Technologies du Web, Master COMASIC
Lab assignment
Six degrees to philosophy

Antoine Amarilli
a3nm@a3nm.net

December 2, 2014

The aim of this lab assignment is to write a small Web application to play a simple game called "Six degrees to philosophy". The user starts at a Wikipedia article of their choice, is presented with the first few internal links of this Wikipedia article, and chooses one of these links to move to the corresponding article. The goal of the game is to reach the "Philosophy" article following links in this fashion, in the minimal number of steps.

In this assignment, you will use the Python programming language, the Flask web framework and the Jinja template engine. The course page `http://a3nm.net/work/teaching/` contains pointers to relevant documentation.

The lab assignment will last three hours. Afterwards, you will be able to continue working on the assignment at home for some time. Your work should be submitted by email to `a3nm@a3nm.net` (as a ZIP archive containing all source code and relevant files) **before Friday 5, 2014, 23:59 Paris time**; it will be evaluated to determine your grade for this course in the COMASIC master.

Points will be deduced if you submit late; erroneous submissions, corrupt files and forgotten attachments will not entitle you to additional time. Your work should be **strictly personal**; suspicions of plagiarism will result in non-passing grades.

A concise implementation of the whole assignment should not be longer than 500 lines (including code, CSS, templates); if your code is much longer then something is wrong.

The dependencies between the various sections are provided so that you can skip to a different section if you are really stuck.

Now that all of this is out of the way, let's get started!

1 Initial setup

Download the source archive provided on the teaching page and decompress it in your home directory ("Répertoire personnel" on the desktop). Open the `degrees/` folder in a file browser and open the file `getpage.py` in your favorite text editor. Open a terminal, move to the right directory by entering `cd degrees`, and run the `getpage.py` file by entering `./getpage.py`. Check that `It works!` is correctly displayed.

2 Querying the Wikipedia API

The Web application that we will build will display, given the current article the player is on, the list of Wikipedia internal links that can be found on this page. We will obtain this information by querying the Wikipedia API.

In this section, we will complete the Python file `getpage.py` to provide a function `getPage(page)` which returns, given a Wikipedia page title `page`, the title of `page` after following redirects, and the page titles of the first few linked pages on `page`. This section depends on Section 1.

1. We want to perform a query on the Wikipedia API to obtain in a JSON-encoded message the HTML text for a given page of the English Wikipedia, following Wikipedia redirects.

   [Redirects on Wikipedia are pages, such as "Filosophy", which redirect to a different page, such as "Philosophy". We need to specify that we wish to follow redirects so that, when we request the page "Filosophy", the API follows the redirect and returns the page "Philosophy".]

   Complete the function `getJSON(page)` to call the URL `http://en.wikipedia.org/w/api.php` with the following `GET` parameters: `format` set to `json`, `action` set to `parse`, `prop` set to `text`, and `redirects` set to `true`.

   Change the code at the end of the file to call the `getJSON(page)` function and check that, when running `./getpage.py` in the terminal, you obtain a JSON-encoded response containing the title and HTML contents of the requested page. Note that you could also use a Web browser to try out the API call yourself and see the response.

2. Looking at the JSON message, figure out where you can find the title of the page (after following redirects) and the HTML text of the page. (I advise to try this on a small page so that the JSON is still legible.) Complete the `getRawPage(page)` function which retrieves the JSON-encoded message using the `getJSON(page)` function and parses it using `json.loads`, so that it returns the page title and the HTML content.

   [Python exceptions such as `KeyError` are caught by putting the code which may raise the exception in a `try` block and defining afterwards an `except KeyError` block. In the provided function, this means that the default value `None` is returned whenever the JSON navigation fails, i.e., when the page does not exist.]

   [If `foo` is a Python object representing a JSON dictionary, `foo['bar']` yields the string or JSON object which is found at the key `bar` of the dictionary.]

   [To return two values from a Python function, we simply write `return a, b`. To retrieve them from a function call, you will simply write `a, b = f()`.]

3. Write a first version of a `getPage(page)` function which gets the title and HTML content of a page using `getRawPage(page)`, parses the HTML content using the BeautifulSoup library, and returns the page title and the list of the values of the `href` attribute of all the links of the HTML content, in the order in which they appear in the document. If the page does not exist, the function should return `None` as title and `[]` (the empty list) as links.

   [The empty list can be declared in Python as `l = []`, and an element can be appended to the end of the list with `l.append(42)`.]

   [To use BeautifulSoup, you should have a look at the examples in the BeautifulSoup documentation linked on the teaching page. Do not worry about imports: all the imports that you are likely to need are already provided in the files.]

4. Observe that `getPage(page)` returns many garbage page titles. A first strategy that we will use to eliminate some irrelevant links is to only keep links appearing in `<p>` elements directly in the `<body>` element (rather than links contained in infoboxes, banners, etc.). Amend the `getPage(page)` function to return the same values but only for the links obtained with this method. (Hint: use the `recursive` keyword argument to `find_all()`.)

   On the example page `https://en.wikipedia.org/wiki/User:A3_nm/COMASIC`, the first link found should now be "Philosophy". Check that "Success" is still returned. Do not worry about the fact that the list of linked page titles that is returned may be empty: this case is rare and we will not try to deal with it.

5. Observe that some invalid page titles are still returned. Devise a way to filter out links that are not internal links to an existing Wikipedia article, and amend the `getPage(page)` function to use it. Check that your strategy correctly eliminates external links (links that do not lead to a Wikipedia page) and red links (links that lead to a Wikipedia page that does not exist yet).

   [You can test using `s.startswith("foo")` if the string `s` starts with "foo".]

Use the test page `User:A3_nm/COMASIC` to check that you return exactly the links after the indication "Good content starts here." eliminating the red link and the external link. Do not worry about other problems such as anchors, percent-encoded characters, etc.; they will be dealt with in Section 4.

6. Amend the `getPage(page)` function to remove the `/wiki/` prefix on the result it returns so that you obtain page titles.

   [In Python, `s[12:34]` returns the substring formed by taking the string `s` from position 12 (included) to position 34 (excluded); `s[56:]` returns the substring from position 56 (included) to the end of `s`. The same works if `s` is a list. This is called slicing*.]*

7. Amend the `getPage(page)` function to return the page titles for the first 10 links at most. (Hint: use slicing.)

3 Writing the Web application

In this section, we will write a web application which allows users to play the "Six degrees to philosophy" game, using the `getPage(page)` function to get the titles of the first few linked pages. This section depends on Section 2.

1. Run the provided file `degrees.py` (with `./degrees.py`), connect to `http://localhost:5000/` using a Web browser, and check that you see "Hello world!". Look at the `degrees.py` file and at the `templates/index.html` file and make sure that you understand what is going on. You should probably have a look to the Flask Quickstart guide linked from the teaching page.

   [The Jinja directive {`% block body %`} defines a block *named `body`. You can ignore it for now. The {`# ... #`}-lines are comments.]*

2. Replace the contents of the body block of `index.html` with a form which performs a `POST` request to `/new-game` when submitted, and contains a text field so that the user can indicate the title of the page at which they wish to start the game. Use `placeholder` or `<label>` so that the purpose of the form is clear to the user. Change the `index` function to stop passing the useless `message` parameter. Refresh your browser and check that the form is correct. Try submitting it, and observe that the Python code does not know how to deal with the form submission yet.

3. Add a `/new-game` route to `degrees.py` for the `POST` method. Retrieve the start page title submitted by the user using `request.form`, store it in the `article` field of the `session` object, and redirect to `/game`. You will probably want to look to the Flask Quickstart guide, sections "The Request Object" and "Sessions". You do not need to use `url_for`.

   [Beware, Python function names cannot contain hyphens.]

   [The `session` object stores the data in cookies that are signed so that the user cannot tamper with the data.]

   [In debug mode, Flask will provide stack traces in the Web browser whenever something bad happens (obviously this should be turned off when publishing a real application), and it will automatically reload any modified files, so whenever you work on your code you do not need to restart Flask. However, if your Python file contains a syntax error, Flask will terminate and you will need to fix the error and then restart your program with `./degrees.py`.]

4. Add a `/game` route to `degrees.py` for the `GET` method in which you render the template `game.html`, passing the session object as a named parameter to `render_template`. Write a `game.html` template containing:

```
{% extends "index.html" %}

{% block body %}
  {{ session.article }}
{% endblock %}
```

[The Jinja `extends` directive in `game.html` means that `game.html` reuses the code of `index.html` except that the contents of the `body` block defined in `index.html` is changed. This ensures that all the HTML boilerplate is only written once.]

Check that you can submit the form and that the value retrieved by `POST` is correctly displayed on the result page.

[The reason why we perform a POST query on `/new-game` is because starting a new game is a state-altering operation. The reason why we perform the intermediate redirect is to ensure that the user can refresh and reload `/game` page without the risk of submitting a new POST request and starting a new game. This idiom is called "post-redirect-get".]

5. Alter the `/game` route to retrieve the current page title from `session['article']`, to call the function `getPage(page)` from Section 2 on the current page title to obtain the page title after redirect and the titles of the linked pages, and pass those results as named parameters to `render_template`. Now, amend the `game.html` template to display the article title after redirect, and to display the list of linked page titles as an HTML ordered list. (Just display the page titles as text, do not try to turn them into HTML links.)

   [In Jinja, you can iterate on a collection by writing:

   ```
   {% for item in items %}
     <li>{{ item }}</li>
   {% endfor %}
   ```

   This will generate one `li` tag per item in `items` containing the text representation of the item.]

6. Amend the `game.html` template so that each linked page title is rendered as an HTML radio button within a form that `POST` request to `/move` when submitted. The linked page title should appear twice: both in the `<label>` for the radio button and in its `value` which will be `POST`ed when submitting the form. You can use the `loop.index` Jinja variable to generate unique IDs for the radio buttons that can be referred to in the `for` attributes of the `<label>` elements. Check that the labels and radio buttons are correctly paired by trying to click on a label in your Web browser and verifying that the corresponding radio button is selected.

   [The `id` attribute in HTML documents should begin by a letter and not by a number.]

7. Add a `/move` route to `degrees.py` for the `POST` method. Retrieve the page to which the user wanted to move, save it as the new `article` attribute in the `session` object, and redirect to `/game`. Check with your Web browser that you can now move from page to page by submitting the form multiple times.

8. Amend the `/move` route to test if the user managed to reach the "Philosophy" page. If they do, redirect them to the index page, and use `flash` to display a notification on the index page indicating that they won the game. Check that you can now play the game and play multiple games in succession. Check that going to a redirect to "Philosophy" (e.g., "Filosophy") also allows the user to win.

   [You may want to read the "Message flashing" section of the Flask documentation, and add to `index.html` (outside of the `body` block but in the HTML `<body>` tag) the code to display flashed messages (the block beginning by {`% with messages = get_flashed_messages() %`}).]

9. Add a `score` field to the `session` which is initialized to zero in the `/new-game` route, is incremented in the `/move` route, is displayed to the user in `game.html`, and is flashed to the user when the game is won.

4 Refinements to the Wikipedia API

In this section, we fix some non-critical problems with `getPage(page)`. This section depends on Section 2.

1. Add a global variable `cache`, and use it to *memoize* the results of the `getPage(page)` function: whenever `getPage("Foo")` is called in an execution of the program, the returned values should be stored in `cache` such that, for subsequent calls of `getPage("Foo")`, the function can return the previous values without performing a useless query on the Wikipedia API. (Do not try to make the cache persist from one execution of the *program* to the other, it should only persist between function *calls* during one program execution.)

   [Hint: You should initialize `cache` *to be an empty* dictionary*, written* {}*.]*

   Check that the cache works by ensuring that there is no delay when you request the same page multiple times (both in the game and when running `./getpage.py` directly).

2. Amend the `getPage(page)` function to decode non-ASCII characters escaped in the `href` attribute. (Hint: Use the `unquote` function from `urllib` on the retrieved value for the `href` attribute. For technical reasons, you will need to turn the result into a Unicode object: if `x` is the output of the `unquote` function, use `unicode(x, 'utf-8')` instead of just `x`.

3. Amend the `getPage(page)` function to remove the URL fragment on page titles (e.g., remove "#History" from "Philosophy#History"). (Hint: Use `urldefrag` from `urlparse`.)

4. For cosmetics, amend the `getPage(page)` function so that underscores in the linked page titles are replaced by spaces.

5. Amend the `getPage(page)` function so that the output contains no duplicates. Preferably, the links should still appear in the same order as they do in the page, and the number of returned links should still be ten if there are ten different links that can be returned.

   [You may need to add an additional `import` *line at the beginning of the file to do this.]*

5 Improving the Web application

In this section, we fix a few problems in the Web application. This section depends on Section 3

1. Run a game, and open a new browser window on the current game by copying and pasting the URL. Perform moves alternatively in one window and in the other window. What happens?

   Amend the `game.html` template to add a hidden field containing the title of the current page, so that the current page is `POST`ed to `/move` so that the `/move` route can retrieve this value and check that the requested move was initiated on a page which reflected the current state of the game. Whenever this check fails, you should redirect to `/game` without altering the `session`, and instead use `flash()` to notify the user that their move was ignored because the game had progressed in a different window.

   Check that this test works properly and that using multiple browser windows on the same game now works in a sensible fashion.

2. Alter the `game.html` template to add a radio button allowing you to request a move to `Philosophy` in every state of the game. Check that you can indeed use it to teleport to "Philosophy" in any situation in violation of the game rules. Understand why this means a user can cheat, even if the template is not modified. Prevent such cheating by amending the `/move` route to check that the requested move is indeed possible by calling `getPage(page)` again and checking if the requested page is an allowed move. If the requested move is illegal, it should be ignored, possibly with an error message.

3. Try to submit the form on `/game` without selecting a radio button. What happens? Prevent this situation by selecting the first radio button in `game.html` (use the `loop.first` Jinja variable).

4. What happens when you reach a page for which `getPage(page)` returns no linked pages? Fix this so that whenever a page contains no links the user is redirected to the index page and a message is flashed to indicate that they moved to a page with no links and that they therefore lost the game.

5. Try to start the game by supplying something that is not the title of a page on the English Wikipedia. What happens? Fix this by redirecting the user to `/` and flashing an error message whenever the requested page does not exist.

6. Try to start the game on the "Philosophy" page. See what happens, and fix your program so that this case is correctly handled. Check what happens if the user enters the name of a redirect to "Philosophy", such as "Filosophy".

7. Fix your program so that an adequate error message is displayed when the starting page provided by the user has no linked pages.

6 CSS styling

In this section, we style the application using CSS so that it is pleasant to use. This section depends on Section 3.

Contrary to previous sections, there is no fixed roadmap: just make the game understandable to users and easy and pleasant to play using CSS. You can get some inspiration from the demo of the reference implementation shown this morning. Here are a few ideas:

- Pages should include a title (`<h1>`) which could be rendered as white text on a header with a dark background at the very top of the page.
- The header should be a link to abort the current game and go back to the index, but should probably not be decorated as a link.
- The current page and score could be displayed at the right of this header.
- Forms could be styled with a dark background, an `outset` border, and a minimum width.
- Form submission buttons should be at the bottom right of the form.
- When making lists of radio buttons, the list bullets should be removed.
- Adequate spacing should be provided between radio buttons.
- The radio button labels could change colors when hovered. The region which changes color should have a width that is common to all the radio buttons (and matches the width of the form).
- The success message and error messages should be styled differently (for this point you may want to use `get_flashed_messages(with_categories=True)`, see the Flask documentation), a possibility is to use red and green backgrounds and to put them just under the header.