Answering Questions with Internet Data: Computational Tools for Social Studies Analysis

Richard Catrambone and Mark Guzdial
School of Psychology and School of Interactive Computing
Georgia Institute of Technology

July 18, 2012

Copyright held by Richard Catrambone and Mark Guzdial, 2012.

Contents

Contents **ii**

List of Figures **iv**

1 Answering Questions: What is Science? **3**
1.1 What is Science? 3
1.2 Methods of Science 6
1.3 The Experimental Method 8

2 Getting Data from the Internet for Computational Analysis **19**
2.1 Starting with Python and SciPy 19
2.2 Where can I find data? 21
2.3 Reading CSV Files 24

3 Plotting **35**
3.1 Your Basic Plot: Slicing Up The World's Population 35
3.2 Options on the Plot 39
3.3 The Plot Thickens: Combining Plots to Determine US and UK Growth Rates 41

4 Descriptive Statistics **47**
4.1 Average or mean: Petroleum Tax Prices 47
4.2 Understanding Measures of Variability 49
4.3 Computing Standard Deviation 50
4.4 Viewing Histogram 52

5 Correlation **55**
5.1 Correlation: A Measure of Association 55
5.2 Computing correlation: Is it the company, or war in the Middle East? 56
5.3 But do we believe it? 62

6 Text Analysis **67**
6.1 Visualizing textual differences: Bacon v. Shakespeare 67
6.2 Counting Text Patterns 76

7 Inferential Statistics and Hypothesis Testing 81
7.1 Inferential Statistics 81
7.2 Hypothesis Testing 90
7.3 Computing a Context: Elections and Unemployment Rates . 94
7.4 Computing a t test 95
7.5 ANOVA: Analysis of Variance 100
7.6 Computing ANOVA: Analysis of Variance 102
7.7 Calculation of Significance of a Correlation 107

8 Multiple Linear Regression and Advanced Experimental Designs 109
8.1 The Multiple Regression Equation 110
8.2 Computing a multiple regression 114
8.3 Final Note on Alternative Experimental Designs 116

A A Brief Introduction to Key Parts of Python 119
A.1 Variables and Assignment 119
A.2 Lists 120
A.3 Dictionaries 121
A.4 Blocks 121
A.5 Functions 122
A.6 FOR loops 122
A.7 Conditionals 124

B Reading from Live Data 125

C Program Listings 131
C.1 CVSfile 131
C.2 fancierplot.py – a run-able plot 132
C.3 US-UK Population Plot for years 1999–2000 132
C.4 Exploring British and American Petroleum Company Stock Prices 134
C.5 Text Analysis: Shakespeare or Bacon? 138
C.6 Hypothesis Testing: Does the unemployment rate make the President? 139

Bibliography 143

Index 145

List of Figures

2.1 IDLE running on Windows 20
2.2 IDLE running on MacOS X 21
2.3 Running the first plot 22
2.4 Visualization of price of wine in England from 1259–1400 . . . 23
2.5 Grabbing the Crown Revenue as text 24
2.6 Viewing the Crown Revenue as text 25
2.7 Plot of Crown Revenue in England by Year 26
2.8 Front page of *The Guardian*'s data journalism page 26
2.9 Excel data from *The Guardian* on perception of crime in the UK 27
2.10 Example CSV data from World Economics Dataset 27
2.11 Text CSV file from UK Perception of Crime data 33
2.12 Plot of perception of overall crime by income level 34

3.1 Our first plot 36
3.2 Our first graph–countries' populations, unsorted 37
3.3 Sorted countries' populations 38
3.4 A graph generated with X and Y values 40
3.5 Making a fancier plot 42
3.6 US and UK Populations, as two subplots 46

4.1 BP and Exxon-Mobil stock prices in 1990, as histograms 54

5.1 Example scatterplot 56
5.2 Data for scatterplot 57
5.3 Uniform distribution 64
5.4 Normal distribution 65

6.1 Francis Bacon's essays with white background, 'the' highlighted 69
6.2 Francis Bacon's essays with black background, 'the' highlighted 71
6.3 Comparing 'the' patterns in Bacon's *Essays* and Shakespeare's *Macbeth* 72
6.4 Visualization of all capitalized letters in the *Essays of Francis Bacon* 76

7.1 Probability of each outcome for flipping 10 coins (computed by Wolfram-Alpha) . . . 83
7.2 Probabilities for flipping 10 coins (computed by Wolfram-Alpha) 83
7.3 Graph of 1000 trials of flipping 10 coins . . . 85
7.4 t table from Wikipedia . . . 93

8.1 Scatterplot of high school and college GPA's . . . 109
8.2 Scatterplot of SAT score and GPA . . . 110

Preface

The Internet makes enormous amounts of data available to everyone, worldwide. What can you *do* with it all? How do you figure it out, come to understand those data, answer questions about it? The short answer is: Not by hand. There are too much data, and it's complicated.

The longer answer is: This book. We developed this book as the course notes for a class *Computational Freakonomics* which we first taught together at the Georgia Tech Study Abroad program at Oxford University in the Summer 2006.The book *Freakonomics: A rogue economist explores the hidden side of everything*[Levitt and Dubner, 2005] (`http://www.freakonomics.com/`) by Steven D. Levitt and Stephen J. Dubner is a NY Times Bestseller that uses economic methods for studying social questions. In the six weeks of this class, we:

- Read and discussed each of the six chapters
- Learned social science methods used in each chapter (led by psychologist Richard Catrambone)
- Learned computer science tools for implementing those methods on data downloaded from the Internet (led by computer scientist Mark Guzdial)

In particular, these course notes provide the details on the computational side of the course so that students have examples of syntax and semantics to work from. All students who take this course are expected to have had some introductory programming.

Why *Computational* Freakonomics? Because it's the computational side that makes it interesting. Using computation, we can access real and authentic data–like the data used in Freakonomics. Using computation, we can work with many more data points than we could manipulate by hand–just like in Freakonomics.

We use computation in two ways in this book:

- The obvious way is to process data, to perform mathematical processes mechanically so that we can analyze large, Internet-based datasets more easily than we could by hand.

- The less obvious way is to provide insight into where the statistical methods are coming from. There are typically *binomial distribution* functions built into any statistical software package. But in this book, we create a function to flip 10 coins, then another to call the flip function 1000 times, and then graph the resulting histogram. We create a normal distribution concretely, using small pieces that we already understand. In this way, we use computation to help us understand the statistical and mathematical concepts.

Other Ways of Using This Book

For teachers not lucky enough to teach a course named "Computational Freakonomics," there are other ways of using this book.

The book can make for an interesting supplement to an introductory computing course. Rich and interesting application domains can improve student motivation and retention[Guzdial, 2010]. The activities and content in this book provide (a) a rich set of examples for applying introductory Python computing knowledge and (b) an interesting context for motivating learning.

There is considerable interest in computational science. This book creates a bridge between introductory computing and use of computation in science. Only using introductory computing knowledge, this book explores how to explore datasets to answer questions empirically.

1 Answering Questions: What is Science?

Humans always have questions. Why is the sky blue? Where did life come from? What makes species different? Is this candidate better than that candidate? Why did that company go under? It's part of what makes humans special and different from other primates. We construct stories to explain our world[Mateas and Sengers, 2003]. Those stories are our *claims* about the world.

Science was developed as a way to answer questions. More specifically, science is a way to develop some assurance of the quality of the claims, the answers to those questions. How do you know if you got it *right*?

1.1 What is Science?

A key aspect to science is doubt. Given an answer to a question, the first reaction of a scientist is, "Is that right?"

One way in which we doubt is to question the common wisdom. Common wisdom might hold that the world is flat, or that *all* our behavior is learned (e.g., if that's true, why does a newborn have behavior?). Scientists seek different models of the world that might explore more or explain better than the common wisdom.

Scientists also doubt themselves and each other. In science we use doubt to question our research and ask whether factors other than the ones that we originally considered might have influenced our results. By doing this we come to see that science is a combination of interaction with the world and logic.

Science is more than just watching. Answers don't just fall out of observations. It is rare that data actually speak for themselves.

Science as a way of knowing

Science is not the only way to look at the world or, specifically, human behavior. Humans have always had questions about themselves. Art, literature, religion are all potentially fruitful ways through which we can gain new ideas about human behavior and experience. However, in contrast to

these other ways of knowing, science offers not only a great source of new ideas but also a powerful method for evaluating the ideas we have about reality.

That is to say, science helps us to know if our ideas about the world are wrong. Scientific methods cannot prove an idea is right, just that, so far, it is not wrong.

Alternative ways of making claims

Science isn't the only way that humans make claims about the world and answer questions. Natural science was used in Persia around the year 1000, and Galileo was a prominent scientist in the early 1600's. Humans existed, answering questions with claims, for thousands of years before science.

Tenacity

Tenacity refers to the acceptance of a belief based on the idea that "we have always known it to be this way." People at various times in human history have said "Women make bad soldiers" or "you can't teach an old dog new tricks." These statements are presented over and over again and accepted as true, but they are rarely examined and evaluated.

This is an all-too-often method of propagating claims, stories about how the world works. For example, television advertising and political campaigns use this technique when they present a single phrase or slogan repeatedly. Put it to a tune in a jingle, and people will walk around repeating it for you.

Authority

A second way we might accept a new idea is when an authority figure tells us it is so. Acceptance based on authority is simple because we only have to repeat and live by what we are told. It's easy.

Authority is useful in situations where we need a quick answer and don't have the time or opportunity to develop an answer another way. Television news shows will often bring on an expert authority to provide insight on a situation. Referring to an authority, especially in areas about which we know nothing, can be useful and beneficial.

Although authority brings with it a stability that allows for consistency, it has drawbacks. The major problem of accepting authority as having sole access to truth is that authority can be incorrect and thus send people in the wrong directions. For example, as long as everyone accepted the view that the earth was the center of the universe, no one though to study the orbit of the earth.

Reason

Reason and logic are the basic methods of philosophy. Reason is "thinking things through," connecting ideas, and making inferences based on the available information.

Reason often takes the form of a logical syllogism.

> Men can't count. Fred is a man. Therefore, Fred can't count.

We all use reason everyday as we try to solve problems and understand relationships. As useful as it is to be reasonable, however, reason alone will not always produce the appropriate answer. Why? One potential problem in the reasoned approach is that our original assumptions must be correct. You might make reasonable inferences based on available data – but then discover that the original data were flawed.

Common Sense

Common sense offers an improvement over acceptance based on tenacity, authority, or reason because it appeals to direct experience. Common sense is based on our own past experiences and our perceptions of the world. We can confirm and refute a claim based on what we know or can experience.

Common sense can be flawed. Our experiences and perceptions of the world might be quite limited. Just as there are optical illusions, there are cognitive illusions that lead us to be certain but wrong in our answers. The world is not flat, despite the common sense appeal of the claim, supported by thousands over centuries.

Furthermore, research in social psychology has shown that we make different psychological attributions depending on whether we observe or participate in a given situation. If we are asked to explain why someone made a bad grade, we tend to make internal attributions such as "She's not a good student" or "He is not smart." However, if we received a bad grade on a test, we would tend to make external attributions such as "I had three tests that day" or "The test was unfair."

Whereas common sense might help us deal with the routine aspects of daily life, it might also form a wall and prevent us from understanding new areas. This can be a problem, particularly when we enter areas outside our everyday experience.

- For example, people considered Albert Einstein's suggestions that time was relative and could be different for different people to be contrary to common sense.

- Likewise, it was considered contrary to common sense when Freud suggested that we did not always know or our motivations or when Skinner suggested that the concept of free will was not applicable to the behavior of most individuals.

- You might assume that a stable process, such as a regular heartbeat, is the healthier one. However, research using nonlinear analysis (chaos theory) has suggested, for example, that the patterns of a healthy heart are erratic and those of a pathological heart can be regular.

1.2 Methods of Science

In science a claim or an idea is evaluated or corrected through

1. Dispassionately observing by means of our bodily senses, and
2. Using reason to compare various theoretical conceptualizations based on experience.

The first method is a direct extension of the common sense approach we just talked about. Unlike a given person's common sense however, science is open to anyone's direct experience. Presumably, any person with normal sensory capacities could verify any observation made by a scientist.

The second method is a direct application of the principles of logic. In this case however, logic is combined with experience to rule out any assumptions that do not accurately reflect the scientific experiment. This blend of direct sensory experience and reason gives science a self-corrective nature that is not found in other ways of accepting ideas about the world.

One important technique is replication in which a procedure is repeated under similar conditions. For example, if an experiment is found to give similar results in different labs and even in different parts of the world, this lends support to the conclusions.

Scientific conclusions are never taken as final but are always open to reinterpretation as new evidence becomes available. If an experiment can't be replicated somewhere, then that's an important observation to add to our understanding. New ways of measurement might lead us to replicate experiments, but observe them with our new measurement device to go beyond the capabilities of our raw senses.

In the rest of this section, we do a quick overview of key methods of science.

Naturalistic Observation

If little is known about a particular phenomenon, it is often useful simply to watch the phenomenon occur naturally and get a general idea of what is involved in the process. Initially this is accomplished by observing and describing what occurs.

This scientific technique is called *naturalistic observation*. A classic example of this approach is Charles Darwin's observation of animals in the Galapagos Islands which formed the basic of his theory of evolution. Naturalistic observation occurs in the setting of the phenomenon because

the phenomenon might behave differently in a laboratory or in another location. An animal might be moved from its original home to a zoo, but behave differently because some key aspect of the original ecology wasn't present in the zoo.

Naturalistic observation is often systematic. A scientist might observe a behavior that is rare and unusual, or only occurs at certain times a day. A single observation can constitute an *existence proof* – something occurs or exists that might not have been known previously. But to *describe* some phenomenon well, it should be observed regularly, or at least, more than once.

A key aspect to naturalistic observation is that the observer attempts *not* to disturb the phenomenon. If the scientist changes the phenomenon in some way, it's no longer naturalistic. If the scientist *tries* to change the phenomenon in some way, the method is more *experimental*.

Correlational Approach

At other times we might want to understand certain aspects of a complex system with the goal of better describing how one aspect of the system might be associated with another aspect. For example, we might want to know whether people who have friends have fewer health-related problems than people who do not or whether eating certain foods is associated with not having cancer.

How would you go about answering such questions? One way is to examine and note the relationship between a person's health and the number of friends that person has. But how are you to understand these data? How do you interpret the results?

Consider an important historical example of a similar question. For years, people wondered about the relationship between smoking tobacco and having lung cancer. Tobacco companies insisted that there was no "smoking gun," i.e., no conclusive evidence that smoking tobacco led to lung cancer.

The first step is to ask whether the two events go together. In this example, researchers sought to determine whether, when one event occurred (a person smoked tobacco), the other event also occurred (the person had cancer). Such a scientific approach is called by various names, including a *correlational approach*.

Just finding that a relationship exists between two events does not allow us to determine exactly what that relationship is, much less to determine that one event actually *caused* the other event to happen. That is, perhaps some other variable controls the other two variables. For example, being a member of some particular cultural group might lead you to drink too much beer, smoke too much, and eat too much fat, and it is the *combination* of those things that leads to cancer.

Experimental Approach

As our knowledge about an area grows, we might get to the point of formulating specific predictions. A prediction is a strong form of a claim – that you understand something so well, that knowing the current state of the world, you can predict what will happen next. In this case, our questions are structured in the form "If I do this, then I expect this other thing will happen." For example, we might predict that more people are likely to help a stranger if they perceive the environment to be safe than if they think it is dangerous. This approach in which we interact directly with the phenomenon we are studying is the *experimental method*.

Let's consider the relationship between the scientist and the research participant in each method.

- With the naturalistic method, the scientist is passive and observes carefully the phenomenon of interest. In the case of psychology or other science of human beings, the phenomenon might be the activity of the research participant. In this method the scientist does not try to change the environment of the research participant. The research participant simply goes about normal activity and the scientist watches, preferably without influencing the participant's behavior. In this way, the scientist can make a detailed description of some aspect of the research participant's natural behavior.

- In contrast, when using the experimental method, the scientist is more active and the research participant's activities are restricted. The scientist intentionally structures the situation so that he or she can study the effect of a particular factor on the research participant's behavior.

- In-between these two approaches are correlational methods which might range from simple observation and correlation of factors to a more active manipulation of one of the factors although without the same degree of manipulation of control that is typical of the experimental method.

Consider many of the “”scientific” claims that you hear on television or read in magazines and look for *alternative explanations* to the claims being made. Why would eating those foods result in weight loss? Could it be that that car's phenomenal gas mileage only occurs under certain conditions? Thinking scientifically is not something you do only when you design experiments; rather, it is a way of approaching all information.

1.3 The Experimental Method

To give you a more accurate understanding of how scientists learn from interacting with the environment, let's consider the following line of fictitious research:

Assume that the makers of a brand of children's cereal, *Roasty-Toasties*, claim that their breakfast cereal helps children to grow. In their enthusiasm to demonstrate the claim and add "scientific evidence" to their television commercials, the company designed the following experiment.

- A group of children were given daily a bowl of Roasty-Toasties with cream, bananas, and sugar
- After several months, each child was weighed.
- It was found that they gained an average of 8 pounds each.

The company concluded that the weight increase was due to the nourishing breakfast, and consequently the company recommended this breakfast for all children.

What are some problems here? One problem was that the children also ate lunch and dinner. Consequently, the weight gain might be due to the food eaten at these other meals.

So, how do we run a new experiment to fix this?

- This time we use two groups of children.We will make sure that the average age and average weight are the same for each group.
- For breakfast, one group received the recommended cereal with cream, bananas, and sugar; toe other was given scrambled eggs.
- The two groups ate approximately the same foods for lunch and dinner.
- After several months each child was weighted.

Let's say that it is found that there was an average gain of 5 lbs in the group that received the recommended breakfast cereal and an average gain of only 1 pound in the group that was given eggs for breakfast.

What's wrong here? Might the weight gain be caused by the cream, sugar, and bananas and not by the cereal? Third experiment!

- In a new study one group received the cereal with cream, sugar, and bananas for breakfast, but now another group received equal amounts of cream, sugar, and bananas (but no cereal) each morning.
- Once again lunch and dinner were approximately the same for both groups and the children's weights at the onset of the study were about the same.

After several months they weighted each child and found that children in both groups gained an average of 5 lbs. The group that received cereal did not gain more weight than the other group.

We call the group that is receiving the treatment that one predicts will cause some change the *experimental group* in an experiment. The group

that gets roughly the ordinary or the traditional, but comparable to the treatment, is called the *control group*.

In any experiment, we have to define our terms, and in particular, we have to develop *operational definitions* of our terms. The makers of Roasty-Toasties wanted to claim that their cereal led children "to grow." In the above experiments, we have operationally defined "to grow" as *"to gain weight."* We could imagine other definitions, such as "to gain height" or even "to gain muscle mass."

There are many different ways to organize an experiment to test a claim. In the rest of this section, we explain several of these.

Quasi-Experimental Design

In the Roasty-Toastie experiment we had a fairly high degree of control over the participants and what they ate. However, one cannot always construct an experimental situation with a great deal of control. For example, when researchers study the psychological reaction to natural disasters such as earthquakes or planes flying into buildings, they can't control when it happens, or who is in the experimental or control groups[1]. Education research in real classrooms cannot always control who is in what class.

In addition, we usually want to know if the results we find in a carefully controlled lab experiment will *generalize* beyond the lab to real life situations. One approach for increasing the generality and relevance of our research is to move the research from the lab to the setting in which the phenomenon that we are studying occurs naturally. For example, in the lab you could not study the psychological effect of experiencing some unplanned event such as the destruction of the World Trade Center in NYC. Neither can you study in the lab the impact of some large-scale intervention policy such as the Head Start Program[2]. If we were studying the effect of anxiety on final examination performance, a possible natural setting would be the actual final examination session of a college course. If our theoretical issue deals with interpersonal relationships between strangers in a large city, then the natural setting is the streets of that city.

Although moving research into the field does not preclude rigorous experimental designs, it is often the case that as we move outside the lab and its highly controlled environment, we find ourselves able to control fewer of the factors that influence the behavior or our participants and thus less able to rule out alternative hypotheses. We consider the lab a *closed system* and the outside world an open system in which the participants are influenced by a number of factors over which experimenters have little control.

---

[1] And it would be unethical to try!

[2] `http://en.wikipedia.org/wiki/Head_Start_Program`

It is possible to perform useful research in the field even with lowered control. For example, one applied study asked whether rear-end collisions could be reduced by adding a warning device to the backs of cars[Voevodsky, 1974][3]. The independent variable was an amber light that indicated the rate of deceleration and was affixed to the rear ends of cabs. A group of cabs that did not have the device served as a control group. At the end of the experimental period, the group of cabs with the device had a rear-end collision rate lower than that of the control group. Thus, one might claim that the warning device reduced rear-end collisions.

We told you that scientists consider any claims and wonder first if that's right. What are the alternative explanations? What types of questions might you ask in evaluating this study? One alternative explanation is a *Hawthorne effect*, where the fact that the experimental group is *observed* might change the group's behavior. Might the drivers in the experimental group have seen themselves as part of an experiment and been more careful in their driving? The author of the research suggested that if there were such an effect, then we would expect to see an overall reduction in accident rates for the cabs. That is, one might expect that both the control and experimental groups would have had lower accident rates during the experimental period than during similar periods in previous years. The rate for front-end accidents, in which the taxi runs into another car was the same for both the experimental taxis and the control taxis. Also, there was no reduction in front-end accidents. The only reduction in accidents was for those causes by other cars running into the cabs from behind, and that reduction occurred only in the experimental group.Thus, it could be assumed that driving a cab with the device did not influence how carefully the driver of the cab drove, but that it did influence the driving of those who were following the cabs.

Suppose you wanted to examine how people's speech patterns change under situations of stress. A team of researchers used the Internet to explore this issue. They wondered if people would feel a greater sense of connection to their fellow humans during times of stress. Now it's one thing to predict a "sense of connection," and another to *measure* it. How would you operationalize this "connection"?

The researchers looked at sites that allow individuals to share online diaries (e.g., blogs). They focused on the period before and after 9/11. Specifically, these researchers downloaded the diaries of 1084 U.S. individuals for a 4-month period including the 2 months before and the 2 months following 9/11. With the average individual updating their diary every 2 days, this resulted in around 72,000 entries overall. The researchers found that in the short term following 9/11, individuals used more plural pronouns such as "we" in their diaries and fewer uses of "I."

Whenever we attempt to study real-life events or to increase the external validity of our inquiries by studying phenomena in real-life situations,

---

[3] `http://www.apa.org/research/action/brake.aspx`

we run a strong risk of decreasing the *internal validity* of our experiments. In most cases the internal validity of our research is decreased as we move from the lab to more natural settings. Yet the overall applicability and relevance of our research might be increased greatly and this might enhance the value of the work. Thus, we are always faced with a tradeoff between (1) precision and direct control over the experimental design and (2) generalizability and relevance to real-life situations.

There are various types of quasi-experimental designs, and here we want to talk about a few of them

Time series design

A time series design is a *within-subjects design*. In a within-subjects design, the performance of a single group of participants is measured both before and after the experimental treatment. Instead of comparing an experimental group to a control group (a *between-subjects design*), we compare participants to themselves.

For example, suppose you have a background in relaxation training and you are talking to the gymnastics coach at the college at which you teach. The coach tells you her team practices well but falls apart at meets; when she hears about your background she asks you to try relaxation therapy with her team for the week prior to the next meet. So, you now decide to compare the team's performance at the next meet to their performance at the prior meet. During the next meet the gymnasts' total score is better than in the previous meet.

Before concluding that the exercises caused these changes however, we should keep in mind that we have used only a single pretest-posttest measure (performance in the two meets). We have no idea how much fluctuation would normally occur between *any* two meets. Perhaps the sharp increase in the score is independent of the relaxation exercises. Not knowing the normal amount of fluctuation between any two measures is a serious weakness of this type of simple time series design.

Interrupted time series design

One way to fix the design is to use multiple pretest and posttest scores in order to give us a better estimate of the normal fluctuations from test to test. Once we know the amount of normal fluctuations we can better interpret the impact of the phenomenon that we are studying.

We might try to fix the gymnast experiment by doing a time series design where we incorporate the scores for several meets before and after the introduction of the relaxation exercises. If the scores rose for several meets after the treatment, it would be reasonable to assume that the apparent change in performance reflected a real shift. But if there was lots of *variance* among the scores, we might decide that the sharp increase in performance after introducing the relaxation procedures was simply chance

fluctuation that would have occurred anyway. If the scores rise reliably after the treatment, but then drop, the results would be consistent with the idea that the gymnast's performance was improved only temporarily. Although the interrupted time series design is a big improvement over the plain time series design, it still leaves us far short of a clear statement of how one variable influenced the other.

However, any number of other events might be contributing to this usually abrupt shift in team performance. Perhaps the increased care and attention given to each athlete was responsible, not the exercises themselves. Maybe some campus event that occurred about that time caused the shift. In an ideal situation we could control for many of these alternative interpretations by including a control group of some sort. However, because all members of the sample under study have been exposed to the relaxation procedure, it is impossible to select a control group. This is an important limitation of the interrupted time series design and it must be kept in mind when we are interpreting the outcome.

As it turns out, it is sometimes possible to create a control group.

Multiple Time Series Design

A multiple time series design attempts to rule out some alternative interpretations by including a control group that does not receive the experimental treatment. Because it uses a second group of participants, the multiple time series design is not a within-subjects design like the interrupted time series design. Rather, it is a between-subjects design.

In our study of the effectiveness of relaxation procedures on the performance of gymnasts, it might be helpful to use a control group of gymnasts from a neighboring college. We could use their total weekly team scores as control data. To the extent that these participants are similar to our original participants on any relevant individual difference variable and are living under similar social and environmental influences, we can assume that the two groups are equal for factors other than the experience of the relaxation training itself.

The difference between the scores of the two teams in the two meets following the special training might give us confidence with the idea that the relaxation procedures had a definite but transient influence on performance. Before accepting this conclusion though, keep in mind that we might have overlooked some social or environmental influence that affected our experimental group and not our control group. For example, some local campus event influenced team spirit and consequently their performance. This later interpretation is always a real possibility when using multiple time series design to study complex phenomena in their natural settings.

On September 12, 1983, New York state put into effect a bill that required a 5 cent deposit on bottles and cans in hopes this would decrease littering. Researchers tested this by counting the number of bottles and

cans found along a highway exit in New York and in a similar site in New Jersey which did not have a returnable law. The researchers made seven observations two weeks apart before the enactment of the law and seven times after the enactment of the law. These researchers reported a decrease from 260 items of liter in NY before the enactment of the law to 145 after. The litter in NJ showed virtually no change; 221 to 214. The researchers made their multiple time series design stronger by measuring nonreturnable littler at both sites which showed no changes and by doing a follow-up at both sites one year later that showed a continuing decrease in litter for the NY site but not for the NJ site.

Now, think like a scientist. Are there alternative explanations? For example, how might the homeless and where they are distributed play a role here?

Nonequivalent Before-After Design

This type of design is used when we want to make comparisons between two groups that we strongly suspect might differ in important ways even before the experiment begins. Because the two groups in this design are initially different, there is an unusually high risk of ultimately confusing the initial differences with the effects of any treatment, which we refer to as the *independent variable*. In any experimental design, you are trying to figure out if the things you can measure and care about (*dependent variables*) are influenced by the independent variable (the one that you can change). Consequently, in this design we avoid simply comparing both groups on a single dependent measure. Instead, each group is given a pretest and a posttest and we compare the amount of change for each group. By comparing the change in scores rather than a set of single dependent measures, we attempt to control more directly for the fact that we are dealing with groups that are different to start with.

This design is widely used in educational research in which we often are interested in comparing different schools, classes, or programs. As an example, suppose you are teaching two sections of an intro philosophy course, one at 8am and one at 3pm. Further, suppose you have two different teaching techniques you want to try out: (a) one in which you just lecture and (b) the other in which you try to get a lot of interaction with your students. You decide to do the discussion version with the 8am class and the lecture with the 3pm class.

What are the dangers to validity here? What are the alternative explanations from the predicted relationship between your independent and dependent variables? One is *non-random assignment*. Students who choose an 8 am class are *different* in attitude from students who choose a 3 pm class. That difference in attitude might interact with engaging in interaction – and whether that interaction leads to learning. So, we would need an attitude questionnaire both before and after the treatment, to try to measure that difference.

Retrospective Design

In most of the experiments we have talked about so far, the experimenter planned a study that would take place in the future. In a retrospective design researchers attempt to examine relationships based on events that have already occurred. The most common use of retrospective designs is in studies of educational techniques, studies of disease, and studies of psychopathology.

For example, in one study researchers looked at the number of listed activities of students in a high school yearbook to determine whether there were differential rates of high school activity between people identified later in life as schizophrenic and non-schizophrenic. The researchers' assumption was that more activities would be indicative of someone who might become schizophrenic. Because the study took place after the fact, these researchers has to determine a control group after the fact.

What control group might we use? The researchers would need a group that was *like* the schizophrenic group as much as possible. For their control group, they chose the students who pictures appeared next to those of the schizophrenic students in the yearbooks.

In one sense you are working backward in the retrospective procedure. You know the outcome (schizophrenia or not) and want to determine the antecedents (things that came before) of this outcome. Thus, one of the important questions is whether you selected a good measure (for example, high school activities) on which to compare the two groups. Of course, this question is unanswerable and for this reason the retrospective design is only a weak form of inference.

Still, testing alternatives this way can lead to a greater understanding of the phenomenon under study. Many retrospective designs are essentially a type of correlational study, so let's turn to this type of design now.

Correlational Design

A typical approach in a correlational design it to identify a situation in which the variables of interest occur, and then passively observe their occurrence. For example if someone were interested in the relationship between how often a baby was held and how often it cried, a first step might be to collect data about these variables for a number of babies. Once data were collected for the two variables–amount of time crying and amount of holding–the correlational statistic could be calculated and this would be one means of defining the degree of relationship between those variables.

We will talk about calculating *correlations* in a couple chapters. Research studies developed with the goal of describing a relationship between two variables but not attempting to show how one variable influences the other are called correlational studies. Correlational procedures are an important initial step in the process of determining a relationship between variables.

For example, Bremner and Narayan[Bremmer and Narayan, 1998] were interested in the manner in which stress influences brain development and memory. The particular area of interest in the brain was the hippocampus which is related to memory processes. As an initial step, they reviewed the literature and found that veterans with combat-related posttraumatic stress disorder (PTSD) showed a significant correlation between levels of combat exposure and hippocampal volume as shown by brain-imaging techniques and lower recall scores on a memory task. Although you might want to conclude that combat stress reduces the size of the hippocampus, these researchers point out that it could be the opposite situation. People who from birth have smaller hippocampus might experience stressful situation as more traumatic and be at greater risk for PTSD.

Consider a bizarre relationship: the correlation between stock market prices and the length of women's skirts. It has been found that when the length of skirts rises, the stock market tends to do better; when skirt length increases, the stock market tends to do worse. This example demonstrates the importance of considering the possibility of unknown factors influencing both variables under study that might produce the relationship. Can you think of a 3rd variable that might influence the other two variables?

As you consider these for yourself, the meaning of the often-quoted statement "correlation does not imply causality" becomes clearer. With correlational designs no variables are manipulated, and thus there are no independent and dependent variables. However, some researchers have attempted to portray a correlational design as if it were of an experimental nature. To illustrate, assume that a researcher was interested in the question of how depression and study habits are related. Some people might try to answer this question by dividing participants at the median into high and low-depression groups based on some measure of depression such as the Beck Depression Inventory[4] and then treating depression as an independent variable and the amount of study time as the dependent variable. This could lead one to interpret the results, which should be interpreted in correlational terms, as if one variable (depression) had caused the changes in the other variable (study habits). But remember, no variable was manipulated; it is therefore not possible to infer that depression affected the amount of study time. In fact, one might just as well draw the opposite inference: The amount of time spent studying influences how depressed a person feels. Tere is also another alternative, called the *third-variable problem* in which a third, unmeasured variable influences the other two. For example, in the skirt-length and stock market price issue, maybe there's a sense of optimism or security in the society that is that actual causal variable.

[4] `http://en.wikipedia.org/wiki/Beck_Depression_Inventory`

Naturalistic Observations

In many ways the method of naturalistic observation derives form what might be our most primitive way of learning about the world: simply paying attention and observing what happens.

One elegant example of this approach sought to determine how men and women carry objects, which was part of a larger theoretical question about gender differences. These researchers simply observed male and female college students as they carried books around campus. In this study it was found that 92% of the women carried books in front of their bodies, with one or both arms wrapped around the books, whereas 95% of the men carried the books at their sides using one hand.

In addition to offering a method to study a focused question such as gender and posture, the method of naturalistic observation can also be useful in extremely complex situations such as studying animals in the wild or the early stages of investigating a phenomenon. If little is known about the phenomenon, we can benefit tremendously from a detailed description of it.

So, there are two key functions of naturalistic observations:

- First, it allows us to amass descriptive knowledge about a phenomenon.
- Second, as we become more familiar with it, we might gain insight about general patterns or lawful relationships in the phenomenon which we can then test using an experimental method in which we manipulate a variable and look at the effects on another variable.

There are concerns with using naturalistic observations.

Data Collection: If participants realize they are being observed, they might behave differently. What a participant's behavior is influenced by the mere presence of the observer, it is called *reactive behavior*. Reactive behaviors tells us what people are like when they know they are being observed; they tell us potentially very little about behavior under normal circumstances.

To keep observations free from reactive behaviors, researchers try to be unobtrusive. A good book on this topic is *Unobtrusive Measures: Nonreactive research in the social sciences*[Webb et al., 2000].

Researchers at the Museum of Science and Industry in Chicago wanted to know how popular was the new hatching-check exhibit, and when it became popular. They noticed that they had to replace the tiles in front of the exhibit every 6 weeks. The erosion of the tiles was a measure of the exhibit's popularity and might be more sensitive than asking people what they liked best and less obtrusive than watching them or installing a camera.

Another data collection challenge is that it is difficult to observe accurately because we are influenced by selective perception, that is, the observations of untrained observers are markedly influenced by what they

expect to see. The extent to which our observations are restricted by our selective perceptions has a great impact on the accuracy of our observations. Fortunately, it is possible to teach observers to observe more accurately.

Participation of Observer with the Observed: Every observer must face two issues.

- The first is whether to conceal one's identify as a researcher.
- The second is whether to participate in the social process one is observing.

The risk in both cases is, again, affecting the behavior of the observed. We are going to re-visit these concerns when we read the gang study discussed in Chapter 3 of *Freakonomics*.

2 Getting Data from the Internet for Computational Analysis

Before we can do any analysis of data, we need to go get it, and get it in a form that we can do some computation on it. In this chapter, we get started with that: Installing Python, and describing how to use it to get data from the Internet.

2.1 Starting with Python and SciPy

Python is a great programming language for exploring data and analyzing it. We are going to be using a particular branch of Python, *SciPy*[1] or Scientific Python. SciPy includes *NumPy* for numeric processing and *Matplotlib*[2] for graphical visualization. This combination allows us to work with Freakonomics-sized data and Freakonomics-style analyses.

This book is not meant to teach you Python. We will assume that:

- Either you had a course using Python[3], and thus met Python already, or
- You know enough about programming from some other course[4] that you can pick up Python from these coursenotes and using other documentation linked to `http://www.python.org`. Matplotlib, in particular, was designed to make it easy to pick up if you know Matlab.

We do provide the first appendix as a refresher on Python.

We are going to use the wonderful implementation of Python available from Enthought (`http://enthought.com/products/epd.php`). Enthought Python is a "batteries-included" implementation of Python, with all the SciPy goodies already implemented, with a powerful development environment included called *IDLE*.

Enthought Python works pretty similar on both Windows and MacOS X implementations. Simply start IDLE on your platform, and you should see

[1] `http://www.scipy.org`
[2] `http://matplotlib.sourceforge.net/`
[3] CS1315 or CS1301 at Georgia Tech
[4] CS1371 at Georgia Tech

a similar interface with similar capabilities on either Windows (Figure 2.1) or MacOS X (Figure 2.2).

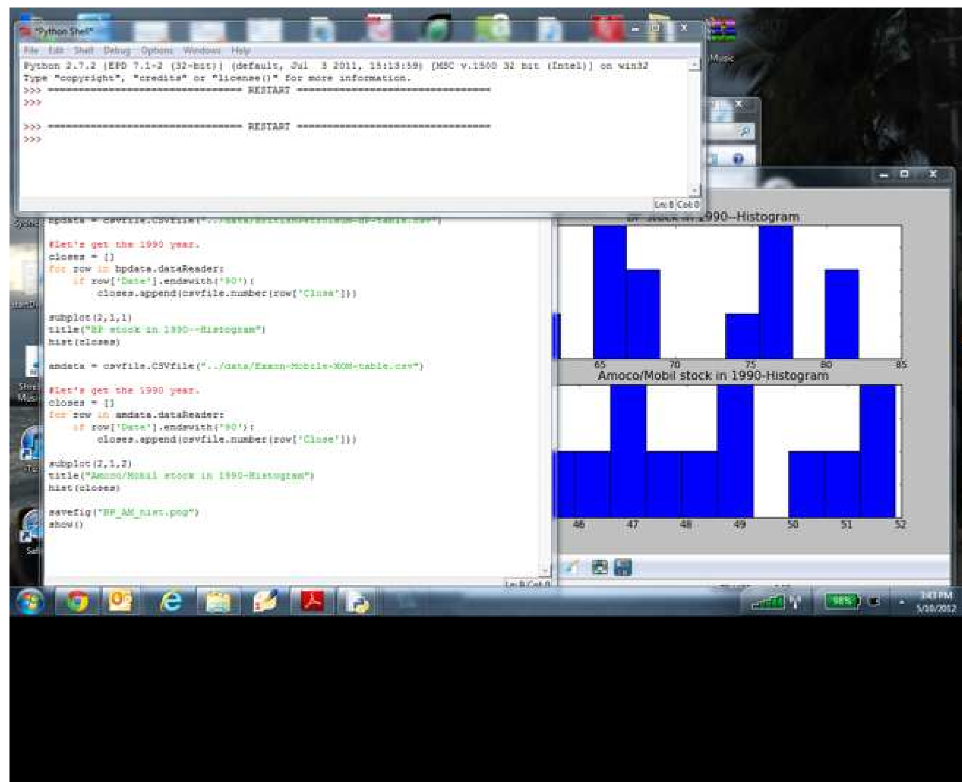


Figure 2.1: IDLE running on Windows

Testing the Installation

Here's how you test your installation. Start up IDLE. Load in Matplotlib by typing **from** pylab **import** ∗ and hitting return. Then do plot ([1,2,3,4]) (or some other numbers. IDLE will respond with a line showing you that a Matplotlib graph had been created. You type show(), and you will see the plot (Figure 2.3).

The interaction might look something like the below:

```
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
"credits" or "license" for more information.
>>> from pylab import *
>>> plot([1,2.5,3.5,6])
```

Figure 2.2: IDLE running on MacOS X

```
[<matplotlib.lines.Line2D instance at 0x01B26418>]
>>> show()
```

2.2 Where can I find data?

Today, the Internet is rich with great sources of data. One reason for this is the rise of Data-Driven Journalism[5]. Journalists seek out data to inform their stories, and then make those data available for others.

In the appendix are three longish Python examples. Aibek Musaev wrote for us three examples of how to read various "live" data sets from Python. These are data sets that are continuously updated, like radiation levels from the Fukushima prefecture every 10 minutes. The code is a bit more complicated than the rest in this chapter, so we move them off into an appendix. These data sets are available in different formats. Aibek provided us with three examples for reading three different kinds of data sets. Thanks, Aibek!

Sources of data

Here are two sources of data for this book, for answering interesting questions on the Internet.

[5] http://en.wikipedia.org/wiki/Data_driven_journalism

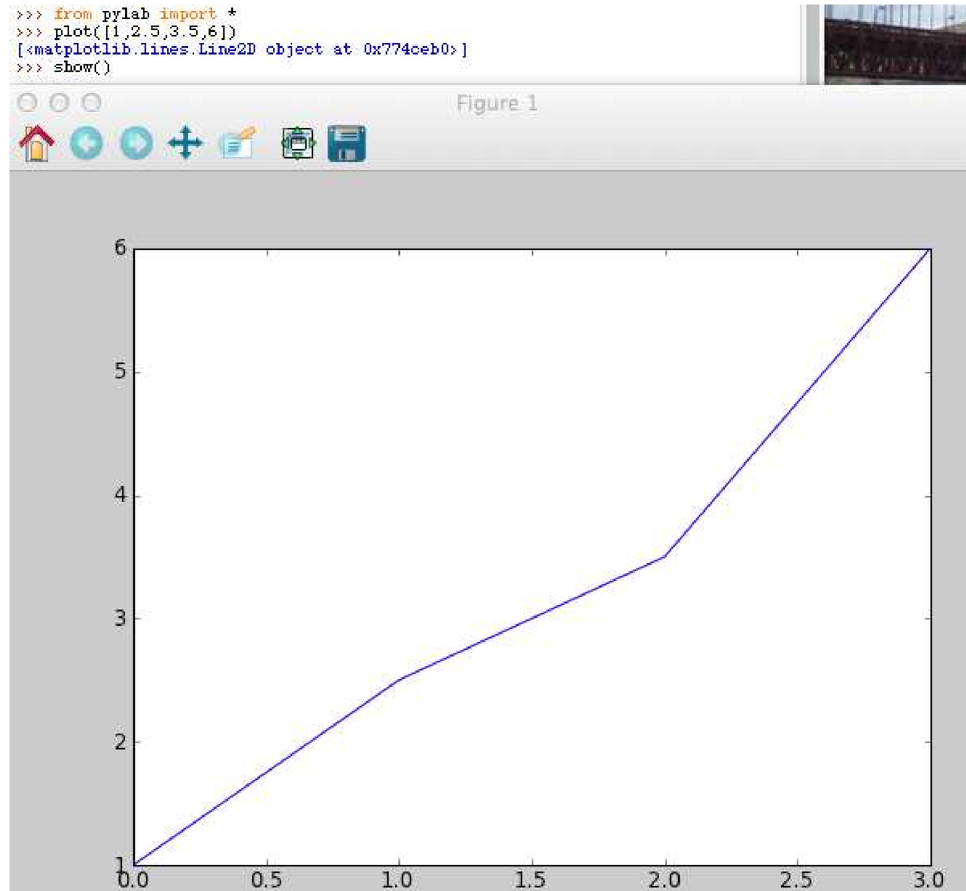


Figure 2.3: Running the first plot

Many Eyes

IBM has created the website *Many Eyes*[6] as a way to engage a wide community in analyzing and visualizing interest data sets. Anyone can upload data, or create a visualization on data that others have uploaded. Commenters can point out interesting aspects of any visualization, generating inferences for others to explore through the available visualization tools. For example, Figure 2.4 is a visualization of the price of wine in pence in England from 1259–1400. Looks like a huge spike in 1307, then a depression in price after 1377. Is that a significant change in price? We'll talk later about how we might know.

We can download the data sets from *Many Eyes* to process in Python.

[6] http://www-958.ibm.com/software/data/cognos/manyeyes/

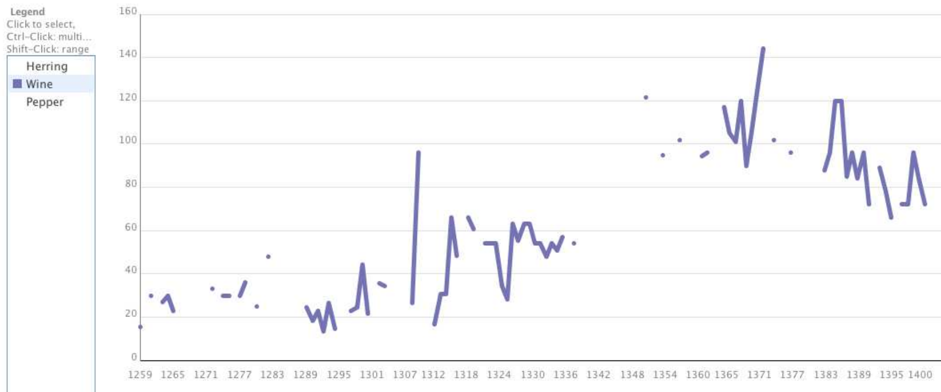


Figure 2.4: Visualization of price of wine in England from 1259–1400

By clicking the *View as Text* button (in Figure 2.5), you can see the data (Figure 2.6) which can then be selected, copied, and pasted into a text file.

From there, we can write a small Python program to read in and process the data, then plot the result (Figure 2.7).

```
import string,sys,pylab

# Make 3 empty lists for the 3 columns
years = []
crownrev = []
revtot = []
# Open the file, and read in all lines
allLines = open("../data/crown-revenue-england.txt",'r').readlines()
# Skip 0, because those are the headers
for index in range(1,len(allLines)):
    # Get the index-th row/line
    line = allLines[index]
    # Split the row into a list of columns
    cols = string.split(line)
    # Add the pieces into the lists
    years.append(cols[0])
    # We'll use eval to convert strings to numbers
    crownrev.append(eval(cols[1]))
    revtot.append(eval(cols[2]))

# Now, plot years by crownrev
pylab.plot(years,crownrev)
pylab.show()
```

Data sets : Crown Revenue in England 1530-1547

Uploaded by: marmarshep Created at: Mar 1 2012
Data source: European State Finance Database
Description:

View as text

	YEAR	CROWNREV	REVTOT
1	1530	49	130
2	1531	49	174
3	1532	49	173
4	1533	49	152
5	1534	49	162
6	1535	54	202
7	1536	55	191
8	1537	55	208
9	1538	69	195
10	1539	79	259
11	1540	100	295
12	1541	100	330
13	1542	100	465
14	1543	100	359
...	...	...	...

Figure 2.5: Grabbing the Crown Revenue as text

Guardian's Data Journalism

The Guardian makes their data available for others to download and analyze[7] (Figure 2.8).

There is a lot of data available there. You can download it as Excel files, which is great for viewing, but harder for manipulation (Figure 2.9). Sometimes, it's easier to copy-paste the data from *The Guardian* into a new spreadsheet, then save it out as a CSV or text file. We'll use some of the data from Figure 2.9 later this chapter.

2.3 Reading CSV Files

It's pretty easy to read and write files in Python. Python uses *backslash notation* for invisible characters, e.g., 'n' is a newline character.

```
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
```

[7] http://www.guardian.co.uk/data

YEAR	CROWNREV	REVTOT
1530	49	130
1531	49	174
1532	49	173
1533	49	152
1534	49	162
1535	54	202
1536	55	191
1537	55	208
1538	69	195
1539	79	259
1540	100	295
1541	100	330
1542	100	465
1543	100	359
1544	101	699
1545	95	732
1546	116	810
1547	55	483

Figure 2.6: Viewing the Crown Revenue as text

```
"credits" or "license" for more information.
>>> #Writing a file
>>> file = open("SampleFile.txt","wt")
>>> file.write("Here is some text!\n")
>>> #Reading a File
>>> file.close()
>>> newfile = open("SampleFile.txt","rt")
>>> newfile.read()
'Here is some text!\n'
>>> newfile.close()
```

Typically, Mark keeps all his data files and Python files in one directory. In some forms of Python, he uses cd (*change directory*) into the given directory, and then starts python. Then, all his files can be accessed without using long paths to special places on the disk. Modern Python implementations (like EPD) are more sophisticated about finding files.

We're mostly going to deal with *CSV* (Comma Separated Values) files in this class. Many of the data sources that one might find on the Internet can produce files of this sort. In this example, we generated a data set from the World Economic Dataset at `http://pwt.econ.upenn.edu/`

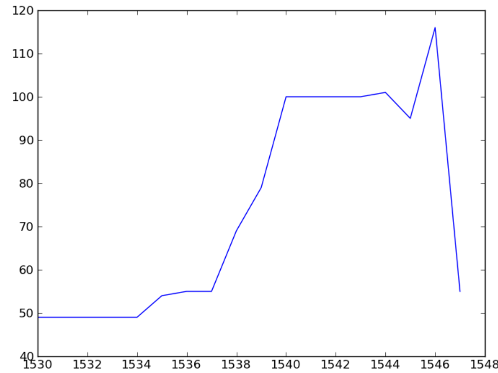


Figure 2.7: Plot of Crown Revenue in England by Year

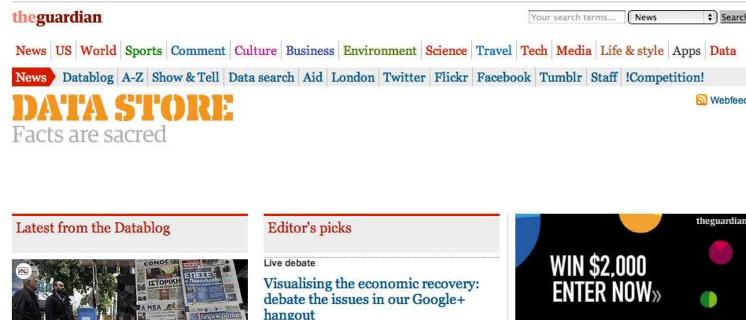


Figure 2.8: Front page of *The Guardian*'s data journalism page

php_site/pwt_index.php (Figure 2.10). We copied the text, opened Notepad, pasted the text into a file, then saved it as somefile.csv.

There are tools built in to Python for handling CSV data. You simply type **import** csv and the package is available. You might be wondering "**import** csv? Didn't we do **from** pylab **import** * a few minutes ago? What's the difference?". When you use **import** csv, you then have to access all the parts of the *module* csv with a *dot operator*, e.g., csv.reader. When you do **from** pylab **import** *, you can access the parts of module pylab just as if they were global (accessible from anywhere, any object) functions, e.g., plot ([1,2,3]) . Sounds like **from**...**import** is the best way of doing it, right? That depends on whether you're *fixing* the code yet. If you have to fix the code, you can update the module in Python by executing reload(csv). If you use **from**...**import**, you can't. When you're developing your own code, it's

Table 1.4 Proportion of adults perceiving crime levels increased in their local area by household characteristics

Percentages | 2008/09 BCS

	Crime type perceived to have gone up 'a lot' or 'a little' in the local area:										
	Overall crime	Bank/credit card fraud	Gun crimes	Knife crimes	Homes being broken into	Cars being stolen	Cars being broken into	Muggings/ street robberies	Vandalism	People getting beaten up	*Unweighted base*
ALL ADULTS	**46**	**52**	**16**	**29**	**33**	**27**	**33**	**29**	**37**	**34**	*33,969*
Structure of household											
Single adult & child(ren)	51	50	20	36	38	28	37	36	40	45	*1,736*
Adults & child(ren)	47	54	17	32	35	27	35	31	39	38	*7,585*
Adult(s) and no children	46	52	15	28	33	26	32	28	36	32	*24,648*
Total household income											
Less than £10,000	45	41	17	28	33	27	32	30	39	35	*5,057*
£10,000 less than £20,000	46	46	14	27	34	28	33	28	39	33	*6,111*
£20,000 less than £30,000	49	56	18	32	37	29	34	32	37	38	*4,582*
£30,000 less than £40,000	47	57	15	30	35	29	36	31	37	37	*3,627*
£40,000 less than £50,000	46	61	16	30	31	27	34	28	40	36	*2,718*
£50,000 or more	45	63	18	33	34	27	32	30	34	33	*4,320*

Figure 2.9: Excel data from *The Guardian* on perception of crime in the UK

At any time, you may change the output format by selecting other format.

comma separated values (.csv)

To return to the previous page, please **don't use the BACK button** of your browser. Click here.

help on how to use the data

```
"country","country isocode","year","POP","csave","rgdptt"
"United Kingdom","GBR","1997","59014","18.53739903","20915.960583"
"United Kingdom","GBR","1998","59237","18.748120916","21692.911061"
"United Kingdom","GBR","1999","59501","17.409775303","22175.425041"
"United Kingdom","GBR","2000","59756","17.766614978","22848.837615"
"United States","USA","1997","268087","21.440527201","30285.74261"
"United States","USA","1998","270560","21.155360892","31365.563666"
"United States","USA","1999","272996","20.420595018","32416.078789"
"United States","USA","2000","275423","20.545083356","33522.99856"
```

Figure 2.10: Example CSV data from World Economics Dataset

often better to **import** so that you can later reload.

Here is the start of a dataset we created of 168 nations' population in the year 2000.

```
"country","country isocode","year","POP" "Angola","AGO","2000","na"
"Albania","ALB","2000","3411" "Argentina","ARG","2000","37032"
"Armenia","ARM","2000","3803" "Antigua","ATG","2000","68"
"Australia","AUS","2000","19157" "Austria","AUT","2000","8110.2"
```

```
"Azerbaijan","AZE","2000","8049"
```

The CSV package knows about how to read individual lines of a CSV file. You open the file, but use that file to create a reader that knows about how to figure out the lines in the file and return the individual pieces in a way that's easily indexable.

```
>>> headerFile = csv.reader(open("pops-2000.csv","rb"))
>>> headerFile.next()
['country', 'country isocode', 'year', 'POP']
>>> headerFile.next()
['Angola', 'AGO', '2000', 'na']
>>> headerFile.next()
['Albania', 'ALB', '2000', '3411']
>>> line = headerFile.next()
>>> line[0]
'Argentina'
>>> line[1]
'ARG'
>>> line[2]
'2000'
>>> float(line[2])  #converts it to be a number
2000.0
```

But even that's not the easiest way to deal with a CSV file. If you assume that the top line is a list of fieldnames (as is common in well-formed CSV files), then you can use a special csv.DictReader to return lines that know what's inside them.

```
>>> headerFile = csv.reader(open("pops-2000.csv","rb"))
>>> headers = headerFile.next()
>>> headers
['country', 'country isocode', 'year', 'POP']
>>> data = csv.DictReader(open("pops-2000.csv","rb"),fieldnames=headers)
>>> data.next()
{'country': 'country', 'country isocode': 'country isocode', 'POP':
'POP', 'year ': 'year'}
>>> data.next()
{'country': 'Angola', 'country isocode': 'AGO', 'POP': 'na', 'year':
'2000'}
>>> nextline=data.next()
>>> nextline['country']
'Albania'
>>> nextline['POP']
'3411'
>>> nextline['year']
```

```
'2000'
>>> nextline.get('country')
'Albania'
```

What next() is returning is a *dictionary*. You can access the dictionary by fieldname, as you can see. You can treat the filenames as indices with square brackets, or using the method get.

Using CSVfile

That's actually enough for you to be able to start downloading and playing with data, but I've tried to make it a little easier. I've created a package called csvfile (Program Program Example #1) that knows about headers and such and provides arrays of data for analysis.

```
>>> import csvfile
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> popdata.headers
['country', 'country isocode', 'year', 'POP']
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> usa = popdata.getRows('country isocode','USA')
>>> usa
[{'country': 'United States', 'country isocode': 'USA', 'POP':
'275423', 'year':
 '2000'}]
>>> usa[0] #just returns the dictionary
{'country': 'United States', 'country isocode': 'USA', 'POP':
'275423', 'year': '2000'}
>>> usa[0]["POP"]
'275423'
```

How do we re-execute lines like popdata = csvfile.CSVfile("pops−2000.csv") so easily? Just press up-arrow. That will allow you to see all the lines you've entered. Hit return on the one you want to execute again. You can also use left and right arrow keys to edit the line before re-executing it.

Where getRows returns all rows (dictionaries) where the field has that value, there is also a method to return a column of all values of a given field.

```
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> pops = popdata.getColumn("POP")
>>> pops[0]
-1
>>> pops[1]
3411.0
>>> pops[2]
37032.0
```

getColumn always returns a bunch of numbers. If there is something that isn't a number in the list (say, the field name "POP"), then a default value of -1 is provided. getColumn is a great tool for getting lists of numbers that we might want to plot–see next chapter.

After doing any of these analyses, the CSVfile needs to be *rewound*. To do the analysis, the file gets read. If you want to do a new search through the data file, you need to rewind to the beginning of the file to do a new search[8].

```
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> usaPop = popdata.getRows('country isocode,'USA')[0]["POP"]
  File "<stdin>", line 1
    usaPop = popdata.getRows('country isocode,'USA')[0]["POP"]
                                                   ^

SyntaxError: invalid syntax
>>> #Forgot the ending quote!
>>> usaPop = popdata.getRows('country isocode','USA')[0]["POP"]
>>> usaPop
'275423'
>>> csvfile.number(usaPop)  #We can use the number converter here
275423.0
>>> popdata.rewind()  #Here's the rewind
>>> ausPop = popdata.getRows('country','Australia')[0]["POP"]
>>> ausPop
'19157'
```

How CSVfile Works

CSVfile will get you started, but at some point, you will have to deal with more complex analyses and data manipulation than it will allow. At that point, you will be writing code *like* CSVfile. It's worthwhile understanding how it works.

CSVfile is written as a *class* from which you create *objects* that understand various *methods* and have various *fields* or *instance variables* associated iwth them.

```
## CSVfile -- a front end to CVS

import csv

def number(input,default=-1):
    try:
        return float(input)
    except:
        return default
```

[8]You're probably realizing from these examples that # is the commenting character in Python. Everything from the # on is ignored on the same line.

The file starts out importing csv since that's necessary for CSVfile to work. A general function is defined number that knows how to convert strings to numbers. Since some values are "na" (not applicable or not available) and others are field names, a default value is created that gets returned whenever a non-number is found.

```
class CSVfile:
    def __init__(self,filename):
        self.filename = filename
        self.rewind();

    def rewind(self):
        self.fp = open(self.filename,"rb")
        headerReader = csv.reader(self.fp)
        self.headers = headerReader.next()
        self.dataReader = csv.DictReader(self.fp,
           fieldnames=self.headers)
```

The next part of CSVfile is definition of the class CSVfile and the initialization method, __init__. This is the method that gets called when we first create a CSVfile, like popdata = csvfile.CSVfile("pops−2000.csv"). You'll notice that all methods in Python start out with the argument of self. That's how Python methods get access to the instance of the class that is being accessed with this method call. Even if your method takes no arguments when you use it, you must still include self as an argument when you define it.

The __init__ method simply saves the input filename as a field (*instance variable*) within the instance, self.filename. Then the rewind method is called. In that method, we open the file (as "rb" which means that it's readable and binary–the csv module likes to be able to get at the binary representation, not just the text), read out the field/header names, then create the DictReader. Notice that we save the headers in an instance variable so that we can access them later.

```
    def next(self):
        return self.dataReader.next()
```

This method allows us to get at individual dictionary rows, if we want, through the next method.

```
    def getRows(self,fieldname,value):
        ret =[]
        for row in self.dataReader:
            if row[fieldname]==value:
                ret.append(row)
        return ret
```

Here's how the getRows method works. It takes a fieldname (like 'country') and a value (like 'Australia') as inputs (and self, as always), then returns a *list* of all the dictionaries where that fieldname matches that value. In the simple population dataset we're using now, that's simple, but one could

also (for example) have more data and pull out all rows for a given year with this method. We create the list that we will be returning with the line ret = []. The square brackets ([]) define a list, and here, an empty list (one with nothing in it to start).

The **for** loop in Python is very powerful. You can iterate through all the rows in the dataset with **for** row **in** self.dataReader–the variable row will take on the value of each row in the data. You can use a **for** loop to iterate through just about anything in Python. Here's an example that iterates through a list to **print** each value in the list.

```
>>> for letter in ['a','b','c']:
...     print letter
...
a
b
c
```

In getRows, we iterate through the list, and everywhere that the row dictionary has the fieldname hold the specified value, we append that row to our return value list, ret. After iterating through everything, we return the return list.

How might you use this elsewhere? You can use **for** loops to iterate all kinds of data, including data that you gathered from different analyses. You could do a search for all the rows with the year 1990, then all the rows with the year 2000, and then iterate through *each* returned list to get the difference in populations.

```
def getColumn(self,fieldname):
    ret = []
    for row in self.dataReader:
        ret.append(row.get(fieldname))
    return map(number,ret)
```

The getColumn method is very similar to getRows. Here, we gather *every* value of the specified filename (like 'POP') and put it in the list. But before we return the list, we map the function number (from the top of csvfile) on to all the values. That's what turns the list of strings (which is what is stored in the CSVfile) to a list of numbers.

Alternative way to read CSV files

Since we started this book, a default way of reading CSV files has been creating in Python. The CSV module is a nice way of managing these files, and is available in every implementation of Python. The CSV module handles things like quoted strings and splitting the line of data into a list.

We're going to start with a slice of the UK Perception of Crime data from *The Guardian*. We copied into a new spreadsheet just the data on perception of crime by income (Figure 2.11).

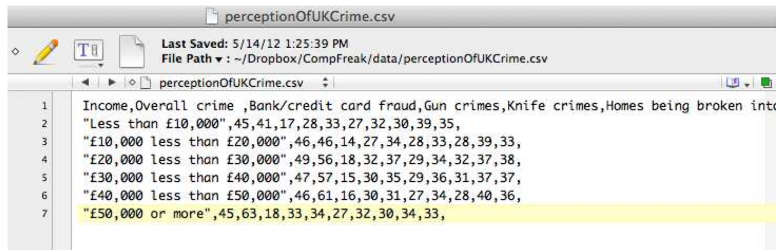


Figure 2.11: Text CSV file from UK Perception of Crime data

We read this data file using a program like this. This program only processes *some* of the data in the file. It generates the graph in Figure 2.12.

```
import csv, string, pylab

# Make 3 empty lists for the 3 columns
incomelevels = []
overallcrime = []
bankfraud = []

# Access the file as a CSV reader
f = open("../data/perceptionOfUKCrime.csv","rb")
reader = csv.reader(f)
# Eat the headers by calling next on the reader
headers = reader.next()
print headers  # Not necessary -- just to see what's there
for row in reader:    # Now read the rest of the rows
    # Row is already a list of columns
    # Add the pieces into the lists
    incomelevels.append(row[0])
    # We'll use eval to convert strings to numbers
    overallcrime.append(eval(row[1]))
    bankfraud.append(eval(row[2]))

# Now, plot overallcrime
pylab.plot(overallcrime)
pylab.show()
```

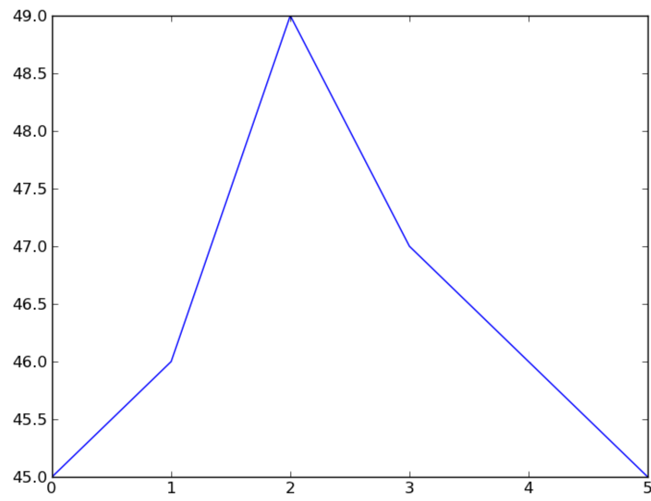


Figure 2.12: Plot of perception of overall crime by income level

3 Plotting

An important part of data analysis is visualizing your data. This chapter describes how to do that.

3.1 Your Basic Plot: Slicing Up The World's Population

To do any plotting, we need to access *Matplotlib* with **from** pylab **import** *.

The basic command to plot is, not surprisingly plot. The function plot can take a variety of different kinds of inputs. The most basic is just a sequence–an array or list of all numeric values.

We saw in the previous chapter how to create an array of populations in the year 2000 for 168 countries. We knew that the first element in the list was from the field name "POP", so we can skip that. It turns out that Python has some powerful tools for grabbing *parts* of a sequence. It's called *slicing*.

For any sequence, you can provide indices for the sequence as square brackets with a colon within them. The first value indicates where to start *from* and the second value indicates the index to stop *before*. That's important–the second value is *not* included in the result. The first index in Python is zero–the first value in any Python sequence is numbered zero. If the first value is missing, it's considered to be 0. If the second value is missing, it's considered to be the length of the list.

```
>>> a=[1,2,3,4,5]
>>> a[0]
1
>>> a[1:]  # From index 1 (second element) to the end
[2, 3, 4, 5]
>>> a[:3]  # From 0 to 2 (not including index 3)
[1, 2, 3]
>>> a[1:3]
[2, 3]
>>> len(a)
5
```

Slicing will work for any sequence, including strings.

```
>>> alpha="abcdefghijklmnopqrstuvwxyz"
>>> alpha[0]
'a'
>>> alpha[2:]
'cdefghijklmnopqrstuvwxyz'
>>> alpha[14:18]
'opqr'
>>> len(alpha)
26
```

All of this is to explain that the list of populations *skipping* the first value is pops[1:]. So, simply put, plotting the populations is plot(pops[1:]).

Once you make a plot, you don't see it. You have a choice what to do.

- You can show() the plot (Figure 3.1). The plot window is really nice and allows you to save it, pan around it, zoom into it.

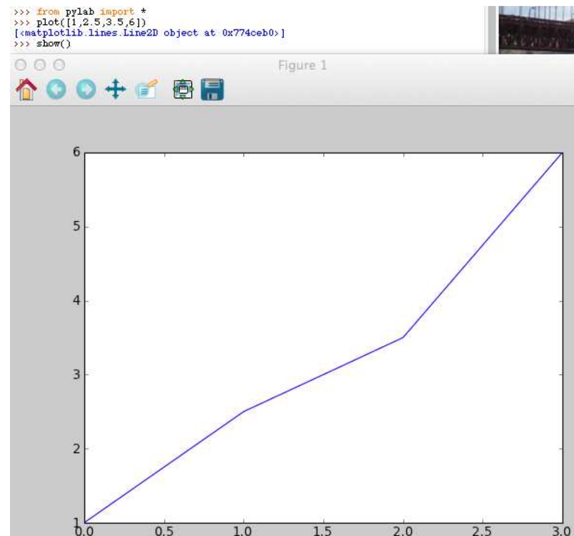


Figure 3.1: Our first plot

- You can savefig.

  The function savefig is really pretty amazing. You simply give it a filename as input, and it tries to save the file in the format specified by the filename. If your filename ends in `.eps`, it will try to save the plot as Encapsulated Postscript (EPS) (Figure 3.2). If your filename ends in `.png`, it will try to save the plot in the portable graphics

format *PNG*. If your filename ends in `.jpg`, it will try to save the plot in *JPEG* format. (Both PNG and JPEG can be inserted into Microsoft Word documents.)

```
>>> plot(pops[1:])
[<matplotlib.lines.Line2D instance at 0x01ABFE90>]
>>> savefig("populations-unsorted.eps")
>>> savefig("populations-unsorted.png")
>>> show()
```

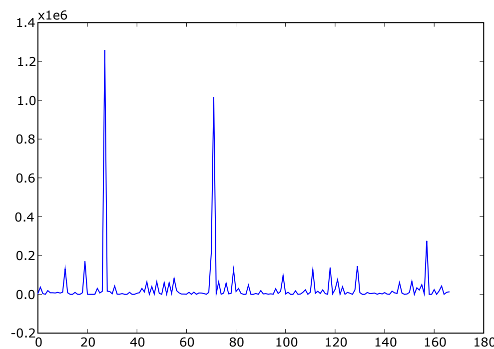


Figure 3.2: Our first graph–countries' populations, unsorted

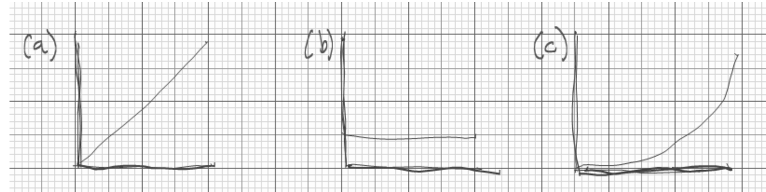
This is a particularly unimpressive graph. The y-axis is obvious–that's populations in thousands. But what's the x-axis? It's countries, in alphabetical order by first name. If you don't provide an x-axis (which you can do, and we'll do it in just a few minutes), the x-axis is assumed to be just the index values of the sequence: 0, 1, 2, and so on.

So let's make the chart a little more interesting. Python knows how to sort any sequence. Let's put the population into sorted order. Our new sequence will start at -1 (see below) – we know that unavailable or label data maps to -1, so we can expect to see at least one of those. The max value is pretty big, 1258821.0314. We then generate the plot, save it, and show it.

```
>>> spops = sort(pops)
>>> spops[0]
-1.0
>>> spops[len(spops)-1]
1258821.0314
>>> plot(spops)
[<matplotlib.lines.Line2D instance at 0x01AE5788>]
```

```
>>> savefig("populations-sorted.eps")
>>> show()
```

Before you look at the plot, think about it. What do you expect to see? How do you think that populations *distribute* around the world? Consider these three possibilities.



Option (a) says that all populations are equally likely in the world, so if you plot them from smallest to largest, it's a gradual slope from left to right. Option (b) says that all populations are roughly the same, so the slope of the line is essentially flat. Option (c) says that all population levels occur, but they grow faster than the gradual slope in (a) would suggest–the biggest countries are much bigger than the smaller countries.

Now take a look at the graph, Figure 3.3. None of the three, is it? What this graph seems to be saying is that most of the populations are roughly *the same* and the curve doesn't really start going up until the very end, where it *shoots* up really fast. A few countries are just enormous, while most are (comparatively speaking) about the same size.

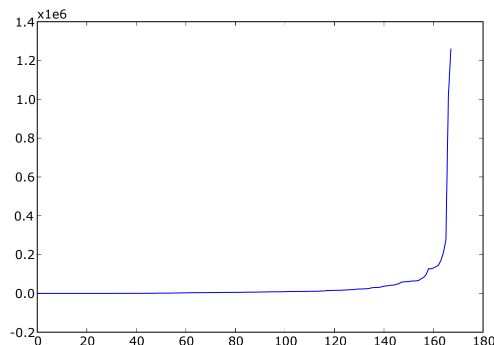


Figure 3.3: Sorted countries' populations

Is that really true? Is that *really* what's going on in this data? One of the nice things about Python's slicing is that you can literally take *slices* through the data with it. What do the first ten value look like? All -1. "Aha!" you may think. "That 'flatness' is just an artifact of populations that we didn't have – NA or Not Available!" Let's take another slice about

half way through (there are 168 values here) – by index 50, the values are *not* -1. And by the end, they are huge.

```
>>> spops[0:10]
[-1.,-1.,-1.,-1.,-1.,-1.,-1.,-1.,-1.,-1.,]
>>> spops[50:80]
[ 1199.   , 1230.   , 1301.   , 1303.   , 1369.   , 1988.   , 2031.   ,
2035.   ,
       2372.   , 2633.   , 2856.   , 3018.   , 3337.   , 3411.   , 3695.   , 3786.9 ,
       3803.   , 3811.   , 3831.   , 4018.   , 4282.   , 4328.   , 4380.   , 4491.   ,
       4527.   , 4886.81, 4915.   , 5024.   , 5031.   , 5071.   ,]
>>> len(spops)
168
>>> spops[160:168]
[  131050.    ,  138080.    ,  145555.008 ,  170406.    ,
210420.992 ,
         275423.    , 1015923.008 , 1258821.0314,]
```

Graphs are obviously darn useful here, but plots alone don't tell us everything. We need to look at some of the numbers. Do we need to look at all the numbers we did above? And did we look at the *right* values above – what if the values *all the way* from 0 to 49 are -1? Would that change your opinion about the graph? In this class, we're going to learn about a variety of techniques for describing values, to get a sense of what the data are doing in a set and how they relate to other data. Graphing is really useful, but it's only one way to look at the data.

3.2 Options on the Plot

The plot method has lots of ways that it can be used. One is that you can pass in two sequences as arguments–one containing the Y values, and the other X axis values (Figure 3.4). Here, we use arange to generate a bunch of *floating point numbers* (not *integers* but numbers with a decimal place) between 0 and 3, spaced out 0.05 apart. We then generate another array, s, by using a special version of sin that iterates over the array t and generates a new array element for s. There's a loop there, but it's hidden inside of sin. It's called a *universal function* (or *ufunc*), and they're documented in the *Numeric Python* documentation.

```
>>> t = arange(0.0, 3.0, 0.05)
>>> t[0:5]
[ 0.  , 0.05, 0.1 , 0.15, 0.2 ,]
>>> s = sin(2*pi*t)
>>> s[0:5]
[ 0.        , 0.30901699, 0.58778525, 0.80901699, 0.95105652,]
>>> plot(t,s)
```

```
[<matplotlib.lines.Line2D instance at 0x00C1BE18>]
>>> savefig("C:/temp/sinplot.eps")
>>> show()
```

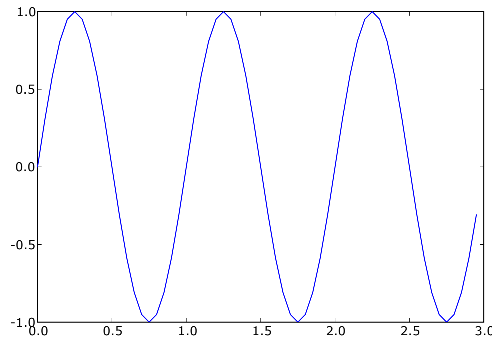


Figure 3.4: A graph generated with X and Y values

Generating plots from files

You don't really want to type in all those commands at the command prompt each time you want a plot. Instead, it's easier to put these commands in a file like this (see also Program Program Example #2):

```
from pylab import *
import csvfile
popdata = csvfile.CSVfile("pops-2000.csv")
pops = popdata.getColumn("POP")
spops=sort(pops)

plot(spops[1:],marker="o",color="r")
title('Populations of countries in the year 2000')
xlabel('Countries in increasing order of population')
ylabel('Population in millions')
grid(True)
show()
```

This "file" ("program"?) is doing the necessary **import** commands, setting up the data, then generating the plot (Figure 3.5). You can use just about any text editor for creating this file–Word might work, but Notepad would be better. *WinEdt* is a really nice editor for text on Windows. There are also editors like emacs and vi that you can use. Just make sure that the filename always ends in '.py' to stand for Python.

You'll notice that we're also doing a lot more tweaking to this graph.

- We can label the X-axis and Y-axis with xlabel and ylabel.
- We can title the whole graph with title.
- We can define a marker for the line. Markers can be ’+’,’,’,’o’,’.’,’s’,’v’,’x’,’<’, or ’>’.
- We can define a color for the line. Here we’re using ’r’ for *red*. Colors can be:

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black (go figure)
w	white
(0.25,0.35,0.5)	An RGB triplet (*tuple*) where the scale is $0..1$
red	Any HTML color name

- Not used here, we can also specify a linestyle: ’-’, ’:’, ’-.’, or ’–’.
- We can use the same color parameters to specify a markeredgecoor and markerfacecolor and even markersize (in points) if we wanted.

Now, how to run this code. In *IPython*, it’s easy. There are commands to cd to the right directory (where you put the file) and then run the file.

```
In [2]: cd C:/Documents\ and\ Settings/Mark\ Guzdial/My\
Documents/Work/CompFreak

In [3]: run fancierplot.py
```

In other forms of Python, you need to *import* the file. If you change the file (to fix a bug, to generate a slightly different plot), you can *reload* the file to re-execute it.

```
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
"credits" or "license" for more information.
>>> import fancierplot
```

3.3 The Plot Thickens: Combining Plots to Determine US and UK Growth Rates

In work in *Freakonomics*, you will often want to compare multiple plots at once. The easiest way to do this is by putting both lines that you care about on the same plot.

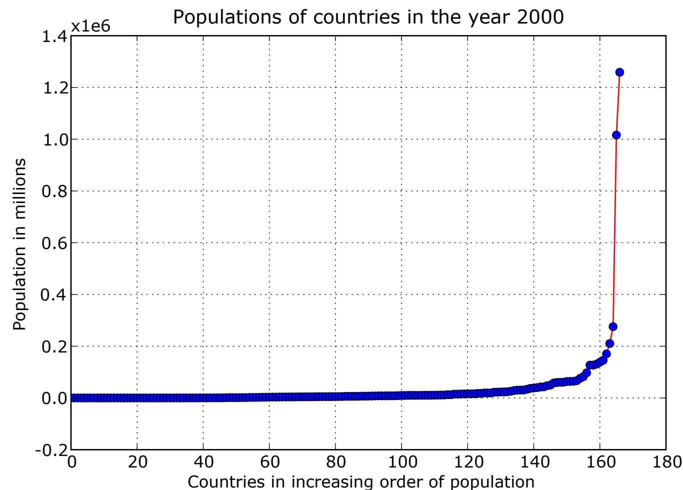


Figure 3.5: Making a fancier plot

I generated a dataset with all the populations (and savings rates, and other fields) for the US and the UK from 1990 to 2000. Here's a segment of what the file looks like:

```
"country","country isocode","year","POP","XRAT","csave","rgdptt"
"United Kingdom","GBR","1999","59501","0.6181","17.409775303","22175.425041"
"United Kingdom","GBR","2000","59756","0.6609","17.766614978","22848.837615"
"United States","USA","1990","249981","1","18.785790541","26365.46609"
"United States","USA","1991","252677","1","18.4248996","25893.640342"
```

We can write code to withdraw the relevant data from this file, and then plot it.

```
from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
uspops = []
for row in usdata:
    uspops.append(csvfile.number(row['POP']))
ukpops = []
for row in ukdata:
    ukpops.append(csvfile.number(row['POP']))
years=range(1990,2001)
print "US",uspops,len(uspops)
print "UK",ukpops,len(ukpops)
```

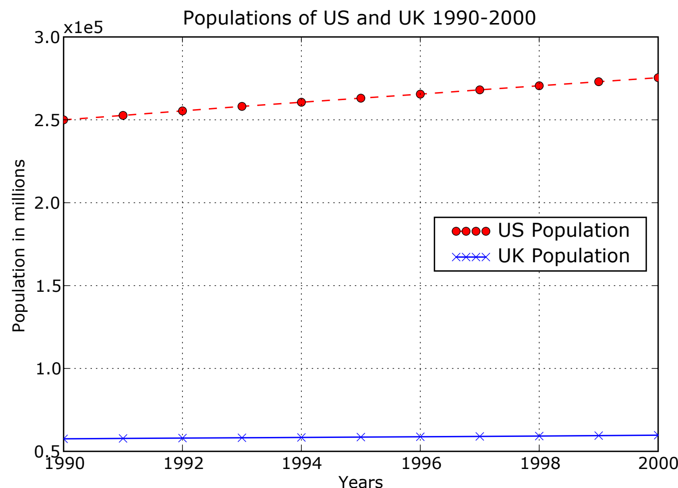
```
print "Years",years,len(years)

plot(years,uspops,'r--o',years,ukpops,'b-x')
legend(('US Population','UK Population'),loc='center right')
title('Populations of US and UK 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)
savefig("us_uk_pop_plot.eps")
show()
```

How it works: We open the datafile, grab the US data, rewind it, then grab the UK data. We make an assumption that the data is still in year order – that could have been a bad assumption, and there are other ways to grab the data so that we don't have to assume that. The data in usdata and ukdata are now in row/dictionary form. To get just the population ('POP') data, we create lists with those values, converted to numbers.

Notice that we can check results with **print** statements. It's okay to have them in your file, mixed in with all the **def** statements. They'll work, and they'll help you figure out what's going on.

We then plot with X (year), and Y (population) data, in the same plot command (Figure **??**). You'll note that we can specify the line color, line style, and marker in a string next to the relevant plot.

From this plot, it looks like the US population has been rising steeply, while the UK population has been essentially flat. That's what it *looks like*, but we'll see later that there are other ways to look at it.



Obviously, the legend function generates a legend. The legend function doesn't *have* to have a loc (location) parameter. We found that the default

(so-called 'best') location is the upper-right, which covered over the US population curve. So I changed the location. How did we figure out *how* to change the legend location? It's not in the Matplotlib documentation (that I could find). Turns out that there's even more documentation within Python using the help function.

```
>>> help(legend)
Help on function legend in module matplotlib.pylab:

legend(*args, **kwargs)
    LEGEND(*args, **kwargs)
    Place a legend on the current axes at location loc.  Labels are a
    sequence of strings and loc can be a string or an integer specifying
    the legend location
    USAGE:
      Make a legend with existing lines
      >>> legend()
      legend by itself will try and build a legend using the label
      property of the lines/patches/collections.  You can set the label of
      a line by doing plot(x, y, label='my data') or line.set_label('my
      data'). If label is set to '_nolegend_', the item will not be shown
      in legend.
        # automatically generate the legend from labels
        legend( ('label1', 'label2', 'label3') )
        # Make a legend for a list of lines and labels
        legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
        # Make a legend at a given location, using a location argument
        # legend( LABELS, LOC )  or
        # legend( LINES, LABELS, LOC )
        legend( ('label1', 'label2', 'label3'), loc='upper left')
        legend( (line1, line2, line3),  ('label1', 'label2', 'label3'), loc=2)
    The location codes are
      'best' : 0,
      'upper right'  : 1, (default)
      'upper left'   : 2,
      'lower left'   : 3,
      'lower right'  : 4,
      'right'        : 5,
      'center left'  : 6,
      'center right' : 7,
      'lower center' : 8,
      'upper center' : 9,
      'center'       : 10,
    If none of these are suitable, loc can be a 2-tuple giving x,y
    in axes coords, ie,
      loc = 0, 1 is left top
      loc = 0.5, 0.5 is center, center
```

As Two Separate Plots

The problem with the legend points out that, sometimes, it doesn't work out well to have the two (or more) lines in the same graph. If the X axes are the same, you can do vertical plots for the same effect. The graphs can be compared, but without having to stick to the same Y axis.

The below program does that using the subplot function (Figure **??**). The subplot specifies how many rows and columns of plots you want (first two arguments) and then which one you're specifying now[1] You'll also see in this program that we add an additional loop to make sure that we're calling up the right year in the right order. This will be important when we try to join data later.

```
from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
# This time, making SURE that they're in year-order
years=range(1990,2001)
uspops = []
for y in years:
    for row in usdata:
        if row['year']==str(y):  #Items in rows are strings
            uspops.append(csvfile.number(row['POP']))
            break #Leave the row loop
ukpops = []
for y in years:
    for row in ukdata:
        if row['year']==str(y):
            ukpops.append(csvfile.number(row['POP']))
            break

# Top subplot: 2 rows, 1 column, subplot #1
subplot(2,1,1)
plot(years,uspops,'r--o')
title('Population of US 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

subplot(2,1,2)
plot(years,ukpops,'b-x')
title('Population UK 1990-2000')
```

[1] What would happen if you changed the rows and columns between subplot calls? Dunno.

```
xlabel('Years')
ylabel('Population in millions')
grid(True)

savefig("us_uk_pop_plot2.eps")
show()
```

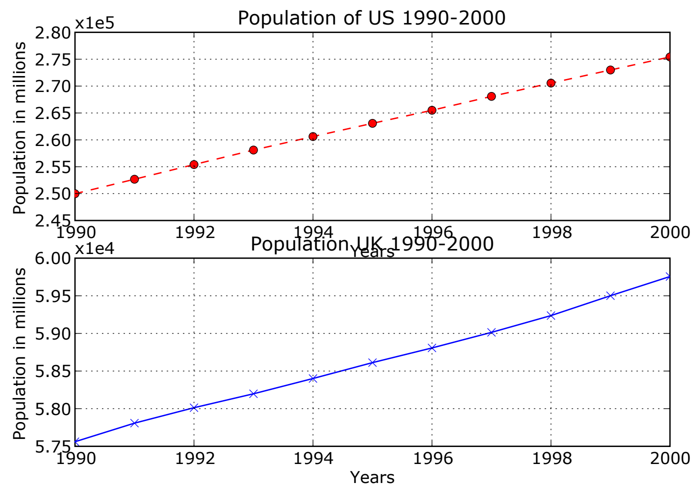


Figure 3.6: US and UK Populations, as two subplots

Notice something important about this double plot. Sure looks like the *slope* of each line *is about the same!*. There are ways to check that assumption later, but it looks like both the US and UK plot have been increasing at very similar rates, but we only see that if we shift the scales appropriately to see how each is increasing.

4 Descriptive Statistics

The way in which we usually describe sets of numbers is with *descriptive statistics*–numbers that reflect the overall picture, range, or distribution of the set.

4.1 Average or mean: Petroleum Tax Prices

The average or mean is simply the sum of the values divided by the number of values.

Let's compute the value of a stock over a given year. From Yahoo Stocks, we can get the monthly value of a stock over some period of time. The below is some of the values for British Petroleum (*BP*).

```
Date,Open,High,Low,Close,Volume,Adj. Close*
1-Jun-06,69.61,72.38,66.20,67.48,4938385,67.48
1-May-06,74.25,76.67,68.50,70.70,3859318,70.70
3-Apr-06,69.50,76.85,69.49,73.72,3520315,73.18
1-Mar-06,66.92,70.68,65.35,68.94,2938130,68.43
1-Feb-06,71.99,72.58,66.01,66.42,3647978,65.93
3-Jan-06,65.50,72.88,65.47,72.31,4301770,71.19
1-Dec-05,67.06,69.25,63.26,64.22,2983219,63.23
```

We can access this data just as we have any other CSV data.

```
In [11]: import csvfile

In [12]: bpdata=csvfile.CSVfile("BritishPetroleum-BP-table.csv")

In [13]: bpdata.next() Out[13]: {'Adj. Close*': '67.48',
 'Close': '67.48',
 'Date': '1-Jun-06',
 'High': '72.38',
 'Low': '66.20',
 'Open': '69.61',
 'Volume': '4938385'}

In [14]: bpdata.next()['Date'] Out[14]: '1-May-06'
```

There are builtin functions to sum across a sequence, and to get the length of a sequence (the number of values there). We can use these to

define an average function. Notice that we multiple by 1.0 to force Python to do floating point arithmetic, rather than simple integer arithmetic.

```
In [15]: a=[1,2,3,4]

In [16]: sum(a)
Out[16]: 10

In [17]: len(a)
Out[17]: 4

In [18]: sum(a)/len(a)
Out[18]: 2

In [19]: (sum(a)*1.0)/len(a)
Out[19]: 2.5
```

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "Closing values",closes
print "Average:",average(closes)
```

How it works: We import pylab and csvfile, as we have in the past. We define a function average which is fine to do in-line. We then open up the file and do a search for all those dates that end in '90' (in order to get the average of the 1990 monthly closing dates). Notice that our loop executes over the dataReader. That's how we did it in csvfile.py.

```
In [26]: run bpAvg1990.py
Closing values [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005, 74.5
, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
Average: 71.9783333333
```

4.2 Understanding Measures of Variability

A measure of central tendency (mean, median, mode) gives us some information about a set of scores; however, it does not give any information about the scores are distributed. To obtain a more complete description of a set of data, we use a second measure in addition to the measure of central tendency. This is a measure of variability.

Measures of variability are merely attempts to indicate how spread out the scores are. One common measure of variability is the *range* which reflects the difference between the largest and smallest scores in a set of data. Although computing the range is easy, the range tells us only about the two extreme scores; it provides no information about the dispersion of the remaining scores if we know nothing about the underlying distribution.

Consider Group A and Group B:

Group A	Group B
2	2
3	5
4	5
5	6
6	6
7	6
8	7
9	7
10	10

In these two distributions of scores, the ranges are the same (8) and the means are the same (6), yet the actual shapes of the distributions are very different. The scores in Group B appear more concentrated in the center of the distribution yet our estimate of range does not reflect this.

To provide a more sensitive distribution of all the scores, we use a second measure of variability of data: the *variance*. The variance is a description of how much each score varies from the mean. Think about the problem we have: How do you come up with a *measure* of how much the data varies from the mean?

One way to describe the dispersion of all the scores would be to subtract each score from the mean and then add these deviations. What is the problem with this solution? The sum would add to 0 because some deviations will be positive and some negative. So, instead we square each deviation and add the squares; we now have all positive numbers. This gives us a description of how much the scores vary from the mean.

We call this a *sum of squares*:

$SS = \sum(\bar{X} - X)^2$

Then, if we divide the sum by the total number of scores, we get an idea about the average deviation for each score which is the *variance*:

$$variance = SS/N = \frac{\sum(\bar{X} - X)^2}{N}$$

If you calculate the variance for Groups A and B you will find that Group B has a lower variance (4) than Group A (6.67).

One problem with variance is that, because it uses squared scores, it does not describe the amount of variability in the same units of measurement as the original data (e.g., seconds versus seconds-squared). Therefore, researchers prefer to use the square root of the variance as their primary *estimate* of variability. This measure is called the *standard deviation*.

$$SD = \sqrt{variance} = \sqrt{\frac{\sum(\bar{X} - X)^2}{N}}$$

Now, one last nicety here. Notice I said the SD was an estimate of variability. That is, we are interested in the variability of the data for the population, not just the sample of data we collected. Because of this, the estimate of the *population variability*, based on the data in the sample, is:

$$SD = \sqrt{variance} = \sqrt{\frac{\sum(\bar{X} - X)^2}{N - 1}}$$

4.3 Computing Standard Deviation

Standard deviation is the amount of *spread* in the values in the data set. If all the values in the data set are exactly the same, then they're all equal to the mean value, and the standard deviation is zero. The higher the standard deviation, the further values differ from the mean.

The standard deviation is the square root of the *variance*. It's the average of the squared differences from the mean. Here's the basic process for figuring out the standard deviation.

1. First, compute the mean.
2. Figure out the difference between each value and the mean, e.g., $x_i - mean$ for all positions i in the data sequence. Then *square* that distance. The square removes the possibility of a negative value, since you don't know which is bigger, x_i or the mean.
3. Sum up all these squared differences, then divide by the number of numbers in the sequence. This is called the *average of the squared differences*, or the *variance*.
4. Finally, take the square root of the whole thing. The idea is to get close to the average of the differences, with the squared-differences and square root in there to deal with positive and negative values.

Let's see a program that implements the standard deviation algorithm. In this program, we gather data from both British Petroleum (BP), but also Exxon-Mobil (*XOM* on Yahoo).

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)



bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** BP ***"
print "Closing values",closes
print "Average:",average(closes)
print "Standard Deviation:",std_dev(closes)

amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
closes = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** Exxon/Mobil ***"
print "Closing values",closes
print "Average:",average(closes)
print "Standard Deviation:",std_dev(closes)
```

The function pow takes the *power* of one number to another number. It works for squaring (pow(something,2)) and for getting square roots (pow(something,0.5)). And here's the run of it.

```
In [30]: run bpStdDev1990.py
*** BP ***
Closing values [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005,
, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
Average: 71.9783333333
Standard Deviation: 6.67530877355
*** Exxon/Mobil ***
Closing values [51.75, 50.630000000000003, 49.0, 49.0, 50.0, 51.88000000000
 47.880000000000003, 48.0, 45.25, 46.25, 47.0, 47.0]
Average: 48.6366666667
Standard Deviation: 2.05769018292
```

Okay, so on average, BP stock had a higher close in 1990 than Exxon-Mobil, but BP also had a higher standard deviation. It varied more over that year than Exxon did. Does that matter? Is it *really* different? And what does *"really different"* mean, anyway?

4.4 Viewing Histogram

Another way of getting a picture of what's happening with a data set is to use a *histogram*. A histogram tells you the number of occurrences of values in a certain range. "Hold on!" you say. "I don't see that in the Matplotlib documentation!" Yet again, you need to use help.

```
In [9]: from pylab import *

In [10]: help(hist)
Help on function hist in module matplotlib.pylab:

hist(*args, **kwargs)
    HIST(x, bins=10, normed=0, bottom=0, orientiation='vertical', **kwargs)
    Compute the histogram of x.  bins is either an integer number of
    bins or a sequence giving the bins.  x are the data to be binned.
    The return values is (n, bins, patches)
    If normed is true, the first element of the return tuple will
    be the counts normalized to form a probability density, ie,
    n/(len(x)*dbin)
    orientation = 'horizontal' | 'vertical'.  If horizontal, barh
    will be used and the "bottom" kwarg will be the left.
    width: the width of the bars.  If None, automatically compute
    the width.
    kwargs are used to update the properties of the
    hist bars
```

```
    Addition kwargs: hold = [True|False] overrides default hold state
```

Here's an example that generates a histogram for each of the BP and Amoco-Mobil stock.

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)



bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,1)
title("BP stock in 1990--Histogram")
hist(closes)

amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.

closes = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,2)
title("Amoco/Mobil stock in 1990-Histogram")
hist(closes)
```

```
savefig("BP_AM_hist.eps")
show()
```

The result is Figure 4.1. This helps some, like showing that BP basically had two common prices during this time, while Exxon-Mobil is more disperse. We call that the *shape* of the distribution.

So, do you think BP and Exxon-Mobil roughly track one another? That is, do they move up or down in the same ways? If they do, one would presume that the impacts on their prices have more to do with external factors (e.g., peace in the Middle East) than with anything in the companies themselves or in the UK or US, respectively. How would we find out? See next chapter...

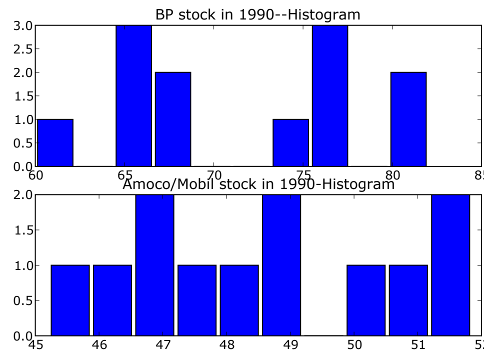


Figure 4.1: BP and Exxon-Mobil stock prices in 1990, as histograms

5 Correlation

How do we *compare* two sequences of values? How do we figure out if BP and Exxon-Mobile change in roughly the same ways at roughly the same times (suggesting that factors external to either company are acting upon both at the same time)? How do we figure out if the UK and US populations grow and shrink at about the same rate (suggesting that whatever factors influence the size of populations are impacting both countries at the same time in the same way)? One way of doing that is with a *correlation*. A correlation is a number that describes how related two data sets are.

5.1 Correlation: A Measure of Association

Suppose we want to look at the relationship between a person's self-professed helpfulness to others (based on a questionnaire score) and the number of times the person was observed helping others over the course of some time period. One way of looking at this relationship is with a *scatterplot* or *scatter diagram* showing these two variables (Figure 5.1). Each point would represent one person and two scores: a score on helpfulness questionnaire and the number of times the person helped someone in the experiment (Figure 5.2). If we noticed that someone who scores low on one measure also scores low on the other, and vice versa, then we'd figure that these variables *are related*.

A useful statistic to calculate to quantify this relationship is a *correlation*. When we perform a correlation, we derive a correlation coefficient which ranges between +1 and -1. A positive correlation represents a linear relationship between two factors such that large values of one measure are associated with large values of the other. There are several types of correlation coefficients that have been developed; one of the most common ones is the *Pearson product moment correlation coefficient*, or r.

$$r = \frac{\sum(zscore_x) * (zscore_y)}{N}$$

where

$$zscore = \frac{\bar{X} - X}{SD}$$

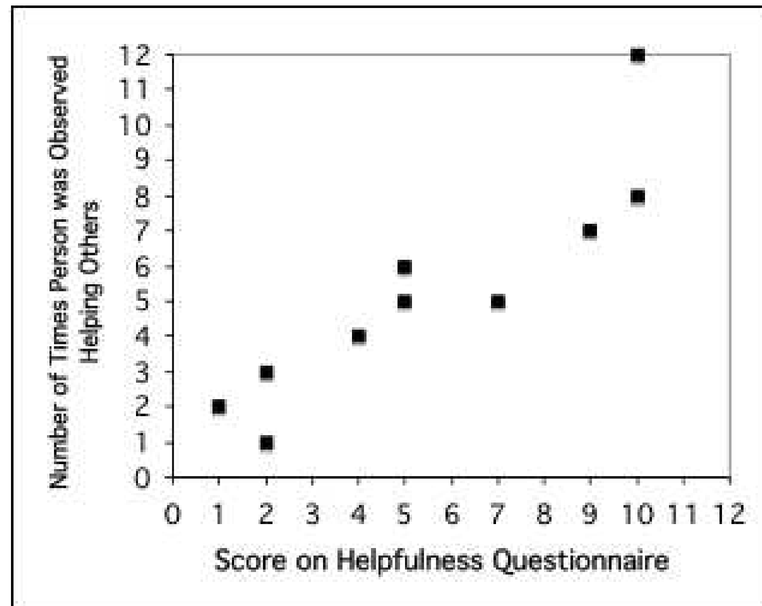


Figure 5.1: Example scatterplot

5.2 Computing correlation: Is it the company, or war in the Middle East?

Let's call one data set x and the other data set y. Each data set should have the same number of elements, call it n. The correlation number is called r. Above, you see a definition for r.

Here's a Python file that computes correlations in two different ways, then processes the data in Figure 5.2.

```
from pylab import *


def r1(x, y):
    """ Original correlation, based on stats texts. """
    if len(x) != len(y):
        print "Sorry! the values must be paired."
        return
    xbar = float(sum(x))/len(x)
    ybar = float(sum(y))/len(y)
    #print "averages for x and y",xbar,ybar
    xsumdiffs = 0
    for num in x:
        xsumdiffs = xsumdiffs + pow(xbar-num,2)
    xsd = pow(xsumdiffs/len(x) ,0.5)
```

Subject	Score (X)	z-score for X	Score (Y)	z-score for Y	(z for X) * (z for Y)
1	2	-1.09	3	-0.76	0.83
2	10	1.41	12	2.21	3.10
3	9	1.09	7	0.56	0.61
4	4	-0.47	4	-0.43	0.20
5	7	0.47	5	-0.10	-0.05
6	10	1.41	8	0.89	1.25
7	1	-1.41	2	-1.09	1.53
8	5	-0.16	6	0.23	-0.04
9	2	-1.09	1	-1.42	1.55
10	5	-0.16	5	-0.10	0.02
	AVG X = 5.5		AVG Y = 5.3		
	SD X = 3.20		SD Y = 3.03		
N= 10		r= 0.90			

Figure 5.2: Data for scatterplot

```
    ysumdiffs = 0
    for num in y:
        ysumdiffs = ysumdiffs + pow(ybar-num,2)
    ysd = pow(ysumdiffs/len(y) ,0.5)
    #print "sds for x and y",xsd, ysd
    zscoresx = []
    for num in x:
        zscoresx.append((num-xbar)/xsd)
    #print "zscores for x:",zscoresx
    zscoresy = []
    for num in y:
        zscoresy.append((num-ybar)/ysd)
    #print "zscores for y:",zscoresy
    sumzs = 0
    for i in range(0,len(zscoresx)):
        sumzs = sumzs + (zscoresx[i] * zscoresy[i])
    return sumzs/len(x)

def stdev(x,xbar):
    sums = 0
    for num in x:
        sums = sums + pow(xbar-num,2)
    return sqrt(sums/len(x))

def zscore(xnum,xbar,sd):
    return (xbar - xnum) / sd

def r2(x,y):
  """ Better correlation, in that it's easier to understand. """
  if len(x) != len(y):
```

```
    print "Sorry! Values must be paired."
    return
  xbar = float(sum(x))/len(x)
  ybar = float(sum(y))/len(y)
  #print "Averages:",xbar,ybar
  xsd = stdev(x,xbar)
  ysd = stdev(y,ybar)
  #print "Standard Deviations:",xsd,ysd
  # Now, compute the sums
  sumzs = 0
  for index in range(len(x)):
    sumzs = sumzs + (zscore(x[index],xbar,xsd) * zscore(y[index],ybar,ysd))
  return sumzs / len(x)

scorex = [2,10,9,4,7,10,1,5,2,5]
scorey = [3,12,7,4,5,8,2,6,1,5]
print "Stats r = ",r1(scorex,scorey)
print "New r = ",r2(scorex,scorey)
```

How it works: r2 is probably easier to read and map to the equation in the previous section. First, we make sure that the lengths of the two arrays (or vectors, in MATLAB-speak) are the same. Then, we compute the averages, and the standard deviations, using a function that does what we saw in the previous chapter. We then write a loop that computes the sum in the numerator, using a zscore for each value in x and y. Finally, we divide by N, which is the same for both arrays.

The r1 version does the same thing, but does all the statistics in-line, without external functions. It's slightly more efficient. But both versions give us the same results:

```
Stats r =  0.900567573182
New r =  0.900567573182
```

Alternative way of computing correlation

If you look in a statistics book, you may see a different definition for r. *Both versions do the same thing!* The traditional statistics book version is easier to compute by hand, but the one above is easier to understand and easier to compute by computer. For the sake of explaining what you might find in books, let's do it here.

$$r = \frac{(n\sum_{i=0}^{n} x_i y_i) - (\sum_{i=0}^{n} x_i)(\sum_{i=0}^{n} y_i)}{\sqrt{(n\sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2)(n\sum_{i=0}^{n} y_i^2 - (\sum_{i=0}^{n} y_i)^2)}} \tag{5.1}$$

Let's talk our way through that mess.

- In the numerator, $(n\sum_{i=0}^{n} x_i y_i)$ is the sum of each pair of x_i and y_i multiplied together. We subtract from that the product of the sum of all the x values $(\sum_{i=0}^{n} x_i)$ and the sum of all the y values $(\sum_{i=0}^{n} y_i)$.

So, the numerator is this sing-song term: the sum of the products of the scores less the product of the sums of the scores.

- In the denominator, it's the square root of a product. The first term in the product is n times the sum of all x values squared *minus* the square of the sum of all x values ($n\sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2$). The second term in the product is the y version of that ($n\sum_{i=0}^{n} y_i^2 - (\sum_{i=0}^{n} y_i)^2$).

We can do this in Python.

```
def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator
```

How it works: Computing n is easy–it's the length. We want to make sure that the two lengths are equal, else the values couldn't possibly be *paired*. The sum of the product of the pairs is just what it sounds like: for all index values i, prod_pairs = prod_pairs + (x[i]*y[i]). The numerator is then numerator = n*prod_pairs - (sum(x)*sum(y)). The sum of the x's squared is for all index values i, x_square = x_square + pow(x[i],2). The y_square is computed in the same way. The denominator

$$\sqrt{(n\sum_{i=0}^{n} x_i^2 - (\sum_{i=0}^{n} x_i)^2)(n\sum_{i=0}^{n} y_i^2 - (\sum_{i=0}^{n} y_i)^2)}$$

is then computed with the code:

```
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
```

The correlation is then numerator/denominator.

Example: Correlating British vs. American Petroleum Stock Prices

Let's look again at our BP vs. Exxon-Mobil petroleum stock prices in 1990. Are these two data sets highly correlated?

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)

def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator


bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
```

```
bpcloses = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        bpcloses.append(csvfile.number(row['Close']))


amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
amcloses = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        amcloses.append(csvfile.number(row['Close']))

print "BP closing values:",bpcloses
print "average",average(bpcloses)
print "number",len(bpcloses)
print "standard deviation",std_dev(bpcloses)

print "Exxon-Mobil (American) closing values:",amcloses
print "average",average(amcloses)
print "number",len(amcloses)
print "standard deviation",std_dev(amcloses)



print "Correlation is ",correlation(bpcloses,amcloses)
```

And here's what the run looks like:

```
In [9]: run bpAmCorrel1990.py
BP closing values: [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005,
74.5, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
average 71.9783333333
number 12
standard deviation 6.67530877355
Exxon-Mobil (American) closing values: [51.75, 50.630000000000003, 49.0, 49.0, 5
0.0, 51.880000000000003, 47.880000000000003, 48.0, 45.25, 46.25, 47.0, 47.0]
average 48.6366666667
number 12
standard deviation 2.05769018292
Correlation is  0.819128769403
```

This correlation r is called the *Pearson Product Moment Correlation*. It has values between -1.0 and 1.0. A negative value suggests a negative correlation–when x goes up, y goes down. A positive value suggests a positive correlation–both values go up at about the same time. A value of 0.82 is pretty darn close to 1.0, so that suggests a strong positive relation.

Is our BP vs. Amoco Result Significant?

But maybe 1990 was a bad year. Maybe if we had looked at 1991 or 1989, we wouldn't have seen any correlation at all, or at least, a much weaker one. What we want to do is consider the probability that what we observed

was just luck. We call this a *significance chance*. Basically, the more values that you have and the stronger the correlation, the more confidence that you can have that the result is *significant*.

The correlation r and the number of pairs are only two of the three variables that you need to determine significance. The last value is the *significance level*, also called the *alpha value*. How sure do you want to be? A common alpha value is 0.05. That means that only 5 of 100 experiments with data samples from these variables would be wrong. If this were medical research, we might want an alpha value of 0.01 or even 0.10. (In Education research, we can sometimes get away with 0.10.)

There is another factor that we're not going to talk about much here, and that's whether you're doing a *one-tailed* or *two-tailed* test. The first means "Are we only testing for one value being *greater* than the other?" where the second one means "Are we testing for *any difference* between the two?" We'll go with two-tailed for now.

The number of pairs is called the *degrees of freedom*[1]. The degrees of freedom for a correlation is $n - 2$.

So, for our data set, we have 10 degrees of freedom (because we have 12 pairs and $12 - 2$ is 10), and we'll use an alpha value of 0.05. Now we need an ever-present correlation table. There are a bunch linked to the class Swiki. The one I'm using is at `http://www.gifted.uconn.edu/siegle/research/Correlation/corrchrt.htm`. The value at *df* 10 and alpha value 0.05 is 0.576. That means that if our r is less than -0.576 it's a significant negative correlation, and if it's greater than 0.576, it's a significant positive correlation.

Since our r value is 0.82, we have a significant relationship. Our inference, then, is that whatever factors influenced the stock price of BP and Exxon-Mobil in 1990, they were mostly the *same* factors, since the stock prices are highly correlated. This doesn't mean that there is a *causal* relationship–just because BP went down, Exxon went down, nor vice-versa. This doesn't say anything about causation at all. But it does say that there is a *relationship*. How would you find out what factors *are* leading to that relationship? Hmm, good question–one for a future chapter.

5.3 But do we believe it?

The correlation computation can feel like a bit of mumbo-jumbo – statistical voodoo. How do we know that it's *really* telling us anything? Let's do some experiments to find out.

We want to get access to our correlation function, so we can put it in a file `correlation.py`.

```
from pylab import *
```

[1]These statisticians have names for *everything!*

```
def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator
```

That way, we can just import it and use it. First test: Are two identical sets of numbers "correlated"?

```
In [10]: from correlation import *

In [11]: correlation([1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10])
Out[11]: 1.0
```

That's good. We would expect two identical sets of numbers to be correlated as strongly as they possibly could be.

But maybe it's easier with small sets of integers. Let's do something more complicated. There is a library in NumPy that can create random arrays. Here's how we create an array of 20 random numbers.

```
In [13]: from RandomArray import *

In [14]: x = random((20,))

In [15]: x[0]
Out[15]: 0.0037765550990622415

In [16]: x[1]
Out[16]: 0.55916408054947242

In [17]: x[19]
Out[17]: 0.36556442411289364
```

Let's make it larger – 1000. With 1000 values, we can start to see the shape of the distribution from the histogram (Figure 5.3). All values between 0.0 and 1.0 are equally likely. The distribution has a few bumps, but overall, it's flat.

```
In [12]: x=random((1000,))

In [13]: hist(x)
Out[13]:
([ 90, 96,126,107, 94, 95, 77,114,111, 90,],
 [ 2.08577149e-004, 1.00040727e-001, 1.99872877e-001, 2.99705027e-001,
       3.99537176e-001, 4.99369326e-001, 5.99201476e-001, 6.99033626e-001,
       7.98865776e-001, 8.98697926e-001,],
 <a list of 10 Patch objects>)

In [14]: savefig("uniform_hist.eps")
```

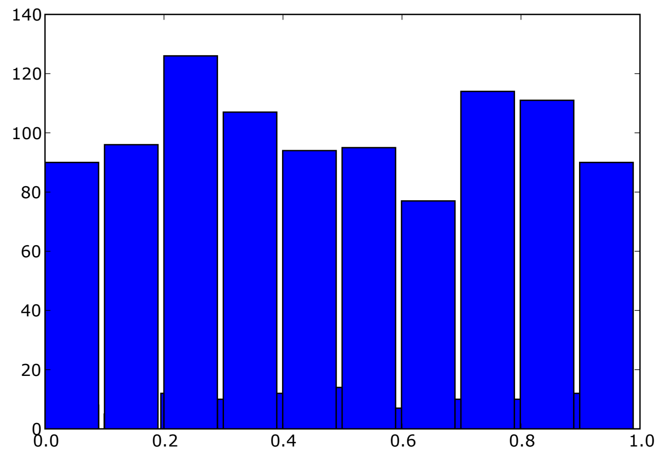


Figure 5.3: Uniform distribution

Okay – now x is 1000 random values. Let's make y two times each value of x. Can correlation still tell that these are similar values?

```
In [16]: y = x *2

In [17]: correlation(x,y)
Out[17]: 1.0
```

Yes, a correlation statistic can still note a strong relationship.

But those are uniform values. Most values in the real world are *normal*–they have a mean value that is really common, and other values are much less common. We can generate that from `RandomArray`, too, using the standard_normal function which assumes a mean of 0.0 and a standard deviation of 1.0.

```
In [19]: x = standard_normal((1000,))

In [20]: hist(x)
Out[20]:
([  7, 24, 56,143,210,255,180, 98, 22,  5,],
 [-3.13742447,-2.52991383,-1.92240319,-1.31489255,-0.70738192,-0.09987128,
       0.50763936, 1.11515   , 1.72266064, 2.33017128,],
 <a list of 10 Patch objects>)

In [21]: savefig("normal_hist.eps")
```

Now, if we look at the distribution of these 1000 values, we see a very different shape. There's clearly a bump in the middle at the average, and other values are less common (Figure 5.4).

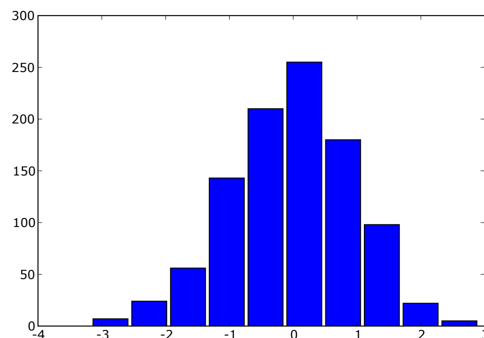


Figure 5.4: Normal distribution

Cool – so now let's see if correlation can help us identify a relationship with normal values.

```
In [23]: y = 2 * x

In [24]: correlation(x,y)
Out[24]: 1.0
```

Yup – it still saw the strong positive correlation.

Let's try one last trick. Basically, what we have been exploring is where the y values are a simple factor (multiplication by two) from x. What if there is some other factor at play? What if the y values *remembered* the last value of y? y_0 is just $2x_0$. But y_1 is $2x_1 + y_1$, and y_2 is $2x_2 + y_1$. If there's any other factor at play, even a very simple model of *memory*, can correlation detect that?

```
In [25]: memory = 0

In [26]: for i in range(0,1000):
   ....:     y[i] = (2 * x[i]) + memory
   ....:     memory = y[i]
   ....:

In [27]: correlation(x,y)
Out[27]: 0.035369277425458617
```

The answer seems to be *no*. Correlation no longer returns a strong r value, even when the y values are completely defined by the x values. This points out the weakness of correlation–which is also a strength. Correlation can't pick out complex relationships. It has to be a simple linear relationship or correlation won't see the relationship. On the other hand, if you *do* find a significant r, you can be pretty darn sure that you do have two related data sets. If you don't find a significant r, it doesn't mean that the values aren't somehow related–it just means that any relationship that's there is more complex than simply linear.

6 Text Analysis

One of the techniques used in *Freakonomics* [Levitt and Dubner, 2005] that is unusual for economists (and other social scientists, for that matter) is *textual analysis*. Levitt and Dubner analyze text strings of answers from students' standardized exams to find cheating teachers in one chapter, and they analyze baby names much later in the book. Computers are absolutely fantastic at textual analysis. We'll only use a couple of techniques in this chapter (enough to do some of what's in *Freakonomics*), but as you'll see, it's amazingly easy to do some really interesting textual analysis.

6.1 Visualizing textual differences: Bacon v. Shakespeare

As you may know, scholars for years have questioned whether Shakespeare really did write the works that he is said to have authored. (See `http://www.urbana.k12.oh.us/699/oh/authorship\%20controversy.html` for a nice summary of the controversy.) Here he was, the son of two illiterate parents with little formal education. Who would believe him to be the greatest English playwright?

One of the earliest authors thought to have penned Shakespeare's works was Francis Bacon. Bacon was a much more likely candidate–well-educated, well-spoken, a well-known writer.

We're not going to come up with anything novel that others haven't tried, but it's a fun context for trying out some interesting techniques. Our strategy will be to find something that correlates highly between two Bacon texts and between two Shakespeare texts, and *then* correlates highly between the Bacon and Shakespeare texts but not other authors' texts. Fortunately, Project Gutenberg[1] has tons of free books on-line. I grabbed *The Advancement of Learning* by Francis Bacon and *The Essays of Francis Bacon*, and then I grabbed *Macbeth* and *Romeo and Juliet*.

[1] `http://www.gutenberg.org`

Visualizing the 'the'

Here's a stupid but fun hypothesis: Authors use a similar number of the instances of the word 'the' in a similar way. It's silly, but easy to check. We're going to start checking by simply visualizing the 'the's.

Reading the file is easy–we use open, read, close. The find method will find a given string. Even more powerful is replace that will replace all instances of one string with another one.

```
In [31]: file=open("essays-bacon.txt","rt")

In [32]: text=file.read()

In [33]: file.close()

In [34]: pat=" the "

In [35]: text.find(pat)
Out[35]: 137

In [36]: text[125:145]
Out[36]: 'ing all over the wor'

In [37]: text.replace(pat,"*"+pat+"*")
Out[37]:

In [38]: text[125:150] #NO CHANGE!
Out[38]: 'ing all over the world, b'

In [42]: "ababab".replace("a","z") #RETURNS the change
Out[42]: 'zbzbzb'

In [43]: newtext=text.replace(pat,"*"+pat+"*")

In [44]: newtext[125:150]  #There's the change!
Out[44]: 'ing all over* the *world,'
```

So here's what we're going to do. We're going to create a file `viztext.py` and give it a highlight method that copies the text file to HTML. The HTML will reduce the font to its lowest possible size, and make the background black. Then we'll make the pattern (the word 'the') red to make it stand out. Visually, we'll be able to scan to see patterns of the word pattern we care about.

```python
def highlight(basename,pattern):
    file = open(basename+".txt","rt")
    text=file.read()
```

```
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="white">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern,newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

At first I tried it against a white background (Figure 6.1), but found that it wasn't very powerful. The red text pattern didn't stand out as much as against a black background (Figure 6.2).

*****The Project Gutenberg Etext of Essays of Francis Bacon***** #1 in our series by Francis Bacon Copyright laws are changing all over the world, be sure to check the copyright laws for your country before posting these files!! Please take a look at the important information in this header. We encourage you to keep this file on your own disk, keeping an electronic path open for the next readers. Do not remove this. **Welcome To The World of Free Plain Vanilla Electronic Texts** **Etexts Readable By Both Humans and By Computers, Since 1971** *These Etexts Prepared By Hundreds of Volunteers and Donations* Information on contacting Project Gutenberg to get Etexts, and further information is included below. We need your donations. Essays by Francis Bacon June, 1996 [Etext #575] [Most recently updated September 1, 2003] *****The Project Gutenberg Etext of Essays of Francis Bacon***** *****This file should be named ebacn10.txt or ebacn10.zip****** Corrected EDITIONS of our etexts get a new NUMBER, ebacn11.txt. VERSIONS based on separate sources get new LETTER, ebacn10a.txt. This etext was created by Judith Boss, Omaha, Nebraska. The equipment: an IBM-compatible 486/50, a Hewlett-Packard ScanJet IIc flatbed scanner, and Calera Recognition Systems' M/600 Series Professional OCR software and RISC accelerator board donated by Calera Recognition Systems. We are now trying to release all our books one month in advance of the official release dates, for time for better editing. Please note: neither this list nor its contents are final till midnight of the last day of the month of any such announcement. The official release date of all Project Gutenberg Etexts is at Midnight, Central Time, of the last day of the stated month. A preliminary version may often be posted for suggestion, comment and editing by those who wish to do so. To be sure you have an up to date first edition [xxxxx10x.xxx] please check file sizes in the first week of the next month. Since our ftp program has a bug in it that scrambles the date [tried to fix and failed] a look at the file size will have to do, but we will try to see a new copy has at least one byte more or less. Information about Project Gutenberg (one page) We produce about two million dollars for each hour we work. The fifty hours is one conservative estimate for how long it we take to get any etext selected, entered, proofread, edited, copyright searched and analyzed, the copyright letters written, etc. This projected audience is one hundred million readers. If our value per text is nominally estimated at one dollar then we produce $2 million dollars per hour this year as we release thirty-two text files per month: or 400 more Etexts in 1996 for a total of 800. If these reach just 10% of the computerized population, then the total should reach 80 billion Etexts. The Goal of Project Gutenberg is to Give Away One Trillion Etext Files by the December 31, 2001. [10,000 x 100,000,000=Trillion] This is ten thousand titles each to one hundred million readers, which is only 10% of the present number of computer users. 2001 should have at least twice as many computer users as that, so it will require us reaching less than 5% of the users in 2001. We need your donations more than ever! All donations should be made to "Project Gutenberg/IBC", and are tax deductible to the extent allowable by law ("IBC" is Illinois Benedictine College). (Subscriptions to our paper newsletter go to IBC, too) For these and other matters, please mail to: Project Gutenberg P. O. Box 2782 Champaign, IL 61825 When all other email fails try our Executive Director: Michael S. Hart We would prefer to send you this information by email (Internet, Bitnet, Compuserve, ATTMAIL or MCImail). ****** If you have an FTP program (or emulator), please FTP directly to the Project Gutenberg archives: [Mac users, do NOT point and click. . .type] ftp uiarchive.cso.uiuc.edu login: anonymous password: your@login cd etext/etext90 through /etext96 or cd etext/articles [get suggest gut for more information] dir [to see files] get or mget [to get files. . .set bin for zip files] GET INDEX?00.GUT for a list of books and GET NEW GUT for general information and MGET GUT* for newsletters. **Information prepared by the Project Gutenberg legal advisor** (Three Pages) ***START**THE SMALL PRINT!**FOR PUBLIC DOMAIN ETEXTS**START*** Why is this "Small Print!" statement here? You know: lawyers. They tell us you might sue us if there is something wrong with your copy of this etext, even if you got it for free from someone other than us, and even if what's wrong is not our fault. So, among other things, this "Small Print!" statement disclaims most of our liability to you. It also tells you how you can distribute copies of this etext if you want to. *BEFORE!* YOU USE OR READ THIS ETEXT By using or reading any part of this PROJECT GUTENBERG-tm etext, you indicate that you understand, agree to and accept this "Small Print!" statement. If you do not, you can receive a refund of the money (if any) you paid for this etext by sending a request within 30 days of receiving it to the person you got it from. If you received this etext on a physical medium (such as a

Figure 6.1: Francis Bacon's essays with white background, 'the' highlighted

```
In [53]: reload(viztext)
Out[53]: <module 'viztext' from 'viztext.py'>

In [54]: viztext.highlight("essays-bacon"," the ")
```

```
def highlight(basename,pattern):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern,newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

The obvious next thing to do is to process both the *Essays* and *Macbeth* to compare the visualizations (Figure 6.3). As one might anticipate for a fairly simple and stupid hypothesis – I don't see anything there, do you?

```
In [54]: viztext.highlight("essays-bacon"," the ")

In [55]: viztext.highlight("macbeth-shakespeare"," the ")
```

Visualizing the Proper Nouns: Using Regular Expressions

So let's consider a different hypothesis: That a unique characteristic of an author is the number of capitalized words that they use. That indicates the number of sentences, but also indicates the number of proper nouns (e.g., names of things) that are used. Perhaps that's a determining factor?

To locate capitalized words, we need something a bit stronger than a text pattern to search for. Computer scientists use *regular expressions* to describe patterns of words, not just specific words. This allows for significant flexibility in exploring text. Think of regular expressions as being like mathematical expressions, in that there are constants and operators, but we're describing patterns of letters, not patterns of numbers.

The name of the package that knows about regular expressions in Python is re. You **import** it to use it.

```
In [3]: import re

In [4]: string = "This is my name: Mark Guzdial.  I live in Decatur."
```

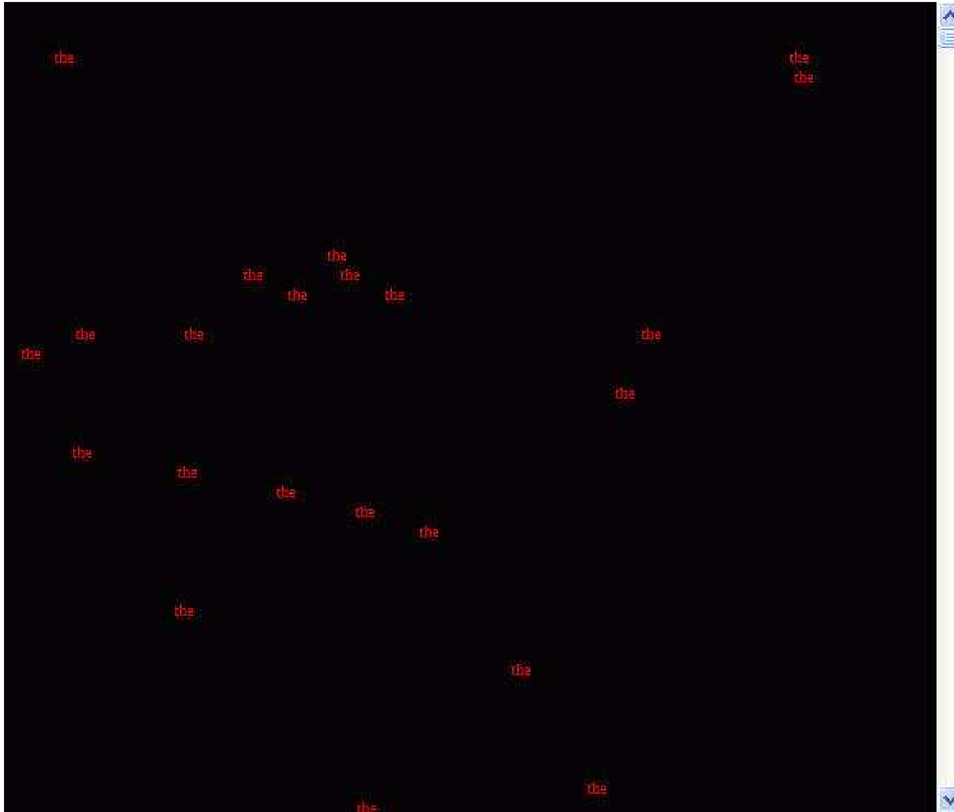


Figure 6.2: Francis Bacon's essays with black background, 'the' highlighted

The match method takes a regular expression and a string as input, then returns a *match object* that describes the match, or literally None if no match is found. The most common uses of the match object is to get the start position and the end position of the match.

Rules for matches

Here are how regular expressions are constructed.

- Any regular character matches only the same regular character. 'M' matches only to 'M', and 'a' matches only to the letter 'a'.

```
In [5]: matchobject=re.search("Mark",string)

In [6]: print matchobject.start()
```

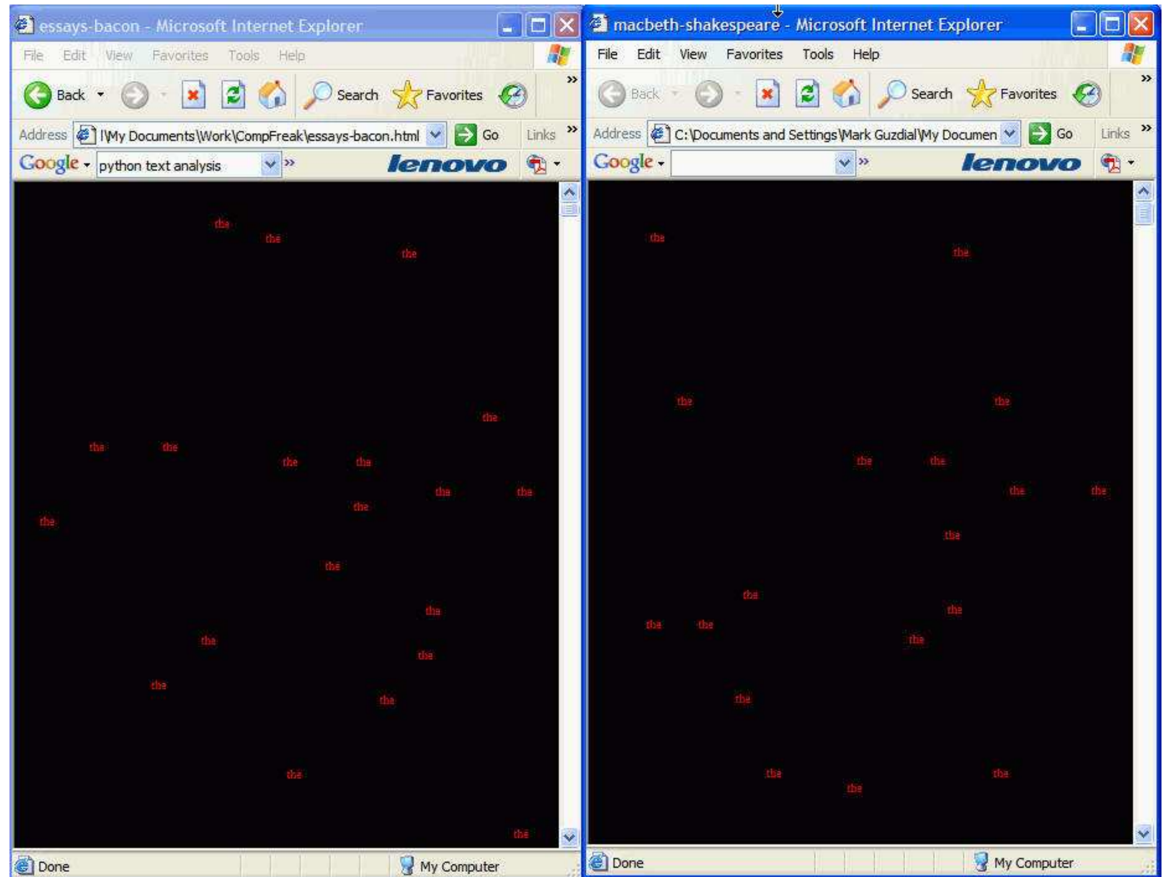


Figure 6.3: Comparing 'the' patterns in Bacon's *Essays* and Shakespeare's *Macbeth*

```
17

In [7]: print matchobject.end()
21

In [8]: string[17:21]
Out[8]: 'Mark'
```

- A period matches anything.
- A $*$ says "repeat zero or more times whatever came before me." The below example looks for a match that starts with a lowercase 'm'

and then is followed by any number of characters–which, of course, matches everything else in the string.

```
In [9]: matchobject=re.search("m.*",string)

In [10]: print string[matchobject.start():matchobject.end()]
my name: Mark Guzdial.  I live in Decatur.
```

Again, if the match doesn't work, you get a None that doesn't understand start nor end

```
In [11]: matchobject=re.search("m.*\b",string)

In [12]: print string[matchobject.start():matchobject.end()]
---------------------------------------------------------------------------
exceptions.AttributeError                            Traceback (most recent call
 last)

C:\Documents and Settings\Mark Guzdial\My Documents\Work\CompFreak\<console>

AttributeError: 'NoneType' object has no attribute 'start'
```

- Putting an "r" before a string treats it as raw mode and backslashes don't get interpreted by Python. A b is supposed to find word boundaries.

```
In [13]: matchobject=re.search(r"m.*\b",string)

In [14]: print matchobject
<_sre.SRE_Match object at 0x00CC7218>

In [15]: print string[matchobject.start():matchobject.end()]
my name: Mark Guzdial.  I live in Decatur
```

- The code S means "anything that isn't *whitespace* (tab, return, space." The s means whitespace. So the below code looks for a word that starts with "m" and has any number of characters before a whitespace character.

```
In [18]: matchobject=re.search(r"m\S*\s",string)

In [19]: print string[matchobject.start():matchobject.end()]
my
```

- Character classes are in square brackets. [A–Z] only matches a single uppercase character. [a–zA–Z] matches any character of any case. The below test looks for an uppercase character at the beginning of a word, followed by any number of any kinds of letters, but only letters.

  ```
  In [3]: match = re.search(r"\b[A-Z][a-zA-Z]*",string)

  In [4]: print match.start()
  0

  In [5]: print match.end()
  4

  In [6]: print string[match.start():match.end()]
  This
  ```

- The character + is like *, but + insists that there must be at least one of what it matches.

There are many other parts of regular expressions, but that's enough to get started.

Finding Capitalized Words

There are a couple more methods besides search that can be useful in regular expression processing. The first is called split. Given a regular expression and a string, it returns a sequence of substrings of everything that does *not* match the pattern.

```
In [13]: chopped = re.split(r"\b[A-Z][a-zA-Z]*",string)

In [14]: chopped[0]
Out[14]: ''

In [15]: chopped[1]
Out[15]: ' is my name: '

In [16]: chopped[2]
Out[16]: ' '

In [17]: chopped[3]
Out[17]: '. '
```

Any regular expression in parentheses is *grouped*. We can later refer to those groupings as 1, 2, and so on here. We can use that with sub which substitutes one pattern with another in a given string.

Here, we use our capitalization pattern to wrap red font coloring around capitalized words.

```
In [24]: newtext = re.sub(r"(\b[A-Z][a-zA-Z]*)",r'<font color=red>\1</font>',str
ing)
In [25]: newtext
Out[25]: '<font color=red>This</font> is my name: <font color=red>Mark</font> <f
ont color=red>Guzdial</font>. <font color=red>I</font> live in <font color=red>D
ecatur</font>.'
```

Using our newfound capability to replace patterns, we can expand our `viztext.py` tool to highlight capitals. (Note the **import** re at the top.)

```
import re

def highlight(basename,pattern):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern,newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()

def highlightCapitals(basename):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Now make the new one
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=re.sub(r"(\b[A-Z][a-zA-Z]*)",r'<font color="red">\1</font>',text)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

It's pretty easy to use, and the result is more interesting than all the 'the's (Figure 6.4).

```
In [26]: import viztext

In [27]: viztext.highlightCapitals("essays-bacon")
```

Figure 6.4: Visualization of all capitalized letters in the *Essays of Francis Bacon*

6.2 Counting Text Patterns

While the visualizations are fun, I doubt that we're going to see any particularly interesting patterns that way. We're going to need to leverage some numeric capability.

Counting our 'the's

The `split` method that we saw earlier for regular expressions also exists for normal strings, and is pretty darn useful. It breaks up a string into a sequence of substrings, using the provided character as the separator. The below example looks for spaces to break up words.

```
In [31]: string Out[31]: 'This is my name: Mark Guzdial. I live in
Decatur.'

In [32]: string.split(r' ')
Out[32]: ['This',
 'is',
 'my',
 'name:',
 'Mark',
 'Guzdial.',
 'I',
 'live',
 'in',
 'Decatur.']
```

There are 10 strings in the output sequence. That means that there were 9 spaces in the original string.

Let's use this to split the text into paragraphs, by looking for two returns (string 'n') in a row. Then, let's count the "the"s in each paragraph. Fortunately, there's a great string method count that does just that.

```
def countText(basename,pattern):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n') #Two returns = paragraph
    # Now, count the number of 'the's in the paragraph
    ret = []
    for s in newtext:
        ret.append(s.count(pattern))
    return ret
```

Now, let's try it.

```
In [44]: import counttext

In [45]: essaysThe = counttext.countText("essays-bacon"," the ")

In [46]: len(essaysThe)
Out[46]: 508

In [47]: essaysThe[0:10] #What do the answers look like?
Out[47]: [0, 1, 2, 0, 0, 0, 0, 0, 0, 0]

In [48]: advThe = counttext.countText
    ("advancement-learning-bacon"," the ")

In [50]: romeoThe = counttext.countText
    ("romeo-juliet-shakespeare"," the ")

In [51]: macbethThe = counttext.countText
    ("macbeth-shakespeare"," the ")

In [52]: len(romeoThe)
Out[52]: 287

In [53]: len(macbethThe)
Out[53]: 271

In [54]: len(advThe)
Out[54]: 597
```

```
In [55]: len(essaysThe)
Out[55]: 508
```

Unfortunately, there are different number of paragraphs in each sample text. So, we'll do a correlation just the first 200 paragraphs. The answer is pretty abysmal. Counting the "the"s doesn't seem to be a useful metric.

```
In [56]: from correlation import *

In [57]: correlation(romeoThe[0:200],macbethThe[0:200])
Out[57]: 0.039670370779755854

In [58]: correlation(advThe[0:200],essaysThe[0:200])
Out[58]: -0.0085796837192241779
```

Counting Capitals

Let's try our second hypothesis, counting the number of capitalized letters. To get the number of capital words in each paragraph, we'll generate the re.split of each paragraph, then count the number of match objects returned. One less than that will be the number of capitals.

```
def countCapitals(basename):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n')
    # Count the capitals
    ret = []
    for para in newtext:
        match = re.split(r"\b[A-Z][a-zA-z]*",para)
        ret.append(len(match)-1)
    return ret
```

Now let's try it.

```
In [61]: advCap = counttext.countCapitals("essays-bacon")

In [62]: len(advCap)
Out[62]: 508

In [63]: advCap[0:10]
Out[63]: [9, 1, 3, 9, 8, 7, 5, 1, 2, 4]

In [64]: essaysCap = counttext.countCapitals("advancement-learning-bacon")
```

```
In [65]: romeoCap = counttext.countCapitals("romeo-juliet-shakespeare")

In [66]: macbethCap = counttext.countCapitals("macbeth-shakespeare")

In [67]: correlation(advCap[0:200],essaysCap[0:200])
Out[67]: -0.10076023917690945

In [68]: correlation(romeoCap[0:200],macbethCap[0:200])
Out[68]: 0.016919364776092155
```

Eww – that correlation isn't very good either. Good thing we're not Shakespearean scholars. . .

7 Inferential Statistics and Hypothesis Testing

In our text analysis, we ran against the problem of having to compare sequences of numbers that weren't paired. Correlations were really the wrong things to use there. We had no reason to believe that paragraph-by-paragraph, our metrics (counting "the"s and capitalized words) would change in-step.

What we really wanted to ask was if the sets were *different*, not in lock-step. What do we mean by different? Well, are there *means* different? Are the averages of each set significantly different? That's what we're going to test in this chapter.

7.1 Inferential Statistics

The important thing to remember is that *inferential statistics* constitutes a set of tools for inferring something from a particular sample to larger populations. Basically we are asking how the statistics of a sample (that is, the mean and standard deviation) match the parameters (the actual mean and standard deviation) of the entire population. The purpose of inferential statistics is to estimate these population parameters. To understand the logic of inferential statistics, we need to consider the topic of probability.

Probability

When we do an experiment, we know that there is a possibility that the results of the experiment could have come out differently if we had performed the same experiment several times. Maybe when we took a sample, we happened to get just the weird ones. Maybe when we ran the experiment, there was a cold front coming in which screwed up the experiment. How would we know?

How do we assess the possibility that we simply grabbed an oddball sample? In empirical research, we usually do this by assuming that the study was run many times and try to guess how the results would have

come out each time. This helps us to understand the results of a particular study.

In a study looking at children's facial expressions, the researcher[Cole, 1986], found that preschool girls aged 3 to 4 displayed more negative facial expressions when the experimenter was no present than when the experimenter was present. Included in Cole's report were the means and standard deviations for the frequencies of various types of facial expressions as well as statistic statements that compared one type of expression with another (e.g., positive versus negative emotions) in various conditions (e.g., with and without the experimenter present). Included in these statistical statements were probability estimates of the form $p < .001$.

The statement $p < .001$ means that if a large number of similar experiments were conducted, we would expect to replicate these results by chance alone less than 1 time in 1000. That is, we would expect, without using the experimental manipulation, to get these results 1 time in 1000. In making such a statement we create a degree of certainty on which to rest our conclusions concerning particular experimental results.

Because an understanding of probability is at the heart of all of this, it is important to have some understanding of its origins and theoretical basis. Questions of probability date back thousands of years. As you might guess, much of what we know about probability came about because some people wanted to make games of chance less chancy for themselves. Roman author Cicero (106–43 BC) tried to understand how one should interpret the event in which four dice were thrown and all came up the same. Was this an act of the gods or a rare event of chance?

It might also interest you to know that making better beer was one of the practical problems that led to the development of one popular statistical technique, the calculation of *Student's t*. Actually it was William Gosset who wrote under the name "Student." He worked for the Guinness Brewery in Dublin and asked what could be inferred about an entire batch of beer from sampling a small portion of it. In other words, what is the probability that this particular sample is similar to the entire batch? To answer that question, you need to have some idea about what is an expected *distribution* of statistics of the population overall.

Normal Distribution

Suppose we flip 10 coins all at once. Each coin could land in one of two possible states, heads or tails[1]. There are 1024 (2^10) possible outcomes if we consider each coin unique, but we can collapse the outcomes into 11 outcome categories (by ignoring which *particular* coins are heads and which *particular* coins are tails). We can see this graphed (Figure 7.1) and see the probabilities for some of the outcomes (Figure 7.2), generated by

[1]We'll ignore the very rare probability of a coin falling on its edge

asking WolframAlpha[2] about “flip 10 coins”.

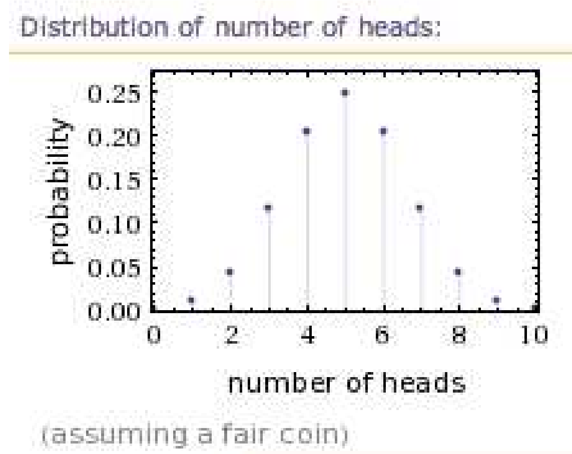


Figure 7.1: Probability of each outcome for flipping 10 coins (computed by Wolfram-Alpha)

Probability results:

all heads	0.09766%
all tails	0.09766%
5 heads and 5 tails	24.61%
at least one head	99.9%
at least one tail	99.9%

(assuming a fair coin)

Figure 7.2: Probabilities for flipping 10 coins (computed by Wolfram-Alpha)

Now, suppose we did this 1000 times. No, let’s simulate doing this 1000 times.

[2] `http://www.wolframalpha.com`

There is a pylab function named random which returns a random value between 0 and 1, all values equally likely.

```
>>> from pylab import *
>>> random()
0.1337721937015831
>>> random()
0.46023620061907944
>>> random()
0.1762070339523374
```

So we can call random(), and if we get less than 0.5, call that a 'head.' Otherwise, it's a 'tail.' We can do 10 of those, and count the number of heads. And we can do 1000 flips of 10 coins, and count how often that many heads comes up. Then generate a graph of all those values. Here's what we get (Figure 7.3):

```
from pylab import *

def flip():
    coin = random()
    if coin < 0.5:
        return 'heads'
    else:
        return 'tails'

def flip10():
    heads = 0
    for times in range(10):
        if flip() == 'heads':
            heads = heads + 1
    return heads


def times1000():
    # Create a dictionary to store our distribution
    distr = {}

    # 1000 times, flip 10 coins
    # Store the number of times THAT many heads comes up
    for times in range(1000):
        numheads = flip10()
        if numheads in distr:
            # If we've seen this count before, add 1
            distr[numheads] = distr[numheads] + 1
        else:
            # If not, it's our first one
            distr[numheads] = 1

    return distr
```

```
trials = times1000()
plot(trials.keys(), trials.values())
xlabel("Number of heads")
ylabel("How many times that many heads was seen")
show()
```

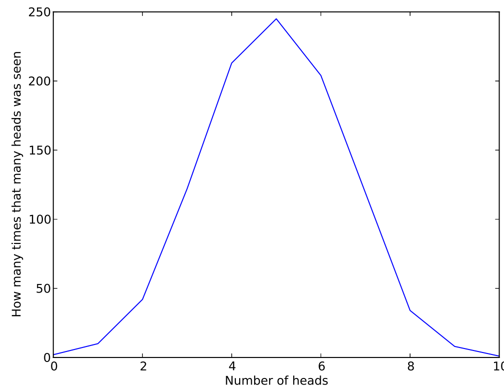


Figure 7.3: Graph of 1000 trials of flipping 10 coins

The curve that we see in Figure 7.3 approximates a *normal curve*. It's the bell-shaped curve that you so often see when looking at statistics like curves, grades, and SAT scores. It's the curve that results from a *normal distribution*.

Let's assume that SAT scores constitute a normal curve. You would know that about 68% of all students taking the SAT get a score that falls within 1 standard deviation (SD) in each direction from the mean. That is part of the definition of a normal curve. We could also say for any person randomly chosen that there is a 68% chance that he or she would have an SAT score within 1 SD of the mean. More than 95% of all students fall within 2 SDs of the mean (considering both directions).

Let's move now from talking about a distribution of individual scores to a distribution of means. Suppose we had available to us all the SAT scores for a given year (i.e., we had the population of scores). Keep in mind it is virtually *never* the case that we have the population of scores for the variables we wish to measure. Suppose we took a random sample of scores from that population and got the mean from that sample. Suppose we did this over and over again.

Now suppose we graph each sample mean that we have obtained. What do you think the resulting curve would look like? Most likely, it would be a normal curve. This finding is based on what is technically called the *central limit theorem* and it lies at the heart of inferential statistics. The

central limit theorem states that if a number of samples are drawn from a population at random, then the means of the samples tend to be normally distributed.

What do you think would happen if the size of each sample was large vs. *very* large? The larger the number of people in each sample, the less variability there will be in the means. If this were represented graphically, the bell-shaped curve would be skinny if the sample sizes were large but broad at the bottom if the sample sizes were small. This is because if you had only a few people in each of your many, many samples, an extreme score from any person would have more influence on the mean. Thus, when you plot your means, you would expect more variability in the scores than if you had many people in each of your many, many samples.

This is an important idea that lies at the heart of determining the probability levels of such statistics as the *t* test and *analysis of variance*. Since the central limit theorem tells us that the population's statistic tend to be normally distributed, our sample is drawn from a normal distribution.

Standard Error of the Mean

We can find the mean of all our sample means and the standard deviation (SD) of all our sample means. The SD of all our sample means has a special name: the *standard error of the mean*. The term *error* here means the difference between our estimate and the true value. We can't know the true value, but we can estimate how far off our sample is.

If we had just one sample of students who took the SAT, what would be the best guess regarding the mean of all students who took the SAT test? We would want to guess the most *common* value. Our best guess would be the *mean* of our sample. So if you ran the experiment several times (took several samples of SAT students), you would get different means from each sample, but all of those means would *likely* be **near** the actual population mean.

We can also estimate how far off we might be with this sample mean from where the population mean would be. To do this we estimate the standard error of the sample mean using the data from our sample. First, we compute the SD of our sample.

Standard Error of the Mean =

$$SE = \frac{SD}{\sqrt{number of values}}$$

As the number of values increases (so the size of the sample gets closer to the size of the population), the standard error gets smaller. Notice that the standard error of the mean is *smaller* than the standard deviation. This means that if you took a sample several times, and took the mean of each sample, the distribution of those means would be *smaller* than the standard deviation of the sample or the population.

For instance, if our sample of scores had a mean of 505 and a SD of 100, and we had data from 900 people, then:

$$SE = \frac{100}{\sqrt{900}} = \frac{100}{30} = 3.33$$

Because the central limit theorem says that the sampling distribution is normally distributed, we can now make the following claims:

- If we repeated our experiment multiple times (which for pragmatic reasons we usually can not do), about 68% of the means obtained would be within 1 SE of our obtained mean (505 3.33) and that about 95% of all sample means would be within 2 SEs (505 6.66)
- Or, to put it another way, in probability terms, that we are more than 95% confident that the population mean is between 498.34 (505-6.66) and 511.66 (505+6.66).

Figuring out Standard Error

Where did that SE formula come from? Let's say that we had a sample of numbers ($x_1 ... x_n$) from a population, a total of all those numbers ($Total_sample$). Imagine that we really could figure out the population's mean ($mean_pop$) and the population's standard deviation (SD_pop). We would expect that the variance in our sample total $Total_sample$ would be n times the population standard deviation squared ($(SD_pop)^2$). The expected variance of $Total_sample/n$ ($mean_sample$) is $\frac{(SD_pop)^2}{n}$, and the standard deviation of $Total_sample/n$ is $SD_pop/\sqrt{n}$.

Let's try to figure this out concretely. There is a random module as part of standard Python that knows how to take a sample. In the below example, I made up a set of integers as a "population" then took a set of three values as a sample. The values 2 and 3 showed up often, since they were doubled up in the population, but these are random selections of three values.

```
>>> from random import *
>>> population = [1,2,2,3,6,-1,3,4]
>>> sample(population,3)
[3, 2, 3]
>>> sample(population,3)
[3, 2, 2]
>>> sample(population,3)
[3, 2, 3]
>>> sample(population,3)
[2, 4, -1]
>>> sample(population,3)
[2, 2, 6]
```

Here is some code that:

- Defines mean,stdev, and stderr.

- Creates a population of 100 random numbers between 0 and 1000.
- Figures out the "unknowable" true population means and standard deviation.
- Then takes four samples, two of size 10 and two of size 20. We compute the mean, standard deviation, and standard error for each sample.
- We also predict the distribution of sample means.

```
import random
from pylab import sqrt

def mean(sequence):
    return (1.0*sum(sequence))/len(sequence)

def stdev(x,xbar):
    sums = 0
    for num in x:
        sums = sums + pow(xbar-num,2)
    return sqrt(sums/len(x))

def stderr(x):
    return stdev(x,mean(x))/sqrt(len(x))

#Our population will be a random 100 values from 0..1000
population = random.sample(range(1000),100)
print "Our unknowable actual population values:"
print "Mean of population",mean(population)
print "Standard deviation of population",stdev(population,mean(population))
print "*** Now, some samples:"
print "= Samples of 10 ="

for trial in range(2):
    somesample = random.sample(population,10)
    print "Small Trial #",trial+1
    print "Sample:",somesample
    trialmean = mean(somesample)
    trialSE = stderr(somesample)
    print "Mean:",trialmean
    print "Standard deviation",stdev(somesample,mean(somesample))
    print "Standard error",trialSE
    print "68% of means should be between:",trialmean-trialSE," and ",trialmean+tria
    print

print "= Samples of 20 ="
for trial in range(2):
    somesample = random.sample(population,20)
    print "Larger Trial #",trial+1
```

```
    print "Sample:",somesample
    trialmean = mean(somesample)
    trialSE = stderr(somesample)
    print "Mean:",trialmean
    print "Standard deviation",stdev(somesample,mean(somesample))
    print "Standard error",trialSE
    print "68% of means should be between:",trialmean-trialSE," and ",trialmean+trialSE
    print
```

It's as if we ran four experiments. You see in the execution below that the standard error decreases with the larger sample size. The means and standard deviations are closer to the population mean with the larger sample size.

```
Our unknowable actual population values:
Mean of population 482.5
Standard deviation of population 274.486338458
*** Now, some samples:
= Samples of 10 =
Small Trial # 1
Sample: [806, 483, 772, 895, 369, 687, 472, 187, 229, 830]
Mean: 573.0
Standard deviation 245.737258062
Standard error 77.7089441442
68% of means should be between: 495.291055856  and  650.708944144

Small Trial # 2
Sample: [142, 651, 883, 129, 165, 494, 466, 340, 439, 748]
Mean: 445.7
Standard deviation 246.992327816
Standard error 78.1058320486
68% of means should be between: 367.594167951  and  523.805832049

= Samples of 20 =
Larger Trial # 1
Sample: [91, 502, 806, 427, 687, 209, 722, 703, 466, 519, 682, 547, 472, 369, 593, 115, 147, 4
Mean: 512.1
Standard deviation 233.740219047
Standard error 52.2659018864
68% of means should be between: 459.834098114  and  564.365901886

Larger Trial # 2
Sample: [330, 602, 494, 382, 472, 707, 806, 883, 583, 370, 658, 659, 176, 333, 924, 136, 428,
Mean: 524.0
Standard deviation 241.536539679
Standard error 54.0092121772
68% of means should be between: 469.990787823  and  578.009212177
```

7.2 Hypothesis Testing

Just to remind you, when we are referring to the statistical term *population*, we mean we are talking about all possible individuals (or all possible samples). Suppose you claim that the students in your class scored higher on the SAT than students around the country. You are essentially claiming that the students in your class are a sample from a population (the population of high-scoring students) that is *different* from the population of all students taking the SAT.

First we form the *null hypothesis*. The null hypothesis states that there is just a single population and therefore the mean of your sample is the mean of the population. If the null hypothesis can not be rejected then we can not claim that your class comes from a different population (e.g., the population of high scoring students).

On the other hand, what if we *can* reject the null hypothesis? That is, we support the *alternative hypothesis*. If the null hypothesis can be rejected, then we make a statistical statement concerning the null hypothesis, namely that if both means come from the same population, then the probability of chance explaining the difference between the sample and the population means is quite low.

Assume the mean of your class was 750 on the math SAT while the mean of all people taking the math SAT is 500. The probability of finding a mean of 750 in a random sample drawn from the population of all people taking the test would be low, but it would be possible. We might say that the sample mean of 750 cold have been drawn from a population with a mean of 500 less than 1 time in 1000. Thus, if we found a sample mean of 750 we would reject the null hypothesis in favor of the alternative hypothesis.

Typically we do not have a population mean to compare to a sample mean. What we usually have are two (or more) sample means to compare to each other. Each sample mean is used to estimate the mean of a population. The question is whether the two means represent the same population or different populations. So, when we use inferential statistics, we are not asking whether Group 1 is different from Group 2 but whether the two groups come from the same population.

T Test

If we have two samples (think $x_1 ... x_n$ and $y_1 ... y_n$), that may or may not come from the same population, how can you tell? You can never know *for sure*, but you can run a statistical test called a *Student t test* that will tell you the probability that these two samples *are* from the same population, but you got a bad set of samples. Maybe you happened to get samples from two different ends of the distribution, so the means and standard deviations *look* very different, but they're not really. You can't know that for sure, but you can figure out the probability that you got bad data. If

that probability is small enough, we believe that they probably are from different populations.

Let's do an example of a *t* test. A car manufacturer that makes a car called the Jupiter just came out with a new model, the Jupiter XL. Some of the modifications made to the car are expected to improve the mpg (miles per gallon) rating of the car while other modifications are not. The manufacturer has hired your firm, an independent consumer research firm, to test the new model. To determine if there is any difference between the mpg rating of the old and new models, you collect a random sample of 5 cars of the old model and 6 cars of the new model. You drive the cars along the same city route and record the average mpg rating of each car. Here are the data:

Old Model		New Model	
Car	*MPG*	*Car*	*MPG*
1	30	1	37
2	34	2	36
3	34	3	40
4	29	4	36
5	33	5	34
		6	33

Step 1: Formulate the hypotheses

When you run a t test, you are comparing two hypotheses. The symbol μ represents the true mean of the population.

- $H_0 : \mu_1 = \mu_2$ This hypothesis states that the true mean MPG rating of the old model is equal to the true mean MPG rating of the new model.

- $H_1 : \mu_1 \neq \mu_2$ This hypothesis states that the true mean MPG rating of the old model is **not equal** to the true mean MPG rating of the new model.

The point of the t test is to choose between these.

Calculate the test statistic

Before you can calculate the t test, you have to calculate the following values from the sample data.

Compute the Sample Mean for Group 1 (old model)

$$\bar{x}_1 = \text{sample mean for group 1} = \frac{\text{sum of scores of group 1}}{\text{number of scores in group 1}} = \frac{30+34+34+29+33}{5} = \frac{160}{5} = 32$$

Compute the Sample Mean for Group 2 (new model)

$$\bar{x}_2 = \text{sample mean for group 2} = \frac{\text{sum of scores of group 2}}{\text{number of scores in group 2}} \frac{37+36+40+36+34+33}{6} = \frac{216}{6} = 36$$

Compute the Sample Variance for Group 1 (old model)

Notice that the divisor is $n-1$ because we are estimating a *population statistic* rather than calculating something about the *sample*.

$$s_1^2 = \text{sample variance for group 1} = \frac{(\text{1st score - mean})^2 + (\text{2nd score - mean})^2 + \cdots + (\text{last score - mean})^2}{(\text{number of scores in group 1}) - 1}$$

$$= \frac{(30-32)^2 + (34-32)^2 + (34-32)^2 + (29-32)^2 + (33-32)^2}{5-1} = \frac{22}{4} = 5.5$$

Compute the Sample Variance for Group 2 (new model)

$$s_2^2 = \text{sample variance for group 2} = \frac{(\text{1st score - mean})^2 + (\text{2nd score - mean})^2 + \cdots + (\text{last score - mean})^2}{(\text{number of scores in group 2}) - 1}$$

$$= \frac{(37-36)^2 + (36-36)^2 + (40-36)^2 + (36-36)^2 + (34-36)^2 + (33-36)^2}{6-1} = \frac{30}{5} = 6$$

Compute the Pooled Sample Variance

$$s_p^2 = \frac{(\text{number of scores in group 1} - 1)s_1^2 + (\text{number of scores in group 2} - 1)s_2^2}{(\text{number of scores in group 1} + \text{number of scores in group 2}) - 2}$$

$$= \frac{(5-1)5.5 + (6-1)6}{5+6-2} = \frac{52}{9} = 5.78$$

Compute the t statistic

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s_p^2\left(\frac{1}{\text{number of scores in group 1}} + \frac{1}{\text{number of scores in group 2}}\right)}}$$

$$= \frac{32-36}{\sqrt{5.78\left(\frac{1}{5}+\frac{1}{6}\right)}} = \frac{-4}{1.46} = -2.74$$

Now that we have a value of -2.74, how do we figure out whether it represents a significant difference between the groups (by which we really mean, are the groups drawn from different populations)? In order to do this, we have to consider the issue of *degrees of freedom*. Degrees of freedom refers to the number of scores that are free to vary when you calculate something.

As a example, imagine that you and two friends are eating out and you are waiting for your order. One of you had ordered chicken salad on rye, another has ordered a yogurt and fruit cup, and the third has ordered pastrami on white bread. The waiter comes with the food but he has forgotten who ordered what. From his standpoint he has a number of degrees of freedom in that he can place the dishes in any of several different combinations. However, if you were to remind him that you ordered chicken salad and the friend of your right the yogurt and fruit cup, this would limit

his degrees of freedom. Once you had fixed in place the order for two of the three selections, the third was determined and not free to vary.

The same is true with numbers. What is someone asked you to pick three numbers that added up to 20 and each number had to be 9 or less. You could pick any number you wanted for the first number; say you pick 7. You could also pick any number you wished for the second; say you pick 8. However, once you picked the first two numbers, the third number could be only one number; in this example it must be 5. Thus, the general idea in degrees of freedom reflects how many scores are free to vary.

So, getting back to the t test, once we have calculated the t statistic, the next step is to look up this value of t in a *t table*. The degrees of freedom in our experiment is the (n of Group 1) plus (the n of Group 2) - 1. So, $5 + 6 - 1 = 9$.

One Sided	**75%**	**80%**	**85%**	**90%**	**95%**	**97.5%**	**99%**	**99.5%**	**99.75%**	**99.9%**	**99.95%**
Two Sided	**50%**	**60%**	**70%**	**80%**	**90%**	**95%**	**98%**	**99%**	**99.5%**	**99.8%**	**99.9%**
1	1.000	1.376	1.963	3.078	6.314	12.71	31.82	63.66	127.3	318.3	636.6
2	0.816	1.061	1.386	1.886	2.920	4.303	6.965	9.925	14.09	22.33	31.60
3	0.765	0.978	1.250	1.638	2.353	3.182	4.541	5.841	7.453	10.21	12.92
4	0.741	0.941	1.190	1.533	2.132	2.776	3.747	4.604	5.598	7.173	8.610
5	0.727	0.920	1.156	1.476	2.015	2.571	3.365	4.032	4.773	5.893	6.869
6	0.718	0.906	1.134	1.440	1.943	2.447	3.143	3.707	4.317	5.208	5.959
7	0.711	0.896	1.119	1.415	1.895	2.365	2.998	3.499	4.029	4.785	5.408
8	0.706	0.889	1.108	1.397	1.860	2.306	2.896	3.355	3.833	4.501	5.041
9	0.703	0.883	1.100	1.383	1.833	2.262	2.821	3.250	3.690	4.297	4.781
10	0.700	0.879	1.093	1.372	1.812	2.228	2.764	3.169	3.581	4.144	4.587
11	0.697	0.876	1.088	1.363	1.796	2.201	2.718	3.106	3.497	4.025	4.437
12	0.695	0.873	1.083	1.356	1.782	2.179	2.681	3.055	3.428	3.930	4.318
13	0.694	0.870	1.079	1.350	1.771	2.160	2.650	3.012	3.372	3.852	4.221
14	0.692	0.868	1.076	1.345	1.761	2.145	2.624	2.977	3.326	3.787	4.140
15	0.691	0.866	1.074	1.341	1.753	2.131	2.602	2.947	3.286	3.733	4.073

Figure 7.4: t table from Wikipedia

Now we use a *t table* – see Figure 7.4[3].The degrees of freedom are listed down the side of this table; across the top are listed the various probability (p) levels for one- and two-tailed tests.

- A one-tailed test refers to a directional prediction, e.g., the MPG of Group 2 (the new model cars) is *greater than* the MPG of Group 2 (the old model cars).

[3]Copied from `http://en.wikipedia.org/wiki/Student's_t-distribution`.

- A two-tailed test refers to a non-directional prediction, e.g., the MPG of Groups 1 and 2 are *different*. The requirement for a two-tailed test is roughly half the requirement for a one-tailed test.

We are making a non-directional prediction, so we look for 2.74 (ignore the sign) next to 9 degrees of freedom. We see that the largest value for which our t statistic is larger than is 2.262. Then we look up to the probability level. So, we can say that our result is significant at the $.05$ level. By convention, a probability of $.05$ or less is considered statistically significant. In formal notation we would write $t(9) = -2.74, p < .05$. This means that if the car experiment were conducted a large number of times, we would expect this result to occur by chance fewer than 5 in 100 times. We say that we "reject the null (H_0) hypothesis."

7.3 Computing a Context: Elections and Unemployment Rates

One of the claims of political pundits is that what the US people are voting on in a presidential election is whether they're doing better or worse than they were four years previously. Let's test that.

- In 1996, Clinton was re-elected over Dole – the American people chose to stick with their party.
- In 2000, Bush won over Gore (even though Gore won the popular vote). The American people *switched* parties. Were they better off than they were 4 years previously?
- In 2004, Bush won over Kerry. The American people *stuck with* the Republican party. Were they about the same as they were four years previously?

I downloaded a data set from the US Bureau of Labor Statistics of US Unemployment Data (one measure of "doing better") over many years.

```
In [70]: import csvfile

In [71]: file = csvfile.CSVfile("USUnemploymentRate.csv"

In [72]: file.next()
Out[72]:
{'Annual': '',
 'Apr': '3.9',
 'Aug': '3.9',
 'Dec': '4',
 'Feb': '3.8',
 'Jan': '3.4',
 'Jul': '3.6',
```

```
 'Jun': '3.6',
 'Mar': '4',
 'May': '3.5',
 'Nov': '3.8',
 'Oct': '3.7',
 'Sep': '3.8',
 'Year': '1948'}

In [73]: file.next()
Out[73]:
{'Annual': '',
 'Apr': '5.3',
 'Aug': '6.8',
 'Dec': '6.6',
 'Feb': '4.7',
 'Jan': '4.3',
 'Jul': '6.7',
 'Jun': '6.2',
 'Mar': '5',
 'May': '6.1',
 'Nov': '6.4',
 'Oct': '7.9',
 'Sep': '6.6',
 'Year': '1949'}
```

Let's use these data to compare 1996, 2000, and 2004. Were they really different? Does the direction of difference match the election results?

7.4 Computing a t test

A *t test* tells us whether the averages of two sets are significantly different or not. The hypothesis we're testing is whether they're the same (H_0), or different (H_1).

Here's the process for a t test computation.

- We need the averages ($\bar{x}_1$ and $\bar{x}_2$) of each group. We know how to do that already.

- We need the variance (s_1^2 and s_2^2)of each group. We can chop that out of our standard deviation function that we created earlier.

- We need the *pooled sample variance*. This is:

$$s_p^2 = \frac{(N_{group1} - 1)s_1^2 + (N_{group2} - 1)s_2^2}{(N_1 + N_2) - 2} \tag{7.1}$$

The overall t statistic is then:

$$\frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s_p^2(\frac{1}{N_1} + \frac{1}{N_2})}} \quad (7.2)$$

The degrees of freedom are $N_1 + N_2 - 1$.

How to do a t Test in Python

Here's an implementation of all of that in Python.

```
def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1/n1)+(1/n2)) ,0.5)
    return t
```

And putting it all together, with reading our unemployment rates, we get:

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
```

```
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)) ,0.5)
    return t

unemdata = csvfile.CSVfile("USUnemploymentRate.csv")
months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year','1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year','2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year','2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average",average(rates1996)
print "variance",variance(rates1996)
print "number",len(rates1996)

print "2000 results"
print "average",average(rates2000)
print "variance",variance(rates2000)
print "number",len(rates2000)

print "2004 results"
print "average",average(rates2004)
print "variance",variance(rates2004)
```

```
print "number",len(rates2004)

print "1996-2000 ttest",ttest(rates1996,rates2000)
print "2000-2004 ttest",ttest(rates2000,rates2004)
```

And here's the run:

```
In [107]: run ttest-electoral.py
1996 results
average 5.40833333333
variance 0.120833333333
number 12
2000 results
average 3.96666666667
variance 0.0987878787879
number 12
2004 results
average 5.51666666667
variance 0.105151515152
number 12
1996-2000 ttest 10.6565919854
2000-2004 ttest -11.8897236086
```

This suggests that 2000 was much better (lower unemployment on average) than 1996, and 2004 was worse off than 2000. But was it significant? We can check a t-test table[4], assuming an *alpha* value (willingness to be wrong) of 0.05 and $(12 + 12 - 1 = 23)$ 23 degrees of freedom, we get a $t_{critical}$ value of 1.714. We reject H_0 if our t-value is greater than $t_{critical}$. Since our t-value is way larger, we say that the difference is significant at the 0.05 level. So, in 2000, people were better off, and in 2004, people were worse off.

Let's revisit our hypotheses:

- In 2000, Bush won over Gore (even though Gore won the popular vote). The American people *switched* parties.
- In 2004, Bush won over Kerry. The American people *stuck with* the Republican party.

That means that *neither* election matched the pundit's prediction – people *were* better off, but they changed parties (ignoring the popular vs. electoral vote complexity). In 2004, they were markedly *worse off* than they were in 2000, but they stuck with the same party.

Maybe the problem is that four years are too long? Maybe we would be better off looking only *two* years off? Let's change our analysis:

[4]We're using `http://www.socr.ucla.edu/Applets.dir/T-table.html`.

```
unemdata = csvfile.CSVfile("../data/USUnemploymentRate.csv")
months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']

#Let's get 1998 year.
rates1998 = []
# getRows returns a set. We want just one, the 0th
row1998 = unemdata.getRows('Year','1998')[0]
for item in months:
    rates1998.append(csvfile.number(row1998[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year','2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2002 year.
rates2002 = []
row2002 = unemdata.getRows('Year','2002')[0]
for item in months:
    rates2002.append(csvfile.number(row2002[item]))

unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year','2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1998 results"
print "average",average(rates1998)
print "variance",variance(rates1998)
print "number",len(rates1998)

print "2000 results"
print "average",average(rates2000)
print "variance",variance(rates2000)
print "number",len(rates2000)

print "2002 results"
print "average",average(rates2002)
print "variance",variance(rates2002)
print "number",len(rates2002)

print "2004 results"
print "average",average(rates2004)
print "variance",variance(rates2004)
print "number",len(rates2004)
```

```
print "1998-2000 ttest",ttest(rates1998,rates2000)
print "2002-2004 ttest",ttest(rates2002,rates2004)
```

Here were our results:

```
1998 results
average 4.5
variance 0.0127272727273
number 12
2000 results
average 3.96666666667
variance 0.00787878787879
number 12
2002 results
average 5.78333333333
variance 0.0106060606061
number 12
2004 results
average 5.51666666667
variance 0.0142424242424
number 12
1998-2000 ttest 12.8703946646
2002-2004 ttest 5.86015899227
```

2004 is a bit better off than 2002 (but still significant), and 2000 is better off than 1998. The pundits would suggest that 2000 and 2004 should both be "stick with" years. That describes 2004, but not 2000. Not clear if two years is a better comparison point than four. Maybe people don't accurately remember four years back, and are better at comparing two years back.

7.5 ANOVA: Analysis of Variance

The t test works well when we have two groups to be compared. What if we had more than two groups? The procedure for this is an *analysis of variance* (*ANOVA*).

An ANOVA can also be used with two groups. An ANOVA uses an F ratio rather than a t ratio. It is called an F ratio because it is named after its developer Sir Ronald Fisher.

Preliminary Calculations

Old Model		New Model	
Car	*MPG*	*Car*	*MPG*
1	30	1	37
2	34	2	36
3	34	3	40
4	29	4	36
5	33	5	34
		6	33
Totals:	160		216
Sample sizes:	5		6
Means:	32		36

We also compute the *grand total of scores* $= 160 + 216 = 376$. We compute the *total sample size* $= 5 + 6 = 11$. We compute the *grand mean* $= 376/11 = 34.18$.

Calculation of sum of squares

SSB = sum of squares between groups

$$= \text{number of scores in group 1 (the mean for group 1} - \text{grand mean})^2 + \text{number of scores in group 2 (the mean for group 2} - \text{grand mean})^2 + \cdots + \text{number of scores in last group (the mean for last group} - \text{grand mean})^2$$

$$= 5(32 - 34.18)^2 + 6(36 - 34.18)^2$$

$$= 43.64$$

SSW = sum of squares within groups

$$= (\text{1st score - mean of the group from which the first score comes})^2 + (\text{2nd score - mean of the group from which the second score comes})^2 + \cdots + (\text{last score - mean of the group from which the last score comes})^2$$

$$= (30-32)^2 + (34-32)^2 + (34-32)^2 + (29-32)^2 + (33-32)^2 + (37-36)^2 + (36-36)^2 + (40-36)^2 + (36-36)^2 + (34-36)^2 + (33-36)^2$$

$$= 52$$

Calculation of mean squares

MSB = mean square between groups

$$= \frac{SSB}{(\text{total number of groups}) - 1} = \frac{43.64}{2-1} = 43.64$$

MSW = mean square within groups

$$= \frac{SSW}{\text{total number of scores in all groups} - \text{total number of groups}} = \frac{52}{11-2} = 5.78$$

Calculation of the F Statistic

$$F = \frac{MSB}{MSW} = \frac{43.64}{5.78} = 7.55$$

Recall that the general idea in degrees of freedom reflects how many scores are free to vary. In an F test we need to be concerned about the degrees of freedom of the MSB and the MSB. The degrees of freedom for MSB is the $number of groups minus 1$; so in our case it is $2-1=1$. Given the grand mean in the experiment, if we know the mean of one group then the mean of the other group has to be a certain value. The degrees of freedom for MSW is $number of scores - number of groups$; so in our case it is $11-2=9$. The logic here is more or less the same.

Now we can look up our F statistic in a ***F table*** – see Figure **??**[5]. There are different tables for different probability levels. Let's look at the one for $p < .05$. The degrees of freedom for MSB are listed across the top, and the degrees of freedom for MSW are listed down the side of this table. So we look for 7.55 at the intersection of 1 and 9 degrees of freedom. We see that the F statistic is larger than the 5.117 listed there. So, we can say that our result is significant at the $.05$ level. In formal notation we would write $F(1,9) = 7.55, p < .05$. Again, this means that if the car experiment were conducted a large number of times, we would expect this result to occur by chance fewer than 5 in 100 times. Again, we say that we "reject the null (H_0) hypothesis."

7.6 Computing ANOVA: Analysis of Variance

Another way of testing the difference between groups is with an *ANOVA* or *Analysis of Variance*. Here, we look at variance more, and we use an *f statistic* rather than the *t statistic* (that is, the lookup table). One thing that's cool about ANOVA is that we can use it for more than two groups, to see if there is any difference anywhere. But we'll use it just for two groups here.

In ANOVA, we need the totals of the groups, the sample sizes, and the means, but also the *grand total* (of all the groups), the *total sample size* (of all groups), and the *grand mean* which is the grand total divided by the total sample size. Strangely enough, we don't actually use the variance in the Analysis of Variance (ANOVA) process.

Here's the process, in computational terms:

- We compute the *SSB*, the sum of squares between groups.

$$SSB = (N_1(x_1 - GrandMean))^2 + (N_2(x_2 - GrandMean))^2 \tag{7.3}$$

[5]From `http://www.itl.nist.gov/div898/handbook/eda/section3/eda3673.htm`.

(You can easily see how this would extend to more groups.)

- We compute the *SSW*, the sum of squares *within* groups. For all items in *all* groups,

$$SSW = (item1-(meanofitem1'sgroup))^2+(item2-(meanofitem2'sgroup))\ldots \tag{7.4}$$

- We then compute the *MSB*, mean square between groups. That's simpler to compute:

$$MSB = \frac{SSB}{numberofgroups - 1} \tag{7.5}$$

- We then compute the *MSW*, mean square within groups.

$$MSW = \frac{SSW}{(totalsamplesize) - (numberofgroups)} \tag{7.6}$$

- The F-statistic is:

$$F = \frac{MSB}{MSW} \tag{7.7}$$

- The degrees of freedom between groups is the number of groups - 1. The degrees of freedom for the total is the total sample size - 1. The degrees of freedom within groups is the degrees of freedom for the total minus the degrees of freedom between groups. I find on some F-statistic tables[6], the between groups is called *df1* and the within groups is called *df2*.

How to do an ANOVA in Python

Here's how we map that process to Python.

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
```

[6]As at http://www.statsoft.com/textbook/sttable.html#f05

```
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2))  ,0.5)
    return t

def anova(seq1,seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize -2)
    return MSB/MSW
```

Then, here's the whole thing, including the reading of the data again.

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
```

```
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)) ,0.5)
    return t

def anova(seq1,seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize-2)
    return MSB/MSW

unemdata = csvfile.CSVfile("USUnemploymentRate.csv")
months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year','1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year','2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))
```

```
unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year','2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average",average(rates1996)
print "variance",variance(rates1996)
print "number",len(rates1996)

print "2000 results"
print "average",average(rates2000)
print "variance",variance(rates2000)
print "number",len(rates2000)

print "2004 results"
print "average",average(rates2004)
print "variance",variance(rates2004)
print "number",len(rates2004)

print "1996-2000 anova",anova(rates1996,rates2000)
print "2000-2004 anova",anova(rates2000,rates2004)
```

And here are the results:

```
1996 results
average 5.40833333333
variance 0.120833333333
number 12
2000 results
average 3.96666666667
variance 0.0987878787879
number 12
2004 results
average 5.51666666667
variance 0.105151515152
number 12
1996-2000 anova 659.75751503
2000-2004 anova 1303.2739726
```

The degrees of freedom between groups is 1 ($2groups - 1$). The degrees of freedom total is 23 ($24 - 1$). The degrees of freedom within groups is 22 ($23 - 1$). The F-statistic there is 4.3009. These values (659 and 1303) are, ahem, a tad bit larger. So, we come up with the same result as with the t-test.

So, for the 2000 and 2004 elections, the pundit's prediction held true. As goes the unemployment rate, so goes the presidential vote.

7.7 Calculation of Significance of a Correlation

Let's go back to correlations for a minute. Think back to our data about self-reported helpfulness being correlated with the number of helpful acts a person was observed doing. When we calculated the correlation between those two variables, we found $r = 0.90$. While that looks like a large number (given that the maximum for a positive correlation is 1.0), we still need to ask whether it is a significant correlation. That is, is the correlation meaningfully different from 0.

So, the null hypothesis is that the true correlation between two variables, X and Y, in the population is 0 and that we found this $.9$ correlation by chance. If the size of the sample, N, on which an observed value of r is based is equal to or greater than 6, then we can calculate a t statistic using the correlation and then use the t table to look up the value to see what probability is associated with it The degrees of freedom are $N2$, so $10 - 2 = 8$.

The formula for calculating $t$in this case is:

$$t = \frac{r}{\sqrt{\frac{(1-r^2)}{(N-2)}}}$$

- So, in our example, $r = .90$ and $N = 10$. - So,

$$t = \frac{.90}{\sqrt{\frac{(1-.9^2)}{(10-2)}}} = \frac{.9}{\sqrt{\frac{.19}{.8}}} = .9/\sqrt{.024} = .9/.154 = 5.84$$

We look up 5.48 with 8 degrees of freedom and find that it is significant beyond the .001 level.

8 Multiple Linear Regression and Advanced Experimental Designs

Consider these two graphs. Each dot in Figure 8.1 represents a single student. The horizontal axis represents the high school GPA of that student, and the vertical axis represents the college GPA of the same student. Each dot in Figure 8.2 represents the *same* students. But now, the horizontal axis represents SAT score, and the vertical axis represents the college GPA of the student.

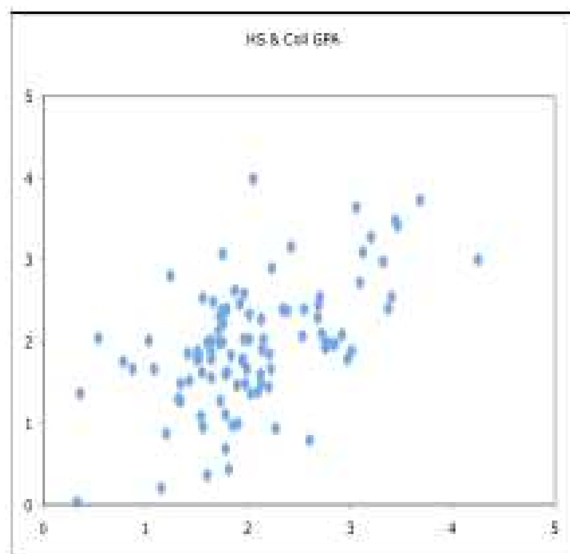


Figure 8.1: Scatterplot of high school and college GPA's

In each of these graphs, draw a line that best represents the relationship between the variables. We're serious – go get a pencil and pen and draw in the book. Right now. We'll wait.

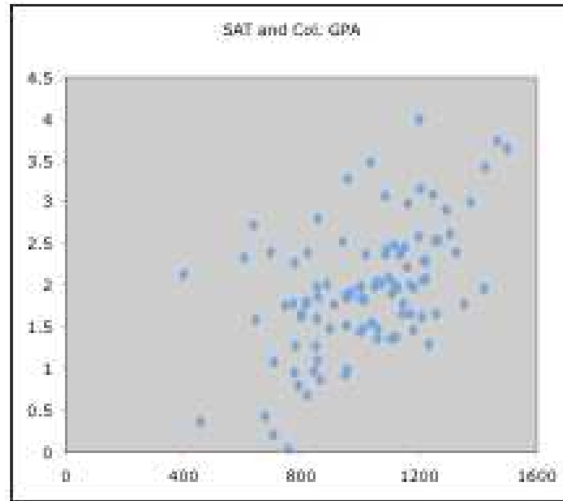


Figure 8.2: Scatterplot of SAT score and GPA

Hmm. Hmm. Hmm.

Back? Okay – what makes that a *good* line? One that crosses lots of dots would be good. One that's "representative" of the dots.

Here's a concrete example: The best line is one that *minimizes* the sum of the distances from each dot to that line. If you cross a dot, then the distance to the line for that dot is zero. If you draw a line far away from all the other dots, the sum of all the distances is huge. You may remember your formula for distance d from a point (x_0, y_0) to a line $ax + by + c = 0$ as:

$$d = \sqrt{(x_0 - x)^2 + (y_0 - y)^2} = |ax_0 + by_0 + c|/\sqrt{a^2 + b^2}$$

We want to minimize all those distances.

Now, what if you move into three dimensions. Instead of fitting a line to 2-D data, we are now fitting a plane for two independent variables (or a space for three independent variables). In the case of our two figures, we would want to predict college GPA as a function of several possible predictors: high school GPA, SAT scores, and maybe quality of letters of recommendation.

8.1 The Multiple Regression Equation

The multiple regression equation takes the form:

$$y = b_1x_1 + b_2x_2 + ... + b_nx_n + c$$

- The x_i are the independent variables, the variables that you expect will predict the outcome or dependent variable y.

- The c is the constant (also called the *intercept*), where the regression line intercepts the y axis, representing the amount the dependent y will be when all the independent variables are 0. Sometimes this has real meaning and sometimes it doesn't. In other words, sometimes the regression line cannot be extended beyond the range of observations, either back toward the y axis or forward toward infinity.

- The b_i's are the regression coefficients, representing the amount the dependent variable y changes when the corresponding independent changes 1 unit. The standardized version of the b coefficients are called the *beta weights*, and these are what are typically shown in regression equations. The beta weight is the average amount the dependent variable increases when the independent variable increases one unit and all other independent variables are held constant.

  For instance, if an independent variable has a beta weight of $.6$, this means that when other independent variables are held constant, if the independent variable increases by 1 unit, the dependent variable will increase by .6 units. The independent variable with largest beta weight is that which, controlling for all the other independent variables, has the largest unique explanatory effect on the dependent variable.

Associated with multiple regression is R^2, *multiple correlation*, which is the percent of variance in the dependent variable explained collectively by all of the independent variables.

Mathematically, $R^2 = 1 - (SSE/SST)$.

$$SSE = errorsumofsquares = \sum (Y_i - \hat{Y}_i)^2$$

where Y_i is the actual value of Y for the i-th case., and $\hat{Y}_i$ is the regression prediction for the i-th case.

$$SST = totalsumofsquares = \sum (Y_i - \bar{Y})^2$$

Thus R^2 will be 0 when regression is as large as it would be if you simply guessed the mean for all cases of Y.

Ultimately our goal in constructing the regression equation is to create an equation that minimizes the differences between the actual values of the dependent variable and the predicted values from the regression equation. This is why computers are so important for doing regressions because many combinations of the variables have to be considered in order to find the best equation.

Note that the beta weights reflect the unique contribution of each independent variable. Joint contributions contribute to R^2 but are not attributed to any particular independent variable. The result is that the beta weights might underestimate the importance of an independent variable that makes strong joint contributions to explaining the dependent variable but which does not make a strong unique contribution.

Multiple regression can establish:

- a set of independent variables that explains a proportion of the variance in a dependent variable at a significant level – through a significance test of R^2; and

- the relative predictive importance of the independent variables – by comparing beta weights.

One can test the significance of difference of two R^2's to determine if adding an independent variable to the model helps significantly. *Regression analysis* is a linear procedure. To the extent nonlinear relationships are present, conventional regression analysis will *underestimate* the relationship of particular independent variables to the dependent variable. That is, R^2 will underestimate the variance explained overall and the betas will underestimate the importance of the variables involved in the nonlinear relationship.

There is a clear relationship between multiple regression and correlation. In a correlation, R^2 is the percent of variance in the dependent explained by the given independent when (unlike the beta weights) all other independent variables are allowed to vary (because we are not considering them).

Testing Significance of Beta Weights

We can use t tests to assess the significance of individual beta weights. specifically testing the null hypothesis that a given beta weight is zero. The t statistic for a beta weight is:

$$t(N-k-1) = \frac{b}{s_b}$$

where b is the regression coefficient or beta weight, s_b is the standard error of the regression coefficient or beta weight, N is the number of subjects, and k is the number of predictor variables. The degrees of freedom on which to evaluate the t statistic is $N-k-1$.

Returning to the problem of predicting college GPA, we ran the sample described by the figures at the start of this chapter through multiple regression. The regression coefficients / beta weights and associated significance tests are shown below:

	b	s_b	t	p
HS GPA	.3764	.1143	3.29	.0010
SAT	.0012	.0003	4.10	.0001
Letters	.0227	.0510	0.44	0

The regression coefficients / beta weights for High School GPA and SAT are both highly significant. The coefficient / beta weight for Letters of recommendation is not significant. This means that there is no evidence that the quality of the letters adds to the predictability of college GPA once High School GPA and SAT are known. A common rule of thumb is to drop from the equation all variables not significant at the $.05$ level or better.

Assessing Importance of a Beta Weight

Assessing a variable's importance using R^2 increments is very different from assessing its importance using beta weights.

- The magnitude of a variable's beta weight reflects its relative explanatory importance controlling for other independents in the equation.
- The magnitude of a variable's R^2 increment reflects its additional explanatory importance given that common variance it shares with other independent variables entered in earlier steps has been absorbed by these variables.

We use each for different purposes. For causal assessments, beta weights are better. For purposes of sheer prediction, R^2 increments are better.

Testing Significance of R^2

The F test is used to test the significance of R^2, which is the same as testing the significance of the regression model as a whole. If $prob(F) < .05$, then the model is considered significantly better than would be expected by chance, so we reject the null hypothesis of no linear relationship of y to the independent variables.

The F is a function of R^2, the number of independent variables, and the number of cases. F is computed with k and $(n - k - 1)$ degrees of freedom, where $k =$ the number of terms in the equation not counting the constant.

$$F = \frac{\frac{R^2/k}{(1-R^2)}}{n-k-1}$$

Regression Fallacy

Imagine that you are testing some educational intervention. You give students a pre-test, to see what they know already. You then do something to the students to help them learn, like forcing them to watch 10 hours of

Teletubbies. You then give students a post-test. By subtracting the pre-test score from the post-test score, you can measure the change in knowledge which you might attribute to learning from your intervention.

In virtually all test-retest situations, the bottom group on the first test will on average show some improvement on the second test and the top group will on average fall back. The regression fallacy consists in thinking that a regression effect must be due to something important. In reality, the effect might just be due to the spread around the regression line due to random factors. We say that the observed test score = true score + chance error.

Imagine an IQ test. If someone scores above average on the first test, we are forced to estimate that his true score is a bit lower than the observed score. If he takes the test again, we have to predict that his second score will be a bit lower than his first score. On the other hand, if he scores below average on the first test, we estimate that his true score is a bit higher than the observed score. Our prediction for the second score is a bit higher than the first score. *It is not the true score that changes from test to test, but our estimate of it.*

8.2 Computing a multiple regression

There are several multiple regression packages for Python. We are going to use one that is particularly good for use with *SciPy*. Download OLS (for "ordinary least squares") from `http://www.scipy.org/Cookbook/OLS` and put it in your source code folder.

To use this program, we need to create a *matrix* with the independent variable data and an *array* with the dependent variable data. We give each of them to the OLS function, with the name of the dependent variable, and a list of the names of the dependent variables. The data set we're using has 100 students in it, with the same variables as we described earlier in the chapter, but not the same 100 students.

```
from pylab import *
import csvfile
import ols

#There are a hundred data elements
# We need an X and y array, each of type "Float"
# There are 100 values in each
# And three variables, plus one constant
y = zeros(100)
X = zeros((100,4))
X[:,3] = ones((100,))

#Read the data
gpafile = csvfile.CSVfile("../data/CollegeGPA.csv")
ydata = gpafile.getColumn("College GPA")
```

```
rownum = 0
for row in ydata:
    y[rownum]=row
    rownum=rownum+1

gpafile.rewind()
skiprow=gpafile.next()  #Skip the headers
rownum = 1
for row in gpafile.dataReader:
    hsgpa = csvfile.number(row["HS GPA"])
    sat = csvfile.number(row["SAT"])
    letter = csvfile.number(row["LetterQual"])
    X[rownum,0]=hsgpa
    X[rownum,1]=sat
    X[rownum,2]=letter
    rownum = rownum + 1

# Call OLS
results = ols.ols(y,X,"College GPA",["HS GPA","SAT","LettterQual"])
results.summary()
```

The results summary from OLS is terrific – it has more statistics in it than we actually know the meanings for! We can see the the model is overall highly significant (the probability on the F statistic is 0.00000). While the values are different, the model from this data set matches the one described above, in that high school GPA and SAT scores are significant predictors of the college GPA, but not the recommendation letters.

```
==============================================================================
Dependent Variable: College GPA
Method: Least Squares
Date:  Thu, 31 May 2012
Time:  12:18:51
# obs:                 100
# variables:         5
==============================================================================
variable     coefficient     std. Error      t-statistic     prob.
==============================================================================
const           0.000000      8933311.925715      0.000000      1.000000
HS GPA           0.401808      0.139627      2.877722      0.004947
SAT             0.000878      0.000348      2.525769      0.013199
LettterQual          -0.013541      0.060947     -0.222171      0.824658
==============================================================================
Models stats                         Residual stats
==============================================================================
R-squared              0.333158         Durbin-Watson stat   1.385116
Adjusted R-squared    0.305080         Omnibus stat         3.112009
F-statistic           11.865618         Prob(Omnibus stat)   0.210977
Prob (F-statistic)    0.000000 JB stat              2.580873
```

```
Log likelihood       -106.755270 Prob(JB)            0.275151
AIC criterion         2.235105        Skew                  0.381562
BIC criterion         2.365364        Kurtosis              3.192490
==============================================================================
```

8.3 Final Note on Alternative Experimental Designs

Now that we have seen correlations, hypothesis testing, and multiple linear regression, we can return to some themes in experimental design that we first addressed in Chapter 1. But now, we can talk more about issues of statistics that we had not yet discussed when we first described experimental design.

We have tried to stress two themes in experimental design and data analysis.

- With respect to design, there is the necessity of beginning an experiment with groups that can be assumed to be equal.
- With respect to analysis, there is the statistical importance of reducing variation due to error (which, for instance, goes into the denominator of the F ratio).

In relation to the need for beginning an experiment with equal groups, we suggested that the random assignment of participants is one of the best techniques for ensuring that no systematic differences are present at the beginning of an experiment. However, there are times when random assignment alone might not be the most appropriate approach to a particular research question. We discuss two approaches to experimental design that do not use random assignment alone.

- The first is a *within-subjects design* in which each participant is exposed to different experimental conditions.
- The second is a *matched-subjects design*.

Again, we want to stress the need for beginning an experiment with equal groups and the importance of reducing the error groups variance (within-subjects variance) of the F ratio.

Within-Subjects Designs

In most cases we've discussed so far, we've compared the performance of one group of participants with the performance of a different group of participants. These designs are called *between-subjects designs*. In within-subjects designs the participant's own performance is the basis of comparison; that is, every participant receives all levels of the independent variable. In these designs we compare the performances of the same set of participants on the dependent variable following different treatments.

Let's look at an experiment performed first as a between-subjects experiment and then as a within-subjects experiment. In a simple between-subjects experiment, the experimenter wants to determine the role of immediate feedback in say, shooting basketballs. One group is instructed to shoot baskets while wearing a blindfold (zero level of feedback) and another group is told to shoot without a blindfold (high level of feedback). We can perform this same study as a within-subjects design in which each participant is exposed to both levels of the independent variable, that is, every participant in the study shoots baskets both with and without a blindfold. The within-subject design accomplishes both of the goals we mentioned earlier: equating groups before the presentation of the independent variable and reducing error variance. Because the same participants are used in each group, we can be certain that before any treatment has begun, the groups are exactly the same.

A within-subjects design also increased the sensitivity of a study by decreasing the error variance because it removes the variance that results from individual variability. The error variance term of the F ratio for a within-subjects design is statistically smaller than that of a comparable between-subjects design. This means that smaller treatment differences are adequate for rejecting the null hypothesis.

A potential problem in the basketball study is that the results from the no-feedback condition might have a possible carryover effect on the feedback condition. The act of shooting a basketball, even without visual feedback, might lead one to perform better in the next condition. That is, practice could confound this design.

One way to control for this potential problem would be to use a *counterbalancing procedure*. For example, we could have all participants first shoot the basketball in the no-feedback condition, then in the feedback condition, then in the feedback condition again, followed by the no-feedback condition. This would generate an ABBA order and thus help to control for the effects that are carried over from one trial set to another.

Matched-Subjects Designs

Within-subjects experiments equate groups by using the same participant in every treatment condition. By using the same participants in each group, the within-groups variance is reduced because a participant will perform more consistently in different situations than different participants will perform in difference situations. What if, rather than using the same participants in different groups, we use participants who are very similar?

This would be a matched-subjects design. A matched-subjects design would allow us to reap some of the advantages of within-subjects designs and simultaneously take advantage of the random assignment of participants that is possible with a between-subjects design. We might need to use a matched-subjects design if we think our treatment might have long-

lasting effects and we can't move participants from condition to condition cleanly.

In a matched-subjects design, we pair participants along some factor and then randomly assign the members of each pair to two separate groups. In this way we can assume that our groups are equal at the beginning of the experiment and we can reduce within-groups variance (error variance). By this method we create a type of design that is a hybrid of within-subjects designs and between-subjects designs. We could not have an much error variance as in a between-subjects design or at little error variance as in a within-subjects design.

The characteristics we use to match our groups of participants obviously are of central importance to this design. In the broadest sense, any physical or mental characteristic of a participant that can be measured may be used for matching. Characteristics such as height, weight, intelligence, anxiety level, achievement motivation, hair color, and emotional sensitivity are all individual characteristics that can be measured. Consequently, they are all potential individual variables on which matching can be based.

For a matching procedure to work, there must be a high correlation between the variable used for matching and the dependent variable. If the participants are matched on a factor that does not correlate with the dependent variable, then the within-groups variance will be no smaller than the variance obtained by random selection alone. In this case the amount of effort required for matching will have been wasted.

A A Brief Introduction to Key Parts of Python

A.1 Variables and Assignment

Variables are names that we use as references for values. They can have pretty much any number of characters you want, but start with a letter, and can include numbers and underscore ("_"). Uppercase and lowercase letters *are* different in Python.

Strings can be delimited with single or double-quotes in Python. We can inspect the values of variables with **print**. You can also just type the variable name and hit return/enter to see the value. There is a slight difference in what each prints.

```
>>> a = "fred"
>>> a
'fred'
>>> print a
fred
>>> a = 'mark'
>>> a
'mark'
>>> print a
mark
>>> print A

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print A
NameError: name 'A' is not defined
>>> A = 12
>>> print A
12
>>> print a
mark
>>> print a,A
mark 12
```

We can make Python do some conversions for us using eval. The float function can do that well, too.

```
>>> eval("123")
123
```

A.2 Lists

Python is particularly good at manipulating lists. Lists are delimited with square brackets ("[ ]"), with commas separating the elements of the list. The first item of the list has an index of zero.

```
>>> mylist = [12,25.4,"testing",'another string',[6,-7]]
>>> print mylist
[12, 25.4, 'testing', 'another string', [6, -7]]
>>> mylist[0]
12
>>> mylist[1]
25.4
>>> len(mylist)  #len() is a function that measures length
5
>>> mylist[5]  # 5 elements, but last index is 4

Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    mylist[5]
IndexError: list index out of range
>>> mylist[4]  # Yes, that's a list inside the list
[6, -7]
>>> mylist[4][1]
-7
>>> mylist[4][0] #Last element of mylist, first element of that
6
```

We can add to the end of the list with append.

```
>>> a = [1,2,3]
>>> a.append([4])
>>> a
[1, 2, 3, [4]]
>>> a.append(5)
>>> a
[1, 2, 3, [4], 5]
```

One of the particular powers of Python is its list manipulations. You can *slice* lists by giving two indices, separated by a colon. They describe a starting point for a slice and an ending point.

```
>>> mylist[0:3]
[12, 25.4, 'testing']
>>> mylist[1:3]
[25.4, 'testing']
```

What's particularly cool is that the slice values can be missing or negative. If you skip the first index, the assumption is "the start." If you skip the last index, the assumption is "the end." If you use a negative index, it counts from the start (first) or the last (second).

```
>>> mylist[:]  # Skip both, get the whole thing
[12, 25.4, 'testing', 'another string', [6, -7]]
>>> mylist[1:] #From 1 to end
[25.4, 'testing', 'another string', [6, -7]]
>>> mylist[:3] #From start to 3
[12, 25.4, 'testing']
>>> mylist[:-1]  #From start, one before end
[12, 25.4, 'testing', 'another string']
>>> mylist[-1:] # Only the last (-1 from start, to end)
[[6, -7]]
```

A.3 Dictionaries

Dictionaries are sometimes called *named arrays* or *associative arrays*. Think of them as lists or arrays, where the indices aren't numbers but strings. Those strings are called the *keys*.

```
>>> mydict = {}  #Creates an empty dictionary
>>> mydict["Mark"] = 'Guzdial'
>>> mydict['Richard'] = "Catrambone"
>>> mydict["Mark"]
'Guzdial'
>>> mydict["Richard"]
'Catrambone'
>>> mydict.keys()  #Call the method keys() on the object mydict
['Richard', 'Mark']
```

Notice that dictionaries are *objects*. We access *methods* on objects using dot notation.

A.4 Blocks

The most unusual feature of Python is that indentation defines blocks. A collection of statements (e.g., after IF, WHILE, or FOR, or as part of the definition of the function) are defined as statements at the same level of indentation. Spaces or tabs will work, as long as all the lines match up. This makes the code quite readable. It can be a pain to debug if you get it wrong.

A block follows a statement that ends with a colon. We'll see that in the next section with functions.

A.5 Functions

We define functions using **def**. We call them by using the name of the function with parameters in parentheses.

```
>>> def myfunction(input):
        print input * 5


>>> myfunction  # This prints the value of the function, not calling it
<function myfunction at 0x7b9bc70>
>>> myfunction(5)
25
>>> myfunction("fred")  #Multiplication works on strings
fredfredfredfredfred
>>> myfunction()  #Must provide one input

Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    myfunction()
TypeError: myfunction() takes exactly 1 argument (0 given)
```

One of the unusual aspects of functions in Python is that a function can **return** multiple values, and you can set multiple variables at once to the return.

```
>>> def return3():
        return 1,2,3

>>> a,b,c=return3()
>>> a
1
>>> b
2
>>> c
3
```

A.6 FOR loops

The **for** loop in Python actually is a *for-each* loop. It processes each element in a collection, with the index variable bound to each item once.

```
>>> for letter in 'thisString':
        print letter


t
h
i
s
```

```
S
t
r
i
n
g
>>> for item in [1,2,[3,4],'alpha','bet']:
        print item


1
2
[3, 4]
alpha
bet
```

The range function lets us create lists of values so that we can use them as indices.

```
>>> range(0,3)
[0, 1, 2]
>>> range(1,3)
[1, 2]
>>> range(3)
[0, 1, 2]
```

```
>>> somelist = [1,2,3,5,6]
>>> print len(somelist)
5
>>> for index in range(0,len(somelist)):
        print somelist[index]


1
2
3
5
6
>>> sum = 0
>>> for index in range(len(somelist)):
        value = somelist[index]
        sum = sum + value


>>> print sum
17
```

A.7 Conditionals

You can test with an **if** statement. After the test (in parentheses), you have a colon, to indicate the “then” block. You can optionally have an **else** clause, also with a colon.

```
if "a" < "b":
        print "True!"
else:
        print "False!"
```

Prints “True!” of course.

B Reading from Live Data

Aibek Musaev wrote for us three examples of how to read various "live" data sets from Python. These are data sets that are continuously updated, like radiation levels from the Fukushima prefecture every 10 minutes. The code is a bit more complicated than the rest in the book, so we moved them here into an appendix. For example, they each define a *class* for making it easier to manipulate the data. You don't have to understand the class to use the code. See the Python code at the bottom of each example as a demonstration for how to use the class.

Live data sets are available in different formats. Aibek provided us with three examples for reading three different kinds of data sets. Thanks, Aibek!

Reading from XML Sources

```
#
# This is an example of reading a data source published
# in XML format and archived to ZIP format from the web.
# It reads international passenger survey time series
# dataset from the Office for National Statistics in UK
# (http://www.ons.gov.uk/ons/index.html).
#

from urllib2 import urlopen
from xml.dom.minidom import parseString
from zipfile import ZipFile
from StringIO import StringIO

# Class for handling passenger survey time series dataset
class PassengerSurveyUK:
    # initialize data in the default constructor
    def __init__(self):
        #internal data structure for passenger survey data
        self._pdata = {}

    # load dataset into internal data structure from url
    # published in xml format and archived to zip format
    def loadFromUrl(self, url):
```

```
        #open connection to the url
        f = urlopen(url)
        #download the zip file
        z = ZipFile(StringIO(f.read()))
        #read the first and only file from the archive
        x = z.read(z.namelist()[0])
        #get document parsed into a DOM
        dom = parseString(x)
        #extract values to an internal data container
        for node in dom.getElementsByTagName('wdp:Section'):
            date = node.getAttribute('Date')
            for e in node.childNodes:
                if (e.nodeType == e.ELEMENT_NODE):
                    value = e.getAttribute('value')
                    self.add(e.localName, date, int(value))
        #close handles
        z.close()
        f.close()

    #Add read data into a local data container
    def add(self, code, date, value):
        if (code not in self._pdata.keys()):
            self._pdata[code] = {}
        coll = self._pdata[code]
        coll[date] = value

    #Demo function: find the year when the passed code had
    #the maximum value
    def maxValueByCode(self, code):
        topDate = ''
        topValue = 0
        coll = self._pdata[code]
        for date in coll.keys():
            if (coll[date] > topValue):
                topDate = date
                topValue = coll[date]
        return topDate, topValue

#instantiate data class
pdata = PassengerSurveyUK()
#load data from url
pdata.loadFromUrl('http://www.ons.gov.uk/ons/datasets-and-tables/downloads/data.zip?
#run a demo function
print pdata.maxValueByCode('GMAT')
```

Reading from HTML data

```
#
# This is an example of reading a data source published
# as HTML. It reads real-time radiation sensor data
```

```
# in Fukushima prefecture from the SPEEDI project.
# This data gets updated every 10 minutes.
#

from urllib2 import urlopen

# Class for handling real-time radiation data in Fukushima
# prefecture from SPEEDI project
class RadiationData:
    # initialize data in the default constructor
    def __init__(self):
        #internal data structure for radiation data
        self._rdata = {}

    # load radiation data into internal data structure
    # from url published as html
    def loadFromUrl(self, url):
        #open connection to the url
        f = urlopen(url)
        #read the url to an array of lines
        lines = f.readlines()
        #close connection
        f.close()

        #extract values to a dictionary
        self.add('Shigeoka', lines[18])
        self.add('Shigeoka', lines[18])
        self.add('Namikura', lines[21])
        self.add('Kamikooriyama', lines[24])
        self.add('Hotokehama', lines[27])
        self.add('Tomioka', lines[30])
        self.add('Mukaihata', lines[33])
        self.add('Ono', lines[36])
        self.add('Ottozawa', lines[39])
        self.add('Yamada', lines[42])
        self.add('Kooriyama', lines[45])
        self.add('Tanashio', lines[48])
        self.add('Namie', lines[51])
        self.add('Kiyohashi', lines[54])
        self.add('Yamadaoka', lines[57])
        self.add('Yonomori', lines[60])
        self.add('Shinzan', lines[63])
        self.add('Futatsunuma', lines[66])
        self.add('Matsudate', lines[69])
        self.add('Shimokooriyama', lines[72])
        self.add('Kumagawa', lines[75])
        self.add('Minamidai', lines[78])
        self.add('Kami-Hatori', lines[81])
        self.add('Ukedo', lines[84])
```

```
    #Add read data into a local data container
    def add(self, area, value):
        self._rdata[area] = self._get_value(value);

    # helper function: return integer value from html line
    def _get_value(self, value):
        # strip the ending html code
        br = '<br />\n'
        str = value.rstrip(br)
        if (str == 'Under survey'):
            # return 0 instead of 'Under survey'
            return 0
        else:
            # convert string to integer
            return int(str)

    #Demo function: find the area with the highest radiation value
    def maxRadiationValue(self):
        maxArea = ''
        maxValue = -1
        for area in self._rdata.keys():
            if (self._rdata[area] > maxValue):
                maxArea = area
                maxValue = self._rdata[area]
        return maxArea, maxValue

#instantiate data class
rdata = RadiationData()
#load data from url
rdata.loadFromUrl('http://www.bousai.ne.jp/mob/rsd.php?lang=en&id=07')
#run a demo function
print rdata.maxRadiationValue()
```

Reading from Excel Workbooks

```
#
# This is an example of reading a data source published
# in Excel format on the web. It reads some historical data
# on eCommerce published on census.gov web site, which
# has loads of data there.
# The library for reading Excel files is called xlrd, which
# can be found here: http://pypi.python.org/pypi/xlrd/
#

from os import remove
from urllib2 import urlopen
from xlrd import open_workbook

# Class for handling historical eCommerce data from census.gov
class eCommerceData:
```

```
# initialize data in the default constructor
def __init__(self):
    #internal data structure for eCommerce data
    self._edata = {}

# Helper function: download file from URL
# We have to download the Excel file, because
# xlrd library works only with local files
def _download_file(self, url, outfile):
    try:
        #open url
        webFile = urlopen(url)
        #create a new local file
        localFile = open(outfile, 'wb')
        #write the contents of the url to the local file
        localFile.write(webFile.read())
        webFile.close()
        localFile.close()
    except IOError, e:
        print "Download error"

# load eCommerce data into internal data structure
# from url published in Excel format
def loadFromUrl(self, url):
    #file name for local Excel file name
    fname = 'temp.xls'
    self._download_file(url, fname)

    #open the workbook
    wb = open_workbook(fname)
    #wrk with each worksheet
    for sh in wb.sheets():
        #data is located on rows 9 through 29 - on both worksheets
        for row in range(8,29):
            industry = sh.cell(row,1).value
            for i in range (0, 5):
                col = 2+i*2
                year = sh.cell(4, col).value
                totalValue = sh.cell(row, col).value
                eCommerceValue = sh.cell(row, col+1).value
                self.add(industry, year, totalValue, eCommerceValue)
    #delete local file after use
    remove(fname)

#Add read data into a local data container
def add(self, industry, year, totalValue, eCommerceValue):
    if (industry not in self._edata.keys()):
        self._edata[industry] = {}
    coll = self._edata[industry]
    coll[year] = [totalValue, eCommerceValue]
```

```
    #Demo function: find the industry with the highest eCommerce value
    #in a given year
    def maxEcommerceByYear(self, year):
        topIndustry = ''
        topValue = -1
        for industry in self._edata.keys():
            coll = self._edata[industry]
            if (coll[year][1] > topValue):
                topIndustry = industry
                topValue = coll[year][1]
        return topIndustry, topValue

#instantiate data class
edata = eCommerceData()
#load excel file
#url is hardcoded, because this class is for a particular Excel file
edata.loadFromUrl('http://www.census.gov/econ/estats/2009/historical/2009ht1.xls')
#run a demo function
print edata.maxEcommerceByYear('1999\nRevised')
```

C Program Listings

C.1 CVSfile

Program Example #0

CVSfile

```
## CSVfile -- a front end to CVS

import csv

def number(input, default=-1):
    try:
        return float(input)
    except:
        return default

class CSVfile:
    def __init__(self, filename):
        self.filename = filename
        self.rewind();

    def rewind(self):
        self.fp = open(self.filename, "rb")
        headerReader = csv.reader(self.fp)
        self.headers = headerReader.next()
        self.dataReader = csv.DictReader(self.fp, fieldnames=self.headers)

    def next(self):
        return self.dataReader.next()

    def getRows(self, fieldname, value):
        ret =[]
        for row in self.dataReader:
            if row[fieldname]==value:
                ret.append(row)
        return ret
```

```
    def getColumn(self,fieldname):
        ret = []
        for row in self.dataReader:
            ret.append(row.get(fieldname))
        return map(number,ret)
```

C.2 fancierplot.py – a run-able plot

Program Example #1

fancierplot.py

```
from pylab import *
import csvfile
popdata = csvfile.CSVfile("pops-2000.csv")
pops = popdata.getColumn("POP")
spops=sort(pops)

plot(spops[1:],marker="o",color="r")
title('Populations of countries in the year 2000')
xlabel('Countries in increasing order of population')
ylabel('Population in millions')
grid(True)
show()
```

C.3 US-UK Population Plot for years 1999–2000

Program Example #2

us_uk_pop_plot.py

```
from pylab import *
import csvfile
import correl
natdata = csvfile.CSVfile("../data/us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
uspops = []
```

```
for row in usdata:
    uspops.append(csvfile.number(row['POP']))
ukpops = []
for row in ukdata:
    ukpops.append(csvfile.number(row['POP']))
years=range(1990,2001)
print "US",uspops,len(uspops)
print "UK",ukpops,len(ukpops)
print "Years",years,len(years)

#print "Correlation:",correl.r(uspops,ukpops)

plot(years,uspops,'r--o',years,ukpops,'b-x')
legend(('US Population','UK Population'),loc='center right')
title('Populations of US and UK 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)
savefig("us_uk_pop_plot.eps")
show()
```

Program Example #3

us_uk_pop_plot2.py

```
from pylab import *
import csvfile
natdata = csvfile.CSVfile("../data/us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
# This time, making SURE that they're in year-order
years=range(1990,2001)
uspops = []
for y in years:
    for row in usdata:
        if row['year']==str(y):  #Items in rows are strings
            uspops.append(csvfile.number(row['POP']))
            break #Leave the row loop
ukpops = []
for y in years:
    for row in ukdata:
        if row['year']==str(y):
            ukpops.append(csvfile.number(row['POP']))
```

```
            break

# Top subplot: 2 rows, 1 column, subplot #1
subplot(2,1,1)
plot(years,uspops,'r--o')
title('Population of US 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

subplot(2,1,2)
plot(years,ukpops,'b-x')
title('Population UK 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

savefig("us_uk_pop_plot2.eps")
show()
```

C.4 Exploring British and American Petroleum Company Stock Prices

Program Example #4

bpStdDev1990.py–computing descriptive statistics of BP and XOM

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 0.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)
```

```
bpdata = csvfile.CSVfile("../data/BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** BP ***"
print "Closing values",closes
print "Average:",average(closes)
print "Standard Deviation:",std_dev(closes)

amdata = csvfile.CSVfile("../data/Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
closes = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** Exxon/Mobil ***"
print "Closing values",closes
print "Average:",average(closes)
print "Standard Deviation:",std_dev(closes)
```

Program Example #5

bpHist1990.py–computing a histogram of each

```
from pylab import *
import csvfile


bpdata = csvfile.CSVfile("../data/BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,1)
title("BP stock in 1990--Histogram")
```

```
hist(closes)

amdata = csvfile.CSVfile("../data/Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
closes = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,2)
title("Amoco/Mobil stock in 1990-Histogram")
hist(closes)

savefig("BP_AM_hist.png")
show()
```

Program Example #6

bpAmCorrel1990.py–computing a correlation between them

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)

def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
```

```
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator



bpdata = csvfile.CSVfile("../data/BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
bpcloses = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        bpcloses.append(csvfile.number(row['Close']))


amdata = csvfile.CSVfile("../data/Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
amcloses = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        amcloses.append(csvfile.number(row['Close']))

print "BP closing values:",bpcloses
print "average",average(bpcloses)
print "number",len(bpcloses)
print "standard deviation",std_dev(bpcloses)

print "Exxon-Mobil (American) closing values:",amcloses
print "average",average(amcloses)
print "number",len(amcloses)
print "standard deviation",std_dev(amcloses)



print "Correlation is ",correlation(bpcloses,amcloses)
```

* * *

C.5 Text Analysis: Shakespeare or Bacon?

Program Example #7

viztext.py

```
import re

def highlight(basename,pattern):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern,newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()

def highlightCapitals(basename):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Now make the new one
    html = open(basename+".html","wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=re.sub(r"(\b[A-Z][a-zA-Z]*)",r'<font color="red">\1</font>',text)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

Program Example #8

counttext.py

```
import re

def countText(basename,pattern):
    file = open(basename+".txt","rt")
```

```
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n') #Two returns = paragraph
    # Now, count the number of 'the's in the paragraph
    ret = []
    for s in newtext:
        ret.append(s.count(pattern))
    return ret

def countCapitals(basename):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n')
    # Count the capitals
    ret = []
    for para in newtext:
        match = re.split(r"\b[A-Z][a-zA-z]*",para)
        ret.append(len(match)-1)
    return ret
```

C.6 Hypothesis Testing: Does the unemployment rate make the President?

Program Example #9

electoral.py – t-test and ANOVA predicting electoral results

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance
```

```
def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)) ,0.5)
    return t

def anova(seq1,seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize-2)
    return MSB/MSW

unemdata = csvfile.CSVfile("../data/USUnemploymentRate.csv")
months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year','1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year','2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2004 year.
```

```
rates2004 = []
row2004 = unemdata.getRows('Year','2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average",average(rates1996)
print "variance",variance(rates1996)
print "number",len(rates1996)

print "2000 results"
print "average",average(rates2000)
print "variance",variance(rates2000)
print "number",len(rates2000)

print "2004 results"
print "average",average(rates2004)
print "variance",variance(rates2004)
print "number",len(rates2004)

print "1996-2000 anova",anova(rates1996,rates2000)
print "2000-2004 anova",anova(rates2000,rates2004)
```

Bibliography

[Bremmer and Narayan, 1998] Bremmer, J. and Narayan, M. (1998). The effects of stress on memory and the hippocampus throughout the life cycle: implications for childhood development and aging. *Developmental Psychopathology*, 10:871–886.

[Cole, 1986] Cole, P. M. (1986). Children's spontaneous control of facial expression. *Child Development*, 57(6):1309–1321.

[Guzdial, 2010] Guzdial, M. (2010). Does contextualized computing education help? *ACM Inroads*, 1(4):4–6.

[Levitt and Dubner, 2005] Levitt, S. D. and Dubner, S. J. (2005). *Freakonomics: A rogue economist explroes the hidden stide of everything*. HarperCollins, New York, NY.

[Mateas and Sengers, 2003] Mateas, M. and Sengers, P. (2003). *Narrative Intelligence*. John Benjamins Pub Co, Amsterdam.

[Voevodsky, 1974] Voevodsky, J. (1974). Evaluation of a deceleration warning light for reducing rear-end automobile collisions. *Journal of Applied Psychology*, 59:270–273.

[Webb et al., 2000] Webb, E., Campbell, D., Schwartz, R., and Sechrest, L. (2000). *Unobtrusive Measures, Revised Edition*. Sage Publications, Inc., Thousand Oaks, CA.

Index

alpha, 98
alpha value, 62
alternative hypothesis, 90
Analysis of Variance, 100
analysis of variance, 86, 98
ANOVA, 98, 100
append, 32, 118
arange, 39
array, 112
associative arrays, 119
authority, 4
average, 47, 48
average of the squared differences, 50

background, 68
backslash notation, 24
Bacon, Francis, 67
    The Advancedment of Learning, 67
    The Essays Of, 67
beta weights, 109
between-subjects design, 12
between-subjects designs, 114
binomial distribution, 2
BP, 47
British Petroleum, 51

causal, 62
cd, 25, 41
central limit theorem, 85
change directory, 25
chaos theory, 6
Cicero, 82
claims, 3
class, 30, 123
close, 68
closed system, 10
color, 41
Comma Separated Values, 25
common sense, 5
control group, 10
correlation, 55, 62
correlational approach, 7
correlational design, 15
count, 77
counterbalancing procedure, 115
CSV, 25
csv.DictReader, 28
csv.reader, 26

Darwin, Charles, 6
data journalism, 21, 24
dataReader, 48
def, 120
degrees of freedom, 62, 92
    t-test, 96
dependent variable, 14
descriptive statistics, 47
df, 62
df1, 101
df2, 101
dictionary, 29
directory, 25
    changing, 25
distribute, 38
distribution, 82
dot operator, 26

Einstein, Albert, 5

else, 122
Encapsulated Postscript, 36
end, 71
Enthought Python, 19
EPS, 36
error, 86
eval, 118
existence proof, 7
experimental group, 9
experimental method, 8
Exxon, 51

f statistic, 100
F table, 100
fields, 30
files
    reading, 24, 68
    writing, 24
find, 68
float, 118
floating point numbers, 39
folder, 25
font size, 68
for, 32, 120
Freud, Sigmund, 5
from-import, 26

Galapagos Islands, 6
generalization, 10
get, 29
getColumn, 30, 32
    defined, 32
getRows, 29, 31
    defined, 31
    usage, 29
global, 26
Gosset, William, 82
grand mean, 99, 100
grand total, 100
grand total of scores, 99
grouped, 74
Guardian, 24

Hawthorne effect, 11
Head Start program, 10
help, 44, 52
highlight, 68
hippocampal volume, 16
hippocampus, 16
histogram, 52
HTML background, 68

IDLE, 19, 20
if, 122
import, 26, 41
    for running files, 41
    from, 26
independent variable, 14
inferential statistics, 81
init method, 31
instance variable, 31
instance variables, 30
integers, 39
intercept, 109
internal validity, 12
interrupted time series design, 12
IPython, 41

JPEG, 37

keys, 119

legend, 43
    location, 43
linestyle, 41
list, 31
loc, 43

Macbeth, 67
Many Eyes, 22
map, 32
marker, 41
markeredgecoor, 41
markerfacecolor, 41
markersize, 41
match, 71
match object, 71
matched-subjects design, 114
MATLAB, 58
Matplotlib, 35
    usage, 35
matrix, 112
mean, 47, 87

differences, 81
means, 81
memory, 66
methods, 30, 119
correlational approach, 7
naturalistic observation, 6
Mobil, 51
module, 26
MSB, 101
MSW, 101
multiple correlation, 109
multiple time series design, 13

named arrays, 119
narrative intelligence, 3
naturalistic observation, 6
naturalistic observations, 17
next, 31
non-random assignment, 14
None, 71
nonequivalent before-after design, 14
normal, 65
normal curve, 85
normal distribution, 85
null hypothesis, 90
number, 31
NumPy, 19, 63

objects, 30, 119
one-tailed, 62
one-tailed test, 93
open, 68
open(), 24
operational definition, 10

p levels, 93
p-value, 82
paired, 59
Pearson Product Moment Correlation, 61
Pearson product moment correlation coefficient, 55
plot, 35, 39
saving, 36
showing, 36
usage, 35
PNG, 37
pooled sample variance, 95
population, 90
population variability, 50
posttest, 12
pow, 52
pretest, 12
print, 32
for debugging, 43
probability, 82
distribution, 82
probability levels, 93
pylab, 84
python, 25

r1, 58
r2, 58
random, 84, 87
range, 49, 121
raw mode, 73
re, 70
reactive behavior, 17
read, 68
read(), 24
reader, 28
reading text files, 68
reason, 5
Regression analysis, 110
regular expressions, 70
grouping, 74
match, 71
related, 55
reload, 27, 41
replace, 68
replication, 6
retrospective design, 15
return, 120
rewind, 30
Roasty-Toasties, 9
Romeo and Juliet, 67
run, 41
via import, 41

sample, 87
savefig, 36

scatter diagram, 55
scatterplot, 55
science
  claims, 3
  definition, 3
SciPy, 19, 112
self, 31
Shakespeare, William, 67
shape, 54
show(), 36
significance chance, 62
significance level, 62
significant, 62
sin, 39
Skinner, B.F., 5
slice, 118
slicing, 35
  usage, 38
slope, 46
social psychology, 5
sort, 37
split, 74
spread, 50
SSB, 100
SSW, 101
standard deviation, 50
standard error
  mean, 86
standard error of the mean, 86
standard_normal, 65
start, 71
stderr, 87
stdev, 87
string to number conversion, 118
Student t test, 90
Student's t, 82
Student's t test, 95
sub, 74
subplot, 45
sum, 47
sum of squares, 49
syllogism, 5

t statistic, 100
t table, 93
t test, 95
t-test, 82
  computation, 91
  degrees of freedom, 96
tenacity, 4
textual analysis, 67
The Advancement of Learning, 67
The Essays of Francis Bacon, 67
third-variable problem, 16
time series design, 12
title, 41
total sample size, 99, 100
tuple, 41
two-tailed, 62
two-tailed test, 93

ufunc, 39
universal function, 39

variance, 12, 49, 50
  defined, 49

WinEdt, 40
within-subjects design, 12, 114
World Trade Center, 10

xlabel, 41
XOM, 51

ylabel, 41

zscore, 58