José Haro Peralta and Peter Verkinderen[1]

"Find for Me!": Building a Context-Based Search Tool Using Python

The last decade has seen the beginning of what could become a methodological revolution in the fields of Arabic and Islamic Studies with the appearance of large collections of digitized classical Arabic texts.[2] The aim of this chapter is to show that open-source tools can be developed by researchers to utilize the existing collections of digital texts more comprehensively. We will focus on the possibilities that easy-to-learn but powerful programming languages like Python offer for advanced search operations. The authors of this chapter use Python for historical research with early Islamic texts and have built an open-source textual analysis toolkit, released under the name Jedli. In the second part of this chapter, we will present the basic building blocks of the Jedli program, with special focus on its context search function. We hope the ideas presented in this chapter can serve as an inspiration for other researchers to build more complex tools for textual analysis.

Jedli was developed within the framework of the research project "The Early Islamic Empire at Work: The View From the Regions Toward the Center," which is based at the University of Hamburg and funded by the European Research Council. This project aims at providing a better understanding of the political and economic structures of the Islamic Empire during its first three centuries by looking at the working mechanisms of five key regions (Fārs, Ifrīqiya, al-Jazīra, Khurāsān, and al-Shām).[3] Although the study of material culture (exemplified in coins and archaeological remains) forms an important part of the project, its

1 ERC project, "The Early Islamic Empire at Work—The View from the Regions toward the Center," University of Hamburg. The research leading to these results has been possible thanks to funding from the European Research Council under the ERC Advanced Grant no. 340362. We express our gratitude to all members of the project for providing useful feedback during the development of the Jedli toolkit as well as to the participants in the "Textual Corpora and the Digital Islamic Humanities" workshop at Brown University (October 17–18 2014), who made important suggestions and remarks on a preliminary version of this article. Special thanks go to our colleague Hannah-Lena Hagemann, whose comments and criticism contributed notably to improve the arguments developed in this article.

2 By 'digitized texts' we do not mean scanned PDFs of text editions, but texts which have been produced directly in a digital format, normally using a double-keying method (i. e., two typists type the same text independently, and the two texts are then compared to filter out typos).

3 See the project website: http://www.islamic-empire.uni-hamburg.de/.

main component is the analysis of textual primary source material, which is combed for information on the administration, economy, and elites of the key regions. The relevant text corpus consists of a large number of 'literary' (in the sense of non-documentary) texts that were written between the eighth and the thirteenth centuries CE and belong to different genres (historiography, geography, law, prosopography, and others).

The sheer magnitude of the corpus, the large scope of the research questions, and the limited research time available call for a strategy to retrieve information from the texts faster and in a more targeted way than is usually possible with traditional means of textual research, such as browsing through the indexes of edited works. This strategy makes use of the opportunities offered by digitized texts.

Collections of digitized Arabic texts began to appear in the 1990s, starting with digitized Qurans and *ḥadīth* works. The first of these collections appeared on CD-ROMs, but the most important ones are now available online.[4] The largest and most developed collection is currently al-Maktaba al-Shamela (http://www.shamela.ws), which has been online since 2005.[5] This digital library contains more than 6,500 books, divided into 76 categories. Not only does al-Maktaba al-Shamela have the largest collection of books, it also has an online platform (http://www.islamport.com), which allows a basic search across all the books within a specific category, and a dedicated desktop program, developed to read and search the collection. The desktop program offers an advanced search engine, which allows users to search for multiple words at a time, using OR and AND operators, in one or more books in the library. The latest version of the search engine also has options for disregarding different combinations of *alif-hamza* and dotted and undotted final *yā'*s and *tā' marbūṭa*s.

Although the inbuilt tools of some of these digital libraries can be used for complex searches in one or more documents, the existing programs are very rigid and do not give researchers control over what they can do with the texts. Digitized texts offer new research opportunities that were unthinkable with printed texts, but even simple tasks such as word counting, let alone more advanced operations, such as an analysis of vocabulary diversity, are not possible

4 For an overview of the most important websites, see http://islamichumanities.org/resources/. Some text collections, such as al-Jāmiʿ al-Kabīr, are still distributed on physical data carriers like flash drives and hard disks.

5 The first version of the program (April 2005) did not have a designated website but was distributed on the Ahl al-Hadeeth forum (www.ahlalhdeeth.com [sic]). The library moved to its own website in 2006.

with the tools offered by digital libraries. Moreover, search results cannot be exported for analysis and visualization in maps or graphs.

It must be added that the text collections also have a number of problems. In his presentation "Collections of Text vs. Textual Corpora, or What We Have and What We Need" at the Textual Corpora and the Digital Islamic Humanities Workshop in 2014 (October 17–18, Brown University),[6] Maxim Romanov pointed out that the currently available collections of digital texts are ill-suited for computational analysis: they aim at reproducing physical books rather than creating truly digital editions of the texts; their scope is limited, often on an ideological basis; the grouping into literary genres is inflexible and sometimes unhelpful; and metadata is incomplete and cannot be updated. We could add that the quality of the digitization is variable and not always based on high-quality editions. Moreover, the critical apparatus and footnotes are not dealt with in a consistent way.

Even if the collections of books that these digital libraries contain leave much to be desired, they do offer large quantities of digitized texts, including many of the most important sources for early Islamic history. These texts can be exported and converted to a format suitable for computational processing, such as .txt files. Once this is done, researchers can overcome the limitations of the tools offered by the above-mentioned digital libraries by building their own tools in a way that suits their needs.

The authors of this article have used the programming language Python to build a number of tools designed to find and retrieve information relevant to our research questions from Arabic texts. Programming languages are basically languages designed to communicate instructions to computers (and other machines). Python offers the advantage of being a a dynamic language, which means that a piece of code can be written and tested immediately, allowing for an interactive development experience of trial and error that eases the learning curve considerably. Python also contains a number of modules that are very suitable for textual analysis.

One of these modules that we are going to use extensively in this article is the RE (Regular Expressions) module. A regular expression is a sequence of characters that defines a pattern. This pattern can be used to search, select, and replace sequences of characters in a text.[7] For example, if we want to find the word

6 For the workshop program, see http://islamichumanities.org/workshop-2014/

7 For a gentle introduction to regular expressions, see Michael Fitzgerald, *Introducing Regular Expressions* (Sebastopol, CA: O'Reilly Media, 2012). More advanced coverage of this topic can be found in Jeffrey E. F. Friedl, *Mastering Regular Expressions* (Sebastopol, CA: O'Reilly

'color' in a text, but we do not know whether it is written in American or British spelling, we can use the following regular expression to match both forms:

```
colou?r
```

The question mark indicates that the preceding token (i.e., the 'u') may or may not be there. Therefore both 'colour' and 'color' will match this pattern.

1. Jedli's Main Functionalities

The authors of this chapter have built a data-mining toolkit for Arabic texts that consists mainly of three functions, namely an indexer, a context search function, and a highlighter. We will explain how these functions work, providing examples of how we use them in our own work. We will also suggest how researchers working on different topics might benefit from using these tools. In the second part of the chapter, we will focus on their technical aspects.

The first tool, the 'Indexer', lists all the pages in which a word appears. It can be used to search for one word at a time or be fed with a whole checklist of words, and it can undertake the search within one or more sources at the same time. Furthermore, it can either return a simple list of page numbers, or —for every page number—the surrounding context in which the word is found.

The main advantage of this function over manually searching for words in indexes of printed volumes is obviously its time-saving effect: the more words one needs to look up, and the more volumes one needs to search, the more time is saved. The Indexer is also more accurate than traditional indexes. In a test using the index of the *Bibliotheca Geographorum Arabicorum*,[8] a collection of exemplary editions of Arabic texts, the Indexer found significantly more results per search word than the printed index.

Furthermore, this Indexer is more powerful and flexible than any of the in-built search tools in the above-mentioned digital libraries, which all have indexing functions that can index a word in multiple sources at the same time. For one, it allows the user to index not only one term at a time, but also to feed it a checklist of search terms, which can be re-used and adjusted at any time. This is very convenient, since new relevant search terms may turn up while

Media, 2006) and in Jan Goyvaerts and Steven Levithan, *Regular Expressions Cookbook* (Sebastopol, CA: O'Reilly Media, 2012).

8 Michael Jan de Goeje, ed., *Bibliotheca Geographorum Arabicorum* (8 vols.: Leiden: Brill, 1870–94).

going through the results of the first search; these new search terms can then simply be added to the checklist for further indexing operations. Using regular expressions, one can restrict the number of results, excluding instances that are unlikely to be relevant. If we are looking for references to the province of Fārs, for example, we might want to leave off the 'outcomes list' instances of the *nisba* 'al-Fārisī' or cases in which the search word is preceded by numbers (as the text is more likely talking about horsemen, *fāris*). Moreover, regular expressions can also be used to define patterns that account for different spellings of words.[9]

The Indexer also gives the user full control over the output of the results: it can either return a simple list of page numbers or also include the contexts in which the word appears. The user can define how many context words before and after the search word are needed in order to determine if a result is relevant. One could also adapt the Indexer to define the context based on criteria other than number of words, e. g., punctuation, a number of lines in a poem, the beginning and end of a biography in a biographical dictionary, or an *isnād* in *ḥadīth* works. In addition, the Indexer saves the results for further reference, currently in an HTML document, but it can easily be adapted to output the results in a format that can be used for further analysis and visualization. For instance, the results of a search could be saved in a .csv document,[10] which can then be used to produce a graph so as to visualize how the results are spread over a selection of sources in order to spot patterns. If the search involves toponyms and is combined with a database of coordinates, it would also be possible to produce a map-based visualization of how different regions of the Islamic Empire are represented in a selection of texts.

The second tool we built is the 'Context Search' function. This tool was developed in the first instance to find information about the governors of our project's five key provinces. The number of sources that can provide information about this is very large, and going through all of them with the help of the Indexer would still require an enormous amount of time. We wanted to develop an approach that would allow us to gather some initial data quickly so we could start working on research hypotheses sooner.

9 To give only a few examples: defective spellings, different combinations of *alif* and *hamza*, dotted and undotted *tāʾ marbūṭa*s, and final *yā*'s. See the second part of the article for practical examples on how to build such regular expressions.

10 CSV stands for 'comma-separated values'; it is a common file format that is used to store tabular data in plain text form. Each line of the file contains a record, and each record has the same number of fields, separated by commas (or other delimiters).

The basic idea behind the Context Search function is that relevant information about a certain topic can be found if we can figure out in which kinds of contexts (as defined by their vocabulary composition) it is likely to appear. The Context Search function gives options to define contexts based on their length (number of words), which terms must appear in them, and even which words should not appear in them.[11] These checks are undertaken by feeding the function with checklists of words. It is therefore very important to build up these checklists carefully in order not to overlook relevant search results.

How we proceeded in our search for governors will illustrate how this tool can be used. In a first step, we used the Indexer to look up all contexts in which the name of a province was mentioned in one source. We then manually selected those search results that were related to governors. In a next step, we analyzed the vocabulary composition of these search results and identified the 'trigger words' in these contexts, on the basis of which we (consciously or unconsciously) had decided that the text fragment talks about a governor. The most effective trigger word was found to be *ʿalā* in combination with the name of the province (e. g., *ʿalā Ifrīqiya*, "in charge of Ifrīqiya"). Other trigger words included *wālī, wallā, wilāya, waliya, aqarra, ʿazala, ghalaba, fī yad, istakhlafa, ʿāmil*, and *dīwān*. These trigger words were put in a checklist, in a .txt document. We also analyzed how close to the name of the province these trigger words were located in order to define a word range that would limit irrelevant context while not excluding relevant context.[12] This word range is partly dependent on the verbosity of the author: in the case of the –very concise– *Taʾrīkh* of Khalīfa b. Khayyāṭ, the most effective word range consisted of eight words before and after the name of the province. More verbose authors such as al-Ṭabarī might require larger contexts.

The Context Search function first runs the Indexer to find all instances of the main search word in the text, setting a word range for the context. Instead of immediately outputting all the search results into a list of page numbers and text snippets, as the Indexer does, the Context Search has an intermediary step: it

11 Al-Maktaba al-Shamela's program allows the user to run a search with multiple search words, which can be connected with AND and OR operators. This is helpful, but the basic search unit in al-Maktaba al-Shamela is the page, which is not the most meaningful unit for textual analysis: on the one hand, the search words might be spread over more than one page, in which case our multiple-word search would not score a hit; and on the other hand, a page contains up to 3,000 characters, which means that it is very possible that the search words, even if they are on the same page, do not belong to the same context.

12 The Context Search could also be adapted to use other types of context range, as described above.

checks whether one or more of the trigger words from our checklist appears in the context. Regular expressions can again be used to account for different spellings of both the main search word and the trigger words and to allow for specific prefixes and suffixes to appear attached to these words but not to other characters. If a trigger word appears in the context, the result is put in the list of final outcomes; if none of the trigger words appear in the context, the result could be put into a separate list of 'probably irrelevant contexts' or immediately discarded. This is an interactive process; carefully checking the output results for irrelevant contexts in the 'relevant' list (and vice versa), and tweaking the checklist and the word range accordingly, will lead to ever better results.

One could also add another checklist of words that signal a context that is very unlikely to be relevant. If we use the Context Search function to look for governors of Fārs, for example, we can put expressions such as *alf fāris, miʾat fāris*, etc. (which refer to cavalry and not to the province) into this list. The Context Search function could then send contexts in which words from this list occur to the irrelevant results list, if no other mention of Fārs is made in the same context.[13]

Finally, the Context Search function can also be adapted so it can be fed with a checklist of main search words instead of only one main search word. For example, in the case of our own research, that checklist might include the names of the five key provinces of our project (Ifrīqiya, al-Shām, al-Jazīra, Fārs, and Khurāsān). The function would then search for information about the governors of all of these provinces at the same time.

The Context Search function is suitable for spotting passages in the sources that potentially contain information about certain topics, so long as these passages can be defined by the presence of specific vocabulary. It could, for example, be used to find information about the prices of certain products in a number of sources. In this case, the main search word might be the term *dīnār* or a checklist of main words that contains a number of currency units, including *dirham*, *qīrāṭ*, and others, together with their plural forms. Another checklist might contain a list of products whose prices we want to know, such as *ḥinṭa*, *shaʿīr*, or *khubz*.

The third function of the Jedli toolkit, the 'Highlighter', marks search words in a text with a user-specified background color. If we want different words to be highlighted in different colors, we can feed the Highlighter with different lists of

13 More checklists could be added, each with different rules for discarding and including contexts. The checklist mentioned in this paragraph's example acts on the main search word; we could, for example, build a third checklist that interacts with the trigger words of the first checklist.

words and apply a different color to each one of them. As with the Indexer, regular expressions can be used to reduce the number of irrelevant results. This function is useful when we want to read through a whole text but pay special attention to those passages that contain a number of keywords that are particularly relevant for our research.

The Highlighter was designed to mark toponyms belonging to the five key provinces of our project in the sources. Each researcher compiled a list of places in their province. These lists were fed to the program, which then produced documents of the sources in which the selected words were highlighted. In the case of toponyms that can apply to different places of the Islamic Empire (e. g., al-Sūs in the Maghrib and in Khūzistān) or could figure in some contexts as something other than a toponym (e. g., Fārs and *fāris*), they were moved to a second list that was highlighted in a different color within the text. A third list was also made, which contains words that often appear in conjunction with words from the second list in contexts where these words are not the toponyms in the province we are looking for (e. g., Khūzistān for al-Sūs, and *alf*, *miʾa*, etc. for *fāris*). This is a process of trial and error: once the Highlighter is run on a text, irrelevant contexts can be spotted in which a specific word is marked. This word can then be moved to the second checklist and these specific contexts analyzed to see whether there are words connected to the search word that signal the context is irrelevant. These signal words can then be added to the third checklist. The result is that one can scroll through a text and identify relevant passages in the blink of an eye, based on the color-coding.

The Highlighter can of course be used to highlight words other than toponyms. It is useful in many of the same cases as the Context Search function, but it can also be used to highlight structural elements of the text in order to make it easier for the reader to navigate the text. In a chronicle, for instance, one could highlight expressions that refer to years or dates in general; one could also highlight words that frequently appear in *isnād*s (e. g., *ḥaddatha*, *akhbara*) so that one immediately sees where a new *ḥadīth*/*khabar* starts.

The output of the Highlighter function is an HTML document that can be opened with any browser. The Highlighter inserts tags around the words to be highlighted, which the browser translates into color. As an additional feature, the program can attach a special symbol (e. g., '$') to every word from the checklists. This symbol is not visible in the text,[14] but it can be searched for, which facilitates reaching those text passages that contain highlighted words. Google

14 In the source code of the HTML document, the symbols are enclosed by tags with the <hidden> attribute to prevent it from displaying in the browser.

Chrome has a useful feature that can be used in conjunction with the Highlighter: when conducting a search with Chrome's built-in search function (Ctrl + F), the browser indicates the location of every search result in the document with a small yellow mark in the scrollbar to the right of the screen. If the hidden symbol is searched for, all sections of the document that contain highlighted words will be indicated in the scrollbar.

The tools we have described above have been made available to researchers in the Jedli toolkit. This toolkit has been released in two forms: one is a set of simple Python scripts that researchers can easily adapt to their own needs by adjusting the code. The other form of Jedli is designed for researchers who are not (yet) willing to interact with programming languages and scripts, but still want to use the powerful search capabilities of Jedli. Its graphical user interface,[15] with its buttons and input fields (see figures 9.1 and 9.2), looks like any other desktop program and does not confront the user with its underlying scripts. On the downside, the program with graphical user interfaces is more difficult to modify and tune to the specific needs of other researchers.

The remainder of this article is intended as an introduction to some of the possibilities that Python and regular expressions offer for the development of tools for textual analysis. It will take the guise of a tutorial on how to build simplified versions of the Indexer and Context Search functions described above. It is not intended as a full-blown introduction to Python,[16] but will build up the argument from very simple operations and explain all pieces of code in a way that should be understandable for people without previous programming experience. All the code examples in this article are available online (https://github.com/jedlitools/find-for-me).

15 The graphical interface of the Jedli toolkit is implemented using the tkinter library, which forms part of the standard package of Python. In order to learn more about this library and how to use it, see the following books: Mark Lutz, *Programming Python* (Sebastopol, CA: O'Reilly Media, 2013), 355–767; Bhaskar Chaudhary, *Tkinter GUI Application Development* (Birmingham: Packt Publishing, 2013).

16 For this, see any of the references mentioned by the Python Foundation at https://wiki.python.org/moin/IntroductoryBooks (modified May 24, 2015) as good starting points for learning the language. Especially recommended is Mark Lutz, *Learning Python* (Sebastopol, CA: O'Reilly Media, 2013).

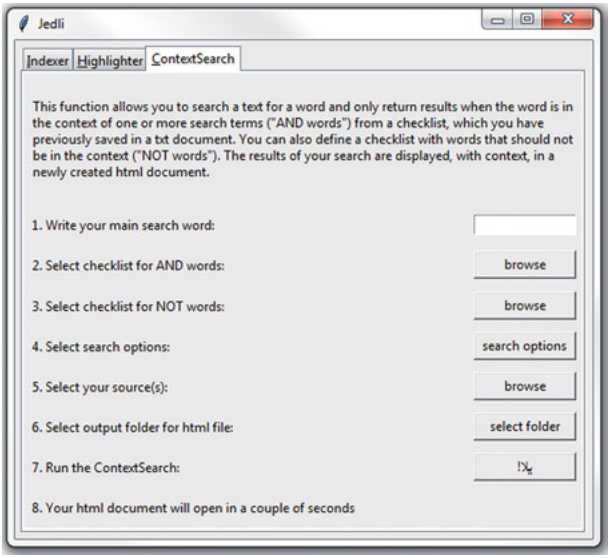


Figure 9.1. Jedli's Graphical User Interface—main screen (February 2015)

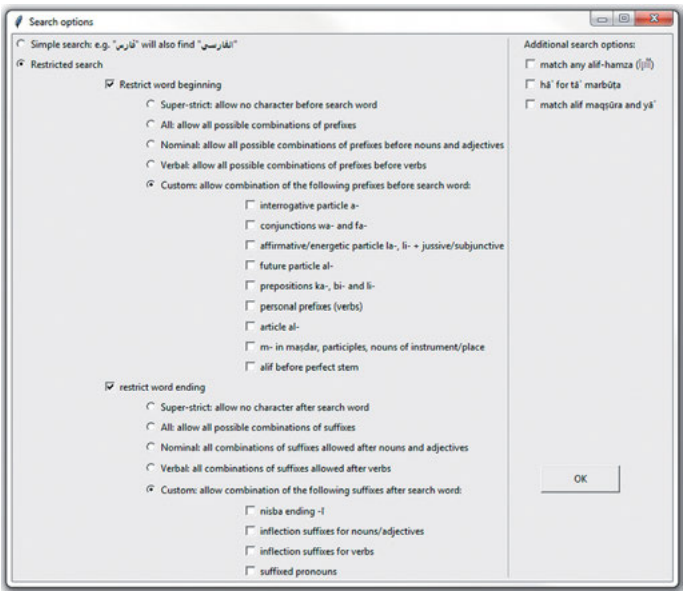


Figure 9.2. Jedli's Graphical User Interface—search options screen

2. Basic Python Operations

In order to use Python, it needs to be installed on the computer.[17] Once Python is installed, we can start using it by clicking the icon of Python's interactive interpreter, called IDLE, in the Start menu. IDLE functions basically like a text editor that assists in writing code.[18]

Once we open IDLE, we have to create a new Python file by pressing Ctrl+n (or using the menu: File > New File) and save it in a new folder. In order to make things easier, we advise placing all the Python files and the texts that we are going to analyze with them in the same folder.[19] For the examples in this article, we will use two texts: al-Balādhurī's *Futūḥ al-buldān* and Khalīfa b. Khayyāṭ's *Taʾrīkh*, which can be downloaded to the recently created folder from the following website: https://github.com/jedlitools/find-for-me. Other texts can also be used for experimentation.[20]

The first step in analyzing a text is 'opening' the text file, i.e., loading it into memory so that it is accessible to the program for processing. Using the "baladhuri_futuh.txt" file as an example, we can open the file with this line of code:

Code sample 1: ex1_basic_funcs.py

```
text = open('baladhuri_futuh.txt', mode='r', encoding='utf-8').read()
```

In this case, we assign the full text of the *Futūḥ* to a variable named `text`, using the = sign. Variables are basically empty memory containers in which values can be stored. Once a value is assigned to a variable in a Python file, we can refer to this variable at any point within the same file. This means that, as long as we keep working within this same Python file, any time we use the variable

17 For the Windows operating system, the installation package can be downloaded from the following URL: https://www.python.org/downloads/release/python-340/. Python comes preinstalled on the Mac OS and most Linux distributions. However, it must be noted that in this article we use Python 3.4. In case the reader has an older version, we advise updating it so the code that we will present here is fully compatible. For more on how to install Python, see the webpage of the Python Foundation or Lutz, *Learning Python*, 1421 ff.

18 For more on IDLE, see Lutz, *Learning Python*, 73 ff.

19 If the .txt files are in a different folder, the directory path where the .txt files reside has to be specified so the program can find them.

20 Additional texts can be downloaded from al-Maktaba al-Shamela in .epub format (by clicking on the mobile phone icon). This format must then be converted to .txt format using a converter such as Calibre or the converter that is distributed with the Jedli toolkit.

`text`, we will be referring to the *Futūḥ* of al-Balādhurī.[21] We can name variables in almost any way we want;[22] we could, for example, have opted also for `baladhuri`, `futuh`, or `source` instead of `text`.

`open()` is a built-in Python function that requires at least one argument (the name of the file we want to open, in this case 'baladhuri_futuh.txt') and admits a number of flags (optional parameters). Arguments and flags are written between the parentheses and separated by commas. The two flags that are of interest for us here are the `mode` and the `encoding` flags. With the `mode` flag, we specify whether we want to open the file for 'reading' (`r` – the function is set to this by default) or for 'writing' (`w`).[23] The `encoding` flag is of fundamental importance when working with non-English texts, since it specifies which protocol the function must use to interpret the characters in the text. In this case, we use the Unicode protocol utf-8. Note that the filename and the flags are all enclosed between quotes; single or double quotes (`' '` or `" "`) can be used for this.

The `.read()` at the end of the line is a method of `open()`; it specifies that the program should load the text from the text file in memory as a string object, i.e., as one continuous sequence of characters. Any change the program makes to the text loaded into the memory will not affect the original text in the .txt file, since we are only working with a representation of it loaded in the memory of our computer.

Now the text is available for any kind of analysis we want to perform. To get an idea of what the text looks like, we can print it. Printing the entire text could overload the interpreter, so we will print only a 'slice' of the text:

Code sample 2: ex1_basic_funcs.py (continued)

```
print(text[0:500])
```

This will print the first 500 characters of the text. In order to run the code, hit the F5 button. IDLE will ask to save the changes made in the file first; after clicking

21 This also means that if we open a new Python file and want to work with the same text, we have to load it in memory and assign it to a variable again, as we did here. There is more to this topic than we can cover here; for more information on how variables work in Python, see Lutz, *Learning Python*, 339 ff.

22 Only alphanumeric characters (numbers and letters) are allowed in the name of the variable. No spaces are allowed (use underscore instead). By convention, we write variable names in lower case; variable names should not be preceded or followed by underscores.

23 It also allows for some additional options that do not concern us here. See Lutz, *Learning Python*, 122 ff and the Python documentation at "2. Built-in Functions, open," last modified May 23, 2015, available at: https://docs.python.org/3.4/library/functions.html#open.

OK, a new window (called the 'shell') will pop up, and the text will be printed there.

We use square brackets behind the variable to refer to the position of the characters that we want to print in the text. This is called slicing. Square brackets can also be used to select one single character of the text (e.g., `text[0]` would print the first character of the text); this is called indexing.[24]

Another simple text operation is calculating how many characters it contains; we can do this with this line of code:

Code sample 3: ex2_basic_funcs.py

```
print(len(text))
```

Pressing F5, we can see that our text contains 723,413 characters. Here we have used the `len()` function, which counts how many elements an object contains.

2.1 Basic Search

To check how often a word appears in the text, or where, we have to import the `re` (Regular Expressions) module. Importing this module in Python is as easy as typing:

Code sample 4: ex3_basic_search.py

```
import re
```

Importing modules is usually done in the very first lines of the code. Like variables, once we import a module in a Python file, it remains available as long as we keep working within the same file. If we want to use this module in a different Python file, we must make the `import` statement at the very beginning of our code. Here, we will be using the function `findall()` from the `re` module, which searches for all the string sequences in the text that conform to a defined pattern. In order to ensure that Python can find the function `findall()` in the module `re`, we have to write `re.findall()`:

24 Index and slice notation in Python always starts with 0. That is, the first element of a string or list (or any other indexable object) is 0, not 1. On the other hand, the last index number in a slice refers to the character before which the slice will be cut off: in our example [0:500], the last character of the slice will be character no. 499, i.e., the 500th character, since we start counting from 0.

Code sample 5: ex3_basic_search.py

```
results = re.findall('البصرة', text)
print(results)
```

We store the outcome of the operation in the variable `results`. The `findall()` function takes two arguments: the first is the pattern we are searching for, and the second is the string in which the function should find this pattern. In our case, the pattern is the literal string "البصرة".

The result of hitting F5 is a list of every word that matches the pattern we have set. Notice that lists in Python are always symbolized by square brackets, and that since our list is a list of strings, every instance in the list is enclosed in quotes. In this case, because our pattern was unambiguous, we end up with a list of repetitions of the search word, one repetition for every time it is present in the text. This is arguably not extremely helpful in this form, but we will presently see how we can use the `findall()` function in more meaningful ways. We could, for example, count how many times the word is mentioned using the `len()` function that we already encountered before:

Code sample 6: ex4_basic_search.py

```
print(len(results))
```

This will print the number of times the search word is mentioned in the text. The power of regular expressions shows better when we build less 'literal' patterns, that is, when we use special symbols to build patterns in a more abstract way. For example, the symbol `\w` stands for any 'word character', which means any letter or digit (so-called alphanumerical characters). This allows us to build a rough regular expression to count all the words in the text:[25]

Code sample 7: ex5_list_of_words.py

```
list_of_words = re.findall(r'\w+', text)
print(len(list_of_words))
```

In regular expressions, the backslash is used to escape (i.e., overrule) the default meaning of a character and give it a different meaning. In the piece of code above, the backslash escapes the literal meaning of the letter `w`, and `\w` refers to any alphanumeric character. Some characters have a special meaning in reg-

25 Note that the code samples in this chapter build on the previous code. If the reader keeps working within the same Python file, this should not be a problem. If a new Python file is opened, it is necessary to import the `re` module in the first line of the code and to assign the *Futūḥ* of al-Balādhurī to the variable text again.

ular expressions by default. For example, a dot always stands for 'any character'; in this case, the backslash escapes this meaning, so that \. refers to a full stop. This use of the backslash may confuse the Python interpreter. It is therefore highly recommended to write an `r` before all regular expressions that include backslashes; this signals to Python to interpret the string as raw literals and removes any confusion over the backslashes.[26]

The plus sign signifies one or more repetitions of the preceding token; in our case, it will match any 'word character' until it reaches a non-word-character, which could be a space, a line break, or a punctuation mark, for instance. There are better ways to count words in a text,[27] but this is good enough for a first experimental approach. Our outcome is 179,788 words.

In order to get an impression of how Python identifies words in the text with the `\w` regular expression, we could print a 'slice' of the list of words, for example the first 50 words:

Code sample 8: ex6_list_of_words.py

```
list_of_words = re.findall(r'\w+', text)
print(list_of_words[:50])
```

We have again used slice notation (see code sample 2), in this case applied to a list. Note that on this occasion, we used the notation `[:50]`, which is identical to `[0:50]`. We can transform this list into a set, which is another Python object similar to the list, but which contains only one instance of every element (that is, it eliminates duplicates), and it does not store its elements in any particular order. It could serve as a rough approximation to know how many unique words the text contains (taking into account all the warnings given in footnote 27 about the inaccuracy of the approach taken here for word counting). We do this with these two lines of code:

Code sample 9: ex7_unique_words.py

```
unique_words = set(list_of_words)
print(len(unique_words))
```

26 See the Python documentation on "Regular expression operations" on this phenomenon (https://docs.python.org/3.4/library/re.html). On raw strings, see Lutz, *Learning Python*, 196–98.

27 Because the `\w` regular expression matches any alphanumerical character, in this case, we also count numbers as words. Note also that this function does not identify prefixes that are attached to words (such as the conjunction *wa-)* as separate words. For a more accurate approach, use the tokenizer that is distributed with the Natural Language Toolkit (NLTK), as discussed below.

Our outcome is 19,781.

For many research topics, keywords can be identified that allow us to select relevant passages in primary sources. Counting how frequently such words appear in a text can be useful at the beginning of a research project, when we want to select those texts that can potentially provide more information about the topic we want to study. A useful function in this context would be one that tells us how frequently a word appears in a text. The following lines of code do exactly that:

Code sample 10: ex8_word_frequency.py

```
word = 'فرضة'
word_instances = re.findall(word, text)
freq_word = len(word_instances)
freq_word = str(freq_word)
print(word + ' appears ' + freq_word + ' times in this text')
```

Testing this code (hit F5) with the *Futūḥ al-Buldān* of al-Balādhurī, we get the following outcome:

```
فرضة appears 2 times in this text
```

The first thing we do in this piece of code is to assign the string فرضة to the variable word. Then we use this variable in the `findall()` function, in order to search for فرضة in al-Balādhurī's *Futūḥ*, which is assigned to the variable `text`. We also use the `len()` function to count how many outcomes the search returns, and we store this value in the variable `freq_word`. In order to output the results to the Python shell, we use the `print` statement, which in this case uses the + sign to concatenate sequences of strings, including the string فرضة, which is stored in the variable `word`. Notice that the `len()` function always returns an integer (a data type different from string), so in order to be able to concatenate the value of the variable `freq_word` with the other strings in the `print` statement, we first need to convert it from integer to string, for which we use the function `str()`.

We can transform this code into a function so we can re-use it at a later point in the file. The following piece of code shows how to do it:[28]

Code sample 11: ex9_word_counter.py

```
def word_counter(search_word, search_text):
    freq_word = len(re.findall(search_word, search_text))
```

28 In IDLE, lines can be indented with the command Ctrl +].

```
freq_word = str(freq_word)
print(search_word + ' appears ' + freq_word + ' times in this text')
```

Functions are defined in Python with the `def` ('define function') statement, which must be followed by the name we want to give our function as well as parentheses and a colon. The parentheses can be empty, or they can contain the arguments needed for the function to work properly. In this case, the arguments are two variables, which we called `search_word` and `search_text`. These variables are then used in the body of the function: `search_word` will be the search pattern in the `findall()` function, and `search_text` will be the string to be searched in that same function. The actual names of the variables are not important, as long as we keep the same names in the body of the function when we refer to them.

Now we can start using ('calling') this function whenever we need it:

Code sample 12: ex9_word_counter.py

```
word_counter('فرضة', text)
word_counter('البصرة', text)
```

As can be seen in this example, we 'call' the function by writing its name and specifying the variables of the function. Note that the variables in the function call follow the same order as the variables in the function definition: `search_word` will be فرضة in the first function call and البصرة in the second; `search_text` will be the `text` variable to which we assigned above al-Balādhurī's *Futūḥ al-Buldān*.

2.2 Generating an Index

With all of these concepts and tools under our belt, we are now ready to extend the capabilities of these functions so they tell us also where exactly the word appears in the text—that is, we can build a simple index generator.

In order to generate an index, we need to find the word or expression we search for and the page reference. The `findall()` function from the `re` module that we have already encountered will serve our purposes for this task well. We have already seen how to find a word in a document using that function. The tricky part in this case is figuring out how to find both the word and its related page number in a single search.

If we have a look at the .txt document,[29] we will see that the pagination follows a very clear pattern, which looks like this:

الجزء: 1 ¦ الصفحة: 263

This pattern is followed throughout the document, and therefore we can describe it in a regular expression: we first have the Arabic word for volume (الجزء), followed by a colon, a white space, one or more digits that represent the volume number, another white space, a broken bar, another white space, the Arabic word for page (الصفحة), colon, again a white space, and finally one or more digits that represent the page number. Digits are symbolized in regular expressions by `\d`, and as we have already seen, the + sign can be used to indicate a repetition of the same token. If we want to express 'one or more digits', we write `\d+`.

If we try to write this regular expression, we will run into a problem with the IDLE editor, because mixing (right-to-left) Arabic and (left-to-right) Latin characters in the code will mess up its display, rendering it unreadable:

```
الجزء: \d+ ¦ الصفحة: \d+
```

One solution to deal with this is to assign the Arabic letters to variables and substitute them in the regular expression, using the + operator to concatenate the strings, as we have seen before:

Code sample 13: ex10_index_generator.py

```
juz = 'الجزء:'
safha = 'الصفحة:'
page_regex = juz + r' \d+ ¦ ' + safha + r' \d+'
```

Now that we know how to find page numbers and how to search for words, we need to find a way to connect these two elements. We can do this by including in our regular expression our search word, the `page_regex`, and all the characters in between. Such a regular expression would look like this:

```
search_regex = word + r'.+?' + page_regex
```

As we have seen above, the dot is a special character that matches any character. The question mark tells the regular expression not to be greedy, that is, to stop at

29 We recommend using the text editor EditPad Pro (http://www.editpadpro.com/, only available for Windows) for this, since it can handle large .txt documents better than other text editors.

the first match of the `page_regex` it encounters. With this regular expression, the result of our search would be a block of text that starts with the search word and ends with the page number of the page on which the search word was found. However, what we really want in the final result is just the page number. To achieve this, we add parentheses around the elements of the regular expression we are interested in, which will ensure that only those elements will be included in the list of results produced by the `findall()` function. Because these parentheses 'capture' the elements they contain, they are called capturing groups in regular expressions. The resulting function would look like this:

Code sample 15: ex11_index_generator.py

```
def index_generator(word, text):
    juz = 'الجزء:'
    safha = 'الصفحة:'
    page_regex = juz + r' \d+ ¦ ' + safha + r' \d+'
    search_regex = word + r'.+?(' + page_regex + ')'
    pagination = re.findall(search_regex, text, re.DOTALL)

    return pagination
```

As we have seen before, regular expressions in Python are always strings, and we can concatenate strings by using + signs. Note that the brackets of the capturing group need to be put between pairs of quotes, because they are part of the search regex (short for regular expression) string; the variables need to be outside of the quotes, however, because otherwise Python will consider them literal strings.

This function contains two new elements that need a short explanation: the first is the use of the flag `re.DOTALL` in the `findall()` function. We said before that the dot in a regular expression matches any character, but in fact, it matches any character except a newline, which is represented by the `\n` character in a string. If we include the flag `re.DOTALL` in the `findall()` function, the dot will match anything, including the newline character. The second is the `return` command. Contrary to the `print` statement, which outputs the result of the function directly to the Python shell, the `return` command returns the result of the function (in this case, the list `pagination`), so we can assign it to a variable and use it later in our code. We can now call our new function—adding between its parentheses the two arguments it needs: the search word and the reference to our text—and print the outcomes:

Code sample 16: ex11_index_generator.py

```
index = index_generator('فرضة', text)
print(index)
```

For the word *furḍa*, the `index_generator()` will return the following outcome:

```
['الجزء: 1 ¦ الصفحة: 333', 'الجزء: 1 ¦ الصفحة: 286']
```

In case the word we are looking for appears very frequently in the text, the list of results will look cluttered. It would be better if we printed every search result on a new line. This is easily done with the following piece of code:

Code sample 17: ex12_index_generator.py

```
index = index_generator('فرضة', text)
for page in index:
    print(page)
```

Here, we use a `for` loop. In a `for` loop, the header line of the `for` statement ends with a colon, and the line(s) that belong to its scope are indented. Note that we use `page` here as a variable to refer to every element in the index list, but we could have given this variable any other name. We can read the statement as: 'print every element in `index`'. This is the result if we run the `index_generator()` now:

```
الجزء: 1 ¦ الصفحة: 333
الجزء: 1 ¦ الصفحة: 286
```

Loops are very powerful and allow us to make our index function much more useful in a number of ways. For example, they allow us to search for several words at the same time:

Code sample 18: ex13_index_more_words.py

```
search_words = ['فرضة', 'البصرة', 'الكوفة']
for word in search_words:
    index = index_generator(word, text)
    print(word)
    for page in index:
        print(page)
```

This will make an index for every word in the list `search_words`. We can take this approach a step further. Instead of using a list of search words defined within our Python code, we could write the words we want to search for in a separate file, e. g., a .txt file. This would be especially convenient if we were to handle a large list of words. Such a file can be accessed by Python with the `open()` function we used before (see code sample 1) and its list of words assigned to a var-

iable. For this, we have to open a text editor and write every search word on a new line (without leaving empty lines between them). Then we save the file in the folder with our source file (in our case, the al-Balādhurī text file), making sure we give the file a .txt extension (which is the default in a text editor).[30] We name this document "checklist.txt." The following lines of code show how to access the checklist and build an index of its words:

Code sample 19: ex14_index_checklist.py

```
search_words = open('checklist.txt', mode='r',
            encoding='utf-8-sig').read().splitlines()
for word in search_words:
    index = index_generator(word, text)
    print(word)
    for page in index:
        print(page)
```

The `open()` function loads the entire document as one string into memory. The encoding name in this case is 'utf-8-sig', which is here necessary in order to drop a byte order marker sequence that would otherwise appear attached to the first word in the list.[31] Notice the `splitlines()` method added at the end of the `open()` function. This method builds a list in which each line of the original document is an individual element. Since we wrote every search word on a new line, each search word from our checklist document is now stored as a separate element in the `search_words` list.

We can go even further: instead of indexing these search words in one text at a time, we could index them in a collection of texts stored in a specific directory or folder. For this, we first need to create a sub-directory within the directory in which we work and store in it all the sources in .txt format that we want to index. We call this directory 'sources'. The following code shows how to build an index of all the words contained in the checklist.txt file for each of the texts stored in the sources directory:

Code sample 20: ex15_index_directory.py

```
import os

search_words = open('checklist.txt', mode='r',
            encoding='utf-8').read().splitlines()

for filename in os.listdir('sources'):
    text = open(filename, mode='r', encoding='utf-8').read()
```

30 You can download a sample checklist from: https://github.com/jedlitools/find-for-me

31 For more on byte order markers, see Lutz, *Learning Python*, 1201 ff.

```
print(filename)
for word in search_words:
    index = index_generator(word, text)
    print(word)
    for page in index:
        print(page)
```

In this piece of code we import a new module, called `os` ('operating system'), which contains a function named `listdir()`. This function builds a list of all the file names in the directory specified as an argument for the function, in our case `sources`. For every file in the list, we first load the text into memory, then print the name of the file, and finally index each of the words from the checklist.txt file (assigned to the variable `search_words`) in that text. Then the program moves on to the next file in the folder, until it reaches the last file.

2.3 Enhanced Search[32]

One problem with searching Arabic texts is that they include diacritics, such as vowels, *shadda*s, and the like, in an unpredictable way. Since these diacritics are represented by separate characters in the text, their presence can sabotage our searches. The easiest way to deal with this problem, if we are not specifically interested in the vowels, is to temporarily remove all of them from the text loaded in memory. We can do this by using the `sub()` function from the `re` module, which allows us to replace one string with another. In our case, we will replace all the diacritics with empty strings (which are coded in Python by a pair of quotes with nothing in between):

Code sample 21: ex16_denoise.py

```
denoised_text = re.sub(r"ó|ǒ|ǒ|ǒ|ọ|ọ|ǒ|ǒ|-", "", text)
```

In addition to the diacritics, we also included the *kashīda* character.[33] As can be seen, all diacritics are separated by the pipe (|) symbol, which in regular expressions signifies the `or` operator. Removing the diacritics from the text makes

32 The discussion that follows draws heavily on regular expressions-related concepts. For further clarification on any of these concepts, see the references mentioned in note 7.

33 The *kashīda* is the character used to elongate (*taṭwīl*) Arabic characters, e.g. in بـــسم الله.

searching for words and expressions easier, since we do not have to account for all possible vowelizations of the words.[34]

So far, we have used regular expressions in a limited way, searching only for simple strings. This may not be a problem if we search for words that form a unique sequence of characters, like البصرة, but it would not return good results if we looked for a word like حكم. Running the `word_counter()` function that we built before (code sample 11) with the string حكم in the *Futūḥ al-buldān* returns 112 hits. The problem is that these hits also include many potentially undesirable outcomes, such as الحكم, أحكم, أصالحكم , حكمة and so forth, because these words also contain the string حكم.

In order to limit our search results only to the words we are interested in, we need to use more complex regular expressions. For example, if we only wanted all instances for the word حكم, we could use the expression `\b`, which identifies boundaries around the word. The expression `\b`حكم`\b` implies that no alphanumeric character can precede the *ḥā'* or follow the *mīm*. As previously stated, when we write regular expressions that contain special characters, we have to write an `r` before the opening quotes of the regex string, like this:

Code sample 22: ex17_word_boundaries.py

```
word = r"\bحكم\b"
word_counter(word, text)
```

This returns only 10 results. However, in Arabic, a number of prefixes and suffixes can be attached to a word without actually altering its meaning, so we may want to include in our list of results all instances of the word with those affixes. For instance, if we also want to include the word when it is preceded by the conjunction *wa-*, we can use this regular expression: `\b`و?حكم`\b`. It will first look for a word boundary, then zero or one occurrences of a *wāw*, then the string حكم, and finally another word boundary. The question mark after the *wāw* serves to make the conjunction optional, that is, to search for the word both with and without it. If we run the `word_counter()` function with this new regular expression, we obtain 12 results (*ex18_prefixes_conjunctions.py*).

If we also wanted to include the prefix *fa-*, we can use the pipe (`|`) character, which symbolizes the `or` operator, between the prefixes: و|ف . To make the presence of the prefixes optional, we will need to group them between parentheses, followed by the question mark: `(`و|ف`)?`. As we have seen before, however, the parentheses form a capturing group, which means that only the elements within

34 See Maxim Romanov, "Python Functions for Arabic," *al-Raqmiyyāt: Digital Islamic History,* January 2, 2013, available at: http://maximromanov.github.io/2013/01–02.html.

the parentheses will be returned as an outcome. We therefore need to include the prefixes in a non-capturing group, which is formed by placing a question mark and a colon after the opening parenthesis: `\b(?:و|ف)?حكم\b`. This regular expression yields 16 results (*ex19_prefixes_conjunctions.py*).

Building on this regular expression, we can now build an expression that includes all the personal prefixes that حكم as an imperfect verb can have, in addition to the conjunctions *fa-* and *wa-*. Since we can have only one of the conjunctions combined with one of the personal prefixes, we just have to add an optional group with the verbal prefixes after the conjunctions in our regular expression: `\b(?:و|ف)?(?:أ|ت|ي|ن|ا)?حكم\b`. This time, the function returns 22 results (*ex20_prefixes_verbs.py*).

If we take into account that prefixes in Arabic always appear in the same relative order, as shown in Table 9.1, we can build a regular expression that uses optional non-capturing groups to define the most frequent combinations of prefixes. Such a regular expression could look like this (*ex21_prefixes_all.py*):

`\bأ?(?:و|ف)?س?ل?(?:ك|أ|ت|ي|ن|ا)?(?:ل|ال)?(?:ا|م|إ)?حكم\b`

Another approach would be to group all prefixes together in one non-capturing group and to define the maximum number of possible combinations among them by using curly brackets—for example:

`\b(?:و|ف|ب|ك|ل|ال|أ|س|ت|ي|ن|ا){0,6}حكم\b` (*ex22_prefixes_all.py*).

Both regular expressions in ex21 and ex22 return the same number of outcomes (93), but the latter regular expression is approximately 15 percent faster.

Using similar regular expressions, we can also deal with the suffixes. Table 9.2 shows the most frequent combinations of suffixes. In this case, we put the optional groups after the search word (ex23_suffixes_all.py):

`\bحكمي?(?:ا|و|ن|ت|ي|ه|ة|ى|ك|ن|ا|ت|م|ه|ك|ا|ن|ه|ا|ت|م|ه|ك|ا|ن|ه|ا|ت|م|ه|ك|ا|ن|ه|ا)?\b`

As with the prefixes, another approach would be to group all suffixes together and indicate between curly brackets how many of them can be combined at the same time (ex24_suffixes_all.py):

`\bحكم(?:ا|و|ن|ت|ي|ه|ة|ى|ك|ن|ا|ت|م|ه|ك|ا|ن|ه|ا|ت|م|ه|ك|ا|ن|ه|ا){0,4}\b`

Both regular expressions return 19 results.

We could now assign these regular expressions for suffixes and prefixes to variables, which we can concatenate with the search string using the + sign:

Code sample 23: ex25_affixes_all.py

```
pre_all = r"(?:و|ف|ب|ك|ل|ال|أ|س|ت|ي|ن|ا){0,6}"
su_all = r"(?:ا|و|ن|ت|ي|ه|ة|ى|ك|ن|ا|ت|م|ه|ك|ا|ن|ه|ا|ت|م|ه|ك|ا|ن|ه|ا){0,4}"
search_regex = r"\b"+pre_all+"حكم"+su_all+r"\b"
```

Table 9.1: Relative order of prefixes in Arabic

1	2	3	4	5	6	7	
interrogative particle	conjunction	affirmative/ energetic particle *la-* *li-* + jussive /subjunctive	future tense particle	preposition *li-*, *bi-*, *ka-* personal prefixes (verbs)	article	participle / maṣdar prefix *mu-* noun of instrument / place *m-* perfect prefix *i-*	[stem prefixes (verbs)][a]
أ	و|ف	ل	س	ا|ن|ي|ت|أ|ا|ل|ب|ك	ال	ا|م|!	[أ|ت|ن|س|ت]

Categories in columns one to seven can be combined; prefixes inside the same categories are mutually exclusive.
[a]Since verbal stems are not only determined by prefixes, but also by infixes, we would opt to make a separate search word for every verbal stem we look for.

There is still another problem when dealing with digital Arabic texts, which is that *hamza*s, *madda*s, and *waṣla*s are not written in a consistent way. Since these combinations have their own Unicode representations, they are considered separate characters in our searches. For example, if we search for the word أصغر with *hamza* in the *Taʾrīkh* of al-Ṭabarī, we obtain 43 outcomes. However, the text also contains an additional 27 instances of the word اصغر written without the *hamza*, which did not show up in our list of results for أصغرwith *hamza*.

Table 9.2: Relative order of suffixes in Arabic

	suffix type	suffix type Arabic	combined suffixes
1	nisba *-ī*	ي	ي
2	female ending *tāʾ*	ت	ت|ا|ي|و|ات|ن|تم|تمو|تما|تن|تا|نا|وا
	female ending *alif*	ا	
	nominal inflection suffixes without *nūn*	ا|ي|و|ات	
	verbal inflection suffixes without indicative-specific *nūn*	ت|ا|ي|و|ات|ن|تم|تمو|تما|تن|تا|نا|وا	
3	verbal inflection final *nūn*	ن	ن
	energetic suffix *-anna*	ن	
4	*tāʾ marbūṭa, alif maqṣūra*	ة|ى	ة|ى|ني|ي|نا|ك|كم|كما|كن|ه|ها|هم|هما|هن|ها|ن
	pronominal suffixes	ني|ي|نا|ك|كم|كما|كن|ه|ها|هم|هن|هما|ن	
	nominal inflection final *nūn*	ن	

Categories in rows one to four can be combined; suffix types within the same category are mutually exclusive so they cannot be combined.

There are two ways to deal with this problem, so that our searches yield all the results we want. One is similar to what we did with the short vowels: replace all combinations of *alif*s with *hamza*s, *madda*s, or *waṣla*s in the text by simple *alif*s, using the `sub()` function from the `re` module that we have already used:

Code sample 24: ex26_alifs.py

```
modified_text = re.sub("[أآإٱ]", "ا", text)
```

If we perform this operation, there will be no more combinations of *alif* with *hamza*, *madda*, or *waṣla* in the text, so we would only have to search for اصغر without *hamza* to obtain all 70 results.

Another option is to act on the level of the search word rather than the searched text: we could make explicit that we are searching for any of the *alif* combinations:

Code sample 25: ex27_alifs.py

```
search_results = re.findall("[أإآا]صغر", text)
```

The square brackets in a regular expression define character classes, which will match any of the characters inside the brackets. This regular expression also yields 70 results in the text of al-Ṭabarī.

The *tāʾ marbūṭa* and the *alif maqṣūra* suffer from similar problems as the *alif* in our texts: sometimes the dots above a *tāʾ marbūṭa* or the dots under a *yāʾ* are left out, and sometimes the *alif maqṣūra* is dotted. We can use the same two strategies as with the *alif*s to deal with these problems.[35]

2.4 Contextual Search

We will now show the basics of a search function that allows the user to define the context in which search words should occur.[36] In order to illustrate how we implemented the `context_search()` function of the Jedli toolkit, we will use here the case of Ifrīqiya in the *Taʾrīkh* of Khalīfa b. Khayyāṭ as an example.

The first thing we need to do is to create a new .txt file (using a text editor), which we might call governors_checklist.txt, and save it in the same directory in which we store our Python files. In this document, we have to write down the list of trigger words related to governmental functions mentioned at the beginning of this article, one word in each line (avoiding empty lines), as we did when we created the checklist.txt file (code sample 19).[37] Then we have to load the text of Khalīfa's *Taʾrīkh* into memory and assign it to a variable, just as we did before with the text of al-Balādhurī. We will also remove its vowels so our searches can work effectively:

Code sample 26: ex28_context_search.py

```
import re
```

---

35 We could replace all instances of the *tāʾ marbūṭa* in the text with a *hāʾ* without dots, and all instances of *alif maqṣūra* with normal *yāʾ* with the following regular expressions:

```
modified_text = re.sub("ة", "ه", text)
modified_text = re.sub("ى","ي", text)
```

If we want to do the change on the level of search, we can use the following regular expressions:

```
search_results = re.findall("البصر[ةه]", text)
search_results = re.findall("مول[يى]", text)
```

36 See above for more on this function.

37 You can download the .txt file containing the list of words from https://github.com/jedli-tools/find-for-me.

```
text = open('khalifa_tarikh.txt', mode="r", encoding="utf-8").read()
text = re.sub(r"ó|ó̠|ó̠̊|ọ|ọ̠|ó̠|ó|-", "", text)
```

This piece of code nicely illustrates how a variable works like an empty box. Assigning the `open()` statement to the variable text, we put Khalīfa's text into the box; then we take it out of the box to remove all vowels with the `sub()` operation and put the modified version back into the same `text` box.

The next thing we need is to write a function that returns a list of words from the governors_checklist.txt file. The following code, based on what we saw in code sample 19, does the job:

Code sample 27: ex28_context_search.py

```
def search_words(checklist):
    search_words = open(checklist, mode='r',
            encoding='utf-8-sig').read().splitlines()
    return search_words
```

Notice that we use here the `return` command instead of the `print` statement, as we did in code sample 15. Now we have to find a way to connect each of the terms from the checklist with the name of the province we are interested in. In order to facilitate this task, we will search first for all contexts in which the name of a region appears, and then check whether in those contexts we can find one of the words from the checklist. But first, we have to figure out how long the context should be—that is, how many words around the name of the region we should retrieve from the text to make sure that government-related words are going to appear in it, in those cases in which the chronicler is giving information about the governors of the region.

In our analysis of Khalīfa's text, we found that the optimal context length for this consists of eight words on both sides of the search word (the name of the region in this case). In order to define words, we will use a regular expression with a pair of special characters: `\s` and `\S`. The former refers to any whitespace character (space, tab, etc.), the latter to any character that is not a whitespace (which will include not only word characters, but also line breaks, punctuation marks, and the like). Essentially, we are defining words here as sequences of one or more non-whitespaces followed by one or more whitespaces.[38] The following

38 Note that this assumption would not be valid for linguistic analysis, because the definition of 'word' that we use here also includes punctuation marks and other non-alpha-numeric characters.

regular expression would capture a context of zero to eight words around a variable called `region`:

Code sample 28: ex28_context_search.py

```
r"(?:\S+\s+){0,8}"+region+r"(?:\s+\S+){0,8}"
```

Note that we expect the variable `region` to be preceded and followed by a whitespace; for this reason, the `\s` and `\S` characters in the regular expression appear in reversed order on both sides of the variable `region`.

In order to substitute the Arabic word for Ifrīqiya with the variable `region` within the regular expression developed above, we have to take into account that this name can appear under a number of variants in Arabic texts: the *alif* might bear the *hamza* either above or below, or not bear a *hamza* at all; the word might end either in a *tāʾ marbūṭa*, which might bear the dots or not, or in an *alif*. Besides, it is possible that the word is preceded by a conjunction and/or a preposition. This is therefore a good case in which we can apply the techniques described above to deal with these kinds of situations:

Code sample 29: ex28_context_search.py

```
region = r"[وفبل]{0,2}"+r"[اأآإ]" +"فريقي" +r"[اةه]"
```

Now we can use the function `findall()` from the `re` module in order to retrieve from the text all contexts in which the word Ifrīqiya appears. The following piece of code achieves this:

Code sample 30: ex28_context_search.py

```
def context_search(region, checklist):
    gov_words = search_words(checklist)
    regex = "(?:\S+\s+){0,8}"+region+"(?:\s+\S+){0,8}"
    contexts = re.findall(regex, text, re.DOTALL)
    outcomes = []
    for passage in contexts:
        for word in gov_words:
            pre_all = r"(?:و|ف|ب|ل|ك|ال|أ|س|ت|ي|ن|ا){0,6}"
            su_all = r"(?:ا|تا|كا|ها|ان|ات|تم|هم|كم|ة|تمو|كن|هما|ي|وا|اني|ات|هن|تن|كما|ة|ه|ى|وا|ن){0,4}"
            regex_w = r"\b" + pre_all + word + su_all + r"\b"
            if len(re.findall(regex_w, passage)) > 0:
                passage_page = index_generator(passage, text)
                passage = re.sub(r"\n", " ", passage)
                outcomes.append((passage, passage_page))
                break
    return outcomes
```

We use the `search_words()` function we defined above to assign the list of words related to governors to the variable `gov_words`. Then we use the regular expression we defined before in code sample 28 to identify all the contexts in which the name of the region appears. We store the outcomes of our search in the variable `results`. Then we create an empty list, named `outcomes`, which we will presently use to store the final results of our function. After that, we check for each of the passages in the `results` list to see whether they contain any of the trigger words[39] from the `gov_words` list. For this, we have to use two `for` loops—one to step through all the passages stored in the variable `contexts`, and another to iterate through each of the words in the `gov_words` list; an if statement checks if the condition is met.[40] If the `findall()` function finds at least one instance of a trigger word in the passage, we use the `index_generator()` function we defined in code sample 15 to find its page number. We then use the `append()` method to add to the `outcomes` list a tuple[41] that contains two elements: the passage itself and the page number. In case a passage contains more than one of the words from the `gov_words` list, it would be added to the `outcomes` list once for every word, because the `if` statement is performed for each word in the `gov_words` list. In order to avoid this, we use a `break` statement, which will stop the `for` loop that steps through the `gov_words` list as soon as the condition is met once and the passage has been added to the `outcomes` list.

In order to call the `context_search()` function, we have to add the required arguments between the parentheses: the name of the region (in our case, the regex formerly defined and stored in the variable `region`) and the name of the file containing the list of words related to governors. The function will return the variable `outcomes`, which must be stored in another variable (here `governors`) so we can print the results:

Code sample 31: ex28_context_search.py

```
governors = context_search(region, 'governors_checklist.txt')
```

39 We use the regular expressions for the prefixes (`pre_all`) and suffixes (`su_all`) that we developed earlier to include possible combinations of affixes that can appear around the trigger words.

40 On the '`if`' statement, see Lutz, *Learning Python*, 320 ff.

41 A tuple can be described as a list that is locked: it is immutable; its elements cannot be changed. Contrary to a list, which is enclosed in square brackets, a tuple is always between parentheses.

If we print the variable `governors`, we will get a list of all the tuples containing the relevant passages and their page numbers that the function has returned. This output is not very readable. In order to produce a more user-friendly formatting, we can print these values in the following way:

Code sample 32: ex29_context_search.py

```
e=1
for s, p in governors:
    print(e, "\n", s, "\n", p, "\n\n")
    e = e+1
```

We use a for loop to step through each of the tuples contained in the list returned by the `context_search()` function. We assign variables for each of the two elements in the tuple (`s` for the passage, `p` for the page) and print these separately, putting line breaks (`\n`) in between. This is called value unpacking, and it allows us to handle separately the elements contained in a tuple.[42] We can also number the results by introducing a new variable, `e`, to which we initially assign the value 1, and increment its value by another unit for every step in the loop.

The current version of the Jedli toolkit allows the user to undertake contextual searches in this way, although we are currently working on an enhanced definition of context that will allow the user to search for more complex contexts. In the Jedli toolkit, the results are not printed to the Python shell, but saved as an HTML file that we can then open with a browser. In the HTML file, the search and trigger words are highlighted in different colors.

3. Conclusions

This article has introduced a number of basic functions that can be developed in Python as building blocks for the implementation of a fairly complex context search function. These building blocks are also core elements of the Jedli toolkit for the textual analysis of Arabic works. The reader of this article will now hopefully understand the Jedli toolkit code without graphical interfaces and be able to adapt it to their own needs and contribute to its improvement. Alternatively, the reader could use these building blocks to develop their own code for textual analysis.

42 This is just an extended way of making variable assignments. For more on value unpacking, see Lutz, *Learning Python*, 341 ff and 396 ff.

Jedli is a basic toolkit for textual analysis, but it represents a first step in the development of more complex tools for more advanced analyses of medieval Arabic texts. One possible direction in the enhancement of Jedli could be to integrate it into existing third-party libraries for Python for complex textual analysis. One such library is the Natural Language Toolkit (NLTK), developed originally by Steven Bird (University of Melbourne), Edward Loper (BBN Technologies), and Ewan Klein (University of Edinburgh).[43] This library includes tools such as a complex tokenizer, stemmers, and several others that allow us to perform lexical or word frequency analysis as well as parts-of-speech tagging, to name just a few. With the help of these tools and a few more lines of code, for example, it is possible to build a simple program that analyzes and measures the degree of similitude between two or more different texts.[44] More specialized libraries are also available that let us perform more complex tasks, such as topic modeling.[45]

Future development in this direction could lead to the implementation of complex analytical tools for linguistic analysis and textual criticism. As a single algorithm can perform the same analysis over and over again through large collections of texts, this approach could allow us to reach a better understanding of the chroniclers' sources, something on which the authors and compilers of the extant text corpus often provided no information. It could also shed light on how traditions were transmitted and modified over time, how words developed new meanings, or how the style of language employed by medieval authors varied according to chronology, geography, or literary genre.

Bibliography

Bird, Steven, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. Sebastopol, CA: O'Reilly Media, 2009.

Chaudhary , Bhaskar. *Tkinter GUI Application Development*. Birmingham: Packt Publishing, 2013.

de Goeje, Michael Jan, ed. *Bibliotheca Geographorum Arabicorum*. 8 vols. Leiden: Brill, 1870–94.

43 The best available introduction to the NLTK is Steven Bird, Ewan Klein, and Edward Loper, *Natural Language Processing with Python* (Sebastopol, CA: O'Reilly Media, 2009). See also the website of the NLTK project: http://www.nltk.org/.

44 Willi Richert and Luis Pedro Coelho, *Building Machine Learning Systems with Python* (Birmingham, Packt Publishing: 2013), 57 ff.

45 Ibid., 75 ff.

Goyvaerts, Jan, and Steven Levithan. *Regular Expressions Cookbook.* Sebastopol, CA: O'Reilly Media, 2012.
Fitzgerald, Michael. *Introducing Regular Expressions.* Sebastopol, CA: O'Reilly Media, 2012.
Friedl, Jeffrey E. F. *Mastering Regular Expression.* Sebastopol, CA: O'Reilly Media, 2006.
Lutz, Mark. *Learning Python.* Sebastopol, CA: O'Reilly Media, 2013.
Lutz, Mark. *Programming Python.* Sebastopol, CA: O'Reilly Media, 2013.
Python Software Foundation. *Python 3.4.3 Documentation.* Last modified May 25, 2015. Available at: https://docs.python.org/3/tutorial/.
Richert, Willi, and Luis Pedro Coelho. *Building Machine Learning Systems with Python.* Birmingham, Packt Publishing: 2013.
Romanov, Maxim. "Python Functions for Arabic." *al-Raqmiyyāt: Digital Islamic History,* January 2, 2013. Available at: http://maximromanov.github.io/2013/01–02.html.