Introduction to the Internet and the World Wide Web

2010 Edition [On-campus Sections]

BY

Greg Baker

Faculty of Applied Sciences
Simon Fraser University
Centre for Online and Distance Education • Lifelong Learning
© Simon Fraser University, 2001–2009

Contents

I Introduction **9**

Course Introduction 11
Course Objectives 11
Course Resources 12
Evaluation 14
About the Course Author 16

II The Web and Web Pages **17**

1 The World Wide Web 19
1.1 Basics of the Internet 19
1.2 Protocols 23
1.3 How Web Pages Travel 25
1.4 MIME Types 26
1.5 Fetching a Web Page 28
Summary 29

2 Markup and XHTML 31
2.1 Making Web Pages 31
2.2 First XHTML Page 32
2.3 XHTML Tags 33
2.4 Why Do Markup? 36
2.5 Another XHTML Page 38
2.6 Attributes 39
2.7 XHTML Reference 40

2.8 Images in HTML 41
2.9 Relative URLs 44
2.10 Choosing Tags 48
Summary 49

3 Cascading Style Sheets 51
3.1 Styles 51
3.2 CSS 52
3.3 External CSS 54
3.4 CSS Details 57
3.5 Why CSS? 61
Summary 63

4 Advanced XHTML and CSS 65
4.1 Validating XHTML 66
4.2 Block vs. Inline Tags 68
4.3 Entities 70
4.4 Generic Tags 72
4.5 Classes and Identifiers 73
4.6 More Selectors 75
4.7 Colours in CSS 76
4.8 Positioning in CSS 79
4.9 Markup for Meaning 84
Summary 87

III Graphics and Design 89

5 Graphics and Images 91
5.1 Graphics and Image Types 91
5.2 Bitmap vs. Vector Images 92
5.3 File Formats 94
5.4 Common Image File Formats 99
Summary 100

6 Design 103
6.1 General Design 103
6.2 Design Principles and XHTML/CSS 108
6.3 Conventions 110
6.4 Readability 112
6.5 Page Design 113
Summary 114

IV Web Programming 115

7 Programming Introduction 117
7.1 What Is Programming? 118
7.2 Starting With Python 119
7.3 Web Programming 121
7.4 Expressions and Variables 123
Summary 125

8 Forms and Web Programming 127
8.1 Forms 127
8.2 Reading Form Input 130
8.3 Handling Types 134
Summary 135

9 More Programming 137
9.1 Python Modules 137
9.2 Making Decisions 138
9.3 String Formatting (Optional) 141
9.4 Debugging 142
9.5 Coding Style (Optional) 144
Summary 145

10 Internet Internals 147
10.1 URLs . . . 147
10.2 Cookies . . . 150
10.3 HTTP Tricks (Optional) . . . 151
10.4 Security and Encryption (Optional) . . . 154
Summary . . . 156

V Appendix 157

A Technical Instructions 159
A.1 SFU Computing Account . . . 159
A.2 CMPT 165 Server Account . . . 160
A.3 More Instructions . . . 160

List of Figures

1.1 How information might get from the SFU web server to a home computer . . . 21
1.2 The conversation that a web client and web server might have when you view a web page . . . 23
1.3 The parts of a simple URL . . . 26
1.4 Some example MIME types . . . 28

2.1 A simple first XHTML page . . . 33
2.2 Display of Figure 2.1 in a browser . . . 36
2.3 A sample web page about houses . . . 38
2.4 Display of Figure 2.3 in a browser . . . 39
2.5 A sample page from the XHTML reference . . . 41
2.6 A web page with an image . . . 43
2.7 Display of Figure 2.6 in a browser . . . 43
2.8 URLs starting at http://www.sfu.ca/~somebody/pics/index.html 47

3.1 The <head> of an XHTML page with an embedded CSS style 52
3.2 A simple XHTML page with a stylesheet applied . . . 55
3.3 A simple CSS style (simple.css) . . . 55
3.4 Figure 3.2 *without* a stylesheet . . . 55
3.5 Figure 3.2 with the CSS in Figure 3.3 . . . 56
3.6 The CSS "Box Model" . . . 58
3.7 Examples of relative units . . . 60

4.1 Some non-valid XHTML with a doctype specified . . . 67
4.2 Part of the validation results of Figure 4.1 . . . 68
4.3 Block and inline tags . . . 69
4.4 Entities required for reserved XHTML characters . . . 71

4.5 Other sample entities 71
4.6 XHTML fragment using a `<div>` 73
4.7 CSS that changes certain classes and identifiers 75
4.8 Effect of the `float` property 80
4.9 Adding a margin around a `float` 80
4.10 An XHTML page with a menu 81
4.11 CSS for Figure 4.10 82
4.12 Screenshot of Figure 4.10 82
4.13 Effect of the `clear` property 83
4.14 Effect of setting `position` to `absolute` 84

5.1 Scaling (a) a vector image and (b) a bitmapped image 93
5.2 Colour dithering 97
5.3 An image with a low-quality lossy compression 98
5.4 Various types of transparency in images 99
5.5 Summary of graphics formats 101

6.1 A design illustrating proximity 105
6.2 A design illustrating alignment 106
6.3 A design illustrating repetition 107
6.4 A design illustrating contrast 109

7.1 Our first Python program 119
7.2 Fetching a static XHTML file 121
7.3 Fetching a dynamic web page 122
7.4 A program that generates a simple web page 122

8.1 Example form on an XHTML page 129
8.2 The display of Figure 8.1 in a browser 129
8.3 A web script that uses the input from Figure 8.1 131
8.4 Example XHTML produced by Figure 8.3 133

Part I

Introduction

Introduction

[This version of the Guide is written for on-campus sections of this course. If you are a distance ed student, you will find that this introduction will not match your section of the course. You can get a copy of the distance ed study guide from the Centre for Online and Distance Education.]

Welcome to CMPT 165: *Introduction to the Internet and the World Wide Web*. This course is an introduction to the Internet and WWW for non-computing science majors. It isn't intended to teach you how to use your computer, starting with "Turn it on." You are expected to have a basic knowledge of how to use your computer. You should be comfortable using it for simple tasks such as running programs, finding and opening files, and so forth. You should also be able to use the Internet.

Course Objectives

Here are some of the goals set out for students in this course. By the end of the course, you should be able to:

- Explain some of the underlying technologies of the World Wide Web and the Internet.
- Create well-designed websites using modern web technologies that can be viewed in any web browser.
- Use graphics appropriately on these pages.
- Design visually appealing and usable websites.
- Create simple dynamic web pages using Python.

Keep these goals in mind as you progress through the course.

Course Resources

Study Guide

This *Study Guide* was originally created for distance offerings of CMPT 165. How closely it is followed by on-campus offerings will depend on the section and instructor.

The optional references for each section are listed at the beginning of the units. You should also look at the key terms listed at the end of each unit.

In some places, there are references to other sections. A reference to "Topic 3.4", for example, means Topic 4 in Unit 3.

In the *Study Guide*, side notes are indicated with this symbol. They are meant to replace the asides that usually happen in lectures when students ask questions. They aren't strictly part of the "course".

Texts

Textbooks and other references can vary depending on your section. Consult your course website or course outline for details.

The optional texts referred to in this *Study Guide* are:

Head First HTML with CSS & XHTML by Elisabeth Freeman and Eric Freeman is an excellent introduction to the concepts of creating web pages with XHTML and CSS, as we will do in Units 2 to 4. It covers much of the same material as the *Study Guide*, but from a slightly different perspective (and a slightly different order). This book is highly recommended.

The Non-Designer's Design Book by Robin Williams (no, not that Robin Williams) is recommended for the material in Unit 6. It is a nice discussion of general design principles and is applicable to any design work, not just web pages. Any edition of this book is acceptable.

Think Python: An Introduction to Software Design by Allen Downey is recommended as a reference for the programming done in the last part of the course. This book is a nice introduction to Python programming.

Online References

There are several online references that are as important as the texts. You can find links to them on the course website, in the "Materials" section.

These resources are very important to your success in this course. They aren't meant to be read from beginning to end like the readings in the *Study Guide*. You should use them to get an overall picture of the topic and as references as you do the assignments.

XHTML Reference: This will be used as a definitive reference for XHTML, which is the markup language used to create web pages. It should be used along with Units 2–4.

CSS Reference: This will be used as a definitive reference for CSS, which is used to restyle web pages. It should be used along with Units 3–4.

Web Materials

The materials that should be common to all on-campus offerings of CMPT 165 can be found at http://www.cs.sfu.ca/CC/165/. You should also find a link to your section's course web site there.

The course website has three main sections:

Materials: This section contains supplementary material directly related to the course content. You can find information about the course references and links that relate to the course material, and download all of the code that is used in this guide if you'd like to try it for yourself or modify it.

You will also find an archive of the course email list.

Technical: This section contains technical information about working with software, or other aspects of the course. This section will be used to supplement information in Appendix A if necessary.

Sections: This area contains links to information for individual offerings of the course. You can find information about your offering including the course supervisor, and your teaching assistants. You will also find the labs and assignments as well as corresponding information on due dates and grading.

Email Communications

The teaching assistants and course instructor will use email to send announcements and tips during the semester. You should read your SFU email account regularly.

CMPT 165 Web Server

Throughout the course, you will be creating web pages. You have web space available on a web server set up specifically for this course. Appendix A and the course website "Technical" section contain information on working with this server.

You must use this server for all web pages in this course.

Evaluation

This information may not apply exactly to all on-campus offerings. Consult your course website or course outline for details.

Labs

There will be weekly labs for this course. You will have registered for a lab section when registering for this course. The lab exercise will consist of some short exercises that you should be able to complete in the allotted time.

Labs won't start until week two or three of the semester, so don't worry about attending the first week.

The labs are intended to keep you up to date on the course material and make sure you get some practice on all of the concepts before having to tackle the assignments.

Since the labs are intended as a learning experience, we don't mind if you discuss them with others in the course. You should do your own work on the labs, but feel free to discuss problems with others.

Assignments

There will also be a few larger assignments in this course. You can find them on the course website. The due dates and instructions on how to submit them are also there.

The assignments are more work than the labs. You will have to figure out more on your own and explore the concepts in the course.

Submitting Your Work

You will submit both the labs and assignments online. You will be sending the web address where you have put your web page(s) and any other information required by the assignment. You must place your labs and assignments in the web space provided for the course. You can find information on uploading the files on the course website in the "Technical" section of the course website.

Exams

Exam details can vary, depending on your section. Consult your course website or course outline for details.

Academic Dishonesty

We take academic dishonesty very seriously in this course. Academic dishonesty includes (but is not limited to) the following: copying another student's work; allowing others to complete work for you; allowing others to use your work; copying part of an assignment from an outside source; cheating in any way on a test or examination.

For information about academic honesty, please refer to Academic Honesty and Student Conduct Policies at http://www.sfu.ca/policies/Students/ .

Copyright

When you create web pages, you must keep in mind other people's copyright. Whenever someone publishes something, it is illegal for others to copy the material except with the permission of the copyright owner.

That means you can't just copy text or images from any website. When you are creating your pages (except where the assignment states otherwise),

you can use images from other sites that indicate that you are allowed to do so or if you have sought *and received* permission. Some sites, particularly clip art sites and other graphics collections, state that you're allowed to use their images for your own sites.

If you do use *anything* from other sites for this course, you *must* indicate where it came from by providing a link to the original source. Failure to do so is academic dishonesty and will be treated as described above.

About the Course Author

I hesitated about writing this section. It seems sort of self-congratulatory—a few paragraphs about who I am aren't so bad, but they set a dangerous precedent. The next thing you know, I'll have $8 \times 10''$ glossy photos of myself and I'll start writing an autobiography.

On the other hand, I'm reminded that students in an on-campus class would learn something about me over the course of the semester. In an effort to duplicate that experience, I relented....

I'm a lecturer at SFU in the School of Computing Science. I started in September of 2000. I finished my M.Sc. in Computing Science at SFU just before that (and I do mean *just*). My undergraduate degree is in Math and Computer Science from Queen's University (Kingston, not Belfast).

I am neither a professor nor a doctor, which causes problems for students who are accustomed to using these titles. For the record, I prefer "Greg".

When I'm not working, I enjoy food and cooking. Unfortunately, I'm getting to an age where enjoyment of food is going to cause buying new pants more often than I like. Vancouver is a great place to be into food. The mix of cultures and available ingredients makes for very interesting cuisine. I'm always interested in good restaurant suggestions if you have any you would like to make.

I'd like to thank Tim Beamish and Rong She, the TAs for the summer 2001 offering of CMPT 165, for helping me prepare this *Study Guide*. I'd also like to thank the students in the summer 2001 offering who suffered through its draft version.

If you have any comments on the course or restaurant suggestions, please feel free to contact me at ggbaker@cs.sfu.ca.

Part II

The Web and Web Pages

Unit 1

The World Wide Web

Learning Outcomes

- Explain how information is transmitted on the Internet.
- Describe how computers are connected on the Internet.
- Describe the way a web page gets to your computer.
- List some services available on the Internet and their protocols.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.

Topic 1.1 Basics of the Internet

As mentioned in the Introduction, we will assume that you have some familiarity with computers and the Internet. In particular, we're going to assume that you have used the Internet and know what the *World Wide Web* (*WWW*) is and how to work with it. You don't have to be an expert, but you should be comfortable using your computer and navigating the web.

Connecting to the Internet

So, what is the *Internet* anyway? Basically, it is a huge network of connected computers. One computer is connected to the next, and they pass information along from one to another. That's really all there is to it—many different computers, all passing information around so it eventually gets to its destination.

When a desktop computer is connected to the Internet, it is probably only connected to one other computer: its *gateway*. The gateway computer might be in a nearby server room (if you're on-campus at SFU, for example), or it might be at your service provider's offices (when you're at home, connected with a cable modem, for example). The gateway will be connected to one or more other computers and will pass along whatever information is sent or received.

When you connect your computer to "the Internet", all that is really happening is that you are getting a connection to a gateway machine. All of your information is passed through the gateway. An *Internet service provider* (or *ISP*) really provides you with a way to connect to a suitable gateway. There are many ways to make that connection, as you will see below.

The gateway and other computers that form the Internet's infrastructure or *backbone* may be connected to many other machines, instead of just one. Their job is to receive information and pass it along in the right direction. They might be computers that are more or less like desktop PCs, or they might be specialized devices called *routers* designed specifically for this job.

For example, Figure 1.1 shows the possible route that a web page might travel to get from SFU's web server (www.sfu.ca) to a computer connected to a cable modem or ADSL. The actual route from www.sfu.ca to a home computer in Vancouver might have about 8 to 10 steps. Routes to other computers on the Internet might have 20 or even more steps.

Computers on the Internet can be any type of computer and can be connected to each other in many different ways. Figure 1.1 illustrates many types of computers, all connected to form part of the Internet. In some sense, the method of connection doesn't matter: they all just pass information from one computer to another. But, the different kinds of connections between computers can pass information at different speeds, and over different distances.

When a home computer is connected to the Internet, it is probably connected by one of the following methods. These can operate over the distance

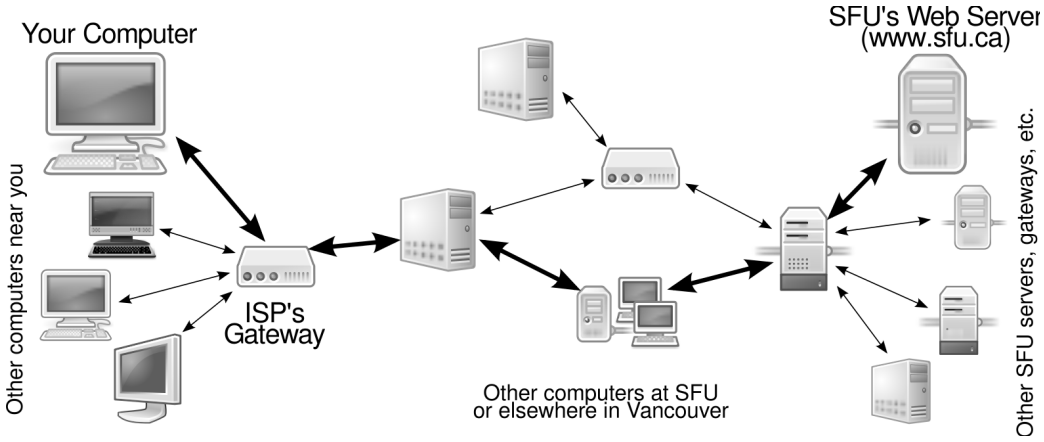


Figure 1.1: How information might get from the SFU web server to a home computer

from your house to the service provider (a few kilometres) and use existing infrastructure (running new wires to everyone's house is too expensive):

***Modem*:** Information is carried to and from your computer over your phone line as sound, the same way a voice call is transmitted. The modem at either end of the phone call translates the sound of the phone call to and from computer data.

***ADSL*:** Information is also transmitted over the phone line but it does not use a traditional "phone call". Information is encoded differently than it is in a modem connection. This lets ADSL transmit data faster than a modem.

***Cable modem*:** With a cable modem, the data to and from your computer is carried over your TV cable.

When a computer connects to a network in an office building or at SFU (or if you have a home network set up), different connection methods are used. These are typically faster than the "home" connections listed above—this is possible because of the shorter distances involved.

***Ethernet*:** The connector for ethernet looks like a fat phone jack. Since it uses wires, it's generally used for desktop computers that don't have to move around. It is several times faster than either ADSL or a cable modem.

***Wi-fi*:** Wireless networking has recently become affordable and more popular. Wi-fi (Wireless Fidelity) is also known as *wireless LAN* (Wireless Local Area Network), *AirPort*, and *802.11*. There are different versions (e.g., 820.11b and 802.11g) with different data transmission rates.

Wi-fi is often used for laptops, which makes it possible to move the machine without having to worry about network cables. Wi-fi is available at SFU and increasingly in other locations.

There are also many other connection types used to make connections between buildings and across cities and over long distances across countries.

Clients and Servers

You may have noticed that in the discussion above, the home computer, the gateway, and the SFU web server are all described only as "computers connected to the Internet". In fact, all of them are connected to the Internet in fundamentally the same way, only the speed of the connection is different. So, why is www.sfu.ca a web server when your computer isn't?

The only real difference is that www.sfu.ca will answer requests for web pages. The gateway and your home computer won't. What makes it possible is the *web server software* installed on the *server*. This program runs all the time and answers requests for web pages. If you ran similar software on your computer, people could get web pages from it as well.

If you did install web server software on your computer, you'd run into some problems. First, the SFU web server has an easy-to-remember name: www.sfu.ca. When home computers are connected to the Internet, they tend to have names like akjx74wuc23nf.bc.hsia.telus.net or h24-84-78-194.vc.shawcable.net—a lot harder to remember and type in correctly. These names also change occasionally, making it even harder to find them.

Second, www.sfu.ca has a very fast connection to the Internet: enough to serve up a lot of web pages at the same time. It's also on all the time, which your computer might not be.

When you want to access web pages, you use a *web browser*. A web browser is one example of *client software*. A web *client* (like Firefox or Internet Explorer) is the software you use to make the request to a web server. The client has to transmit the request to the server, receive the response, and then process the information so you can use it.

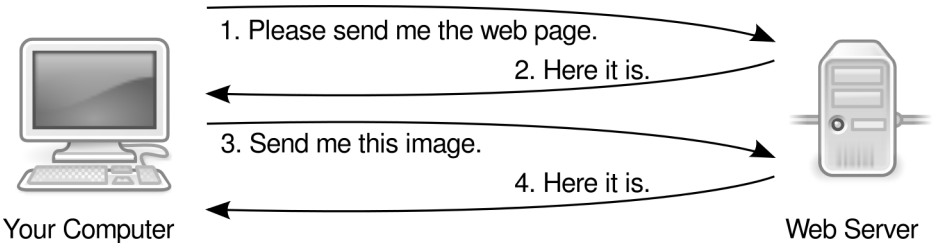


Figure 1.2: The conversation that a web client and web server might have when you view a web page

Think of the process like a phone call: the client "calls" the server, the server answers, and they have a conversation. The client must be set up to make the call; the server must be set up to answer it.

Client and server software let people use computers to transfer information over the Internet. Everything that you transfer is done by interacting with client software for that type of connection. For example, when you load a web page with Firefox (a web client), it will ask the web server for the web page itself and then for any images or other files that it needs to display the page. Figure 1.2 shows the conversation that might happen to load a web page with one image on it.

☞ The term "web server" may be a little confusing here because it's used to refer to both the computer itself (www.sfu.ca is a computer sitting in a closet somewhere in Strand Hall) and the software that's running on the computer (www.sfu.ca runs the Apache web server software).

▶ Find out what kind of connection is used between your computer and your ISP. If you have a home network, does it use ethernet or wi-fi?

Topic 1.2 Protocols

When a client and server talk to each other, they have to agree on how they will exchange the various pieces of information that are required to get the job done. The "language" that the client and server use to exchange this information is called a *protocol*.

For example, when they are exchanging web pages, the client and server need to encode the requests and responses shown in Figure 1.2. They also must be able to indicate errors and other behind-the-scenes messages (like "Page not found" or "Page moved to *here*").

Web pages are transferred using a protocol called the *HyperText Transfer Protocol* (*HTTP*). We will discuss HTTP more in Topic 1.3 and Topic 1.5.

Information on the Internet

There are many different kinds of information that travel over the Internet. The one you're probably most familiar with is web traffic—web pages and all of the graphics, sounds, and other files that go with them. But, a lot of other stuff travels around as well—the web is just one of many ways that information can travel across the Internet.

Here are some other ways information can be exchanged between computers on the Internet that you might be familiar with:

***Email*:** Even if you check your email on the web (with SFU's Webmail or something similar), the mail itself still has to get from the sender to the receiver. If someone at Hotmail sends email to your SFU account, it has to travel from the computers at Hotmail to the mail server at SFU.

***Instant messaging*:** The various instant messaging services each have their own protocols (which is why they don't generally work together). These include ICQ, AOL Instant Messenger, Yahoo! Messenger, and MSN Messenger.

***Peer-to-peer file transfer*:** These methods of file transfer sidestep the traditional client-server model and let people transfer files directly from one client to another. (Technically, the "client" software is performing the duties of a client and a server.) Because of their dubious legality, they tend to come and go, but they have included Napster, Kazaa, and Bittorrent.

***FTP*:** FTP stands for "File Transfer Protocol". It is an older method of transferring files but it is still used to transfer some things.

***Network gaming*:** Games that allow network play generally act as a client; a server is run by the company that produced the game. The client and server exchange information about the game: moves that are made,

changes in the game's "map", and other information to make sure the game works properly for all users.

Each of the examples above uses a different protocol to exchange information. That is why you need different client software for each of them—they need clients that speak different languages.

There are many other services that you might not think of. There are protocols for sharing files and printers on a small network (like Windows File/Print Sharing). There are many other protocols, for things like clock synchronization and remote access to computers, that you might never use directly.

- Are there more examples of protocols/client software that you use?

Topic 1.3 How Web Pages Travel

We have covered web servers and web clients in Topic 1.1. The two computers have to talk to each other in order to deliver web pages.

As noted in Topic 1.2, the "language" a client and server use to talk to each other is called a *protocol*, and the protocol used to transfer web pages is called *HTTP*. You may have noticed that web addresses start with "http://"; this part of the address tells your web browser that you want it to use HTTP to talk to the server. A web server's job is to communicate using HTTP with any clients that connect to it.

URLs

A *URL* (*Uniform Resource Locator*) is the proper name for an Internet address. URLs are also sometimes called *URIs* (Uniform Resource Identifiers). Here are some examples:

http://www.sfu.ca/
http://www.sfu.ca/~somebody/page.html
http://www.w3.org/Addressing/
https://my.sfu.ca/
ftp://ftp.mozilla.org/pub/mozilla/releases/
mailto:somebody@sfu.ca

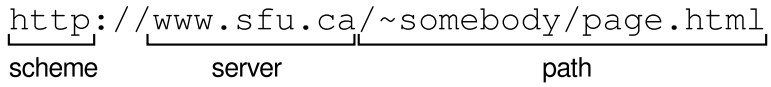


Figure 1.3: The parts of a simple URL

(Actually, all of these are "absolute" URLs. Relative URLs will be introduced in Unit 2.)

The first three URLs are HTTP URLs—they refer to information that is on the web. The others refer to information that you access using different protocols.

The fourth one, https://my.sfu.ca/ works almost like a web page, but it uses a "secure" connection to transmit the data. All of the information passed back and forth between the client and the server is encrypted so none of the computers in between can eavesdrop. HTTPS is often used for sensitive information like banking, passwords, and email.

The last two items in the list are URLs for information accessed by other protocols. They are an FTP URL (information on an FTP server) and an email URL (a link that will let you send an email to that address).

The protocol that should be used to access a particular URL is indicated by its *scheme*. The scheme is the part of the URL before the colon (:). For web (HTTP) URLs, the scheme is "http" or "https" for secure web traffic. The URL schemes "ftp" and "mailto" indicate FTP and email URLs. There are many other less common URL schemes used by specific applications.

For HTTP and FTP URLs, the first part after the scheme indicates the server that should be contacted to exchange information. The rest of the URL after the server is the *path* of the file. We will discuss the other parts of an HTTP URL in Topic 10.1.

- Find the URL of some web pages that you visit often (in the location bar of Firefox). Identify the scheme, server, and path of each.

Topic 1.4 MIME Types

Any type of data can be transferred over HTTP, including web pages themselves (which are written in a language called *HTML*, as we will see in Unit 2). Graphics on web pages are also sent by HTTP (in formats called GIF, PNG,

and JPEG, which will be discussed in Unit 5). All of the other files you get from the web are also sent by HTTP: video, audio, Microsoft Office documents, and so on.

When your web browser receives these files, it has to know what to do with them. For example, it treats graphics data very differently from a text file. The browser also has to know what program to open for files it can't handle itself, like Acrobat files, MS Office documents, and MP3 audio files.

You may be used to looking at the *file extension* to figure out what type of file you have. For example, MS Word documents are usually named something like essay.doc; the .doc indicates that it is a Word document. Other extensions include .html or .htm for web pages and .pdf for Acrobat files.

Unfortunately, the file extension can't be used on the web to decide the type of the file. For one thing, some operating systems don't use file extensions, so we don't know for sure they will be available. It's also possible that the browser wouldn't even know a file name when the data is sent. We will also see other reasons later in the course.

Instead of using file extensions, the type of data sent by email or HTTP is indicated by a *MIME type*. MIME stands for "Multipurpose Internet Mail Extensions".

MIME types were first used for email attachments (thus the name). Like files transmitted over the web, email attachments need to have their type indicated, and you can't rely on the extension to tell you. So, it makes sense to use the same solution in both places.

The MIME type indicates what kind of file is being sent. It tells the web browser (or email program) how it should handle the data. MIME types are made up of two parts. The first is the *type*, which indicates the overall kind of information: text, audio, video, image, and so on. The second is the *subtype*, which is the specific kind of information.

For example, a GIF-format image will have the MIME type image/gif—indicating that it is an image (which might give the browser a hint what to do with it, even if it doesn't know the subtype) and the particular type of image is GIF. Most web browsers know how to display GIF images themselves, so they wouldn't have to use another application to handle it. There are more examples in Figure 1.4.

Microsoft's Internet Explorer browser doesn't handle MIME types correctly. It often ignores them and then tries to guess the type itself (and is sometimes wrong). This is one of the reasons that we suggest you don't use IE for this course.

MIME type	File contents	How it might be handled
text/html	HTML (web page)	display in browser as a web page
application/pdf	Acrobat file	open in Adobe Acrobat
application/msword	MS Word document	open in Word
image/jpeg	JPEG image	display in browser
audio/mpeg	MP3 audio	open with iTunes
video/quicktime	Quicktime video	open with Windows Media

Figure 1.4: Some example MIME types

When you put a file on a web server, the server *usually* determines the MIME type based on the file's extension. This isn't always the case—in fact, later in the course, we will see cases where we have to set the MIME type manually. For the moment, the MIME type will come from the file's extension. However, you should remember that the type and the extension are different things.

- You can determine the MIME type of a file with Firefox in the "Page Info" window (in the "Tools" menu). Try going to a web page and have a look—you should see text/html.
- When you're viewing an image (right-click and select "View Image"), you will see the image's MIME type in the "Page Info" window. Try it out with a few images.

Topic 1.5 Fetching a Web Page

Let's look back at Figure 1.2, now that we know a few more details about the process.

Suppose that the web page that is being requested is at the URL http://cmpt165.csil.sfu.ca/~you/page.html and that one of the graphics on the page is the JPEG image at http://cmpt165.csil.sfu.ca/~you/pic.jpg .

We can now give a more detailed description of what happens in the conversation indicated by each of the four arrows in Figure 1.2.

1. The web browser contacts the server specified in the URL, cmpt165.csil.sfu.ca. It asks for the file with path /~you/page.html .

2. The server responds with an "OK" message, indicating that the page has been found and will be sent. It indicates that the MIME type of the file is text/html—it's an HTML page. Then it sends the contents of the file, so the browser can display it.
3. The browser notices that the web page contains an image with URL http://cmpt165.csil.sfu.ca/~you/pic.jpg . It contacts cmpt165.csil.sfu.ca and asks for the path /~you/pic.jpg .
4. The server again responds with an "OK" and gives the MIME type image/jpeg, which indicates a JPEG format image. Then it sends the actual contents of the image file.

Once the web browser has the HTML for the web page and all the graphics that go with it, it can draw the page on the screen for you to see.

Summary

This unit gives you a brief introduction to how the WWW and the Internet work. By now you should have a better understanding of what goes on behind the scenes when you use the Internet for various tasks. This information will help you later in the course and should be useful anytime you're using the Internet and things go wrong—you can often fix or work around problems if you understand what's going on.

Key Terms

- World Wide Web (WWW)
- Internet
- gateway
- Internet service provider (ISP)
- client
- server
- web server
- client software
- web browser
- protocol
- HTTP
- URL
- scheme
- path
- MIME type
- subtype

Unit 2

Markup and XHTML

Learning Outcomes

- Create web pages in XHTML with a text editor, following the rules of XHTML syntax and using appropriate XHTML tags.
- Create a web page that includes links and images using XHTML.
- List some common XHTML tags.
- Use relative URLs to refer to other resources on a website.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.
- Browse the XHTML reference pages linked in the "Materials" section of the course website.
- (optional) Review *Head First HTML*, Chapters 1–4.

Topic 2.1 Making Web Pages

Web pages are described to your computer using a *markup language* called XHTML. *XHTML* stands for "eXtensible HyperText Markup Language". Older versions of the markup language are called *HTML*. HTML and XHTML are quite similar; we will only cover XHTML (version 1.0) in this course. (We will come back to what a "markup language" is later.)

XHTML is used to describe all of the parts of a web page in a way that computers (and computer programs) can understand. In particular, a program called a *web browser* is used to display XHTML (or HTML). Web browsers include Mozilla Firefox, Internet Explorer, and Safari. These programs can read XHTML from the Internet (using HTTP) or from the computer's hard drive.

This brings up a very important point that you need to remember when creating web pages: you *always* rely on the user's web browser to display your web pages. There may be some differences in the way your XHTML is displayed, depending on their browser, operating system, and other factors. This is unavoidable and just part of making web pages.

Editing XHTML

XHTML (and HTML) is created with a *text editor*. You might not have used a text editor before: some include TextPad, Notepad, and BBEdit. All of these programs are just a way to enter plain text. They do not include options for formatting, graphics, or other features. So, word processors (like Microsoft Word) are not text editors.

You can find links to download a text editor on the course web page. If you use Windows, Notepad will be installed, but you will find it much easier to work with a more full-featured text editor.

You will use a text editor to edit not only XHTML, but CSS and Python later in the course. Text editors are also used to edit programs in other languages like Java, C, and others.

Web pages can also be created with graphical editors like Dreamweaver and FrontPage. These do not generally do a good job of creating XHTML. We have already seen one reason: XHTML may be displayed differently, depending on the user's browser, so it doesn't make sense to think of it as having a single "look". Graphical editors give you this (false) impression. They can **not** be used in this course.

Topic 2.2 First XHTML Page

Once you have a text editor that you can use on your computer, you can start creating XHTML pages (and other text-based documents, of course). Start

```
<html>
<head>
<title>Page Title</title>
</head>
<body>
<h1>Page Heading</h1>
</body>
</html>
```

Figure 2.1: A simple first XHTML page

your text editor program—you will use this program to create XHTML.

So, what do you have to type into the editor to create a web page? Start by having a look at Figure 2.1. It contains a (very simple) XHTML page.

The text you see in Figure 2.1 is XHTML code. Code like this is sent from the web server to the web browser whenever you view a web page. When creating an XHTML page with a text editor, you type code like this and save it with a file name ending in .html.

- This would be a good time to start experimenting on your own. Open up your text editor and type in the text you see in Figure 2.1. Save it as first.html (or any other name, just make sure it ends in .html).
- View the page with a web browser by selecting "Open File" from the File menu and opening the file you just saved.

Topic 2.3 XHTML Tags

Some parts of the XHTML file are instructions to the browser, such as `<title>` and `<h1>`. These are *XHTML tags*. These aren't seen by someone viewing the page in a browser—they are just used to tell the browser how the page is structured. The browser uses the tags to decide how to display information on the screen.

The tags are the "markup" that make XHTML a *markup language*. There are other markup languages (that we won't discuss in this course) that have other types of markup codes. But, all markup languages have some kind of commands like the tags in XHTML that instruct the *viewer* on how the document should be displayed.

In XHTML (and HTML), tags are used to describe the parts of the document. A web browser is used as the viewer, to display the page so it can be read.

In XHTML, all formatting is done with tags. Spacing, blank line, and other formatting in the text don't matter. The rule that web browsers follow is: Any number of spaces, returns, and tabs are displayed like a single space.

For example, these four examples of an `<h1>` are exactly the same as far as the browser is concerned:

```
<h1>Page Heading</h1>
<h1>Page        Heading</h1>
<h1>Page
       Heading</h1>
<h1>
Page Heading
</h1>
```

The way you space out the tags and contents of your XHTML files is up to you. Try to be consistent and make it easy to find the parts of the page you need, as is done in this guide.

All tags in XHTML must be in lower case. So, `<body>` is okay, but `<BODY>` and `<Body>` aren't. Older versions of HTML allowed uppercase tags, but XHTML doesn't.

- ▶ Try editing first.html, adding or removing extra spaces. Reload the page in the browser—its appearance shouldn't change.

Closing Tags

You might have noticed that all of the tags in Figure 2.1 are in pairs. For example, the `<title>` tag is followed by `</title>`. The first version (like `<title>`) is called the *opening tag*. The second (like `</title>`) is the *closing tag*. The stuff between the opening and closing tags are the tags' *contents*.

For example, in Figure 2.1, the `<title>` tag contains the text "`Page Title`". The `<html>` tag contains all of the other text in the figure.

The opening and closing tags have to be "nested" together properly. That is, the last tag opened must be the first one closed. For example, these pairs of tags are nested properly:

```
<em><a></a></em>
<a><em></em></a>
```

These are not:

```
<em><a></em></a>
<a><em></a></em>
```

Another way to look at it: any time you have two sets of overlapping tags, one must be *entirely inside* the other. One cannot be partially inside and partially outside the other.

Some XHTML Tags

Here are (all but one of) the XHTML tags we saw in Figure 2.1:

`<html>...</html>`: This tag wraps the entire XHTML page. It is the first tag opened and the last one closed. (It is still just "`html`", even though we are using XHTML, not HTML. The tag wasn't changed when the markup language was renamed.)

`<head>...</head>`: This tag is used to hold information *about* the page. We we have only seen one tag so far that is allowed in `<head>`.

`<title>...</title>`: This is the one tag that is allowed inside the `<head>`. It gives the page title—the browser usually displays the title in the window's title bar. For example, the display of Figure 2.1 might look like this in Firefox:

`<body>...</body>`: This contains the actual contents of the page: the stuff you see in the main window of the browser.

These tags will be in every XHTML page you create. They must be arranged as you see in Figure 2.1: `<head>` before `<body>`, `<title>` inside of `<head>`, and so on.

Most of the other additions we make to pages will go in the `<body>`, since it holds the main contents of the page. We will only see a few more tags that go in the `<head>`.

The one other tag that appears in Figure 2.1 is `<h1>`. This is the first tag we have seen that is allowed in the `<body>` of an XHTML page.

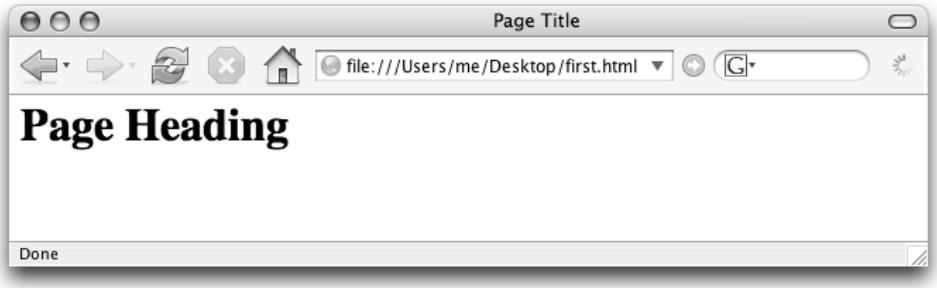


Figure 2.2: Display of Figure 2.1 in a browser

The `<h1>` is used to indicate a *heading*. This is the "largest" heading, usually the main title for the page. Generally, each page should contain a single `<h1>` at the top of the page, with the same text as the `<title>`.

You can see a screenshot of the XHTML page from Figure 2.1 in Figure 2.2. If you type the code in a text editor, save it as an .html file, and open that with your web browser, you should see something similar.

Let's look at one more tag before we move on: the `<p>` tag. The `<p>` tag is used to indicate a paragraph. Each paragraph on your web pages should be wrapped in a `<p>...</p>`. For example, we could edit the `<body>` of Figure 2.1 to:

```
<h1>Page Heading</h1>
<p>Welcome to my page.  This is the first paragraph.</p>
<p>
This is the second paragraph.
</p>
```

- Try editing first.html so its body is as above. Reload the page in the browser so you can see the changes.

Topic 2.4 Why Do Markup?

You may be asking at this point: why do we have to create web pages with this markup language? As mentioned earlier, there are graphical tools such as FrontPage and Dreamweaver that can be used to create web pages.

As mentioned in Topic 2.1, web pages **will** look different for different users. This depends on several factors, including:

- The browser being used. Different browsers (Firefox, Safari, Internet Explorer) will have differences in the way pages are displayed. Even different versions of browsers have different capabilities. The operating system being used (Windows, MacOS, Linux) and the fonts installed on the computer can also affect the appearance.
- The size of the browser window. Of course, the number of words that fit on a line (and other aspects of the formatting) depend on the size of the browser window. Remember that some users don't keep their browser "maximized" to the full size of their screen, preferring to be able to view a web page and another program at the same time.
- The size of the font the user prefers. Some users need larger fonts (because of poor eyesight or a very high resolution monitor) or like smaller fonts (to get more text on the screen at once) and set their browser's defaults accordingly.
- The format of the browser. Don't forget that people can view web pages on devices other than regular computers. For example, many phones and PDAs are capable of viewing web pages. Ultra-mobile PCs, which have screens smaller than regular laptops, are also becoming increasingly popular. There are also browsers that will read web pages to visually-impaired users.

All of these factors are out of your control as a web page author. You need to make web pages that are flexible enough to adapt to these different circumstances.

When working with a graphical web page editor, it is difficult to keep this reformatting in mind, since you see only one representation of the page. It is very easy to create pages that don't reformat well when necessary. The XHTML code they generate is also not generally well suited to the wide variety of devices that access the web.

XHTML (or HTML) produced by-hand is typically much better-suited to being used in a variety of ways on the web. When creating XHTML yourself, you can use more descriptive tags (like the paragraph tag), because you know what kind of content you are creating, and not just the way it should appear.

```
<html>
<head>
<title>Finding Housing</title>
</head>
<body>
<h1>Finding Housing</h1>

<h2>Types of Housing</h2>
<p>There are many types of housing you can either
<a href="http://en.wikipedia.org/wiki/Renting">rent</a>
or buy.  For example,</p>
<ul>
  <li>a detached house</li>
  <li>an apartment</li>
  <li>a townhouse</li>
</ul>

<h2>Other ways</h2>
<p>You can also find housing in other ways.  Many students live
with their parents or in residence.</p>
</body>
</html>
```

Figure 2.3: A sample web page about houses

As you learn about XHTML (and later CSS), you will see techniques that can be used to create pages that can be used in a variety of circumstances, by many types of users.

Topic 2.5 Another XHTML Page

We will now look at another example XHTML page. Have a look at Figure 2.3. It is similar to Figure 2.1, but contains a few more tags. When it is displayed in a browser, it might look like Figure 2.4

There are a few new tags in Figure 2.3:

`<h2>...</h2>`: This tag is used to represent a "second level" heading. This tag is used to indicate sections within the page. There are other levels of headings as well: `<h3>` is used to represent subsections inside of the `<h2>` sections.

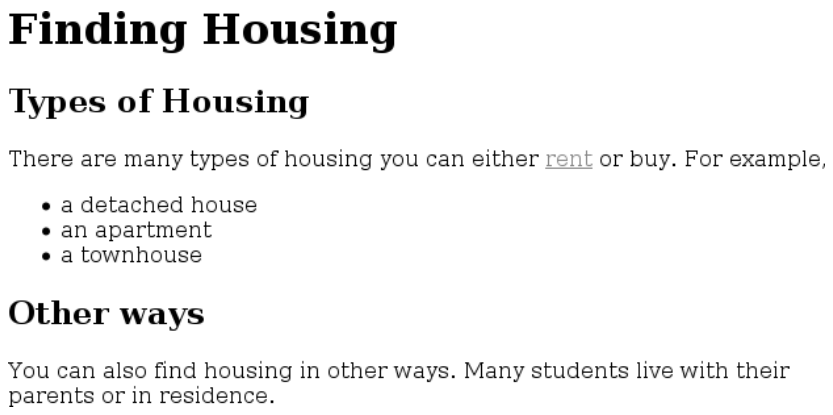


Figure 2.4: Display of Figure 2.3 in a browser

`<ul>...</ul>`: This tag represents an *unordered list*. That is, a list of things where their *order* is not important. In Figure 2.3, the types of housing could have been presented in any order, so this tag is appropriate. The *only* thing that can go inside `<ul>` are one or more `<li>` tags.

`<li>...</li>`: This tag is used for a *list item*. It can *only* go inside `<ul>` or one of the other list tags.

`<a>...</a>`: The `<a>` tag is used to create a link to another page. When this link is followed, it will take the user to the URL http://en.wikipedia.org/wiki/Renting. We will see more details in the next topic.

We will see a few more tags as we go on. You should also find more in the XHTML reference on the course website. This reference is described in Topic 2.7.

- Either type in the code from Figure 2.3 or download it from the course website. Make some changes in your text editor and load the page in a browser. Make sure you know how all of the tags are being displayed.

Topic 2.6 Attributes

Let's have a closer look at the link tag in the last example:

```
<a href="http://en.wikipedia.org/wiki/Renting">rent</a>
```

The `href` part is an *attribute*. An attribute is used to modify a tag in some way. For `<a>`, the `href` attribute gives the destination URL for the link. (The word "attribute" here should be pronounced with the stress on the first syllable, not on the second as in the verb that is spelled the same way.)

The attribute's *value* is given after the equal sign, and it must be in quotes. In the example, the value is `http://en.wikipedia.org/wiki/Renting`.

Attributes are always put in the *opening* tag, after the tag name, and before the `>`. If you want to specify more than one attribute, they are separated by a space. For example:

```
<a href="page.html" class="internal">...</a>
```

The attributes that each tag can have are described in the online XHTML reference, described in the next topic.

Topic 2.7 XHTML Reference

There are many more XHTML tags and attributes than have been mentioned here, and you should explore them further. You can visit the definitive reference in the "Materials" section of the course website.

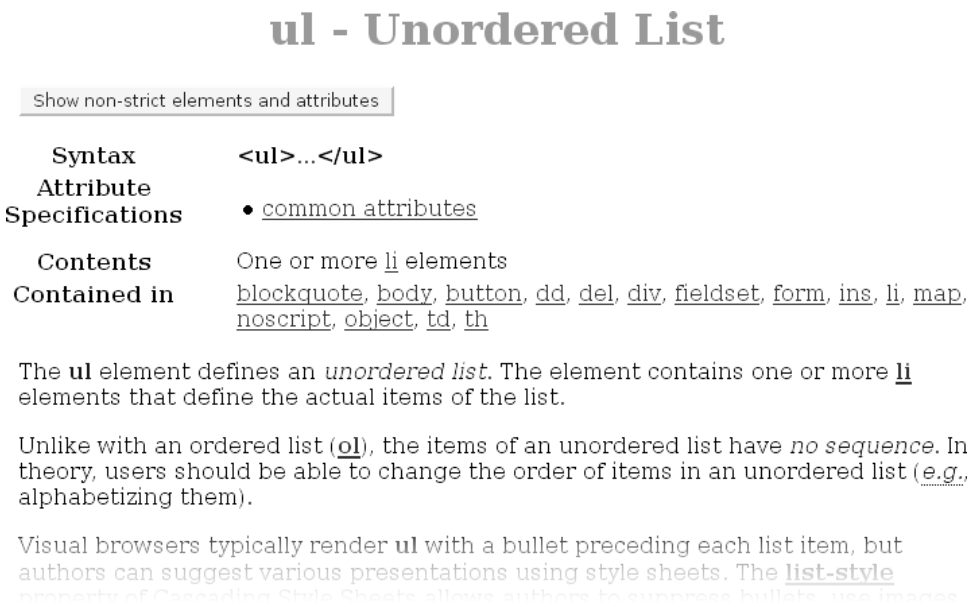
Figure 2.5 shows part of the reference page for the `<ul>` tag.

The **Syntax** line gives the general usage of the tag. Empty tags (which will be covered in Topic 2.8) will be displayed with the short-form closing tag. The **Attribute Specifications** lines give a list of possible attributes for this tag and their values.

The **Contents** line indicates what you may put inside the tag. In Figure 2.5, we see that "one or more" `<li>` tags can go in a `<ul>`. The **Contained in** line gives the tags that they can be placed in. Figure 2.5 indicates that `<ul>` can be placed (directly) inside of the `<blockquote>`, `<body>`, ... tags.

The text below these lines describes what the tag is for, how it should be used, and its attributes.

As you explore these pages, you will notice the "Show non-strict..." button at the top of the page. This will display tags and attributes that are *deprecated*, which means that using them is discouraged because they will be removed from future versions of XHTML. There are better ways to accomplish the same task, often with style sheets, which are discussed



ul - Unordered List

Show non-strict elements and attributes

Syntax	...
Attribute Specifications	• common attributes
Contents	One or more li elements
Contained in	blockquote, body, button, dd, del, div, fieldset, form, ins, li, map, noscript, object, td, th

The **ul** element defines an *unordered list*. The element contains one or more **li** elements that define the actual items of the list.

Unlike with an ordered list (**ol**), the items of an unordered list have *no sequence*. In theory, users should be able to change the order of items in an unordered list (*e.g.*, alphabetizing them).

Visual browsers typically render **ul** with a bullet preceding each list item, but authors can suggest various presentations using style sheets. The **list-style**

Figure 2.5: A sample page from the XHTML reference

in Unit 3. You should just leave these tags and attributes hidden for this course.

- Have a look at the XHTML reference pages and familiarize yourself with some of the tags that are available.
- Modify one of the XHTML files you have been working on to use some other tags.

Topic 2.8 Images in HTML

We will discuss creating and editing images in Unit 5. For the moment, we will just look at how to put an image that has already been created on a web page.

Images are inserted with the `<img>` tag. This tag has two *required* attributes: that is, an `<img>` tag without them is illegal. The first required attribute is `src`, which is used to indicate the URL where the image can be

found. The second is `alt`, which is used to specify alternate text for the image.

The alternate text is used in several situations. It can be displayed if the image cannot be loaded for some reason (network congestion, bad URL, deleted file, etc.), if the image hasn't been downloaded yet, or if the browser does not support images. When you are creating `alt` text, you should try to write text that gives users as much of the meaning of the image as possible.

☞ Browsers that don't support images are rare, but there are some still in use. For example, the visually impaired, who cannot use a graphical browser, often use a text browser and a speech synthesizer. Also note that search engines will look at the `alt` text for your images, not at the images themselves.

The `<img>` tag is *empty*, so the closing tag is unnecessary. You insert an image in the following way:

```
<img src="house.png" alt="picture of a house" />
```

Empty tags are ones that aren't allowed to have any contents. That is, you aren't allowed to put *anything* between the opening and closing. So, if you were going to use the closing tag as before, it would have to look like this:

```
<img src="house.png" alt="picture of a house"></img>
```

The "`/>`" closing is a short form that can be used with empty tags—the browser should treat the two lines of code above in exactly the same way. In this guide, we will be using the short form where possible.

The `<img>` tag must be inside of a paragraph, heading, or other similar tag. It cannot be placed directly inside the `<body>`.

☞ `<img>` actually must be inside any *block-level tag*, which will be explained in Topic 4.2.

Figure 2.6 shows a web page with an image included. In this case, the image file house.png must be placed in the same directory as the .html file, so it will be found by the browser.

You can specify the size of the image, in pixels, using the `height` and `width` attributes. (If you're unsure what a "pixel" is, ignore this until Unit 5 and then come back here.) With this information, the browser can display the page before the images are downloaded, since it knows how much space it needs to leave for them. As a result, your page will be displayed faster, especially for people with a slow connection, and it is a good idea. So, we might do something like this:

```
<html>
<head>
<title>Finding Housing</title>
</head>
<body>
<h1>Finding Housing</h1>

<h2>Types of Housing</h2>
<p>There are many types of housing you can either
<a href="http://en.wikipedia.org/wiki/Renting">rent</a>
or buy.  For example,</p>
<ul>
  <li>a detached house
      <img src="house.png" alt="drawing of a house" /></li>
  <li>an apartment</li>
  <li>a townhouse</li>
</ul>

<h2>Other ways</h2>
<p>You can also find housing in other ways.  Many students live
with their parents or in residence.</p>
</body>
</html>
```

Figure 2.6: A web page with an image

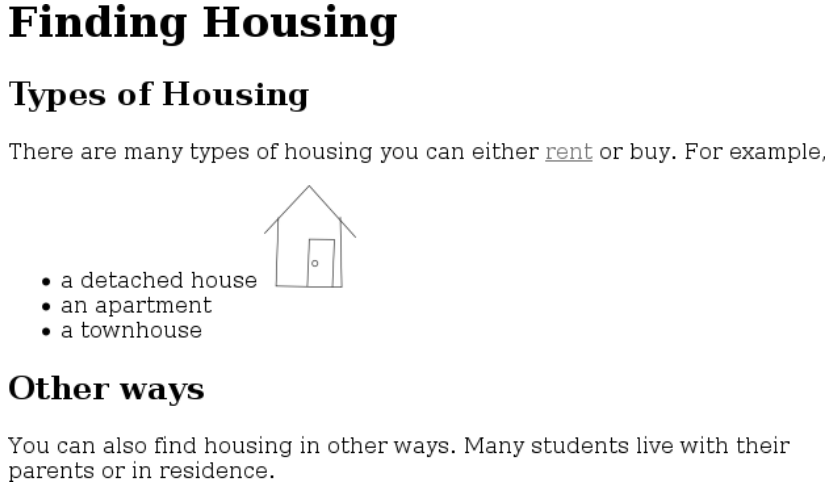


Figure 2.7: Display of Figure 2.6 in a browser

```
<img src="house.png" alt="picture of a house"
width="70" height="78" />
```

When an image is inserted in this way, the browser treats it like a character (admittedly, a funny-shaped one) in the current paragraph. If you want an image that "floats" along the left or right margin, you should use style sheets (see Unit 3).

- Put an image on one of the pages you have been editing. You can download an image from any website for this (right-click or shift-click on the image and select "Save"). Have a look at the discussion of copyright in the Introduction before you start taking images from outside sources.

Topic 2.9 Relative URLs

We have now seen two places in XHTML that we need to indicate a URL: the destination of a link and the source for an image:

```
<a href="http://www.google.com/">search the web</a>
<img src="http://www.sfu.ca/sfu.gif" alt="SFU crest" />
```

In either case, you just need to specify a URL where the browser can find the information it needs.

It is quite common to link to a page or reference an image on the current site. In these cases, it would be quite cumbersome to type out the full URL of the page/image *every* time, since they are usually quite long. It would also be a problem if you wanted to move a website to another location—you would have to fix every single link and image's URL.

Both of these problems are addressed by *relative URLs*. Instead of having to type the full URL every time, with relative URLs you can just indicate the *changes* from the current URL. Relative URLs don't start with a scheme name (http://) and don't specify a server either.

The URLs we have seen previously (that start with a scheme like http:// and specify a server) are called *absolute URLs*. Absolute URLs indicate everything about the location of a document that is needed to fetch it and don't depend on the browser knowing some "current" location.

There are a couple of ways to form relative URLs. We will build complexity in the next few examples.

File Name Only

In the simplest form of a relative URL, you specify only a file name. For example, you might have noticed that in the image tag example in Topic 2.8, we specified only `src="house.png"`, not a full URL. This was a relative URL.

In this case, the browser looks for a file with that name *in the same directory* as the current page. To display the image as you see in Figure 2.7, the house.png file was placed in the same directory as the XHTML file, and the page was opened in a browser.

This can be done with a link as well:

```
<a href="page2.html">the next page</a>
```

When the user clicks this link, they will be taken to the page in page2.html, in the same directory as the current file.

Directory and File Name

Having all of your files in a single directory is fine for small sites, but for larger sites, you will want to organize your files into directories. Relative URLs can be used to navigate between directories, as well as to files in the same directory.

You can give a directory name and a file name, separated by a forward slash (/). The web browser will move into the given directory and look for the named file.

Backslashes (\) are never used in URLs. They occasionally find their way onto web pages because they are common in Windows path names, but this is an error.

For example, have a look at this link:

```
<a href="section1/page2.html">
```

When the user clicks this link, the browser will look for the file page2.html in the directory section1. The directory section1 must be in the same directory as the current page. So, you could read `section1/page2.html` as "move into the section1 directory and look for page2.html".

Another example in an image tag:

```
<img src="party/IMG0001.JPG" alt="us at a party" />
```

It is often convenient to put all of your images in a directory (or directories). In this case, the browser looks in the party directory and finds the image IMG0001.JPG.

Note that file names are case-sensitive. For example, IMG0001.JPG and IMG0001.jpg are considered to be *different files* when you put them on a web server. Make sure the cases of your file names and URLs match.

Moving Up

The last possibility for relative URLs that we will discuss is how to move up a level through the directory structure. That is, if the current file is in a directory, and you want to refer to a file in the directory above it.

To do this with a relative URL, use the special directory name "..". The two-dots directory name has a special meaning: move up a level.

For example, consider this link:

```
<a href="../menu.html">
```

This will look for the file menu.html in the parent directory.

The same can be done with the image tag:

```
<img src="../logo.png">
```

Combining

These forms can be combined and repeated. For example, if you want to use the landscape.jpg file in the directory tofino, which is itself in the vacations directory, you can use an image tag like this:

```
<img src="vacations/tofino/landscape.jpg">
```

Similarly, you can move up two directory levels:

```
<a href="../../menu.html">
```

URL	Destination
img.png	http://www.sfu.ca/~somebody/pics/img.png
../file.html	http://www.sfu.ca/~somebody/file.html
../../test.html	http://www.sfu.ca/test.html
dir/img.png	http://www.sfu.ca/~somebody/pics/dir/img.png
http://www.cs.sfu.ca/	http://www.cs.sfu.ca/ (absolute URL)

Figure 2.8: URLs starting at http://www.sfu.ca/~somebody/pics/index.html

Another Perspective

If you would like another way to understand relative URLs, you can think about what the browser does to build the new URL. This is the same information that was presented above, but it's another way to think about it.

The destination of a link with a relative URL depends on the URL of the page it's on. Suppose we are currently looking at the page http://www.sfu.ca/~somebody/pics/index.html. Assume that the examples of relative URLs below are on that page.

Here's what the browser does to create the destination URL:

1. Start by dropping the file name from the current URL (everything after the last slash), so we have http://www.sfu.ca/~somebody/pics/.
2. For every ../ at the start of the relative URL, drop another directory from the end of the URL. So, if there is one, we'd have: http://www.sfu.ca/~somebody/.
3. Put the rest of the relative URL on the end of the current URL.

Figure 2.8 contains examples of URLs that could be links on the page http://www.sfu.ca/~somebody/pics/index.html and the URL the user would be taken to if they clicked on it.

All of the forms of relative URLs can be used in links, images, and anywhere else you need to specify a URL in XHTML.

Whenever possible, you should use relative URLs on your web pages. Then, if you move your page to a different location (say, from your hard drive to your web space), the links will still work. It also takes less typing.

☞ You may have noticed URLs with no file name on the end. For example, "http://www.apple.com/" or "part1/" (absolute and relative URLs, respectively). What happens when a browser requests a URL like this depends on the server, but typically the server looks for the file index.html and returns its contents. If you name the front page of a site index.html, you won't have to specify a file name in URLs to it.

▶ Put some links and image tags on a web page. Try both an absolute and a relative link. Upload these to the course web server and make sure the URLs still work as expected.

Topic 2.10 Choosing Tags

When creating an XHTML page, you should use tags that have a *meaning* that is as close as possible to the contents you are marking up. You shouldn't worry (yet) about the appearance of the page. Markup that is done this way is called *semantic markup*.

☞ Semantic: having to do with meaning. So, semantic markup is markup that concerns itself with the meaning of the contents (as opposed to their appearance).

For example, if you are giving instructions to complete some task, you will probably have a list of instructions. Since you have a *list*, either the `<ol>` or `<ul>` tag is appropriate. Since it matters what order the steps appear in, the `<ol>` (ordered list) tag should be used, with an `<li>` (list item) enclosing each individual step.

If you want to highlight an important word in a sentence, you can use either the `<em>` (emphasized text) or `<strong>` (strongly emphasized text), depending on the level of emphasis you want. For example, the word "meaning" in the first sentence in this topic is emphasized—the `<em>` tag would be appropriate for this content in XHTML.

Have a look at the index of the XHTML reference. You don't have to memorize all of the tags, but you should have an idea of what's there, so you can find the right tag when you need it.

You should not use the tags that have no meaning, but only specify an appearance, like `<b>` (bold text) and `<small>` (small text).

We will discuss more details and reasons for doing this in Topic 3.5, when we have a more complete picture of both XHTML and CSS.

▶ Look back at some of the XHTML pages you have created. Do you see any places where a more semantically meaningful tag could have been used?

Summary

After doing this unit, you should be comfortable creating a simple web page using a text editor. You should know some of the basic tags and be learning more.

Key Terms

- markup
- XHTML
- tag
- opening/closing tags
- attribute
- link
- absolute and relative URLs
- semantic markup

Unit 3

Cascading Style Sheets

Learning Outcomes

- Use CSS to style XHTML content.
- Describe commonly used CSS properties.
- Use CSS to implement a simple visual design.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.
- Browse the CSS reference pages linked in the "Materials" section of the course website.
- (optional) Review *Head First HTML*, Chapters 8, 10.

Topic 3.1 Styles

So far, we haven't really had any ways to control the appearance of XHTML pages. All we could do was use appropriate tags and hope that the browser displayed it appropriately. All of the tags we used specified the meaning or purpose of its contents (a heading, list, etc.), not its appearance (red, bold, etc.). We relied on the browser to associate an appearance with the meaning we gave.

```
<head>
<title>Sample page with an embedded style</title>
<style type="text/css">
  h2 {
    text-align: center;
  }
  blockquote {
    font-size: smaller;
    font-style: italic;
  }
</style>
</head>
```

Figure 3.1: The `<head>` of an XHTML page with an embedded CSS style

This was deliberate: XHTML was designed (as were later versions of HTML) to specify only *content*, not visual appearance.

In order to specify the appearance of a page, we use a *style* (or *stylesheet*). The style will give information about how each type of content should look.

As a (non-web-based) example of style information, have a close look at the formatting of this guide. You may have noticed that each topic heading has a particular font size, with a line above, left-justified topic number, and right-justified title. Each newly introduced term (like "style" in the previous paragraph) is italicized. Both of these are examples of style information that is specified with a style: the headings and new terms could be styled in many ways; with a style, we choose these appearances.

Similarly, when using XHTML, we will be able to use styles to specify information like "all `<h2>`s will be bold and in an 'extra-large' font", or "every `<dfn>` will be italicized".

Topic 3.2 CSS

For XHTML (and HTML) pages, style information is given in a language called *CSS* (*Cascading Style Sheets*). CSS can be embedded in XHTML files or (as we will see in the next topic, in a separate file).

The `<style>` tag is used to embed CSS style information in an XHTML page. It is placed in the `<head>` of the page. Have a look at the code in Figure 3.1. It contains the `<head>` of an XHTML page.

The CSS code in this example is contained in the `<style>` tag. It contains two *rules*. Let's look at the first one:

```
h2 {
  text-align: center;
}
```

This rule affects all `<h2>` on the page. The "`h2`" in the CSS is called the *selector*: it selects what will be modified.

The rest of the rule (inside the `{...}`) indicates the changes to be made. The changes are specified by one or more *declarations*. The declaration in this example specifies that each `<h2>` will be centred (instead of left-aligned, which is the default).

The second rule in Figure 3.1 contains two declarations and affects any `<blockquote>`s on the page (and everything inside them). It will make the font smaller and italic.

Declarations contain two parts: the *property* and the *value*. The property specifies what about the content will change; the value specifies what it will change to. Notice that the property is followed by a colon (`:`) and the value is followed by a semicolon (`;`).

- ▶ Create an .html file that includes both an `<h2>` and a `<blockquote>`. Include the `<style>` as in Figure 3.1 and reload the page to see how it changes.

Like in XHTML, the spacing in CSS doesn't matter. In this guide, we will write all stylesheets as you see in Figure 3.1: with the declarations indented, so they can be easily distinguished from the selectors.

What Can CSS Do?

As with XHTML, we won't cover every possible property and value here. For a full list of CSS properties, see the CSS reference online.

There are actually two versions of CSS (at the time this is being written). The reference on the course website only covers version 1 (*CSS1*). You are welcome to use features from version 2 (*CSS2*), but you will have to worry more about what properties are supported by common browsers. Only CSS1 will be discussed in this guide.

There are many aspects of a page's appearance that can be controlled by CSS. Each of these can be controlled for each tag, and in more fine-grained ways that we will see later:

- Font: the font itself (font family), size, whether the font is italicized or not, etc.
- Colours: the text colour, background colour, background image, etc.
- Text: the spacing, alignment, indentation, etc.
- Margins, borders, and padding: See below.
- Positioning: Where elements are placed on the page.
- Others: List formatting, etc.

See the online reference for the exact code used to control each of these.

▶ Have a look at the CSS1 reference on the course website.

▶ Experiment with some of the other CSS properties not mentioned here.

Topic 3.3 External CSS

While CSS can be embedded in XHTML files as described above, this isn't always the best way to include style information on a web page. CSS can also be placed in a separate file. CSS files are created with a text editor and saved with the extension .css. The .css file should contain only CSS code, no HTML tags (like `<style>` or any others). We will call these separate stylesheets *external stylesheets*.

Once we create CSS files, we can tell the web browser to apply that style to many web pages. This is done by using the `<link>` tag in the page's `<head>` like this:

```
<link rel="stylesheet" href="nice.css" type="text/css" />
```

This will instruct the browser to load the stylesheet nice.css from the same directory as the XHTML file. The value of the `href` attribute is a URL, so it can be any absolute or relative URL, as described in Topic 2.9.

You can see a full example of an XHTML page with a stylesheet attached in Figure 3.2. See Figure 3.3 for the CSS file that goes with it.

```
<html>
<head>
<title>Simple Page</title>
<link rel="stylesheet" href="simple.css" type="text/css" />
</head>
<body>
<h1>Simple Page</h1>
<p>This is a simple page to demonstrate CSS.  This is <em>the
first</em> paragraph on the page.  Here's a list:</p>
<ul>
  <li>one</li>
  <li>two</li>
  <li>three</li>
</ul>
</body>
</html>
```

Figure 3.2: A simple XHTML page with a stylesheet applied

```
body {
  font-family: sans-serif;
}
h1 {
  text-align: center;
  text-transform: uppercase;
}
ul {
  list-style-type: circle;
}
```

Figure 3.3: A simple CSS style (simple.css)

Figure 3.4: Figure 3.2 *without* a stylesheet

Figure 3.5: Figure 3.2 with the CSS in Figure 3.3

Figure 3.4 shows a screenshot of the XHTML page Figure 3.2 with no stylesheet applied. Figure 3.5 shows the same page with the simple.css style from Figure 3.3 applied.

Notice that each of the changes specified in Figure 3.3 has been made. Any aspect of the formatting that isn't explicitly changed by the stylesheet remains the same. For example, the text in the `<em>` is still italicized; the margins are the same.

Why Use External CSS?

It may seem easier to embed stylesheets in XHTML using the `<style>` tag than to create an external CSS and link to it with `<link>`. When you are initially experimenting with CSS, this is true.

Embedding CSS with `<style>` becomes problematic when you're working with more than one XHTML file. If you wanted each page to have the same style, you would have to copy the CSS in the head of each page.

Imagine having dozens or hundreds of pages that make up a website and realizing that you needed to update the style: you would have to edit each page separately.

If you use an external stylesheet and link to it, you only have to make your CSS changes in one place: the .css file. The changes to the separate file would immediately be reflected on each page that links to it.

Because of this, you're encouraged to get into the habit of using external CSS files and the `<link>` tag. It will make your life much easier as your websites expand past a single page. Feel free to use `<style>` as you're experimenting with CSS, but move to external stylesheets when creating a real website.

There is a third way to include style information in XHTML. You can specify style information for a single element with the `style` attribute like this:

```
<p style="font-color: red">This is a red
paragraph.</p>
```

This style of CSS is even harder to work with than `<style>`, since style information has to be given *with each tag*, not just with each page.
You may see the `style` attribute used occasionally on other sites, but save yourself some pain later and don't use it on your pages. We will see much better ways to apply CSS to specific parts of the page in Topic 4.5 and Topic 4.6.

- Make a copy of the XHTML file you created with the stylesheet from Figure 3.1. Create a separate .css file containing the style information. In the XHTML file, use `<link>` to link to that CSS and remove the `<style>` tag.

  This page should look the same as it did when the CSS was in the `<style>` tag.

Topic 3.4 CSS Details

In this topic, we will look at some of the things you need to know about CSS to work with it effectively.

CSS "Boxes"

There are several CSS properties that can control the spacing and other stuff around a tag. These include the margins, border, and size of the element itself.

The way these properties relate might not be obvious at first. Once you understand the meanings of the terms, you should be able to start working with these properties and controlling the layout of your pages.

See Figure 3.6 for a diagram of the way CSS sees the "boxes" of content that make up a page.

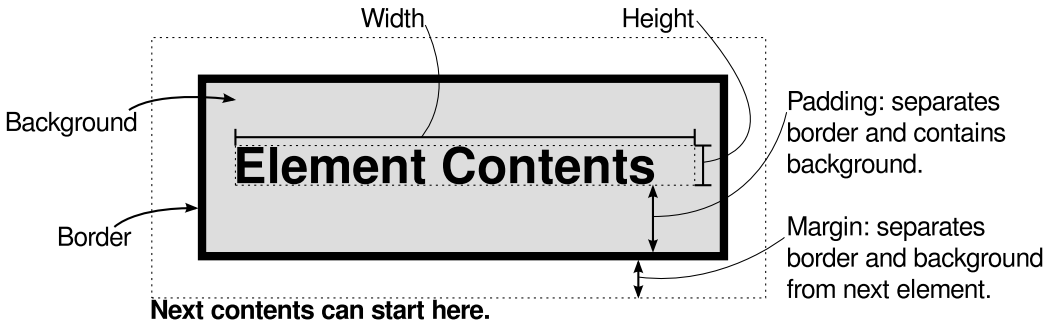


Figure 3.6: The CSS "Box Model"

Border: The border is a line around the contents. In CSS, you can control the border for each of the four sides independently. You can select any colour and a variety of line styles for the border. In Figure 3.6, there is a solid black border around the element.

Background: You can control the background colour and image for individual elements on the page. The background colour extends to the border. It is grey in Figure 3.6.

Padding: The *padding* is one part of the way to control the amount of space around an element on the page. The padding specifies the space between the content itself and the border/edge of the background.

Margin: *Margins* are the other part of spacing. The margins specify space between the border/background and the nearest other element on the page.

Width and Height: These measurements specify the dimensions of the (invisible) box that encloses the contents themselves. You can control the size of the element by setting these values.

☞ Internet Explorer sometimes treats CSS `width` and `height` values improperly: it incorrectly incorporates the size of the padding and border into the given measurement. This shouldn't happen if you produce valid XHTML with a doctype, as described in Topic 4.1.

If an element doesn't have a background or border, then padding and margins look the same. As a result, you might find that you cannot get rid of space that is set by the browser defaults. For example, you might be

trying to adjust the padding, when the browser's default style uses margins. If you want to carefully control spacing around a tag, start by clearing *both* the padding and margins, then add back the spacing you want later in the CSS:

```
margin: 0;
padding: 0;
```

You should experiment with the box properties in CSS to get a feeling for how they interact. See the online CSS reference for details.

- Create a simple XHTML page (or use one you have already been experimenting with) and apply these CSS declarations to the `<h1>`:

  ```
  background-color: #030;
  color: white;
  ```

  Load it in the browser to see the browser's default margins and padding for the `<h1>`.

- Add these to the style for the `<h1>` and reload:

  ```
  border-style: solid;
  border-color: #0F0;
  border-width: medium;
  ```

- Now, add some declarations that change the margin and the padding. Make sure you know the difference between the two. For example:

  ```
  padding: 0.5em;
  margin-bottom: 1em;
  ```

  [We will discuss these units of length more in the next subtopic.]

Lengths in CSS

When specifying widths, paddings, and other properties in CSS, we need to give a *length*. For example, when setting the padding, we need to give the distance from the border to the contents somehow. There are several length units that can be used in CSS.

It's usually best to specify lengths relative to the current font size. This can be done with the unit `em`, which represents the current font size. For

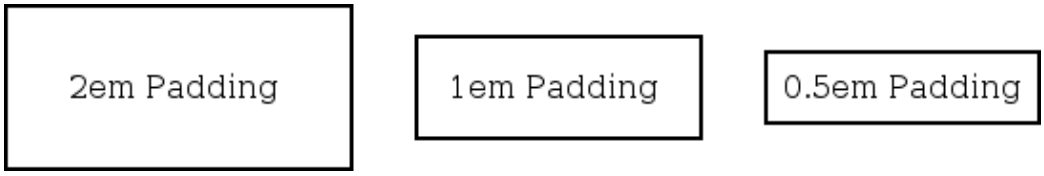


Figure 3.7: Examples of relative units

example, setting `padding: 0.5em;` will set the space between the text and border to be half the current font size.

See Figure 3.7 for an example of using relative units to specify various amounts of padding around some text. If the browser's font size was made larger or smaller, the whole design would scale proportionately.

The benefit of using relative units is that the design of the page will scale properly as the font size changes. Remember that you don't get absolute control over the display of your page—it is done by the reader's browser, and they may want a larger or smaller font. With good CSS design and relative units, the page should still look good.

There are other units that can be used in CSS as well. Here are some examples:

```
p {
  margin-top: 50%;
  margin-left: 1.5cm;
  text-indent: 2em;
  border: 2px solid black;
}
```

This CSS fragment sets the top margin for all paragraphs to half of its previous value. The left margin is set to 1.5 centimetres. The first line of each paragraph will be indented by double the font size. Each paragraph will be given a border that is two pixels wide.

The less-flexible units (like `cm` and `px`) can be used if you have a good reason to do so. There are other length units that can be used as well. Details can be found in the online CSS reference.

Fonts in CSS

The font used for the text of an element can be changed with the `font-family` property. But again, you are limited by the user's browser when it comes to displaying particular fonts: you have no way to guarantee that all of your readers will have the fonts you need.

The solution to this is to specify several options for the font. A list of fonts can be given, in order of preference:

```
font-family: "Verdana", "Helvetica", "Arial", sans-serif;
```

In this case, the browser will use the font "Verdana" if it is available. If not, it will look for "Helvetica", then "Arial", then any sans-serif font.

The last item here is a *generic font family*. There are five generic font families defined in CSS, and your list of fonts must end with one of them. The idea is that every web browser should be able to find *one* of each type.

Here is another example of specifying a font in CSS:

```
font-family: "Bodini", "Georgia", "Times", serif;
```

Topic 3.5 Why CSS?

Before CSS was commonly used by web page designers, visual information for web pages was included in the HTML itself. There were tags and attributes in HTML that gave visual information such as colour, font, and alignment. These parts of HTML are now *deprecated* (that is, they should not be used) in favour of CSS.

As we have discussed, XHTML and CSS are used for distinct purposes. XHTML is used *only* to describe the content of the page: what the parts are and what their meaning/purpose is. Think about the XHTML tags you have seen: paragraphs, lists, headings. All of these specify what kind of content there is; they don't directly indicate an appearance. As mentioned in Topic 2.10, you should choose XHTML tags to maximize the meaning you convey in the tags.

CSS is used to give information about how all of these parts should be presented. For example, there are many different ways that a heading could be presented and still be interpreted as a heading by the reader. We have been keeping the ideas in the page separated from the way they look.

Why are we making this distinction and using two different languages to create web pages?

- There are more options for controlling the appearance of pages with CSS than in old HTML designs. This gives more flexibility when designing a page.
- The combination of XHTML and CSS actually produces smaller files (even though there are two files instead of one). Smaller files result in faster transfer times and quicker response for the user.

  When appearance was specified in HTML, it was necessary to indicate the appearance of *every* paragraph separately. With CSS, you can specify a look for every paragraph at once in the CSS. When you use the same `.css` file on many pages, the `.css` file only has to be downloaded once (with the first page), so loading times (for subsequent pages on the site) are even faster.
- Using XHTML and CSS for different purposes forces authors to separate the *meaning* of the content from its appearance. Authors must ask themselves *why* they want different parts of the page to look a particular way. If some part of the page is a list, the author should use `<ul>` or `<ol>`, as appropriate. Then, CSS can be used to get a particular appearance.

  This helps keep pages organized and makes it easy to create a consistent design. This can also make it possible to have experts create the relevant content: a designer can create CSS, and an author can create the XHTML.
- It's easy to update the appearance of an entire site. As long as a single external stylesheet has been used, there is only one file to be changed when you want to change the site's design.

  Old HTML-only layouts were often very difficult to change. Not only would every instance of a paragraph (for instance) need to be updated, but the designs were often quite fragile, and small changes could have unintended consequences.
- Having only content in the XHTML files allows the pages to be used in many situations. For example, it's possible to create a different stylesheet that will be used on phones, PDAs, or other small-screen

devices; or you could create a different stylesheet with larger fonts for users who need them.

The stylesheet can also be ignored if necessary. For example, it's possible for a speech-browser to read a well-constructed XHTML page to a user who can't see. It does this by applying its own "style" that specifies the type of voice used for different parts of a page.

All of this can be done without changing the XHTML.

- Finally, search engines can easily read pages designed this way. Search engines only look at XHTML (since they don't care about appearance, only content). A well-designed XHTML page will contain a lot of information about what the parts of the page are (titles, lists, etc.), without information about presentation mixed in. Search engines can use this information to get a better idea of what the page is about.

  Pages designed with semantic XHTML and CSS are typically ranked better by search engines because they are easier for the software that runs the search engine to understand.

When creating XHTML pages, remember to focus only on the meaning of all of the parts of the page. The idea is to give as much information as possible to programs other than standard browsers that use your page: search engines, specialty browsers, and so on. These programs can use the XHTML tags to figure out what the parts of the page are *for* and interpret them appropriately.

You can then use CSS to make the pages look the way you want. It can take some practice to get comfortable working with CSS: experiment and don't get discouraged.

Another tip that will help make your pages better for search engines: don't write something like "click here" for your links. Some search engines use the link text to determine what the linked page is about. "Click here" isn't useful: try to create link text that describes what the page is about.

Summary

After you complete this unit, you should be able to apply cascading style sheets to an XHTML page/site. You should be able to create CSS files

that make visual changes to a page. Experiment with the CSS properties introduced here and make sure you understand what they do.

While you are doing labs and assignments, you should experiment further with style sheets and become more comfortable with them.

Key Terms

- CSS
- rule
- selector
- declaration
- property
- value
- padding
- margin
- em (length unit)

Unit 4

Advanced XHTML and CSS

Learning Outcomes

- Use XHTML to create valid web pages.
- Design XHTML so it can be easily styled with CSS.
- Develop CSS rules to create particular appearances.
- Make reasonable guesses about CSS colour codes for a given colour.
- Given a visual design, construct a CSS that implements it.
- Justify the separation of content and structure (in XHTML) from appearance (in CSS).
- Select appropriate XHTML tags to correctly describe the different parts of the page.

Learning Activities

- Read this unit and do the questions marked with a "►".
- Browse through the links for this unit on the course web site.
- Browse the XHTML and CSS reference pages linked in the "Materials" section of the course website.
- (optional) Review *Head First HTML*, Chapters 6, 7, 9, 11, 12.

Topic 4.1 Validating XHTML

Several "rules" of XHTML have been mentioned. For example, `<li>` can only be inside a `<ul>` or `<ol>`. The XHTML reference gives many such rules for the places various tags can be used. But, what if you don't follow these rules?

Web browsers always do their best to display a page, even if everything isn't exactly correct. For example, most browsers will properly display the following as an emphasized link, even though it isn't nested correctly:

```
<a href="bad.html"><em>improperly nested tags</a></em>
```

Browsers want to be able to display as many pages as possible, so they try to work around bad XHTML. Because they let some errors pass, many people creating web pages wrongly assume that *every* web browser will display incorrect XHTML that they produce, and so they develop bad habits.

When you are creating web pages, you should assume that web browsers are very picky about what they display—that will give you the best chance that your pages will work in all browsers. Again, remember that there are a lot of web browsers: you can only test a few.

To help solve this problem, several people have developed *HTML validators* (or just *validators*). These programs check your page against the formal definition of XHTML (or HTML) and report any errors. Putting your web page through a validator will help you find errors that your browser let pass and discourage bad habits.

For a validator to work, you must tell it what version of HTML or XHTML you are using. To do so, you insert a *document type* (or *doctype*) declaration in your XHTML file as the first line, even before the `<html>`. The document type we will be using is:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

(This declaration can be on one line or split across two. You can copy and paste it from one of the examples on the website—you don't have to type it yourself or memorize it.)

This line indicates that the document is written in XHTML version 1.0 as defined by the World Wide Web Consortium (*W3C*). You should add this line to the start of every web page from now on.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Non-Validating HTML</title>
</head>
<body>
<h1>Non-Validating HTML</h1>
<b><p>This is not the way to make a bold
paragraph.</p></b>

<p>Here are some <b><i>badly nested tags</b></i>

</body>
</html>
```

Figure 4.1: Some non-valid XHTML with a doctype specified

☞ We are using the *strict doctype* of XHTML 1.0 in this course. This doctype doesn't allow some tags that shouldn't be used anyway in "good" XHTML. There is also a *transitional doctype* that does allow these, but it isn't allowed for this course.

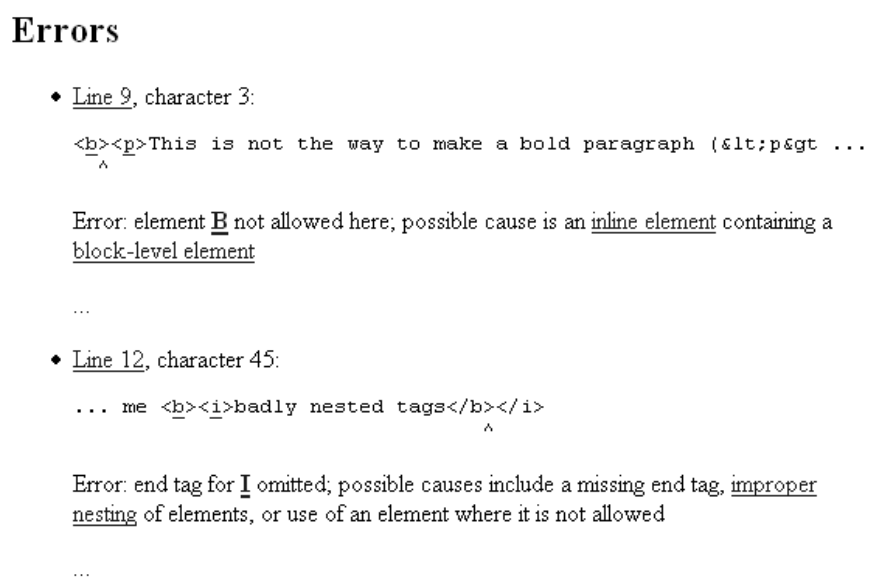
In addition, for XHTML documents, you have to specify a *namespace* for the document. To do so, you add an attribute to the `<html>` tag:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

☞ The namespace is new with XHTML; it wasn't present in older HTML versions. You don't need to worry about why it's there; just make sure you include it.

Once the XHTML version and namespace are specified on your web page, you can visit one of the validators on the web and give it the URL of your page. Links to validators can be found on the course website in the "Technical" section. The validator will retrieve your web page (either from a web server or from your computer) and check it for errors. See Figure 4.1 for an example of some non-valid XHTML and Figure 4.2 for a validator's output.

Figure 4.1 can be found in the "Examples" section under "Materials" on the course website. You can also find some instructions for working with the validators on the course website.



Errors

- Line 9, character 3:

```
<b><p>This is not the way to make a bold paragraph (&lt;p&gt; ...
  ^
```

Error: element **B** not allowed here; possible cause is an inline element containing a block-level element

...

- Line 12, character 45:

```
... me <b><i>badly nested tags</b></i>
                                  ^
```

Error: end tag for **I** omitted; possible causes include a missing end tag, improper nesting of elements, or use of an element where it is not allowed

...

Figure 4.2: Part of the validation results of Figure 4.1

☞ When working on assignments that you need to validate, don't wait until the last minute to try the pages in a validator. You might occasionally encounter errors you don't understand and will want to ask for help on fixing them.
You might also find that you have problems making your XHTML or CSS behave the way you want and that you can fix them by corrrecting validation errors.

▶ Try to validate some of the XHTML pages you have been working on (after adding the doctype and namespace as described above). If necessary, fix the errors so that it does validate.

Topic 4.2 Block vs. Inline Tags

Many of the rules that the validator enforces have a common rationale. For example, you might have found that an `<img>` can't be placed directly inside a `<body>` (it must be in a `<p>` or other tag). An `<p>` can't be put inside a heading; it must be after it.

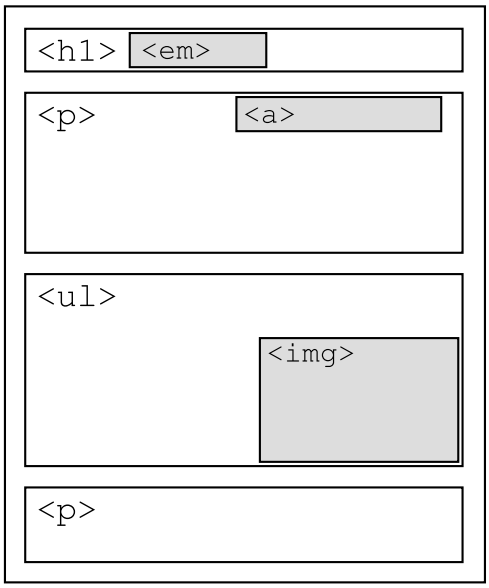


Figure 4.3: Block and inline tags

These restrictions are a result of two different types of tags.

Block tags are the tags that create "blocks" on the page. That is, block tags are arranged one after the other, vertically on the page: a block tag must always be displayed *below* the previous block tag, never beside (unless you move things around with CSS, of course). The block tags include the heading tags (`<h1>` to `<h6>`), paragraphs (`<p>`), lists (`<ul>`, `<ol>`), and some others (`<address>`, `<blockquote>`).

Inline tags are tags that can be inside a block tag and form part of a line. Inline tags can be displayed beside each other, or beside plain text. Inline tags are used to indicate the type or meaning of text (`<em>`, `<strong>`, `<abbr>`), and some special-purpose tags (`<a>`, `<img>`).

Figure 4.3 shows a general illustration of the layout of a page. The inline tags are highlighted with a grey background; block tags have a white background.

Once we understand block and inline tags, many of the rules enforced by the validator can be expressed simply:

- Only block tags can be placed inside the `<body>`. Placing inline tags or text (without a tag around it) inside the `<body>` is an error.

- Inline tags (and text) can be placed inside block tags.
- Block tags can't go inside inline tags.

One common problem is captured here: an `<img>` must be inside a block-level tag like `<p>`.

There are a few more rules that don't fall into these categories. For example:

- `<ol>` and `<ul>` can contain only `<li>` (and `<li>` can't go anywhere else)
- `<a>` can't contain another `<a>`
- `<head>` must contain exactly one `<title>`

Topic 4.3 Entities

There are several special characters in XHTML that are used for special purposes. For example, all XHTML tags are enclosed in `<>`. So, if we want to display a < on a web page, we can't just type it into the XHTML because the browser will assume that it is indicating the start of a tag.

There are also characters that you probably don't have on your keyboard but might want to include on a web page, like "Φ", "°", and "è".

To print this symbol or other special characters, we use *entities*.

All entities start with an "`&`" and end with a "`;`"—the entity for a less-than sign is "`&lt;`". So, to display "7 < 10" on a web page, we would type this in our XHTML document:

```
7 &lt; 10
```

The entities that are required to avoid having to type special XHTML characters are listed in Figure 4.4.

There also many other entities for characters that can't be typed on standard English keyboards. There are some examples in Figure 4.5.

There are several *named entities* that have a easy-to-remember name (like `&copy;`). There are many other characters available that don't have names. These must be accessed by *numeric entities*.

Numeric entities start with a number sign (`#`) and use a character number to indicate the character (like `&#8810;` or `&#x226A;`). When using numeric

Description	Entity	Display in Browser
less than	`<`	<
greater than	`>`	>
ampersand	`&`	&
double quote	`&quot;`	"

Figure 4.4: Entities required for reserved XHTML characters

Description	Entity	Display in Browser
copyright sign	`&copy;`	©
degree sign	`&deg;`	°
Greek capital phi	`&Phi;`	Φ
infinity	`&infin;`	∞
opening double quote	`&ldquo;`	“
closing double quote	`&rdquo;`	”
much less than	`&#8810;`	≪

Figure 4.5: Other sample entities

entities, you must remember that not all browsers will be able to display all characters. So, these characters might not display properly for all readers.

If you're writing only English text, there are very few reasons that you'll need to use numeric entities. Modern browsers and operating systems have nearly universal support for the named entities.

Character Sets

The computer must know how to convert the number in a numeric entity to a character on the screen. It uses a *character set* for this.

A character set is a translation of characters to and from numbers. For example, the number 65 corresponds to a capital "A"; 97 to a lowercase "a"; 8810 to the much-less-than symbol, "≪".

In order for the web browser to correctly translate the data sent from the server, it must know what character set is being used. The character set is often sent along with the MIME type of a web page. If the default is incorrect for your pages, it can be overridden in the XHTML code itself. A tag like this can be inserted into a document's `<head>`:

```
<meta http-equiv="Content-type"
    content="text/html; charset=utf-8" />
```

This indicates that the page has MIME type `text/html` and uses the character set "`utf-8`".

Here, "`utf-8`" refers to the *Unicode* character set. Unicode is a character set designed to be able to represent all written languages. It contains characters for English, Greek, Chinese, Farsi, and so on.

Unicode is supported by all modern web browsers, so there is little reason to use any other character set. But, even if the browser knows what characters to display, that doesn't guarantee fonts will be available with these characters.

For example, your computer may have fonts with Greek, Chinese, and Farsi characters, but it probably doesn't have fonts to display Gothic or Linear B (ancient scripts that can be represented in Unicode). For these characters, most browsers display a question mark or other indicator of a character that can't be displayed.

Topic 4.4 Generic Tags

In Topic 2.10, we said that XHTML tags should be used for content that matches their meaning—the heading tags should be used only for headings, and so on. But what if there is no appropriate tag defined in XHTML for a particular part of your page?

There are two *generic tags* that can be used when none of the other built-in tags are appropriate: the `<div>` and `<span>` tags. The `<div>` tag is used for block content (part of the `<body>`) and `<span>` is used for inline content (part of a block).

It's hard to come up with many uses for `<span>`. The built-in inline tags are sufficient to mark up most content. There are few types of inline content that need `<span>`, but not many.

On the other hand, `<div>` is used much more often. It can be used to identify any section of your page. This is often necessary for formatting with CSS.

For example, many web pages have a list of links to other pages on the site: a site menu. There is no XHTML tag for a "menu" or similar concept, so you would have to use `<div>` to mark this section for CSS formatting.

```
<h1>Page on My Site</h1>

<div id="menu">
  <h3>Our site</h3>
  <ul>...</ul>
  <h3>External Links</h3>
  <ul>...</ul>
</div>

<p>First paragraph of the page...</p>
```

Figure 4.6: XHTML fragment using a `<div>`

Figure 4.6 shows XHTML for (part of) a web page with a menu. The `<div>` in the example could be later reformatted with CSS to look like a menu.

The `<div>` and `<span>` tags should always be given a meaningful `class` or `id`, as we will discuss in the next topic.

Topic 4.5 Classes and Identifiers

As you work with CSS, you may find that you don't always want to style *every* instance of a tag. For example, you only want the `<ul>` tags in the menu to have a particular background, or you want all `<code>` tags that contain XHTML code to be highlighted in a particular way.

In order to do this, you need to be able to indicate which tags are affected in the XHTML, and to indicate changes for only some instances of the tag in CSS. So far, we have only been able to select *all* instances of a particular tag in CSS.

The `class` and `id` attributes can be used in XHTML to distinguish particular elements. Here are three examples:

```
<code class="html">&lt;h1&gt;</code>
<div id="menu">...</div>
<p class="aside">By the way...</p>
```

The value for the `class` and `id` can be anything—there is nothing special about the words "`html`", "`menu`", or "`aside`" above. You should choose a

word that describes the type of content in the tag, in the same way you would choose a tag that matches the content.

The `class` and `id` attributes are used for similar reasons, but they are different.

For `id`, the value *must be unique* on the page. That is, you cannot use `id="menu"` more than once on a page—only one element can have that identifier. So, `id` should only be used for things you *know* will occur only once on a page, like a menu or footer.

A `class` can be used many times on a single page. Several elements could have `class="html"`. So, `class` should be used when there might be multiple instances of that kind of content. There might be several "samples of XHTML code" and "aside paragraphs" on a page, so class was used in the above examples.

There is another unique property of `id`. An identifier can be used as a *fragment* in a URL. For example, you could link to the relative URL page.html#menu. This would display the page so that the browser window is scrolled to the start of the element with `id="menu"`.

Selecting by Class and ID

Once the appropriate parts of the page are distinguished by `class` and `id` attributes, you can move to your CSS file and begin changing the appearance of only those elements.

Figure 4.7 shows a sample stylesheet that modifies only particular classes and identifiers of a tag. These rules restyle the contents of the three examples of `class` and `id` use above.

The first rule in Figure 4.7 affects only `<code>` tags with `class="html"`. The second affects only `<div id="menu">`. The last affects only paragraphs with `class="aside"` specified.

As you can probably guess from the example, selecting by `class` and `id` is straightforward. To select a class, the selector is the tag name, a period (`.`), and the class name. To select an identifier, use the tag name, a number sign (`#`), and the identifier.

It's also possible to select *any* tag with a particular `class` and `id` by simply leaving out the tag name. For example, the CSS selector "`.important`" by itself selects *any* tag with `class="important"`.

```
code.html {
  color: #700;
}
div#menu {
  background-color: #bbf;
}
p.aside {
  margin-left: 3em;
  font-size: smaller;
}
```

Figure 4.7: CSS that changes certain classes and identifiers

Being able to select only certain classes and identifiers gives you a lot more flexibility in CSS. It is now possible to make very fine-grained changes to the appearance of your pages.

Topic 4.6 More Selectors

We have now seen three ways to select parts of an XHTML page in CSS:

Tag selectors: Using the tag name by itself selects all instances of that tag (even if they have a `class` or `id`). For example:

```
body { ... }
li { ... }
```

Class selectors: We can select only tags with a particular `class` specified. For example:

```
p.important { ... }
.dialog { ... }
```

Identifier selectors: Similarly, tags with a particular `id` can be selected. For example:

```
div#header { ... }
#first { ... }
```

There are a few other ways to select parts of an XHTML page in CSS that are worth knowing:

Tags within others (contextual selectors): It is possible to select particular tags *only if* they are inside another, by separating the selectors with a space. For example, to select only `<li>` tags that are inside a `<ul>`:

```
ul li { ... }
```

This can also be combined with other selectors. For example, to select only list items in your menu:

```
div#menu li { ... }
```

Link states (pseudoclasses): Links in XHTML are generally displayed differently, depending on their current state.

You are probably used to seeing this on the web, even if you haven't thought about it: links are blue but turn purple if you have already visited their destination; they are usually red *when* you are clicking on them. All of these can be customized with CSS:

```
a:link { color: #00E; }
a:visited { color: #529; }
a:active { color: #F00; }
```

The three rules here set the colour for an unvisited link (`a:link`), a visited link (`a:visited`), and a link that's currently being clicked (`a:active`). These colours are approximately the same as the default colours in most browsers.

There are a few more ways to select content that we won't go into here. See the online CSS reference for more information.

There are also other pseudoclasses in CSS version 2. They aren't as well supported by browsers, so we won't go into them here.

Topic 4.7 Colours in CSS

There have been a few examples where we have had to set a colour in CSS. For example, above in Topic 4.6, we used this rule to set the link colour:

```
a:link { color: #00E; }
```

The "`#00E`" certainly deserves some explanation.

There are two ways to specify a colour in CSS. The easiest way is to use one of the 17 built-in keywords:

```
h2 { color: navy; }
```

See the online CSS reference for the full list of keywords.

The keywords are easy, but with only 17 of them, they are quite limited. Colours can also be specified with *numeric colour values*, which are much more flexible.

About Colours

Before we start giving numeric colour values, you should know at least a little about how colour works.

If you've ever mixed colours together, you've probably done it with paint. Paints and dyes are mixed with the primary colours cyan (blue), yellow, and magenta (red) in the *CYM* colour model. With paint, the coloured surface absorbs parts of the (usually white) light that illuminates it. Since the paint *removes* parts of the reflected light, this is also referred to as the *subtractive colour model*.

However, paints and computer screens create colours differently. Computer screens actually produce light. Computer screens start black, and light is added to produce a colour, using the *additive colour model*. The primary colours in the additive colour model are red, green, and blue. Additive colour is also called the *RGB* colour model.

To mix light to produce a colour on the screen, we will have to mix various amounts of red, green, and blue to get the desired result.

Working with RGB

We will use the three-character method of specifying a CSS colour. The amount of each primary colour is specified with a character on this scale:

`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`

The `0` end of the scale is little of the colour (dark/off). The `F` end is a lot of the colour (bright/on).

The three primary colours are specified in the order red, green, blue. For example, the colour `#F70` indicates a lot of red, a medium amount of green, and no blue.

Here are some more examples:

Code	Description/Rationale	Colour
`#000`	everything off	black
`#FFF`	everything bright	white
`#F00`	lots of red, no green or blue	red
`#0F0`	lots of green, no red or blue	green
`#060`	some green, no red or blue	dark green
`#444`	a little of everything	dark grey
`#BBB`	more of everything	light grey

Hopefully, the "description" will give you some idea of how to reason about these colour codes. Here are some more examples that might not be as obvious:

Code	Description/Rationale	Colour
`#FF0`	red and green	yellow
`#F80`	red and some green	orange

If you are used to working with paints, these probably don't make any sense. But with light, combining the primary colours red and green makes yellow. If we decrease the amount of green, the colour starts to drift toward red (`#F00`) and becomes orange.

In older versions of HTML, colours could only be specified with six characters, like `#B5123B`. The first two characters specify the red value (`B5` in the example), the second two specify green (`12`), and the last two blue (`3B`). These can also be used in CSS, but we won't go into details because they are more difficult to work with by hand. If you are using a colour selector in a graphics program, it may show you a six-character colour code: you can copy and paste this into CSS if you want to use the colour.

To convert from a three-character to six-character colour code, simply double the characters. For example, `#F80` and `#FF8800` are the same colour.

Taking the *first* character of each pair in a six-character code will give you a similar (but probably not exactly the same) three-character colour code. For example, `#B5123B` is close to `#B13`.

These colour codes can be used anywhere you need to specify a colour in CSS. For example, the text colour (`color`), background colour (`background-color`), and border colour (`border-color`).

- Do some experimenting with these colour codes. Try changing the `background-color` of a page's body, so you can get a good look at the colour you are experimenting with. Change the colour and reload the page—see if the colour displayed matches your expectations.

Topic 4.8 Positioning in CSS

Most of the properties in CSS are relatively easy to use. As you review the online CSS reference, you can experiment with them and get a feeling for what they do.

There are some properties that aren't as straightforward. In particular, the properties that are used to move elements around the page can be hard to figure out.

The `float` Property

The `float` property is probably the easiest way to move parts of the page around; it is also the best supported by browsers.

Applying `float` to an element lets it "float" to the side of the page. That is, the element sits at the side of the page, while the following content flows around it.

As a visual example, see Figure 4.8. Think of the contents here as three tags. The first is represented by solid lines; the second by a box with an "X"; and the third by dashed lines.

As you can see, when the second element (the box) is floated to the left of the page (by the `float` property), the following text (dotted lines) flows around it.

Once the *floating element* has finished, the remaining content returns to the margins, flowing under the floating element. For some floating elements, this is not the desired effect. To avoid this, the remaining elements can be given an appropriate margin, as in Figure 4.9.

For a more concrete example, we will create a website with a link menu on the left side of each page. Creating our XHTML first, we create markup considering the meaning of each part of the page. See Figure 4.10.

Notice that we have given an `id` to the menu list so we can distinguish it from any other lists that may be on the pages. The `<div id="main">` will

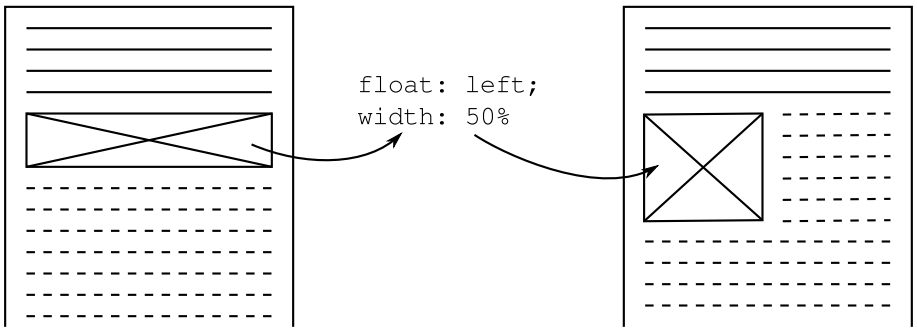


Figure 4.8: Effect of the `float` property

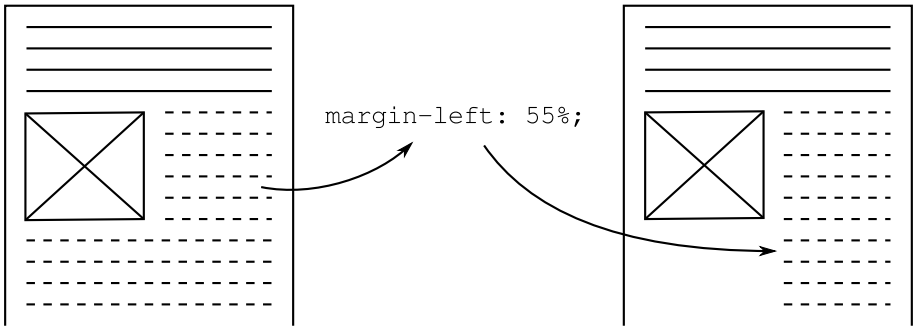


Figure 4.9: Adding a margin around a `float`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Positioning Example</title>
<link rel="stylesheet" href="positionstyle.css" type="text/css" />
</head>
<body>
<h1>Positioning Example</h1>

<ul id="menu">
  <li><a href="about.html">About Us</a></li>
  <li><a href="pics.html">Pictures</a></li>
  <li><a href="friends.html">Links</a></li>
</ul>

<div id="main">
<p>This is my website.  This is my website. ... </p>
<p>It's all about me.  It's all about me. ... </p>
</div>

</body>
</html>
```

Figure 4.10: An XHTML page with a menu

hold the main contents of the page. We need to wrap all of this content in a single block so it can be manipulated as one section; the `<div>` tag is used because there is no other "main section" tag to use in XHTML.

Figure 4.11 contains a simple stylesheet that can be applied to this page.

The `<div id="main">` has been given a large-enough left margin that it won't "undercut" the menu (with a little room to spare for nice separation).

Also notice that a tip from Topic 3.4 has been applied: both the margin and padding of the menu list and list items have been set to zero, to make sure we know how they are spaced. Either margins or padding can be added back later to get the desired look.

Figure 4.12 contains a screenshot of the page in Figure 4.10 with the stylesheet from Figure 4.11 applied.

```
ul#menu {
  float: left;
  width: 7em;
  margin: 0em;
  padding: 0em;
}
ul#menu li {
  margin: 0em;
  padding: 0em;
  list-style-type: none;
}
div#main {
  margin-left: 8em;
}
```

Figure 4.11: CSS for Figure 4.10

Figure 4.12: Screenshot of Figure 4.10

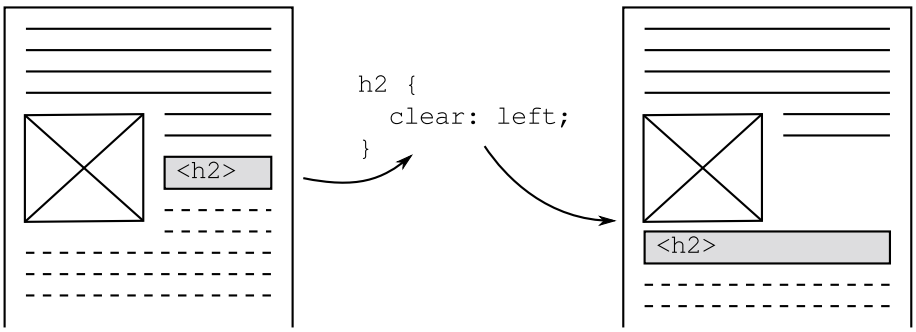


Figure 4.13: Effect of the `clear` property

The `clear` Property

The CSS `clear` property goes hand in hand with `float`.

When you start moving elements around the page with `float`, you may find that there are things you *do not* want beside a floating element.

For example, you might not want to start a new section (`<h2>`) beside a floating image/figure—the new section should start on a fresh line, without being interrupted by content from the previous section.

To achieve this with CSS, the `clear` property can be applied to the `<h2>` tags. This will cause `<h2>` tags to be moved down so they are below any floating content.

You can give a value of `left`, `right`, or `both` to `float`, indicating that the element should be moved below anything floating to the left, right, or both margins, respectively.

For a visual representation of what `clear` does, see Figure 4.13. In this example, `clear: both` would have the same effect (since there is nothing floating to the right).

The `position` Property (Optional)

The `position` property can also be used to move elements around the page. It is more powerful than `float`, but it can be harder to use, and not all aspects are supported by all browsers.

☞ The `position` property was introduced in CSS version 2. You'll have to look at a CSS2 reference for more information. It is the only CSS2 property discussed in this guide.

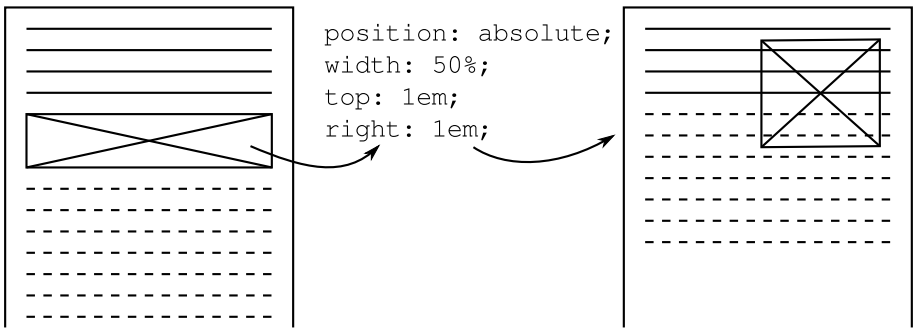


Figure 4.14: Effect of setting `position` to `absolute`

Setting the `position` to `absolute` or `relative` takes the element entirely out of the text flow, and it sits on top of the rest of the page in another "layer".

Figure 4.14 gives a visual example. As you can see, it's possible to create overlapping elements with `position`. This makes it both powerful and difficult to work with.

Setting `position` to `absolute` allows you to use the `left`, `right`, `top`, and `bottom` properties to set the position of the element relative to the page itself. Setting `position` to `relative` lets you move the element relative to its original position.

☞ The `position` property can also be set to `fixed`, which makes it stay in the same place on-screen, even when the page is scrolled in the browser. Unfortunately, Internet Explorer doesn't support this, up to at least version 7.

- Experiment with the `float` property, perhaps starting with Figure 4.10 and Figure 4.11. Try setting `float` to `right`.
- Try using `float` to move a figure (an image and its caption) so the text flows around it.

Topic 4.9 Markup for Meaning

We have already discussed some reasons for writing markup and how to create useful markup.

In Topic 2.4, we saw reasons to write markup directly. In particular, we want to create a document that is flexible and can be used in many situations. The best way to do this is to ignore the way the page looks (to a certain extent) and worry about the markup itself.

In Topic 2.10, we discussed choosing XHTML tags to maximize the amount of meaning we convey in the tags themselves (semantic markup). This is done so a browser (or other computer program) can figure out what the parts of your page are and interpret the page correctly. We have since learned about the generic tags (`<div>` and `<span>`) that can be used when no other tag has the right meaning.

Finally, in Topic 3.5, these ideas were combined with CSS. With CSS, we can take the semantic XHTML we have written and restyle it so it takes on the appearance we want.

At the risk of being repetitive, this would be a good time to look back at these sections. Look at the advice in each, now that you know more about both XHTML and CSS.

With these things in mind, here are the (approximate) steps you should use to create a good website:

1. Start by writing XHTML code for a few of the pages on your site. Don't worry about what the pages look like at this point; just use the tags that are semantically appropriate to each piece of content on the page.

   Do not use tags just because they "look right". Use them for meaning. Use meaningful `class` and `id` values when necessary.

2. Make sure these pages are valid XHTML.

3. Create an external stylesheet for your site and link to your XHTML files. Start to change colours, positioning, and other aspects of the page's design.

4. As you create the stylesheet, you may find that it's necessary to return to the XHTML and add `class` or `id` attributes to some content. You may also need to add `<div>` tags to wrap parts of the page that you need to be in a single element for modification in CSS.

   For example, if you want to `float` several things together, they need to be wrapped in a single tag, like a `<div>`. The `<div class="main">` used in Figure 4.10 is another example of an addition to the XHTML made for the stylesheet.

5. Create XHTML for any remaining pages on your site, keeping in mind the additions you had to make to the XHTML in the last step. Check these pages for validity as well.

Common Mistakes

As mentioned above, you shouldn't use XHTML tags because of the way they look, but because of their intended meaning. But, many web authors slip into bad habits and don't do this (or never knew they were supposed to).

Here are some common mistakes that authors make. Avoid these when creating web pages:

Heading tags for non-headings: People often use the smaller headings (`<h4>`–`<h6>`) for paragraphs or page footers because they like the font that comes with their default style. Of course, you should use an appropriate tag (`<p>` or `<div class="footer">` for these examples) and change the appearance with CSS.

The `<br>` tag for spacing: The `<br>` (line break) tag should *never* be used to create a blank line. It should only be used when a line must end to properly convey meaning. For example, it might be used to mark up poetry or a mailing address:

```
<address>Your Name<br />
123 Fake St.</address>
```

The `margin-top` and `margin-bottom` properties can be used to open up space before or after particular parts of the page.

The table tags for layout: Before CSS was implemented by the majority of browsers, there were few options for authors who wanted to control the layout of their pages. The `<table>` tag was used to move elements around. This lead to complicated HTML pages that are very difficult to work with.

The table tags should only be used to mark up tabular data. For example, it would be appropriate to use the table tags to mark up Figures 4.4 and 4.5 for a web page.

Of course, CSS should be used to control the position of elements on the page.

The appearance-only tags: There are several XHTML tags that have no meaning associated with them, only an appearance (`<b>`, `<i>`, `<big>`, `<small>`, `<sub>`, `<sup>`).

These tags should not be used. Pick a more meaningful tag instead.

Transitional tags: There is another doctype for XHTML 1.0 that allows use of *transitional tags*. These tags are intended to allow a smoother transition from pre-CSS web pages to modern sites.

Don't design web pages as if it's still 1998. Let's all move on.

The ` ` entity for spacing: The ` ` entity is for a *non-breaking space*. It looks like a space, but the browser isn't allowed to go to a new line there. For example, it's considered bad style to break a line in the middle of someone's name, so in XHTML, you might type `John Smith`.

Some authors use several of these to create extra horizontal space. Of course, you should use appropriate CSS properties instead (probably `margin-left` and `margin-right`).

Of course, there are many other ways to misuse XHTML markup, but these are the most commonly seen.

When it comes to the "meaning-only" tags mentioned above, one could argue here that `<span>` doesn't have any meaning associated with it either. So if you're allowed to use `<span>`, why not use these tags too?
We'll leave the argument to some web standards mailing list somewhere. These tags might be acceptable, but they should be used like `<span>`: they must have a meaningful `class` or `id` specified.

▶ Look back at some of your earlier work in this course. Did you do any of these things? Do you know how to do it the "right" way now?

Summary

This unit contains a lot of new information and techniques. It will probably take you a while to go through it and absorb all of the information. Experimenting is the key to actually figuring out how these things work, so make sure you (at least) do the labs and questions within this unit.

Once you have worked through this unit, you should be reasonably proficient at creating both semantic and valid XHTML, and at restyling these pages with CSS to achieve the appearance you want in your pages.

Key Terms

- validator
- valid XHTML
- doctype
- block tag
- inline tag
- entity
- character set
- Unicode
- generic tags
- class
- identifier (`id`)
- RGB
- floating element

Part III

Graphics and Design

Unit 5

Graphics and Images

Learning Outcomes

- Compare and contrast the characteristics of basic types and formats of computer graphics.
- Identify an appropriate format for a given image.
- Select an appropriate graphics file format for use in a web page.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.
- Install your graphics program as described in the "Technical" section of the course web site.
- (optional) Review *Head First HTML*, Chapter 5.

We have already seen how to use the `<img>` tag in Topic 2.8. In this unit, we will learn more about the images themselves, both on the web and elsewhere.

Topic 5.1 Graphics and Image Types

The term *computer graphics* refers to using a computer to create or manipulate any kind of picture, image, or diagram. There are many different ways to create computer graphics, and you should choose a program that fits your

needs. We cannot cover them all here, so we will discuss some of the common concepts and terms.

There are two basic ways to store an image in a computer: vector graphics and bitmapped graphics.

You are probably most familiar with *bitmapped graphics* (sometimes called *raster graphics*). Bitmapped graphics are used in *paint programs*. When you use bitmapped graphics, the image is stored as a grid of small squares or *pixels*. Each pixel in the grid is assigned a colour. If the pixels are small enough, the image will closely resemble the intended image.

The other basic image type is *vector graphics*. Vector graphics are used in *drawing programs*. A vector format stores a description of the image, using various shapes like circles, lines, curves, and so on. For example, a vector image could contain the information "a green line from *here* to *here*, a red filled circle with centre *here* and with *this* radius".

Some of the bitmap graphics programs you might know are Photoshop, Photo-Paint, Paint Shop Pro, GIMP, and Graphic Converter. Since most web graphics are bitmap graphics, these are the programs you will likely want to use for this course. Some vector programs are Inkscape, Corel Draw, and Adobe Illustrator. Most vector graphics programs have the ability to include bitmaps in vector images, but it doesn't make them paint programs—the bitmaps are just one type of object that can be inserted into the vector image.

- What graphics programs do you have on your computer? What have you used in the past? Do those programs work with bitmap or vector images?
- Open up the bitmap graphics program you're using for this course (probably the GIMP; you can find instructions on the GIMP in the "Technical" section of the course website). Open a new image and draw something.
- Experiment with a vector drawing program. You can download Inkscape for free.

Topic 5.2 Bitmap vs. Vector Images

Bitmap and vector images each have strengths and weaknesses when it comes to storing various kinds of images and to being manipulated in certain ways. Because of the way they store their images, they are also edited differently.

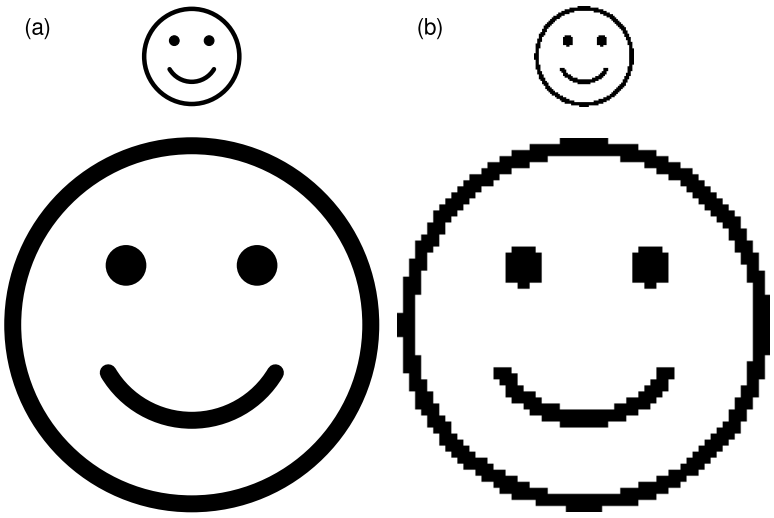


Figure 5.1: Scaling (a) a vector image and (b) a bitmapped image

Scaling and rotating: Since bitmap images are just a grid of pixels, they can't be easily scaled up. Basically, the only way to enlarge the image is to make the pixels larger, which makes the images look blocky. With vector images, the shapes that make up the image can be scaled up or down smoothly. See Figure 5.1.

When rotating an image or doing other manipulations, the situation is the same. With a bitmap image, the program can only work with the pixels in the image, and there is often some distortion in the final image. Vector images can generally be modified more smoothly, resulting in a cleaner image.

Display and printing: Computer displays and printers are bitmap devices: they use a grid of pixels to produce the image you see. Printers have a much higher resolution than computer monitors.

That means that vector images must be converted to bitmaps before they are displayed. On modern computers, the amount of computation required to do this isn't significant; when computers were slower, it was a concern. It can still matter if it must be done often (during an animation, for instance).

If the size of the bitmap isn't exactly the same as the desired output

size, it must be scaled. This is why bitmapped images often don't print well: because printers have such a high resolution, the pixels become noticeable.

Producing: Programs that edit bitmap images are more common, and the file formats are more universally accepted. The SVG format (discussed later) is causing more common use of vector images, however.

Devices such as digital cameras and scanners produce bitmap images because of the way their sensors work. Editing vector images is often easier because shapes can be manipulated separately.

File size: Vector files are often smaller than bitmaps because there is simply less information to store.

To see why, consider the information that has to be stored in either case. For a bitmap image, the computer must store millions of pixel colours. For a vector image, it only has to store the list of shapes in the image.

- If you have created any computer graphics in the past, did you use a bitmap program or a vector program? Looking at these strengths and weaknesses, do you think you made the right choice?

Topic 5.3 File Formats

There are many ways to store images. Each is called a *file format*. Once we have decided to use a bitmap or vector format, the program we're using must still be told how to store and represent the information we need on the disk. Since there are many choices to make here, many different image formats have been created.

Each file format has a different way of converting the image information into a string of information that can be stored on a disk or transmitted over the Internet. We won't discuss the details of how each format works. The process is far too complicated for this course. We will just discuss some of the capabilities of common formats. The differences in file formats are the reason you can't take a GIF file, rename it .jpg , and expect things to work. To convert a file from one type to another, a program must convert the image data from one type to the other.

The program Graphic Converter for the Macintosh can handle 145 different graphics formats. Some have more strengths than others and are used more often.

Some common bitmap formats are GIF (graphics interchange format), JPEG (joint photographic experts group), PNG (portable network graphic), BMP (Windows bitmap), and TIFF (tagged image file format); most bitmap programs can read and write all of these examples.

Some common vector formats are SVG (scalable vector graphics), EPS (encapsulated postscript), CMX (Corel meta exchange), PICT (Macintosh Picture), and WMF (Windows metafile). Most vector editing programs can read and write most of these types.

Since there are so many formats to choose from, it can be difficult to decide on one. Here are some things to keep in mind:

- Type: does the format store the type of image you want to use? (bitmap or vector)
- Portability: can others use images in this format?
- Colour depth: does the format store the number of colours you need?
- Compression: compression can make a smaller file, at the cost of the compression/decompression time.
- Transparency: did you need some parts of the image to be clear?

The first two of these considerations should be self-explanatory. The other three need some explanation.

Colour Depth

Image formats can only store so many different colour values, depending on the number of bits assigned to each pixel. The format must store the level of each of the three primary colours (or *components*) in order to specify the colour (as we saw for CSS colours in Unit 3). The *bit depth* indicates the number of colours that can be used in the image.

A *24-bit image* can indicate any one of 2^{24} colours for each pixel. Each of the components is stored on a scale to $2^8 = 256$. This is usually enough colours because most people cannot distinguish between two colours that differ only by one unit. Thus, full-colour photos look good when stored in 24-bit colour.

Similarly, *15-bit colour* can have any one of 2^{15} colours, with 2^5 possible values for each component in each pixel. It's sometimes also referred to as *16-bit colour*. Full-colour images still look good in 15-bit colour, but you can usually tell the difference between the same image in 15- and 24-bit.

Formats with fewer colours usually use *paletted colour* or *indexed colour* for their images. Here, some colours are chosen to be in the image's *palette*. The colours in the palette are the only ones that can be used in the image. For each pixel, we only need to store a number that refers to a position in the palette.

For example, an *8-bit image* has a palette of $2^8 = 256$ colours. An 8-bit paletted image can use up to 256 colours chosen from the 2^{24} possible 24-bit colours. A *1-bit image* can have 2^1 colours, usually just black and white.

Using an image format with more colours gives you more flexibility, but it usually also means a bigger image file as well because more information must be stored for each pixel. It is often desirable to work in 24-bit colour and then convert to a lower bit depth to create a smaller file for storage or transmission.

If you try to store an image in a file that can't hold as many colours as the image has, some decisions must be made. Usually, the graphics program will choose a set of colours that closely matches the ones in your image. For colours that aren't found in the palette, the program can either pick the closest colour or do *dithering*.

Dithering is the process of creating a pattern of pixels that fools the eye into seeing a colour that is between the two. See Figure 5.2 for an example, using black and white pixels to make a grey square. If you hold the page back, the dithered square on the right should look grey. If everything has gone well in the printing process, it should be about the same colour as the pure grey square on the left. If you look very closely at the pure grey square, you can probably see some fine dithering in it as well; this dithering is done during the printing process since printers and copiers only have black ink to work with.

Compression

It is also possible to create a smaller file with *compression*. Without compression, bitmap images can be very large. For example, a 640×480 pixel

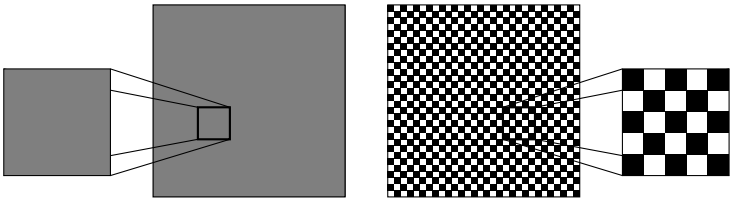


Figure 5.2: Colour dithering

image with 24-bit colour depth would take

$$640 \times 480 \times 24 \text{ bits} = 7372800 \text{ bits} = 900 \text{ kB}\,.$$

Images this large would take a long time to transfer over the Internet. This is the reason the uncompressed Windows BMP format is not used on the web.

There are many different ways to compress data. The various methods fall into two categories: *lossless compression* and *lossy compression*.

Most image compression techniques are *lossless compression*. When data is compressed and then uncompressed with a lossless algorithm, it is *exactly* the same.

When an image is compressed with a *lossy compression* algorithm and then uncompressed, it may be slightly different. This technique is often acceptable for images, particularly if the changes are so slight that they can't be seen with the naked eye.

The JPEG format was created to store photographs at a high quality in very small files. It uses a lossy compression format. JPEG images might change some small details, but the changes can't usually be seen, except at the lowest quality settings. See Figure 5.3 for an example of a small image (a) that has been compressed and uncompressed with a low-quality lossy format (b). Note that this isn't a full-colour image; that's why the JPEG compression has done such a bad job with it—it wasn't made for that job.

The GIF format uses the LZW compression algorithm, and the Unisys corporation had a patent on this method. The PNG format was created so there would be a free lossless format available for use on the web. In this course, we recommend that you use PNGs so that you do not have to worry about misusing the patented GIF format.

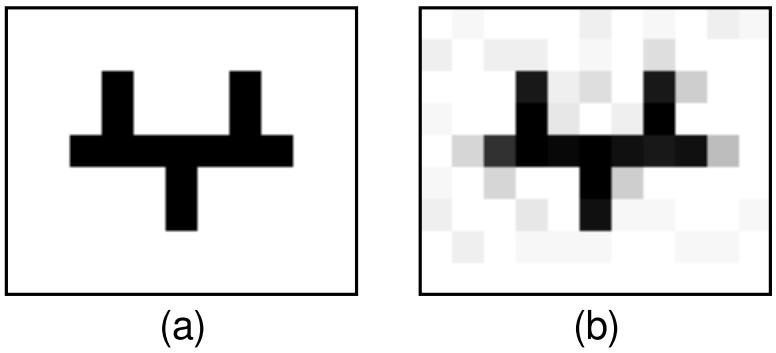


Figure 5.3: An image with a low-quality lossy compression

Transparency

Another feature that is important for some images is *transparency*. Some image formats can indicate that part of the image is to be transparent so whatever is behind it will show through. This technique is often used on web pages so that images appear to be shapes other than rectangles—the image is still a rectangle, but the background shows through some of it, making it seem otherwise.

Most image formats don't support transparency, so all images appear square, as in Figure 5.4(a).

Some formats, GIF in particular, support *simple transparency* (or *binary transparency*). Some of the pixels in the image are marked as transparent, so the background is visible through those parts of the image. This format can be used to make an image appear any shape; an example can be seen in Figure 5.4(b).

Finally, some image formats support a more general method of transparency called an *alpha channel*. The image has the image information and a mask, called the alpha channel, that indicates how transparent each part of the image is.

Formats and programs that support alpha channel transparency can do partial transparency, as seen in Figure 5.4(c). The PNG format supports full transparency, as do graphics programs like Photoshop and GIMP.

☞ Unfortunately, Internet Explorer doesn't support partial transparency in PNG images, as of version 7.

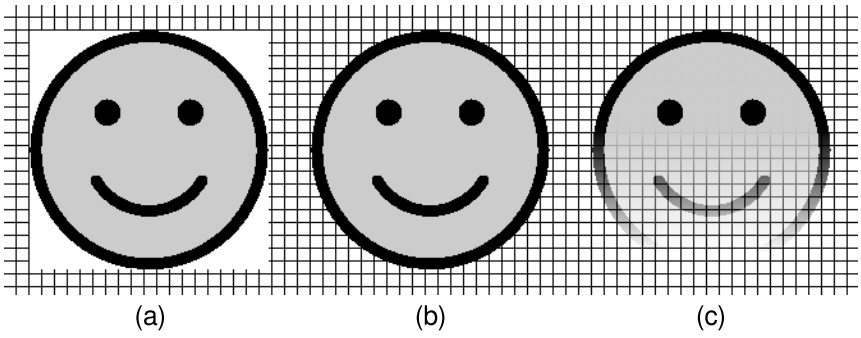


Figure 5.4: Various types of transparency in images

Topic 5.4 Common Image File Formats

Since some image formats are used more commonly than others, we will describe some of them and what they are usually used for.

These formats are the ones common on the World Wide Web:

JPEG: JPEG (Joint Photographic Experts Group, pronounced *jay-peg*) is a bitmap format with lossy compression that is intended for either 24-bit colour or 8-bit grey-scale photographs—the kind of thing what would come out of a digital camera or from scanning a photograph. It does an excellent job of storing this kind of information but a poor job with other kinds of images. Figure 5.3 is an example of what happens when an image that isn't a photograph is compressed with JPEG.

GIF: The GIF (pronounced *jiff*) bitmap format uses a lossless compression algorithm. GIF is well supported on the web and can be used for simple animations and simple transparency. It is, however, limited to 256 colours in an image (8-bit colour). It is also burdened by a patent on its compression algorithm.

PNG: The PNG (pronounced *ping*) bitmap format was created as a free replacement for GIF. It uses a better and free compression algorithm. It can also support up to 24-bit colour and full transparency. The PNG format came along after GIF, so it wasn't supported in some older browsers. All browsers that support any graphics support PNG, so you should be able to use it safely.

SVG: SVG (Scalable Vector Graphics) is a vector format that was created by the WWW Consortium in the hopes of making vector graphics possible on web pages. SVG is increasingly supported by web browsers and can be edited by most vector graphics programs, including Inkscape, which can be downloaded for free.

The following formats, while not common on the web, are often used in other types of digital multimedia:

BMP: The Windows BMP format is a bitmap format. It is usually uncompressed (for speed), but it can use a simple compression algorithm.

TIFF: TIFF (Tagged Image File Format) is a common bitmap format among people who do publishing and graphic design. It can be compressed in several different ways and and can hold colour depths up to 24-bits.

EPS: EPS (Encapsulated Postscript) is a vector format. It is quite widely supported and can be used in most drawing programs.

Most computer graphics programs also have a file format of their own, usually called *native formats*. For example, Photoshop uses PSD files, the GIMP uses XCF files, Corel Draw uses CDR, and so on. These file formats use either no compression or a lossless compression scheme. They are designed so that they can store any of the information that particular program can produce.

So, you should be able to save these files and open them back up without losing any information. That is a good reason to use them while you are working on an image. When you're ready to publish it or pass it along to others, you should probably convert it into one of the more universal formats described above.

- If you have created any computer graphics in the past, what file format(s) did you use? Do you think you made the right choice(s)?
- What is the native format that your image editing program uses (if any)?

Summary

This unit isn't intended to teach you everything about image editing or even how to use a particular program. You should know some of the basic terms

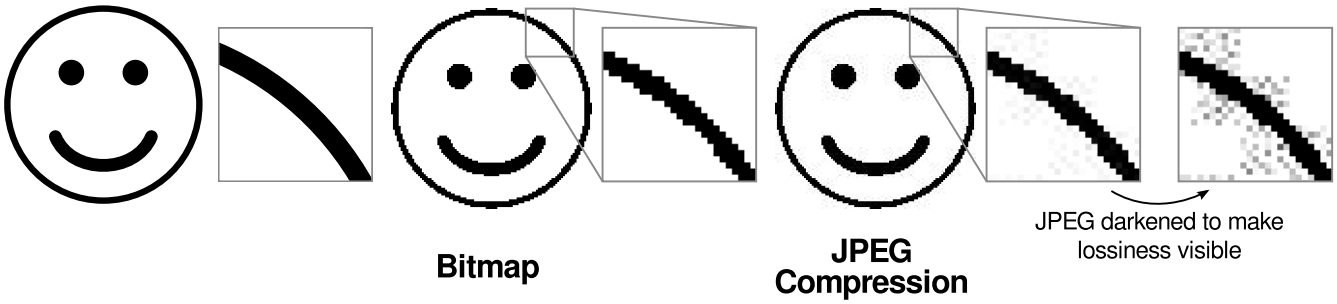


Figure 5.5: Summary of graphics formats

and ideas behind computer graphics. With this background knowledge, you can easily learn more about graphics and how to use a graphics program for your assignments.

See Figure 5.4 for one last example of what the same image looks like when stored with various graphics formats.

Key Terms

- bitmap graphics
- vector graphics
- pixel
- colour depth
- paletted or indexed colour
- dithering
- compression
- lossless compression
- lossy compression
- transparency
- alpha channel
- graphics file format

Unit 6

Design

Learning Outcomes

- Explain some principles of design.
- Apply design principles to the creation of websites.
- Evaluate the design of websites and other materials.
- Create websites where the user can quickly navigate to the desired information.

Learning Activities

- Read this unit and do the questions marked with a "►".
- Browse through the links for this unit on the course web site.
- (optional) Read the first half of *The Non-Designer's Design Book*.

Topic 6.1 General Design

When we discuss how to design web pages, we must first describe some general principles of design. This section discusses design for any medium.

The points made here are based on the first half of *The Non-Designer's Design Book* by Robin Williams. Williams lays out four principles of design for creating well-designed documents. The four principles are *proximity*, *alignment*, *repetition*, and *contrast*. Used together, these ideas can help you

change a poor design into one that is visually pleasing and easy to get information from.

There are a few other things you should do when you are designing anything. Keep in mind that the whole point of your design is to make it easy for viewers to get the information they want. Many web pages fail here—how often do you find yourself hunting through a page for the link you want? Remember, it's all about information.

When you're working on your design, have other people look at it. What's the first thing that they see? Is that what you want them to focus on? Can they find the information you want them to get from your presentation?

Finally, you should design for your medium. Surfing the web, you get the impression that a lot of people don't know that the World Wide Web isn't a magazine or television. Every medium has its own strengths and weaknesses when it comes to getting a message across. You have to keep these aspects in mind so you don't try to force your design into a medium where it doesn't fit.

Proximity

Stated concisely, you should "group related items together" (p. 15). The elements on your page should not be scattered randomly or all grouped tightly together.

Related items should be placed near each other, and unrelated items should be separated by some space. If you follow this principle, those viewing your document should realize what items are related before they start to read. It will make it easier for them to scan your document and find the relevant information.

Many inexperienced people seem to be afraid to leave any blank space on their page (*whitespace*). Whitespace helps separate unrelated topics, and you shouldn't avoid it.

When it is properly used, proximity helps your document look organized. Thinking about how items should be grouped together might even help you understand the organization of your material better.

Figure 6.1 is an example of a design that illustrates the concept of proximity. Notice that the contact information, address, phone number, and website are grouped together. Also, the logo and name of the business are separated from this information.

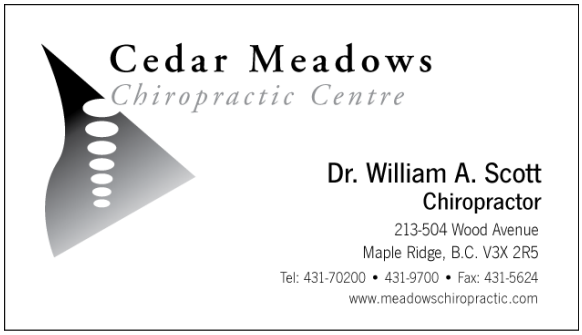


Figure 6.1: A design illustrating proximity

Alignment

"Alignment" refers to the position of something on the page. Williams states that "nothing should be placed on the page arbitrarily. Every item should have a visual connection with something else on the page" (p. 31). A common mistake beginners make is placing elements wherever they fit without considering the way they line up with other things on the page. Remember that everything should line up with something else.

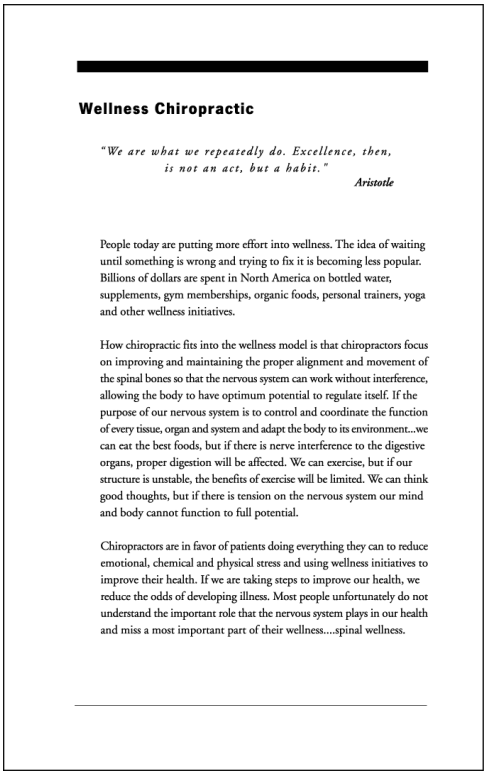
Alignment should create a "line" on the page that the eye can follow. It should also make your page look organized. This "line" will give the reader a good idea of how to follow the information. Centring text doesn't always do a good job of creating this line.

Alignment is illustrated in Figure 6.2. On this page, the left side of the rules and heading are aligned, as are the left sides of the quote and main text. The right sides of the rules and main text are aligned.

Of course, you don't have to align *everything* on the page. In Figure 6.2, the title and the text do not have the same left alignment.

Repetition

The idea here is to "repeat some aspect of the design throughout the entire piece" (p. 49). For example, you might use the same distinctive font, rule, bullet, or colour in your entire presentation. You can repeat anything that the reader can recognize.



Wellness Chiropractic

"We are what we repeatedly do. Excellence, then, is not an act, but a habit."

Aristotle

People today are putting more effort into wellness. The idea of waiting until something is wrong and trying to fix it is becoming less popular. Billions of dollars are spent in North America on bottled water, supplements, gym memberships, organic foods, personal trainers, yoga and other wellness initiatives.

How chiropractic fits into the wellness model is that chiropractors focus on improving and maintaining the proper alignment and movement of the spinal bones so that the nervous system can work without interference, allowing the body to have optimum potential to regulate itself. If the purpose of our nervous system is to control and coordinate the function of every tissue, organ and system and adapt the body to its environment...we can eat the best foods, but if there is nerve interference to the digestive organs, proper digestion will be affected. We can exercise, but if our structure is unstable, the benefits of exercise will be limited. We can think good thoughts, but if there is tension on the nervous system our mind and body cannot function to full potential.

Chiropractors are in favor of patients doing everything they can to reduce emotional, chemical and physical stress and using wellness initiatives to improve their health. If we are taking steps to improve our health, we reduce the odds of developing illness. Most people unfortunately do not understand the important role that the nervous system plays in our health and miss a most important part of their wellness....spinal wellness.

Figure 6.2: A design illustrating alignment

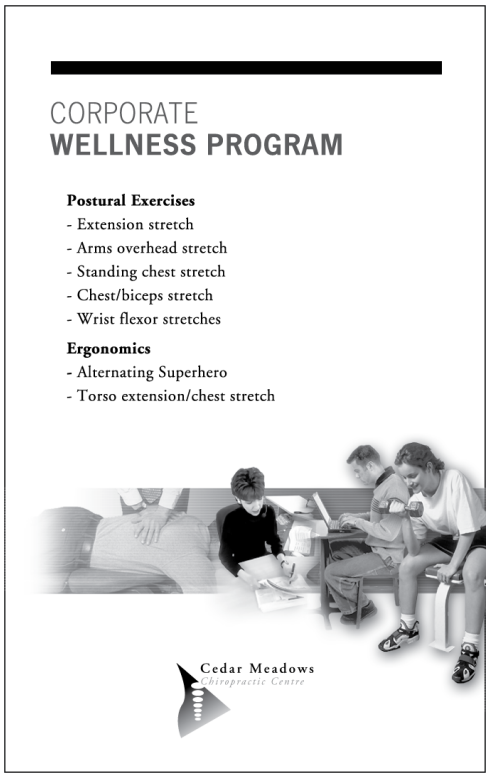


Figure 6.3: A design illustrating repetition

Along with alignment, repetition will unify your presentation so it all looks like part of the same creation. A repeated element gives the user something to hang on to and gives the presentation a consistent feel.

On the other hand, you shouldn't repeat too much. Everything in your presentation shouldn't look the same.

In Figure 6.3, the fonts for the headings and lists are repeated, as are the bullets. Notice that there are also several repeated items from Figure 6.1 and 6.2. These include the heavy horizontal rule, fonts, and logo. These elements make it clear that all of these designs fit together.

Contrast

"If two items are not exactly the same, then make them different. Really different" (p. 63). The reader should be able to tell at a glance what parts

of the page serve the same purpose.

Contrast can be created by using a different typeface, colour, background, border, and so on. These differences should be obvious; nothing should be just a little bit different from something else.

A common problem in word processing documents is using a 12 point body font, with 14 point headings. In this case, it is too difficult to distinguish between a heading and a short paragraph. It would be better to have the headings 16 point, bold, and in a (noticeably) different font or colour.

You shouldn't be afraid to try something new with your design to create contrast. The worst thing that could happen is that it will look ugly and you will have to change it back. Remember, when you're working with a computer, you can do that.

As is the case with repetition, you should not have too much contrast. The point is to make some elements stand out. If everything is in a different font and colour, nothing will stand out, and your whole presentation will look cluttered.

In Figure 6.4, there are three types of text on the page: the heading, the main text, and the questions for the reader. These groups are distinguished by their fonts and background colour, which makes it clear that each has a distinct purpose on the page.

- Look at some ads in a magazine or newspaper. Find some that use these four principles well and some that don't. How would you change the poorer ones?
- Critique your earlier assignments with respect to these principles. How did you do?
- Critique Figures 6.1 to 6.4 for all four design principles.

Topic 6.2 Design Principles and XHTML/CSS

Williams's design principles can be used when you are designing web pages. Remember that the web isn't a magazine or poster; design for the web is different. No matter how much you might want to, you can't use the same visual layout on a web page that you use for a poster. (On the other hand, you can't include a hyperlink on a poster.)

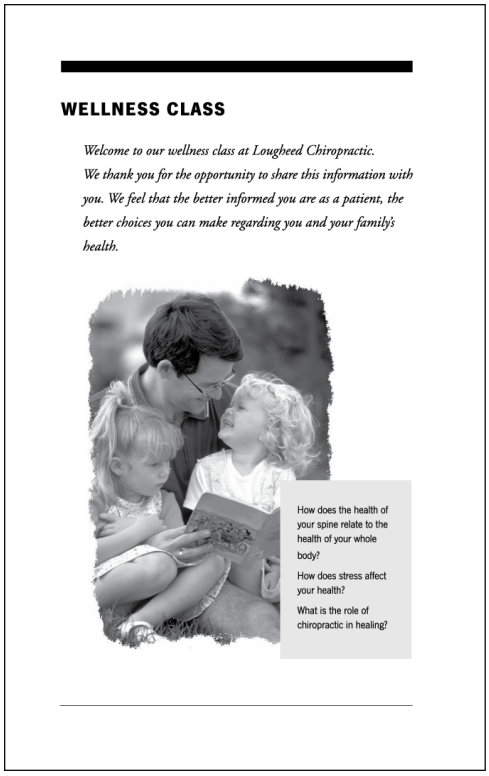


Figure 6.4: A design illustrating contrast

If you stay within the confines of what XHTML and CSS are intended to do, you can still do some very interesting design. In addition, if you do, your web page will look and act like a web page, not like a print or TV ad. It will be a lot less confusing for your users.

To achieve proximity on a web page, you have to think about how your information fits together. You can then separate the different groups of information using headings and the CSS margin properties. When you are creating a website, you also have the option of moving some information to another page—which certainly qualifies as separation.

Web pages usually have a strong left alignment with no style sheet changes. You can further affect alignment using the `text-align` property for text and the `float` property for images and other elements.

CSS can easily be used to create repetition. You can create a distinctive style for heading, links, or other common elements on your pages. You can also use the `list-style-type` and `list-style-image` properties to give all of your lists a distinctive bullet. It is also easy to use CSS to apply a particular colour for accents on your pages (links, borders, etc.).

Finally, contrast is easily lost on web pages. You should be careful that your headings do not look too similar and that they stand out from other elements. You should also make sure that your links stand out from the rest of your page—some web authors make links look almost exactly like the surrounding text, which makes it difficult for users to find them.

Topic 6.3 Conventions

People make a lot of "assumptions" about how things work in their daily lives. For example, if someone walks up to you, extends their right hand and says "Hi", you probably know what to do: shake their hand and say "hi" back. You know to do that based on a lot of experience in similar situations. These aren't "rules" that you must follow, but things that you know you *should* do. It is a social *convention*.

There are many conventions on the World Wide Web as well. For example, if you see a list of links on the left or right of the page, near the top, you know that it's probably the site menu. Large text near the top of a page is probably the page title. There are many more examples.

When creating a web page (or creating anything else for people to interact with), you should use these conventions to your advantage. If you follow common conventions for your website, you can use the knowledge your readers already have, so they can instantly figure out how to use your site.

Also, conventions usually become common because they work well. Using common conventions will help you create a site that's easy to navigate and work with.

Remember: users spend most of their time on *other sites* and are used to the way they work. If you make your site work differently, your users will be confused.

Example

Let's look at one common convention as an example: blue/purple underlined text. When you see this, you know it's a link.

As discussed in Topic 4.6, you can change the colours and other properties of links with CSS. But, should you? If you make the links on your side orange and italics (but not underlined), people visiting your site won't immediately know how things work. They might figure it out, or they might just leave annoyed.

You can probably get away with changing *one* property of the links on your site without leaving visitors bewildered. For example, you could remove the underline but leave your links blue/purple.

You should also avoid having any *other* (non-link) text that's blue and underlined. Users will certainly try to click on it, thinking it's a link.

As you browse the web, you will encounter many more such conventions. Try to keep them in mind when designing your sites. Your page doesn't have to look *exactly* like every other pages but there are many aspects of the way pages work and are navigated that you should follow.

- Test your knowledge of web conventions: Where is the "log out" link on most websites? Where can you find information about the author of an online news article? From the main page of a blog site, how do you make a comment on an entry?

Topic 6.4 Readability

Much of this section is based on a column "How Users Read on the Web" by Jakob Nielsen. For more information, see http://www.useit.com/alertbox/9710a.html. You can find a link on the "Links" page of the course website.

When people write web pages, they often assume that visitors will read every word, top to bottom. But, most visitors don't "read" web pages this way: they scan for the information they want. Visitors often scan titles, headings, and links, and make a very quick decision about whether they want to look further.

When creating pages, you should keep this in mind. You should make it easy for visitors to quickly scan your page and find the information they want. If you don't, a visitor might scan your page, decide that it doesn't contain useful information, and go elsewhere.

There are several things you can do to make your page "scannable":

- Divide your page into sections. Have a clear, meaningful heading for each part (`<h2>` for sections, `<h3>` for subsections).
- Keep your pages short. Divide your site into pages of a manageable size, and keep the amount of text to a minimum.
- Create obvious areas of the page for main parts the users may be looking for: page title, menus, and so on.
- Follow conventions, so users can use what they know about the rest of the web when scanning your site.
- Make your links clear. As mentioned in Topic 6.3, your links should look like links. Also, don't use link text like "click here". So, **don't do this**:

  ```
  <a href="xyz.html">Click here</a> for more
  information about XYZ.
  ```

  Use link text that can stand alone if someone reads only that text: the link text should describe the destination of the link. For example:

  ```
  We also have <a href="xyz.html">more information
  about XYZ</a>.
  ```

Here, the link text "more information about XYZ" describes the destination of the link, without having to read any of the surrounding text.

- Avoid "rollovers". A *rollover* is a menu that pops up when the user moves the mouse over it. Sites often use rollovers so they can have a lot of options in their menus, without taking up a lot of space on their page.

  But, because users often scan pages, they often don't realize the rollover menus are there, and certainly can't easily scan their contents. Users may navigate away without ever noticing the hidden options.

▶ How easy has it been to scan the web pages you have created? How could you have reorganized them to facilitate scanning?

▶ Think about sites you visit frequently. Are they easy to scan for relevant information? Have you *ever* read them top to bottom?

Topic 6.5 Page Design

When designing a site, many people forget that there are many ways to get to the pages of their site. For example, visitors might have followed a link from a search engine, or they might have followed a link from another site.

So, you must remember that someone looking at one of your pages doesn't necessarily know how it fits into the rest of your site, or even what the rest of your site is about. They also can't click their browser's "back" button to move back within your site.

Because of this, every page on your site should have enough information for a reader to orient themselves. There should be information about where they are on the site. Is it a small corner of a large site, or the top of a site on this topic?

You should also include good information about what *this* page is about. Having a meaningful `<title>` and `<h1>` can make a big difference in how easy it is to figure out the content of the page.

Finally, you shouldn't have any "dead end" pages with no links back to other parts of your site. You should have a link up to the menu, or other navigation that makes sense for your site.

Summary

This unit should help you design *good* websites and other multimedia presentations. A lot of guidelines have been presented here, and it's hard to keep them all in mind at once. When you're doing assignments, look back through this unit and try to put the guidelines into practice.

Key Terms

- proximity
- alignment
- repetition
- contrast
- conventions
- readability

Part IV

Web Programming

Unit 7

Programming Introduction

Learning Outcomes

- Explain what programming is and what Python is.
- Create simple Python programs.
- Develop Python web scripts using output and variables.
- Create programs that create web pages when uploaded to a web server.
- Compare and contrast the actions performed by the client and the server when a static vs. a dynamic request is made.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.
- (optional) Read Chapters 1 and 2 in *Think Python*. .

The next units cover some of the material found in *Think Python: An Introduction to Software Design*. You can do these readings to get another perspective on the material covered in this *Study Guide*. You shouldn't worry about any *new* material in *Think Python*, just use it to reinforce the material covered here.

Topic 7.1 What Is Programming?

The last part of this course focuses on *computer programming*. We won't assume you have programmed before, so we will start at the first step: what is programming?

Basically, a computer program is a list of steps that a computer can follow to accomplish some task. The hard part about creating a computer program is that computers are dumb: they need *a lot* of detail specified so that they can follow the instructions. The instructions also have to be specified in a way the computer can understand.

A *programming language* is a particular way of expressing instructions to a computer. There are many programming languages. They are all designed for different reasons, and all have strengths and weaknesses, but they share many of the same concepts.

We have already covered XHTML, which we said was a *markup* language. You might be wondering what makes a *programming* language different from a markup language. For one thing, a markup language is used to create a document, whereas a *programming* language is used to create a computer *program*.

Also, there are some things that a programming language must be able to do that XHTML cannot. XHTML doesn't have any variables (Topic 7.4), conditional statements (Topic 9.2), or other features needed to create programs. Anything called a "programming language" will have these (or something equivalent). Basically, writing XHTML (or HTML) isn't "programming".

Why Learn to Program?

There's a good chance that if you're in this course, you'll never make a living at programming. So, why would you bother learning how to do it?

For a lot of people, the answer may be "because it's fun." Programming is an interesting challenge.

It is also nice to have another tool you can use to solve problems. If you know how to program, it's surprising how often you will find that a quick program is the easiest way to deal with a problem.

```
print "Hello"
print "I'm a program."
```

Figure 7.1: Our first Python program

Topic 7.2 Starting With Python

In this course, we will be using the *Python* programming language. You can download Python for free. Python is also an excellent programming language for people who are learning to program.

You may be wondering why this course doesn't teach programming in C++ or Java. These are the languages that you probably hear about most often. The reason is simple: we aren't trying to make you into computer programmers.

C++ and Java are very useful for creating desktop applications and other big projects. We aren't doing that here, and you'll probably never do it. Languages like Python are a lot easier to work with and are well suited to web programming. C++ is rarely used for web programming because it isn't well suited to the task.

Python programs are made up of *statements*. A statement is basically an instruction in the program. The statements that make up the program are followed *in order*.

For example, Figure 7.1 contains a simple Python program with two statements. In Python, statements are typed one per line.

In order for the computer to follow the instructions in Figure 7.1, we have to type them in somewhere and create a Python program. Then, we need to *run* or *execute* the program.

You can type Python code into a text editor and run it all at once, but the Python software comes with its own editor, which we will discuss here.

See the course website for instructions on working with the Python software.

The `print` Statement

The two statements in Figure 7.1 are both `print` statements.

The `print` statement in Python is used to put text on the screen. Whatever comes after it on the line will be put on the screen.

Any text in quotes, like `"Hello"` in the first line, is called a *string*. Strings are just a bunch of characters. They have to be placed in quotes to distinguish then from Python commands. If we had left out the quotes, Python would have complained that it didn't know what "`Hello`" meant, since there is no built-in command called `Hello`.

Here are some more examples of the `print` statement:

```
print "Goodbye"
print "100 + 65"
print 100 + 65
```

Here are the results of these statements:

```
Goodbye
100 + 65
165
```

The first two `print` statements should be no surprise. They work the same way as the others we have seen: they display the string they are given.

In the last `print` statement, the stuff to be printed isn't in quotes, so it isn't a string. In this case, it's an *expression*. As you can see, the `print` statement can also be used to display the result of an expression.

Expressions

An expression is basically a calculation that is specified for Python to carry out. The example expression above, `100 + 65` is a *numeric expression*. That is, it's an expression that, when evaluated, gives a number as its result.

There are other types of expressions in Python as well. Expressions can produce strings, or other types of Python information.

We will explore these more later.

- If you haven't already, type in the program from Figure 7.1, following the instructions from the course website.
- Add a few more `print` statements to that program (one per line). Run it again and see what happens. Try some other expressions.

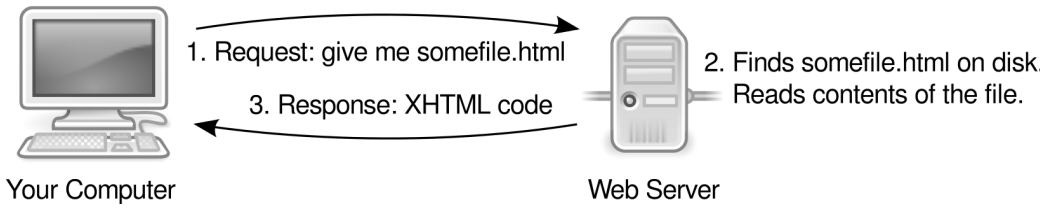


Figure 7.2: Fetching a static XHTML file

Topic 7.3 Web Programming

It is possible to create web pages with programs (written in Python or just about any other language). This is the application of programming that we will focus on in this course.

Up to this point in the course, the web pages we have made have just been XHTML code saved in .html files. When these pages were requested by a web browser, the contents of the files were sent as-is.

These are *static pages* on the web server. Static pages are simply read from the server's disk and sent back to the web browser that requested them. All of our graphics and stylesheets have been static as well. For a visual representation of a static request, see Figure 7.2.

When we create pages with a program, the program will be executed *every time* the page is requested. That means that the page might be different with every request.

Pages that are created by running a program for every request are called *dynamic pages*. In this case, the server finds the requested file, *runs the program*, captures the program's output (what it prints), and sends that back to the browser. For a visual representation of a dynamic request, see Figure 7.3.

☞ A web server has to be configured so it knows which files are static and which should be executed as dynamic requests. On the course web server, Python programs are served as dynamic requests.

Notice in Figure 7.2 and Figure 7.3 that the web browser doesn't have to do anything differently for a dynamic request: it requests a page and receives XHTML to display. The browser doesn't care where the XHTML came from; it just displays it.

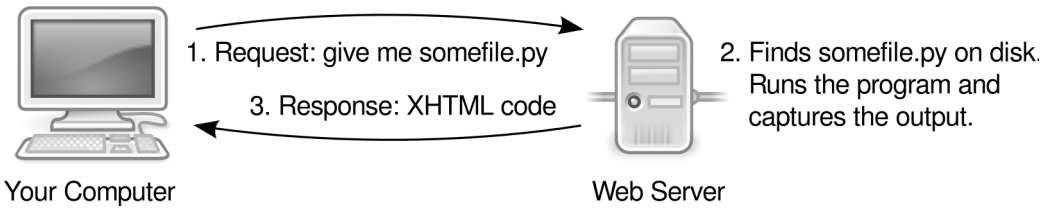


Figure 7.3: Fetching a dynamic web page

```
print "Content-type: text/html"
print
print "<html><head>"
print "<title>Python did this</title>"
print "</head><body>"
print "<p>Here I am</p>"
print "</body></html>"
```

Figure 7.4: A program that generates a simple web page

The terms *web programming*, *web scripting*, and *CGI scripting* are used to describe creation of programs that dynamically generate web content.

Creating Python Web Scripts

If you read the above carefully, you already have a pretty good idea of what your Python program has to do to create a dynamic web page: it has to output (`print`) the XHTML code for the page.

The code for a web script can be entered in IDLE or another text editor, just like any other Python program. Save it as a .py file. These files can be uploaded to your web space on the course web server just like any other file.

Figure 7.4 contains a simple Python web script. This is a fairly simple Python program. You should be able to predict what will be printed if you run it in IDLE:

```
Content-type: text/html
    [Blank line]
<html><head>
<title>Python did this</title>
```

```
</head><body>
<p>Here I am</p>
</body></html>
```

Most of its output looks like XHTML code, and that's what we want. This is the XHTML that will be displayed by the web browser when the program is uploaded as a dynamic web page.

The first two lines don't produce XHTML code, however.

Content-type

A web script can produce more than just XHTML. In fact, you can write web scripts to produce any type of content: JPEG images, PDF documents, or anything else. So, when the web server sees a .py file, it has no way to determine what type of content it will produce. It doesn't know what MIME type to send to the browser along with the content.

That means that we, in the web script, need to indicate the MIME type ourselves. That is what the first line does. It prints out an *HTTP header* that indicates the MIME type of the data we're about to send.

This line must be "`Content-type: `" followed by a MIME type. For all of our programs, it will be the same:

```
print "Content-type: text/html"
```

The next `print` statement simply prints a blank line. This is necessary to separate the HTTP header(s) from the actual contents.

Once this is done, you can print out any XHTML code.

- ▶ Type in Figure 7.4 and save the program as a .py file. Upload this to the course web server, and visit the page in your browser. You should see an XHTML page displayed, the same as you would for a static .html file.

Topic 7.4 Expressions and Variables

So far, none of our programs have been very interesting. They have all just run from top to bottom, printing something with each line. This is because we have only had one type of statement (`print`) and some very limited calculations.

In order to write programs that do more, we need to add some more types of statements and other Python features.

We have seen that it's possible to do calculations in Python:

```
print 33 * 5                                    ⇒  165
print "abc" + "def"                           ⇒  abcdef
```

In this guide, we will use the "⇒" to indicate the output of a `print` statement or the result of a calculation. So, the first `print` statement above prints "`165`".

The second statement above contains a *string expression*. When the `+` sign is applied to strings, it joins or *concatenates* them.

The `print` statement works well when we want to immediately display the results of an expression. But, it's often necessary to save the result of some calculation so it can be used later.

Results of any calculation can be stored in a *variable*. A variable is just a part of the computer's memory that you can use in your program to hold information temporarily.

To store a value in a variable, a *variable assignment statement* is used. A variable assignment statement looks like this:

```
area = 12 * 7
```

After this statement is executed, a variable named `area` will be created, and the result of the calculation (84) will be stored in that variable.

The `*` is used to indicate multiplication in Python. It can also be used to represent string repetition, as we will see in the next example.

The `=` is used to indicate a variable assignment, with the name of the variable on the left and the expression to evaluate on the right. Variable names in Python can contain letters, numbers, and underscores (`_`). Note that the *result* of the calculation is stored in the variable (the number 84), not the expression itself.

To use the value stored in a variable, just mention it by name in an expression. In this example, the variable containing the course name is created and used to create several lines of output:

```
course = "CMPT 165"
print course
print "This is " + course
print course*2
```

The output of this program is:

```
CMPT 165
This is CMPT 165
CMPT 165CMPT165
```

As we see more examples of Python programs, we will use variables more.

Summary

You should now be able to start writing programs that do simple calculations and produce web pages when uploaded to the server.

If you're intimidated about starting to write programs, don't panic. Writing your first few programs will be difficult, since you're learning about programming, Python, the Python tools, and error messages all at once. After you get a few programs working and get the feeling of the steps you have to take, it will get easier.

Key Terms

- computer programming
- programming language
- Python
- web programming/web scripts
- static page
- dynamic page
- `Content-type`
- expression
- variable
- assignment statement

Unit 8

Forms and Web Programming

Learning Outcomes

- Create XHTML forms to capture user input.
- Create Python programs to generate web pages, using a user's input.
- Convert the types of values in Python as appropriate for calculations and output.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.
- (optional) Review *Head First HTML*, Chapter 14.

We have now seen (1) a tiny introduction to programming and (2) how to create web pages on the fly with a program.

But, things don't get really interesting until you can somehow get input from the users of your site—then you can make web applications that really *do* something.

Topic 8.1 Forms

When we create web scripts, we can't get input from the user the way most programs do, by waiting for the user to click or type something. On the web, you have probably entered terms into a search engine or typed your address

into an e-commerce site. These are examples of *forms* on web pages, and we can use them to get input from the user when we are web scripting.

The data in forms is usually sent back to a web script on the web server. In this topic, we will introduce the XHTML tags needed to create forms. In Topic 8.2, we will write web scripts to process the data and send back an XHTML page in response.

The `<form>` tag is wrapped around the entire form. It shouldn't affect the appearance of the page; it's just used as a marker to indicate what parts of the page are part of the form. The `<form>` tag must contain block tags: content inside a `<form>` must be in a `<p>`, `<div>`, or other block tag.

The `<form>` tag must be given an `action` attribute. This will indicate the URL of the web script that will get the results of the form. That is, the program that we will write to handle the form data and produce the XHTML results. We will have to write the action script—this will be explored in Topic 8.2.

The form in Figure 8.1 will send its results to a web script named sample.py in the same directory as that page.

The `<input>` tag is used to put controls on the form. The `type` attribute of the tag indicates the kind of control, for example, a text input box, a button, or a check button. We also usually specify a `name` for each control. The name is used so we can refer to the control later.

Two types of `<input>` are used in Figure 8.1, and their appearance in a browser can be seen in Figure 8.2.

The `text` type of input is a single-line text box; two examples are (a) and (b) in Figures 8.1 and 8.2. Since each `<input>` should be given a unique name so we can refer to it later, these text boxes are named `text1` and `text2`. These names could be anything, like the name for an `id` or `class`, and they should be somewhat descriptive.

The `value` attribute can be used to put some text in the box initially, as has been done in (a). The attribute `size` indicates how many characters wide the box should be, and `maxlength` gives the maximum number of characters that the user will be allowed to enter, as in (b).

Another type of input shown in Figures 8.1 and 8.2 is a `submit` button, (c). The user can click on the button to send their results to the script from an `action` attribute of the form. The `value` attribute is used to give the text that should be placed on the button itself.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Form Example</title>
<link rel="stylesheet" href="formstyle.css" type="text/css" />
</head>
<body>
<h1>Form Example</h1>

<form action="sample.py">
<div class="formin"> (a)
  <input type="text" name="text1" value="A textbox" />
</div>
<div class="formin"> (b)
  <input type="text" size="6" maxlength="10" name="text2" />
</div>
<div class="formin"> (c)
  <input type="submit" value="Go!" />
</div>
</form>

</body>
</html>
```

Figure 8.1: Example form on an XHTML page

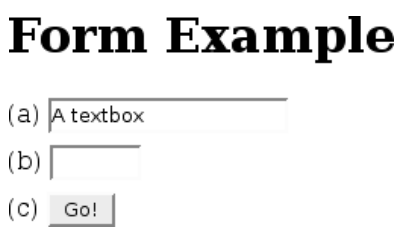


Figure 8.2: The display of Figure 8.1 in a browser

There are several other possible values for the `type` attribute of the `<input>` tag. You can find information about these types and how they are used in the online XHTML reference.

- Try creating a web page with a form and some controls on it. Of course, the controls won't do anything yet, but you should be able to see them. Try validating the page you create. Give the form an `action` with the file name of a script that you can create in the next topic.

Topic 8.2 Reading Form Input

Web scripts, like the ones we started to write in Topic 7.3, can read information that is entered into forms, like the ones created in Topic 8.1

Our job here will be to write the "action" script for the form.

Figure 8.3 contains a Python web script that takes the user's input. Since the `<form>` tag's `action` attribute gives the filename `sample.py` as the URL for submission, Figure 8.3 would have to be saved in a file named sample.py in the same directory as Figure 8.1.

Here is what each part of that program does:

`import cgi`

The method of passing information from an XHTML form to a web script is called *CGI* (*Common Gateway Interface*). The Python module `cgi` provides functions that can be used to work with CGI data. Here, the module is loaded so it can be used later.

Modules will be discussed further in Topic 9.1.

`form = cgi.FieldStorage()`

This loads the form data (using the `cgi` module) into a variable called `form`. Once the data is in a variable, we can work with it. The first two lines of this program will probably be exactly the same in any web script you create.

```
text1 = form.getvalue("text1")
text2 = form.getvalue("text2")
```

These lines take data from the form variable and put it into simple string variables that we can easily work with. The expression

```
import cgi
form = cgi.FieldStorage()

text1 = form.getvalue("text1")
text2 = form.getvalue("text2")

# print HTTP/HTML headers
print """Content-type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html><head>
<title>A CGI Script</title>
</head><body>
"""

# print HTML body using form data
print "<p>In the first text box, you entered " + text1 + ".</p>"
print "<p>In the second text box, you entered " + text2 + ".</p>"
print "</body></html>"
```

Figure 8.3: A web script that uses the input from Figure 8.1

`form.getvalue("text1")` retrieves the string that the user typed in the input with `name="text1"`.

We will keep the names of the variables and the inputs the same, so they are easy to keep track of. That is, the string typed into an input with `name="age"` will go into a Python variable `age`.

If you try to get a value that wasn't part of the form, you will get a special value `None` back. This will likely cause an error later in your program. For example, if you have asked for something like `form.getvalue("abcd")`, but there is no input with `name="abcd"` in the submitted form, the variable will be `None`.

`# print HTTP/HTML headers`

This line is just a Python *comment*. It has no effect on the program when it runs. It's just there for someone reading the code. In this case, it indicates what the next few lines do.

`print """Content-type ... <body>"""`

The next eight lines are actually a single `print` statement.

Many web scripts have to output large chunks of XHTML code that won't change. In Figure 7.4, the whole program is just a collection of `print` statements. It's often tedious to write these statements.

In Python, there's a shortcut you can use whenever you need to work with large chunks of text. You can use a pair of `"""` to wrap up a large string. A *triple-quoted string* can also contain line breaks, which regular strings (quoted with `"`) cannot.

These two fragments of Python code do exactly the same thing:

```
print "Line one"
print "Line two"

print """Line one
Line two"""
```

This makes it easy to copy and paste large pieces of XHTML from a static file into a Python program. You can just copy the code into a `print """..."""` wherever it should go.

We could have written eight separate `print` statements here. The result would have been exactly the same (but it's hard to print quotes in the doctype, so this just works out better).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html><head>
<title>A CGI Script</title>
</head><body>

<p>In the first text box, you entered A textbox.</p>
<p>In the second text box, you entered other.</p>
</body></html>
```

Figure 8.4: Example XHTML produced by Figure 8.3

`# print HTML body using form data`

Another comment. Again, it indicates what the next few lines do.

```
print "<p>In the first ...  " + text1 + ".</p>"
print "<p>In the second ...  " + text2 + ".</p>"
```

These lines actually output the contents of the `<body>`. Since the variables `text1` and `text2` contain the text the user entered, we can use them here.

Each line combines some text that is always the same (`"<p>...entered "` and `".</p>"`) with the contents of a variable. If the user entered "abcdef" in the first text box in Figure 8.1, the first of these lines would print

```
<p>In the first text box, you entered abcdef.</p>
```

`print "</body></html>"`

A final `print` statement to close the open tags and complete the XHTML page.

Suppose someone visits the XHTML page from Figure 8.1 and leaves the first text box containing "A textbox" and enters "other" in the second text box.

When they click the submit button, sample.py from Figure 8.3 will produce the XHTML code in Figure 8.4; this will be sent to the user's browser. When displayed in the browser, it will look exactly like a static page with the same contents.

- Modify the program in Figure 8.3 and experiment with different output.
- Write your own web script that takes some form data and displays it.

Topic 8.3 Handling Types

We have already seen that there are different kinds of information that Python can handle. For example, these expressions highlight the difference between two types:

```
1 + 65                                  ⇒  66
"1" + "65"                              ⇒  "165"
```

The first expression (`1 + 65`) adds two numbers together: one plus sixty-five is sixty-six. The second (`"1" + "65"`) concatenates two strings: the string resulting from joining the character "1" to the characters "6" and "5" is "165".

The difference between the expressions is in the types of the values. The first expression takes two numbers and results in a number. The second takes two strings and produces a *string* (a sequence of characters). The `print` statement can output either type.

Converting to Numbers

Whenever we get input from the user, it will be a string. They have typed a sequence of characters, so Python represents it as a string. When we get form data (`form.getvalue("name")`), it will always be a string.

If we want to do arithmetic operations on some numbers the user enters, the strings have to be converted to numbers.

The `int` and `float` functions can be used to convert a string to a number. The `int` function converts to an integer—a number with no decimals. The `float` function coverts to a *floating point* number—a number that can have a fractional part.

Here are some examples:

```
int("123")                              ⇒  123
float("123")                            ⇒  123.0
float("123.45")                         ⇒  123.45
int("123.45")                           ⇒  error
```

These would typically be used when you are first fetching the user's input. Suppose you have asked the user to enter their age in an input with `name="age"`. Their age is an integer, so you would probably want to convert it to an integer right away:

```
age = int( form.getvalue("age") )
```

This code takes the text the user entered in the text input (which is retrieved with `form.getvalue("age")`) and converts it to an integer (with `int`) before storing it in the variable `age`.

If you are asking the user for a number that may have a fractional part, like a length, you should convert it appropriately:

```
length = float( form.getvalue("length") )
```

Converting to Strings

When printing a value, it's often necessary to convert it to a string.

The `print` statement will convert a single value automatically, but you often want to output more than just a single variable or other value. For example, in Figure 8.3, the variable contents were printed out as part of a sentence that was constructed by concatenating strings (with `+`).

But, to join items together with `+`, they must both be strings. For example, if `count` is a variable containing a number, this would cause an error:

```
print "<p>Number so far: " + count + "</p>"
```

In order to put a number into the string we're going to print, it must be converted to a string. This can be done with the `str` function:

```
print "<p>Number so far: " + str(count) + "</p>"
```

Basically, if you need to print a number, you should convert it to a string first.

Summary

There aren't many new programming skills in this unit. It is really about taking the programming ideas from Unit 7 and adding user input from XHTML forms. You should also be able to deal with numbers in the program as well: converting string input to numbers, and numeric values to strings for output.

Key Terms

- XHTML form
- input tag
- triple-quoted string
- CGI
- type

Unit 9

More Programming

Learning Outcomes

- Develop Python web scripts using conditionals, modules, and functions.
- Make decisions in programs and take appropriate actions.

Learning Activities

- Read this unit and do the questions marked with a "►".
- Browse through the links for this unit on the course web site.
- (optional) Read Sections 3.1–3.4, 5.1–5.7, and Appendix A in *Think Python*.

Topic 9.1 Python Modules

In Python (and other programming languages), you aren't expected to do everything from scratch. Some prepackaged functions come with the language, and you can use them whenever you need to. So far, we have seen `int`, `str`, and a few others.

In modern programming languages, there are so many functions for common tasks provided, that it doesn't make sense to have all of them available all the time. Many of the provided functions are separated into *libraries*. In Python, each part of the total built-in library is called a *module*.

In our web scripts, the `cgi` module is used to grab the user's input so we can use it in our program. The statement "`import cgi`" tells Python that

we want to use the `cgi` module in our program. If all of the modules were loaded all the time, our programs would take a very long time to start up. So, we have to import the modules we're actually going to use.

Inside the `cgi` module, there is a function named `FieldStorage` that we used to extract the CGI data. Since the function is in the `cgi` module, we used it by calling `cgi.FieldStorage()`.

As you can see, to call a function in a module, we first import the module. Then, the function is called by giving the module name, a period, and the function name. So, the `sin` function from the `math` module would be called as `math.sin(...)`.

As another example, the `random` module contains many functions that generate random numbers. These can be used to introduce an element of chance into a game.

Looking at the online reference for the `random` module, you'll see that it contains a function named `random` that returns a random number between 0 and 1 (possibly including 0, but never 1). For example:

```
import random
    ⋮
print random.random()
```

There are Python modules to do all kinds of things—far too many to mention here. There is a reference to the Python modules linked from the course website. You will be pointed to relevant modules when necessary.

Topic 9.2 Making Decisions

All of our programs so far have run straight from top to bottom. We need to have some way to implement conditional code. That is, code that only runs under certain circumstances. For example, we might want to ask the user for a password, and display certain information only if the password they enter is correct.

In order to do this, we need two things: a way to express the condition (e.g., is the password correct?), and a way to indicate which code runs only when the condition is true.

Boolean Expressions

First, we will explore the first problem: expression conditions. These conditions are written with *boolean expressions*. The two *boolean values* are "true" and "false". A boolean expression is any expression that evaluates to true or false.

We have already seen numeric operators (like `-` and `*`) that do a calculation and produce a number. We will now see some *boolean operators* that result in a boolean value.

Here are some basic boolean operators:

`<`	less than
`>`	greater than
`==`	equal to
`!=`	not equal to

So, each of these expressions evaluates to a boolean value. Assume the variable `count` contains the integer 10:

```
1+2 < count                    ⇒ True
1+2 > count                    ⇒ False
count == 11                    ⇒ False
count != 11                    ⇒ True
```

In Python, `True` and `False` are special values indicating the two boolean values.

To check to see if two values are equal, the `==` operator is used. Note the difference between `=` and `==`. The `=` is used for variable assignment—you're telling Python to put a value into a variable. The `==` is used for comparison—you're asking Python a question about the two operands. Python won't let you accidentally use a `=` as part of a boolean expression, for this reason.

The `if` Statement

The most common way to make decisions in Python is by using the *`if` statement*. The `if` statement lets you ask if some condition is true. If it is, the *body* of the `if` will be executed.

For example, consider this program fragment:

```
if num < 10:
    print "num if less than 10."
    print "It sure is."
print "This code runs no matter what."
```

We will assume that a number has been assigned to `num` before this code starts. If `num` has a number less than 10, the condition (`num < 10`) is true, so this code will print three lines:

```
num if less than 10.
It sure is.
This code runs no matter what.
```

If `num` contains a number 10 or larger, the condition is false and this code only prints one line:

```
This code runs no matter what.
```

As you can see, the two indented `print` statements are not executed when the `if` condition is false. These two statements make up the *body* of the `if` statement. The last `print` is executed no matter what; it isn't part of the `if`.

In Python (unlike many programming languages), the amount of space you use is important. The only way you can indicate what statements are part of the `if` body is by indenting, which means you'll have to be careful about spacing in your program.

How much you indent is up to you, but you have to be consistent. Most Python programmers indent 4 spaces, and all of the example code for this course is written that way.

In the Python interpreter, a "..." prompt is used to indicate that you're expected to keep typing because you haven't finished the statement yet. This will happen if you try an `if` statement in the interactive interpreter.

The `else` Clause

We have seen how to check a condition and execute some code if it's true. It is quite common to want to check a condition, execute some code when it's true, but execute *other code* if it's false.

In the `if` statement, you can specify an ***`else`*** *clause*. The purpose of the `else` is to give an "if not" block of code. The `else` code is executed if the condition in the `if` is false.

For example, suppose we were checking the password a user entered. The code might look like this:

```
passwd = form.getvalue("passwd")
if passwd=="secretpass":
    print "<p>Password correct."
    print "Here's the secret information:</p>"
      ⋮
else:
    print "<p>Sorry, incorrect password.</p>"
```

Topic 9.3 String Formatting (Optional)

So far, we have built all of our strings for output with the concatenation (`+`) operator. This can quickly become cumbersome:

```
print "Hello, " + name + "."
print "<p>Total needed: " + str(total - onhand) + ".</p>"
```

There is an easier way to substitute values into Python strings. The *string formatting operator* can be used to insert values into marked places in a string.

For example:

```
print "Hello, %s." % (name)
print "<p>Total needed: %i.</p>" % (total - onhand)
```

We build the *template string* for the output by indicating a substitution with a `%` and a character. The character after the `%` indicates the type of value that will be substituted. A `%s` indicates that a string will be inserted; a `%i` indicates substitution of an integer; `%f` indicates a floating point value.

After the string, the `%` (string formatting) operator indicates that you want to do a substitution. Then, you give a list of values to fill in. So, using the string formatting operator looks like this:

```
"template string" % (substitutions)
```

You have have as many substitutions (the `%s`, `%i`) as you like indicated in the format string. Values are taken in order from the list of substitutions that follow.

```
label = "Item 1"
width = 15
height = 6
print "<p>%s: %i x %i</p>" % (label, width, height)
```

This code will print this line:

```
<p>Item 1: 15 x 6</p>
```

The string formatting operator gives you a very quick and flexible way to create output, based on values in variables.

Topic 9.4 Debugging

Unfortunately, when you write programs, they usually won't work the first time. They will have errors or *bugs*. This is perfectly normal, and you shouldn't get discouraged when your programs don't work the first time. *Debugging* is as much a part of programming as writing code.

Section 1.3 and Appendix A in *Think Python: An Introduction to Software Design* cover the topic of bugs and debugging very well, so we won't repeat too much here. You are encouraged to read those before you start to write programs on your own.

Beginning programmers often make the mistake of concentrating too much on trying to fix errors in their programs without understanding what causes them. If you start to make random changes to your code in the hopes of getting it to work, you're probably going to introduce more errors and make everything worse.

When you realize there's a problem with your program, you should do things **in this order**:

1. Figure out where the problem is.
2. Figure out what's wrong.
3. Fix it.

Getting It Right The First Time

The easiest way to get through the first two steps here quickly is to write your programs so you know what parts are working and what parts might not be.

Write small pieces of code and **test them as you go**. As you write your first few programs, it's perfectly reasonable to test your program with every new line or two of code.

It's almost impossible to debug a complete program if you haven't tested any of it. If you get yourself into this situation, it's often easier to remove most of the code and add it back slowly, testing as you do. Obviously, it is much easier to test as you write.

Don't write your whole program without testing and then ask the teaching assistants to fix it. Basically, they would have to rewrite your whole program to fix it, and they aren't going to do that.

As you add code and test, you should temporarily insert some `print` statements. These will let you test the values that are stored in variables so you can confirm that they are holding the correct values. If they don't, you have a bug *somewhere* in the code you've written and should fix it before you move on.

Finding Bugs

Unfortunately, you won't always catch every problem in your code as you write it, no matter how careful you are. Sooner or later, you'll realize there is a bug *somewhere* in your program that is causing problems.

Again, you should resist the urge to try to fix the problem before you know what's wrong. Appendix A of *Think Python: An Introduction to Software Design* talks about different kinds of errors and what to do about them.

When you realize you have a bug in your program, you're going to have to figure out where it is. When you are narrowing the source of a bug, the `print` statement can be your best friend.

Usually, you'll first notice either that a variable doesn't contain the value you think it should or that the flow of control isn't the way you think it should be because the wrong part of an `if` is executed.

You need to work backward from the symptom of the bug to its cause. For example, suppose you had an `if` statement like this:

```
if length*width < possible_area:
```

If the condition doesn't seem to be working properly, you need to figure out why. You can add in some `print` statements to help you figure out what's really going on. For example:

```
print "l*w:  " + str(length*width)
print "possible:  " + str(possible_area)
if length*width < possible_area:
    print "I'm here"
```

When you check this way, be sure to copy and paste the exact expressions you're testing. If you accidentally mistype them here, it could take a *long* time to figure out what has happened.

You'll probably find that at least one of the `print` statements isn't doing what it should. In the example, suppose the value of `length*width` wasn't what we expected. Then, we could look at both variables separately:

```
print "l, w:  " + str(length) + ", " + str(width)
```

If `length` was wrong, you would have to backtrack further and look at whatever code sets `length`. Remove these `print` statements and add in some more around the `length=...` statement.

Topic 9.5 Coding Style (Optional)

Whenever you're writing a computer program of any kind, there's more to worry about than getting it to work. You should also make sure your program is easy for you or someone else to read and follow.

There are no fixed rules for creating a good program. It's a matter of style, just like writing good essays. But, there are some guidelines you should follow to get yourself started in the right direction:

- Use descriptive names for all of your variables and functions. Don't use variables names like `x` and `x2`. Try to describe what the variable or function is for: `total`, `calc_tax`
- Include comments (lines that start with `#`) to describe parts of your code that are hard to follow. For example:

```
# get the user's input and make sure it's a number
```

All of these tips will make it easier for someone else to read your code and also make it easier for you to come back and work on your code later. In addition, if your code is easier to read, it will probably be easier to debug.

▶ Look back at any code you have written. How easy is it to read? Have a look at someone else's code. Can you follow it?

Summary

This unit adds some more tools that you can use in Python to make your programs do different things.

Consider reading the optional topics, even though they aren't required. They provide useful information about programming.

Key Terms

- module
- boolean value
- boolean expression
- `if` statement
- `else` clause
- string formatting operator
- debugging
- coding style

Unit 10

Internet Internals

Learning Outcomes

- Identify the parts of a URL and their uses.
- Describe what cookies are and why they are used.
- Describe some things that can be done with HTTP besides simply transferring files.
- Identify uses of encryption on the Internet and outline why it is necessary.

Learning Activities

- Read this unit and do the questions marked with a "▶".
- Browse through the links for this unit on the course web site.

Topic 10.1 URLs

We have learned a lot about the web since we first discussed URLs in Topic 1.3. As we have discussed different things that can be done on the web, more has been added to our URLs.

Query String

You might have noticed that when we create forms and use them to access web scripts, the data from the form is transmitted as part of the URL. For example, if we had a form with two text inputs named `text1` and `text2`, as in Figure 8.1, we would submit it and see a URL link to this in the browser's location bar:

http://cmpt165.csil.sfu.ca/~student/sample.py?text1=Hi&text2=There

The part after the `?` is called the *query string*. It contains an encoded version of the CGI data. The different fields are separated by an ampersand, `&`. Note that if you want to include a URL like this in an `<a>` tag, you need to use the `&amp;` entity to encode each ampersand.

When you send CGI data this way, it will be recorded in logs of web traffic and will generally make a very long and awkward URL. You can avoid this problem by using the *post* method to transmit your form data. You just have to change the way your form submits:

```
<form action="sample.py" method="post">...</form>
```

This way there will be no query string in your URL, and the form data is transmitted out of sight of the user.

Fragment

In Topic 4.5, we discussed the `id` attribute and pointed out that these identifiers can be used as *fragments* or *anchors*. This is a way of creating a link to a position within a page.

To include an anchor in a URL, put it at the very end (after a query string), starting with a #. Suppose we have a page at the URL http://cmpt165.csil.sfu.ca/~student/page.html with an identifier `<h2 id="contents">`. To create a link that scrolls to this `<h2>`, we would use this URL:

http://cmpt165.csil.sfu.ca/~student/page.html#contents

Fragments can also be used in relative URLs on a web page:

```
<a href="../page.html#contents">...</a>
```

URL Encoding

In HTML, entities are needed to let us display characters like < that are used for special purposes in the language. We run into similar problems in URLs. The characters ? and # are used to indicate the query string and fragment, respectively. What if we have a file name or CGI data that contains a question mark?

There are also characters that aren't allowed in a URL, like spaces. In order to allow URLs to transmit these file names and form data, these characters must be encoded.

Characters that aren't allowed in URLs are encoded with the hexadecimal number of their character value. If someone entered "Who?" in a form, it would be encoded like this:

http://cmpt165.csil.sfu.ca/~student/test.py?name=Who%3F

The most common character you might see is a space, for example if someone entered "John Smith" into a form. A space is encoded as %20. The URL including the query string would look like this:

http://cmpt165.csil.sfu.ca/~student/test.py?name=John%20Smith

URL Encoding is also used for file names. If you upload a file with the name More Info.html, its URL will look like this:

http://cmpt165.csil.sfu.ca/~student/More%20Info.html

Fortunately, you don't usually have to worry about URL encoding. When you use websites, it's done automatically by the browser. If you ever need to create a link with encoded characters, you can just type it into the Location bar in Firefox, which will convert it automatically. You can also type the problem characters into a form and get the encoded version from the query string after you submit.

- Use the <a> tag and a URL with a query string to create a link to a web script script without using a form.
- Try `method="post"` in a form.
- Use an `id` attribute on a tag on a web page and use a fragment to link to it.
- Experiment with URL encoding by typing some punctuation into an HTML form and look at the query string that is used when you submit the form.

Topic 10.2 Cookies

When a user accesses web pages from your site, there isn't really any way for you to connect one page view to another. There's no way of knowing if the same user is following links from one page to another—all you know is which pages are being loaded. You can only guess that they are being loaded by the same user. In short, the web is *stateless*—every page is loaded independently.

This fact can cause problems if you need to follow a particular user from one page to the next. For example, suppose you were creating an online shopping application. You would need to keep track of the items that users placed in their "shopping carts" so you could give them the right total when they eventually got to the "checkout" page.

One way to keep track of a user on the web is to use *cookies*. Cookies are small pieces of information that a website can store on the user's computer. Once a web server has stored a cookie, it can retrieve its value and use it to determine who the user is, what is in their shopping cart, and so on.

Because they can be used to track users, many people don't like accepting cookies from all websites. Some companies, Doubleclick in particular, have huge databases of people's browsing habits that they have collected through their cookies.

Firefox includes sophisticated tools for managing who can set a cookie in your web browser. It also contains a "Cookie Manager" that you can use to view and delete the cookies stored in your browser.

Cookies can be created when a page loads, as part of the HTTP conversation between the server and client. Cookies come with an expiry date. They are stored on your hard drive until they expire.

There is a Python module named `Cookie` that can be used in CGI scripts to create cookies. We won't discuss it here, and you will not be expected to use it in this course.

▶ Use Firefox's "Cookie Manager" to see what cookies you have stored.

Topic 10.3 HTTP Tricks (Optional)

So far, we have only thought of HTTP as a way to transfer web pages from the server to the client. This is its primary job and all you usually have to worry about. But there are many other things that can be done with HTTP that are worth knowing about, particularly if you want to create "real" websites. You should keep these possibilities in mind and remember when they are needed.

Caching

If your web browser needs the same page twice, do you really need to transfer the whole thing over the network each time? You could store a copy on the hard drive and use it when you need it again. This is called *caching*.

For example, when you use the same CSS file for your entire site, a browser will load it the first time it views a page from your site, but after that, it can cache the file and use it as the reader browses your site, without having to reload it every time.

Caching decreases the amount of network traffic and makes pages appear faster. It is important for files like graphics and style sheets that appear on every page.

Caching can cause a problem if a file changes. Suppose a website's style sheet changes between visits. How is your web browser supposed to know whether it can use its cached copy or has to reload the style sheet?

A web browser can ask the web server to send a file *only if* it has changed since it last downloaded the file. It does this by sending information about the version of the file it has cached in the HTTP request. If the server has a newer version, it sends the file. If not, it just tells the browser that its cached version is up to date.

In addition to your web browser's cache, your ISP might have a cache server that stores web pages that their subscribers have recently accessed. This is one way they can decrease the amount of information that must be transmitted between them and the outside Internet and thus reduce their expenses. Cache servers can be entirely invisible to the users.

Web scripts can't usually be cached since their content could change every time they are viewed.

When you create web pages, you should keep caching in mind. For example, if you use the same style sheet file for every page, most users will only have to transfer it once. It will save them time and reduce your bandwidth costs. If you use a different style sheet on each page, every one will be loaded separately.

The same is true for images. If you use a single image (maybe a site logo) on every one of your pages, users probably won't notice the time it takes the image to transfer. If you use a different image on every page, the images will have to be loaded every time the user goes to a new page.

Redirects

As you have travelled the WWW, you have probably followed a link to a site and found a message like "We have reorganized our page, so you should go to...". You have probably also seen many "Not Found" errors because pages have moved to another location.

This is a sign that people don't understand HTTP or the features of their web server. People browsing a website should *never* have to see messages like this.

The web server can handle moved pages transparently. When the user requests a page, the web server can indicate that the page has moved to another URL. The web browser will follow the *redirect* without the user ever seeing it. Search engines also understand automatic redirects and will take them into account when returning search results.

Without an automatic redirect, all links from other websites or search engines will take users to the old page. Basically, when somebody else links to your website, it is a *good* thing and you don't want to break those links.

The way a redirect is set up depends on the web server you are using.

Content Negotiation

As you should know by now, different web browsers have different capabilities. Some can display SVG images; some can't. Some can display Unicode characters, and some can't. It's possible that every person browsing the web has a different combination of files that they consider acceptable.

There is a way of dealing with these differences using HTTP that isn't very well known. It is possible for a web client and server to exchange information about what files are acceptable and decide on the best version. This process is called *content negotiation*.

When a web browser requests a web page, it can send along information about acceptable information in four areas:

File type: The browser sends a list of MIME types that it knows how to handle, and a level of "preference" for each type. So, the browser might prefer an XHTML file, but if all that's available is a PDF, it will accept that and try to find a program to open it. Browsers that can display a SVG image might be sent that, while other browsers would receive the PNG version of the same image.

Language: The web browser should have a list of the languages that the user knows how to read and an order of preference. The web server will try to send a page that the browser can actually read. On multilingual websites, using these preferences lets different people read the website in different languages with no changes to the site and no effort for the user.

Character set: Some web browsers know about Unicode and some don't. There are also several other character sets used for various reasons that browsers may or may not be able to handle.

Compression: Most web browsers can automatically handle files that have been compressed, uncompressing them when they are received. This allows pages to load faster, since they are smaller and can be sent more quickly. Negotiating compression allows web servers to take advantage of this when it's available but to send regular uncompressed files when it isn't.

When these four kinds of information are combined, content negotiation allows web authors to create websites with many different visitors in mind. All visitors will see the content that is best suited to them.

Once content negotiation is activated in the web server, the only change that must be made on the website is to link to files *without* an extension. For example, instead of linking to `page.html`, the link would simply point to `page`. The browser and server would then negotiate for the XHTML file or another variant if one is available.

Content negotiation is not used frequently on the web, partially because most people don't know it is possible. Even for those that do, content negotiation presents problems.

When web browsers are installed, they usually assume that system's default language is the *only* one that the user can read. If users don't change this setting and go to a website written in another language, they may get an error message that says there is no page that's acceptable.

☞ Content negotiation has been used on the course website. If you have a non-English version of your operating system, you may have had to change your browser's settings to view the website.

A side benefit of content negotiation is that you don't have to use file extensions in your URLs. If you have used content negotiation and want to turn a static XHTML page into a Python web script, you would just have to replace the page.html file with page.py. Since all of the links only point to page, they will all still work correctly.

Topic 10.4 Security and Encryption (Optional)

Have a look at Figure 1.1 again. As we have just seen, there are many other computers between your home computer and SFU that pass the data you send from one to the other. You probably have no idea who runs these intermediate computers. There's no reason you would, and it's not easy to determine. Most people don't care.

All of the information that you send across the Internet passes through intermediate computers like these. It's possible for any of these computers to watch all of the information that passes through them and scan for things like passwords and credit card numbers that you want to keep secret.

☞ When the Internet was originally designed, there were only a few sites connected, mostly universities. Everybody knew and trusted the administrators at the various sites that might be responsible for passing along their data. Now, with millions of people sending information across the Internet at any time and thousands of service providers, blind trust isn't really an option.

It is possible to keep secrets, even when you are passing information around this way. The information can be *encrypted* so that only the intended recipient can decrypt it and read the contents. The computers in between can pass the information along, but they can't (easily) decrypt it to see what's inside. The details of how these *encryption* methods work is beyond the scope of this course.

Most information that is transmitted over the Internet isn't encrypted. HTTP (web) traffic and emails are sent without encryption.

Secure HTTP (*HTTPS* or *S-HTTP*) is a version of HTTP where all information is encrypted when it is transmitted. Using HTTPS ensures that any sensitive information you send will only be seen by the intended receiver. URLs that begin with https:// instead of http:// use the encrypted version of HTTP. HTTPS isn't always used because encrypting and decrypting data is extra work for web servers.

Most web browsers have a small icon in the bottom of their window to indicate whether or not they are using a secure connection. You should keep your eye on this icon when sending passwords and other sensitive information.

You may have used *FTP* (File Transfer Protocol) to transfer files over the Internet. So, you may have been wondering why you can't use FTP to transfer files to the course web server.

FTP isn't encrypted. Since you have to provide your password to upload files into your account, it would be sent unencrypted every time you logged in. The technical staff in Computing Science don't allow FTP for security reasons. The *SCP* (Secure CoPy, also called SFTP) protocol does use encryption, so it was chosen as an alternative.

It is also possible to encrypt email by adding an encryption program to your email program. The most common encryption programs for email are *PGP* (Pretty Good Privacy) and *GPG* (GNU Privacy Guard).

Whenever you send information like credit card numbers or sensitive passwords over the Internet, you should make sure you are using some kind of encrypted connection.

- You probably use some programs not listed here on the Internet. Do they encrypt the data they send? Are there secure versions?
- Try to find an encryption plug-in for your email client.

Summary

After completing this unit, you should have a better understanding of how some parts of the Internet work. As in Unit 1, you should understand what's happening behind the scenes when you use the WWW.

Key Terms

- query string
- fragment
- URL encoding
- cookie
- caching
- redirect
- content negotiation
- encryption

Part V

Appendix

Appendix A

Technical Instructions

Learning Outcomes

This material is intended to help you get over some of the technical hurdles necessary to get started in the course. You won't be tested on it.

Learning Activities

- Browse the "Technical" section on the course website for more information on the software used for this course.
- Install whatever software you need (when you need it).
- Get acquainted with the software needed for the course.

If there are any updates to these instructions, they will be posted in the "Technical" section of the course website.

This course and these instructions assume that you have a basic understanding of how to use your computer—you can open programs, load and save files, use menus and buttons, and so on.

Topic A.1 SFU Computing Account

If your SFU account isn't activated yet, you will need to do it for this course. You can go to http://my.sfu.ca/ to activate your account. Your campus email account will be receiving email from the course email list, so make sure you check it regularly.

Topic A.2 CMPT 165 Server Account

In addition to your regular SFU computing account, you will have an account on a web server set up just for this course. You will use the file space on this server for your assignments. It is set up to make the web programming we will be doing in the last part of the course as straightforward as possible.

This account will have the same userid and password as your SFU computing account. TRU Open Learning students will likely not have their account set up automatically. They should email the course supervisor as soon as possible to get it created.

See the instructions on the course website for information on transferring files to this web server.

You won't be able to access this server after you're done the course, so make sure you keep your own copies of the files you upload there if you think you might want them in the future.

Topic A.3 More Instructions

For more technical instructions, see the "Technical" section of the course website. Most instructions are there so they can be easily updated if the software changes.

Index

⇒, 124
*, 124
+, 124
../, 46
/, 45
<a> tag, 39
<address> tag, 86
<b> tag, 87
<big> tag, 87
<body> tag, 35

 tag, 86
 tag, 72
<form> tag, 128
<h1> tag, 35
<h2> tag, 38
<h3> tag, 38
<head> tag, 35
<html> tag, 35, 67
<i> tag, 87
<img> tag, 41
<input> tag, 128
 tag, 39
<link> tag, 54
<p> tag, 36
<small> tag, 87
 tag, 72
<style> tag, 52
<sub> tag, 87
<sup> tag, 87
<table> tag, 86
<title> tag, 35
 tag, 39
%, 141
 , 87
\, 45
1-bit image, 96
15-bit colour, 96
16-bit colour, 96
24-bit image, 95
8-bit image, 96
802.11, 22

<a> tag, 39
About Colours, 77
absolute URLs, 44
additive colour model, 77
<address> tag, 86
ADSL, 21
Advanced XHTML and CSS, 65
AirPort, 22
Alignment, 105
alpha channel, 98
anchors, 148
Another Perspective, 47
Another XHTML Page, 38
Attributes, 39

<b> tag, 87
backbone, 20

background, 58
backslash, 45
Basics of the Internet, 19
<big> tag, 87
binary transparency, 98
bit depth, 95
Bitmap vs. Vector Images, 92
bitmapped graphics, 92
block tags, 69
Block vs. Inline Tags, 68
block-level tag, 42
BMP, 97, 100
body, 139, 140
<body> tag, 35
Boolean Expressions, 139
boolean expressions, 139
boolean operators, 139
boolean values, 139
borders, 58
box model
    in CSS, 57

 tag, 86
bugs, 142

C++, 119
Cable modem, 21
Caching, 151
Cascading Style Sheets, 51, 52
case
    in URLs, 46
CGI, 130
cgi module, 130
CGI scripting, 122
Character Sets, 71
Choosing Tags, 48
class, 73
class selectors, 75
Classes and Identifiers, 73
clause
    else, 140
clear, 83
client software, 22
Clients and Servers, 22
closing tag, 34
    short form, 42
Closing Tags, 34
CMPT 165 Server Account, 160
CMPT 165 Web Server, 14
Coding Style (Optional), 144
Colour Depth, 95
Colours in CSS, 76
Combining, 46
comment, 132
Common Gateway Interface, 130
Common Image File Formats, 99
Common Mistakes, 86
components, 95
Compression, 96
computer graphics, 91
computer programming, 118
concatenates, 124
Connecting to the Internet, 20
content, 52
Content Negotiation, 152
Content-type, 123
contents, 34
contextual selectors, 76
Contrast, 107
Conventions, 110
Converting to Numbers, 134
Converting to Strings, 135
Cookies, 150
cookies, 150
Creating Python Web Scripts, 122
CSS, 52
    box model, 57

selectors, 75
CSS "Boxes", 57
CSS Details, 57
CSS1, 53
CSS2, 53, 83
CYM, 77

Debugging, 142
declarations, 53
deprecated, 40, 61
Design, 103
Design Principles and XHTML/CSS, 108
Directory and File Name, 45
dithering, 96
`<div>` tag, 72
doctype, 66
strict, 67
transitional, 67
document type, 66
drawing programs, 92
dynamic pages, 121

Editing XHTML, 32
`else` Clause, 140
Email, 24
empty tag, 42
encrypted, 155
Entities, 70
entity
named, 70
numeric, 70
EPS, 100
Ethernet, 21
Example, 111
execute, 119
Expressions, 120
Expressions and Variables, 123
External CSS, 54

Fetching a Web Page, 28
file extension, 27
File Formats, 94
native, 100
File Name Only, 45
Finding Bugs, 143
First XHTML Page, 32
`float`, 79
floating element, 79
floating point, 134
font
generic family, 61
Fonts in CSS, 61
`<form>` tag, 128
Forms, 127
forms, 128
Forms and Web Programming, 127
Fragment, 148
fragment, 74
fragments, 148
FTP, 24, 155

gateway, 20
General Design, 103
generic font family, 61
Generic Tags, 72
Getting It Right The First Time, 143
GIF, 99
GPG, 155
Graphics and Image Types, 91
Graphics and Images, 91

`<h1>` tag, 35
`<h2>` tag, 38
`<h3>` tag, 38

Handling Types, 134
<head> tag, 35
heading, 36
heading tags, 36
height
    in CSS, 58
How Web Pages Travel, 25
HTML, 31
    uppercase/lowercase, 34
    validating, 66
<html> tag, 35, 67
HTML validators, *see* validators
HTTP, 24, 25
    secure, 155
HTTP header, 123
HTTP Tricks (Optional), 151
HTTPS, 155
HyperText Transfer Protocol, 24

<i> tag, 87
id, 73
id selectors, 75
identifier selectors, 75
if Statement, 139
Images in HTML, 41
<img> tag, 41
index.html, 48
indexed colour, 96
Information on the Internet, 24
inline tags, 69
<input> tag, 128
Instant messaging, 24
Internet, 20
Internet Internals, 147
Internet service provider, 20
ISP, 20

Java, 119

JPEG, 99

Lengths in CSS, 59
 tag, 39
libraries, 137
<link> tag, 54
link state selectors, 76
list item tag, 39
lossless compression, 97
lossy compression, 97

Making Decisions, 138
Making Web Pages, 31
margins, 58
markup
    semantic, 48
Markup and XHTML, 31
Markup for Meaning, 84
markup language, 31, 33
meaning, 62
MIME type, 27, 123
MIME Types, 26
Modem, 21
module, 137
More Instructions, 160
More Programming, 137
More Selectors, 75
Moving Up, 46

named entities, 70
namespace, 67
native formats, 100
 , 87
Network gaming, 24
non-breaking space, 87
None, 132
numeric colour values, 77
numeric entities, 70
numeric expression, 120

opening tag, 34

`<p>` tag, 36
padding, 58
Page Design, 113
paint programs, 92
palette, 96
paragraph tag, 36
path, 26
Peer-to-peer file transfer, 24
PGP, 155
pixels, 92
PNG, 99
`position`, 83
Positioning in CSS, 79
post, 148
`print` statement, 120
Programming Introduction, 117
programming language, 118
property, 53
protocol, 23, 25
Protocols, 23
Proximity, 104
pseudoclass selectors, 76
Python, 119
Python Modules, 137

Query String, 148

raster graphics, 92
Readability, 112
Reading Form Input, 130
Redirects, 152
Relative URLs, 44
relative URLs, 44
Repetition, 105
required, 41
RGB, 77
rollover, 113
routers, 20
rule
    in CSS, 53
run, 119

S-HTTP, 155
scheme, 26
SCP, 155
Secure HTTP, 155
Security and Encryption (Optional), 154
Selecting by Class and ID, 74
selector, 53
selectors
    CSS, 75
semantic markup, 48
SFU Computing Account, 159
simple transparency, 98
slash, 45
`<small>` tag, 87
Some XHTML Tags, 35
`<span>` tag, 72
Starting With Python, 119
stateless, 150
statement
    `print`, 120
    variable assignment, 124
statements, 119
static pages, 121
strict doctype, 67
string, 120, 134
string expression, 124
String Formatting (Optional), 141
string formatting operator, 141
`<style>` tag, 52
`style` attribute, 57
`style` attribute, 57
Styles, 51

stylesheet, 52
<sub> tag, 87
subtractive colour model, 77
subtype, 27
<sup> tag, 87
SVG, 100

<table> tag, 86
tag
    empty, 42
    opening and closing, 34
tag selectors, 75
tags
    block, 69
    inline, 69
Technical Instructions, 159
template string, 141
text editor, 32
The clear Property, 83
The float Property, 79
The position Property (Optional), 83
The print Statement, 119
The World Wide Web, 19
TIFF, 100
<title> tag, 35
transitional doctype, 67
transitional tags, 87
Transparency, 98
triple-quoted string, 132
type, 27

 tag, 39
Unicode, 72
Uniform Resource Locator, 25
unordered list tag, 39
URIs, 25
URL, 25
    absolute, 44
    relative, 44
URL Encoding, 149
URLs, 25, 147

Validating XHTML, 66
validators, 66
value, 53
    attribute, 40
variable, 124
variable assignment statement, 124
vector graphics, 92
viewer, 33

W3C, 66
web browser, 22, 32, 34
Web Programming, 121
web programming, 122
web scripting, 122
web server software, 22
What Can CSS Do?, 53
What Is Programming?, 118
whitespace, 104
Why CSS?, 61
Why Do Markup?, 36
Why Learn to Program?, 118
Why Use External CSS?, 56
Wi-fi, 22
width
    in CSS, 58
wireless LAN, 22
Working with RGB, 77
World Wide Web, 19
WWW, 19

XHTML, 31
    strict, 67
    transitional, 67

validating, 66
XHTML Reference, 40
XHTML Tags, 33