An Introductory Python Programming Guide

**Table of Contents**

**Chapter 1: Introduction**

Complete language

A rich set of types

Capabilities: statistics and beyond

**Chapter 2: Installing Python for statistical applications**

Installing Python

Downloading Python

Installing (full) Anaconda

Installing MiniConda

Installing alternatives

Running Python in a Jupyter notebook

Tuples, Strings, Lists, Dictionaries and Arrays

Entering a function

Numpy, Nan, Inf, and Vectorization

Style

Organization

Chapter 3: Data File Manipulation

Importing data

Reading data from text files

Reading data from Excel files

Viewing the data

Additional resources

Exercises

Data Frames

Understanding the structure

Creating Data Frames

Data transformation

Managing missing values

Testing for missing values

Exercises

**Chapter 4: Exploratory Data Analysis**

Prerequisites

Descriptive statistics

Numerical data

Central Tendency

Variability

Range

Percentiles

Variance

Shape

Outliers

Putting it together

Frequency Tables

Margins Tables

Simplifying

Exercises

Visualizing data

Getting started with plotting in Python and Jupyter notebooks

Scatter Plot

Histogram

Box Plot

Stem-and-leaf plots

Advanced plotting with the bokeh module

Grammar of graphics

The Basics

Aesthetic mappings

Specifying geometric shapes

Statistical Transforms

Grouping into multiple subplots

Additional resources on bokeh

Other visualization libraries

Exercises

Moving forward with Python and Data Science

Data selection

Additional resources

Final advice

Chapter 1: An Introduction to Python

Python is a relatively new programming language. Its roots go back to 1989 and the Centrum Wiskunde & Informatica (abbr. CWI; English: "National Research Institute for Mathematics and Computer Science"), a research center in the field of mathematics and theoretical computer science in Amsterdam, Netherlands. There Guido van Rossum, a young researcher at the time, started work on his idea of a new language that, he intended, would be easy to learn, could easily interact with the operating system and could call subroutines written in the C language. He released his code on the UUCP network, a predecessor to the Internet, in 1982.

Guido (as Mr. van Rossum is called within the Python community after his username on the language discussion boards) initially started work on Python as an alternative to a research language he had been working on called ABC. He named the language after the *Monty Python's Flying Circus* television show. This initial irreverence has continued within the language community and many module names are puns, based on Monty Python sketches, or make references to snakes.

As the language has evolved it has brought in new ideas in computer science and software engineering, stealing, as Guido has put it, the best ideas from other languages.

Unlike most other popular languages and statistical packages, Python has never had a corporate owner or backer. The choice of what to add to the core language is discussed on-line by developers and users until either a consensus develops or, if that doesn't happen, until a decision is made by Guido. Within the Python community Guido is referred to as the BDFL (Benevolent Dictator For Life).

One of the early strengths of the language was its support for add-ins called 'modules'. Any (sufficiently skilled) user is free to develop a module for their particular problem and then can contribute that module to all other Python users. The premier statistics module, which we will be using in this guide, is Pandas. Developer Wes McKinney started working on pandas in 2008 while at AQR Capital Management, a financial management firm in Connecticut, out of the need for a high performance, flexible tool to perform quantitative analysis on financial data. Before leaving AQR he was able to convince management to allow him to open source the library. Another AQR employee, Chang She, joined the effort in 2012 as the second major contributor to the library. The name “pandas” is a contraction of PanelData, an advanced data structure supported in the module.

References:

https://en.wikipedia.org/wiki/History\_of\_Python

https://en.wikipedia.org/wiki/Centrum\_Wiskunde\_%26\_Informatica

https://en.wikipedia.org/wiki/Guido\_van\_Rossum

https://en.wikipedia.org/wiki/Pandas\_(software)

A complete language

So let's look at the characteristics of the language. And let's contrast Python with the C language, which has influenced the design of many other programming languages, including Java and JavaScript.

A simple C program that would print "Hello" three times on separate lines could look like this:

#include <stdio.h>

int main(int argc, char \*argv) {

int i;

char \*msg = "Hello";

for (i = 0; i < 3; i++)

{

printf("%s/n", msg);

}

return 0;

}

And the same thing in Python would look like this:

for msg in ["Hello"]\*3:

print(msg)

To run the C program you would have to type it into a file and then run a compiler, which is a separate program, before you could run your program. To run the Python program you would just start the "python" program, type the two lines of code, add a blank line at the end, and the program would run.

The advantage of a C program is that it is very fast to run compared to the Python program. We are explicitly telling the computer what to do at a low level and this results in little wasted time when the code must be executed. In contrast the Python program is actually doing a lot of work that it is not telling us about. This makes writing the code, and testing the code, much easier and faster.

In small programs like these two the runtime may be 100 times faster. In a more realistic load, like a larger program which takes longer to run, the Python version may be something like 6 times slower. Consider that today the cost of computers is very small compared to the cost of a skilled programmer or data scientist. This was not the case when the C language was defined.

So, what is Python doing behind the scenes which makes a programmer's life easier but the program slower?

First thing in the C program is the "#include" statement. This brings into the program information that the compiler needs in order to understand the print statement. Python doesn't need this since the print statement is built into the language.

Next in C is a "main" function declaration. In the C language this name tells the computer where to start running the program at. Python has a simpler rule, it scans a file noting all the definitions in the file and any statements that are not within the definition of a function or class it runs.

Inside the C function we see two statements that define variables. Back when 64K of memory cost a month's salary it was important for the programmer to know where they were using memory. Having to explicitly define all variables for the program helped with that book keeping. Python in contrast reserves memory for a variable when you first assign a value to it and lets the memory reservation go away when you move on to where the program can't use that variable any more. So, the work a programmer is doing in their head tracking memory when coding in C is now done for them by the Python language.

Now we come to the "for" loop. In both language the work "for" is a keyword that indicates we are going to do something multiple times. The way it is written in C is with a variable that is used to count how many times you go through the loop. The variable is not used for anything else in this example. In Python a for loop operates by stepping across the members of a list. So, we don't need the index variable, we just need to specify the list. That is what '["Hello"]\*3' does, it specifies a list with the word "Hello" in it, repeated three time. If we wanted to specify a loop to happen a specific number of times Python provides the built-in function range(n) which returns a list with 'n' elements, starting at 0 and increasing until n-1. But the range function is rarely needed in well designed Python programs since looping can be done by iteration over the data structure that C would need to index into.

In C, after a 'for' we see a statement enclosed in curly braces ('{}'). These are there to let the computer know where the statements to repeat are located. A well formatted C program will have these lines indented under the for (as has been done here) so that the programmer can tell by looking what is repeated by the loop. But this indenting is not required in C. In Python however the way the computer knows what is to be repeated by the loop is the repeated statements are indented under the for statement.

So, summing up the attributes of Python that we see in just this short two-line program:

- Dynamic: The code is run immediately when it is read by the Python program. variables are created and destroyed as needed, and their type is based upon what is put into the variable.

- Higher level structures are built into the language. Thus we loop over lists which are a high level structure.

- Code structure is shown by indents. Unlike many other languages the way the code is indented is not just to make it easy to read, it is the way the structure is defined.

A rich set of types

I have mentioned above, without giving a definition, that variables have a 'type'. The type of a variable determines what operations may be performed on that variable. Some types act like you would expect them to based on what you learned in elementary school. So, 1 + 2 is an operation on integers that gives you 3. And 1.5 / 0.45 is the division of two decimal numbers. Other types are not so obvious but once explained quickly become understood. A string type is indicated by putting quotes around a sequence of characters. Either single quote or double quotes may be used. So, what does 'ten' + 'five' mean? Here we have two strings (string meaning a string of characters) that have a plus sign between them. The plus operation when used with a string type on either side is defined to mean joining the strings together, so the result is 'tenfive'. Some operations are defined that can work with a different type on either side. If you looked at “1 + 1.6” you wouldn't immediately see that you are adding two different types, but the computer will see it that way.

We think of integer numbers as points taken out of the real numbers, and mathematically that is true. The computer however stores integers in a different way than it stores real numbers. This gives rise to two different types, 'int' and 'float'.

An int is stored in the computer much like you probably learned binary numbers work. 0 is represented by a computer storage location filled all with binary zeros in all the bit positions. Change the lowest bit to a binary one and you have a 1. Adding a 1 to this will get you the binary 000...010 (where the ellipsis represents all the zero bits in between the beginning positions and the end) - this represents 2. This pattern can be extended to cover the negative integers by making the top bit a one and doing some manipulations on the lower bits. This is getting away from my point: how the int type is stored in the computer is baked into the hardware of the computer.

Now let us take a quick look at how a real number is stored in the computer. We need a representation that will cover a wide range of values but does not need to be infinitely precise. If we are within 0.000000000001 of the exact answer that will be 'good enough'. So the real number is actually stored in the computer memory as a range of bits that represent an exponent followed by bits that represent the value. This type is referred to as a 'float' because the decimal point floats up and down the number. When a float number that is very big or very small is displayed it will show the value part followed by the letter 'e' (for exponent) followed by the power of 10 that the value is to be multiplied by. So 2.220446049250313e-16 is a very small value because it is preceded by fifteen 0's (the 'e-16' part). Similarly, 2.6913770132388557e+99 is a huge value since after the first number would be 98 more digits if you were to write it out in the standard way. This is where the approximate value of

the float type shows up. This huge number is the value of pi raised to the 200th power. If you were to calculate this precisely the 18th digit, the digit after the final 7 in the number above would not be zero. As stored in the computer all digits after that final 7 are 0’s. The value given is however close enough for practical purposes.

The point of this excursion into bits is to explain that types control how the computer calculates. Going back to the example of 1 + 1.5 we see that the first number is an int and the second is a float. Usually you can’t add two things that are of different types. Fortunately, Python has rules that tell it that when an int and a float are to be added change the int to the equivalent float value and then add the numbers. Python also has rules for the mathematical expressions that work with ints that produces integers of essentially unlimited size. Languages like C that use the integer type of the computer have a limit to how big or negative an integer is allowed to be. In Python if an operation on an int is going to produce a result that exceeds the size of an integer on the underlying computer Python will grab another integer sized storage space and use that as the high order part of the number. So, Python is able to calculate and display huge integers such as 200! (200 factorial) which has 375 digits. When speed is important though, as it is when running operations on millions of data points, Python can call upon numerical packages that use the types built into the computer hardware.

So types, and how they are tracked, are important to a language. Python has a lot of types built in and more can be added from modules of code that are imported into a program. A quick look at the important types that you will deal with follows.

The lowest level types are ints, floats and strings which we have already discussed. These are the lowest level since they correspond to abilities that are built into the hardware of your computer.

Then there are the types that are more complex combinations of code and data and are built into the language. The most important of these are the list, the set, and the dict (short for dictionary). These will be discussed below. After Python has been installed, you can try things out.

There are types that are created by programmers and that live in modules that must be imported into your program. The date and time types are an example. These two are part of the Python language and can always be imported into your program. There are other types, such as the DataFrame, that are part of optional packages. Some of these types are written in Python but some, which could be slow to execute, are written in C so that they run at the best possible speed.

Finally there are types that can represent things you write yourself. These are functions, classes, and modules.

In Python something that you can refer to in the program will have a type, and in the most general case is called an 'object'. Python keeps track of the types for you and will give an error if you try to do something with an object that cannot be done with it. For instance an int is not a function and cannot be called. But a function cannot have 1 added to it. Trying to do either of these will result in an error message.

Capabilities: statistics and beyond

There are many useful capabilities built into Python, and the chance to create programs that use these capabilities is one reason that the language is valued beyond its ability to run statistical analyses. For instance the http.server.SimpleHTTPServer module allows the easy creation of a web server that serves static files. The json module has a mode that can take data written in the JSON format (which is often compacted to remove extra space and lines) and print it out in a more easily understandable format. The csv module hides the inconsistencies of reading csv files written by different programs, making it easy to write code that handles the differences. There are so many capabilities available in the standard library that it is said that the Python language comes "with batteries included".

Beyond the built in capabilities are the modules and packages that people have written and contributed into the Python ecosystem. Packages are a combination of a Python program that can be called from your Python code and subroutines written in other languages. This combination offers the best of both fast execution speed and the ease of programming in Python. The statistical packages have already been mentioned. Complementing them are modules for web development, linear algebra, neural network development, and graphics, just to name a few.

In the next chapter we will install Python and start running some code.

Chapter 2: Installing and Using Python

Installing Python

There are several choices to make when you decide to install Python.

Five years ago, the language made a major change as it moved from version 2.6 to 3.0. At the time of the transition there was a lot of confusion as to whether one should use the 2.x version or go with the improvements in the 3.x line. A big part of the trouble was that in using Python a lot of the useful routines are held in "modules" that are separate pieces of code, apart from the main language. And when 3.0 was announced most of those modules needed to be updated to work with the new language version. To help those with older code, a new version of the 2.x line, 2.7 was released that included some new capabilities, but that is the end of the old line. This 2.7 release is still officially supported, and you can still download it but no new capabilities will be added to it. The 3.x line is currently (mid-2017) up to version 3.6 and has capabilities and performance improvements which make it the compelling choice. And almost all the optional modules that originally held back a transformation to 3.x have been updated to use the new line. At this point the only reason to choose to download the 2.7 version is if you work for a company that has existing code that has not been transitioned and you need to work with that old code.

Another choice is whether you are running on a 32bit computer or a 64-bit computer. This is a measure of how many bits are handled by the computer at one time when doing arithmetic. 64-bit computers are generally faster than 32-bit computers. Each of the major operating systems (Mac, Windows and Linux) has different way to find out this information.

Apple Macintosh

If you are running on an Mac that is recent enough that it is still supported by Apple, you are running on a 64-bit computer.

Windows

Go to Start and bring up the start menu. In the right hand column move the mouse over the "Computer" entry and right-click to bring up a menu. Click on the word "Properties" in the menu which will bring up the Systems Information panel. In the System section is "System Type" which will be either "64-bit Operating System" or "32-bit Operating System".

Linux

Open a terminal window (Control-Alt-T) and on the command line type "arch" (short for architecture). It will print out either x86\_64 (for a 64-bit Linux) or x86\_32 (for a 32-bit Linux).

Downloading Python

Python is available packaged in several distributions. A distribution consists of not only the core Python language but a variety of additional packages targeted towards specific uses. The package maintainers do the work of making sure that a particular version of Python is compatible with all the additional packages and they keep up to date with changes from the package developers.

For this guide, we are going to use the Anaconda distribution. Anaconda calls itself an "*Open Data Science Platform*". Anaconda includes all the major Python statistical packages as well as packages for graphing and data visualization. It includes an interactive development environment (IDE) that is useful for debugging your code as well as the Jupyter notebook programs that allow you to interactively explore data using Python.

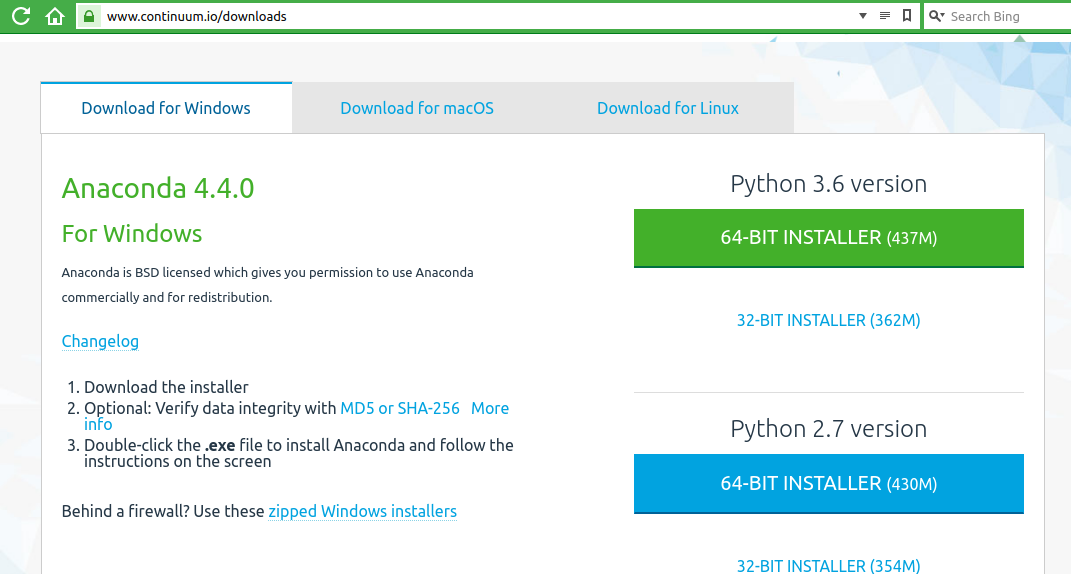
There are two version of the Anaconda distribution. One is full Anaconda. Over 150 packages are automatically installed with Anaconda. Over 250 additional open source packages can be individually installed from the Anaconda repository with the conda install command (although for the statistical analysis done in this course you won't need to install anything more). Installing full Anaconda is easy. The downside is that you have to download all the packages, including the ones you do not need, and that takes a while (the Windows version for example is 437 Megabytes). And when installed on disk it takes up about two gigabytes of space. For instance, one of the default packages is astropy for doing astronomy calculations. It takes up 1.4 Megabytes of disk space. Only a few people will use this package. However, if you have a cable modem quality or better Internet connection and sufficient spare disk space installing full Anaconda is the way to go.

The alternative is to download the miniconda distribution. miniconda installs quickly since all it contains is the Python language and 'conda', a package installer. With miniconda installed you can then download just the packages you need for this course. How to do that, and which packages those are, will be detailed below. The download and the disk space usage will be about half of what full Anaconda takes.

The official installation documentation for doing either of the following installs is [available](https://conda.io/docs/download.html) on the Conda documentation site.

Installing (full) Anaconda

1. Go to <https://www.continuum.io/downloads>
2. Scroll about a third of the way down the page to the section with three tabs across a page division. The tabs are titled "Download for Windows", "Download for macOS", and "Download for Linux"
3. Click on the tab for your operating system
4. Click on the download button for Python 3 (as of this writing 3.6 is the current version).



*Note for Windows users:* Many firewalls are configured not to allow executable downloads from the Internet, especially business networks with centrally managed PCs. If you are behind such a firewall, clicking the button will not be able to pull down the installer. There is, at the bottom of the left column of the tabbed division, a link to a page where a zipped version of the installer is available and you will probably be able to download that.

After you have clicked the button the installer will begin to download. A pop-up will ask for your work email address so that Continuum can send you the 'Conda Cheat Sheet'. This is optional and you can click the 'No Thanks' link if you prefer not to give out your email address.

4. Complete the remaining steps shown on the download tab for your operating system.

You will be asked to provide a directory to put the files into. The default is a file under your user id's home directory.

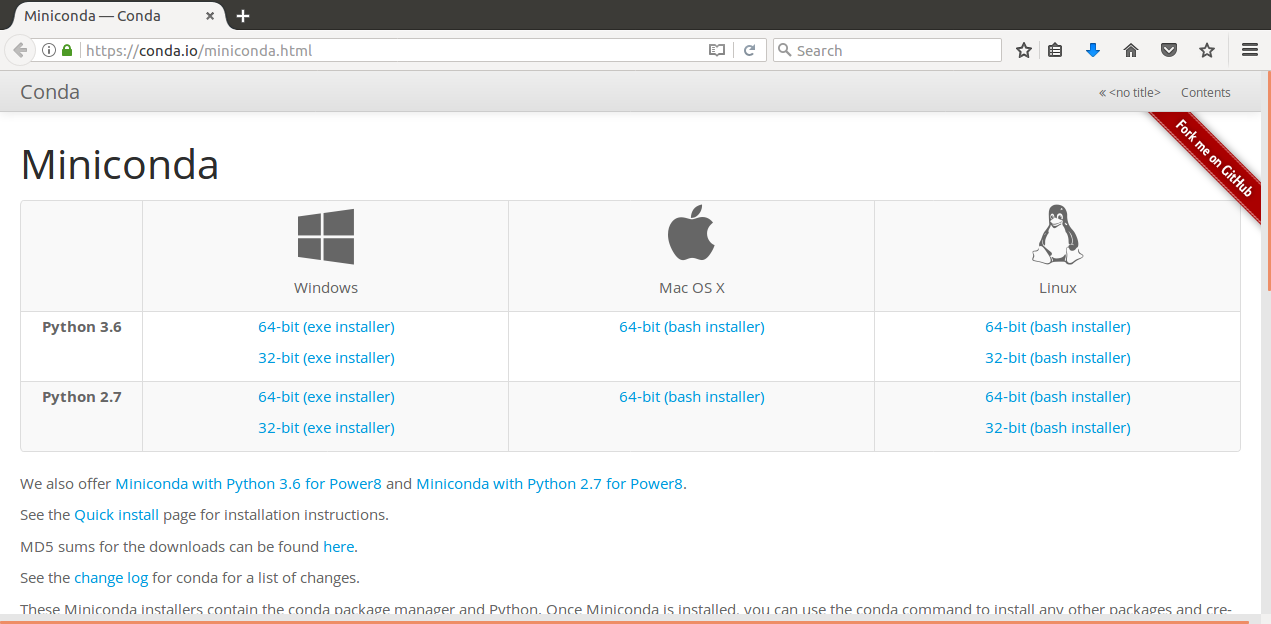
*Note for Windows users:* If you install Anaconda in your home directory you will not need the Administrator privileges required to write to the System and Program Files directories. The install does attempt to write to the Registry file to set the path to where the programs are installed. A tightly locked down system may not be able to run the install because of this security. See your System Administration group for help completing the installation in this case.

*Note for Mac and Linux users*: At the end of the installation process the installer will ask if you want your PATH variable set to include the new software. Generally, this is what you want. However, if you or a system administrator has modified your ~/.bashrc file it is possible that the change will not be effective. If this is the case please edit your PATH in the .bashrc file (if you have changed it) or seek the assistance of the person who made the change.

*Note for Mac users:* The install process must be run by a user who has permissions to modify the Applications directory. The Mac installer does not currently support personal installation even though it shows it as an option.

Installing MiniConda

1. Go to <https://conda.io/miniconda.html>



2. Click to download the appropriate version based on your OS and computer bit size.

3. Save the downloaded file and open the folder that you saved it in.

4. Double click the installer

5. Follow the prompts

6. Windows user*:* bring up the "Anaconda-prompt" program that has been installed.

*In Windows10* Start -> Apps Installed -> Anaconda3 -> Anaconda-prompt

Mac and Linux user: bring up a new terminal window.

Type "conda install anaconda-navigator statsmodels" into the shell. Answer "y" to the question "Proceed?".

8. Type "anaconda-navigator" into the shell

This will bring up the graphical installer / app runner

9. Find "Jupyter notebook" and click the install button

10. Find "spyder" and click the install button

11. Go back to "Jupyter notebook" and click the "Launch" button

After a minute or so, your web browser will opens up with a view on a notebook. The notebook’s initial view shows you the files and folders in your home directory.

*For Windows Users:* Bring up a file viewer and go to C:\ProgramData\Miniconda3\Scripts. Find the "anaconda-navigator" application. Click once to select and then right click to bring up the menu of file actions. Choose "Create a Shortcut". When the shortcut is created drag it to the desktop.

Installation alternatives

Beyond the focus of the guide, there are a few ways that you can run Python, and the Anaconda packages, in the cloud and interact with it through a web browser.

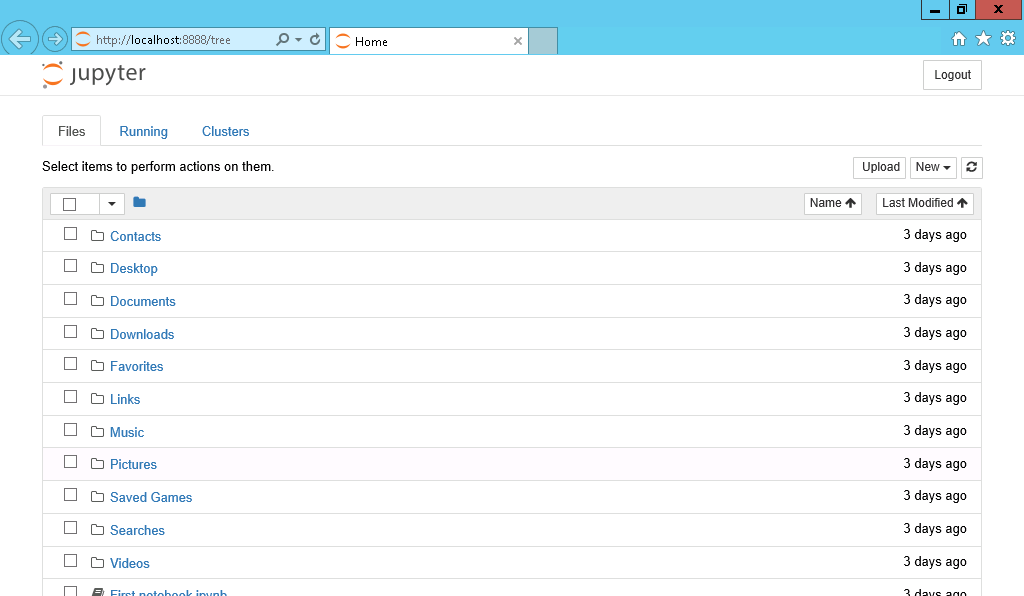
The free tiers of [AWS](https://aws.amazon.com/) (Amazon Web Services) (tried it) and [Google AppEngine](https://cloud.google.com/appengine/) (haven’t tried it) give you enough power to run the software used in this guide. These are time limited trials and billing starts after some set time period has elapsed. One year in the case of AWS.

There is also the [Python Anywhere](https://www.pythonanywhere.com/) service. Their free tier does not allow running a Jupyter notebook, but the smallest paid tier allows it $5.00/month as of this writing. All of Anaconda is already installed on a *Python Anywhere* instance.

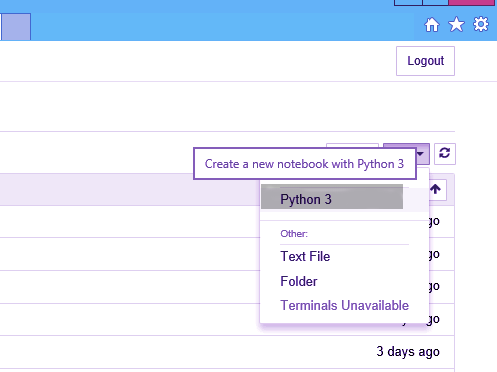
These suggestions may be an alternative to having to buy or upgrade a personal computer.

Running Python in a Jupyter notebook

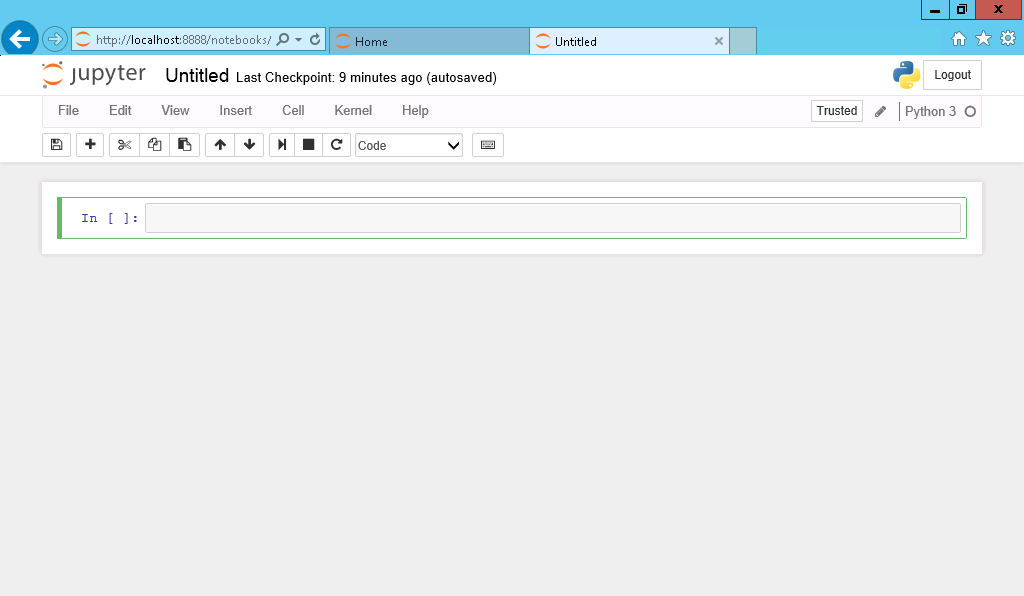
When the notebook application starts it will first show a command window and then a web browser window will cover it and the page in the web browser will show a view of your local directories and any saved notebook files.



Above is what the initial screen that you will see when you run jupyter notebook looks like.



From the initial screen that shows your files, go to the "New v" button in the upper right portion of the screen and click it. From the drop-down menu that appears select "Notebook/Python3" and click.



A new tab will open with a "notebook" and an empty "cell".

Click in the cell and type in a Python expression. Since standard arithmetic expressions are valid Python expressions, enter "2+2". When you type enter after this line you are moved to the next line where you could enter another expression. Since we just want this expression evaluated type shift-enter (holding the shift key down tap either the enter or return key). Within the output area of the cell the answer is printed and a new cell is created below the first one.

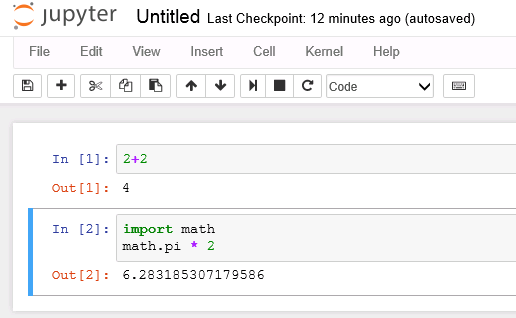
Note that one cell at a time is active and can be typed into. The active cell is shown with a highlight border around it. You can change which cell is active at any time by clicking in another cell.

Now let's bring in some (slightly) more advanced math. Enter the following lines in the new (empty) cell:

import math

math.pi \* 2

This time, rather than typing shift-enter type ctrl-enter (holding the ctrl or control key down tap the enter or return key). You remain in the cell rather than moving to a new cell and the answer is printed out for you.



If you forget what the key sequence is to execute a cell, type escape-h (clicking the escape key followed by clicking the “h” key) which will bring up a help page of key assignments.

Now change the second line to

math.cos(math.pi \* 2)

And run the cell again (control-enter again. By now you should have this down and I won’t repeat the keystroke, just tell you to run or execute the cell). The output this time is 1.0 which is the cosine of 2 pi radians as is taught in trigonometry classes. It is comforting to see that our calculations agree with standard mathematical knowledge.

So, what is the "import math" line doing? It tells Python to consider the name "math" as the name of a module. When a module is imported Python goes out to find the module, initializes any data structures that may be needed, and from then on "math" is now the entry point to finding anything in that module.

So how do you know what is in the module? Reading the documentation is a good start. The official documentation for all the standard modules is available on the python.org website [here](https://docs.python.org/3/py-modindex.html). You can also see the web based documentation (assuming you are connected to the Internet) by clicking on the "Help" item in the Jupyter toolbar and selecting Python. This opens Python's documentation in a new browser tab or window. The Jupyter notebook has other help information built in. Go to a new cell (shift-enter) and type "math." and click the tab key. The period after "math" is what tells Python you are looking for something in the math module. A drop-down list appears with all the names that are defined in the math module. Use the up and down arrow keys to move to a name and click enter. If you type the first few characters the list is filtered down to just the names with those starting characters. When you type enter while a name is highlighted, the chosen name is entered where you were typing. If you want to get rid of the drop-down list after it appears just click the escape key (labeled "esc" usually). Now that you have a name, you can get a short description of what the name is. It could be a constant like math.pi or it could be a function and you need to know how to call it. Put a question mark in front of the name in an empty cell and execute the cell. A window pops up at the bottom of the screen showing information about the function. To dismiss the window, click the "x" in the upper right corner.

In Chapter 1 of this guide the idea of "type" was discussed. You can ask Python for the type of any name. Enter the following in a cell and execute:

a=4

type(a)

The answer comes back

int

Now go back and put a decimal point after the 4, making it read "a=4.", and execute. This time the answer is

float

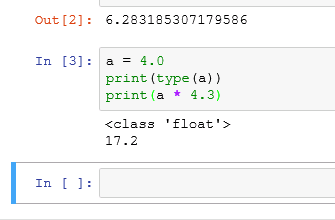
So far, we have been getting our output by using the internal rule that if the last thing in a cell has a value, print out that value. This is useful for quick interactions but if you want to see a number of different responses from the execution of a cell you want to "print" them. To do this you just call the built-in function print and tell it what you want it to print. To tell it what to print you put a value or expression in parentheses after the word "print".

Let's modify our cell again. After the line "type(a)" put a new line

print (a \* 4.3)

Now when you execute the cell it outputs 17.2 and does not print the information about the type of variable a. This is because getting the type is no longer the last line and since there is nothing specified to do with the value Python ignores it.

Change the second line to "print(type(a))" and this time when you execute the cell you will get both the type information and the result of the calculation.



A word about the numbers you see appearing to the left of both the input and output areas of a cell. When a cell is first created it has no number and no output area. To the left you will just see "In [ ]:" but after the cell is executed there is a number within the square brackets. This number is the counter of when the cell was executed. Since you don't have to execute the cells in order from top to bottom it is sometimes important to see what cells were executed before other cells.

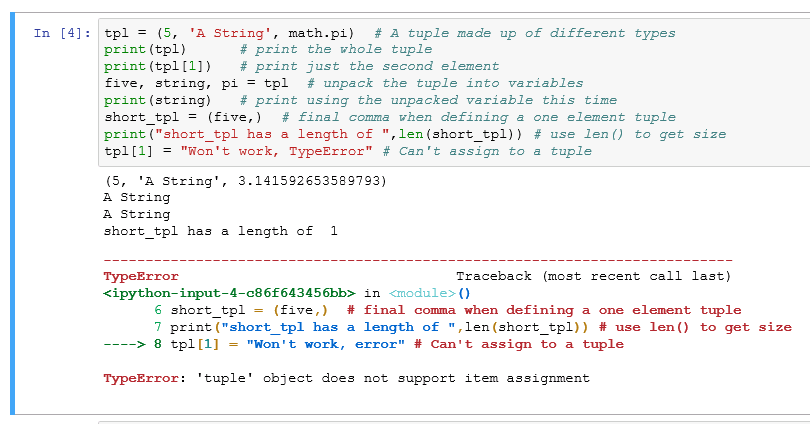
Tuples, Strings, Lists, Dictionaries and Arrays

There are a number of composite types built into Python that are very useful. What follows is just an introduction. Included are links to the full official documentation pages with full details about all the functionality that Python supports. These types are composite types because they can have multiple elements that are sub-parts of them. The sub-parts can be of any Python type and do not have to all be the same type, although they usually are.

Python refers to these as ‘sequence types’. There are a number of [common operations](https://docs.python.org/3.6/library/stdtypes.html#common-sequence-operations) that all of the sequence types implement, such as calling len() to find out how many elements are in the sequence and being able to use a “for” statement to loop over the sequence element by element. I will talk about the more useful operations when I talk about each of the types below.

The simplest composite type is the tuple. It can have one or more elements within it and you can pull the elements out either by assigning the elements to individual variables or by indexing to pull out a particular element. Once a tuple is created it can’t be changed, although you can assign another tuple to the variable which holds a tuple and it will look like it changed. Because the elements of a tuple cannot be changed they are quick to create and don’t use much memory. Because they are ‘light-weight’ Python uses tuples internally for a number purposes which you do not see as a casual user of the language. For the programmer, tuples are mostly useful for packing together into one variable a number of elements and passing that around.

To create a tuple, you write a left parenthesis, one or more variables or constants separated by commas, and then a right parenthesis. If the tuple has only one element you must include a comma after the element. To pull a value out of the tuple you use square brackets with an integer inside to pull out that position’s element. You can also put a number of variables equal to the number of elements in a tuple to the left of an equals sign and the elements in the tuple will be ‘unpacked’. This means the elements are each assigned to each variable; the first variable gets the first element, the second variable the second element, etc. It is an error if you don’t specify the correct number of variables. To determine the number of elements in a tuple, you can use the len() function.



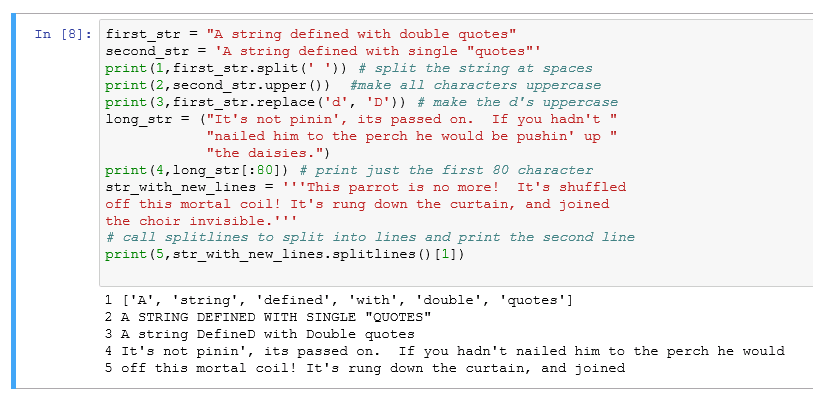
Above is some code playing with tuples and ending with an error when an assignment to an element within a tuple is attempted. There are four print statements and four lines of output (before the error message). Make sure you understand how each print statement is producing the corresponding line of output.

There is further discussion of tuples in the [official tuple documentation](https://docs.python.org/3.6/library/stdtypes.html#tuples).

The **string** type has already been mentioned. It is used to hold a string of characters. Unlike other sequence types the elements of a string must all be the same type - characters. Like the tuple, one you define a string you can’t change the characters within. To edit the string, you have to create a new string, copying the unchanged part from the previous string and adding in your changes. Since you often put the new string in the variable where the old string was it looks like you have changed it, but you haven’t. Aside from the common sequence operations the string type has [special string functions](https://docs.python.org/3.6/library/stdtypes.html#string-methods) that it supports.

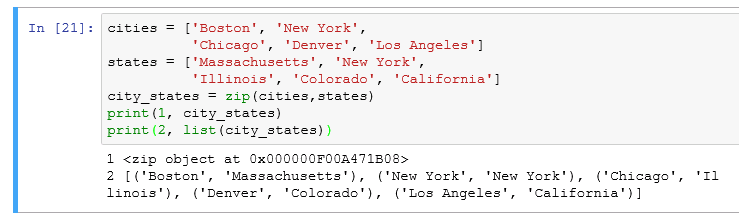
To create a string, you put quotes around it. A single quote creates a string with the characters that follow until another single quote is found on the line. If you start with a double quote it takes all the characters until a matching double quote. It is an error to have the string open when you get to the end of a line – that is you must have the closing quote on the line with the opening quote. Because you can use either style of quotes it is easy to specify a string that contains the other type. If a string is too long to fit on one line there are a couple of ways to handle this. If the string is to be joined together and not have newline characters in it you can close the quote on the first line and any whitespace (spaces, newlines or tabs) are ignored until the next quote and the string will go into memory without any break. In this case you need to let Python know that the string is one long expression by putting parentheses around the all the parts on the multiple lines. If the newlines are intended to be part of the resulting string you can use triple quotes and everything between the starting triple quote and the ending triple quote, including newlines, are part of the string value.

You can *capitalize* a string (capitalize only the first character), make it all *lowercase*, make it all *uppercase*, or make it titlecase (first letter of each word is capitalized). There is even a function swapcase which makes upper case lowercase and lowercase uppercase. You use *split* to break a string apart into a list of strings with the break happening wherever a character you specify is found. It is so common to break a string based on lines there is a special splitline function. Or you can join a sequence of strings together with joinspecifying the character or characters that you want between the strings when they are joined. The “in” operator (not a function) will check is a specified substring is found within a string. This is usually enough but if you want to know where in the string the substring is found the find function will return that offset. If what you are looking to do is replace one substring by another string there is the *replace* function.

Here are a few string definitions and examples of the use of string functions. Notice that the functions all return a new string (or list of strings) and the original is not changed. This time each print statement has been given a number to print out at the start of its output so that you can easily compare the original string with the print statement result that came out of the function call.

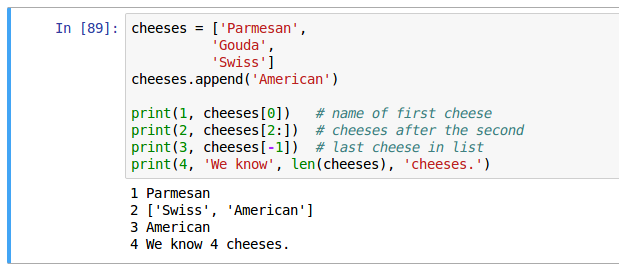
Next in the hierarchy of sequence types is the **list**. The full [documentation for the list](https://docs.python.org/3.6/library/stdtypes.html#lists) is on the python.org website. The list is a sequence (possibly empty) of Python types, but unlike a string or tuple a list is ‘mutable’. That means the program can change the list, make it longer or shorter, rearrange the elements or replace them.

The usual ways to create a list are to specify a series of variables within square brackets, separated by commas, or to create an empty list and then use the append function to add to the list as you iterate over some loop that is creating elements, such as reading from a file. There is also a special form called a list comprehension that this guide is going to skip over in an effort not to go too deep. A final way to create a list is by using the list() call and passing it a generator object. What is a generator object? Let’s look at an example.



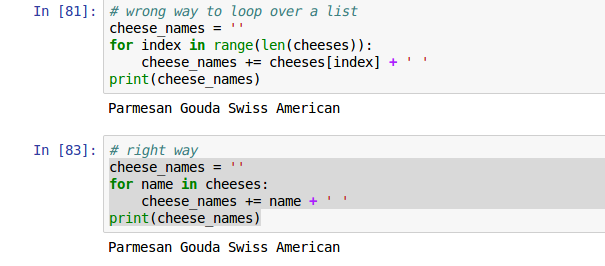
Here two lists have been constructed by explicitly specifying the contents between square brackets. The first list is city names and the second list is the corresponding state names. [***Zip***](https://docs.python.org/3.6/library/functions.html#zip)is called to pair the list elements together. In the first print call we see the result of zipping the two lists together is not a list, it is an object with no representation other than its type and location. What we have is a generator. This is an object that can be called repeatedly and each time it returns the next element in its sequence of elements. Generators take very little space in memory compared to creating the list. To create the list from the generator in the second print statement the list constructor is called with the generator as its argument and a list is created.

There are a couple things to note. In creating the original lists two lines were used for each definition in order to make this more readable. Whenever you have parentheses or brackets any newlines in between are ignored. In this case the starting bracket means that we can add newlines without violating Python's rule that a statement is contained on one line – since the newlines within the brackets don’t count. In the second output, the list consists of five tuples, showing how the elements of a list can be other types. This will become an important idea when we get to arrays which are stored as lists contained within lists.



Here we create a list of cheese names and access the names with numbers that are the positions of the cheese in the list. Python lists start at position 0. The square brackets with a number in them after the list name is what tells Python to retrieve an item from the list. A colon within the square brackets indicates a range of list items. If the first number in the range is missing (this is, we have a colon with nothing before it and a number after) this indicates starting at the first element in the list. Similarly, if within the square brackets we have a number, a colon, and then the closing square bracket we go from the indicated position to the end of the list. Negative numbers count back from the last position in the list. The len() function returns the number of elements in the list. One of the useful things to do with a list is loop over its elements and apply some processing to every element. In Python, this should be done by specifying the list in the for statement, not by using the index functionality. In some languages, there is no way to specify a loop over a list’s elements and you are forced to loop over an index series.

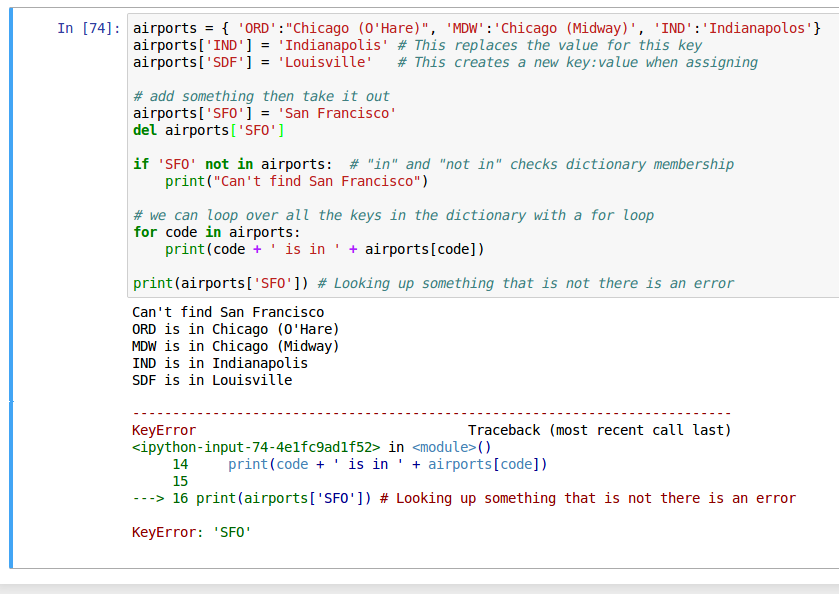
Don’t do this in Python.



In the first example, we are using the *range* function to generate the numbers from 0 up to the parameter passed to range, in this case the length of the list. So, the loop happens over a sequence, just not the sequence that we are interested in. The variable named in the for statement (“index”) is then used to pull the value we want out of the list. I the second example we step through the list and the variable (“name”) becomes, as we repeat the statement in the loop, one of the list’s elements after another.

The most complex type built into Python is the **dictionary**. A [dictionary](https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict) is a mapping between objects of one type and objects of (possibly) another type. One object is called the “key” and that can be used to retrieve another object, the “value” for that key. So that it is impossible to modify the key in a dictionary the key must be an immutable data type – such as a string, a number, or a tuple. The value can be any valid python object, such as a list, a string, another dictionary, even the name of a function that you, the programmer, created.

A dictionary is created either by creating an empty dictionary and assigning elements to it by key or with a key-colon-value series of items that create a dictionary with keys and values specified at creation. You can also create a dictionary with values and then add more values.

In this example, we

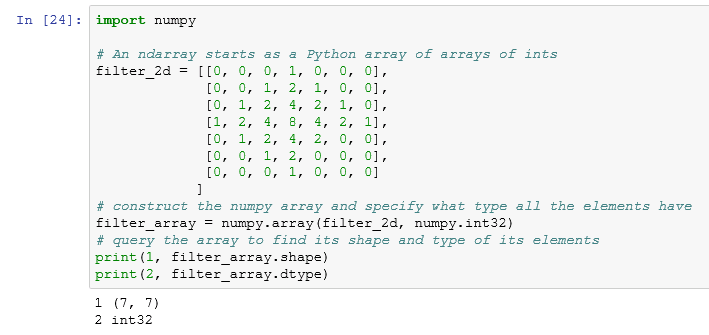
* create a dictionary with some initial content,
* add a few more elements to it,
* add another element,
* remove an element with the *del* statement,
* check for the existence of a key in the dictionary,
* loop over the dictionary (the keys are returned one by one into the variable named in the for statement),
* and finally look up in the dictionary something that is not there to show that this produces an error.

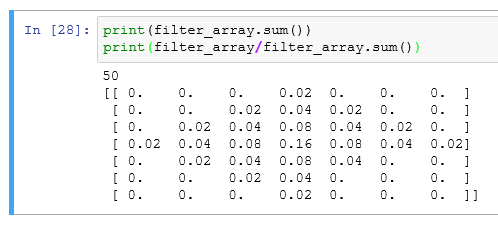
A final structure to be discussed in this section is the numpy ndarray, part of the numerical python package. The actual data type is numpy.ndarray, but it is often referred to as array because array is the name of the constructor function. The documentation of the array is [in the numpy documentation](https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#the-n-dimensional-array-ndarray), not in the Python documentation where the structures discussed to far have been.

An ndarray is a multidimensional container of items of the same type and size. Usually the ndarray is created with a fixed size which does not change. The requirement that all the items be the same type and size, and that the size be known when the array is created, is unlike the usual Python list and dictionary. The number of dimensions and items in an array is defined by its shape, which is a tuple of N positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype). Each ndarray has a specific dtype which is specified when it is created and never changes.

As with other container objects in Python, the contents of an ndarray can be accessed and modified by indexing or slicing the array. Slicing is specifying a range for the index using two integers separated by a colon. The content can also be accessed via the methods and attributes of the ndarray.

Different ndarrays can share the same data, so that changes made in one ndarray may be visible in another. That is, an ndarray can be a “view” to another ndarray, and the data it is referring to is taken care of by the “base” ndarray.

In this example, we use regular Python syntax to create a list of lists, with the elements of the inner lists all being the same type. Then we call numpy.array to create the numpy array. The two print statements show how the array can be queried to show what its shape and dtype are. This is important since many operations can only be done with arrays that have the same or compatible shapes and dtypes (dtype means “data type”).

One of the powerful aspects of numpy arrays is that a single call can take the place of an operation that, if written in Python, would require nested loops. The equivalent function inside numpy is written in C and runs much faster than if it was written in Python. In this example, we show that we can sum all the integers in the array with one call to sum (). We can also get an array produced by dividing an array of in by an int. In the second print, we use numpy’s implicit loop over an array to do the division for us with a single line of code. The result of dividing an int by and int is a float and we see the output is an array of floating point numbers.

The ndarray is the basis for building a “DataFrame” which is the way that data is stored for statistical analysis. We will be coming back to the array, the DataFrame, and how they are manipulated later in this guide.

Entering a function

Since Python is a programming language we can write a function. Once the cell that the function is defined in executes the function, it can be called from any other cell. For this example, you are going to type in a function that executes the [quadratic formula](https://en.wikipedia.org/wiki/Quadratic_formula) for finding the roots of a quadratic equation.

Create an empty cell and put the following code in it. (The code in this section is text that you can copy rather than images. You are encouraged however to type the code in yourself so that when there are errors you can learn by figuring out what the error message is referring to and fixing it.

import math

def quadratic\_eqn(a, b, c):

''' Solves for the roots of a quadratic equation in normal form

using the quadratic formula.

Inputs a,b,c: coefficients of x^2, x, and constant

Returns: x1 and x2, the roots which give an equivalent function

when put in the form a \* (x - x1) \* (x - x2)

'''

x1 = (-b + math.sqrt(b \* b - 4 \* a \* c))/(2 \* a)

x2 = (-b - math.sqrt(b \* b - 4 \* a \* c))/(2 \* a)

return x1, x2

Then execute the cell with shift-enter. If you have typed everything correctly there will be no output because you have not calculated anything, you have only defined the function.

Let's look at the parts of the function. The first line starts with "def" which is Python's way of saying "define function". After that is the function name that you are going to define ("quadratic\_eqn" in this case), and following the name is a pair of parentheses that may hold the names of input values. If the function has any input the names of the variables that the function expects to have passed to it are listed with a comma separating the variables. The next set of lines is optional, but highly recommended. This is the "doc string" that allows you, the programmer, to tell another programmer what the function does, what the input parameters represent, and what the output means. Even if you are the only person who will read this code it is recommended that you write a doc string for any non-trivial function so that when you come back to it a few months from now you won't have to puzzle over the code to figure this information out. The doc string starts with three quote marks (single or double quotes, both are accepted although single quotes are more common). The doc string ends with three of the same quote marks that started the doc string. The doc string is indented under the def line. This indent makes it clear both to you the programmer and to the Python interpreter that the doc string goes with the function. Once you have established an indent everything that goes with the function has the same indent. "If" statements and loops have bodies that are further indented as we will see shortly. The next two lines calculate x1 and x2. These two names "x1" and "x2" are just names for the results of the calculations specified after the equals sign. The actual names chosen are not significant, they are just something that tries to inform the reader of the code (you) what these values represent. Also notice that unlike in Algebra you must write out all the operations explicitly. The asterisk (\*) is used to represent multiplication. A slash (/) is used to represent division. To group together what is being divided parentheses are used. The last line of this function is a return statement that tells Python which of the values in the function are to be returned to the caller. Python allows you to return multiple values in one return statement, they are listed with a comma between them.

Once you have got the cell with the function definition to execute move to a new cell and put in some test cases.

print(quadratic\_eqn(1, 4, 4))

print(quadratic\_eqn(1, 0, -4))

If there is some error in the function you may get an error message and what is called a stack trace. The stack trace shows what function called what other function and at which line the call occurred. Hopefully you won't get any such error at this point. If you do note which line is it saying the error occurred in and check that you have typed that line in correctly.

Hopefully what you get is the following output:

(-2.0, -2.0)

(2.0, -2.0)

A general factoring result fr ax^2 + bx + c equals a(x - x1)(x - x2). Ours results mean that x^2 + 4x + 4 is equal to (x + 2)(x + 2) and that x^2 - 4 is equal to (x - 2)(x + 2).

But if you try

print(quadratic\_eqn(1, 0, 4))

In our system you get an error like this one:

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-35-f17c3cb99b5b> in <module>()

1 print(quadratic\_eqn(1, 4, 4))

2 print(quadratic\_eqn(1, 0, -4))

----> 3 print(quadratic\_eqn(1, 0, 4))

<ipython-input-33-1f3b69237b7a> in quadratic\_eqn(a, b, c)

8 when put in the form (x + x1) \* (x + x2)

9 '''

---> 10 x1 = (-b + math.sqrt(b \* b - 4 \* a \* c))/(2 \* a)

11 x2 = (-b - math.sqrt(b \* b - 4 \* a \* c))/(2 \* a)

12 return x1, x2

ValueError: math domain error

A domain error is saying that the argument given to the function is not something that it can compute. In this case, math.sqrt cannot take the square root of a negative value. Python has built in complex numbers which can allow us to compute the factors even if they are complex.

If you look at our calculation of x1 and x2 you see that we are repeating the calculation of the value that we are taking the square root of. We can "refactor" our code to take this calculation out and do it just once. This should not change the answers the code gives. So, edit the code so that it looks like this:

def quadratic\_eqn(a, b, c):

''' Solves for the roots of a quadratic equation in normal form

using the quadratic formula.

Inputs a,b,c: coefficients of x^2, x, and constant

Returns: x1 and x2, the roots which give an equivalent function

when put in the form a \* (x - x1) \* (x - x2)

'''

discriminant = b \* b - 4 \* a \* c

x1 = (-b + math.sqrt(discriminant))/(2 \* a)

x2 = (-b - math.sqrt(discriminant))/(2 \* a)

return x1, x2

Execute the cell with shift-enter. You will get no output but the active cell will advance to the next cell. Now execute the cell with the test calls in it. You should get the same results as before, two good answers and a stack trace showing a domain error.

So, we can add a check in between where the discriminant is calculated and where it is used. If the discriminant is less than zero we will make it a complex number by adding the following:

if discriminant < 0:

discriminant = complex(discriminant)

If you only made this change and ran it you would get a domain error again but this time it would be math.sqrt complaining that it can't take the square root of a complex number. But there is a module "cmath" that has the same functions as math but can handle complex numbers. We don't want to use cmath.sqrt all the time because it is about 3.5 times slower than an ordinary square root. But when the discriminant is negative we have to use it. In Python functions can be assigned to variables and so we can have a variable my\_sqrt and define it as either cmath.sqrt or math.sqrt depending on the discriminant.

The code now looks like this (with changes highlighted).

import math

import cmath

def quadratic\_eqn(a, b, c):

''' Solves for the roots of a quadratic equation in normal form

using the quadratic formula.

Inputs a,b,c: coefficients of x^2, x, and constant

Returns: x1 and x2, the roots which give an equivalent function

when put in the form a \* (x - x1) \* (x - x2)

'''

discriminant = b \* b - 4 \* a \* c

if discriminant < 0:

discriminant = complex(discriminant)

my\_sqrt = cmath.sqrt

else:

my\_sqrt = math.sqrt

x1 = (-1 \* b + my\_sqrt(discriminant))/(2 \* a)

x2 = (-1 \* b - my\_sqrt(discriminant))/(2 \* a)

return x1, x2

Notice that the "if" statement line ends with a colon and the lines that are to be executed if the "if" is True are indented under the "if". When the "if" test is False we can have an "else:" clause at the same indent level as the "if" above it and then one or more statements to be done that are indented below the "else:" line. If we wanted to have more tests between the "if" and the "else" Python has an "elif <condition>:" statement.

Now when you execute the cell that has the test statements:

print(quadratic\_eqn(1, 4, 4))

print(quadratic\_eqn(1, 0, -4))

print(quadratic\_eqn(1, 0, 4))

You get the following nice output

(-2.0, -2.0)

(2.0, -2.0)

(2j, -2j)

Notice that for imaginary numbers Python uses the symbol 'j' for the square root of -1. Mathematicians generally use "i" for this while most fields of engineering use "j". Python went with the engineers on this one.

Code Comments

Comments can be inserted in any Python code by typing an octothorpe ("#"). Anything from the "#" to the end of the line is ignored. So, the following is valid Python and runs as if the comment was not there:

print(quadratic\_eqn(1, 0, 4)) # this one has complex roots

Help on Functions

Already mentioned was that you could type a question mark and a function name and get help on that function. You can also use the built-in help function.

?math.sin # information is presented in a pop-up

help(math.sin) # information is presented in the output area of the cell

And the result:

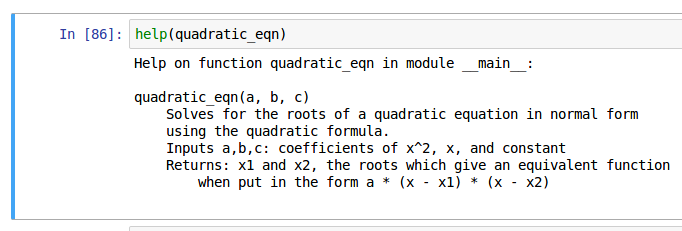
Help on built-in function sin in module math:

sin(...)

sin(x)

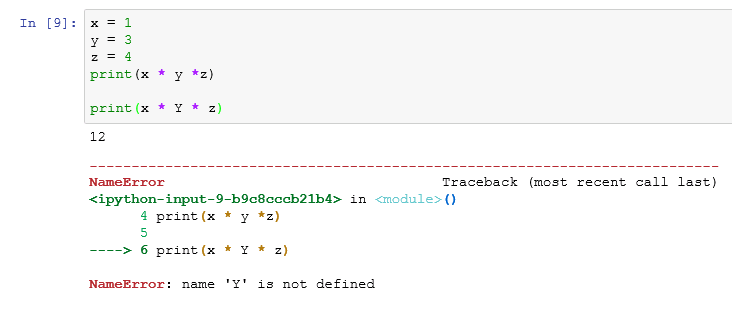
Return the sine of x (measured in radians).

When you put a doc string into functions you define you can get help on your functions. This is very nice when you need to remember the order of the parameters or what it returns.



Case Sensitivity

Note that Python is a case sensitive programming language. Meaning all variables, functions, and objects must be called by their exact spelling:

The “12” that is printed is the result of the first call to print, the error is the result of the second call that referred to the variable "Y" (uppercase) instead of "y" (lowercase).

Numeric types and operations

In addition to int, float, and complex there are two more numeric types in the standard library, decimal and fractions. Decimal types hold floating point numbers with user-definable precision. Fractions are numbers with an integer numerator and denominator. All numeric types (except complex) support the following operations:

x + y sum of x and y

x - y difference of x and y

x \* y product of x and y

x / y quotient of x and y

x // y floored quotient of x and y

x % y remainder of x / y

-x negated

+x unchanged

abs(x) absolute value or magnitude of x

int(x) x converted to integer

float(x) x converted to floating point

complex(re, im) a complex number with real part re, imaginary part im.

im defaults to zero if not present.

complex()

c.conjugate() conjugate of the complex number c

divmod(x, y) the pair (x // y, x % y)

pow(x, y) x to the power y

x \*\* y x to the power y

Numpy, Nan, Inf, and Vectorization

When we start venturing out to do computation for scientific purposes the numeric types build into Python are not sufficient. To handle this, we turn to the numpy (numeric python) package.

Numpy features arrays that are like Python lists except all the elements are the same type (type default: this is float64). The standard math functions above are extended so they work across all the elements of an array. This is actually done in C code that is called by the Python function in numpy so that the computation is quite fast.

Numpy extends the data types with two new elements, np.nan and np.inf. np.nan represents Not-a-Number and is returned from a function in the case of things like a domain error. Any mathematical operations that take in a NaN also return one. np.inf is a floating point constant with no exact value since it represents infinity. In any comparisons, the variable with a value of inf will be the greater one unless the other variable also holds inf.

As an example, enter the following in a cell of your notebook and execute it.

import numpy as np

a = np.array([0, 1, -1, np.inf, np.nan])

print(1/a)

print(np.sqrt(a))

for elem in a:

print(elem)

The result is

[ inf 1. -1. 0. nan]

[ 0. 1. nan inf nan]

0.0

1.0

-1.0

inf

nan

~/anaconda3/lib/python3.6/site-packages/ipykernel/\_\_main\_\_.py:3: RuntimeWarning: divide by zero encountered in true\_divide

app.launch\_new\_instance()

~/anaconda3/lib/python3.6/site-packages/ipykernel/\_\_main\_\_.py:4: RuntimeWarning: invalid value encountered in sqrt

Note that once we create the array we can take its inverse and the inverse operation is applied to every member of the array. If we take the square root of the array similarly the sqrt operation is applied to every member. The inverse of inf is 0 and the inverse of 0 is inf. Operations applied to the NaN in the last position always return a NaN. Taking the square root of -1.0 returns a NaN. We can loop over the elements in array just like we loop over elements in a Python list.

A key difference between a numpy array and lists or arrays in many other languages is a topic known as vectorization. What does this mean? It means that many functions that are to be applied individually to each element in a vector of numbers require a loop assessment to evaluate; however, in numpy many of these functions have been coded in C to perform much faster than a for loop would perform. For example, let’s say you want to add the elements of two separate arrays of numbers (x and y).

# add two lists pair-wise in base Python

lx = [1, 3, 4]

ly = [1, 2, 4]

lsum = []

for x\_elem, y\_elem in zip(lx, ly):

lsum.append(x\_elem + y\_elem)

print(lsum)

# same thing but with numpy array and vector operator

x = np.array([1, 3, 4])

y = np.array([1, 2, 4])

print(x+y)

The produces the two lines

[2, 5, 8]

[2 5 8]

In the first example, we have to loop over the lists of numbers. "zip" is a standard Python built in function that takes two sequences and returns the first elements from each sequence, then the second elements, then the third, etc. until the shortest list runs out of elements. The result list has to be created before the loop as an empty list and then we append each sum as it is calculated. With numpy arrays however the "+" operator has been extended so that it looks to see that each side being added is the same size and then in one call does the addition for every element pair. If you were to pass in two lists of different sizes into the first example, the explicit loop, it would run without an error even though this is probably not what is wanted when adding vectors. In numpy however two arrays of different sizes will result in an error being raised and the calculation will stop.

You can also do other operations, such as comparisons, on two arrays of the same length and get all the elements compared at once.

print(x>y)

[False True False]

Note that when a numpy array is printed the elements are separated by spaces, but when a Python list is printed the elements have commas between them.

When one side of an operation is a constant or has a length of 1 then numpy will extend the constant or array to match the dimension of the other side.

print(x+5)

print(x+np.array([5]))

Both produce the result

[6 8 9]

But if you have differing dimensions with a length that is not one you get an error:

print(x+np.array([5, 1]))

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-68-50341737a046> in <module>()

---> 18 print(x+np.array([5, 1]))

ValueError: operands could not be broadcast together with shapes (3,) (2,)

Style

“*Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read*.” - Hadley Wickham

As code is a medium of communication, it is important to realize that the readability of code does in fact make a difference. Well styled code has many benefits to include making it easy to i) read, ii) extend, and iii) debug. Unfortunately, Python does not enforce its guidelines for code styling. However, this should not lead you to believe there is no style to be followed. Over time implicit guidelines for proper code styling have been documented.

The standard style guide for Python is known as [Pep8](https://www.python.org/dev/peps/pep-0008/).

If you are using Pep8 style you make all indentations a multiple of 4, you use only spaces for indenting not tabs, you put spaces around all operators and if you have to break a line you do it after (not before) a binary operator like + or -. There are a lot more rules but these are some of the ones you will use most often. The rules are not too strict and emphasize readability.

Organization

Organization of your code is also important. There’s nothing like trying to decipher 2,000 lines of code that has no organization. The easiest way to achieve organization is to comment your code. Working in the Jupyter notebook can help. Define one function per cell and write doc strings. The work in a notebook should flow from the top to the bottom even if in getting it working you are jumping around. Break logical groupings of operations together into functions instead of having a long sequence of code. If you would need to explain what is being done at a point in the code if someone else were reading it, put in a comment. Don't comment the obvious. Comments should explain why something is being done, not what is being done.

This is an example of a bad comment:

x = x + 1 # increase x by 1

Many statistical analyses follow a sequence of Extract - Transform – Load, where Extract means reading in the data, Transform is preparing the data for the statistical analysis, and Load is reading the transformed data into the analysis functions. This preparatory phase is referred to as ETL. Usually, this is followed by a Report or Visualization phase. Write your code so that these phases are separate and can easily be found by reading the comments.

Chapter 3: Data File Manipulation

The first step to any data analysis process is to get the data. Data can come from many sources but two of the most common include text and Excel files. The first section of this module covers how to import data into Python by reading data from common text files and Excel spreadsheets and shows how to view this data. For purposes of this chapter we’ll use the [CustomerData\_Merrimack.csv](https://www.dropbox.com/s/ke4ucb7rg1ldswi/CustomerData_Merrimack.csv?dl=1) and [CustomerData\_Merrimack.xlsx](https://www.dropbox.com/s/wmqfxsc9cw8xsud/CustomerData_Merrimack.xlsx?dl=1) files to illustrate.

Once we have imported our data we usually want to do some initial investigation to understand the structure of our data and perform basic data manipulations such as filtering, selecting, arranging, creating, and summarizing variables. We may also want to join separate data sets and take care of missing values. The second section of this module covers these fundamental activities.

Importing data

Reading data from text files

Text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Text file formats use delimiters to separate the different elements in a line, and each line of data is in its own line in the text file. Therefore, importing different kinds of text files can follow a fairly consistent process once you’ve identified the delimiter.

Although there is a module in the base Python library to read in csv files (called [csv](https://docs.python.org/3.6/library/csv.html#module-csv)) we are going to use a function in the pandas library. The pandas library is the primary Python module to handle statistical data. To read in our .csv file we first import the pandas module and then use [read\_csv](http://pandas.pydata.org/pandas-docs/stable/io.html#csv-text-files) to load the data and save it as an object named customer. Within read\_csv we often simply need to provide the path to the data. In this example, I have downloaded the CustomerData\_Merrimack data and saved the file in a “data” sub folder in my working directory. (To show what you type versus what is output I have put “>>> “ in front of lines to be input. These are not meant to be input, although if you do input them into a cell of a Jupyter notebook they will be ignored.)

>>> import pandas

>>> customer = pandas.read\_csv("data/CustomerData\_Merrimack.csv")

(there is no output printed)

And that is it. The data is now in the variable ‘customer’.

read\_csv offers many options for reading in data, which you can read about [here](http://pandas.pydata.org/pandas-docs/stable/io.html#basic). However, for simply importing the data set into Python the default arguments work just fine. The [pandas I/O Tools](http://pandas.pydata.org/pandas-docs/stable/io.html#io-tools-text-csv-hdf5) also offers functions to import a wide variety of data formats including json, Python’s pickle format, and data from SQL and other statistical packages such as SAS.

A fixed-width data format can be read in from a text file with the function [pandas.read\_fwf](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_fwf.html#pandas-read-fwf). This function takes as it first parameter the name of the file to read from and then a second keyword parameter, either colspecs= or widths=. colspecs is a list of pairs ints giving the column where the field starts and the column one past the end of the field. So, a specification like [(0,5), (5,11)] would say there are two fields, the first is in column 0 to 4 and the second is in columns 5 to 10. The width of the field is always the difference between the two integers in the pair. In this example the two fields are right next to each other so we could instead specify the fields positions with the widths parameter. The value of the widths parameter is a list of integers giving the widths of the fields. If the first field is 5 characters wide and the second is 6 the value for the widths parameter is [5,6].

Reading data from Excel files

With Excel still being the spreadsheet software of choice it is important to be able to efficiently import and export data from these files. Often, Python users will simply resort to exporting the Excel file as a .csv file and then import into Python using pandas.read\_csv; however, this is far from efficient. This section will teach you how to eliminate the .csv step and to import data directly from Excel.

This pandas.read\_excelmethod works with both legacy .xls formats and the modern xml-based .xlsx format. Similar to pandas.read\_csv, the pandas.read\_excel functions are based on a C++ library so they are extremely fast. Unlike most other packages that deal with Excel, pandas.read\_excel has no external dependencies, so you can use it to read Excel data on just about any platform.

import pandas

customer = pandas.read\_csv("data/CustomerData\_Merrimack.xlsx")

This code reads in the first data sheet in the file and loads it into the ‘customer’ dataframe.

To read in Excel data with pandas.read\_excel you will commonly use the pandas.excel\_file object and its read\_excel() function. The excel\_file.sheet\_names property (not a function, so don’t follow it with parentheses) allows you to read the names of the different worksheets in the Excel workbook.

>>> xlsx = pandas.ExcelFile('data/CustomerData\_Merrimack.xlsx')

>>> print(1, xlsx.sheet\_names) # a property: list of sheet names

>>> df = pandas.read\_excel(xlsx, 'Sheet1') # bring in a sheet as a dataframe

>>> print(2, len(df))

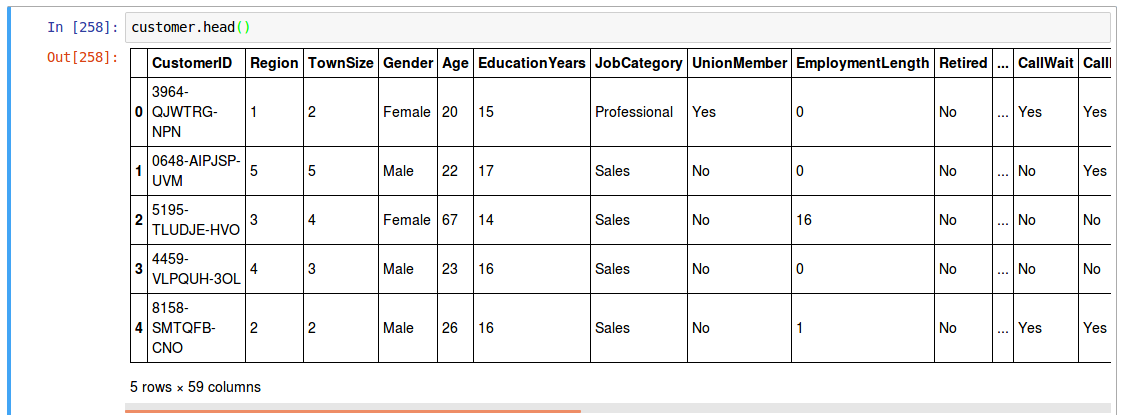
1 ["Sheet1", "Sheet2", "Sheet3"]

2 5000

In this case, CustomerData\_Merrimack.xlsx does not have named spreadsheets. If you look at the workbook you’ll see that all the data is contained in the first spreadsheet. There are many options that you can incorporate (i.e. skipping lines or columns, changing the variable names, etc.). You can check out many of these more advanced options [here](http://pandas.pydata.org/pandas-docs/stable/io.html#excel-files); however, for most basic Excel spreadsheets the default arguments will suffice.

**Viewing the data**

Once you have imported the data there are several ways to get an initial view of this data prior to performing any analysis. First, you can ask for the [head()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.head.html#pandas.DataFrame.head). This shows us the first five rows of data in a sliding browser frame, and shows us that the data has 59 variables.



To see the names of all the columns you can print the columns property. This also shows the dtype of all the objects if they are the same. In this case they differ so the dtype shown at the end is ‘object’.

>>> print(customer.columns)

Index(['CustomerID', 'Region', 'TownSize', 'Gender', 'Age', 'EducationYears',

'JobCategory', 'UnionMember', 'EmploymentLength', 'Retired', 'HHIncome',

'DebtToIncomeRatio', 'CreditDebt', 'OtherDebt', 'LoanDefault',

'MaritalStatus', 'HouseholdSize', 'NumberPets', 'NumberCats',

'NumberDogs', 'NumberBirds', 'HomeOwner', 'CarsOwned', 'CarOwnership',

'CarBrand', 'CarValue', 'CommuteTime', 'PoliticalPartyMem', 'Votes',

'CreditCard', 'CardTenure', 'CardItemsMonthly', 'CardSpendMonth',

'ActiveLifestyle', 'PhoneCoTenure', 'VoiceLastMonth', 'VoiceOverTenure',

'EquipmentRental', 'EquipmentLastMonth', 'EquipmentOverTenure',

'CallingCard', 'WirelessData', 'DataLastMonth', 'DataOverTenure',

'Multiline', 'VM', 'Pager', 'Internet', 'CallerID', 'CallWait',

'CallForward', 'ThreeWayCalling', 'EBilling', 'TVWatchingHours',

'OwnsPC', 'OwnsMobileDevice', 'OwnsGameSystem', 'OwnsFax',

'NewsSubscriber'],

dtype='object')

And to see the types of the columns look at the dtypes property.

>>> print(customer.dtypes)

CustomerID object

Region int64

TownSize object

Gender object

Age int64

EducationYears int64

JobCategory object

UnionMember object

EmploymentLength int64

Retired object

HHIncome float64

DebtToIncomeRatio float64

CreditDebt float64

OtherDebt float64

...

Additional resources

In addition to text and Excel files, there are multiple other ways that data are stored and exchanged. Commercial statistical software such as SPSS, SAS, Stata, and Minitab often have the option to store data in a specific format for that software. In addition, analysts commonly use databases to store large quantities of data. Pandas has good support to work with these additional options which we did not cover here. The following provides a list of additional resources to learn about data importing for these specific cases:

* Pandas manual section on [I/O Tools](http://pandas.pydata.org/pandas-docs/stable/io.html#io-tools-text-csv-hdf5)
* Working with databases is done with [pandas.io.sql](http://pandas.pydata.org/pandas-docs/stable/io.html#sql-queries) functions. Python has a standard SQL interface that allows a Python program to run essentially unchanged, except for authentication information. A driver module must be present. Anaconda comes with drivers for Oracle, MySQL, PostgreSQL, SQLite, sybase, firebird, and Open Database Connectivity databases.
* Importing data from other commercial software: The pandas reader functions help you load data files from other programs such as [SAS](http://pandas.pydata.org/pandas-docs/stable/io.html#sas-formats), [Stata](http://pandas.pydata.org/pandas-docs/stable/io.html#reading-from-stata-format), and [GoogleBigTable](http://pandas.pydata.org/pandas-docs/stable/io.html#google-bigquery).
* There are also reader functions that load from non-proprietary formats like [feather](http://pandas.pydata.org/pandas-docs/stable/io.html#feather), [msgpack](http://pandas.pydata.org/pandas-docs/stable/io.html#msgpack) and [JSON](http://pandas.pydata.org/pandas-docs/stable/io.html#reading-json).
* Data from R can be loaded through the [rpy2](http://pandas.pydata.org/pandas-docs/stable/r_interface.html#rpy2-r-interface) interface to R.
* Reading data written by SPSS into Python can be done with libraries from IBM (publisher of SPSS) or using rp2 interface to have R do the reading. Information is [here](https://stackoverflow.com/questions/14647006/is-there-a-python-module-to-open-spss-files).

Exercises

1. Download and read in this [flights.csv](https://www.dropbox.com/s/jtkdultbfp2a6sk/flights.csv?dl=1) file. Can you figure out how to read in the first line to see the

titles? Try reading in the first 1,000 lines and only the first 6 columns (check out the [documentation](http://pandas.pydata.org/pandas-docs/stable/io.html#csv-text-files) for read\_csv).

2. Read in this [facebook.tsv](https://www.dropbox.com/s/bpmgrke55lcw13g/facebook.tsv?dl=1) file. This is a fixed-width column file so you will have to determine the width of the fields.

3. What spreadsheets are in this [mydata.xlsx](https://www.dropbox.com/s/mbmy3u36u6un463/mydata.xlsx?dl=1) file? Practice reading in the different spreadsheets.

4. What spreadsheets are in this [PEW Middle Class Data.xlsx](https://www.dropbox.com/s/p6pcqf52acfktcp/PEW%20Middle%20Class%20Data.xlsx?dl=1) file? Can you read in the 3.Median HH

income, metro spreadsheet (hint: you’ll need to skip a few lines)?

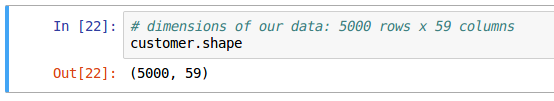
Data Frames

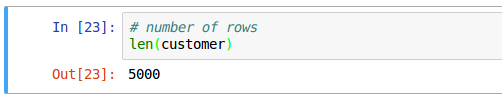
The customer data that we loaded in the previous section is available in Python as an object of type pandas.DataFrame. A data frame is the most common way of storing data in Pandas and, generally, is the data structure most often used for data analysis. Under the hood, a data frame is a C language structure that can be thought of as a list of equal-length vectors. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. As a result, data frames can store different types of objects in each column (i.e. float, integer, string). In essence, the easiest way to think of a data frame is as an Excel worksheet that contains columns of different types of data but are all rows are of equal length.

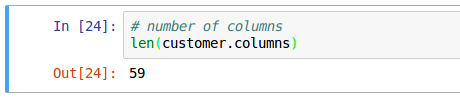
Understanding the structure

Besides viewing our data as we did in the last section we can get a quick understanding of our data frame with a few functions and properties of a data frame object. First, we can get the dimensions of our data together or individually:

The dimensions of our data are in the shape property.

The number of rows is the len of the data frame.

The number of columns is the len of the columns property.



We can also check out the names of our variables with the columns property.



Creating Data Frames

Although we read in our data, which created our customer data frame, we can also create data frames

within Python code with the [pandas.DataFrame](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas-dataframe) class constructor. In this case I’ll create a simple 3x4 data frame (which gets named df) and then I will assess its basic structure:

# math and pandas must already be imported. uncomment as needed

# import math

# import pandas

# define the columns as lists of data

col1 = list(range(1,4)) # list of numbers from 1 until before 4

col2 = ["this", "is", "text"]

col3 = [True, False, True]

col4 = [2.5, 4.2, math.pi]

# data to pass is a dictionary mapping column names to col data

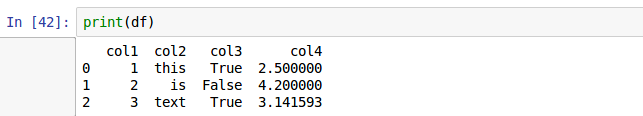
data = { 'col1':col1, 'col2':col2, 'col3':col3, 'col4':col4 }

# create the DataFrame

df = pandas.DataFrame(data)

You can now assess the structure of a data frame. This is small enough you can print the whole thing out.

.

Now let’s use some of the DataFrame’s properties to access information about this particular DataFrame.

# Retrieving general information about a DataFrame

print(df,'\n') # print the entire DataFrame followed by blank line

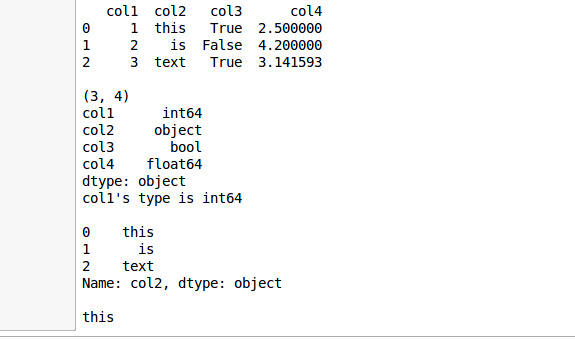
print(df.shape) # how many rows and columns

print(df.dtypes) # What are the types of all the columns

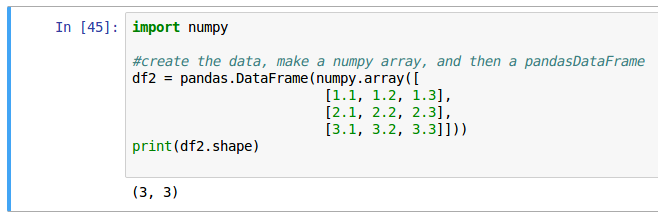
print("col1's type is", df.dtypes['col1'], '\n') # type of a column

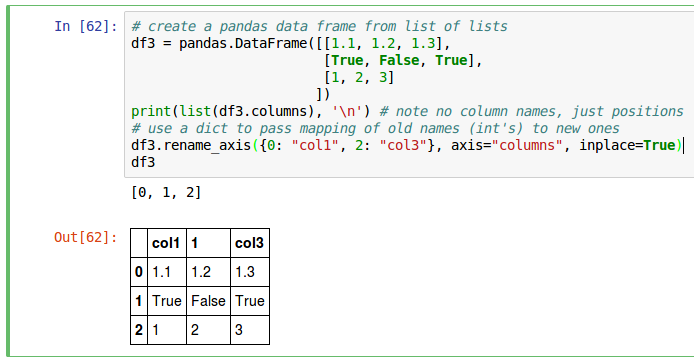
print(df['col2'], '\n') # retrieve all of a column

And this code returns



We can also convert a pre-existing numpy.array to a data frame. The following illustrates how we can turn a list or a numpy.array into a data frame.

Actually, if you don’t have a numpy array you can create a pandas.DataFrame directly from a list of lists. The columns won’t have any names, just positions. You can rename the columns you are interested in to more useful string names using a call to rename\_axis.



Data transformation

Transforming your data is a basic part of data wrangling. This can include filtering, summarizing, and ordering your data by different means. This also includes combining disparate data sets, creating new variables, and many other manipulation tasks. Many fundamental data transformation and manipulation functions exist in pandas.

If you are coming back to this section having just brought up a new Jupyter notebook remember to import pandas.

import pandas

Also, we’ll continue to use our customer data (the loaded CustomerData\_Merrimack.xlsx data that is in the variable ‘customer’) to illustrate.

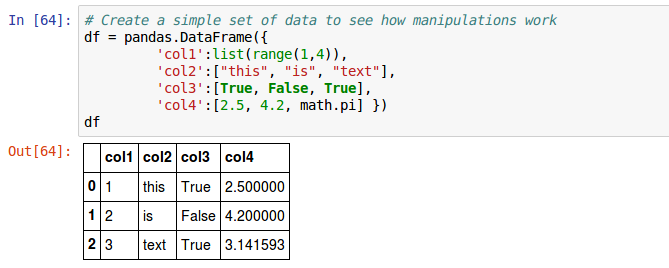
In this section I’ll illustrate the six key data manipulations that allow you to solve the vast majority of your data manipulation challenges:

* Pick observations by their values.
* Pick variables by their names.
* Reorder the rows.
* Create new variables with functions of existing variables.
* Collapse many values down to a single summary.
* Combine multiple operations with the pipe operator.
* Join separate data sets.

These can all be used in conjunction with DataFrame.group\_by() which changes the scope of each function from operating on the entire data set to operating on it group-by-group.

These operations are similar in that

* They are functions of a DataFrame;
* The parameters of the function describe what is to be done;
* The result is a new DataFrame.

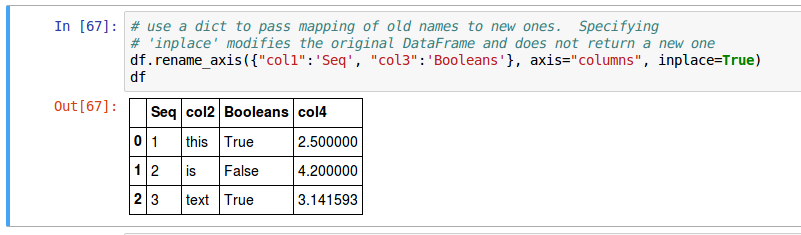
Together these properties make it easy to chain together multiple simple steps to achieve a complex result. 

Let’s dive in and see how these functions work. To show you a simple case of the manipulations that should be easier to follow, the discussion will first be about this simple DataFrame named ‘df’.

Renaming columns

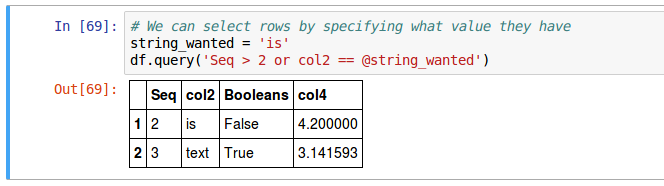
Once you have a DataFrame created from someone else’s data you may not agree with the column names they used (or the names you used in the past). The [DataFrame.rename\_axis](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.rename_axis.html#pandas-series-rename-axis) call allows you to either create a new DataFrame with renamed columns, or modify the existing DataFrame in place. This function will be used to make a permanent change to the data. If you do not specify “inplace=True” in your call this function works like the rest and returns a new DataFrame with the requested changes. If you do specify “inplace=True” in the parameter list the change is made to the caller and nothing is returned.

The old names and new names are passed to “rename\_axis” function in the form of a dictionary object with the keys being the old names and the values being the new names. In this example below the new names are specified in a dict that is created in the call. Remember, a dictionary is created by specifying curly braces (“{“ and “}” around a comma separated sequence of pairs, each of with has a colon between the parts, key and value, of the pair.) If there were any more columns to be renamed the dict should be created and assigned to a variable and the variable used in the call. This is to help with readability. Readability is a great help if you ever want to return to the notebook to copy code for a similar analysis.



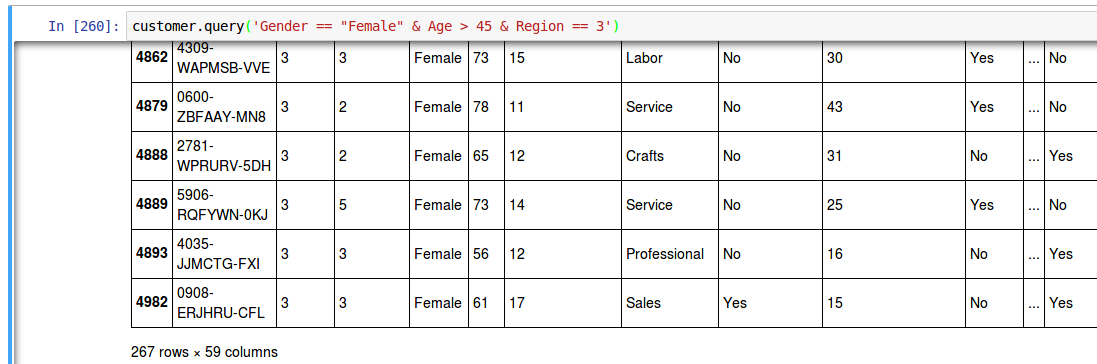
Pick rows of observations by their values.

To pick out rows where a particular variable’s value meets some criteria we call the [DataFrame.query](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.query.html?highlight=query#pandas-dataframe-query) function. You pass a boolean expression in as the parameter which will then be evaluated for each row to see if the row should be returned. If you want to refer to Python variable values in the query string prefix the variable name with an at-sign(@).



To use querying effectively, you have to know how to select the observations that you want using the comparison operators. DataFrame.query supports the standard suite: >, >=, <, <=, != (not equal), and == (equal).

We can add additional conditions to filter by. For example, if we want to filter our customer file for female customers that are older than 45 in region 3:



Note that multiple conditions are joined with a single ampersand (“&”) and not the double ampersand that would be used in a Python language if statement. You can also use the symbol “|” as “or” in the query statement. If there is possible ambiguity about the order in which operations are to be done you can group operations with parentheses. A statement “A=’a’ | B=’b’ & C=’c’” needs parentheses since it could mean either “(A=’a’ | B=’b’) & C=’c’” or “A=’a’ | (B=’b’ & C=’c’)”.

What if want to find all female customers over 45 that live in region 3 or region 5. There are two ways to approach this. We could write it as a query with an “or” expression included in parentheses.

customer.query('Gender == "Female" & Age > 45 & '

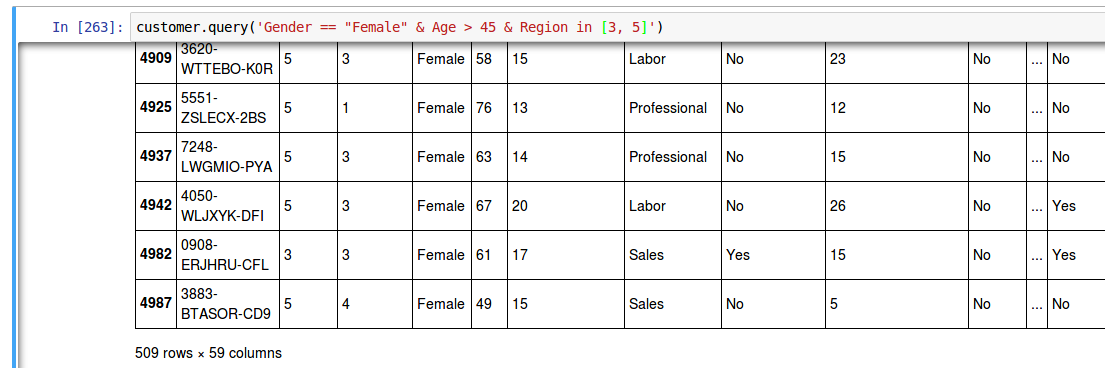
'(Region == 3 | Region == 5)')

(Remember that in Python, within the parentheses of a function call a string can be continued on another line by closing the quote on the first line, having only newlines and spaces until an opening quote on the next line and then continue the string.)

Or we could use the “in” keyword that takes a column name and a list of values.

customer.query('Gender == "Female" & Age > 45 & Region in [3, 5]')

You can see using the “in” operator reduces the amount of code to type and is usually clearer to read. Go ahead and try these in your Jupyter notebook.

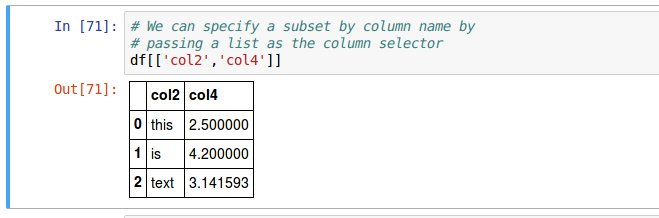


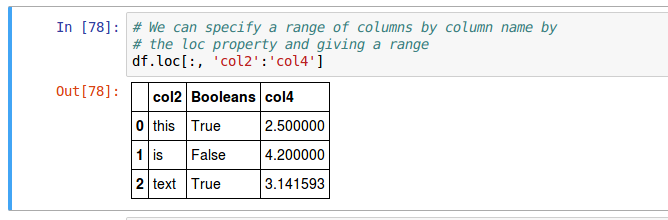
Pick columns by their names.

To pick a single column from a DataFrame you specify the name with square brackets after the DataFrame name.

df['Seq']

More useful is to specify a number of columns. It’s not uncommon to get data sets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you’re actually interested in. To retrieve just a selected set of columns you pass a list within the square brackets (which gives a funny looking double square brackets syntax).

You can also request a range of columns by using the [loc](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html#pandas.DataFrame.loc) property. The first parameter in the list below is a colon indicating we want all rows, and the second is a range of columns.



In the customer DataFrame we can select specific columns by name with:

customer.loc[:, ['CustomerID', 'Region', 'TownSize', 'Gender']]

And we can select all columns between CustomerID and Gender with

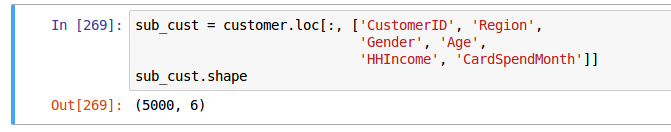
customer.loc[:, 'CustomerID':'Gender']

Let’s go ahead and reduce the number of variables that we are using for the rest of this chapter.

sub\_cust = customer.loc[:, ['CustomerID', 'Region',

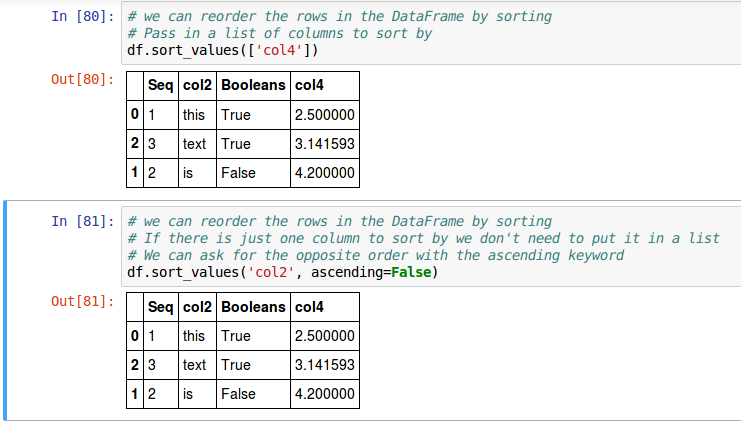
'Gender', 'Age',

'HHIncome', 'CardSpendMonth']]



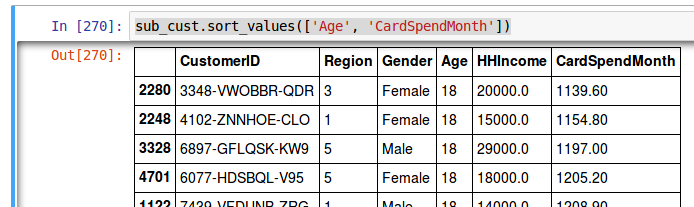
Reorder the rows

The order of the rows in the DataFrame can be modified by sorting them with the [DataFrame.sort\_values](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.sort_values.html#pandas-dataframe-sort-values) function. The first parameter is a column name to sort by or a list of column names to sort in order. If you want the sort result to be the reverse of a normal sort specify an additional parameter ascending=False.



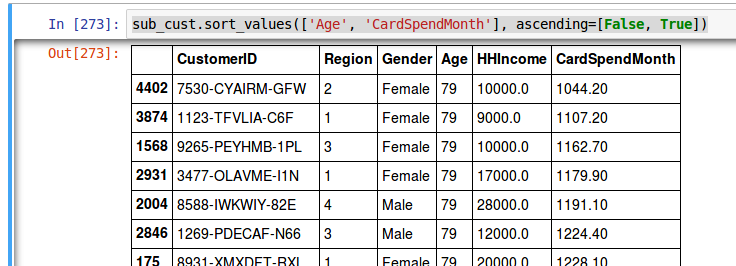
Returning to our cut-down customer data we can sort it by age and card spending with

sub\_cust.sort\_values(['Age', 'CardSpendMonth'])



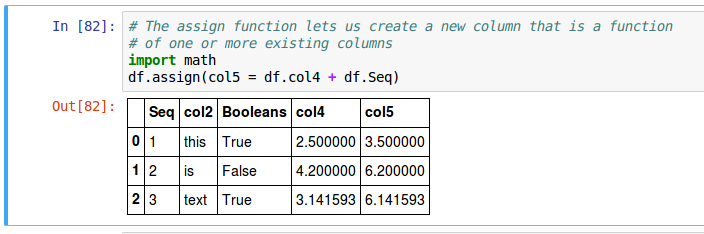
Or we might ask for highest ages first but still smallest balance first. To do this, the ascending parameter value must be a list specifying for each variable in the sort-by list the direction of the sort for that column.

sub\_cust.sort\_values(['Age', 'CardSpendMonth'], ascending=[False, True])

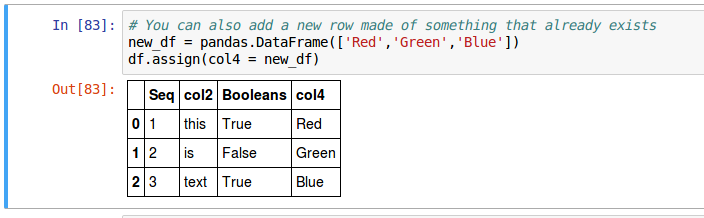


Create new variables with functions of existing variables

You can create a new column and for the new column’s value specify a mathematical function of other column values with the [assign](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.assign.html#pandas.DataFrame.assign) function.



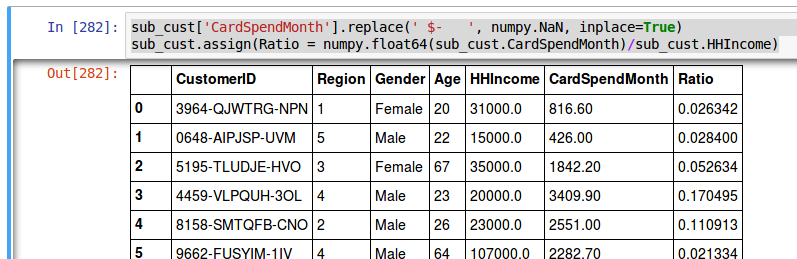
You can also calculate the variables in another list and add the list to the DataFrame with assign.



So, in our sub\_cust DataFrame we can calculate the ratio of card spending to income, and assign it to a new column Ratio. Because a null value in CardSpendMonth is encoded as the string ' $- ' we first need to change these values to a null that will work with arithmetic. This will be discussed further in a few pages. The next line calculates the Ratio column for all rows and adds the column to the returned DataFrame.

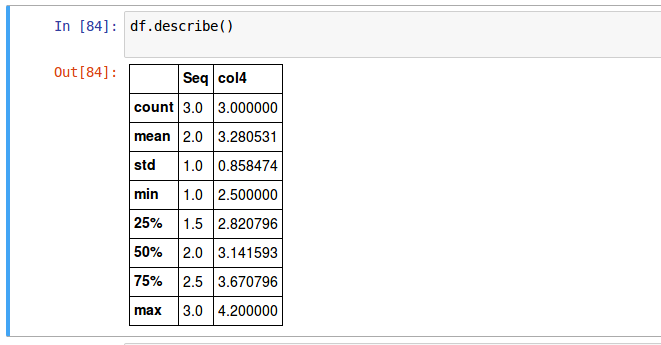
sub\_cust['CardSpendMonth'].replace(' $- ', numpy.NaN, inplace=True)

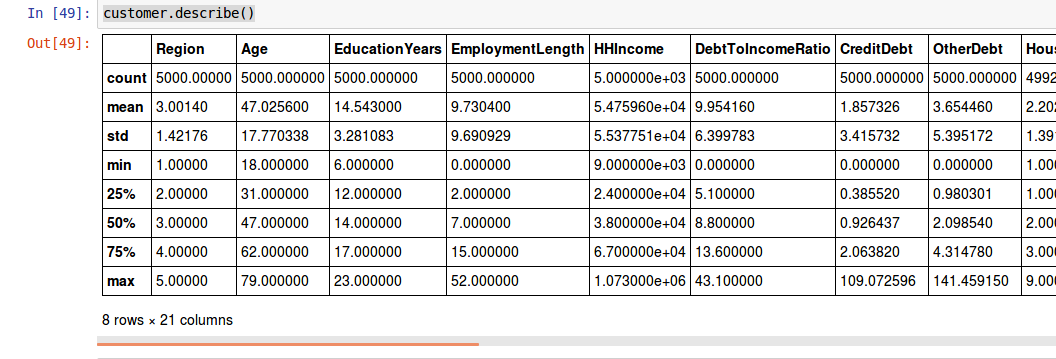
sub\_cust.assign(Ratio = numpy.float64(sub\_cust.CardSpendMonth)/sub\_cust.HHIncome)



Collapse many values down to a single summary

There is a quick function [describe](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.describe.html#pandas.DataFrame.describe) that calculates common summary statistics for numerical columns of the DataFrame.

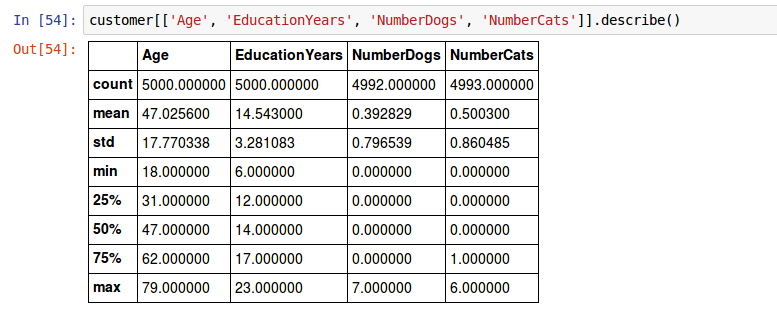
This isn’t too interesting. We should describe the customer data.



Combining multiple operations

Because the return from the functions we have looked at is a DataFrame we can combine operations by following one call with another call.

In the example below a new DataFrame with just the named columns is created and then describe is called.

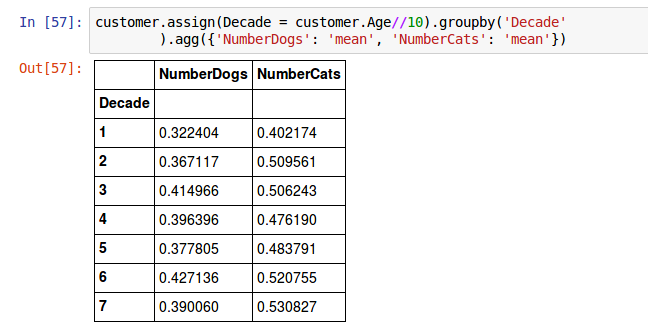


This is interesting. In this dataset, at least 75% of the people have no dog and 50% have no cat.

Let’s create some data about how pet ownership varies by age.

customer.assign(Decade = customer.Age//10).groupby('Decade'

).agg({'NumberDogs': 'mean', 'NumberCats': 'mean'})



Here we first call assign and create a new column Decade that is the person’s age divided by 10. (The double slash divisor is Python’s way of saying that we want an integer from the division, not a fraction.) We then call [groupby](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html#pandas.DataFrame.groupby) to create groups for the analyses, grouping on the column that was just created. Lastly we call [agg](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.agg.html#pandas.DataFrame.agg) to ask for the mean of a couple of the columns.

Join separate data sets

Often we have separate data frames that can have common and differing variables for similar observations and we wish to join these data frames together. pandas offers the [join](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.join.html#pandas-dataframe-join) function that provides multiple ways to join data frames in its ‘how’ parameter: ‘left’, ‘right’, ‘outer’, or ‘inner’.

To help you learn how joins work, we’ll use the following data:

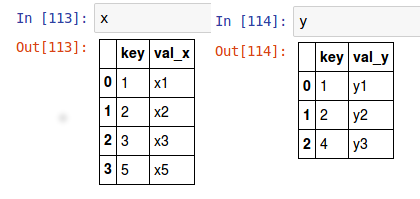
Data for Joins

x = pandas.DataFrame({'key':[1, 2, 3, 5],

'val\_x':['x1', 'x2', 'x3', 'x5']})

y = pandas.DataFrame({'key':[1, 2, 4],

'val\_y':['y1', 'y2', 'y3']})



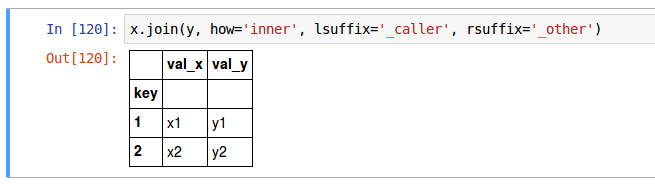
To do a join you have to set an index on the fields you are going to match. This is done with the [set\_index](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.set_index.html#pandas-dataframe-set-index) function. The first, and required, parameter is a list of column names that will become the key. Since I intend to join these tables multiple times I pass the inplace parameter set to True and the table is modified.

x.set\_index(['key'], inplace=True)

y.set\_index(['key'], inplace=True)

To do a join

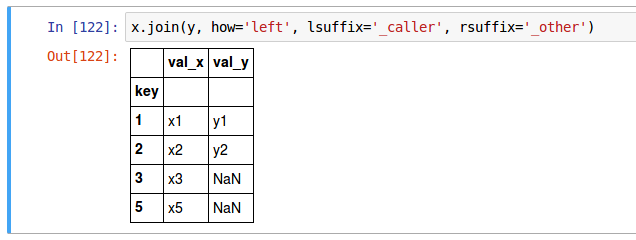
An inner join matches pairs of observations whenever their keys are equal:



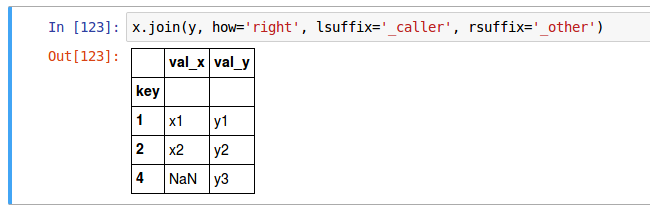
An inner join keeps observations that appear in both tables.

The remaining three types of joins add elements to columns from one or both of the tables. When a key value is present in one table but is not present in the other the value for columns from the other table are set to NaN (Not a Number). We will discuss NaN more below.

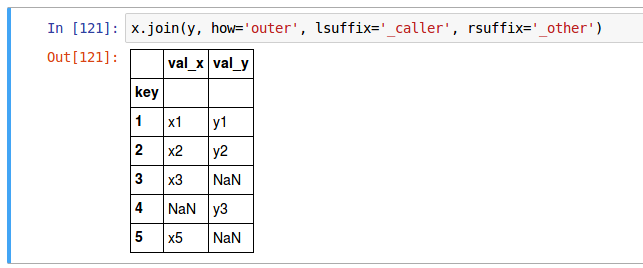
• A left join keeps all observations in x, the first table and values where the key is not present in the second table get set to NaN.



• A right join keeps all observations in y even if there is not a matching key in x.



• An outer join keeps all observations in x and y.



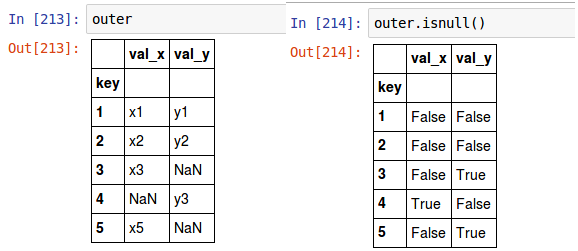
There are additional ways to join data to include filtering joins and set operations. You can learn more about these joining methods and managing relational data sets [here](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.join.html#pandas.DataFrame.join).

Managing missing values

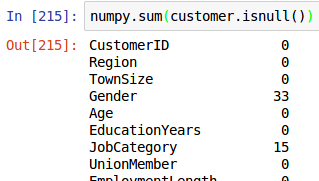
A common task in data analysis is dealing with missing values. In pandas, missing values are often represented by NaN or some other value that represents missing values (i.e. 99). In the previous section we saw how to filter individual variables for missing values and how to exclude them from summary statistics. In this section I illustrate how to test for, recode, and exclude missing values across your entire data frame.

Testing for missing values

To identify missing values, [pandas.Series.hasnans](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.hasnans.html?highlight=nan#pandas.Series.hasnans) is a property which returns True for columns that contain missing values represented by NaN. To identify missing values in a DataFrame you can use the isnull() function which return a DataFrame with the same shape as the DataFrame it was applied to but is all boolean values indicating whether a null was present in that element.



For a realistic size of dataset, you’ll realize that it is not very useful. However, we can quickly identify the number of missing values across our data set by wrapping the DataFrame.isnull() with numpy.sum():



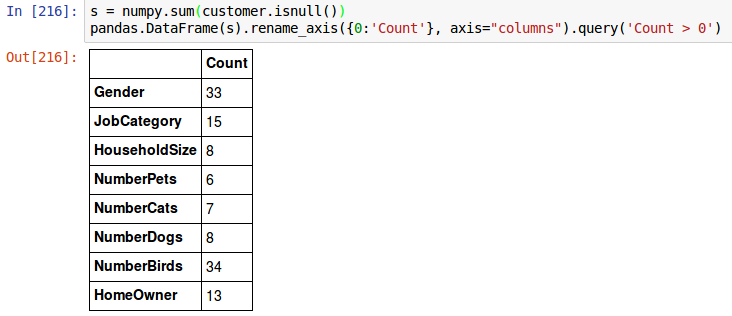
I haven’t shown all the output since it has a lot of 0’s mixed in. We would like to not show those but the result returned from sum is not a DataFrame that we know how to manipulate. We can construct a DataFrame with the vector as input and then use the rename\_axis call to give the column a name, and then use the query call to drop the 0’s. Here we see that our missing values are concentrated in 8 of our 59 variables.

s = numpy.sum(customer.isnull())

pandas.DataFrame(s).rename\_axis({0:'Count'}, axis="columns"

).query('Count > 0')

(Note that we can insert a newline in to make the code more readable if we are inside parenstheses.)



Recoding missing values

To re-code missing values; or re-code specific indicators that represent missing values, we can use normal subsetting and assignment operations. For example, you may not have noticed but the CarOwnership and CarBrand variables have used the string “-1” to represent a non-applicable value (these folks do not have cars). If we wanted to re-code these values as NaN we can simply [replace](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.replace.html#pandas-dataframe-replace) the data for these elements and reassign the values. (the inplace=True parameter is passed since we are changing the value in the data and don’t want a copy with the changed data returned.)

customer['CarOwnership'].replace('-1', numpy.nan, inplace=True)

customer['CarBrand'].replace('-1', numpy.nan, inplace=True)

Now if we check how many missing values are in each column we see CarOwnership and CarBrand included in the list. This time sort\_values is also called with a reverse sort so that the column with the largest number of NaNs is first.

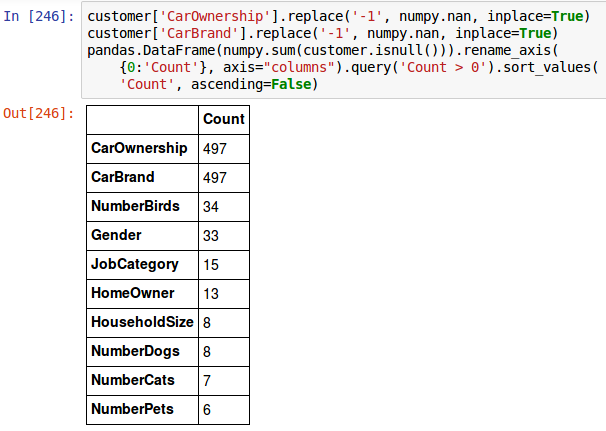
customer['CarOwnership'].replace('-1', numpy.nan, inplace=True)

customer['CarBrand'].replace('-1', numpy.nan, inplace=True)

pandas.DataFrame(numpy.sum(customer.isnull())).rename\_axis(

{0:'Count'}, axis="columns").query('Count > 0').sort\_values(

'Count', ascending=False)

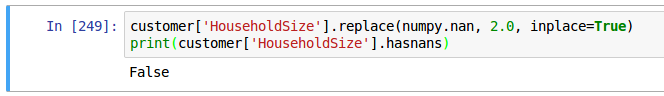


We can also use this same approach to impute new non-missing values for any of these variables. For example, if we wanted to assign the median house hold size to those customers where this data is missing we could do so. The median household size across all customers is 2.0.



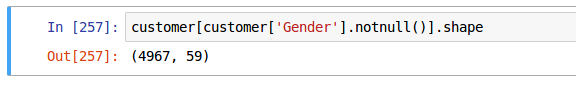
Thus, we can subset this variable for all those observations where the data is missing and add this value as the imputed value.

In the example below, we first replace the missing values with the median value. Then we use the hasnans property to check that our change was effective. hasnans is a property of a column (technically a DataFrame column is a pandas.Series) that is True or False depending upon whether there are any nulls in the column.



Excluding missing values

Observations with missing values can provide interesting insights; however, sometimes we desire to simply remove all observations with missing values. We can exclude missing values by using a selection to omit all rows containing missing values. In this case, we use the column’s [notnull](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.notnull.html#pandas.Series.notnull) function that generates a column of booleans which the DataFrame’s selection logic (what the square brackets are doing) filters out the False values.

.

If you want to exclude all rows with null values the [dropna](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.dropna.html#pandas-dataframe-dropna) function on the DataFrame can be used.

customer.dropna(axis=0, how='any')

Looking back to where we calculated the number of nulls in Gender we see that the number was 33. Adding 33 to the number of rows after the nulls are removed (4967) gives us the original number of 5000 rows.

Exercises

1. Install and load the [nycflights13](http://bit.ly/nycflights13) package. View the flights data that is built into this package.

2. What are the dimensions of this data set? What are the names of all the variables?

3. How many missing values are in the flights data? Omit all rows with missing values in them? Now

how many observations does your data have?

4. Filter this data for only those flights that occurred on December 25th.

5. Select the following variables: carrier, dep\_delay, arr\_delay.

6. Create a new variable titled “recoup” that subtracts dep\_delay from arr\_delay to signal how much time in the air the flight recouped.

7. Compute the mean recoup value by carrier

8. Arrange the summarized recoup values to see which carrier tended to recoup the most (or lose the least amount of time while in the air).

9. Now perform #4-8 above in one step by combining them into one statement that passes the data from one analysis to the next.

**Chapter 4: Exploratory Data Analysis**

This module will show you how to systematically explore your data, a task that statisticians call exploratory data analysis, or EDA for short. EDA is an important part of any analytics project and should be performed prior to applying more sophisticated modeling techniques to ensure you have a firm understanding of your data. Moreover, many fundamental, day-to-day questions that organizations are asking can often be found with simple EDA approaches.

EDA is an iterative cycle where you:

* Generate questions about your data.
* Search for answers by visualizing, transforming, and describing your data.
* Use what you learn to refine your questions and/or generate new questions.

Your goal during EDA is to develop an understanding of your data. The easiest way to do this is to use questions as tools to guide your investigation and then use the skills you’ve learned throughout this guide to answer these questions. Fortunately, there is no definitive process to perform EDA. Ultimately, it is a creative process with no hard rules. However, this module will illustrate several approaches to understand common attributes of your data.

Throughout this chapter we’ll perform EDA on the CustomerData\_Merrimack.xlsx data. Although not

comprehensive, this case study will exemplify the entire process to include:

* Compute descriptive statistics
* Visualize your data
* Perform basic statistical inference
* Identify and measure relationships between variables

**Prerequisites**

In this chapter we’ll be using several packages that provide fundamental EDA capabilities. Let’s install and load the Pandas module along with a few others we’ll use:

import pandas as pd

import numpy as np

import matplotlib as plt

This time I am giving these commonly used modules synonyms by using the “as” keyword in the import statement. In Python using synonyms for modules is generally considered to be bad style as it makes it harder for a reader (including yourself) to quickly determine what the code is doing. Within the data science community these three modules are used so often that these abbreviations are in common use and easily understood.

Let’s go ahead and read in our CustomerData\_Merrimack.xlsx data and remove observations with missing values. To drop the null rows the [dropna](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.dropna.html#pandas-dataframe-dropna) function of the DataFrame is useful. It can drop either complete rows (pass axis=0) or columns (pass axis=1) that contain nulls. The “how=’any’ parameter used here says to drop a row if it includes any nulls. It is also possible to drop rows that have over a specific number of nulls or rows where a specific column contains nulls.

customer = pd.read\_csv('data/CustomerData\_Merrimack.csv').dropna(axis=0, how='any')

**Descriptive statistics**

Descriptive statistics are the first pieces of information used to understand and represent a data set. Their goal, in essence, is to describe the main features of numerical and categorical information with simple summaries. These summaries can be presented with a single numeric measure, frequency distributions, using summary tables, and many other ways. Here, I illustrate the most common forms of descriptive statistics for numerical and categorical data, but keep in mind there are numerous ways to describe and illustrate key features of data.

**Numerical data**

Central tendencies

Variability

Outliers

Putting it together

**Central Tendency**

There are three common measures of central tendency, all of which try to answer the basic question of which value is the most “typical.” These are the mean (average of all observations), median (middle observation), and mode (appears most often). Each of these measures can be calculated for an individual variable or across all variables in a particular data frame. For example, if we are interested in finding the central tendency measures for the CardSpendMonth variable we can employ the following:

>>> customer['CardSpendMonth'].mean()

3381.8199673002337

>>> customer['CardSpendMonth'].median()

2775.2

Unfortunately, there is not a built-in function to compute the mode of a variable[ˆmode]. However, we can create a function that takes the DataFrame’s column as an input and gives the mode value as an output. Later, we’ll see easier ways to identify most common values (and value ranges).

>>> import collections

>>>

>>> def get\_mode(series):

>>> counter = collections.Counter(series)

>>> return counter.most\_common(1)

>>>

>>> get\_mode(customer['CardSpendMonth'])

[(0.0, 7)]

The answer tells us that the most common value is 0.0 which occurs 7 times. If there were other values that occurred 7 times those values would also be in the array of tuples that is returned.

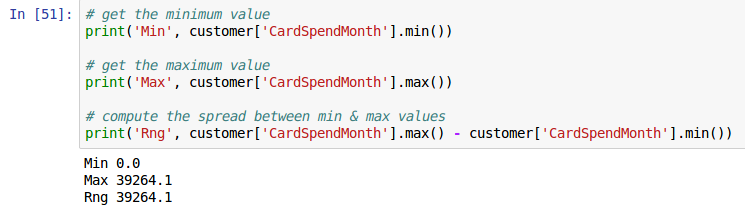
**Variability**

The central tendencies give you a sense of the most typical values (prices in this case) but do not provide you with information on the variability of the values. Variability can be summarized in different ways, each providing you a unique understanding of how the values are spread out.

**Range**

The range is a fairly crude measure of variability, defining the maximum and minimum values and the

difference thereof. We can compute range summaries with the [min](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.min.html#pandas.Series.min) and [max](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.max.html#pandas.Series.max) functions:

**Percentiles**

Given a certain percentage such as 25%, what is the dollar value such that 25% of cardholders are below it? This type of question leads to percentiles and quartiles. Specifically, for any percentage p, the pth percentile is the value such that a percentage p of all values are less than it. Similarly, the first, second, and third quartiles are the percentiles corresponding to p = 25%, p = 50%, and p = 75%. These three values divide the data into four groups, each with (approximately) a quarter of all observations. Note that the second quartile is equal to the median by definition. These measures are easily computed in Pandas with the [quantile](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.quantile.html#pandas-series-quantile) function on a Series.

>>> # quantile() takes a parameter of a number between 0 and 1.0. 0.0 is the min,

>>> # 1.0 is the max, and .5 is the median. .25 and .75 give 25% and 75% marks.

>>> print('Quartiles\n', customer['CardSpendMonth'].quantile([0.0, .25, .50, .75, 1.0]))

>>>

Quartiles

0.00 0.0

0.25 1836.4

0.50 2775.2

0.75 4194.6

1.00 39264.1

Name: CardSpendMonth, dtype: float64

>>> # we can customize quantile() for specific percentiles

>>> tenths = [x/10.0 for x in range(0,11)]

>>> print('Deciles\n', customer['CardSpendMonth'].quantile(tenths))

>>>

Deciles

0.0 0.00

0.1 1225.40

0.2 1650.60

0.3 2006.18

0.4 2377.48

0.5 2775.20

0.6 3233.32

0.7 3840.04

0.8 4629.72

0.9 6131.82

1.0 39264.10

Name: CardSpendMonth, dtype: float64

>>> # we can quickly compute the difference between the 1st and 3rd

>>> # quantile, the interquartile range

>>> import operator

>>> print('IQR', operator.sub(

\*customer['CardSpendMonth'].quantile([.75, .25])))

IQR 2358.2

An alternative approach is to use the [describe](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.describe.html#pandas-series-describe) function which is a generic pandas function used to produce min,1st quantile, median, mean, 3rd quantile, and max summary measures. However, though we do not see a difference here, note that the 1st and 3rd quantiles produced by summary may differ from the 1st and 3rd quantiles produced by the default quantile. The reason for this is due to the lack of universal agreement on how the 1st and 3rd quartiles should be calculated. Eric Cai provided a good [blog post](https://chemicalstatistician.wordpress.com/2013/08/12/exploratory-data-analysis-the-5-number-summary-two-different-methods-in-r-2/) that discusses this difference in the R functions.

>>> customer['CardSpendMonth'].describe()

count 4893.000000

mean 3381.819967

std 2464.986272

min 0.000000

25% 1836.400000

50% 2775.200000

75% 4194.600000

max 39264.100000

Name: CardSpendMonth, dtype: float64

**Variance**

Although the range provides a crude measure of variability and percentiles/quartiles provide an

understanding of divisions of the data, the most common measures to summarize variability are variance and its derivatives (standard deviation and mean/median absolute deviation). We can compute each of these using the pandas. Series functions var for variance, std for standard deviation, and mad for mean absolute deviation. Pandas does not have a function for median absolute deviation but we can easily write one.

# variance

>>> print('var', customer['CardSpendMonth'].var())

var 6076157.323241854

# standard deviation

>>> print('std', customer['CardSpendMonth'].std())

std 2464.9862724246263

# mean absolute deviation

>>> print('mad', customer['CardSpendMonth'].mad())

mad 1685.7853588644582

# median absolute deviation

>>> def medad(df, col):

>>> """Calculate median absolute deviation on a DataFrame df's column col."""

>>> med = df[col].median()

>>> if 'newcol' in df.columns:

>>> raise NameError('Cannot calculate median absolute deviation on a column named newcol')

>>> return df.assign(newcol = np.abs(

>>> med - df[col])

>>> )['newcol'].median() \* 1.4826

# median absolute deviation

>>> print('medad', medad(customer, 'CardSpendMonth'))

medad 1635.9008399999998

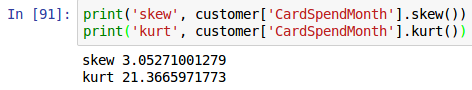
**Shape**

Two additional measures of a distribution that you will hear occasionally include skewness and kurtosis. Skewness is a measure of symmetry for a distribution. Negative values represent a left-skewed distribution where there are more extreme values to the left causing the mean to be less than the median. Positive values represent a right-skewed distribution where there are more extreme values to the right causing the mean to be more than the median.

Kurtosis is a measure of peakedness for a distribution. Negative values indicate a flat (platykurtic) distribution, positive values indicate a peaked (leptokurtic) distribution, and a value near 3 indicates a normal (mesokurtic) distribution.

We can get both skewness and kurtosis values using the moments package. Here, since our skewness is

positive we can tell monthly card expenditures is right-skewed and since the kurtosis is much greater than 3 this suggests a very peaked distribution.

**Outliers**

Outliers in data can distort predictions and affect their accuracy. Consequently, it’s important to understand if outliers are present and, if so, which observations are considered outliers. The scipy.stats module provides a number of useful functions to systematically extract outliers.

If you want to remove all rows that have extreme values you could do the following. For each column, first compute the Z-score of each value in the column, relative to the column mean and standard deviation. Then takes the absolute of Z-score (because the direction does not matter, only if it sufficiently far from the mean). Call DataFrame.all(axis=1) to ensure that for each row, all columns satisfy the constraint. Finally, use the result of this condition to index the dataframe.

>>> from scipy import stats

>>> customer[(np.abs(stats.zscore(customer)) < 3).all(axis=1)]

Another approach would be for each of your data frame columns, get the quantile with:

>>> q\_hi = customer["CardSpendMonth"].quantile(0.995)

>>> q\_lo = customer["CardSpendMonth"].quantile(0.005)

and then filter with

>>> customer[(q\_lo < customer["CardSpendMonth"]) &

>>> (customer["CardSpendMonth"] < q\_hi)]

This removes the outer 1% of values whether they would count as outliers or not. In the filter expression the parentheses are mandatory. The expression with “&” between the square brackets is computing a boolean series that will be used to remove the rows that do not meet the criteria.

Another criteria for removing outliers is based on removing values less than or greater than the "whiskers" on a boxplot (1.5 x IQR or more below 1st quartile or above 3rd quartile)

>>> high\_q, low\_q = customer['CardSpendMonth'].quantile([.75, .25])

>>> irq = high\_q - low\_q

>>> median = customer['CardSpendMonth'].median()

>>> customer[(customer["CardSpendMonth"] < median + (1.5 \* irq)) &

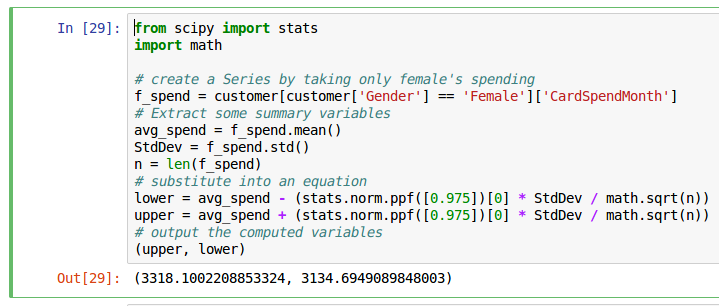
>>> (customer["CardSpendMonth"] > median - (1.5 \* irq))]

In this case we are again creating within the square brackets a boolean Series that is then used to remove the rows where the expression is False.

scipy.stats has methods trim1() and trimboth() to cut the outliers out in a single row, according to the ranking and an introduced percentage of removed values.

**Putting it together**

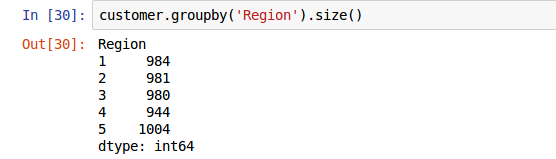
As we saw in the last module, we can use these functions for easy exploratory data analysis. Here we compute the average monthly card expenditure and the 95% confidence interval around that value. Thus, our level of 95% certainty about the true mean for female card expenditures is within the interval between $3,134.70 and $3,318.10 assuming that the original random variable is normally distributed, and the samples are independent.

Categorical Data

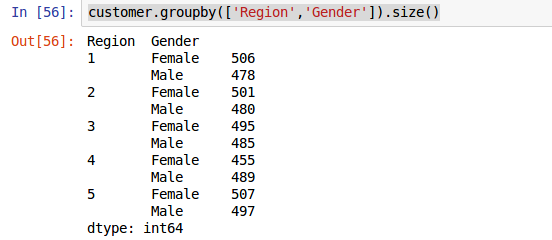
* Frequency tables
* Proportions tables
* Marginal tables
* Simplifying

**Frequency tables**

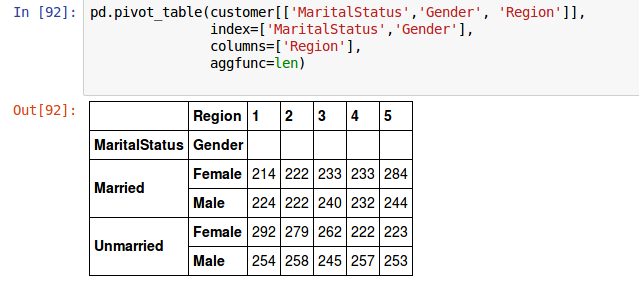
To produce contingency tables which calculate counts for each combination of categorical variables we can use panda’s groupby() function. For instance, we may want to get the total count of observations that fall into each region category.



If we want to understand the number of observations by region and gender we can produce a multi-level groupby by specifying a list of columns.



There is also the function [pandas.pivot\_table](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.pivot_table.html#pandas-pivot-table) that allows us to create three-plus dimensional contingency tables. In this case we assess the count of customers by marital status, gender, and region:

Proportions TablesWe can also produce contingency tables that present the proportions (percentages) of each category or

combination of categories. To do this we simply include in an arithmetic function the frequency tables produced by pivot\_table(). The output of pivot\_table and group\_by automatically applies a scalar arithmetic function to all elements. The following reproduces the previous tables but calculates the proportions rather than counts:

>>> # Percentage of clients across regions

>>> customer.groupby('Region').size() / len(customer)\*100

Region

1 20.110362

2 20.049050

3 20.028612

4 19.292867

5 20.519109

dtype: float64

We can add round() to the output of groupby and pivot\_table to round our values to a specified decimal:

# Percentage of clients across regions to 2 decimal places

>>> (customer.groupby('Region').size() / len(customer)\*100).round(2)

Region

1 20.11

2 20.05

3 20.03

4 19.29

5 20.52

dtype: float64

# percentages of clients across regions & gender to 2 decimal places

>>> (customer.groupby(['Region','Gender']).size() \* 100 / len(customer)).round(2)

Region Gender

1 Female 10.34

Male 9.77

2 Female 10.24

Male 9.81

3 Female 10.12

Male 9.91

4 Female 9.30

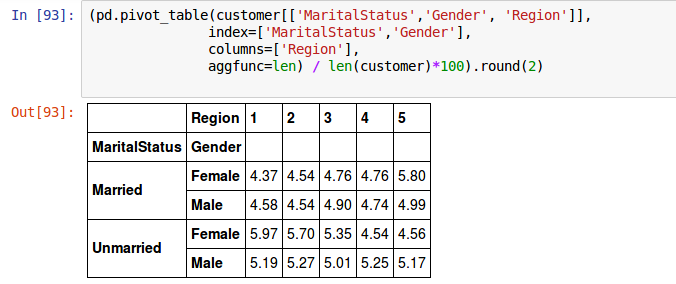
Male 9.99

5 Female 10.36

Male 10.16

dtype: float64

# percentages of clients across regions, gender & marital status



**Margins tables**

Margins show the total counts or percentages across columns or rows in a contingency table. For instance, if we go back to the cross classification counts for gender by region:

>>> print(pd.pivot\_table(customer[['Gender', 'Region']],

>>> index=['Gender'],

>>> columns=['Region'],

>>> aggfunc=len)

>>> )

Region 1 2 3 4 5

Gender

Female 506 501 495 455 507

Male 478 480 485 489 497

We can compute and add the column and row margins to the pivot\_table by adding the parameter “margins=True”.

>>> print(pd.pivot\_table(customer[['Gender', 'Region']],

>>> index=['Gender'],

>>> columns=['Region'],

>>> margins=True,

>>> aggfunc=len)

>>> )

Region 1 2 3 4 5 All

Gender

Female 506.0 501.0 495.0 455.0 507.0 2464.0

Male 478.0 480.0 485.0 489.0 497.0 2429.0

All 984.0 981.0 980.0 944.0 1004.0 4893.0

**Simplifying**

Since the output of a pivot\_table is a DataFrame we can apply follow-on functions easily. For example, we can find what region and job category combinations have the largest number of observations. Here we see that our customers tend to be in sales more than any other job category and that is consistent across all regions.

We first select out the columns of interest from the rest of the DataFrame and then call pivot\_table specifying the aggregation function “len” which takes a count of the number of elements in the region – job category bucket.

>>> customer[['JobCategory', 'Region']].pivot\_table(

>>> index=['JobCategory'],

>>> columns=['Region'],

>>> margins=True,

>>> aggfunc=len)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Region | 1 | 2 | 3 | 4 | 5 | All |
| JobCategory |  |  |  |  |  |  |
| Agriculture | 51.0 | 39.0 | 40.0 | 33.0 | 47.0 | 210.0 |
| Crafts | 95.0 | 84.0 | 79.0 | 97.0 | 91.0 | 446.0 |
| Labor | 145.0 | 133.0 | 136.0 | 123.0 | 134.0 | 671.0 |
| Professional | 264.0 | 286.0 | 277.0 | 271.0 | 259.0 | 1357.0 |
| Sales | 310.0 | 328.0 | 321.0 | 302.0 | 338.0 | 1599.0 |
| Service | 119.0 | 111.0 | 127.0 | 118.0 | 135.0 | 610.0 |
| All | 984.0 | 981.0 | 980.0 | 944.0 | 1004.0 | 4893.0 |

We start with an expression that creates a count of rows per region

>>> customer.groupby('Region').size()

Region

1 984

2 981

3 980

4 944

5 1004

dtype: int64

We can then create a new column called “Percent” that represents the proportion of each region. Since the result of applying “size” to the output of groupby is a Series we first create a DataFrame object that we can add the column to. We then define the Percent column by making use of pandas implicit loop over all rows when you apply a scalar mathematical operation to a Series.

>>> df = pd.DataFrame({'RegionCount':customer.groupby('Region').size()})

>>> df['Percent'] = (df['RegionCount'] \* 100 / len(customer)).round(2)

>>> print(df)

RegionCount Percent

Region

1 984 20.11

2 981 20.05

3 980 20.03

4 944 19.29

5 1004 20.52

>>> # percentages of clients across regions & gender

>>> df = pd.DataFrame({'RegionGenderCount':

>>> customer.groupby(['Region','Gender']).size()})

>>> df['Percent'] = (df['RegionGenderCount'] \* 100 /

>>> len(customer)).round(2)

>>> print(df)

Region Gender

1 Female 506 10.34

Male 478 9.77

2 Female 501 10.24

Male 480 9.81

3 Female 495 10.12

Male 485 9.91

4 Female 455 9.30

Male 489 9.99

5 Female 507 10.36

Male 497 10.16

The example above allows us to identify the fact that each gender by region represents about 10% of the data. Thus, we can be confident that we have equal dispersion of clients across the gender categories across all the regions served.

Now let’s do something a bit more complex that takes a number of steps in pandas. We will compute the percentage of clients for a particular region by gender.

We start by creating a data frame that contains the counts of region by gender like in the last example.

>>> df = pd.DataFrame(customer.groupby(['Region','Gender']).size())

>>> print(df)

0

Region Gender

1 Female 506

Male 478

2 Female 501

Male 480

3 Female 495

Male 485

4 Female 455

Male 489

5 Female 507

Male 497

So now we have counts of the number of customers by region and gender. The counts are in a column named 0 (that is an integer 0, not a string).

In the next line of code, we take the dataframe and give the column a name with the rename function then we apply the unstack function to get the males and females back into one row. This gives us a multi-index and the rows have hierarchical names which are printed out. You can see they are tuples consisting of the two levels of the index.

>>> df = df.rename\_axis({0:'NumCust'}, axis="columns").unstack()

>>> print(df)

>>> print(df.columns.values)

NumCust

Gender Female Male

Region

1 506 478

2 501 480

3 495 485

4 455 489

5 507 497

[('NumCust', 'Female') ('NumCust', 'Male')]

Now we know the names of the rows it is a simple arithmetic statement to calculate the sum for the row and then we use assign to create the row and give it a name and a value. We again use pandas implicit loop over rows to calculate all the values. First we calculate the row-wise sum and then in the next step we calculate the percentages because the per cent calculation relies on the new RegTot calculation..

>>> *# sum df[('NumCust', 'Female')] + df2[('NumCust', 'Male')] for all rows*

>>> df = df.assign(RegTot = (df[('NumCust', 'Female')] + df[('NumCust', 'Male')]))

>>> df = df.assign(pctMale = (df[('NumCust', 'Male')] \* 100 /

>>> df['RegTot']).round(2),

>>> pctFemale = (df[('NumCust', 'Female')] \* 100 /

>>> df['RegTot']).round(2)

>>> )

>>> print(df)

NumCust RegTot pctFemale pctMale

Gender Female Male

Region

1 506 478 984 51.42 48.58

2 501 480 981 51.07 48.93

3 495 485 980 50.51 49.49

4 455 489 944 48.20 51.80

5 507 497 1004 50.50 49.50

**Exercises**

1. What is the mean, median, standard deviation, and 85th percentile for HHIncome?

2. What is the 95% confidence interval for male and female mean HHIncome?

3. For the HHIncome variable identify how many observations are beyond 3 standard deviations from the mean?

4. How many observations are in each JobCategory?

5. What is the percentage of customers that are in the Sales JobCategory?

6. What are the percentages of customers in the Sales JobCategory across each region? Are these

proportions consistent across gender?

**Visualizing data**

* + Getting started with plotting in Python and Jupyter Notebooks
  + Advanced plotting

**Getting started with plotting in Python and Jupyter notebooks**

For quick data exploration, pandas plotting functions can provide an expeditious and straightforward approach to understanding your data. The matplotlib module can produce very nice plots with a little more work. This chapter should teach you the basics, and give you a good idea of what you want to do next.

To start, enter the following “magic” command in a cell in the Jupyter notebook.

>>> %matplotlib inline

When you execute the cell (shift-enter), this sets up the environment for plots to be output to your notebook. When the notebook is stored, the plots get stored with it.

It does not however import the plotting module matplotlib and make its functions available. To do this we execute the following import statement. Note that we are importing it as a synonym “plot”. This again is one of the few times when we use a synonym because it is an accepted name and thus will be familiar to people reading your code. Other Python modules are imported using their whole name with the exception of pandas (pd) and numpy (np).

>>> import matplotlib as plot

When you have executed the line above you are ready to start plotting.

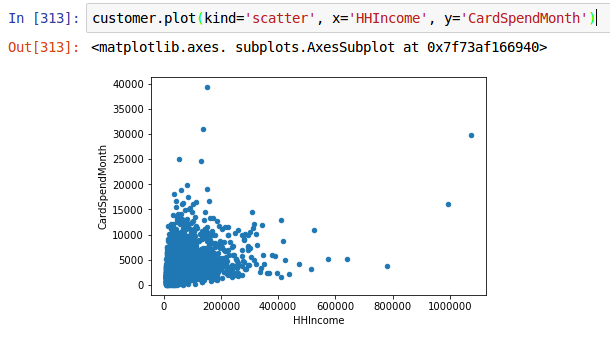
The following visualizations are covered:

* Scatter plot
* Line chart
* Bar chart
* Histogram
* Box plot
* Stem & leaf plot

**Scatter Plot**

To make a simple scatter plot use the plot() function of the data frame with a vector of x values and a vector of y values. Here we look at the relationship between household income and monthly card expenditures:

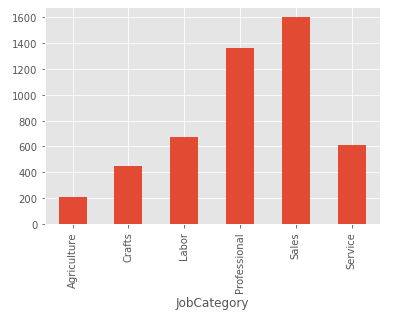
>>> customer.plot(kind='scatter', x='HHIncome', y='CardSpendMonth')



Bar Chart

To make a bar chart of values, you first need to compute the counts for a categorical variable and then pass that to plot(). For example, we can visualize the frequency of job categories.

>>> customer.groupby('JobCategory').size().plot.bar()

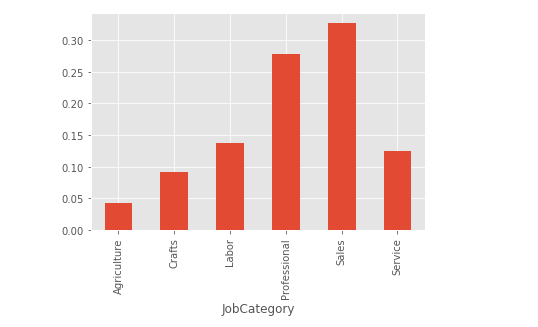


In this example we call ***plot.bar()*** rather than ***plot(kind=’bar’)***. Either will produce the same plot. All the calls to plot have a function within the ***plot*** namespace calling the same function as ***plot(kind=***. Since there is only one index that is used as x by default and there is only one column so that is used for the vertical axis by default.

Similarly, we can convert this to a proportions chart by dividing each value by the count for all the categories. Again we use the implicit loop created when dividing a panda.Series by a scalar to divide all the categories with one statement.

>>> TotCount = customer['JobCategory'].count()

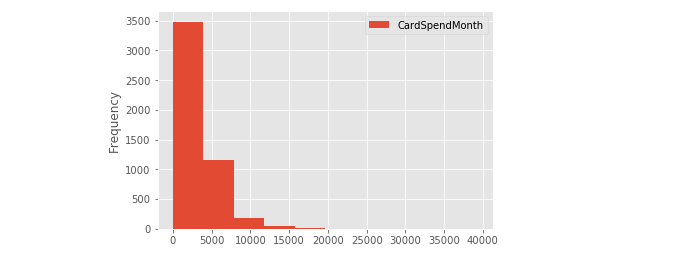
>>> (customer.groupby('JobCategory').size()/TotCount).plot.bar()



**Histogram**

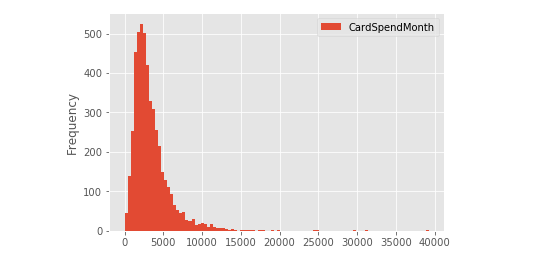
To make a histogram, select the desired column of the ***DataFrame*** and call ***plot.hist().*** Pass it the name of the column you want summed into bins as the “***by***” parameter. By default the data is summed into ten bins.

**>>>** customer[['CardSpendMonth']].plot.hist(by='CardSpendMonth')



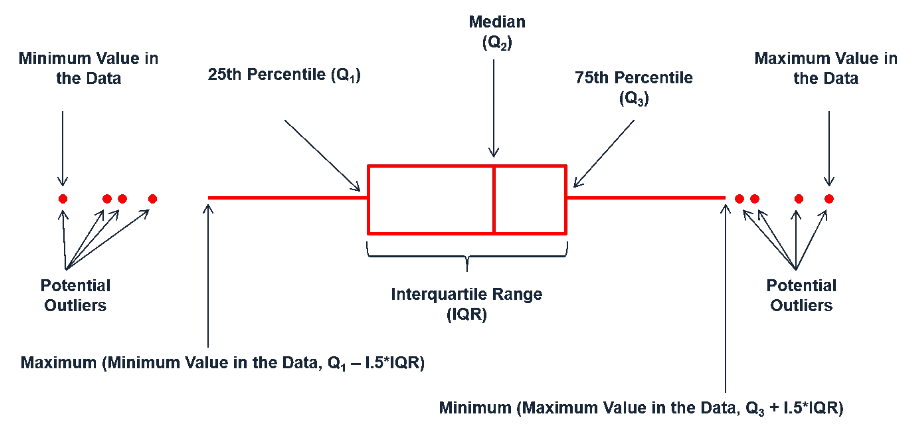
You can also use the “***bins***” argument to determine the number of the bins.

>>> customer[['CardSpendMonth']].plot.hist(by='CardSpendMonth', bins=100)



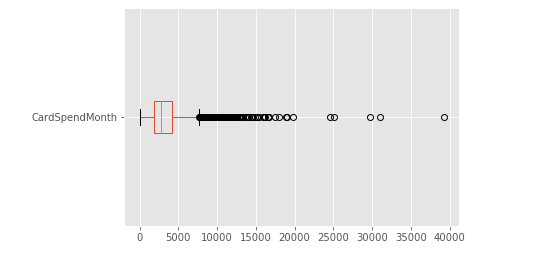
**Box Plot**

Box plots are an alternative way to illustrate the distribution of a variable and is a concise way to illustrate the standard quantiles, shape, and outliers of data. As the generic diagram indicates, the box itself extends, left to right, from the 1st quartile to the 3rd quartile. This means that it contains the middle half of the data. The line inside the box is positioned at the median. The lines (whiskers) coming out either side of the box extend to 1.5 interquartile ranges (IQRs) from the quartiles. These generally include most of the data outside the box. More distant values, called outliers, are denoted separately by individual points.

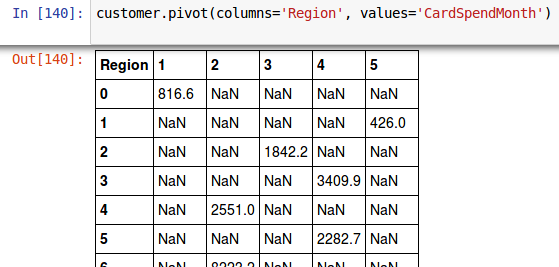
Box Plot Description

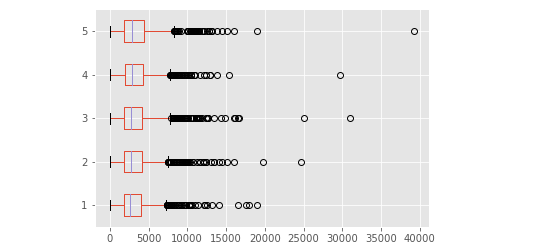
To make a box-whisker plot (aka box plot), use ***plot.box().*** By default the whiskers are vertical. To make it stretch across the width of the page pass the parameter and value ***vert=False***:

>>> customer[['CardSpendMonth']].plot.box(vert=False)



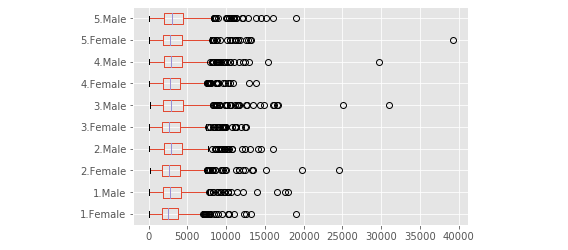
To get a box plot that displays the distribution of monthly expenditure values across the different regions we create a pivot table to move the different regions into columns.

We can now create the box plot from the columnized data:>>> customer.pivot(columns='Region', values='CardSpendMonth').plot.box(vert=False)



We can also assess interactions. In this case we look at the distribution of monthly expenditure values across the different regions and genders.

To do this in pandas we need to create an index that will be our axis. This is done by creating a ***Series*** that consists of a string combining the region and gender columns for each row. We then use a call to ***groupby()*** this index and call ***cumcount()*** giving us the count of the occurrence of the index value for each of the new index values. We then use this new series as the index in pivoting the data, which breaks it into separate columns for each unique element in the index.



**Stem-and-leaf plots**

Pandas, unlike R, does not have a function to make a stem-and-leaf plots simply. A stem-and-leaf plot is primarily useful for smaller data sets since details can get lost in the compression that needs to be done to fit a large number of observations onto a line. The stem-and-leaf diagram is only useful on integer columns.

If you need to do a stem-and-leaf diagram in python you can install an optional package “stemgraphic” that can be downloaded from the Python Package Index (pypi.python.com). To install the package, you would issue the following command from a command line prompt:

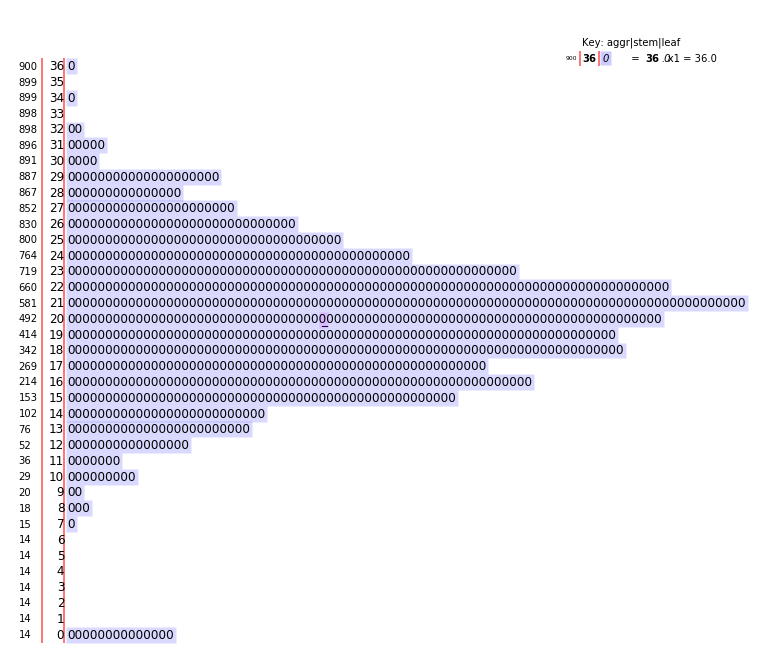
pip install stemgraphic

Then in your Jupyter Notebook issue the following command:

>>> import stemgraphic

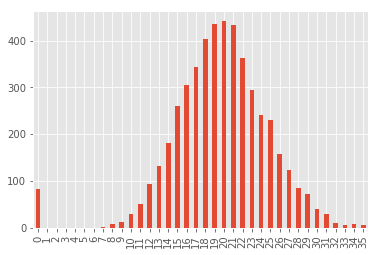
>>> stemgraphic.stem\_graphic(customer['TVWatchingHours'])

And this is the result. Note that there are a number of people who watch no television and then a mode (most common value) around 20 hours. Because of the large number of observations, the “y” scale has been compressed.



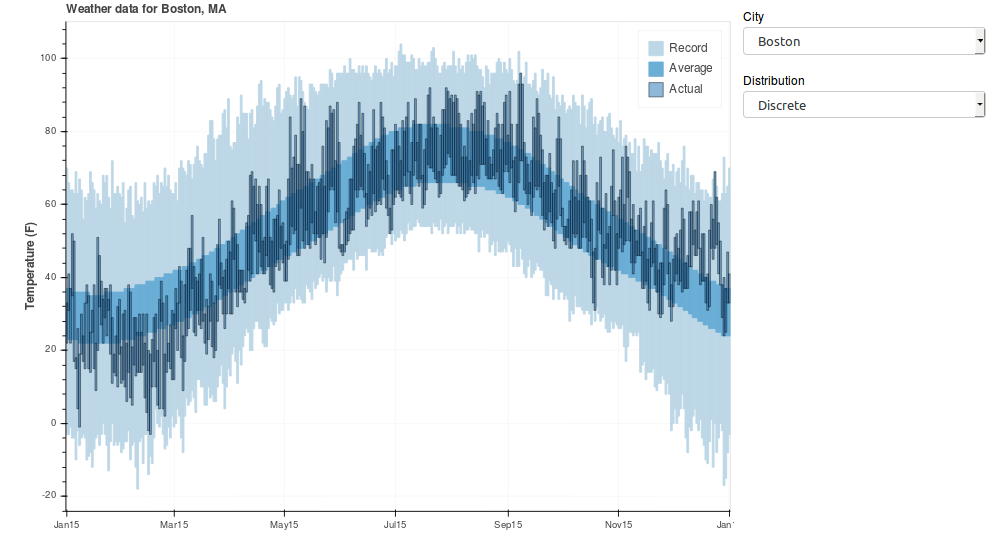
Most of the data from the stem-and-leaf diagram can be visualized with a bar plot which is very easy to specify in pandas.

>>> customer.groupby('TVWatchingHours').size().plot.bar()



**Advanced Plotting with the *bokeh* module**

Being able to create visualizations (graphical representations) of data is a key step in being able to communicate information and findings to others. In this chapter you will learn to use the ***bokeh*** module to declaratively make beautiful plots or charts of your data. Although ***pandas*** (through ***maplotlib***) does provide built-in plotting functions, the ***bokeh*** module implements the Grammar of Graphics. This makes it particularly effective for describing how visualizations should represent data. Learning this library will allow you to make nearly any kind of (static) data visualization, customized to your exact specifications such as [this re-creation](https://demo.bokehplots.com/apps/weather) of a classic visualization provided in Edward Tufte’s book Visual Display of Quantitative Information.



This section will provide a general introduction to the ***bokeh*** syntax to include:

* Grammar of graphics: Grammar of graphics gives us a way to talk about parts of a plot
* The basics: Understanding the basics of the ***bokeh*** grammar
* Glyphs: Mapping variables to visualization characteristics
* Glyph properties: Plotting data with geometric shapes

**Grammar of graphics**

Just as the grammar of language helps us construct meaningful sentences out of words, the Grammar of Graphics helps us to construct graphical figures out of different visual elements. This grammar gives us a way to talk about parts of a plot: all the circles, lines, arrows, and words that are combined into a diagram for visualizing data. Originally developed by Leland Wilkinson, the Grammar of Graphics was adapted by Hadley Wickham to describe the components of a plot, including

* the data being plotted
* the geometric objects (circles, lines, etc.) that appear on the plot
* a set of mappings from variables in the data to the aesthetics (appearance) of the geometric objects
* a statistical transformation used to calculate the data values used in the plot
* a position adjustment for locating each geometric object on the plot
* a scale (e.g., range of values) for each aesthetic mapping used
* a coordinate system used to organize the geometric objects
* the facets or groups of data shown in different plots

Wickham further organizes these components into layers, where each layer has a single geometric object, statistical transformation, and position adjustment. Following this grammar, you can think of each plot as a set of layers of images, where each image’s appearance is based on some aspect of the data set. Altogether, this grammar enables us to discuss what plots look like using a standard set of vocabulary. And similar to how ***pandas*** provide efficient data transformation and manipulation, ***bokeh*** provides more efficient ways to create specific visual images.

**The Basics**

To create a plot:

1. Import the figure and show function() from the bokeh.plotting module.
2. Call the figure() function. This creates a blank canvas.
3. Specify how you want to map variables to visual aspects. In this case we are simply mapping the HHIncome and CardSpendMonth variables to the x- and y-axes.
4. You then add new layers that are the geometric objects which will show up on the plot. In this case we add circle‘s to add a layer with points elements as the geometric shapes to represent the data.
5. After all the specification is done you call show(), passing the figure object.

Import the functions that will be called. This only needs to be done once in the notebook.

>>> from bokeh.plotting import figure, show, output\_notebook

Tell ***bokeh*** to send the output to the Jupyter Notebook. This only needs to be done once in the notebook.

>>> output\_notebook()

After output\_notebook is run a message is written out saying “BokehJS successfully loaded.”

In all the plots below, it is assumed that the two lines above have been executed. If you get an error saying ”NameError: name 'figure' is not defined“ then you need to re-execute the import from ***bokeh.plotting*** and probably execute output\_notebook as well. There is no problem executing these multiple times in the notebook, it is just unnecessary.

Create the canvas. This is done for every graph or figure you create.

>>> plt = figure()

If necessary, transform the data to create the data to be plotted. In this example this is not needed.

Define the markers (***bokeh*** calls them ***glyphs***) to add to the canvas that was created for this plot.

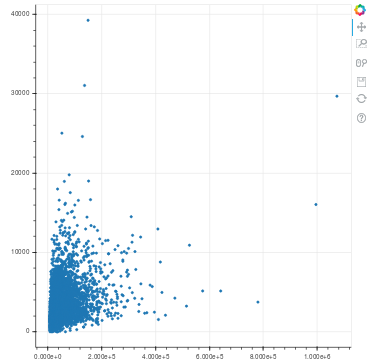
>>> plt.circle(

>>> x = customer['HHIncome'],

>>> y = customer['CardSpendMonth'])

Tell the system we are finished and to show the plot.

>>> show(plt)



If you do this in your Jupyter Notebook you will find that the plot has a limited interaction capability. Using the icons on the right-hand side you can zoom in or out and pan using the mouse. You can also save your plot to a file by clicking on the floppy disk icon.

**Aesthetic mappings**

Aesthetic mappings in the Grammar of Graphics take properties of the data and use them to influence visual characteristics, such as position, color, size, shape, or transparency. Each visual characteristic can thus encode an aspect of the data and be used to convey information.

For example, we can add a mapping to a color characteristic to identify retired vs. non-retired customers:

def retiredcolor(row):

if row['Retired'] == 'No':

return 'green'

elif row['Retired'] == 'Yes':

return 'red'

else:

return 'gray'

plt = figure()

plt.circle(

x = customer['HHIncome'],

y = customer['CardSpendMonth'],

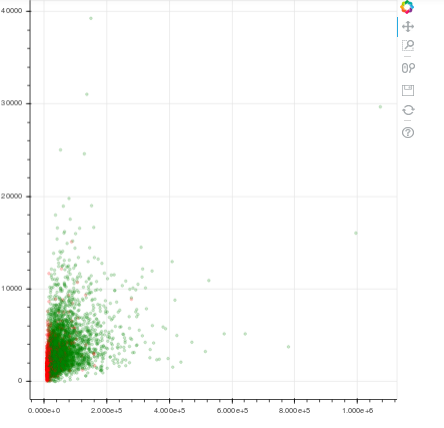
fill\_alpha = 0.2, line\_alpha = 0.2,

color = customer.apply(retiredcolor, axis=1))

show(plt)

Here a function has been defined that takes the value in the ‘Retired’ column and maps it to a color. If we were to execute “customer['Retired'].unique()” to tell us what values are possible for this column we would get “array(['No', 'Yes'], dtype=object)” telling us the possible values for ‘Retired’ are ‘No’ and ‘Yes’. So, why the final else: clause? If the analysis is run in the future on a new data set and that data set is not as clean (e.g. someone uses “yes” instead of “Yes”) we are assured that not only will the code run without error it will not produce a misleading graph.

To get a ***Series*** of color names that we can pass in to specify the color of the circles we call the ***apply*** function and pass it the name of our function that maps the input value to a color. Because ***apply*** will apply the function to all the columns by default, and we just want the function to apply to a column, we have to specify “axis=1”. (To help remember which axis 0 is and which 1 is you can think that the numeral 1 looks like a column.) In this example the color has also been made transparent by specifying “fill\_alpha = 0.2, line\_alpha = 0.2,” so that overlapping circles will show as a darker color.



**Specifying geometric shapes**

Building on these basics, ***bokeh*** can be used to build almost any kind of plot you may want. These plots are declared using functions that follow from the Grammar of Graphics.

The most obvious distinction between plots is what geometric objects they include. ***Bokeh*** supports a number of different types of glyphs, including:

* annulus, arc, asterisk, triangle, wedge and many more for drawing individual points (e.g., a scatter plot)
* line, ray, and multi\_line for drawing lines (e.g., for a line charts)
* bezier and quadratic for drawing smoothed lines (e.g., for simple trends or approximations)
* rect and hbar for drawing bars (e.g., for bar charts)
* patch and quad for drawing arbitrary shapes
* image and image\_url for placing images in the chart
* patches for drawing polygons in the shape of a map (You can access the data to use for maps of the U.S. by using the import “from bokeh.sampledata.us\_counties import data as counties”. For an example see [this map of Texas.](http://bokeh.pydata.org/en/latest/docs/gallery/texas.html)).

Each of these glyphs have similar parameter lists but drop or add additional parameters as needed for their particular purpose. For example, the ***text()*** glyph requires that you supply the text to be displayed. Almost all glyphs require an x and y mapping at the bare minimum.

You can create multiple plots to be output together by specifying the individual canvases in a call to gridplot.

In this example, code two calls to figure() create two graph canvases. The same data is plotted in both plots. Here we take only the rows where the JobCategory is ‘Agriculture’. The first has the default grid lines that are spaced equidistantly The second has specified that both the x and y axes are logarithmically spaced. Then the call to show() passes the output of gridplot(). The parameters to gridplot are a list of the canvases to be shown together and the number of columns to show the canvases in. Also specified is the width and height (in pixels) of the plots.

First, the code:

df = customer[customer['JobCategory'] == 'Agriculture']

# first plot, linear axes

plt1 = figure()

plt1.circle(

x = df['HHIncome'],

y = df['CardSpendMonth'])

# second plot, log-log axes

plt2 = figure(x\_axis\_type = 'log', y\_axis\_type = 'log')

plt2.circle(

x = df['HHIncome'],

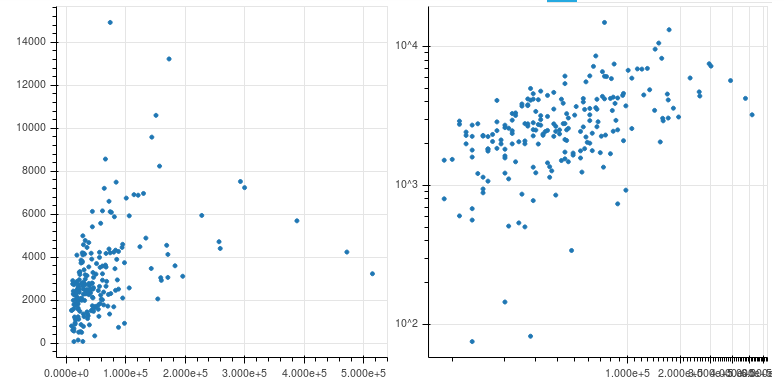
y = df['CardSpendMonth'])

# show them both together

show(gridplot([plt1, plt2], ncols=2,

plot\_width=380, plot\_height=380))

And here is the output:



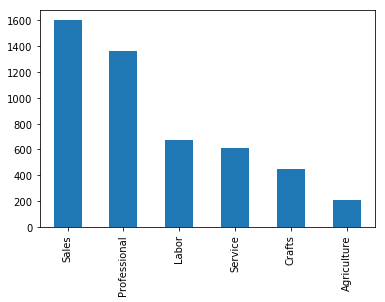
If you want counts or averages of categorical variables there is a powerful routine ***bar*** that can be used.

In this example we pull out the JobCategory column of the data and summarize it with ***value\_counts***().

This creates a new Series column that has row indices with the category name. Remember in pandas that rows can have names, just like columns can. We can then call the bar routine on this Series and we get a bar plot.

counts = customer['JobCategory'].value\_counts()

counts.plot.bar()



One of the powerful aspects to ***bokeh*** plotting is that you can easily add additional glyphs to a plot so that additional data can be shown together. We have already seen that with the plot above which showed retired customers with a different color circle than other customers.

**Statistical Transforms**

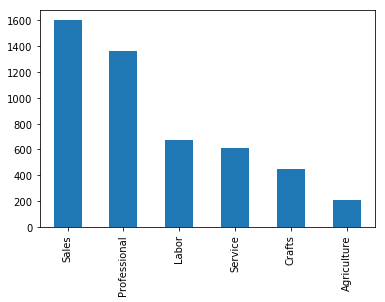
If you look at the below bar chart, you’ll notice that the the y axis was defined for us as the count of

elements that have the particular type. This count isn’t part of the data set (it’s not a column in our

customer data), but is instead a statistical transformation specified by the call to value\_counts which we then plot by calling DataFrame’s plot.bar function.

counts = customer['JobCategory'].value\_counts()

counts.plot.bar()



The DataFrame plotting functions are quick and easy plotting functions but if you want interactivity or for more control you can do a similar plot in ***bokeh***.

In the first statement we set the counts variable to a DataFrame created from the Series that is returned by value\_counts. Note that we start the value with an open parenthesis that is matched by the closing parenthesis on the third line. This is done so Python knows that this is a multi-line statement. If the code is not in a grouping operator (like parentheses or brackets) a Python statement ends at the end of the line. Once we have created the DataFrame we want to give some names to the index and column so that the plotting function will know which is which. The first call to rename\_axis passes a dictionary with the current name and the desired name and specifies that we want to rename a column. The second call to rename\_axis renames the index. In this case we don’t pass a dictionary, just a string, the new name of the index. Now the data is prepared we call figure like we have done previously, but this time passing an optional parameter. To make it fit nicely the plot height is set to 250 pixels. We then apply data to this graph by calling ***vbar***(). Passing to vbar the parameters x, top, and source creates the information needed for a histogram. The width parameter gives the proportion of space taken by the bars. Here the bars take up 90% of the space.

counts = (pd.DataFrame(customer['JobCategory'].value\_counts())

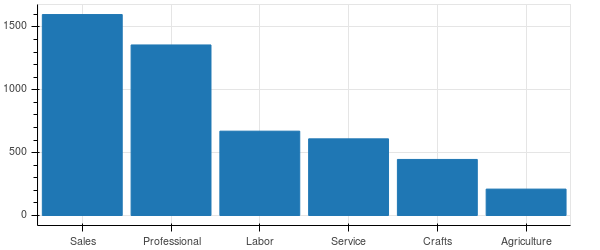
.rename\_axis({'JobCategory':'JobCounts'}, axis=1)

.rename\_axis('JobCategory'))

plt = figure(x\_range=list(counts.index), plot\_height=250)

plt.vbar(x='JobCategory', top='JobCounts', source=counts, width=0.9)

show(plt)



Say we want a scatter plot showing HHIncome vs Age, and we also want to overlay a line plotting the mean HHIncome at each Age group. This is an example of the general problem of plotting data plus a statistical value. So, the first thing we need to do is create the datasets that will be plotted. The scatter plot is just a selection of a couple of columns from the larger dataset. To create the dataset with the median value for each age, we call the ***pandas.DataFrame.pivot\_table()*** function and pass in numpy’s median() function as the method to be called to aggregate the multiple values for each age into a single value. This produces a ***pandas.Series*** object, a single column of values with an index that is the age associated with the value that is on that row of the series. Now that we have the data we can plot it, first creating a ***bokeh.plotting.figure*** to hold the graph and then calling scatter() to produce the scatter plot and line() to add the line on to of the scatter plot.

incomeVage = customer[['HHIncome','Age']]

medIncomeVage = incomeVage.pivot\_table(values='HHIncome',

index='Age', aggfunc=np.median)

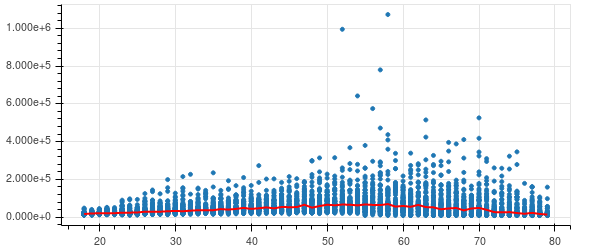
plt = figure(plot\_height=250)

plt.scatter(x=incomeVage['Age'], y=incomeVage['HHIncome'])

plt.line(x=list(medIncomeVage.index), y=medIncomeVage,

color='red', line\_width=2)

show(plt)



**Grouping into Multiple Subplots**

In bokeh you can specify multiple plots to be plotted together by using the gridplot() function to group multiple figures together.

For this example, we will be using the different JobCategories in the customer data. The ***unique***() function will give us a ***pandas.Series*** with these values. The list returned from ***unique***() will be in a unspecified order but the standard Python function ***sorted***() will put the job category names in alphabetical order.

for c in sorted(customer['JobCategory'].unique()):

print(c)

The output from the above is:

Agriculture

Crafts

Labor

Professional

Sales

Service

So, we want to produce a different ***figure*** for each of the 6 JobCategory values In Python this is the sort of thing that you write a subroutine for. But what would we write for a single JobCategory? We should start simply and build up. Something new in this chart is specifying logarithmic axes when we create the figure. By specifying

c='Agriculture'

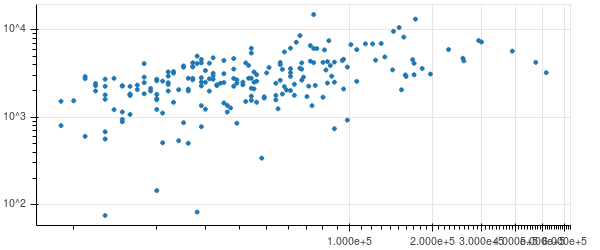
df = customer[customer['JobCategory'] == c]

plt = figure(x\_axis\_type = 'log', y\_axis\_type = 'log')

plt.circle(x = df['HHIncome'],

y = df['CardSpendMonth'])

show(plt)

Now we can write a routine that will produce this chart. We will pass in the name of the JobCategory in a parameter to the routine. In addition to making the creation of the figure (called plt in the code) a subroutine there are a couple of new elements here. One is that when selecting the data from the customer DataFrame we are specifying that the values for HHIncome and CardSpendMonth must be greater than zero. This is because we are plotting the values on a log-log graph and the log of zero is negative infinity which will cause an error when we plot another JobCategory which includes zeroes. The other thing that has been added is that in the call to figure we are specifying a title (so that we can tell one JobCategory graph from another) and we are specifying a plot height and width rather than going with the automatic sizing we would otherwise get.

def category\_plot(category):

df = customer[(customer['JobCategory'] == category) &

(customer['HHIncome'] > 0) &

(customer['CardSpendMonth'] > 0)]

plt = figure(x\_axis\_type = 'log', y\_axis\_type = 'log',

title=category, plot\_height=250, plot\_width=250)

plt.circle(x=df['HHIncome'], y=df['CardSpendMonth'])

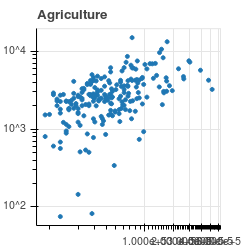
return plt

category='Agriculture'

plt = category\_plot(category)

show(plt)

we get out:



Now, the next step in our incremental development of multiple plots it to write the code that will create a chart for each JobCategory and display the charts in a nice layout. The code is straightforward. It calls the category\_plot that we created in the last step and creates a Python list holding the plots. It then calls gridplot() and a nice layout is produced. The layout is three cells over two since we are passing in a list with six graphs and we have specified that we want three columns of graphs.

from bokeh.layouts import gridplot

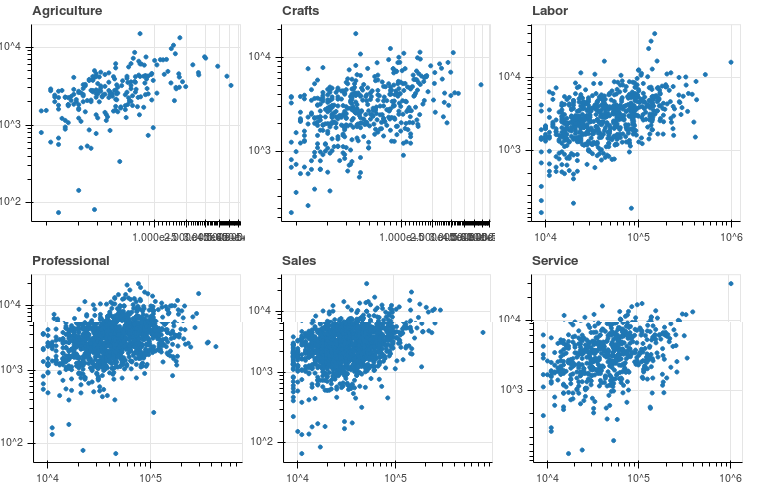
figures = []

for cat in sorted(customer['JobCategory'].unique()):

figures.append(category\_plot(cat))

show(gridplot(figures, ncols=3, toolbar\_location=None))

The result is shown below:



**Additional Resources on *bokeh***

<http://bokeh.pydata.org/en/latest/docs/user_guide.html> This is the best place to start. There are chapters on many topics that were not covered here and it goes into many more plotting options that can be covered here.

<http://bokeh.pydata.org/en/latest/docs/reference/plotting.html> This is the reference for the figure() call and the glyph methods.

<https://www.youtube.com/results?search_query=bokeh+python> There are a number of tutorials and tech talks available on youtube.

Doing a web search for ‘python bokeh blog’ turned up a number of well written articles and github code repositories that give examples of different plotting problems and their solutions.

<https://stackoverflow.com/questions/tagged/bokeh> StackOverflow has frequent questions about bokeh that get answered by experienced developers. The answers are then searchable for later site visitors to learn from.

**Other Visualization Libraries**

bokeh interactivity – This guide has not touched on using bokeh as a server for web based data visualizations. The official docs for this are at <http://bokeh.pydata.org/en/latest/docs/reference/server.html>.

<https://d3js.org/> D3 is a javascript library for producing interactive plots. When paired with a Python program doing the server-side work it can create compelling data displays. The javascript side in the user’s browser displays the data and responds to mouse clicks and the Python side on the server can load data and do analysis then pass the results back to the browser side.

**Exercises**

Use visualizations to answer the following questions:

1. Does the distribution of EducationYears appear normally distributed?
2. Is distribution of EducationYears across JobCategory fairly consistent?
3. Are there more “younger” customers than “older” customers?
4. Does there appear to be a relationship between Age and CardSpendMonth?
5. Do certain credit card holders appear to spend more per month than others? Does this differ across gender or region?

**Moving forward with Python and Data Science**

**Data selection**

Much of data science is learning how to apply statistical analysis to the data you have in preparation for building a model which can predict how data in a future or alternate dataset will behave. To test your model before more data is available it is common to partition your data into two sets. One set is used to create the model and the second set is used to test the model to see if it gives similar answers. How this is done in Python shows the flexibility of the language and its available modules. It also shows how you will proceed in doing more advanced data analysis in Python.

The first way to do a selection relies upon data structures built into the Python language, lists and sets, and on ***pandas*** data selection using a logic statement about which rows to include in a new ***DataFrame***.

train=customer.sample(frac=.5, random\_state=1)

print('Train', train.shape)

test = customer.select(lambda x: int(x) not in set(train.index))

print('Test:', test.shape)

Which outputs

Train (2446, 59)

Test: (2447, 59)

The code computes a training set of data by calling ***pandas.DataFrame***’s sample method. Here I have specified a start value for the random number generator. This is not usually necessary but is done here so that your results when running the code will match the text’s results. The fraction of rows that are taken to the training set of data is controlled by the frac parameter, here .5, or 50%. ***pandas.DataFrame***’s sampling routine does not have a way to specify that rows not selected for inclusion in the training sample are to be returned in a testing sample. To generate our test dataset, we use the Python “set” data structure. We create a set of the train.index and use the set membership operation to create a set of rows that are not in the training sample.

Another way to do the operation of generating training and testing sample sets is to use ***scikit-learn***’s [ShuffleSplit](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ShuffleSplit.html#sklearn-model-selection-shufflesplit)() function. This function returns a generator that can split a dataset multiple times – the n\_splits parameter. Each time the generator is advanced it returns the indices that are part of the training set and the indices that are part of the testing set. Usually a generator is advanced by looping over it with a “for” loop but in the code below we use the next() function since we only want one split. From the call is returned two arrays, the training set indices and the testing set indices. We then create the ***DataFrame***s by calling the take() method on the customer dataset. The take() function takes as a parameter a list of indexes to extract from the dataset and returns a new DataFrame.

from sklearn.model\_selection import ShuffleSplit

ss = ShuffleSplit(n\_splits=1, test\_size=0.5,

random\_state=1)

(train\_indexes, test\_indexes) = next(ss.split(customer))

train = customer.take(train\_indexes)

test = customer.take(test\_indexes)

print('Train', train.shape)

print('Test:', test.shape)

The output is the same as the previous example.

This same sort of ability to do things in multiple ways carries through to other aspects of doing statistics with Python. Simple descriptive statistics (e.g. mean, median, standard deviation) are available in the built-in ***statistics*** module that is available in all versions of Python since 3.4.

Loading the pandas module give you access to fast data structures (the ***pandas.Series*** and ***Pandas.DataFrame***). It also gives you access to [computations and descriptive statistics](http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-stats), like covariance, kurtosis, and pairwise covariance of columns in addition to the descriptive statistics in the statistics module.

For detailed statistics you will have to go to the [scikit-learn](http://scikit-learn.org/stable/) and the [statsmodules](http://www.statsmodels.org/stable/index.html) modules. The scikit-learn [tutorials](http://scikit-learn.org/stable/tutorial/index.html) have in-depth examples of statistical learning and sample code.

An interesting example of how there are multiple ways to compute statistics in Python is the blog post [four ways to conduct one way ANOVA using python](https://www.marsja.se/four-ways-to-conduct-one-way-anovas-using-python/).

**Additional resources**

A good list of resources available for learning data analysis with Python is on the web [here](http://www.datacommunitydc.org/blog/2013/07/python-for-data-analysis-the-landscape-of-tutorials). There are also some books on the topic, "[Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython](https://www.amazon.com/Python-Data-Analysis-Wes-McKinney/dp/1449319793)" Wes McKinney (Author) and "[Python Data Science Handbook: Essential Tools for Working with Data](https://www.amazon.com/Python-Data-Science-Handbook-Essential/dp/149191205)" Jake VanderPlas (Author).

**Final advice**

As you work on learning to program you will get error messages and your code will not give you the answers you are looking for. The source of the problem is often because you are passing the wrong type of object to a subroutine. Use the documentation to learn what is expected. Occasionally the reference documentation will be unclear as to what is expected. This is often because the full explanation is in the User Guide or the Tutorial. It is usually helpful to work through the User Guide material for the package you are using. It will acquaint you with the language that the writer is using to describe their program which will help when you turn to the reference pages.

To help debug your program sprinkle in lots of print statements so that you can see what data is being generated. In working with pandas, it is common to have a problem because you got back a Series when you were expecting a DataFrame (or vice-versa). When you try to call a function that is on one of these types on the wrong type you will get a run time error. Or you will call a function and pass in the wrong type of data. Don’t despair. Read the fine manual (RTFM). Look for answers on [StackOverflow](https://stackoverflow.com/). If time allows, take a break and come back with a refreshed view.