the row of the observation.

```
Olongesailie['HA'][3]        # third row of the column 'HA'  
Olongesailie['HA'][:5]       # first five rows of the column 'HA'
```

Now let's do some more concrete subsetting the data - say we only need the location of the site and the number of large cutting tools and handaxes (the 'HA' column). Note double brackets!

```
handaxes = Olongesailie[['Level', 'Strata', 'Large.cutting.tools', 'HA']]  
print handaxes.head()
```

If we are not actually that interested in the first 3 sites, we can specify which rows we want.

```
handaxes_selection = Olongesailie[['Level', 'Strata', 'Large.cutting.tools', 'HA']] [3:]  
print handaxes_selection.head()
```

If you want to exclude columns rather than rows we can use the `drop` function similar to the one we used to get rid of NaNs; just change the axis argument to 1 (0 - removes rows; 1 - removes columns). For example, we can remove the 'Large cutting tools' column. Type:

```
handaxes_noLCTs = handaxes.drop(['Large.cutting.tools'], axis = 1)
```

Quite often we want to work only with data which meet a certain condition. For example, we are interested only in handaxes coming from the lower strata.

```
handaxes_lower = handaxes[handaxes['Level']=='Lower']  
print handaxes_lower
```

Imagine that we have a minimum number of artefacts threshold, say 3. The data can be reduced to only these sites which contain over 3 handaxes.

```
large_assemblage = handaxes[handaxes['HA'] > 3]  
print large_assemblage
```

Finally, sometimes we want to sample the dataset - i.e. pick up a number of observation at random and perform the analysis only on them. We can achieve that using the `sample()` method⁶.

```
handaxes_sample = handaxes.sample(n = 5)
print handaxes_sample
```

The argument `n` specifies how many rows should be sampled from the dataset.

⁶ If it gives you an 'Attribute error: Data Frame object has no attribute sample'. Check which version of pandas you have installed (just type in the console `pd.__version__`). If it is below 0.16.xx reinstall it. Go to the command line/terminal and type `conda uninstall pandas` (it will prompt you to confirm - hit 'y' and enter) and after the previous version is uninstalled: `conda install pandas`. Restart spyder.

5. Common statistical tests

There are a few statistical tools used across the board by archaeologists working in commercial context and in academia. We assume the reader's previous knowledge of the statistical tools we present here so we will concentrate on showing how to implement and run different statistical tests rather than explain what they actually do.

Note to Python 2.7 users, insert this line at the beginning of your script:

```
from __future__ import division
```

In Python 2.7 (but not Python 3.x) the integer division will return an integer. For example $7 / 2$ will produce 3. This line prevents the truncation, so $7 / 2$ is 3.5. If you are not sure which version of Python you have type in the console $7 / 2$ and see the result.

Read in the data from 'Handaxes.csv'. It contains measurements of various dimensions (L - Length, B - Breadth, T -Thickness) of 600 handaxes coming from the Lower Palaeolithic site of Furze Platt. Use the `head()` and `hist()` functions to check the data.

5.1 Descriptive statistics - mean, mode, standard deviation, variance

A quick and dirty way of getting all descriptive statistics in one line of script is the `describe()` method. Create a new script or type in the console:

```
from scipy import stats
handaxes.describe()
```

	L	L1	B
count	600.000000	600.000000	600.000000
mean	121.915000	41.640000	70.865000
std	25.947831	14.622817	14.047503
min	69.000000	15.000000	34.000000
25%	103.000000	32.000000	61.000000
50%	118.000000	39.000000	70.000000
75%	136.000000	49.000000	80.000000
max	242.000000	136.000000	123.000000

You should get a nice printout (Figure 10) of all descriptive statistics for all columns containing numerical values. However, if you prefer to calculate them one by one, they all follow the same general structure. For example, if you are interested in the Length ('L'):

Mean and median: `handaxes['L'].mean()` and `handaxes['L'].median()`

Minimum and maximum value: `handaxes['L'].min()` and `handaxes['L'].max()`

Standard deviation and variance: `handaxes['L'].std()` and `handaxes['L'].var()`

If you want to run descriptive statistics only on the subset of data which meets a certain condition, you can do it inside the function. For example, let's calculate the mean length of handaxes in each level (Lower, Middle and Upper).

```
handaxes[handaxes['Level'] == 'Lower']['L'].mean()
handaxes[handaxes['Level'] == 'Middle']['L'].mean()
handaxes[handaxes['Level'] == 'Upper']['L'].mean()
```

It looks like the length is slightly increasing the younger the handaxes but it is difficult to ascertain that the difference between assemblages from different levels are significant without running some statistical tests.

⁷ Integers are the 'full' numbers such as 1, 5, 4036, while floating point numbers contain a fraction, for example 0.01, 8.43526, or 4036.000004.

5.2 Checking if the data is normally distributed

Most of the statistical test used in archaeology assume that the data is normally distributed (check this [tutorial](#) if you are not sure about your knowledge of the normal distribution). Thus, the first step is always to check what kind of distributions we are dealing with. To get a general idea most people produce a histogram to see how their data is distributed. Let's subset the data and plot it.

```
LBT = ['L', 'B', 'T']
lower_levels = handaxes[handaxes['Level'] == 'Lower'][LBT]
lower_levels.hist()
middle_levels = handaxes[handaxes['Level'] == 'Middle'][LBT]
middle_levels.hist()
upper_levels = handaxes[handaxes['Level'] == 'Upper'][LBT]
upper_levels.hist()
```

A more formal way is to test the data for normality.

```
stats.normaltest(middle_levels)
```

If the result is higher than your alpha level (0.05 is the most commonly accepted significance threshold) the test is not significant, i.e., the data is normally distributed. If you run it through all levels, you will see that in practice only the breadth of handaxes is consistently normally distributed. However, this is expected as the dataset is large, which means that even small deviation from the perfect distribution will fail the test.

5.3 Student's T-Test and MWW Rank-Sum tests

Student's T-Test is probably the most commonly used test for comparing two sets of data. We can use it to, for example, compare the breadth of handaxes between the Lower and Upper levels at Furze Platt.

```
print stats.ttest_ind(lower_levels['B'], upper_levels['B'])
```

The test returns the T statistics and the p-value (in that order). If the p-value is below your alpha level (commonly 0.05) then the difference between the data is statistically significant.

Student's T-Test assumes that the observations come from a normal distribution. As we have seen this is not always the case. Instead of the Student's T-Test we can use WTT Rank Sum test. Let's see if the length of the handaxes significantly differ between the levels.

```
print stats.ranksums(lower_levels['L'], middle_levels['L'])
```

Similarly to Student's T-Test the test returns the z statistics and the p-value (in that order).

5.4 ANOVA

To check what differences are significant between all three levels we can run a simple ANOVA. This time we will feed all the data into it.

```
print stats.f_oneway(lower_levels, middle_levels, upper_levels)
```

The output consists of two lists (Figure 11), the first contains the f-values and the second the p-values. The three numbers correspond to the length, breadth and thickness columns

respectively. You can clearly see, that the only significant differences are in the length of the artefacts (the p-value 3.0122220e-07 is given in scientific notation. It is 3.0122220×10^{-7} , i.e. 0.00000030122220).

```
In [84]: print stats.f_oneway(lower_levels, middle_levels, upper_levels)
(array([ 15.39949012,   2.06153328,   0.6796489 ]), array([ 3.01222200e-07,
1.28163678e-01,   5.07186583e-01]))
```

Scipy 'stats' library contains other common statistical tests (e.g., [chi-square](#), [Pearson](#), [Kolmogorov-Smirnov](#), [Shapiro-Wilk](#)), all of which work in similar fashion to the above.

5.5 More complicated data analysis

Python can handle even advanced statistical operations, including regression analysis, time series, nonparametric models and more via the statsmodels [module](#). Follow this [link](#) for a tutorial on multiple regression analysis. This [tutorial](#) provides detailed, step by step instructions including code snippets on performing Principal Component Analysis in Python. For time series check this extensive [tutorial](#). There are also a number of more specialist libraries, including some for [non-parametric models](#), [spatial statistics](#) and [Bayesian](#).

6. Plotting

Python plots used to be renowned for their clunkiness and plainness. However, with the recent updates of the plotting library - matplotlib - these criticism became outdated. A strong coordination between pandas, numpy and matplotlib libraries make it a powerful tool for scientific visualisation. Start with importing matplotlib:

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

Follow it with a line which turns 'old' python graph style into a shiny new design:

```
pd.options.display.mpl_style = 'default'
```

For these of you that are hooked up on the graphics style of the R library 'ggplot' - hacking python to give you ggplot has never been easier. Simply add this line to your script.

```
plt.style.use('ggplot')
```

To show the python plotting capabilities we will use the Snodgrass dataset. It represents 91 house pits from the site of Snodgrass in Missouri, USA. Read in the data as snodgrass. We can start straight away with visualising everything in one summary plot. Type in the console:

```
snodgrass.hist()
```

This gives an impressive, although somehow small display of all the data. Let's try again with a bit bigger one.

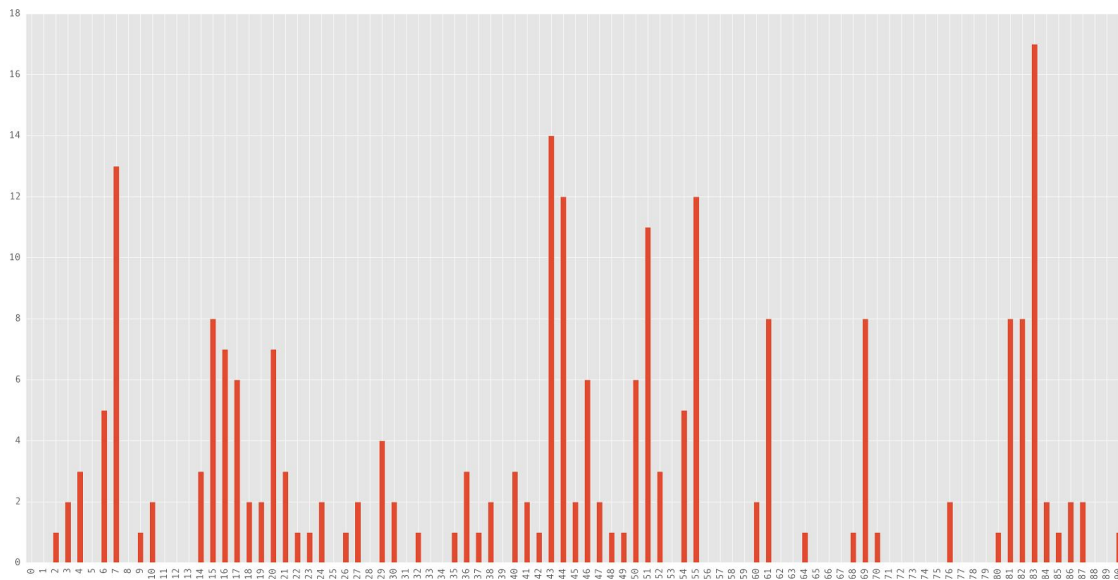
```
snodgrass.hist(figsize = (10, 10))
```

Let's now focus on the frequencies of specific artefacts. Start with getting the names of the columns, type in the console:

```
list(snodgrass.columns.values)
```

We could, for example, make a barplot of the projectile points found on the site:

```
snodgrass.Points.plot(kind = 'bar', figsize = (20, 10))
```



This produces a graph (Figure 12) showing the number of points in each house pit. It looks ok but it doesn't tell us much apart from there is quite a variance in the frequency of points. However, with a little bit of data manipulation, we can start plotting much more interesting patterns. Let's investigate the spatial distribution of artefacts. The site has been divided into three areas - 1, 2, and 3 - recorded in the column 'Segment'. We will calculate the mean of each type of artefact in each of these areas and plot them.

First specify which columns contain artefact counts:

```
snodgrass_artefacts = [ 'Points', 'Abraders', 'Discs', 'Earplugs', 'Effigies', 'Ceramics']
```

Pivot the table so that we get a data frame with the mean number of each artefact type per area:

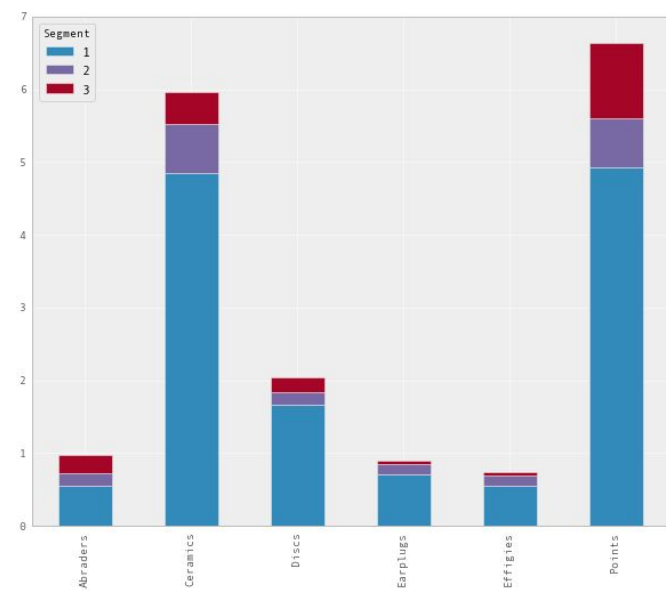
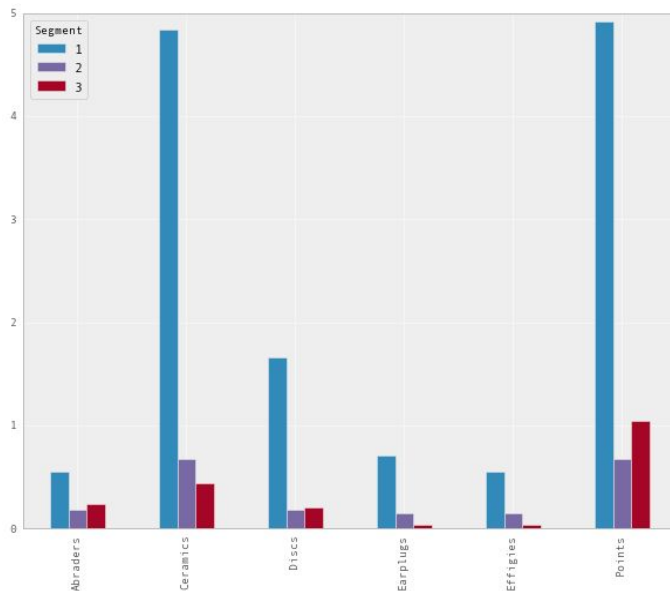
```
snodgrass_areas = snodgrass.pivot_table(snodgrass_artefacts, columns = 'Segment', aggfunc = 'mean')
```

And finally, plot it:

```
snodgrass_areas.plot(kind = 'bar')
```

This is a much more meaningful bar plot (Figure 13a) than the first we tried, but we can make it even better. Let's start with stacking it (Figure 13b).

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

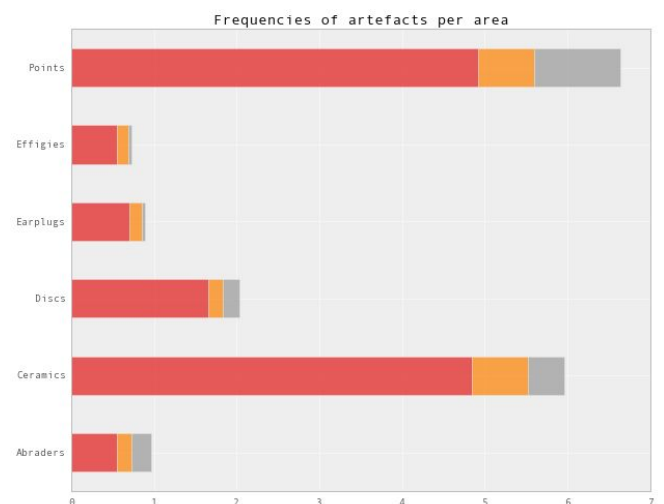
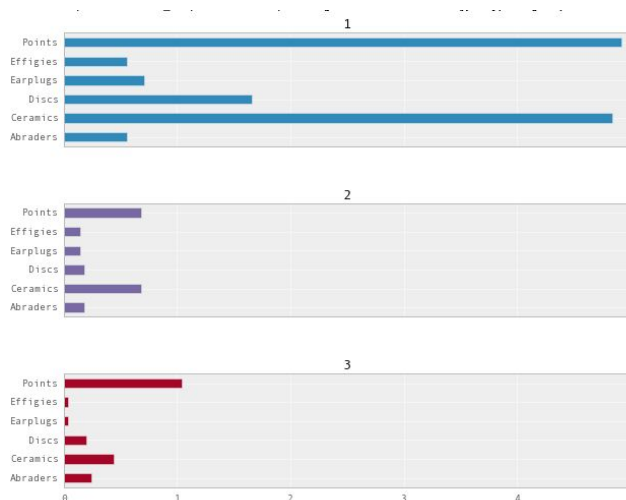


Alternatively we can also place each of the plots into a separate figure (Figure 14a):

```
snodgrass_areas.plot(kind = 'barh', subplots = True, legend = False)
```

There are a number of arguments we can pass to the plotting function in order to make the plot neater. In this example we will flip it on its side, remove the legend, give it a different set of colours⁸, set transparency, and give it a title (Figure 14b).

```
snodgrass_areas.plot(kind = 'barh', stacked = True, legend = False, colormap = 'Set1',  
alpha = 0.7, title = 'Frequencies of artefacts per area')
```



One of the most commonly used type of plot is the boxplot also known as box and whisker plot. If you are not sure about how it works, check [this interactive tutorial](#). To show our data as a boxplot, type in the console:

⁸ Matplotlib [default colourmaps](#) are rather plain. However, a number of online tools provides colour palettes that can be easily plugged into python. [Colorbrewer](#) and [Tableau](#) have some good examples.

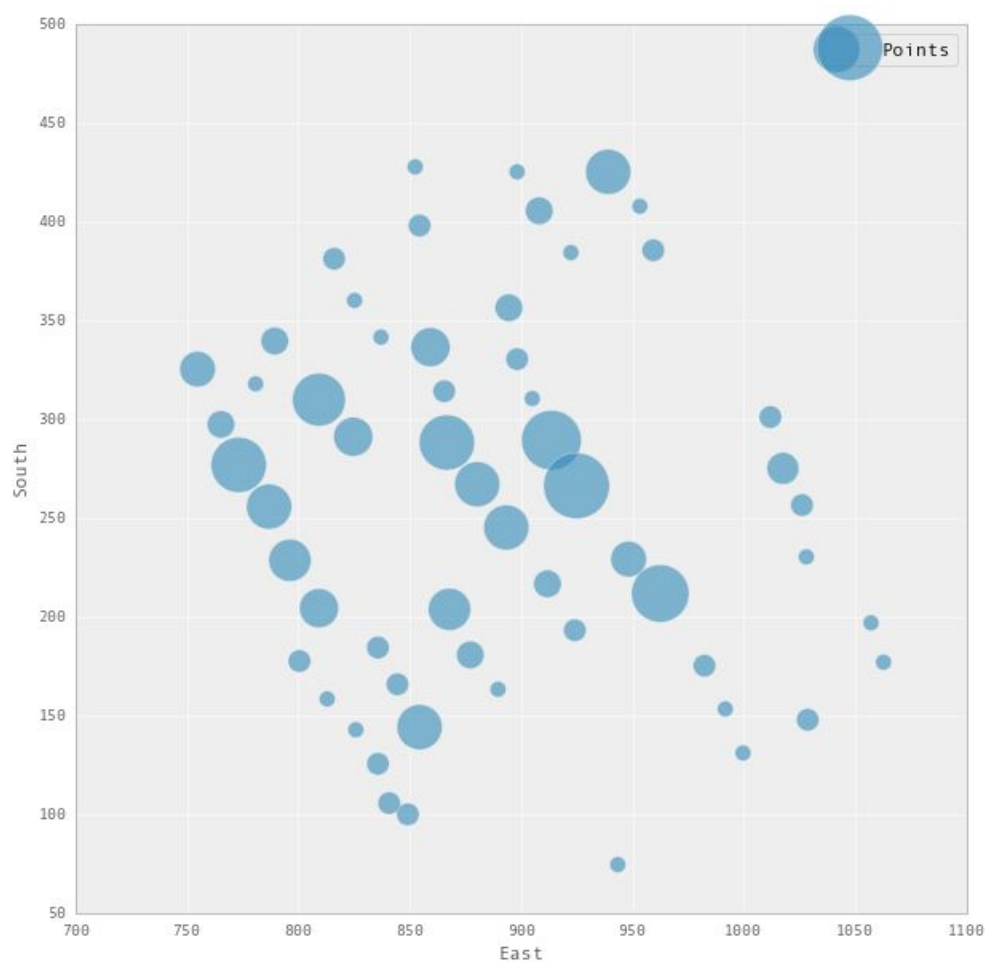
```
snodgrass_areas.boxplot()
```

Finally, we can use the scatterplot function to show the distribution of different types of artefacts on the site. We will use the coordinates in columns 'East' and 'South' to plot the position of each pithouse. Type in the console:

```
snodgrass_distribution = snodgrass.plot(kind = 'scatter', x = 'East', y = 'South')
```

Let's now plot the distribution of one of the artefact types: points. We will show the frequency of points by specifying the size ('s') argument as a proportion to the number of points in each pithouse (we multiply it by 100 to make the results more visible) (Figure 15).

```
snodgrass_frequency = snodgrass.plot(kind = 'scatter', figsize = (10, 10), x = 'East', y = 'South', s=snodgrass['Points']*100, alpha = 0.6, label= 'Points')
```

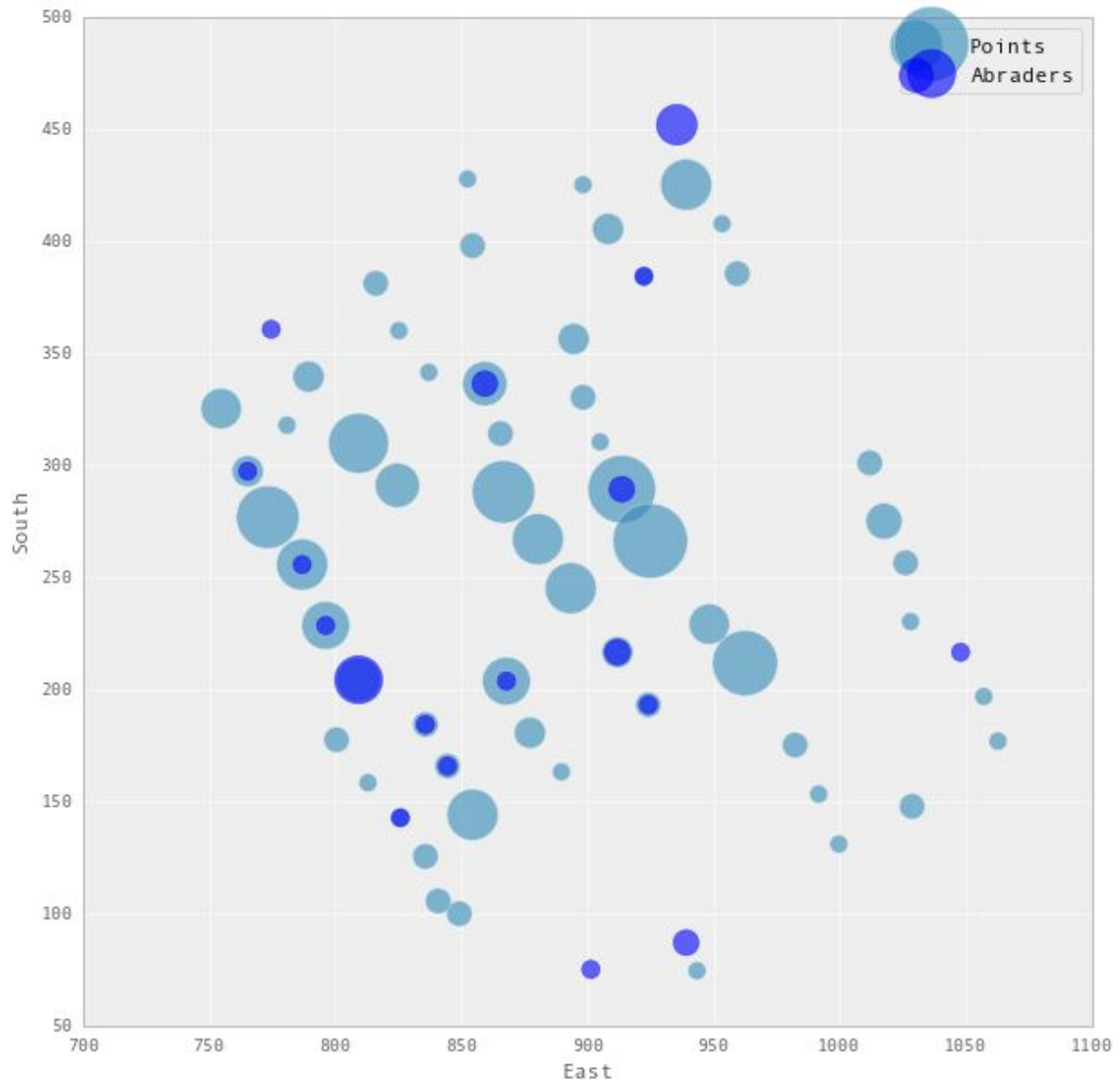


To compare the distribution of different types of artefacts on one plot, pass the name of the original plot as the 'ax' argument. If you are running the script from the console, separate the calls with a semicolon or you can just type them into a new script in the editor window and run it from there (Figure 16).

```
snodgrass_frequency = snodgrass.plot(kind = 'scatter', figsize = (10, 10), x = 'East', y = 'South', s=snodgrass['Points']*100, alpha = 0.6, label= 'Points') ;
```

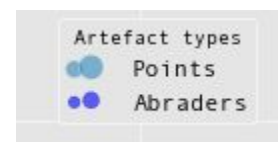


```
snodgrass.plot(kind = 'scatter', x = 'East', y = 'South', s=snodgrass['Abraders']*100,
color = 'b', alpha = 0.6, label= 'Abraders', ax = snodgrass_frequency)
```



Matplotlib is a very flexible tool - virtually every element of the plot can be changed to one's liking. For example, in this case the legend looks slightly off with the symbols much too big for the frame. Also, we want to make sure that it stays in the upper right corner and finally, it could do with a title. This can be easily amended in one line of code using the 'legend' function (Figure 17).

```
snodgrass_frequency.legend (title = 'Artefact types', markerscale = 0.3,
loc='upper right', bbox_to_anchor=(0.99, 0.99))
```



Note that if you simply write this in the console, it will not output a full figure. However, if you save the plot to a file, the changes will be there⁹. To save a plot as a publication ready figure, use the `savefig()` method.

```
fig = snodgrass_frequency.get_figure()
fig.savefig("Snodgrass_artefact_distribution1.png", dpi = 300)
```

⁹ If you want to see the figure, write the plotting code in the script editor. When you run the full code it will produce the most up-to-date version of the figure.

7. The Batch mode

The main strength of using a scripting language for data analysis is the ability to automatise many processes. Once the script is written, it can be used any number of times, therefore reducing the need to repeatedly do the same manipulation on a dataset. Think that time when a new batch of data came in and you had to redraw all the plots again, or when someone delivered 52 separate files with data and you spent a week copy-pasting it into one Excel file. If you have a Python script these kinds of tasks take on average up to 10 seconds. We will give a few simple examples as a taster of Python's capabilities, but once you become more confident with scripting Python's the possibilities are endless.

In the previous example we plotted the distribution of different types of artefacts found at the site of Snodgrass. You might have noticed that we had to repeat the same code for each artefact type. Let's start a full 'stand alone' script that can be used for creating graphs showing the distribution of all of the artefact types. Start with reading the file in:

```
snodgrass = pd.read_csv("Snodgrass.csv", header = 0)
```

Specify which columns contain artefact counts:

```
snodgrass_artefacts = [ 'Points', 'Abraders', 'Discs', 'Earplugs', 'Effigies', 'Ceramics']
```

Create a color palette (this one was picked from colorbrewer2.com):

```
my_colors = ['#d73027', '#fdae61', '#ffffbf', '#a6d96a', '#abd9e9', '#4575b4']
```

Set up the plot:

```
scatter1 = snodgrass.plot(kind = 'scatter', x = 'East', y = 'South', figsize = (10,10))
```

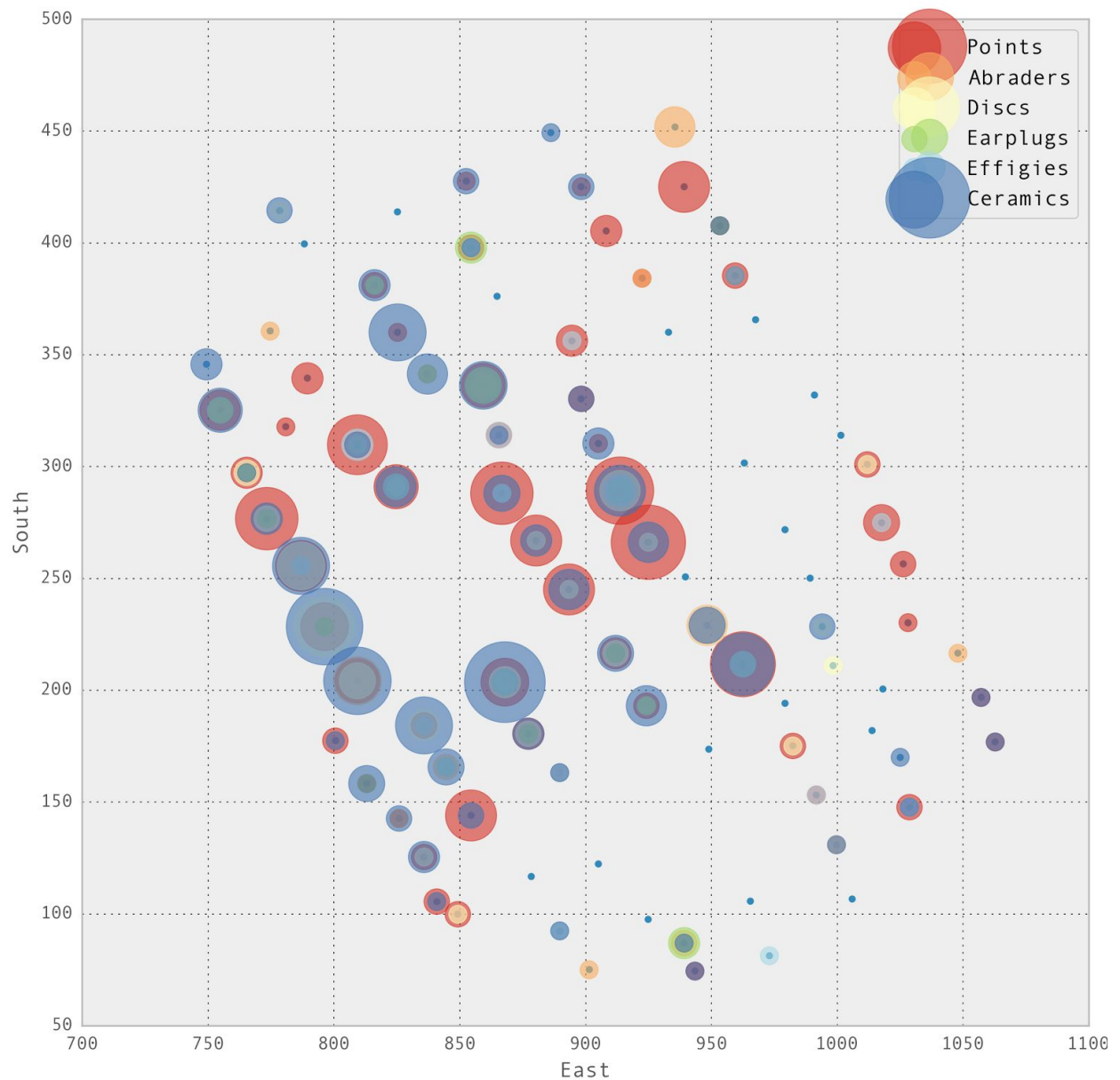
The core of the function is a for-loop. Type in the script:

```
for artefact in snodgrass_artefacts:
    my_color = my_colors[snodgrass_artefacts.index(artefact)]
    snodgrass.plot(kind = 'scatter', x = 'East', y = 'South',
s=snodgrass[artefact]*100,
    color = my_color, alpha = 0.6, label = artefact, ax = scatter1)
```

The for-loop takes each element ('artefact') from the list ('snodgrass_artefacts') and performs a set of operations on it (i.e., all the indented statements). In our case we need to determine a new colour at each iteration of the loop, otherwise all the artefact types will be plotted in the same colour. To do so we take the index of 'artefact' and pass it as the index to the 'my_color' list. To understand how exactly this happens add some print statements:

```
for artefact in snodgrass_artefacts:
    print artefact
    print snodgrass_artefacts.index(artefact)
    my_color = my_colors[snodgrass_artefacts.index(artefact)]
    print my_color
```

We then add a new plot to the figure. If you compare this line of code with the original one, you can see that we replaced the word 'Points' or 'Abraders' with 'artefact'. So `s=snodgrass['Points']*100` and `label = 'Points'` became `s=snodgrass[artefact]*100` and `label = artefact`. This is because with each iteration of the loop the 'artefact' stands for a different element of the 'snodgrass_artefacts' list, first 'Points', then 'Abraders', then 'Discs', etc. It does not matter how long the list is, by changing its contents you can create any graphs you want in no time (see Figure 18 for a plot of all artefact types).



The for-loop (together with the if-statement) is the cornerstone of all programming and can be used for any task which involves repeating the same action. Let's go back to the example of multiple files that need to be put together. This is a common occurrence, in particular when a piece of equipment separates the files into separate 'day' folders. Previously we

uploaded the snodgrass dataset as one csv files. However, this time it comes with one .csv for each pithouse, 91 files in total. Download the folder 'snodgrass'. Start a new script, and type:

```
pithouses = range(1, 92)
columns = [ 'East', 'South', 'Length', 'Width', 'Segment', 'Inside', 'Area', 'Points',
'Abraders', 'Discs', 'Earplugs', 'Effigies', 'Ceramics', 'Total', 'Types']
temp = []
```

The first lines gives a list from 1 to 91 (note we had to specify the range as (1, 92) - the function excludes the second value). If you're unsure how it looks like, just type `print pithouses`. The second line contains the list of column names as the files come without a header. Finally, we create an empty list to append the already opened and read files before we concatenate them together. Now the for-loop:

```
for house in pithouses:
    path = 'snodgrass/pithouse%d.csv' %house
    df = pd.read_csv(path, names = columns)
    df['Pithouse'] = house
    temp.append(df)
```

Similarly to the plotting example, the for-loop takes each element (house) of the list (pithouses) and performs a number of operations on it. First, the files are located in the 'snodgrass' folder and are called 'pithouse1.csv', 'pithouse2.csv' etc. Thus as we pass the file path, it needs to be different in every iteration. The %d symbol in the path name means 'insert whatever is specified later on', in our case that's a number picked from the pithouses list:

```
for house in pithouses:
    path = 'snodgrass/pithouse%d.csv' %house
```

If you're unsure whether you understand how exactly it works, use the print statements:

```
for house in pithouses:
    print house
    path = 'snodgrass/pithouse%d.csv' %house
    print path
```

Second, we read in the file using the familiar `pd.read_csv()` method. We pass the column name list as an argument so that we get a header for the table. In the third line we add a new column - 'Pithouse' - to make sure that we do not lose the pithouse numbers which are only recorded in the file names. Finally, we append the data to the empty list ('temp') we created beforehand.

To put it all together, we can use the `concat()` method. It takes all the elements from our temporary list ('temp') and concatenates it into one data frame. As we read the files we gave each a separate index - the 'ignore_index' argument removes them.

```
data = pd.concat(temp, ignore_index = True)
```


You can now check how your data looks like with `data.info()` and `data.hist()`. If it resembles the data frame we used in the previous section then congratulations, you just saved yourself hours of tedious work!

6. Going forward

Trying to write one's script instead of settling for a more familiar environment of Excel or SPSS may be daunting to many. However, Python is a very widely used programming language and the resources available for anyone who wants to learn are vast. For those of you who have the time, we recommend the [Python for Data Analysis](#) textbook and [Think Python](#) for more general applications. If you want to try to script something and you don't know how to get on with it, google it! 'plotting heatmaps pandas tutorial' or 'chi square analysis python' will give you hundreds of websites, youtube videos and blog posts. Choose a tutorial that you can understand - they are pitched at people with different levels of coding experience so not every one will be suitable for you. You will also find that the documentation of a specific package ([pandas](#), [matplotlib](#), [stats](#) or [Python](#)) contains many basic information you may need (what arguments a function takes, what are the defaults, etc). Getting error messages is part of the development process and you will come across many of them, some of which may sound rather cryptic. Again, copy-paste the error message into google - hundreds of thousands of people use Python everyday, someone must have encountered the same problem before and asked for help. Similarly, if you don't know how to do something, there are no prizes for spending hours trying to figure it out on your own. Google 'how to... read a text file with no spaces/plot horizontal boxplots/save a figure to a pdf...' and the answer is likely to be there. And if you are truly stuck and cannot find the solution online, use [stackoverflow](#) - an online forum when people post IT related questions. Good luck!

8. Additional notes

The most commonly used format for data manipulation in archaeology is the .xls or .xlsx (Excel file). This is not a problem as transferring your data between .csv and .xlsx is simple.

- Transferring data from Excel

If you want to save an Excel spreadsheet in one of these formats navigate to the File menu, choose 'save as'. Open the scroll down menu and choose '.txt' or '.csv' (Figure 18).

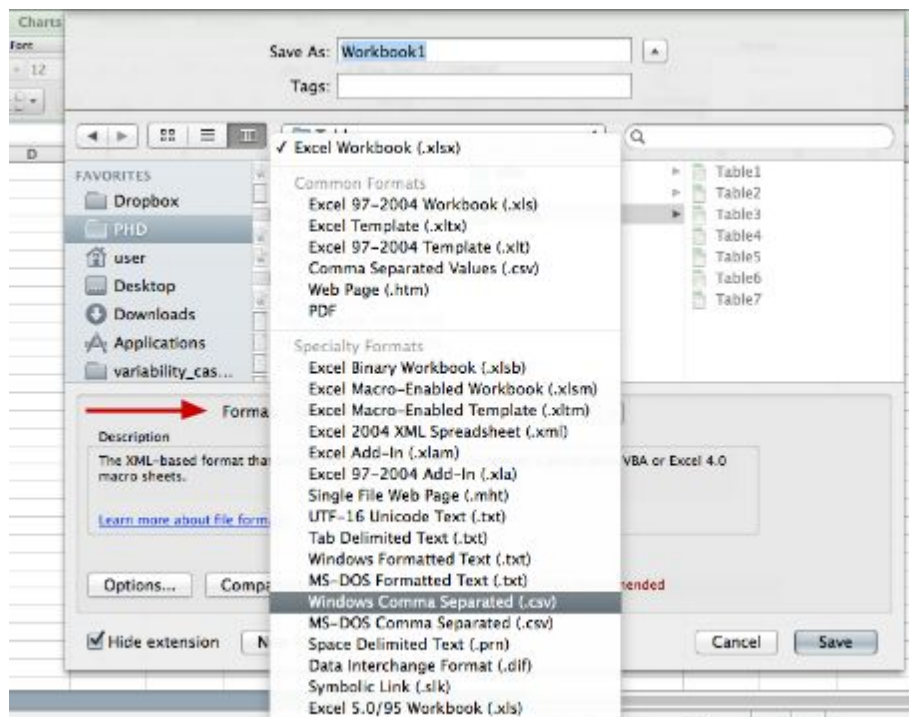


Figure 18

- Reading data from .txt or .csv files into Excel

To open a .txt or .csv file in Excel do not double click on it (it will probably open in Notepad). Instead start Excel, open the File menu, choose 'Open' and navigate to the file you want to read. Figure 19 shows the dialogue window. Click on 'delimited' and hit 'Next'. In the next window, tick 'Space' if are opening a .txt file or 'Comma' if it is a .csv file. Hit 'Finish'.

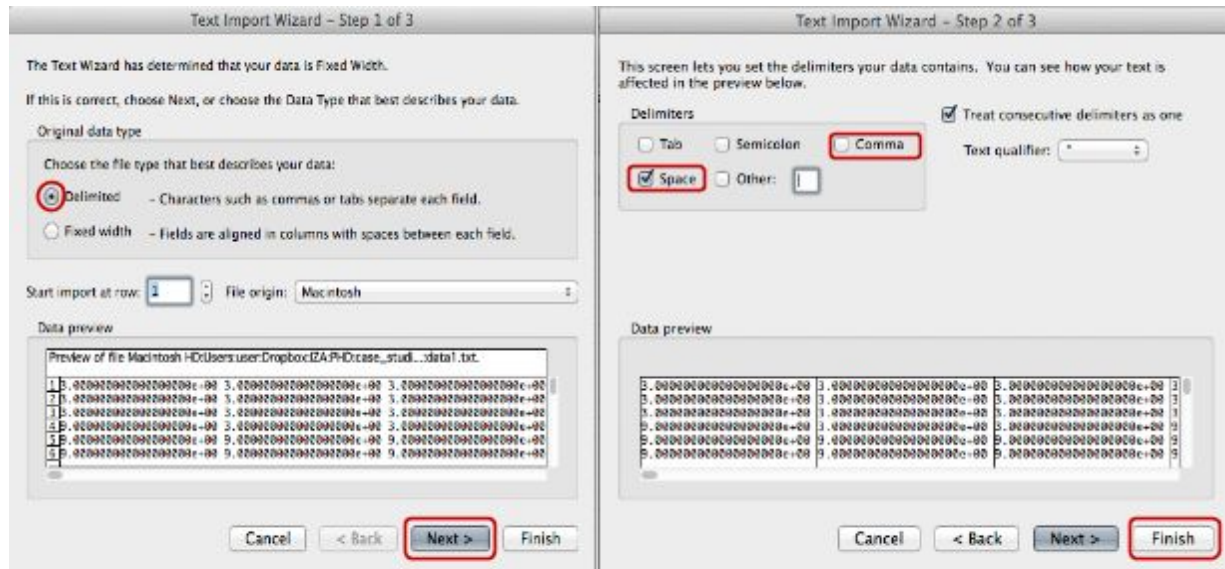


Figure 19