Practical 2: Making and viewing basic HTML pages

Upon completion of this session you should be able to:

- create a basic HTML-formatted document
- add examples of the major text elements, their tags and their attributes: paragraphs, line breaks, headings, horizontal rules, and lists
- put the document in your **public_html** area on the Deakin system, either by saving it there or uploading it
- ensure that the proper permissions have been set to allow it to be delivered by the Deakin web server, and
- *validate* that document to ensure that it conforms to the *document type definition* specified at the top of the document
- use simple Javascript to enhance pages
- create simple PHP pages

Background

Uniform Resource Locators (URLs)

Every web 'document' (resource) must have a unique identifier that specifies its name, and where it is located (and thus from whence it can be retrieved). This specification is made up of the resource (file name); the area (directory) it is stored in; the name of the server where it is stored; and the Internet service that is used to retrieve it from that server.

For example, if this file were being accessed its URL might be

```
http://www.deakin.edu.au/sit104/practical2.html
```

Actually, the file is not there (it's in another directory) - so if you try to access it with this URL, the server will not be able to find it. In this situation, the server cannot send the file, so it sends back to the client (your browser) a page with a message to this effect. Such problems are identified by server error codes; this "File not found" message is error code 404, and is often just called a *404 error*. (Of course, you may just have mis-typed the file name, and the resource really is there - which is why it says *file not found*, rather than *file does not exist*!)
Some servers may try to help the user by listing files that it *can* find with similar names to the requested one, but most will just send the error message and it is up to you to find the correct URL and try again.

There is another common server problem: the file is where the URL says it is, but the server is not allowed to send it to the client. Remember that when a server sends a web document to a browser it is actually *copying* the original file - and the right to do this has to be set for any files to be delivered (otherwise the server is 'stealing' copies to send to browsers). If these access permissions are not set properly, the server will generate error code 403 ("You do not have permission to access this resource"), usually called a *403 error*. We will look at making sure these permissions are set for your 'public' area on the Deakin server later in this practical.

Basic structure of web documents

The development of the hypertext markup language (HTML) is evident in its emphasis on document structure, and its weaker support for what we might call *layout* elements. All documents that are to be 'marked up' in HTML are expected to be made up of one or more *structural elements*. These include:

- paragraphs
- headings
- lists
- rules (horizontal separators)

There are also 'meta-elements' that can include other elements (the body, the div and the span), and other constructs (notably tables and images). Crucially, HTML also provides various ways of embedding *links* to other documents (or to specific points within the current document, or another document); this is the primary mechanism for navigating between documents.

As we shall see in subsequent practicals, these provide a basic - but hardly comprehensive - set of features that we might wish to use. In particular, the perceived deficiencies of HTML for controlling layout and appearance have largely been addressed by style sheets (which we will look at in a later session).

Some basic principles govern the syntax and use of HTML.

  - HTML documents are text files (ASCII or Unicode encoded).
  - The basic structure of a document is indicated by *tags*, which are short text items within the 'pointy' brackets, < and >.
  - In current and future versions of HTML all tags are *containers*; that is, they should have a starting tag

(such as) and a matching closing tag ( ). This even applies to tags which do not have an obvious 'close', such as the line break
, images <img ...>, and the horizontal rule <hr>.
- The tags should all be lower-case (not <P>).
- Almost all tags have one or more *attributes* that apply to that tag, such as "id" and "style".
- The names and values of attributes are *always* in lower case; values are *always* enclosed in quotation marks.
- In tags that have numerical attributes, alternative number systems may be allowed (**absolute** values in pixels or points, for example, or **relative** values as percentages).
- Tags do not overlap, but they can sometimes be nested (one tag pair completely 'inside' another). There are complex rules that govern this; you should not have a paragraph or line break 'inside' a heading, for example, but you can make part of the text in a heading 'bold' by using a tag around it (but that must of course be closed before the closing heading tag, otherwise they would be 'crossed').
- Only content placed within the <body>...</body> container will be displayed (*rendered* is the correct term) by a browser. However, material in the <head>...</head> section - particularly scripts and style definitions - may also affect the displayed content.
- The particular set of tags (and their attributes and their values) being used in the document is specified in the <doctype ...> tag which should be the first line in an HTML document. This specifies the *document type definition* (DTD) being applied. Here is an example of a DTD (if you look at the page source of the page your are looking at in your browser, you will find it is indeed the very first line):

  <!doctype html public "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

  This is the definition of all the tags and their attributes that a validator will use to check if your HTML is 'correct'. It says that
  - the DTD is *public* (can be accessed and used by anyone)
  - should you (or more importantly a browser) wish to see it, a copy can be found at the URL given
  - it is the XML-compatible XHTML version 1.0
  - you might be using tags or attributes which were legal in earlier versions of HTML ('Transitional', rather than 'Strict'), but not in the version listed, and a validator should allow for that; the most important consequence is that this allows you to use both attributes and styles, even though Strict HTML 4.0 and XHTML 1.0 both *require* that you only use styles; they have the same set of tags.

  HTML 5 DTD is simple and more popular today. Please feel free to choose your HTML DTD and use valid tags defined by that DTD:

  <!doctype html>

Tasks

1. For this and subsequent sessions you can use a standard text editing program, such as Notepad or Notepad++. This is a perfectly acceptable 'low-tech' method, and tends to concentrate your attention on what you are creating.
2. If you are using a text editor, you will probably get just a 'blank' page. Cut and paste the following to make a minimal but 'complete' HTML page:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8">
<title>Untitled Document</title>
</head>
<body>

</body>
</html>
```

Look carefully at the document and make sure you understand its structure. Notice how

- all the tags (with the exception of the *meta* tag) are **containers**; that is, they have both an opening part (such as <body>) and a closing part (</body>)
- various tags are **nested** (all of one container is inside another) but they never 'cross'

Change the text inside the title tags from *Untitled Document* to something like **Session Two**. Save and name the document (make sure it has either htm or html as the file extension). If you are on-campus you

should have access to the **public_html** area in your home directory, so you can save it there. If you are doing this session elsewhere, you can use an FTP program such as Winscp to transfer the file to Deakin. Also, to minimise problems, it is recommended that you stick rigorously to the following placement and naming conventions:

- Create sub-directories in your public_html area and place files in them; try putting all your material for this unit in a folder called *sit104*, and each practical in a folder within that called *prac2* and so on.
- Make all your file and directory names *lower case*.
- Do *not* use spaces in file or directory names; if you want to 'simulate' a space to make long names more legible, use an underscore character (as in public_html). In my experience this creates numerous (avoidable) problems, as some servers will put a space character (%20) into the file name when they deliver it to a client, but some browsers don't properly recognise it.

3. Include at least one example of all the elements in the bullet list below *inside the body section* of your document.
    - paragraphs
    - line breaks
    - horizontal rules
    - headings (levels 1-6)
    - different types of lists (ordered, unordered, description (previously called definition))

   You will need some content, of course; you could just type stuff, download some text, or use the text from a Word document (save as text, *not* as HTML!). However, those with a sense of history and style might want to use the Lorem Ipsum site to generate the 'dummy' Latin text
4. As you develop the document, check the contents by loading it into a browser, or using the Preview mode in an HTML editor.

5. Making your web pages available to the world at large via the Deakin web server requires that various system-level access controls be set. When a user requests a web document from a server, it is really asking for a 'free' copy of the file; this is not allowed unless you (the 'owner' of the file) say it is. This means the server must be given *read* and *execute* access to the files (and the directories they are stored in). They must not, on the other hand, be able to delete or change your files (which requires *write* access). This can be done 'manually', by setting access privileges (using the Unix *chmod* command) on files and directories. You need to Telnet (by SSH) to your home area and apply the *chmod* command (with appropriate parameters, eg. chmod 644 prac2.html. If you get this done correctly, you should be able 'see' your document by entering its URL into your browser. If the access settings are not right, you will probably get a '403 error' page, indicating that the user does not have the right to see the page. If you get a '404 error' (file not found) you need to check the spelling and syntax of the URL you have entered.
   If your Deakin username were *fgow*, for example, and you had called your file prac2.htm, then the url would be

   `http://www.deakin.edu.au/~fgow/prac2.htm`

   Note that
    - you do **not** include the public_html as a directory: that is what the *tilde* (~) is for; it 'points' the server to your public_html area
    - if you put the file in a subdirectory (such as sit104) in your public_html area you need to include that directory name as well, to give

      `http://www.deakin.edu.au/~fgow/sit104/prac2.htm`

6. The final step is to **validate** your document to see if you have used only correct tags and attributes, and used them properly. The best way to to do this is to use an online validation service, of which the system provided by the World Wide Web Consortium (W3C) is the most widely-used. You **must** have a proper DTD, and a meta tag that specifies what character encoding system the file is using (the fifth line in the basic document listed earlier). The latest version of the W3C validator system works by your entering a full URL for your document if it is on a web server such as www.deakin.edu.au, or by simply uploading your file. If there are no problems with your document you should get a page with a message to that effect. If there are problems, you will get a page with some error messages which should help you identify and fix them.

Starting with Javascript

There are many situations - some trivial, some critical - in which we may want to use examples of *client-side scripting* to enhance the basic HTML-formatted content of a web page. Starting with this session, we will look at simple scripts that use Javascript (the most widely-used client-side scripting language) to 'liven up' web documents. We can use scripts to, for example,

- get input from readers (which we will do in this session)
- display simple messages to readers (which you will see an example of when try the quiz at the bottom of this page)
- display system information (such as time and date)
- create popup windows
- create image 'rollover' effects

Later, we will turn our attention to a *much* more significant use of scripting: to *validate* (check the contents of) web forms when users fill them in.

Let's start with something simple. Suppose we want to 'personalise' a page by asking the reader for a name and using that name inside the page; here's how we can do this:

1. Add a *script* (piece of code) to a normal HTML page that tells the browser to 'pop up' a data entry box (a *prompt*) for the user to enter their name; usually this is placed in the <head> section , which is where you should put the following three lines of code:

```
<script type="text/javascript">
   var sName = prompt("What is your name?","")
</script>
```

The prompt actually has two strings associated with it. The first ('What is your name?') is how you tell the user what you want them to input. The second is used if you want some default content; that is, something automatically displayed in the text box that the user can overwrite or just accept. For example,

```
<script type="text/javascript">
   var sName = prompt("What is your name?","Batman")
</script>
```

2. Add some 'code' to the *body*of the page that tells the browser to display the entered name within the text of the page at the appropriate point:

```
Hi <script type="text/javascript">
document.write(sName)
</script>, welcome to the SIT104 home page.
```

Put these pieces of code into the document you made earlier (or create a new one just for this purpose - but make sure it is a valid HTML document!). Load the page into your browser and see what happens. You should notice several things about the code:

- Placing the first 'snippet' of code in the head section of the page ensures this bit of script is executed when the page is loaded, and so *before* the body contents are displayed. It tells the browser to 'pop up' a text input dialog (*prompt*) and stores the contents - what the user types in - into a variable called *sName* (variables in Javascript are *case sensitive*; sName is different from SName - change it and see what happens). Because the user types in text, the variable sName is by default a *string*.
- The second code 'snippet' tells the browser to display the contents of the variable sName. Programmers will probably have noticed that Javascript looks a lot like C and C++, and it does have many things in common with those languages. The syntax and keywords are very similar, and it has common data types (integers, strings, arrays ...). If you were to put the fist snippet after the second one (in the body pf the page rather than in the head), you would get an error because you would be trying to use (display) a variable that does not yet exist (it is being declared in the *var* statement, not just getting a value). Try it and see.
  Some programmers among you may even have worked out that what it is actually happening is that the second snippet is invoking the 'write' *method* of the 'document' *object*, and passing the string variable

sName as a parameter. Javascript is indeed a (fairly primitive) object-oriented language.

Arrgh!!! I can almost hear the screams of those of you who hate programming. Take a lesson from the *Hitchhiker's Guide to the Galaxy*: **DON'T PANIC**. All of the coding we will do in Javascript will be based on using and 'playing with' code snippets (like these) with which you will be provided; you can look up more about them - and find many others - in one of the recommended texts. As we look at different examples of scripting over the next eight or so sessions, I am sure that you will find that using scripts is not as difficult as you might at first fear. As an example, the little self-help quiz at the bottom of this page (and there will be one at the end of every practical) is 'managed' with Javascript. When you click on the 'Check my answers' button, a Javascript *function* (more C++) is executed that checks your choices against the right answers. (I know it's not very secure - but it's a quiz, not a test!) Have a look at the source for this page to see what this function looks like; it's in the head section. If you don't understand all the code now, don't worry; you should be able to understand all of it well before we reach the end of the unit.

Starting with PHP

The other major form of scripting that is used on the Web is *server-side* scripting. Here we also add code to the basic HTML document, but it is now 'executed' by the server *before it is sent to the browser*. This means that we can create *dynamic* web pages, whose content might be different each time the page is delivered - as opposed to standard HTML pages, whose contents will normally only change if someone edits them directly.

In this unit we are going to use server-side scripting for two purposes:

1. To enhance web page content in ways that are not possible just using client-side scripting.
2. To access information unavailable to the client - specifically, to access *databases*, and use information from such databases in our pages.

There are several important server-side systems, and we will look at them more deeply in lectures later in the unit. For the practical side of the unit, we are going to use **PHP**. It was designed to enhance web pages, which is why it is called PHP; it was originally an acronym for **P**retty **H**ome **P**ages, but we now use the (much less entertaining) **H**ypertext **P**re-**P**rocessing.

Like Javascript, PHP is a programming language, but it necessarily vastly more powerful and flexible than Javascript. Again, the message is - don't panic; you will be provided with code to use and modify, and we will steadily build your understanding of the system through these weekly snippets.

The first and most obvious thing to notice about PHP documents is that we have to make sure that the web server knows that they are to be processed before they are sent. Actually, we are assuming that the server actually knows what to do with PHP files; indeed, processing PHP files is not a 'normal' part of job of web servers, so we need to know that this support is available. The good news is that the main Deakin server has PHP support (as we will soon prove), and so do most web servers.

One critical consequence of this is that you can't test PHP pages by loading them into a browser; they have to be *delivered* from a server. In these first few sessions we will use the Deakin server {www.deakin.edu.au}, but later we will explain how you can develop PHP 'off-line', either by installing a simple *interpreter* that checks the PHP and simulates a server's responses, or by installing a real, PHP-enabled web server on your computer at home and developing pages properly without Internet access. For now, however, we will assume that you are saving these files onto your 'home area' (on-campus students) or uploading them using FTP (off-campus students).

All we have to do get a web server to treat a page as PHP rather than HTML (and to process it) is to give it a name with a .php extension. If there is no PHP, just normal HTML, the page is sent as usual; copy one of the files from earlier in the session, change the extension to .php, upload it to the Deakin server, and check that it is unchanged. Don't forget that you must use a full URL:

```
http://www.deakin.edu.au/~myusername/sit104/prac2.php
```

Now let's create a 'real' PHP page with some code in it. Create a new file with a default (valid) structure; it should look something like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8">
<title>Untitled Document</title>
</head>
<body>

</body>
</html>
```

All our PHP code will go into the body of the document.

- Enter the following code into the body section

```
<?php echo "<h3>Hello, my name is Joanna Kerr</h3>"; ?>
```

  and check that the page looks the way you should expect. Obviously, put your own name in! The **echo** statement is PHP's basic way of 'writing' output, and it can contain HTML tags as well as text and variables (more in later sessions!).

- Create another new 'dummy' page and enter the following into the body section

```
<?php
echo 'This server is running PHP version: ' . phpversion();
?>
```

  What version of PHP is the Deakin server running? (Notice also the different *layout* of the code, which is spread over several lines; it is of no significance to the server, but you might find it more readable.)

- Create another new 'dummy' page and enter the following into the body section

```
<?php phpinfo(); ?>
```

  What output does this produce (it's pretty spectacular for one line of code!)? Look especially at the sections headed *Environment*, *Apache Environment* and *PHP Variables*, and notice the extra *modules* that this server has installed (including mysql and Oracle). All PHP-enabled web servers have the same core functionality (if they are the same version - this changed between PHP version 4 and 5), but you can add other functionality if you run your own server. For example, the server in my office is a Windows version of the Apache web server; running this allows me to add support for ASP (Active Server Pages) and JSP (Java Server Pages), the other two key server-side scripting systems, to Apache.
  Actually, you do not need to install your own server to test your PHP pages 'at home'; you could install a PHP *interpreter* that works with the server if it is there, but that can be accessed from an editor to check if your pages are working. This will even work for database access, of the kind that we will get to in a few weeks and required for Assignment Two. Instructions on where to get this software - and how to install it - will be provided in a couple of week; but remember that you do not need this setup to do the practical work and assignments, as it is all available through the Deakin computing environment. If you want to create a setup 'at home' that can do the things that we are going to do with PHP, then we will help you do that; if not, don't worry, you don't need to. Setting up servers is **not** part of this unit, but it is really not too hard.

- Create another new 'dummy' page and enter the following into the body section

```
<?php

// Prints 'today'
echo date("l");

// Prints current date and time, e.g. Wednesday 15th of January 2003 05:51:38 AM
echo date ("<p>l dS o F Y h:i:s A");

// Prints something like July 1, 2000 is on a Saturday
echo "<p>July 1, 2000 is on a " . date ("l", mktime(0,0,0,7,1,2000));

?>
```

  We are using two built-in PHP functions, *date* and *mktime*. The first is pretty straightforward; it tells us the current date and time. The second is more complex; in the words of the PHP manual, it "Returns the Unix timestamp corresponding to the arguments given. This timestamp is a long integer containing the number of seconds between the Unix Epoch (January 1 1970) and the time specified."
  Okay, that's pretty unintelligible but ... fortunately, all you need to notice is that the last three values in the function [**7,1,2000**] correspond to the date we are interested in (July 1st 2000 ==> month 7, day 1, year 2000). Modify this to check what day Christmas (or your birthday, or some other special day) falls on in, say, 2010.
  Again, the layout (empty lines between statements) is not significant, but may make the code more legible. The lines starting with double slashes // are PHP's way of including comments in the code; any code that follows (to the right of) these is ignored by the server. To comment out larger blocks of code you could put them at the beginning of every line but, as in C, there is a more efficient way:

```
/*
echo "None of these statements will be executed.";
z=34;
echo "Go Giants!.";
*/
```

- Create another new 'dummy' page and enter the following into the body section

```
<?php

putenv( "TZ=Australia/Melbourne");
$datestring = date( "H:ha F jS Y ", getlastmod());
echo "Last modified: ".$datestring;

?>
```

What information is being displayed here? When might it be useful to have this information on a web page or site?

Finally, there is no reason why we can't mix server-side *and* client-side scripting; we use PHP to create our pages, and include in them Javascript that is 'executed' when the page reaches the browser; we will do this in later sessions, and you may well choose to do it in your second assignment.