SDPublisher: Getting Started and Reference Documentation

Reference documentation, v. 1.1. 21 September 2009.

1. Getting started with Scholarly Digital Publisher

This is the documentation for SDPublisher, to accompany the public release of SDPublisher 1.1 in September 2009. Areas where further change is likely or desirable are noted in red text. Patience!

Comments, suggestions: to Peter Robinson

This document sets out a series of exercises introducing the Scholarly Digital Publisher ('SDPublisher') system. By the end of this series of exercises, you will:

- Have seen how SDPublisher works
- Have met most of the commands which enable SDPublisher to work
- Have seen enough of SDPublisher to know if it can do what you want it to do (if the answer is not 'yes' we would like to know what it cannot do) - and to decide if you like the way it does it
- Have learnt enough to make your own electronic publications with SDPublisher.

1.1 Introducing SDPublisher

SDPublisher can be thought of as a next-generation version of the Anastasia Publishing System, also developed by people associated with Scholarly Digital Editions, UK: see www.sd-editions.com/anastasia. SDPublisher retains the distinctive model of Anastasia: it sees XML as a stream as well as a hierarchy. Accordingly, it does not fit either of the common models of XML processors: it is neither a 'push XML' nor a 'pull XML' system, but a 'stream XML' system. It is therefore highly suited to processing documents characterized by multiple overlapping hierarchies: showing a book by pages, or by chapters, for example. Another distinctive feature of SDPublisher is that **you do NOT have to use XSLT** (though you can if your really want to). For many of us, this is a notable advantage.

SDPublisher incorporates the 'Pixelise' application first developed by Andrew West, and carried on further by Andrew, Zeth Green, Ralph Janke and Peter Robinson. Andrew West was responsible for the key decisions, to use Berkeley DB XML as the 'back end' database, to use Python as the scripting language, and to use the Django framework as the Python environment. As of 28 April 2009, the code for SDPublisher was at pixelise.org.

While SDPublisher shares the same model of text as does Anastasia, it is different from Anastasia in almost every other respect. Anastasia required that the XML be transformed into a series of binary files before it could be published (using the 'GroveMaker' application), which meant that even the smallest changes to the document required complete reprocessing of all the data. Further, the publishing system of Anastasia was built as a C-language module compiled into Apache 1, and used TCL as a scripting language. This meant that Anastasia could only run with Apache webservers, and indeed the restriction to Apache 1, now outdated, together with the use of 'C', made maintenance of Anastasia increasingly difficult. Further, TCL is not a popular language. In contrast, SDPublisher:

- Uses a database for all data storage. This means document fragments can be updated dynamically and the results seen immediately. By default, SDPublisher uses Berkeley DB XML, an exceptionally powerful and robust native XML database (www.oracle.com/database/berkeley-db/xml/). However, we have designed SDPublisher so that one could use another XML database, or indeed any database at all.
- Uses Python as the program environment and language. This means that SDPublisher could

run on any server supporting Python: that is, all major server systems. Python is a very popular and powerful language.
- Uses the Django framework in Python. This is a very widely-used publishing system, much-favoured in high-traffic websites.

1.2 Starting out with SDPublisher; what you need

We suggest you work your way through this document first. This contains a simplified introduction to show how SDPublisher works. After you work your way through this, you could go to the Reference documentation section, for a more formal account of the functions and tools available in SDPublisher.

You will need the following:

- Networked access with an up-to-date web browser
- A reasonable knowledge of XML
- A text-only editor, for example NoteTab on Windows or BBEdit on the Macintosh. Do NOT try and use Microsoft Word.

This documentation has been written for Macintosh OS X, Windows, and Linux systems.

1.3 Getting ready to start

First, you have to have get an appropriate version of Python. As of 21 September 2009, for Macintosh this was 2.5 or later (up to 2.6.2: we have not tested SDPublisher with the in-process Python 3). For Windows, you have to use a 2.5 version (2.5.4, for example). If you have Windows, you have to get it as follows:

- Download and install Python 2.5 (2.5.4; not 2.6 or later) from http://www.python.org/. By default, Python will install in C:\Python25 (for Python 2.5). Currently (September 2009) the Python bindings for Berkeley DB XML only work with Python 2.5.
- You will need to run Python from the command prompt. To do this, start the command prompt application and type 'set path=%path%;C:\python25' at the prompt, followed by return. In Windows XP, you get to the command prompt by choosing 'Run' from the start menu and typing 'cmd'.
- Check that Python is correctly installed by typing 'Python' at the prompt. If all is well, you will receive cheering messages

If you have a Mac with later than 10.5, no trouble: Python 2.5 is already installed. You could also install 2.6.2 over this, if you wish, from http://www.python.org/. It will automatically be installed in the right place. You can check that Python is correctly installed by typing 'Python' at the terminal prompt. (We have not tested SDPublisher on Mac systems older than 10.4.)

Second, you have to get Berkeley DB XML. Download and install from http://www.oracle.com/technology/software/products/berkeley-db/xml/index.html. As of September 2009 the operational version for Windows was 2.4.13: install this version, not the later 2.4.16 or 2.5.13. Installation on a Mac is much more demanding. There is no binary, so you have to compile it yourself from the terminal. You will need to have the Mactinosh developer tools, including the gcc compiler: get these from http://developer.apple.com/Tools/ (warning! this is a 1 GB download!). Then download the Unix/Posix version of DB XML; cd into the toplevel of the downloaded directory, type 'sh buildall.sh' and wait several hours. As of September 2009, 2.4.16 worked on Macintosh 10.5; we have not yet tested 2.5.13.

Third, you have to install the Python bindings for DB XML. This is easily done in Windows. In the folder 'C:\Program Files\Oracle\Berkeley DB XML 2.4.13\python' you will see the

executable 'dbxml-2.4.13.win32-py2.5'. Just double click on that and the bindings will install into the Python 2.5 installation. On a Macintosh, you will need to cd into the 'dbxml/src/python' folder and type 'sudo python setup.py install' (and be prepared to wait some time).

Fourth, you have to get Django, from http://www.djangoproject.com/download/. As of September 2009, the latest official version was 1.1. Download the Django-1.1-final.tar.gz file from here. For Windows, you will need PKZip, or WinZip, or similar, to unpack this into the folder Django-1.1-final. You should have this folder in your C:\Program Files folder. Now, install Django. You may be able to do this just by double-clicking on the 'set up' icon in the Django-1.1-final folder (however, this often fails). Alternatively, do this from the command line by cd-ing into the folder ('cd C:\Program Files\Django-1.1-final') and then typing 'python setup.py install'. On the Mac, the installer should do all this for you.

Finally, you are ready to get SDPublisher. SDPublisher is distributed as a single folder, from a link in www.sd-editions.com/SDPublisher, or directly at http://www.sd-editions.com/SDPublisher/SDP110/SDPublisher.zip. (If you just wanted the latest Pixelise, without the SDPublisher wrapping, you could get it by typing 'bzr branch lp:pixelise' into the terminal; you need Bazaar installed for this.) Download that folder and uncompress it somewhere accessible. Now cd into that folder ('cd SDPublisher'). You can (at last) see something happen now by typing 'python manage.py runserver' at the prompt. You will get various messages, to the effect that a 'development server is running at http://127.0.0.1:8000/'. You can check that it actually is by starting an internet browser and typing the address ' http://127.0.0.1:8000/' into the browser. You should see a Django 'page not found' page. We are getting close! Type ' http://127.0.0.1:8000/SDPintro/text'. You will see this documentation appear in the browser window, as a SDPublisher electronic book.

Before we go on, let's look at what we have downloaded. You will see in the SDPublisher folder the following:

- __init__.py (and possibly __init__.pyc): files used internally by Python/Django. You should never need to deal with these
- manage.py: this file gives access to various management functions in SDPublisher. See 3.1 below
- origin: this folder contains the material (from the first three chapters of Darwin's 'Origin of Species') used in the next sections of this tutorial to introduce how a SDPublisher digital book is made
- pixelise: this folder contains the core 'pixelise' application: the XML processing engine used by SDPublisher. Pixelise, named after Andrew West's cat, is the core application within SDPublisher
- SDPintro: this folder contains this documentation, presented as a SDPublisher electronic book
- settings.py: this file holds information needed by Django and Python, governing how SDPublisher works
- urls.py: this file tells Python and Django about the URLs used by SDPublisher

Open up settings.py. Most of this is internal information used by Django and Python, which you do not need to know about. You do need to understand the section 'INSTALLED_APPS':

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'SDPublisher.pixelise',
```

```
    'SDPublisher.SDPintro',
    'SDPublisher.origin',
)
```

The first four lines are used by Django. The last three lines concern us:

- SDPublisher.pixelise: this tells Django about the pixilise application, contained in the folder 'pixelise'
- SDPublisher.SDPinto and SDPublisher.origin: this tells Django about the SDPublisher books 'SDPintro' and 'origin', which Django thinks are actually applications contained in the folders 'SDPintro' and 'origin'

1.4 Your first SDPublisher publication: showing the text

Now, we are really ready to go! We are now going to make our first XML SDPublisher-powered application, using a specimen XML file 'origin.xml', containing the text of the first three chapters of the 1859 edition of Darwin's 'On the Origin of Species'. The 'origin' folder contains the 'origin.xml' file and various other useful files and folders to help you on your way:

- __init__.py and models.py (and possibly __init__.pyc, models.pyc): files used internally by Python/Django
- origin.xml: the source XML for this tutorial publication, encoded in TEIlite form. You could open this up to check that indeed what we have is XML
- origin.dbxml: the DB XML database ready to receive the XML for the 'origin' publication. This should be empty when you start
- pixelates: a folder containing a single subfolder, 'origin', containing a file 'base.py'. This file holds what we call 'pixelates': instructions on exactly how individual XML elements should be presented.
- templates: a folder containing a single subfolder, 'origin', containing various html files. This file are actually html templates, providing the frame for the html we will create from the origin.xml
- urls.py: this file is used together with the 'urls.py' file in the parent SDPublisher folder to construct the URLs for the 'origin' SDPublisher book
- views.py: this file holds the key processing scripts used by SDPublisher for this book
- www: this folder contains materials, such as static HTML and other files, which might be needed by the publication

First, load the file origin.xml into your new publishing system by typing 'python manage.py putxml origin -f origin/origin.xml -n text' at the terminal prompt (make sure you are in the SDPublisher directory). This should load 'origin.xml' into the Berkeley DB XML dabatase system, ready for you to publish. It is possible that this file is already included in the database, in which case you will get an error message 'origin/origin.xml is already in the database. Skipped.' In that case, you could either carry straight on, or type 'python manage.py rmxml origin -n text' into the terminal to remove it, followed by 'python manage.py putxml origin -f origin/origin.xml -n text' to put it back.

We can start work on this publication. We need to create an appropriate URL for our publication. First, you have to edit the urls.py file in the 'SDPublisher' folder. Open this and add the line

```
(r'^origin/', include('origin.urls')),
```

to the urlpatterns variable, so that this reads, ignoring the lines commented out by #:

```
urlpatterns = patterns('',
```

```
    (r'^SDPintro/', include('SDPintro.urls')),
    (r'^origin/', include('origin.urls')),
)
```

This line tells SDPublisher that when a URL request arrives beginning '..origin/', look in the file 'urls.py' inside the 'origin' folder for the rest of the URL. So now, we have to edit the second 'urls.py' file, contained in the 'origin' folder. In that file, add the line

```
(r'text/', 'origin.views.text'),
```

to the urlpatterns variable, so that this reads, ignoring the lines commented out by #:

```
urlpatterns = patterns('',
    (r'text/', 'origin.views.text'),
)
```

The effect of the two urls files is to link the URL '/origin/text/' with the Python function 'origin.views.text': this is the function 'text' in the file 'views.py' in the folder 'origin'. We now need to restart the server to see these changes take effect. Stop the server by using control-c in the terminal/command prompt window, and restart it with 'python manage.py runserver'. If you now type 'http://127.0.0.1:8000/' into your browser, you should be told 'Page not found' and told that Django expects a URL beginning 'origin/'. So this time, try 'http://127.0.0.1:8000/origin /text/'. You will get a different message: Django has recognized the url, but gives the error "Tried text in module origin.views. Error was: 'module' object has no attribute 'text'".

We have to edit the 'views.py' file, in the 'origin' folder. Open it, and you will see the following:

```
from pixelise.core import Collection
from django.shortcuts import render_to_response

def foo(request):
    p = Collection(request, 'origin')
    results = p.query("//text")
    if results.hasNext():
        text = results.next()
    else:
        return render_to_response('origin/error.html', {'message': "Can't find text element"})
    text_content = p.process_element(text, 'origin/base.py', False, None)
    return render_to_response('origin/text.html', {'page_content': text_content})
```

SDPublisher expected there to be a function 'text' in this file. There is no such function. Change the line 'def foo(request):' to 'def text(request)' and save the file. Now there is a function 'text', and things should begin to happen.

Access 'http://127.0.0.1:8000/SDPublisher/origin/text/' again. This time, you should see the text of the first three chapters of the 1859 editon of Darwin's 'Origin of the Species' leap out at you

We are making progress. This is what 'views.py' does:

- The first two lines import functions SDPublisher needs to operate, from Pixelise and Django.
- The line "p = Collection(request, 'origin')" opens up the 'origin.dbxml' database we have made, holding all the xml (try replacing 'origin' by 'SDPintro' in this line to see what happens! why?)

- "results = p.query("//text")" goes and finds the 'text' element in the database (try replacing 'text' by 'TEI.2' or 'teiHeader' or 'XXX' to see what happens)
- The next lines test if the element sought has been found. If it has, the element found is put in the variable 'text'. If it has not, we are sent an error message.
- The next line (containing the process_element call) is the core of SDPublisher. The element is sent for processing by the file 'base.py' in the 'origin' folder within the 'pixelates' folder, and the results put in the variable 'text_content'
- The last line is the Django 'render_to_response' variable: this sends the result back to the browser, using the template file 'origin/text.html' and assigning the content of the 'text_content' variable to 'page_content'

1.5 Your first SDPublisher publication: formatting elements

So far, we have done nothing with the XML except return the raw text, which looks rather ugly. To refine it, we have to use the XML itself to determine how each element should be presented. We do this through a 'pixlelate' file: in this case, the 'base.py' file, found in the origin/pixelates /origin folder. (Notice that this file is placed in the subfolder 'origin', not directly within the 'pixelates' folder. This is to ensure that we use the 'base.py' file for 'origin', not any other 'base.py' file for any other book. This could actually be very useful: if you wanted all your publications to use the same 'base.py' file, just put it directly in a 'pixelates' folder for any book -- for example, in a 'pixelates' folder within the 'pixelise' folder -- and it will be found by every book.)

Open that file. At the beginning, you will see the lines:

```
PIXELISE_PATTERNS = {
    'div': 'div',
}
```

The PIXELISE_PATTERNS structure is how SDPublisher associates XML elements with functions. Here, the XML element 'div' is associated with a function called 'div'. Add to this structure the line

```
'p': 'p',
```

so that it appears:

```
PIXELISE_PATTERNS = {
    'div': 'div',
    'p': 'p',
}
```

Now, we define what we want to do with the XML p elements. Add this to the base.py file, below the PIXELISE_PATTERNS statement:

```
def p(element, state, context):
    if state == 'begin':
        html = "<p>"
        return html
    if state == 'end':
        html = "</p>"
        return html
```

This function will be called every time a element is met as the SDPublisher traverses the document. It is called with three parameters:

- The element itself, as a Python object with a distinct set of properties (see below)
- State: this is one of 'begin', 'content' or 'end' depending on whether we are at the start of the element itself, dealing with its content, or at the end of the element
- Context: this is a Django response object. This can be very useful in some contexts: you can, for example, have one function store information by writing it to the response object and another function can then extract that information, effectively as a global variable. See the Django documentation on this.

The next lines say simply: if we are at the beginning of the element, insert a into the HTML stream; if we are at the end of the element, insert a .

Here is a more complex instance, which begins to show the power and ease of SDPublisher. We want the title of the whole document to appear in larger, bold type. The title is held in a <head> element, which is the child of element with an attribute 'type' set to 'book'. This code tests whether a <head> element is the child of a element with an attribute 'type' set to 'book'; if it is, it places the text within a HTML <H1> element. Here is the code which enables this, in the views.py file. First, you need to include this line in the PIXELISE_PATTERNS declaration:

```
PIXELISE_PATTERNS = {
    'div': 'div',
    'p': 'p',
    'head': 'head',
}
```

Then you need to have the function for treatment of <head> elements, as follows:

```
def head(element, state, context):
    isbookhead = False
    html=""
    div = element.get_parent_element()
    try:
        if div.get_attribute_value('type')=="book":
            isbookhead = True
    except:
        return
    else:
        if state == 'begin' and isbookhead:
            html = "<H1>"
        if state == 'end' and isbookhead:
            html = "</H1>"
        return html
```

The first two lines of this function set the variable 'isbookhead' to the default value of 'False', and initialize the variable 'html' as an empty string. The third line locates the element which is the parent of this <head> element, and puts it in the variable 'div'. The next lines get the value of the attribute 'type' on this element and tests: is it equal to 'book'? If so, the variable 'isbookhead' is set to 'True'. Note that this is done within a 'try/except' block: this is to catch the case where the element has no 'type' attribute. For cases where 'isbookhead' is 'True', the following lines place the HTML <H1> element around the text of the header.

1.6 Your first SDPublisher publication: viewing by chapter

So far, we have been doing rather simple things. Let's move on. The 'Origin' is divided into chapters, and an obvious step would be to let the reader view one chapter at at time. Here is how we do this.

First, we need to add a line to the 'urls.py' file, so that SDPublisher can pick up calls by chapter number. We add this line to the 'urls.py' file in the 'origin' folder:

```
(r'chapter/(?P<chapter>[^/]+)$', 'origin.views.chapter'),
```

to the urlpatterns variable (so that this reads, ignoring the lines commented out by #)

```
urlpatterns = patterns('',
    (r'text/', 'origin.views.text'),
    (r'chapter/(?P<chapter>[^/]+)$', 'origin.views.chapter'),
)
```

This activates the URL "http://127.0.0.1:8000/origin/chapter/xxx" and links it to a 'chapter' function in the views.py file, passing this function the string 'xxx' as the value of a 'chapter' parameter. We now need to write a 'chapter' function to pick up and process this url call, in views.py:

```
def chapter(request, chapter=None):
    print "%s" % (chapter)
    p = Collection(request, 'origin')
    results = p.query("//div[@n='CH%s']" % (str(chapter)))
    if results.hasNext():
        text = results.next()
    else:
        return render_to_response('origin/error.html', {'message': "Can't find chapter %s" % (s
    the_content = p.process_element(text, 'origin/base.py', False, None)
    return render_to_response('origin/text.html', {'page_content': the_content})
```

Now, type "http://127.0.0.1:8000/SDPublisher/origin/chapter/1" into your browser, and you should see the whole of the first chapter appear; chapter two will appear for "http://127.0.0.1:8000/SDPublisher/origin/chapter/2"; etc. (Note the use of the "print" statement to write the chapter number out to the command window. This is a very useful diagnostic device, similar to the use of 'puts' in Anastasia.)

1.7 Your first SDPublisher publication: viewing by page

All XML publishing systems can (or should) do what we have just done. The next example shows what is special about SDPublisher (and its predecessor, Anastasia).

The 'Origin', like every other book (and indeed most manuscripts), is divided into pages. Obviously, one would like to view the book page by page. It is exactly this which most XML systems make difficult. The second thing everyone learns about XML is that it has problems with overlapping hierarchies. As almost every edition of a text from a primary source is going to have overlapping hierarchies (text divided into chapters, paragraphs, sentences; but the primary source spreads the text across different pages, with the breaks between pages not corresponding with the breaks in the text) this is a problem. Thus: every chapter break in the 1859 'Origin' falls on a page break. But, the paragraph and sentence breaks do not:

paragraphs and sentences run across page breaks.

SDPublisher, like its predecessor Anastasia, is designed explicitly to cope with this situation. Essentially, unlike virtually every other XML processing tool, SDPublisher (like Anastasia) treats the XML not as a tree, but as a stream. Thus, it is possible to start processing at any point in the stream and end at any point in the stream: for example, to start at the beginning of a page and end at the end of a page.

So, let's make this happen. We are going to ask SDPublisher to show us one page, and only one page, of the 'Origin'. First, we need to add a line to the 'urls.py' file, so that SDPublisher can pick up calls by page number. We add this line to the urls.py file:

```
(r'page/(?P<page>[^/]+)$', 'origin.views.page'),
```

to the urlpatterns variable (so that this reads, ignoring the lines commented out by #)

```
urlpatterns = patterns('',
    (r'text/', 'origin.views.text'),
    (r'chapter/(?P<chapter>[^/]+)$', 'origin.views.chapter'),
    (r'page/(?P<page>[^/]+)$', 'origin.views.page'),
)
```

This activates the URL "http://127.0.0.1:8000/SDPublisher/origin/page/xxx" and links it to a 'page' function in the views.py file, passing this function the string 'xxx' as the value of a 'page' parameter. We now need to write a 'page' function to pick up and process this url call, in views.py:

```
def page(request, page=None):
    p = Collection(request, 'origin')
    #Grab all the page breaks
    pbn, page, pb = getAllPages(p, page)
    print 'pages found %s' % (len(pbn))
    if pbn == False or page == None:
        return render_to_response('origin/error.html', {'message': "Can't find page %s" % (str(p
    else:
        page_content = p.process_element(pb, 'origin/base.py', True, None)
        #Work out the next and previous pages
        thispb = pbn.index(page)
        if thispb==0:
            previouspb = None
        else:
            previouspb = pbn[thispb-1]
        if thispb+1 == len(pbn):
            nextpb = None
        else:
            nextpb = pbn[thispb+1]
        return render_to_response('origin/text.html', {'page_content': page_content, 'current_p
```

This function calls another function:

```
def getAllPages(p, page):
    #Grab all the page breaks
```

```
    results = p.query("//pb")
    if results == None:
        return (False, False, False)
    pbn = []
    while results.hasNext():
        if results.next().get_attribute_value('id')[0]=='S':
            pbn.append(results.next().get_attribute_value('n'))
    #Grab the pb
    pb = None
    page = str(page)
    results = p.query("//pb[@n='%s']" % page, 1, 1)
    if results.hasNext():
        pb = results.next()
    else:
        page = None
    return (pbn, page, pb)
```

The first function, 'page', receives the number of the page sought. It then calls the function 'getallpages' with this page number. This function gets all the pages by an XQuery-formatted call: 'results = p.query("//pb")' and puts them in a 'results' list. Our example file follows the 'Trojan Horse' treatment devised by Steve DeRose, where each pb element marking the beginning of a page is matched by another pb element marking the end of the page: the start of page 7 is marked with <pb id="S-7-1859" corres="E-7-1859" n="7"/> , and the end with <pb id="E-7-1859" corres="S-7-1859" n="7"/>. The function tests the id attribute of each pb element found: if it begins with 'S' it appends each page break found to a list 'pbn'. The function then searches for the particular page sought, with another XQuery call ('p.query("//pb[@n='%s']" % page, 1, 1)'): if it finds it, the element is put into the 'pb' variable, and the function then returns three values: the list of page-break 'n' values, the 'n' value of the page sought and the <pb> element corresponding to the page sought.

The 'page' function then resumes, and takes up these three values returned by 'getallpages'. It prints out the number of pages found (72). It then calls the process_element function with the <pb> element found. Note that the third parameter is set to 'True': this tells SDPublisher to process not just this element, but every following element found. Thus, in your browser you should see the whole text beginning with the page sought. If you set the third parameter to 'False' you will see that the text disappears, as only the content of the <pb> is shown: and as it has no content, nothing is shown.

But, this is not doing what we want. It is showing every page to the end of the book, not just the page we want. We want to stop at the end <pb> element, corresponding to the start <pb> element for this page. We do this by changing what happens when we meet a <pb> element, in the 'base.py' file called by process_element. First, we add to the PIXELISE_PATTERNS structure at the beginning of the file the line

```
'pb': 'pb',
```

so that it appears:

```
PIXELISE_PATTERNS = {
    'div': 'div',
    'p': 'p',
    'head': 'head',
    'pb': 'pb',
```

```
}
```

Now, we add a 'pb' function to the 'base.py' file, as follows:

```
def pb(element, state, context):
    id = element.get_attribute_value('id')
    if id[0]=='E':
        return {'stop_processing':True}
```

As SDPublisher traverses the document from the starting page requested, this element is called every time a <pb> element is met. The function then tests the value of the 'id' attribute. If it begins with an 'E' then it is the end <pb> element corresponding to the start <pb> element we began with, and the function then returns with the value 'stop_processing' set to 'True' (this 'stop_processing' variable is the equivalent of the Anastasia 'finish' variable). This stops SDPublisher at that point. Thus you will see the text of only this page in the browser , just as we want.

1.8 Your first SDPublisher publication: generating links; template files

The 'page' function in 'views.py' does a few more things, after it puts the content of the page into the 'page_content' variable. First, it finds out if there are pages preceding and following this page: if there are, their page numbers are put into the 'previouspb' and 'nextpb' variables. Then, rather opaquely, all this information (the 'page_content', the 'page', 'previouspb' and 'nextpb' variables) all appears in this line, which creates the text we see in our browser:

```
return render_to_response('origin/text.html', {'page_content': page_content, 'current_page':
```

This is a Django function, designed to send content in response to a browser request ('render_to_response'). The first parameter gives the name of a template file, 'origin/text.html', while the following parameters state variables to be used in this template file. You will find the template file 'text.html' in the subfolder 'origin' in the folder 'templates' in the 'origin' folder:

```
{% extends "origin/base.html" %}
{% block content %}
<div class="nav">
    {% if previouspb %}
        <a href="{% url origin.views.page previouspb %}" target="_parent">Previous {{previou
    {% endif %}
    {{ current_page }}
    {% if nextpb %}
        <a href="{% url origin.views.page nextpb %}" target="_parent">Next {{nextpb}}</a>
    {% endif %}
</div>
<div id="page">
    {{ page_content|safe }} </div>
{% endblock %}
```

What is happening here? The first line tells us that this file 'extends' another file, 'base.html', which we will look at in a moment. The next line says: all that follows is the content of a 'block' variable, named 'content'. The next lines state what is to go into that variable: first, some html ( ) and then a Django construct '{% if previouspb %}'. We recall that the number of the previous page was assigned to a variable 'previouspb' in the Django

'render_to_response' call. If there was a page before this (that is, previouspb is not 'None') then the following line comes into play:

```
<a href="{% url origin.views.page previouspb %}" target="_parent">Previous {{previouspb}}
```

This Django call constructs the url for the previous page. First, the '% url' indicates that we are making a url. Then, the function corresponding to the url, origin.views.page, is given. Django looks up the urls declared in the urls.py file and finds that this function corresponds to the url declared in the line

```
(r'^page/(?P<page>[^/]+)$', 'origin.views.page')
```

It then places the value of the previouspb variable in the url, so creating the url 'http://127.0.0.1:8000/SDPublisher/origin/page/7' when the value of 'previouspb' is '7'. Similarly, another url is created for the link to the next page. Then, all the HTML contained in the 'page_content' variable is written into a element, and the page is ready for display in the browser.

We are almost finished this demonstration of SDPublisher. There is one more piece of the puzzle we have not yet explained. The template file 'text.html' contains the line '% extends "origin/base.html"'. Open the file 'base.html' in the subfolder 'origin' of the 'templates' folder:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Strict//EN" "http://www.w3.org/TR/xhtml1/l
<html>
    <head>
        <title>Darwin's Origin</title>
        <link type="text/css" rel="stylesheet" href="origin/www/origin.css" />
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    </head>
    <body>
    {% block content %}{% endblock %}
    </body>
</html>
```

Here we see the familiar frame of an HTML page: the opening <html> statement, followed by the <head> element, and further on, within the <body> element, the Django {% block content %} statement, which embeds the text returned to the 'text.html' template file. We also see, in the <head> element, a link to a css stylesheet, 'origin.css', contained in a folder 'origin/origin /www'. At present this document is not being found. To find this document we need to add a line to the urls.py file (in the 'origin' folder), to handle the call to the 'origin.css' file:

```
(r'www/(?P<path>.*)$', 'django.views.static.serve', {'document_root': 'www'}),
```

Thus, the whole urlpatterns variable now looks like:

```
urlpatterns = patterns('',
    (r'text/', 'origin.views.text'),
    (r'chapter/(?P<chapter>[^/]+)$', 'origin.views.chapter'),
    (r'page/(?P<page>[^/]+)$', 'origin.views.page'),
    (r'www/(?P<path>.*)$', 'django.views.static.serve', {'document_root': 'www'}),
)
```

This added line allows us to feed requests for files (html, css, image files, etc) into SDPublisher, and tells SDPublisher where to look for them: in this case, in a 'www' folder within the 'origin' folder. Thus, the call to the css file 'origin/www/origin.css' will look for the file in that 'origin' folder.

You will see that we have a 'www' folder within the 'origin' folder, containing the file 'origin.css'. This file contains the line:

```
body {background-color: yellow;}
```

Now, when you reload the text, the browser will now find this file, and use this command to set the background colour to a rather horrible yellow. You can use the same technique to display image files: a call to 'http://127.0.0.1:8000/origin/www/fig1.gif' would display the image file 'fig1.gif' in the 'www' folder.

1.9 Starting your own project

So far, we have been working with the specimen XML file 'origin.xml' and the files provided with it. You may now be ready to start out with your own XML file. Presuming you have your own xml file in TEI format, 'mybook.xml', this is how you can get started fast on making an SDPublisher book from this file:

- Start the new SDPublisher book by typing 'python manage.py startxml mybook' at the prompt. You should see the folder 'mybook' now appear in the 'newedition' folder
- Tell SDPublisher about this new book (as outlined in 1.4 above) by editing the 'settings.py' file (that is: adding 'SDPublisher.mybook' to the INSTALLED_APPS variable)
- Copy your 'mybook.xml' file into the new 'mybook' folder. Then, input it into the database by typing 'python manage.py putxml mybook -f mybook/mybook.xml -n text' at the prompt
- Edit the urls.py file in the SDPublisher folder so that the urlpatterns variable contains the line '(r'^mybook/', include('mybook.urls')),'. Create a new urls.py file (or copy one over) in the 'mybook' folder, and edit it so that the urlpatterns variable contains the line '(r'text/', 'mybook.views.text'),)
- Edit the 'views.py' file in the 'mybook' folder, so that it contains these lines:

```
from pixelise.core import Collection
from django.shortcuts import render_to_response

def text(request):
    p = Collection(request, 'mybook')
    results = p.query("//text")
    if results.hasNext():
        text = results.next()
    else:
        return render_to_response('mybook/error.html', {'message': "Can't find text element"})
    text_content = p.process_element(text, 'mybook/base.py', False, None)
    return render_to_response('mybook/text.html', {'page_content': text_content})
```

- Make a new subfolder 'mybook' inside the 'mybook/pixelates' folder, and move the 'base.py' file inside the 'pixelates' folder into that subfolder.
- Make a new folder 'templates' in the 'mybook' folder, and a new subfolder 'mybook' within it. Copy over the files 'text.html', 'error.html' and 'base.html' from the 'origin/templates/origin' folder into that directory. You will need to open 'templates/mybook/text.html' and alter the line '{% extends "origin/base.html" %}' to {% extends "mybook/base.html" %}'

Once you have done this: stop and restart the server by typing 'Control-C' followed by 'python manage.py runserver' into the terminal. Now, if you type 'http://127.0.0.1:8000/mybook/text/' into your browser, you should see all the text of your XML. Now, you can start refining your book using the methods outlined in this document.

We have now reached the end of this 'Getting started' documentation. The following sections provide reference documentation for SDPublisher, and notes on some useful utility functions and tools to work with SDPublisher.

2 SDPublisher Reference Documentation

2.1 The Pixelise processor

The core processing module of SDPublisher is Pixelise. Typically, SDPublisher works as follows:

- The server recognizes a URL (e.g. 'SDPintro/..') as requiring handling by the SDPublisher system, and passes the whole URL to SDPublisher. In a production system, SDPublisher will be running within a Python server module.
- SDPublisher matches the URL to those listed in the 'urls.py' files in the main folder and in the subfolder for each book, and identifies the SDPublisher book, module and function which should handle the call. Thus, the call 'SDPublisher/origin/text/' is mapped to 'origin.views.text': that is, to the book 'origin', to the module 'views', instantiated as a file 'views.py', to be handled by the function 'text' in that file.
- Up to this point, we have been using Django to receive the URL and work out what should be done with it. Now, we invoke Pixelise. First, we work out what XML body of data corresponds to the SDPublisher book. We do this by importing the pixelise 'Collection' object, using the line 'from pixelise.core import Collection'. From this Collection object, you gain access to the XML you want through the call 'p = Collection(request, 'mybook')'. This finds the XML database for 'mybook' in the 'Collection' object and places it in the variable 'p'. Then we work out just what XML element in that book corresponds to the URL call and we have Pixelise retrieve that element, from the XML database for this book, contained in the 'p' variable.
- Now we have retrieved the element: we send the element for processing by the core Pixelise function, process_element (see below). Essentially, this allows us to transform the XML into HTML (or anything we like), in a very efficient fashion.
- After Pixelise does its work, we return to Django, and use the sophisticated Django template language to 'plug' the Pixelise output into a webpage.

Pixelise (the core processing module of SDPublisher) has a few specialized constructs (keywords, variables, etc), of which you need to be aware. You have met most of them in the 'Getting started' section. The two most important single functions in Pixelise are 'Collection' and 'process_element'. 'Collection' is the entire single body of XML data known to the publisher system: that is, all the XML databases which it has access to, and from which you wish to extract all the information you want to supply the reader. In the model above, SDPublisher works out which Pixelise Collection corresponds to the book sought. It then uses an XQuery function, 'query', to retrieve the element sought, and then passes the element to the process_element function.

process_element is called with four parameters, thus process_element(text, "base.py", True, None):

text	The first parameter must be a valid XML element mapped to a Python object. See below for details of the properties of the object.

base.py	The second parameter must be the name of a Python script file contained in the 'Pixelates' folder. This file must contain a PIXELISE_PATTERNS statement, mapping elements to functions.
True	This third parameter is what distinguishes Pixelise, and SDPublisher, from most XML processors. When set to True, this parameter tells Pixelise: process this element AND carry on past the end of element, either to the end of the document or until a stop_processing (see below) is met. When set to False (the default) the processor stops at the end of the element, and returns.
None	This is a Django response object, and Pixelise will pass this parameter into every element method function in the 'pixelates' files. This can be very useful in some contexts: you can, for example, have one function store information by writing it to the response object and another function can then extract that information, effectively as a global variable. See the Django documentation on this.

Note that process_element can be used recursively. That is: you could call (say) a <ref> element, and discover that you would like to embed the contents of a <note> element at that point. You could call that process_element for that <note> and append what is returned to the output stream.

We have discussed a further Pixelise core function, process_element_range. This would take at least one further parameter: the XML element at which processing should stop, thus avoiding the use of the stop_processing construct. For example, one could call this function with the start and end <pb> elements for a particular page and so show only the text on that page. Additional parameters for character offsets from the XML elements might also be passed in: so one could start 16 characters after (or before?) a particular element, and process the XML span up to 30 characters after (or before?) a particular element. You can see why we have not yet done it!

Every time the Pixelise processor meets an element, it looks up that element in the PIXELISE_PATTERNS list at the beginning of the .py file called to process that element, as the second parameter of the process_element call. Note that one can qualify elements by their paths in the PIXELISE_PATTERNS list, and have the same function called for any number of element and element combinations. In this example, head elements one or two divs deep in the document will both be processed by a function 'head1'; all other head elements will be processed by a function 'generic_head'

```
PIXELISE_PATTERNS = {
    'body/div/head': 'head1',
    'body/div/div/head': 'head1',
    'head': 'generic_head',
}
```

If the processor finds the element name specified in the list, it calls the function in the list specified for that element three times as it processes the element, with three parameters. The first parameter is the XML element itself that you want to process, mapped to a Python object: you can use the object methods in the next sections to get information about the XML element. The second parameter is set as indicated below.

begin	When given as the value of the second parameter in an element call,

	indicates that the Pixelise processor is at the beginning of the element
content	When given as the value of the second parameter in an element call, indicates that the processor is about to process the content of the element
end	When given as the value of the second parameter in an element call, indicates that the processor is at the end of the element

The function call can return with three possible values: text, to be written to the output stream; 'hide_content' and 'stop_processing':

text	Text returned by the element call is appended by the processor to the output stream. Thus: 'return html' adds whatever text is contained in the 'html' variable to the output stream.
hide_content	If this Pixelise variable is set to 'True' when an element call (with the second parameter set to 'content') returns, the content of this element is hidden -- that is, the processor skips past all element content to the end of the element. Thus: if the element function returns with the statement "return {'hide_content':True}" the content is hidden.
stop_processing	If this Pixelise variable is set to 'True' when an element call (with the second parameter set to 'end') returns, the Pixelise processor stops and returns control to the function which called process_element. Thus: if the element function returns with the statement "return {'stop_processing':True}" the processing stops.

There is one other Pixelise variable which you might meet: PIXELISE_OUTPUT_LIMIT. This is a safety net: if the number of elements processed exceeds the number set in PIXELISE_OUTPUT_LIMIT, Pixelise will stop. You can set PIXELISE_OUTPUT_LIMIT in the settings.py file. It is a good idea to set this to something real: it is quite possible to send Pixelise into an endless loop (e.g. by using process_element to call an ancestor, which then calls the starting element, which calls the ancestor, and so on forever) and this will guard against that.

2.2 Element methods: getting elements

Within SDPublisher, every XML element is mapped to a Python object, with a rich set of methods. For a notional element, there are many methods available for every Python object in SDPublisher, corresponding to the element (see 3.3 for information on how to list all the methods available for any element). We here describe the Pixelise methods created by us to allow retrieval of XML elements or of information about them. All of these take the form 'xxx_xxx', with underscores: get_attribute_value for example. In a DBXML implementation, there are many other methods for each element: 'getNextSibling' for example. We provide Pixelise wrappers for those 'native' DB XML functions which we have found useful: thus, instead of calling getNextSibling you should call get_next_sibling. This is intended to make it possible for SDPublisher to be implemented with other database systems. We recommend that you use only the Pixelise functions described here. If you do this, you will find it very easy to use your SDPublisher files with any database which provides an implementation of SDPublisher. Formal documentation of the SDPublisher API is given at demo.pixelise.org/api/.

Many of these methods are self-explanatory, and called without parameters. For example:

```
firstchild = div.get_first_child()
```

retrieves the first XML element which is a child of this particular div element, and assigns it to the variable firstchild.

We give details of all Pixelise methods. In this section, we list methods which for, any given element, retrieve related elements: siblings, parents, children, etc.. Note that throughout, if the method cannot find the element sought, it will return a Pixelise exception error. This allows the calling function to deal gracefully with the failure to find the element, by placing the call in a 'try...except' block. This guards against the situation, all too familiar from Anastasia, where the program cannot find an element but carries on as if it had, usually with dire consequences.

get_parent_element	Has no parameters: retrieves the element which is the immediate parent. If there is no such element, an exception is thrown. Example: div.get_parent_element() returns the immediate parent of the element.
get_ancestor_by_name	Has one parameter, the name of the element sought; retrieves that ancestor, regardless of how far above in the document tree that ancestor is. If there is no such ancestor, an exception is thrown. Example: div.get_ancestor_by_name('body') retrieves the <body> element which is the nearest ancestor of the element.
get_first_child	Has no parameters: retrieves the element which is the first child. If there is no such element, an exception is thrown. Example: div.get_first_child() returns the first child of the element.
get_last_child	Has no parameters: retrieves the element which is the last child. If there is no such element, an exception is thrown. Example: div.get_last_child() returns the last child of the element.
get_child	Has one optional parameter, the name of the element sought. If the name of the element sought is given, retrieves the child which is an immediate child (not a grandchild, or more distant descendant) of the parent element and which matches the name sought. If there is no such child, an exception is thrown. If the name of the element sought is not given, then retrieves the first child found. Example: div.get_child('p') retrieves the <p> element which is the first child of the element; div.get_child() retrieves the first child of the element.

get_child_by_name	[Deprecated in 1.1: use get_child with the name of the element sought as a parameter]
get_descendant_by_name	Has one parameter, the name of the element sought; retrieves the child which is a descendant (child, grandchild, or more distant descendant) of the parent element. If there is no such child, an exception is thrown. Example: div.get_descendant_by_name('p') retrieves the first <p> element which is a descendant of the element
get_children	Has one optional parameter, the name of the elements sought. If the name of the elements sought is given, returns a Python list of the immediate children of the parent element which match the name sought. If the name of the element sought is not given, then retrieves a Python list of all the immediate children of the parent element. If there is no such child, an exception is thrown. Example: allps=div.get_children('p') retrieves a list of the <p> elements which are children of the element and puts the list in the variable 'allps'. You can iterate through this list using the Python 'for' call: 'for p in allps:'.
get_children_by_name	[Deprecated in 1.1: use get_children with the name of the element sought as a parameter]
get_next_element	Has one optional parameter: the name of the element sought. If the name of the element sought is given, retrieves the next element which matches the name sought. If there is no such element, an exception is thrown. If the name of the element sought is not given, retrieves the next element, of any kind (including text elements) in the document stream. If the next element is a child, it retrieves the first child; if it is a right sibling, it retrieves that; else it looks up the element ancestors and through their children and siblings till it finds the next element. If there is no next element (we are at the end of the document), an exception is thrown. Example: div.get_next_element() retrieves the next element from the beginning (not the end) of the element
get_next_element_by_name	[Deprecated in 1.1: use get_next_element with the name of the element sought as a parameter]
get_previous_element	Has one optional parameter: the name of the element sought. If the name of the element sought is given, retrieves the previous

	element which matches the name sought. If there is no such element, an exception is thrown. If the name of the element sought is not given, retrieves the previous element, of any kind (including text elements) in the document stream. If the previous element is the last child of the left sibling, it retrieves that last child; if it is a left sibling, it retrieves that; else it looks up the element ancestors and through their children and siblings till it finds the previous element. If there is no previous element (we are at the beginning of the document), an exception is thrown. Example: div.get_previous_element() retrieves the last element before the beginning (not the end) of the element
get_previous_element_by_name	[Deprecated in 1.1: use get_previous_element with the name of the element sought as a parameter]
get_next_node	Works as for get_next_element, but ignores '#text' nodes (that is, nodes containing only character data). It has one parameter, set to either 'True' or 'False': if 'True' it ignores the children of the starting node, iterating only through siblings and ancestors. If there is no next node, an exception is thrown. Example: div.get_next_node('True') finds the next node from the beginning (not the end) of the element which is not a child of the element or a '#text' node.
get_previous_node	Works as for get_next_node, but locates the previous node in the document stream, not the next node. If there is no previous node, an exception is thrown.
has_right_sibling	Has no parameters; returns 'True' if the element has a right sibling of any kind; 'False' if it does not. Example: div.has_right_sibling() returns 'True' if the element has a right sibling of any kind; 'False' if it does not.
has_left_sibling	Has no parameters; returns 'True' if the element has a left sibling of any kind; 'False' if it does not. Example: div.has_left_sibling() returns 'True' if the element has a left sibling of any kind; 'False' if it does not.
get_next_sibling	Has one optional parameter, the name of the element sought. If the name of the element sought is given, retrieves the element with this name which is the next (right) sibling. If there is no such element, an exception is thrown. If the name of the element sought is

	not given, retrieves the next sibling. Example: p.get_next_sibling('l') returns the <l> element if the <p> element has a right sibling which is a <l> element; throws an exception if it has not.
get_next_sibling_by_name	[Deprecated in 1.1: use get_previous_element with the name of the element sought as a parameter]
get_previous_sibling	Has one optional parameter, the name of the element sought. If the name of the element sought is given, retrieves the element with this name which is the previous (left) sibling. If there is no such element, an exception is thrown. If the name of the element sought is not given, retrieves the left sibling. Example: p.get_previous_sibling('l') returns the <l> element if the <p> element has a left sibling which is a <l> element; throws an exception if it has not.
get_previous_sibling_by_name	[Deprecated in 1.1: use get_previous_element with the name of the element sought as a parameter]

Note that character data nodes (that is, nodes containing only text, not elements) have the name '#text'.

2.3 Element properties: getting information about elements

In this section, we list Pixelise methods for retrieving information about particular elements:

get_attribute_names	Has no parameters: retrieves a list of the names of attributes for the element. If there are no attributes, the method returns 'None'. Example: div.get_attribute_names() returns a list of the attributes for the element.
get_attribute_value	Has one parameter, the name of the attribute whose value is sought. If the element has an attribute of that name, the method returns the attribute value as a string. If there is no attribute with this name, an exception is thrown. Example: for the element div.get_attribute_value('type') returns the string 'book'.
create_path	Has no parameters; retrieves the element path from the top of the document to the element as a Python list. Example: within this document, the path to this element <cell> element would be returned as TEI.2 text body div div div table row cell.
print_element_debug	Has no parameters; retrieves the name of the element and its attributes. Example: for the element div.print_element_debug() returns the string

	' '.
get_node_name	Has no parameters: what this method returns depends on the type of object calling this method. If the object is an XML element, the method returns the name of the element. Examples: for the element div.get_node_name() returns the string 'div'; for the attribute 'type="book"' attribute.get_node_name() returns the string 'type'.
get_node_value	Has no parameters: what this method returns depends on the type of object calling this method. If the object is an XML attribute, the method returns the value of the attribute. Example: for the attribute 'type="book"' attribute.get_node_value() returns the string 'book'. (It's hard to see how this could work with an element instead of an attribute!)
isspace	Has no parameters: returns 'True' if the element contents are just white space (spaces, tabs); 'False' if they are not. Example: div.isspace() returns 'True' if the element contains only white space; 'False' if it does not.

2.4 Query functions

SDPublisher provides support for most (if not all) XQuery searches via the method 'query', available on every XML collection within Pixelise. You retrieve the XML collection you are interested in by a call to the Pixelise Collection object: "p=Collection(request, 'origin')" retrieves the XML for the book 'origin'. In the DB XML implementation, this is the contents of a single .dbxml database ('container' in DB XML parlance). The typical process in SDPublisher is as follows (see 2.1 above):

- A SDPublisher function (eg 'chapter') is called, with a parameter pointing to the document section to be processed (e.g. '1' for chapter 1)
- SDPublisher identifies and initializes the Pixelise collection appropriate to this document, thus: 'p=Collection(request, 'origin')'
- For this 'p' object, representing the entire XML document, Pixelise then retrieves the XML element sought by 'p.query'

The 'query' method is called with three parameters, thus p.query("//pb", 1, 50):

//pb	The first parameter must be the query itself, formatted as an XQuery string. This searches for all <pb> elements
1	The second parameter specifies the first result we want returned. If this is '1' then the first 'hit' is the first result returned; if it is '50', the 50th hit is the first result returned
50	The third parameter specifies the last result we want returned. If this is '1' then the first 'hit' is the last result returned; if it is '50', the 50th hit is the last result returned

Thus: using p.query with only one parameter will return all matches of the query string: hence, p.query("//pb") will return all <pb> elements. If you wish to limit the hits returned, you must specify both the second and third parameters. p.query("//pb", None, 1, 50) will return the first

50 hits; p.query("//pb", None, 51, 100) the next 50, etc.

Here are some sample XQuery commands, embedded in p.query:

p.query("//text")	Retrieves all <text> elements
p.query("//pb[@n='7]", 1, 1)	Retrieves the first <pb n="7"/> element
p.query("//p[contains(., "deviation")]')	Retrieves all <p> elements containing the word "deviation".
p.query("//feed /author="peterrr73"")	Retrieves all <feed> elements containing an <author> element which contains the text 'peterrr73'.

Note that the results of a query method are always returned as a Python list. If nothing is found, the query returns 'None'; if something is found, you can extract it by using the Python 'next()' method, and iterate through the results using the Python 'hasNext method()' (or just use the Python 'for' method, thus:

```
results = p.query("//pb")
    if results == None:
        return (False)
    pbs =[]
    while results.hasNext():
        pbs.append(results.next())
    return (pbs)
```

This fragment runs a search for all page breaks. If it finds any, it initializes a Python list (pbs) and appends each page break to the list.

2.5 Template and Pixelate file locations

In the 'Origin of Species' example given above, we place the Pixelate file 'base.py' inside a subfolder 'origin', inside the 'pixelates' folder. We then call that file by requesting 'origin/base.py' in the process_element call. Similarly, we have the templates files 'text.html' and others in a subfolder 'origin' within the 'templates' folder, and we request 'origin/text.html' in the render_to_response call.

We could have done without the 'origin' subfolders, placed the 'base.py' and 'text.html' files directly within the 'templates' and 'pixelates' folders, and called them simply by invoking 'base.py' and 'text.html'. Why do we not do that, and instead use the apparently unnecessary 'origin' subfolders?

We do this because of how Django finds pixelate and template files. Django knows to look for these in 'pixelates' and 'templates' folders. Thus, when it is told to find 'base.py' it looks in every pixelate folder it knows about: that is, in the pixelate folders for both 'SDPintro', 'origin', and every other book you have. As soon as it finds a 'base.py' file in any of these, it stops looking and uses that one. Thus, when we call 'base.py' while processing the book' origin' it could easily bring back the 'base.py' file for 'SDPintro' (or any other book at all). Django does exactly the same for templates: the request for 'text.html' while processing 'origin' could bring back the 'text.html' file for 'SDPintro', not 'origin'.

You can stop Django returning the 'wrong' pixelate or template file by specifying a path for the file, thus 'origin/base.py' will find only the 'base.py' file in a folder 'origin' in a 'pixelates' folder.

That is how we do it in these examples.

Although this seems annoying, you can make good use of how Django does this. You could have a single 'base.py' or 'text.html' file which you want used by all your books. Just place this file in a 'pixelates' or 'templates' folder within one of your books, invoke it as plain 'base.py' or 'text.html' without a path, and every book which invokes plain 'base.py' or 'text.html' without a path will find it.

3 Utility functions and tools

3.1 SDPublisher management functions: startxml and others

SDPublisher provides a few useful functions, for setting up and managing publications. These are all run from the command line, inside the folder containing the books you want to publish. The next commands presume you have typed 'cd SDPublisher' from the command line to move into the SDPublisher folder.

startxml	Example: python manage.py startxml mybook This creates a folder 'mybook' within the Django project folder, containing various starting folders and files which SDPublisher will need (for example: a skeleton 'views.py' file; a 'pixelates' folder, and an empty DB XML container 'mybook.dbxml' etc.) Note that after running startxml the user has to edit the settings.py file in the parent folder, by editing the INSTALLED_APPS list and adding various lines; see 1.4 above.
indexxml	Example: python manage.py indexxml mybook -f mybook/mybook.xml This creates an index of the xml in 'mybook.xml', in the folder 'mybook'. For larger documents, this can speed up various operations immensely. Counterintuitively, in DB XML it is faster to create the index before inputting the XML into the database, using the next command.
putxml	Example: python manage.py putxml mybook -f mybook/mybook.xml -n text This inserts the xml in 'mybook.xml', in the folder 'mybook', as the document 'text' into the DB XML container 'mybook.dbxml'. If there is already a document 'text' in that DB XML container, an error message is generated (see rmxml below) This makes use of the document feature in DB XML -- not supported, I expect, in all XML or other databases
rmxml	Example: python manage.py rmxml mybook -n text This removes the document 'text' from 'mybook'. You can then run 'putxml' to reinput the edited document.
putallxml	Example: python manage.py putallxml mybook -d mybook/mybookxml This locates the folder 'mybookxml' in 'mybook', finds all the files with the extension '.xml' in that folder, and adds all of them to book 'mybook'. You can specify all files with a different extension by adding an '-e' flag to the call: '-e tei' will add all files with the extension '.tei'.

There is scope for adding many more command line management functions -- and for refining the ones we have. But, beyond the most basic level, this might be excessively dependent on the individual database we happen to be using. Perhaps we should not go down that route.

3.2 DB XML command line functions

It can be useful to manipulate the DB XML database directly, rather than through the Python interface. To do so, you need to move into the same folder as the dbxml file for your book: thus, if you are in the folder 'SDPublisher' holding the folder 'mybook' you would need to type 'cd mybook' to move into the 'mybook' folder holding 'mybook.dbxml'.

Before executing any of the following commands, you need to start the dbxml interpreter from the command line, after making sure that you are in the folder holding the dbxml file. You can start the interpreter just by typing

```
dbxml
```

Followed by the return key at the command prompt. You may have to give the full path to the dbxml application: thus (for example):

```
"C:\Program Files\Oracle\Berkeley DB XML 2.4.16\bin\dbxml"
```

You should then get the command prompt 'dbxml>'. The following commands are then available:

openContainer	example: openContainer origin.dbxml Opens the specified .dbxml container file. All subsequent operations are on this file until it is closed. All the following examples presume that the container origin.dbxml has been opened by an openContainer call.
putDocument	example: putDocument text origin.xml f Inserts the document 'origin.xml' into the container under the name 'text'.
removeDocument	example: removeDocument text Removes the document with the name 'text' from the container.
query	example: query 'collection("origin.dbxml")//ab[@id="CH1-100-100-1859"]' Finds the <ab> element with id="CH1-100-100-1859" in origin.dbxml (the first sentence of the 1859 'Origin').
print	example: print Prints the last object found. Following the last call, the <ab> element and its content will be printed.
removeNodes	example: removeNodes 'collection("origin.dbxml")//div[@n="CH1"]//ab' Removes all <ab> elements in the element.
sync	example: sync Writes the .dbxml container to disc, so saving all changes made.
exit	example: exit Quits DB XML.

Like all industrial-strength databases, DB XML has a powerful indexing capacity. Once your

source XML grows beyond a certain size, you will find that DB XML indices dramatically increases performance: on a 28 megabyte XML file we have found search performance improving by a factor of 30,000 to one. Here are some indexing commands, to be executed from the command line after the .dbxml container has been opened:

addIndex "" target node-attribute-equality-string	Indexes all attributes with the name 'target'.
addIndex "" id node-attribute-equality-string	Indexes all attributes with the name 'id'.
addIndex "" w node-element-presence	Indexes all <w> elements.
addIndex "" p node-element-presence	Indexes all <p> elements.

We have found that it is not only queries which are sped up by indexing: all DB XML operations appear faster. There is an overhead, as indices might add 25% or more to the database size, which might itself be around three times larger than the original XML. But the speed advantage is worth the extra storage needed.

3.3 Running Pixelise from the shell

An advantage of the Python scripting environment is that one can open a Python 'shell' and access the full range of SDPublisher functions from the shell. This is a useful way of checking what is happening for any given book inside SDPublisher. Here is an example of a Python shell session within SDPublisher:

```
C:\Documents and Settings\All Users\editions>python manage.py shell
      Python 2.5.4 (r254:67916, Dec 23 2008, 15:10:54)
      [MSC v.1310 32 bit (Intel)] on win32
      Type "help", "copyright", "credits" or "license" for more information.
      (InteractiveConsole)
   >>> from pixelise.core import Collection
   >>> p = Collection('origin')
   >>> results = p.query("//ab[@id='CH2-100-100-1859']")
   >>> firstab = results.next()
   >>> print firstab
<ab n="100" id="CH2-100-100-1859">BEFORE applying the principles ... </ab>
```

The key here is the first command: 'python manage.py shell'. This starts the shell. The next command imports the Collection method from Pixelise, and the line 'p = Collection('origin')' initializes the dbxml database for the book 'origin' and attaches it to the variable 'p'. Thereafter, all happens exactly as it would in a .py script: we retrieve the first sentence of the second chapter and print it to the window.

There are many other useful tools you can run from the shell. For example, you can see a list of all the methods available for a given object by typing the object name and then pressing the tab key twice (this may require IPython installation).