Dexy News

January 2011

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

DEXY is a new open source software project for building and automating documents. Dexy will help you present your code, data and graphs elegantly, easily and correctly in any format, even a blog. This newspaper was created using Dexy, and in it we discuss how Dexy was designed and what it can do.

Dexy for Code

IF you write code, even just the occasional R or Matlab script, then you also need to write about code, and Dexy can help. Dexy makes it easy to do things like keep a personal code journal, write blog posts including code samples, and incorporate code into journal articles, books or any other documents you need to write. For example, you can include syntax highlighted code like this (it's much prettier in colour but this newspaper is black and white):

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

and you can see the results of running this code, like this:

```
hello, world
```

Dexy easily supports multiple languages in a single document, for example here is some R:

```
> x <- 5
> rep(1, 3)
[1] 1 1 1
> x^2
[1] 25
>
```

With Dexy, you never type dead code into your document, instead Dexy reads your source code and transforms it according to filters that you specify. These filters do things like run your code through an interpreter, apply syntax highlighting, or both. Filters can do just about anything, and you can chain filters together.

This approach makes it fast and natural to write documents which incorporate code. You write code in code files using your text editor or IDE of choice, and write text in your document in any* format you want, then use Dexy to bring them together. This way your code is testable, runnable and can be included in many different documents (such as your paper, your poster and your presentation slides).

*In theory, Dexy should be able to work with any text-based format (such as plain text, LaTeX, or HTML), any binary format which can be produced from a text-based markup, or any binary format which can be accessed via a scriptable interface. If you don't see your preferred format, then please ask us via the forum http://discuss.dexy.it or twitter @dexyit and we'll do our best to support it. Obviously, it's easier to support open source formats since we can install these and test them out, but we'll do our best to help with proprietary formats.

Dexy for Science

DEXY was designed to be a powerful tool for document automation, especially scientific document automation. A fully automated document is also a reproducible document (if you share your sources), so Dexy is also a tool for reproducible research.

Dexy helps incorporate the results of data analysis into your document in such a way that their provenance can be traced back, and the results can be automatically updated if new data becomes available. Graphs, movies, or audio files can also be automatically generated and included in your document. You can also describe your methodology by incorporating your scripts directly into your document.

Because Dexy makes it easy to re-use material in multiple documents, it's easy to keep your lab notebook, working paper, journal article, poster and presentation slides in sync and up to date. It's also very easy to blog about your research as you go, or after you've published your paper. Work you've done in one place can be easily re-used elsewhere, so by the time you come to publish your paper you have refined your scripts by writing about them in several different documents.

Blogging with Dexy

DEXY was not specifically written with blogging in mind, but it was written to be a very flexible framework. So, it was a delight but not a huge surprise to find that Dexy is a great way to blog. For those who have tried it, blogging about code can be a painful experience. Copying and pasting code while preserving the indentation, trying to find a good solution for syntax highlighting, typing untested code directly in to your blog's editor. All of these are frustrating, slow and prone to error.

With Dexy's approach, you can keep code in its own files so you can test and run it. You have access to any syntax highlighting library you want. You can upload a draft to your blog so you can preview it, revise it and refresh it as needed, then publish whenever you are ready. Most of the articles for the Dexy blog (any that involve code examples) are published in this way.

Dexy does this by talking to your blog's API (Application Programming Interface), the "back door" to your blog. The WordPress API has been most extensively developed, this works both with self-hosted WordPress blogs and with WordPress.com blogs. Some support is available for Tumblr and Posterous (although because of limitations with these systems their Dexy integration isn't quite as good at this time). Support for Blogger is coming soon, the GData API is extensive so this should eventually have full support for Dexy.

see http://blog.dexy.it/241

Try Dexy

Dexy is open source software licensed under the generous MIT license. While Dexy does work and works very well, it is still an early project so you might encounter bugs. Also, the interface might change if we develop better ways to do things. I would encourage you to try Dexy out, and I would also encourage you to use a good distributed version control system with remote backups, to proofread all documents carefully, and to include assertions and other forms of validation in your scripts.

To learn more about Dexy, to see more documentation and to learn how to install Dexy or obtain the source code, visit http://dexy.it. To see more examples of Dexy in action and to keep up to date with the project, follow the Dexy blog at http://blog.dexy.it

IF your work doesn't call for a tool like Dexy, we would really appreciate if you passed this newspaper on to someone who might find it useful. Good departments to try are computational biology, astronomy or computer science.

This newspaper was written by Ana Nelson, the creator of Dexy, who would really like your feedback on the format, the content or on Dexy itself. Find her at ScienceOnline2011 (come along to the Dexy demo if you can make it), email her ana@ananelson.com or say hi on twitter @ananelson.

If you want to create a newspaper like this one to promote your software project or as a handout for your next presentation or poster session, then check out http://www.newspaperclub.co.uk. This newspaper was typeset in LaTeX using the sciposter document class (you can see the preamble if you flip this paper over to the back).

Newspaper Club

Newspaper Club is a service that helps people and communities make their own newspapers.

Editorial: Writing About Code

The ultimate goal of Dexy is to let you focus on writing, rather than stressing about the technicalities of pulling a complex document together. The act of writing is valuable in itself as well as for the end result.

I believe that writing about code is good for your code, as well as for yourself. Explaining your code in words, seeing this on the page next to the code itself, is a superb form of feedback. It will be obvious to you when your code needs to be reworked, and your explanation in the familiar realm of human language might suggest ways in which to do that. The sensation of seeing your code and your words beautifully presented on a printed page is also a welcome reward in what can sometimes feel like a very abstract and intangible realm of programming.

If you are able to explain your code in words in this way, then this document will prove invaluable to you over time when you have forgotten the details of a particular implementation. I say "over time" and this can mean "over lunch" or "over 5 minutes" as well as "over 2 years". A personal code journal is a great discipline which is easy to do with a tool like Dexy, and which will save you lots of "now why did I do it this way?".

You should write for yourself, and then you should write for others. With Dexy, you write what you need to write without restriction as to the format or layout. With many tools, you are shoehorned into a particular type of document. This type of document might not be what you want to produce, or the presence of a default format may mean that you don't stop to think about what type of document would best suit your audience.

Built-in API documentation such as JavaDoc can be useful, but it also can be very limiting. And, putting documentation comments in your source code may be reasonable, but it means you can only 1 opportunity to explain something, despite the fact that you may want to write different messages for different audiences about the same code.

With Dexy, you have the freedom to write in any format you wish, so you can make a conscious decision as to what you want to write. Documentation, either of your code or your methodology, is a vital form of communication which can help others to make use of (and cite) your work. In the movement for better scientific communication, and better scientific use of programming, Dexy is a valuable ally.

GarlicSim Example

DEXY was inspired by the particular challenges of writing up simulation-based research. When your data is generated by your code, and will be regenerated many times while you work due to refinements and modifications of your code, an automated approach becomes practically a necessity. The need to work with multiple languages simultaneously (say, Java for your simulation code and R for your analysis), to show code in various formats (raw, with syntax highlighting, as a transcript, just what's written to STDOUT when run), to track provenance of data files and their corresponding simulation parameters, to be able to maintain synchronization and share code between multiple documents describing the same research: this combination of requirements was not met by any existing tool, and once resolved in Dexy results in a tool which meets each individual need simply and effectively.
As a demonstration of Dexy's various capacities, it makes sense to return to Dexy's origins and write about a simulation and its resulting data. We will use the GarlicSim library, a simulation framework written in Python, and its built-in Prisoner's Dilemma game.

Documenting Source Code

In this section we will look at the implementation of the Prisoner's Dilemma simpack in GarlicSim. As far as Dexy features, we are simply taking a source code file, splitting it into sections to make it easy to discuss individual elements, and applying syntax highlighting for clarity.

The State class defines the environment for our simulation:

```
ROUNDS = 7
NUMBER_OF_PLAYERS = 70

class State(garlicsim.data_structures.State):

    def __init__(self, round, match, player_pool):
        self.round = round
        self.match = match
        self.player_pool = player_pool
```

This simulation acts on a population of agents (called "players" in this case), which are assigned a random strategy when they are initialized:

```
    @staticmethod
    def create_messy_root():
        global player_types
        state = State(
            round=-1,
            match=0,
            player_pool=[
                random_strategy_player() for \
                i in xrange(NUMBER_OF_PLAYERS)
            ]
        )
        state.prepare_for_new_match()
        return state
```

The game is organized into rounds, each round lasting for a set number of steps (as defined by the constand ROUNDS). At the beginning of each round, the agents are paired off with a partner they will play against for that round. Also, the agent with the lowest score is removed from the population and replaced with a new agent having a randomly chosen strategy.

```
    def prepare_for_new_match(self):
        '''
        Note: this function is not strictly a "step function":
        it manipulates the state that is given to it and then returns it.
        '''
        pool = self.player_pool
        loser = player_with_least_points(pool)
        pool.remove(loser)
        pool.append(random_strategy_player())

        self.pairs = pair_pool(self.player_pool)
```

So, the distribution of strategies present in the population will change over time based on each strategy's performance within the given population.
Here is the code which pairs agents off at the start of each round:

```
def pair_pool(player_pool):
    '''
    Takes a player pool and returns a list of random pairs of players.
    Every player will be a member of exactly one pair.
    '''
    assert len(player_pool) % 2 == 0
    result = []
    pool = player_pool[:]
    while len(pool) > 0:
        pair = random.sample(pool, 2)
        result.append(pair)
        pool.remove(pair[0])
        pool.remove(pair[1])
    return result
```

Each pair then plays the Prisoner's Dilemma game against each other at each step in the round (so this is an Iterated Prisoner's Dilemma). In the Prisoner's Dilemma, each prisoner can either stay silent or can implicate the other player in a crime. (for more info see http://en.wikipedia.org/wiki/Prisoner's_dilemma) In this case a return value of True to each player's play() method indicate silence, False indicates confession.

```
def play_game((x, y), round):
    x_move = x.play(round)
    y_move = y.play(round)

    assert x_move in [True, False]
    assert y_move in [True, False]
    if x_move == True and y_move == True:
        x.points += 1
        y.points += 1
    elif x_move == True and y_move == False:
        x.points += -4
        y.points += 2
    elif x_move == False and y_move == True:
        x.points += 2
        y.points += -4
    elif x_move == False and y_move == False:
        x.points += -1
        y.points += -1

    x.other_guy_played(y_move)
    y.other_guy_played(x_move)
```

Each player decides what to do depending on their strategy, which was chosen randomly at initialization. The "angel" strategy never implicates the other party:

```
class Angel(Player):
    def play(self, round):
        return True
```

While the "devil" strategy always does so:

```
class Devil(Player):
    def play(self, round):
        return False
```

The "tit-for-tat" strategy starts out by not confessing, and in subsequent turns it does whatever the other player did in their previous turn. Thus it rewards the other player for acting like an angel, and punishes the other player for acting like a devil.

```
class TitForTat(Player):
    def play(self, round):
        if round == 0:
            return True
        else:
            return self.last_play

    def other_guy_played(self, move):
        self.last_play = move
```

The simulation is actually run by repeatedly calling the step() method:

```
    def step(self):

        state = copy.deepcopy(self, StepCopy())
        state.clock += 1

        state.round += 1
        if state.round == ROUNDS:
            state.round = -1
            state.match += 1
            state.prepare_for_new_match()
            return state

        for pair in state.pairs:
            play_game(pair, state.round)

        return state
```

Running a Simulation

Now that we have explored the library source code, let's look at how we would actually run a simulation. We are going to write a Python script to automate running the simulation, and we start with importing the garlicsim library and the simpack containing the Prisoner's Dilemma example:

```
import garlicsim
from garlicsim_lib.simpacks import prisoner
```

Then we import some other libraries which we need

```
import csv
import json
```

We define some constants:

```
NUMBER_OF_PLAYERS = 70 # hard coded into simpack
NUMBER_OF_STEPS = 1000
```

Now we need a way to collect the data generated by the simulation. We will just store the info in a text file, so we set this up next:

```
csv_filename = "{{ a.create_input_file('sim_output', 'csv', False) }}"
csv_file = open(csv_filename, "w")
data_writer = csv.writer(csv_file)
```

You might notice that we don't actually specify a filename here, at least not anything that looks like a normal filename. We are going to run this script through a filter before we execute it, and this filter will insert a random filename for the data file. This provides a provenance for the data and helps to ensure that data doesn't get corrupted by using the same file name and creating ambiguities over whether the data in that named file corresponds to a particular run.
Now we write the column names to the file:

```
data_writer.writerow(["step", "agent", "points", "strategy"])
```

And now we're ready to initialize the simulation:

```
state = prisoner.State.create_messy_root()
i = -1
```

Let's define a method to collect simulation data. We want to get information about each agent in each time period:

```
def collect_data():
    for j in range(NUMBER_OF_PLAYERS):
        agent = state.player_pool[j]
        strategy = agent.__class__.__name__
        data_writer.writerow([i, j, agent.points, strategy])
```

And we are going to call this straight away to collect the initial state of the system:

```
collect_data()
```

Now let's actally run the simulation:

```
for i in range(NUMBER_OF_STEPS):
    state = garlicsim.simulate(state)
    collect_data()
```

We're finished writing to the CSV file so we can close this now:

```
csv_file.close()
```

Finally, while we have collected the simulation state in each time step in our text file, we also want to capture the simulation parameters since we may want to refer to these later. We use JSON as a format to store these data, and again we do our random filename trick:

```
json_filename = "{{ a.create_input_file('sim-params', 'json', False) }}"
json_file = open(json_filename, "w")

json.dump({
    'number_of_players' : NUMBER_OF_PLAYERS,
    'number_of_steps' : NUMBER_OF_STEPS
}, json_file)

json_file.close()
```

That's it! Here is what the whole script looks like when it is run:

```
Python 2.7 (r27:82508, Jul  3 2010, 21:12:11)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ### @export "imports-garlicsim"
... import garlicsim
>>> from garlicsim_lib.simpacks import prisoner
>>>
>>> ### @export "imports-other"
... import csv
>>> import json
>>>
>>> ### @export "constants"
... NUMBER_OF_PLAYERS = 70 # hard coded into simpack
>>> NUMBER_OF_STEPS = 1000
>>>
>>> ### @export "setup-csv"
... csv_filename = "dce017de-9a56-4ba0-947c-22ce31f46b44.csv"
>>> csv_file = open(csv_filename, "w")
>>> data_writer = csv.writer(csv_file)
>>>
>>> ### @export "header-row"
... data_writer.writerow(["step", "agent", "points", "strategy"])
>>>
>>> ### @export "init"
... state = prisoner.State.create_messy_root()
>>> i = -1
>>>
>>> ### @export "collect-data"
... def collect_data():
...     for j in range(NUMBER_OF_PLAYERS):
...         agent = state.player_pool[j]
...         strategy = agent.__class__.__name__
...         data_writer.writerow([i, j, agent.points, strategy])
...
>>> ### @export "initial-collect"
... collect_data()
>>>
>>> ### @export "step"
... for i in range(NUMBER_OF_STEPS):
...     state = garlicsim.simulate(state)
...     collect_data()
...
>>> ### @export "cleanup"
... csv_file.close()
>>>
>>> ### @export "json"
... json_filename = "f77531fe-8ab5-4486-934f-ddce26f70459.json"
>>> json_file = open(json_filename, "w")
>>>
>>> json.dump({
...     'number_of_players' : NUMBER_OF_PLAYERS,
...     'number_of_steps' : NUMBER_OF_STEPS
... }, json_file)
>>>
>>> json_file.close()
```

And here is what the start of the CSV data looks like:

```
step,agent,points,strategy
-1,0,0,TitForTat
-1,1,0,TitForTat
-1,2,0,Angel
-1,3,0,Angel
-1,4,0,Angel
-1,5,0,Devil
-1,6,0,Devil
-1,7,0,Devil
-1,8,0,TitFo
```

Analyzing The Data

Now that we've run our simulation, we want to do something with the data we've collected. We are going to use R to prepare some graphs and do some calculations. We read in the CSV and JSON data generated by the simulation, and we store some additional calculations in a new JSON file. Here is the script as it looks when run, after random filenames have been substituted:

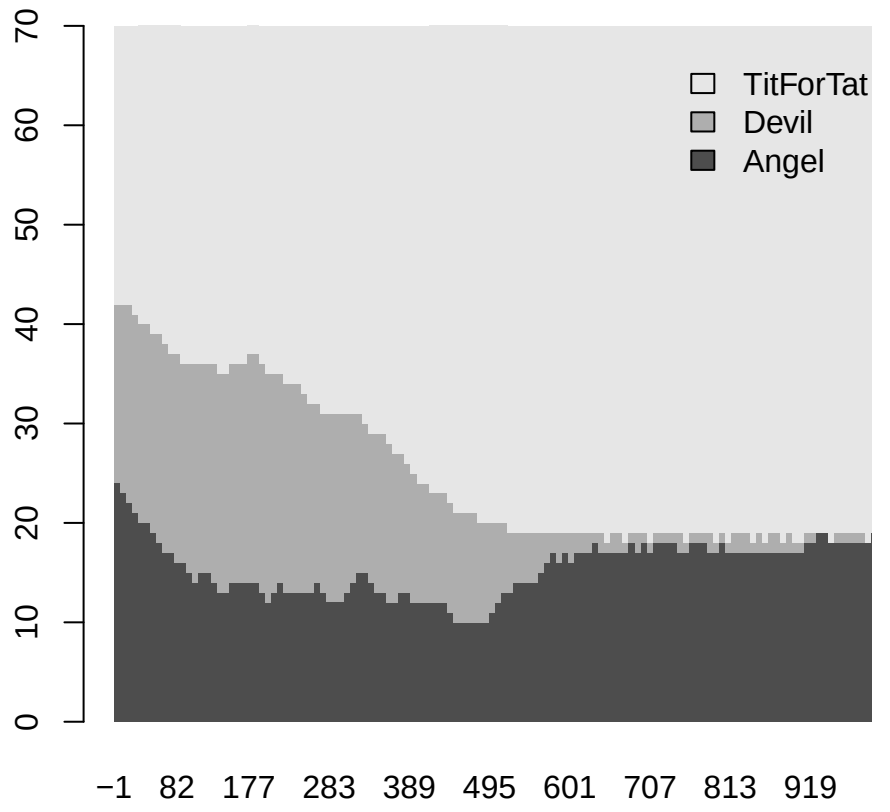
```
> library(rjson)
>
> data = read.csv("dce017de-9a56-4ba0-947c-22ce31f46b44.csv", sep=",")
>
> strategy.counts <- table(list(strategy=data$strategy, step=data$step))
>
> pdf(file="9bc87662-f575-4e4d-8e32-aa64f6ddb5d3.pdf", width=5.5, height=5.5)
> barplot(
+     strategy.counts,
+     legend.text = TRUE,
+     border=NA,
+     args.legend = pairlist(bty='n')
+ )
> dev.off()
null device
          1
>
> last.period = subset(data, data$step==max(data$step))
>
> pdf(file="8529c8c8-5993-4a09-8c72-ba4acf3a7325.pdf", width=5.5, height=5.5)
> hist(last.period$points, main="", ylab="Number of Agents", xlab="Final Score")
> dev.off()
null device
          1
>
> json.filename = "670e9dbe-b335-4f2a-94e2-81033be3bad1.json"
> sim.results <- list( max_points_in_last_period = max(last.period$points))
>
> # Show the contents...
> sim.results
$max_points_in_last_period
[1] 654

>
> write(toJSON(sim.results), json.filename)
>
```

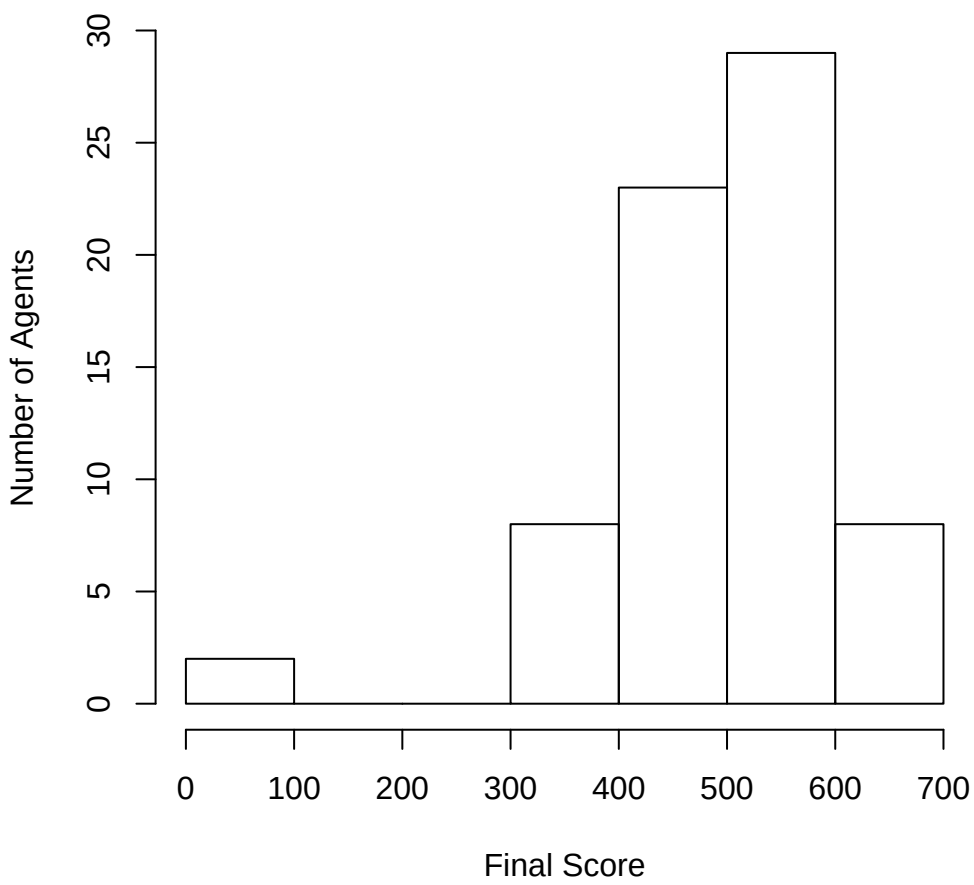
So in our documents, we can now make statements like:

> The simulation was run for 1000 steps with 70 players. The maximum number of points in the last period was 654.

With none of those numbers being manually typed in. So when running the simulation again, if the numbers change, the document is updated automatically. Likewise, graphs such as this graph of the number of agents following each strategy over time are also automatically refreshed as necessary:



This histogram shows the distribution of points in the final period of the simulation (note that as low-scoring agents are removed and replaced these new agents start with a score of 0):

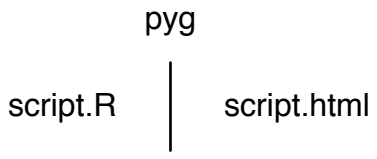


How Dexy Works

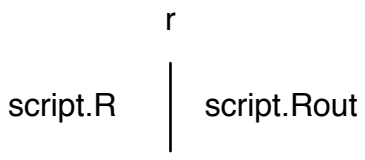
Dexy is built on a few simple concepts which are very powerful when combined. This section talks about these concepts and shows how they are implemented in Dexy. You don't need to read this section, but if you're curious it will help you understand what's happening behind the scenes.

Filters

The most noticeable element in Dexy are the filters. Filters transform the text that gets passed through them. Filters are written in Python and they can do anything you want them to, including interact with remote APIs or call command line processes. (Note that you can easily write a filter in a language other than Python and this can be called as a command line process.)
For example, a filter may take source code and transform it by applying syntax highlighting:



Or a filter may take source code and transform it by running it through an interpreter and returning the transcript:



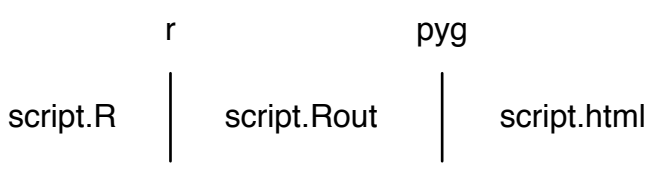
Here is an R script:

```
x <- 5
rep(1, 3)
x^2
```

And here is this same R script after passing through the 'r' filter:

```
> x <- 5
> rep(1, 3)
[1] 1 1 1
> x^2
[1] 25
>
```

You can run text through multiple filters chained together, so that each filter runs on the output of the previous filter:



Here is the output of the R script after passing through the 'r' filter and then the 'pyg' filter for (rather subtle in this case) syntax highlighting:

```
> x <- 5
> rep(1, 3)
[1] 1 1 1
> x^2
[1] 25
>
```

It is significant that we start with a file called 'script.R'. In Dexy, you write your code in standalone files and use the filtering system to pull this code into your document after it has been transformed in some way (there is a dummy filter called 'dexy' which pulls in the original unmodified text if you don't want it transformed in any way). There are many advantages to this approach of having source code live in its own separate files, as opposed to the approach taken by tools such as noweb where you have code blocks typed in delimited sections within a document. You can re-use the same source code in several places within a single document, and in various different formats within a single document, all of which will be kept in sync (because they all come from the same source file). You can also re-use code examples in multiple different documents. You can run the file outside of Dexy too, since it's just a normal code file with the expected file extension. In fact, you can use Dexy to document files in situ, especially with Dexy's remote URL feature which lets you fetch remote files, say from a version control system, and transform them with filters. And because you are working with normal code files you can continue to use your favourite text editor or IDE without having to change modes for Dexy.

Filter Examples

Filters are responsible for most of Dexy's functionality, they are also the main way in which you can customize Dexy, and they have been designed to be easy to write. For example, here is a filter which just returns the first ten lines of the text it is passed:

```
class HeadHandler(DexyHandler):
    ALIASES = ['head']
    def process_text(self, input_text):
        return "\n".join(input_text.split("\n")[0:10]) + "\n"
```

That's the whole thing! For simple filters like this one, you only need to define a process_text method which returns the transformed text. The ALIASES constant defines how you can refer to this filter to tell Dexy that you want to apply it. So for this filter, you would write "$long.txt|head$" to indicate that you wanted to invoke the 'head' filter on the contents of the file long.txt.
As mentioned earlier, splitting documents into sections is an important capability of Dexy since it allows you to write, say, tutorials with the code sample presented in bite-size chunks which can be easily and clearly explained. This sectioning is implemented by the Idiopidae filter, which implements a process_text_to_dict() method:

```
class IdioHandler(DexyHandler):
    INPUT_EXTENSIONS = [".*"]
    OUTPUT_EXTENSIONS = [".html", ".tex", ".txt"]
    ALIASES = ['idio', 'idiopidae']

    def process_text_to_dict(self, input_text):
        composer = Composer()
        builder = idiopidae.parser.parse('Document', input_text + "\n\0")

        name = "input_text%s" % self.ext
        if self.ext == '.pycon':
            lexer = PythonConsoleLexer()
        else:
            lexer = get_lexer_for_filename(name)

        fn = self.artifact.filename()
        formatter = get_formatter_for_filename(fn, linenos=False)

        output_dict = OrderedDict()

        for i, s in enumerate(builder.sections):
            lines = builder.statements[i]['lines']
            formatted_lines = composer.format(lines, lexer, formatter)
            output_dict[s] = formatted_lines

        return output_dict
```

If you wish to split your file up into sections using some other system then you just need to implement a similar filter which implements process_text_to_dict().
Some filters take care to preserve this sectioning. For example here is a Python filter which runs each section of code through the Python interpreter, while preserving the names of each section. Importantly this all happens within a single Python session so that variable names etc. are preserved between sections.

```
class ProcessSectionwiseInteractiveHandler(DexyHandler):
    EXECUTABLE = '/usr/bin/env R --quiet --vanilla'
    VERSION = "/usr/bin/env R --version"
    PROMPT = '>'
    COMMENT = '#'
    TRAILING_PROMPT = "\r\n> "
    INPUT_EXTENSIONS = ['.txt', '.r', '.R']
    OUTPUT_EXTENSIONS = ['.Rout']
    ALIASES = ['rint']

    def process_dict(self, input_dict):
        output_dict = OrderedDict()

        proc = pexpect.spawn(self.EXECUTABLE, cwd=self.artifact.artifacts_dir)
        proc.expect(self.PROMPT)
        start = (proc.before + proc.after)

        for k, s in input_dict.items():
            section_transcript = start
            start = ""
            proc.send(s)
            proc.sendline(self.COMMENT * 5)
            if self.artifact.doc.args.has_key('timeout'):
                timeout = self.artifact.doc.args['timeout']
            else:
                timeout = None

            proc.expect(self.COMMENT * 5, timeout = timeout)

            section_transcript += proc.before.rstrip(self.TRAILING_PROMPT)
            output_dict[k] = section_transcript

        return output_dict
```

The process_dict() method expects you to accept a dict and return another dict.
The DexyHandler class defines a default process() method which looks to see if you have defined a process_text(), process_dict(), or process_text_to_dict() method in your subclass:

```
    def process(self):
        """This is the method that does the "work" of the handler, that is
        filtering the input and producing output. This method can be overridden
        in a subclass, or one of the convenience methods named below can be
        implemented and will be delegated to. If more than 1 convenience method
        is implemented then an exception will be raised."""
        method_used = None

        if hasattr(self, "process_text"):
            if method_used:
                raise Exception("%s has already been called" % method_used)
            if len(self.artifact.input_data_dict.keys()) > 1:
                raise Exception("""You have passed input with multiple sections
                                to the %s handler. This handler does not preserve
                                sections. Either remove sectioning or add a call
                                to the join filter before this handler.""")
            input_text = self.artifact.input_text()
            output_text = self.process_text(input_text)
            self.artifact.data_dict['1'] = output_text
            method_used = "process_text"

        if hasattr(self, "process_dict"):
            if method_used:
                raise Exception("%s has already been called" % method_used)
            input_dict = self.artifact.input_data_dict
            output_dict = self.process_dict(input_dict)
            self.artifact.data_dict = output_dict
            method_used = "process_dict"

        if hasattr(self, "process_text_to_dict"):
            if method_used:
                raise Exception("%s has already been called" % method_used)
            input_text = self.artifact.input_text()
            output_dict = self.process_text_to_dict(input_text)
            self.artifact.data_dict = output_dict
            method_used = "process_text_to_dict"
```

```
        if not method_used:
            # This code implements the neutral 'dexy' handler.
            self.artifact.data_dict = self.artifact.input_data_dict
            method_used = "process"

        return method_used
```

If you define more than one of these, Dexy raises an error. Dexy also raises an error if you try to pipe content which is in sections through a process_text() method as this won't preserve the sections.
So, you can create a filter by subclassing DexyHandler and implementing one of the three special method names, or if you need more control you can override the process() method itself, as in the dot handler:

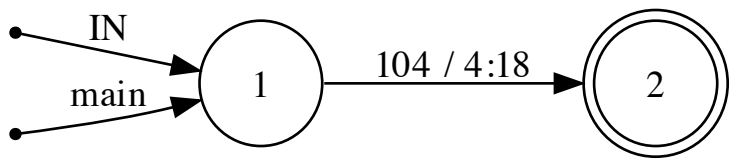
```
class DotHandler(DexyHandler):
    INPUT_EXTENSIONS = [".dot"]
    OUTPUT_EXTENSIONS = [".png", ".pdf"]
    ALIASES = ['dot', 'graphviz']

    def process(self):
        self.artifact.auto_write_artifact = False
        self.artifact.generate_workfile()
        wf = self.artifact.work_filename(False)
        af = self.artifact.filename(False)
        ex = self.artifact.ext.replace(".", "")
        command = "/usr/bin/env dot -T%s -o%s %s" % (ex, af, wf)
        self.log.info(command)
        ad = self.artifact.artifacts_dir
        self.artifact.stdout = pexpect.run(command, cwd=ad)
```

The dot handler takes dot code like this:

```
digraph hello {
        rankdir=LR;
        node [ shape = point ];
        ENTRY;
        en_1;
        node [ shape = circle, height = 0.2 ];
        node [ fixedsize = true, height = 0.65, shape = doublecircle ];
        2;
        node [ shape = circle ];
        1 -> 2 [ label = "104 / 4:18" ];
        ENTRY -> 1 [ label = "IN" ];
        en_1 -> 1 [ label = "main" ];
}
```

And renders it using graphviz:



The 'latex' filter which was used to render this newspaper into PDF is another example of a filter which overrides the process() method:

```
class LatexHandler(DexyHandler):
    INPUT_EXTENSIONS = [".tex", ".txt"]
    OUTPUT_EXTENSIONS = [".pdf", ".png"]
    ALIASES = ['latex']

    def generate(self):
        self.artifact.write_dj()

    def process(self):
        latex_filename = self.artifact.filename().replace(".pdf", ".tex")
        latex_basename = os.path.basename(latex_filename)

        f = open(latex_filename, "w")
        f.write(self.artifact.input_text())
        f.close()

        # Detect which LaTeX compiler we have...
        latex_bin = None
        for e in ["pdflatex", "latex"]:
            which_cmd = "/usr/bin/env which %s" % e
            latex_bin, s = pexpect.run(which_cmd, withexitstatus = True)
            if s == 0:
                self.log.info("%s LaTeX command found" % e)
                break
            else:
                self.log.info("%s LaTeX command not found" % e)
                latex_bin = None

        if not latex_bin:
            raise Exception("no executable found for latex")

        command = "/usr/bin/env %s %s" % (e, latex_basename)
        self.log.info(command)
        # run LaTeX twice so TOCs, section number references etc. are correct
        ad = self.artifact.artifacts_dir
        self.artifact.stdout = pexpect.run(command, cwd=ad, timeout=20)
        self.artifact.stdout += pexpect.run(command, cwd=ad, timeout=20)
```

Templating

The most important filter right now is probably the Jinja filter. Jinja is a templating system which allows you to place special tags in your document to indicate that dynamic content should go there. This is how you can write a document and pull in the output of running your source code files through various filters. Because the support for Jinja has been implemented as a filter, Dexy will be able to support other templating systems very easily (Python-based systems most easily), although for now Jinja is the only supported system.
Much of the work of the Jinja filter is in making available the output of the various filters that have been run. Actually running Jinja itself is only a few lines of code. The documentation at http://jinja.pocoo.org is the best source of reference for what you can do with Jinja, bearing in mind that for LaTeX documents Dexy automatically changes the jinja tag style from curly brackets to angle brackets so as to avoid clashing with LaTeX's use of curly brackets.
Here is the source of the Jinja handler:

```
from dexy.handler import DexyHandler

from jinja2 import Environment
import os
import pexpect
import re
import simplejson as json

class JinjaHelper:
    def ri(self, query):
        # --system flag needed or else ri complains about multiple versions
        command = "ri --system -T -f simple %s" % query
        return pexpect.run(command)

    def read_file(self, filename):
        f = open(filename, "r")
        return f.read()

class JinjaHandler(DexyHandler):
    INPUT_EXTENSIONS = [".*"]
    OUTPUT_EXTENSIONS = [".*"]
    ALIASES = ['jinja']

    def process_text(self, input_text):
        document_data = {}
        document_data['filenames'] = {}
        document_data['sections'] = {}
        document_data['a'] = {}

        # TODO move to separate 'index' handler for websites
        # create a list of subdirectories of this directory
        doc_dir = os.path.dirname(self.artifact.doc.name)
        children = [f for f in os.listdir(doc_dir) \
                    if os.path.isdir(os.path.join(doc_dir, f))]
        document_data['children'] = sorted(children)

        self.artifact.load_input_artifacts()
        for k, a in self.artifact.input_artifacts_dict.items():
            common_prefix = os.path.commonprefix([self.artifact.doc.name, k])
            common_path = os.path.dirname(common_prefix)
            relpath = os.path.relpath(k, common_path)

            if document_data['filenames'].has_key(relpath):
                raise Exception("Duplicate key %s" % relpath)

            document_data['filenames'][relpath] = a['fn']
            document_data['sections'][relpath] = a['data_dict']
            document_data[relpath] = a['data']
            for ak, av in a['additional_inputs'].items():
                document_data['a'][ak] = av
                fullpath_av = os.path.join('artifacts', av)
                if av.endswith('.json') and os.path.exists(fullpath_av):
                    print "loading JSON for %s" % fullpath_av
                    document_data[ak] = json.load(open(fullpath_av, "r"))

        if self.artifact.ext == ".tex":
            print "changing jinja tags for", self.artifact.key
            env = Environment(
                block_start_string = '<%',
                block_end_string = '%>',
                variable_start_string = '<<',
                variable_end_string = '>>',
                comment_start_string = '<#',
                comment_end_string = '#>'
                )
        else:
            env = Environment()
        template = env.from_string(input_text)

        # TODO test that we are in textile or other format where this makes sense
        if re.search("latex", self.artifact.doc.key()):
            is_latex = True
        else:
            is_latex = False

        document_data['filename'] = document_data['filenames']
        template_hash = {
            'd' : document_data,
            'filenames' : document_data['filenames'],
            'dk' : sorted(document_data.keys()),
            'a' : self.artifact,
            'h' : JinjaHelper(),
            'is_latex' : is_latex
        }

        try:
            result = str(template.render(template_hash))
        except Exception as e:
            print "error occurred while processing", self.artifact.key
            raise e

        return result
```

Dependencies, Topological Sort and Smart Caching

What makes Dexy interesting and useful is that a document A (such as $article.html|jinja$) can depend on another document B (such as $script.R|pyg$). So, we need a way to tell Dexy that A depends on B. Then Dexy needs to figure out that B needs to be processed first, and Dexy needs to have a way of making the output of B available to A.
We tell Dexy which items to process and what the dependencies are using a specification file. Then Dexy implements a topological sort to determine an order for the documents so that all dependencies are processed first. Dexy then processes

the documents in order, and stores the output in a special directory using carefully chosen filenames. The filenames are chosen so that if you run Dexy again and nothing has changed in the source, Dexy will use its stored version of the file rather than running the filters gain. This saves a lot of time, especially if you have a large document which depends on dozens of other files and only 1 of them has changed.
Here is the specification file for this newspaper.

```
{
  "hello.R|dexy" : {},
  "hello.R|pyg|pyg|l" : {},
  "hello.R|r|pyg|l" : {},
  "hello.R|r" : {},
  "dexy.sh|pyg|l" : {},
  "dexy.sh|bash" : {},
  "prisoner.R|jinja|r|pyg|l" : {
    "inputs" : ["garlicsim_prisoner.py|jinja|pycon|pyg|l"]
  },

  "*.py|idio|l" : {},

  "newspaper.tex|wrap|pyg|l" : {},
  "newspaper.tex|jinja|latex" : { "allinputs" : true},

   ".dexy|dexy" : {},

  "*.dot|dexy" : {},
  "*.dot|dot|p" : {},

  "@artifact.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/artifact.py"
  },
  "@document.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/document.py"
  },
  "@controller.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/controller.py"
  },
  "@handler.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/handler.py"
  },
  "@interface.py|pyg|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/interface.py"
  },
  "@topological_sort.py|pyg|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/dexy/topological_sort.py"
  },
  "@pyg_handler.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/handlers/pyg_handler.py"
  },
  "@python_handlers.py|idio|l" : {
  "url":"https://bitbucket.org/ananelson/dexy/raw/tip/handlers/python_handlers.py"
  },
  "@subprocess_handler.py|idio|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/handlers/subprocess.py"
  },
  "@jinja_handler.py|pyg|l" : {
    "url":"https://bitbucket.org/ananelson/dexy/raw/tip/handlers/jinja_handler.py"
  }
}
```

This is a rather complex specification, but you can see that some files are listed by name with specific other files mentioned as inputs. Other directives are wildcard directives which apply to all files with a given extension. Names starting with @ are 'virtual' files, there isn't a file called artifact.py in this directory, but Dexy will fetch the contents at the specified URL and pretend this is a local file called artifact.py, which is then put through the 'pyg' filter for syntax highlighting. The 'l' filter doesn't do anything itself, but it forces the output to have a .tex file extension, and this tells 'pyg' to use LaTeX formatting codes rather than the default of HTML. Many filters change their behaviour depending on the file extension accepted by the next filter in the chain.
Dexy's Controller class processes the config, builds up a list of all files needing to be processed and determines the ordering to use:

```
    def process_config(self):
        def parse_doc(input_directive, args = {}):
            # If a specification is nested in a dependency, then input_directive
            # may be a dict. If so, split it into parts before continuing.
            try:
                a, b = input_directive.popitem()
                input_directive = a
                args = b
            except AttributeError:
                pass

            tokens = input_directive.split("|")
            glob_string = os.path.join(self.path, tokens[0])
            filters = tokens[1:]

            docs = []

            # virtual document
            if re.search("@", glob_string):
                virtual = True
                if not self.allow_remote:
                    raise Exception("""You are attempting to access a remote file.
                                    You must enable --dangerous mode to do this.
                                    Please check Dexy help and call the dexy
                                    command again.""")
                glob_string = glob_string.replace("@", "")
            else:
                virtual = False

            regex = fnmatch.translate(glob_string).replace(".*", "(.*)")
            matcher = re.compile(regex)

            files = glob.glob(glob_string)

            if len(files) == 0 and virtual:
                files = [glob_string]

            for f in files:
                create = True

                inputs = []
                if args.has_key('inputs'):
                    if isinstance(args['inputs'], str):
                        raise Exception("""this input should be an array,
                                        not a string: %s""" % args['inputs'])
                    for i in args['inputs']:
                        for doc in parse_doc(i):
                            inputs.append(doc.key())
                m = matcher.match(f)
                if m and len(m.groups()) > 0:
                    rootname = matcher.match(f).group(1)

                # The 'ifinput' directive says that if an input exists matching
                # the specified pattern, we should create this document and it
                # will depend on the specified input.
                if args.has_key('ifinput'):
                    log.debug(f)
                    if isinstance(args['ifinput'], str):
                        ifinputs = [args['ifinput']]
                    else:
                        ifinputs = args['ifinput']

                    for s in ifinputs:
                        log.debug("evaluating ifinput %s" % s)
                        ifinput = s.replace("%", rootname)
                        log.debug("evaluating ifinput %s" % ifinput)
                        input_docs = parse_doc(ifinput, {})
                        for input_doc in input_docs:
                            log.debug(input_doc.key())
                            inputs.append(input_doc.key())

                    if len(input_docs) == 0:
                        create = False

                if args.has_key('ifnoinput'):
                    ifinput = args['ifnoinput'].replace("%", rootname)
                    input_docs = parse_doc(ifinput, {})

                    if len(input_docs) > 0:
                        create = False

                if args.has_key('except'):
                    if re.search(args['except'], f):
                        create = False

                if create:
                    # Filters can either be included in the name...
                    doc = Document(f, filters)
                    doc.args = args
                    # ...or they may be listed explicitly.
                    if args.has_key('filters'):
                        doc.filters += args['filters']

                    # Here we are assuming that if we get a key with blank args
                    # this should not override a previous key. A key which does
                    # have args should override any previous key.
                    key = doc.key()
                    if len(args) == 0:
                        if self.members.has_key(key):
                            doc = self.members[key]
                        else:
                            self.members[key] = doc
                    else:
                        self.members[key] = doc

                    is_partial = os.path.basename(doc.name).startswith("_")
                    if args.has_key('allinputs') and not is_partial:
                        doc.use_all_inputs = True
                    for i in inputs:
                        doc.add_input_key(i)


                    docs.append(doc)
            return docs

        def get_pos(member):
            key = member.key()
            return self.members.keys().index(key)

        def depend(parent, child):
            self.depends.append((get_pos(child), get_pos(parent)))

        self.members = OrderedDict()
        self.depends = []

        # Create Document objects for all docs.
        for k, v in self.json_dict.items():
            parse_doc(k, v)

        # Determine dependencies
        for doc in self.members.values():
            doc.finalize_inputs(self.members)
            for input_doc in doc.inputs:
                depend(doc, input_doc)

        ordering = topological_sort(range(len(self.members)), self.depends)
        ordered_members = OrderedDict()
        for i in ordering:
            key = self.members.keys()[i]
            ordered_members[key] = self.members[key]
        self.members = ordered_members
```

This is the topological sort algorithm which is used:

```
# Original topological sort code written by Ofer Faigon
# (www.bitformation.com) and used with permission

def topological_sort(items, partial_order):
    """
    Perform topological sort. items is a list of items to be sorted.
    partial_order is a list of pairs. If pair (a,b) is in it, it means that item
    a should appear before item b. Returns a list of the items in one of the
    possible orders, or None if partial_order contains a loop.
    """

    def add_node(graph, node):
        """Add a node to the graph if not already exists."""
        if not graph.has_key(node):
            graph[node] = [0] # 0 = number of arcs coming into this node.
```

```
def add_arc(graph, fromnode, tonode):
    """
    Add an arc to a graph. Can create multiple arcs.
    The end nodes must already exist.
    """
    graph[fromnode].append(tonode)
    graph[tonode][0] = graph[tonode][0] + 1

# step 1 - create a directed graph with an arc a->b for each input
# pair (a,b).
# The graph is represented by a dictionary. The dictionary contains
# a pair item:list for each node in the graph. /item/ is the value
# of the node. /list/'s 1st item is the count of incoming arcs, and
# the rest are the destinations of the outgoing arcs. For example:
# {'a':[0,'b','c'], 'b':[1], 'c':[1]}
# represents the graph: c <-- a --> b
# The graph may contain loops and multiple arcs.
# Note that our representation does not contain reference loops to
# cause GC problems even when the represented graph contains loops,
# because we keep the node names rather than references to the nodes.
graph = {}
for v in items:
    add_node(graph, v)
for a, b in partial_order:
    add_arc(graph, a, b)

# Step 2 - find all roots (nodes with zero incoming arcs).
roots = [node for (node, nodeinfo) in graph.items() if nodeinfo[0]==0]

# step 3 - repeatedly emit a root and remove it from the graph. Removing
# a node may convert some of the node's direct children into roots.
# Whenever that happens, we append the new roots to the list of
# current roots.
sorted = []
while len(roots) != 0:
    # If len(roots) is always 1 when we get here, it means that
    # the input describes a complete ordering and there is only
    # one possible output.
    # When len(roots) > 1, we can choose any root to send to the
    # output; this freedom represents the multiple complete orderings
    # that satisfy the input restrictions. We arbitrarily take one of
    # the roots using pop(). Note that for the algorithm to be efficient,
    # this operation must be done in O(1) time.
    root = roots.pop()
    sorted.append(root)
    for child in graph[root][1:]:
        graph[child][0] = graph[child][0] - 1
        if graph[child][0] == 0:
            roots.append(child)
    del graph[root]

if len(graph.items()) != 0:
    # There is a loop in the input
    return None
return sorted
```

The Document class is responsible for actually processing the specified filters:

```
def run(self, controller):
    self.controller = controller
    self.step = 0

    artifact, artifact_key = self.create_initial_artifact()
    log.info("(step %s) %s -> %s" % \
             (self.step, artifact_key, artifact.filename()))

    for f in self.filters:
        artifact_key += "|%s" % f
        self.step += 1

        if not self.controller.handlers.has_key(f):
            print self.controller.handlers.keys()
            raise Exception("""You requested filter alias '%s'
                            but this is not available.""" % f)
        HandlerClass = self.controller.handlers[f]
        h = HandlerClass.setup(
            self,
            artifact_key,
            artifact,
            self.next_handler_class()
        )

        artifact = h.generate_artifact()
        if not artifact:
            raise Exception("no artifact created!")
        self.artifacts.append(artifact)

        log.info("(step %s) %s -> %s" % \
                 (self.step, artifact_key, artifact.filename()))

    return self
```

With the results of each step being cached by the Artifact class.

```
class Artifact(object):
    """
    This is the Artifact class ... just testing docstrings...
    """
    @classmethod
    def setup(klass, doc, key, handler, previous_artifact = None):
        art = klass()
        art.doc = doc
        art.key = key
        art.data_dict = OrderedDict()
        art.dirty = False
        art.auto_write_artifact = True
        art.additional_inputs = {}
        art.artifacts_dir = art.doc.controller.artifacts_dir
        if previous_artifact:
            art.input_ext = previous_artifact.ext
            art.input_data_dict = previous_artifact.data_dict
            art.input_artifacts = previous_artifact.input_artifacts
            art.additional_inputs = previous_artifact.additional_inputs
            art.previous_artifact_filename = previous_artifact.filename()
        art.dexy_version = VERSION
        if handler:
            art.handler_source = inspect.getsource(handler.__class__)
            art.handler_version = handler.version()
        return art
```

The artifact object calculates a hashstring to identify its contents:

```
    def set_hashstring(self):
        if self.dirty:
            self.dirty_string = time.gmtime()

        hash_dict = self.__dict__.copy()

        # Remove any items which should not be included in hash calculations.
        del hash_dict['doc']

        self.hashstring = hashlib.md5(hash_dict.__str__()).hexdigest()
```

And uses this hashstring for the stored filename:

```
    def filename(self, rel_to_artifacts_dir = True):
        filename = "%s%s" % (self.hashstring, self.ext)
        if rel_to_artifacts_dir:
            filename = os.path.join(self.artifacts_dir, filename)
        return filename
```

Thus we see the interaction of the specification, the topological sort, and Dexy's smart caching.

Dexy's Interface

Dexy is run via a command line tool, a web-based interface to Dexy is also in development. Calling Dexy with the -h flag shows the available command line options:

```
dexy -h
```

Currently these options are as follows:

```
usage: dexy [-h] [-x EXCLUDE_DIR [EXCLUDE_DIR ...]] [-v] [-n] [-u] [-p]
            [-a ARTIFACTS_DIR] [-l LOGS_DIR] [-c CACHE_DIR] [-s] [--setup]
            [-g CONFIG] [-d]
            dir

positional arguments:
  dir                   directory of files to process with dexy

optional arguments:
  -h, --help            show this help message and exit
  -x EXCLUDE_DIR [EXCLUDE_DIR ...], --exclude-dir EXCLUDE_DIR [EXCLUDE_DIR ...]
                        Exclude directories from processing by dexy, only
                        relevant if recursing. The directories designated for
                        artifacts, logs and cache are automatically excluded,
                        as are .bzr, .hg, .git, .svn, ignore.
  -v, --version         show program's version number and exit
  -n, --no-recurse      do not recurse into subdirectories (default: recurse)
  -u, --utf8            switch encoding to UTF-8 (default: don't change
                        encoding)
  -p, --purge           purge the artifacts and cache directories before
                        running dexy
  -a ARTIFACTS_DIR, --artifacts-dir ARTIFACTS_DIR
                        location of artifacts directory (default: artifacts)
  -l LOGS_DIR, --logs-dir LOGS_DIR
                        location of logs directory (default: logs) dexy will
                        create a dexy.log file in this directory
  -c CACHE_DIR, --cache-dir CACHE_DIR
                        location of cache directory (default: cache)
  -s, --short           Use short names in cache
  --setup               Create artifacts and logs directory and generic .dexy
                        file
  -g CONFIG, --config CONFIG
                        name of configuration file
  -d, --dangerous       Allow running remote URLs which may execute dangerous
                        code, use with care.
```

Dexy requires a number of directories to be present when it is run. This is because Dexy uses these directories, but also to make sure that you are in an appropriate location and are aware that Dexy will be generating lots of files in this location. You can call dexy with the –setup flag to create these directories for you, and you can also specify a different name to be used for any of these standard directories in case they should conflict with files already present in your system.

Dexy writes to a log file in logs/dexy.log and it is recommended that you tail -f this file while you are working. If you run into difficulties with your documents, then this log file is the first place you should look for clues. The log file also tells you where the results of processing files have been stored

```
newspaper.tex -> artifacts/d21c030ce79ea164c7cfe78549916dc9.tex
newspaper.tex|jinja -> artifacts/88088ef5e3fe631e7d44c7bcbe5f0b2e.tex
newspaper.tex|jinja|latex -> artifacts/df3a5eaafa2ded1723914268d5e6eafd.pdf
```

The final steps are written to more canonical filenames in the cache/ directory, this is also indicated in the log

```
saving artifacts/df3a5eaafa2ded1723914268d5e6eafd.pdf to
     cache/newspaper/newspaper.tex-jinja-latex.pdf
```

If you want simpler canonical filenames, you can use the -s or –short flag in which case the output would simply be newspaper.pdf, i.e. the original file base name with the final file extension. The website at http://dexy.it is generated in this way.

Source of this Newspaper

This is the beginning of the .tex file containing the source for this newspaper. This shows you the LaTeX preamble and also some Jinja tags for including dynamic Dexy content. Full source code is available from http://bitbucket.org/ananelson/dexy-examples/newspaper

```
\documentclass[custom, plainsections]{sciposter}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Modify paper size and font size in papercustom.cfg in your tex install
%  /usr/local/texlive/2010/texmf-dist/tex/latex/sciposter/papercustom.cfg
%\renewcommand{\papertype}{custom}
%\renewcommand{\fontpointsize}{14pt}
%\setlength{\paperwidth}{31.7cm}
%\setlength{\paperheight}{45.7cm}
%\renewcommand{\setpspagesize}{
%  \ifthenelse{\equal{\orientation}{portrait}}{
%    \special{papersize=31.7cm,45.7cm}
%    }{\special{papersize=45.7cm,31.7cm}
%    }
%  }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\usepackage{cwpuzzle}
\usepackage{fancyvrb}
\usepackage[ascii]{inputenc}
\usepackage[pdftex]{graphicx}
\usepackage[greek, english]{babel}
\usepackage{multicol}
\setlength\columnseprule{0.2pt}

\renewcommand{\rmdefault}{ptm}
\newcommand{\megasize}{\fontsize{100pt}{20pt}\selectfont}
\usepackage{../assets/bw}

\setmargins[1.5cm]

\pagenumbering{arabic}

\title{Dexy News}
\author{Ana Nelson}
\begin{document}
\rmfamily

\begin{megasize}
\includegraphics{../newspaper/logo.pdf}
\hspace{0.7in}
Dexy News
\end{megasize} \hfill January 2011


\begin{multicols*}{3}

\small
<< d['001.c|pyg|l'] >>

\vspace{0.5cm}

\PARstart{D}{exy} is a new open source software project for building and
automating documents. Dexy will help you present your code, data and graphs
elegantly, easily and correctly in any format, even a blog. This newspaper was
created using Dexy, and in it we discuss how Dexy was designed and what it can
do.

\vspace{10pt}
\hrule
\vspace{10pt}

\large
Dexy for Code
\small

\vspace{5pt}

\PARstart{I}{f} you write code, even just the occasional R or Matlab script,
then you also need to write about code, and Dexy can help. Dexy makes it easy
to do things like keep a personal code journal, write blog posts including code
samples, and incorporate code into journal articles, books or any other
documents you need to write. For example, you can include syntax highlighted
code like this (it's much prettier in colour but this newspaper is black and
white):

<< d['001.c|pyg|l'] >>

\noindent and you can see the results of running this code, like this:

\begin{Verbatim}
<< d['001.c|c'] >>
\end{Verbatim}

Dexy easily supports multiple languages in a single document, for example here
is some R:

<< d['hello.R|r|pyg|l'] >>

With Dexy, you never type dead code into your document, instead Dexy reads your
source code and transforms it according to filters that you specify. These
filters do things like run your code through an interpreter, apply syntax
highlighting, or both. Filters can do just about anything, and you can chain
filters together.

This approach makes it fast and natural to write documents which incorporate
code. You write code in code files using your text editor or IDE of choice, and
write text in your document in any* format you want, then use Dexy to bring
them together. This way your code is testable, runnable and can be included in
many different documents (such as your paper, your poster and your presentation
slides).

*In theory, Dexy should be able to work with any text-based format (such as
plain text, \LaTeX, or HTML), any binary format which can be produced from a
text-based markup, or any binary format which can be accessed via a scriptable
interface. If you don't see your preferred format, then please ask us via the
forum http://discuss.dexy.it or twitter @dexyit and we'll do our best to
support it. Obviously, it's easier to support open source formats since we can
install these and test them out, but we'll do our best to help with proprietary
formats.
```