Python for Archivists: Breaking Down Barriers Between Systems

Gregory Wiedeman
SUNY Albany

Python for Archivists: Breaking Down Barriers Between Systems

Description

[Excerpt] Working with a multitude of digital tools is now a core part of an archivist's skillset. We work with collection management systems, digital asset management systems, public access systems, ticketing or request systems, local databases, general web applications, and systems built on smaller systems linked through application programming interfaces (APIs). Over the past years, more and more of these applications have evolved to meet a variety of archival processes. We no longer expect a single tool to solve all our needs and embraced the "separation of concerns" design principle that smaller, problem-specific and modular systems are more effective than large monolithic tools that try to do everything. All of this has made the lives of archivists easier and empowered us to make our collections more accessible to our users.

Yet, this landscape can be difficult to manage. How do we get all of these systems that rely on different software and use data in different ways to talk to one another in ways that help, rather than hinder, our day to day tasks? How do we develop workflows that span these different tools while performing complex processes that are still compliant with archival theory and standards? How costly is it to maintain these relationships over time as our workflows evolve and grow? How do we make all these new methods simple and easy to learn for new professionals and keep archives from being even more esoteric?

Keywords

Python, collection management, digital archives

Python for Archivists:
Breaking down barriers between systems

Gregory Wiedeman

SUNY Albany

Working with a multitude of digital tools is now a core part of an archivist's skillset. We work with collection management systems, digital asset management systems, public access systems, ticketing or request systems, local databases, general web applications, and systems built on smaller systems linked through application programming interfaces (APIs). Over the past years, more and more of these applications have evolved to meet a variety of archival processes. We no longer expect a single tool to solve all our needs and embraced the "separation of concerns" design principle that smaller, problem-specific and modular systems are more effective than large monolithic tools that try to do everything. All of this has made the lives of archivists easier and empowered us to make our collections more accessible to our users.

Yet, this landscape can be difficult to manage. How do we get all of these systems that rely on different software and use data in different ways to talk to one another in ways that help, rather than hinder, our day to day tasks? How do we develop workflows that span these different tools while performing complex processes that are still compliant with archival theory and standards? How costly is it to maintain these relationships over time as our workflows evolve and grow? How do we make all these new methods simple and easy to learn for new professionals and keep archives from being even more esoteric?

These problems are becoming more and more obvious as we implement more systems and the technology of our donors and users continually evolves. Donor organizations might have important records managed by web applications with APIs that need to be collected with custom scripts. Having trouble importing Encoded Archival Description (EAD) files into ArchivesSpace? Python can extract the problematic data into a CSV where it can be cleaned with OpenRefine, and then converted back into EAD or into ArchivesSpace itself through the API. Dislike the data entry system in ArchivesSpace? You can create box and folder lists with a spreadsheet, save it as a CSV, and then use a Python script to upload it with the API. Born-digital accessions often arrive on external hard drives. Python can be used to read a complex directory structure and export it to a CSV, and EAD file, or also directly into the ArchivesSpace API. This can also be done while moving and bagging files with the bagit-python library[1]. All of these problems and more can be easily combated—and even automated—with some basic Python knowledge.

These are the challenges for archivists of today and tomorrow. A big part of an archivist's job is now designing automated workflows that leverage many different tools. We now have a big toolkit of open and interoperable applications and much of our effort will be dedicated to reformatting, cleaning, evaluating, and moving data between all of these systems. Python is the perfect power tool for this problem. Its syntax is relatively simple and intuitive, it is extremely well-documented with a wide user base, and it has an enormous number of libraries optimized to solve just about every problem we have.

Python is a general-use programming language. That means it exists independently from any specific software and interacts with information purely as textual or binary data. Python does not know what ArchivesSpace is, or EAD, or Open Refine – and it does not care. It understands strings, integers, and floats, which allow it to easily read, write, and reformat data from all open systems. Python truly is a Swiss

Army knife for the software world, and Python scripting is a skill you can use in just about every situation.

While most librarians and archivists often choose more accessible tools, there is published work that shows the use of Python to solve a variety of problems in libraries and archives. Heidi Frank discuss how a Python-based workflow enabled them to extract MARC records from Archivist's Toolkit, and modify them for importing into the libraries' general catalog. She concludes by describing how it is "…hard to imagine a world of cataloging…" without the use of the methods she describes. "The increase in collaborations between catalogers and coders, as well as the increase in catalogers who *are* coders, is proving to be the next logical direction for the field of bibliographic description and access.[2] In another context, Jacob L. Nash and Jonathan Wheeler developed an automated workflow to automate the creation of Simple Archive Format (SAF) packages for batch upload into DSpace.[3] At the Rockefeller Archive Center, Amy Berish reflected on learning Python as a processing archivist, describing how, "It became obvious to me that having both traditional archival training and a toolbox of technical skills would be particularly useful in an age where workflows have the ability to be more efficient through automation.[4]"

This will be an introduction to using Python to move many of the types of data archivists work with between different formats and systems for technical archivists who are familiar with the basics of coding and are somewhat comfortable on the command line. This is not meant to be an introduction to programming or computer science or designed to be comprehensive in any way. I will not cover some of the basic fundamentals of coding, like data types (strings, floats, etc.), variables, or operators. If you do not have any experience coding, I would recommend learning web technologies first as their use of browsers seem to make them much easier for new coders to learn.

My goal is to make using Python more accessible to archivists by providing easy examples of how to manipulate and use the data we see every day. My hope is that even without understanding much of the language itself, beginner readers can run some code and get some results to understand some of the possibilities and challenges of working with Python. Readers should come away with an understanding that may either help them to better manage technical implementations generally, or some relevant hands-on tasks to experiment with as they continue on to learn from the many other great introductions to programming with Python.[5]

Python: Not so scary after all

Perhaps the most daunting challenge to using a high-level programming language like Python is not the intellectual difficulty of the code or the syntax itself, but the accessibility of the technology itself. Most of the work professional archivists and librarians do is more conceptually difficult than doing basic tasks with Python, but the broader technical complexity and freedom make it a daunting challenge. Not only does Python coding lack a graphical user interface (GUI), but there is no application at all – no structure or easy setup. In fact, getting a basic development environment set up, running a script, and reading and writing data to files might be the most difficult tasks I will cover here. Setting up Python on Windows can be particularly frustrating. Half the battle is setting up a basic environment that will run your code and get some results that enable you to get comfortable with it over time. After that, Python gets much less intimidating.

Another issue with starting to work with Python is whether to use Python 2 or Python 3. Essentially, the Python maintainers made some changes with Python 3 that are not backwards compatible, so they are still updating Python 2 as well. Code libraries took a long time to migrate but the important ones are finally Python 3-ready. Beginners should now clearly start with Python 3.[6] The only difference you will notice at first is that in Python 3, print requires parentheses. If you work with Python 2, the next thing

you'll likely notice is that your code will start blowing up when you work in non-ASCII text, since Python 2 assumes everything is in ASCII unless you tell it otherwise. There are ways to work with Unicode in Python 2, but they are confusing and not worthwhile when you consider Python 3 handles non-ASCII text much more effectively. While all of the example code will run with both Python 2 and 3, the article will assume you are working in Python 3.

Setting up Python

To work with Python, you need the Python software itself, a text editor, and a command-line shell. It is easy to get sidetracked and spend lots of time with different text editors and shells – both of which do not really matter for our purposes. Many Linux operating systems, like Ubuntu, come with both Python 2 and Python 3. Mac OSX should include Python 2 by default, and Windows does not come with either by default.

If you are running a Linux distribution like Ubuntu, just open Terminal and type *python3 –V*. This should list your Python version. If that does not work, try *python -V*, which should be Python 2. If you do not get these results, then you may need to download and install Python 3 manually, and then add it to your shell path[7]. On Mac OSX you should be able to run *python -V* in Terminal to get your version of Python 2, but you will need to download and run the Python 3 Mac OSX installer, which should also add Python 3 to your shell path[8]. The Python documentation should also provide some help in getting Python 3 running on OSX, but provides more detail than may be helpful.[9]

Setting up Python on Windows is a bit trickier. First open up a command line shell to see if you already have Python already installed. The default shell for Windows 7 is still cmd.exe, but modern systems have Windows Powershell which will also work. Updated versions of Windows 10 also have a newer default shell simply called Command Prompt that seems like a mix of cmd.exe, Powershell, and Bash. There is also

an experimental version of Bash that runs on newer Windows 10 machines[10]. All of these shells will work fine.

First, check if you already have Python by opening any command line shell and enter *python -V*. If you get an error, you will need to download and run the Windows installer for Python 3.[11] Now, the version of Python you install matters. If you have a modern 64-bit version of Windows you will have the option of installing either the 64-bit or the 32-bit (x86) version. You should probably choose the 64-bit version, but for our purposes it will not matter as long as you note which version you are running. All additional libraries you add will have to be compatible with this Python install, so just make sure you use the same versions as your Python install.

Another issue is selecting a version of Python that has a compatible lxml installer, which we will use to work with XML. Lxml is an add-on library for working with XML that has full XPATH support[12]. On Windows you have to install it manually because of some dependencies, and there is not always an installer for the latest version. Thus, you may have to install an older version of Python to easily install lxml. For example, at the time of writing the latest version of Python is 3.5.x, and lxml is up to 3.6.4[13]. The 3.6.4 version of lxml does not have an executable installer, only Python wheel files for pip which are considerable more difficult to work with. Lxml version 3.6.0 and older have the much-easier .exe installers, but are only compatible with Python 3.4.x and below[14]. The easiest way to get everything up and running at the time of writing is to install the older Python 3.4.x and use a matching installer for lxml 3.6.0. However, this may change in the future if .exe installers become available for the latest versions of lxml. As detailed above, be sure to use the 64-bit version of lxml if you chose that version of Python. The lxml installers list either win32 for 32-bit or win-amd64 for 64-bit in the installer filename.

Setting Python up to run from the command-line on Windows can be challenging for such a simple task. First, you need to find the directory where Python is installed. By default, Python 3.5.x and later installs itself in your *%APPDATA%* directory, which is typically *C:\Users\[USERNAME]\AppData\Local\Programs\Python*. This should also be the case for users of Python 3.4.x and below that do not have administrator rights. If you are installing 3.4.x or earlier and have administrator rights, Python is likely to install itself directly in your *C:* drive as *C:\Python34*.

Take a look at the directory where Python was installed, and you should have a folder named for the version of Python you downloaded. Mine is labeled Python34, but Python 3.5.x may install as Python35-32 or Python35-64, depending on the version you selected. In that directory you should find python.exe. Next, you need to right-click on your My Computer (This PC), and select Properties. On the left side you should see a link for "Advanced System Settings" which will open a new window.
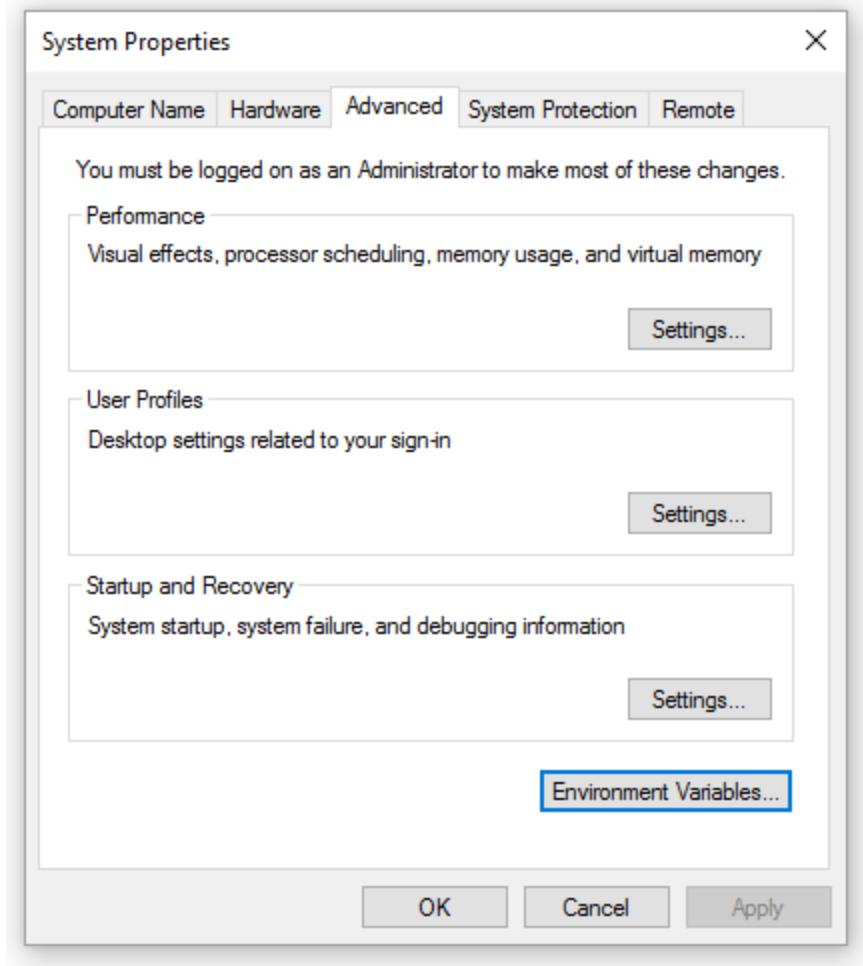
Figure 1: Finding the Environment Variables option on Windows 10

One the bottom there should be a button for Environment Variables. This will open yet another window. Under "System variables" at the bottom select "Path" in the list and click the Edit button.
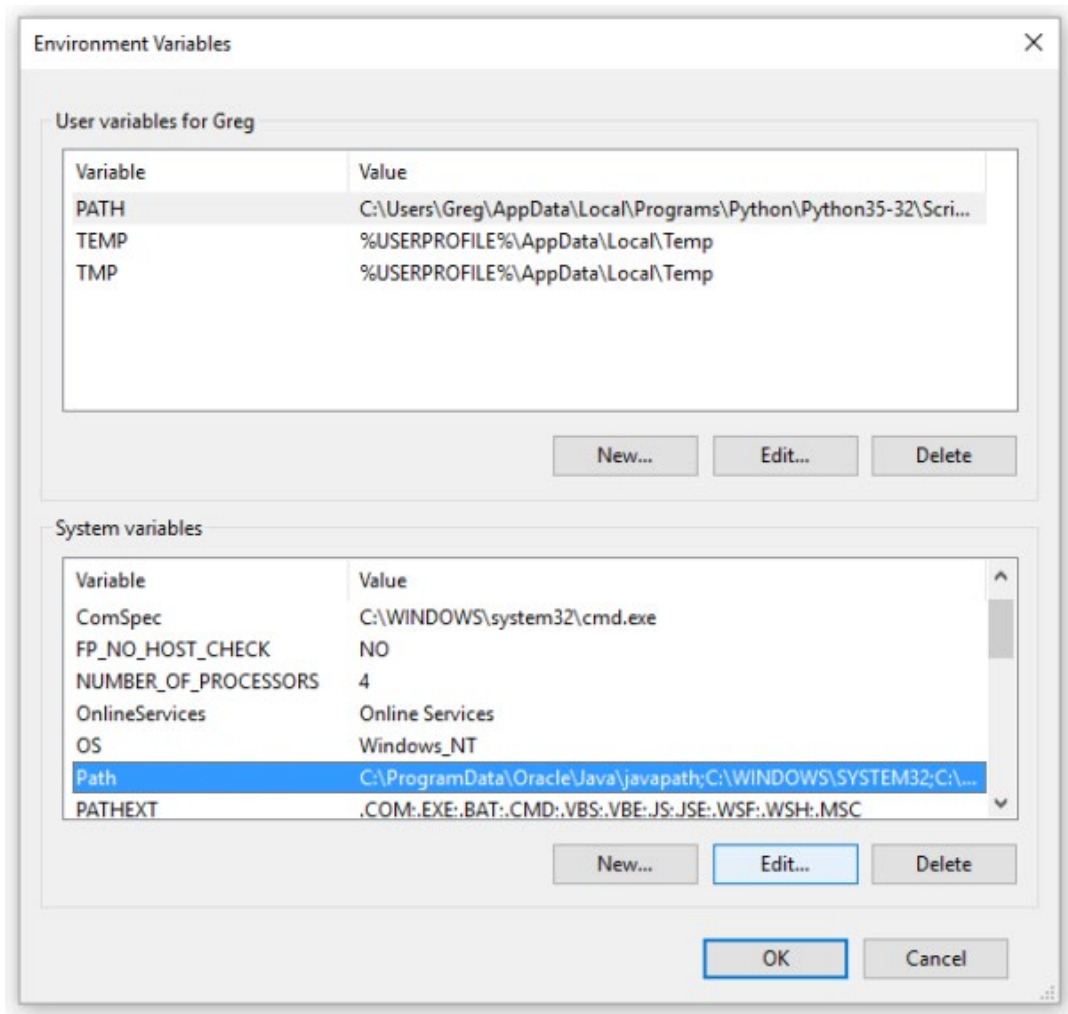
Figure 2: Finding the Path variable to edit on Windows 10

Windows 10 now has new window where you can click the "New" button to add a new path. This should be the path to the python.exe file you found above, typically something like *C:\Users\[USERNAME]\AppData\Local\Programs\Python\Python35-32* or *C:\Python34*. If you are working on an older version of Windows, you will see a text box with a long string of paths. Here you need to find the end without deleting anything, and add semicolon (;) and then the full directory path that holds python.exe.
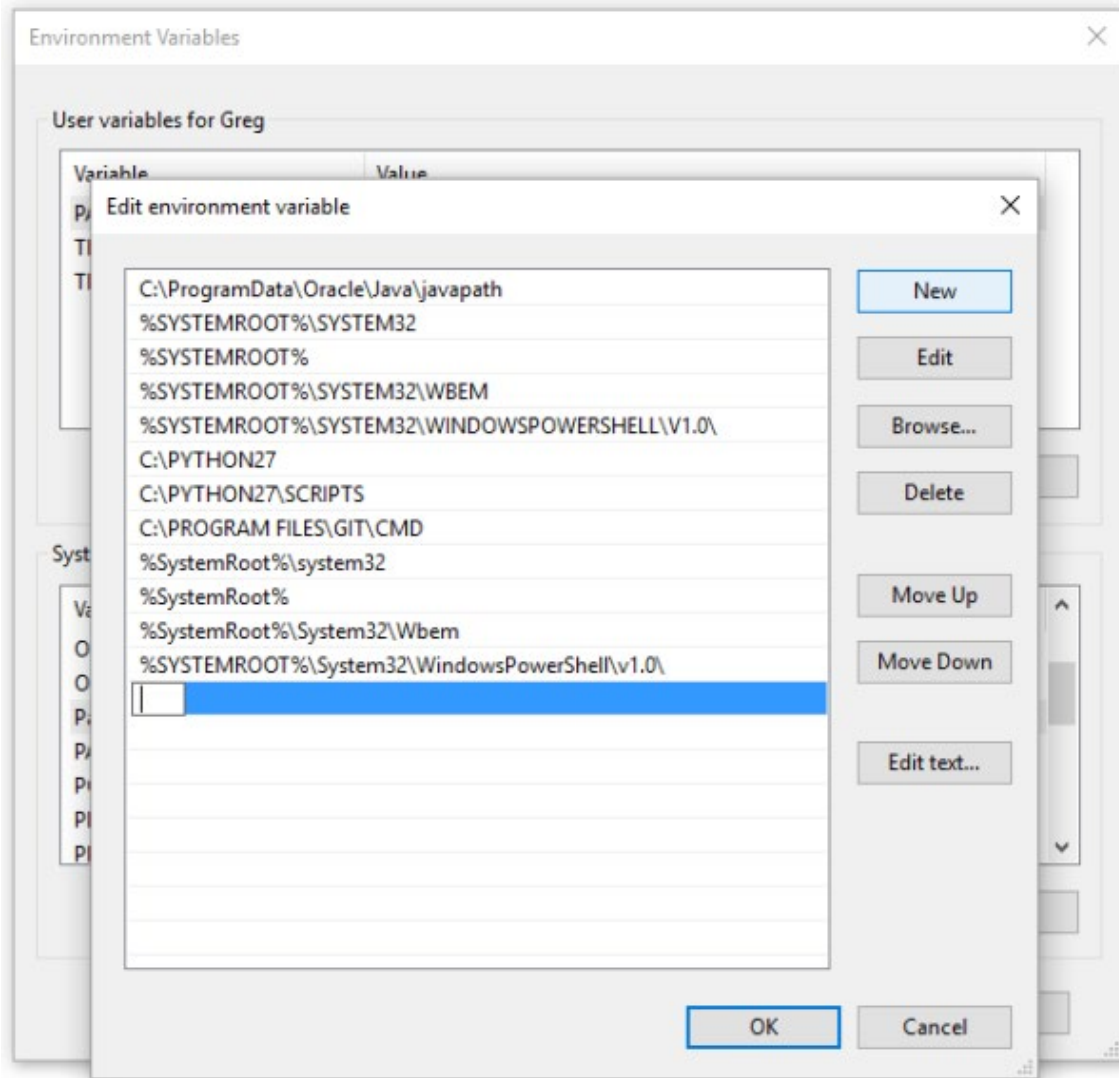
Figure 3: Adding the Path to python.exe on Windows 10
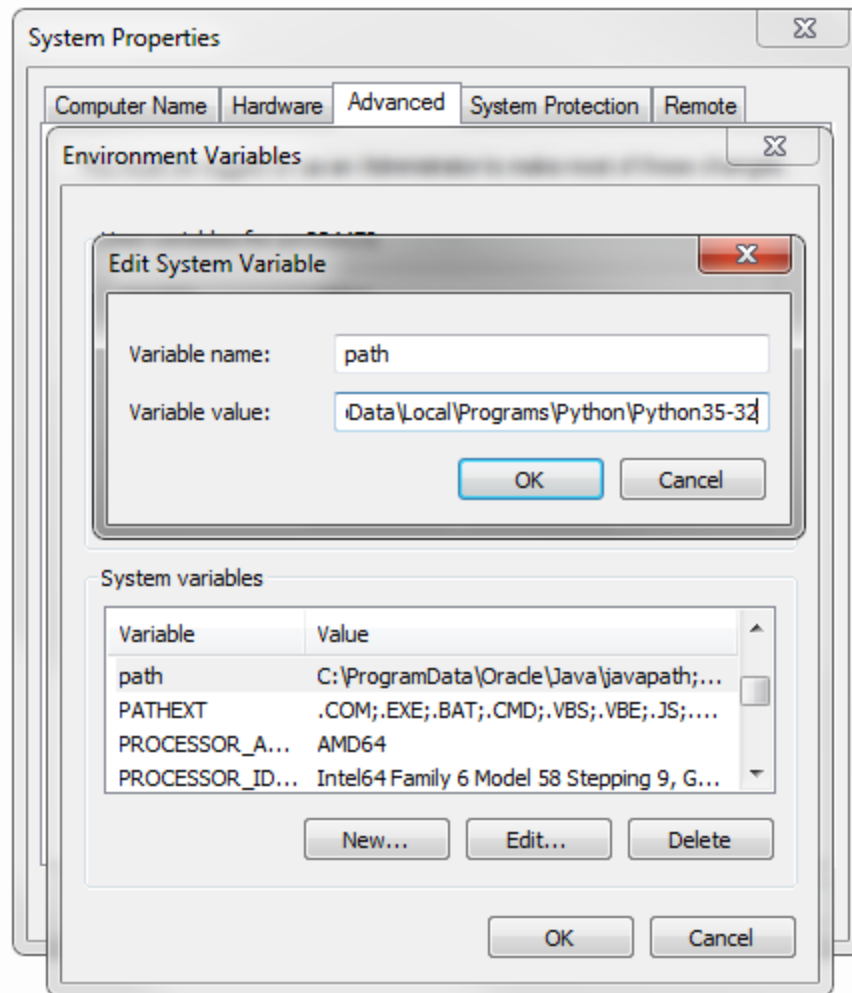
Figure 4: Adding the Path to python.exe on Windows 7

You should also add the Scripts directory which should be located in the same folder as python.exe. This could be *C:\Users\[USERNAME]\AppData\Local\Programs\Python\Python35-32\Scripts or C:\Python34\Scripts*. Now, close any command line shells you have open then reopen one. You should be able to type *python -V* and get your python version, and *pip -V* if you added the Scripts directory correctly.[15]

Once your Python environment is up and running, you can write and run some basic scripts. A Python script is just a text file. To create a script you just open up a text editor, create your Python script, and save it as a file with the .py extension, such

as *whatever.py*. Then you just open your command line shell and cd into the directory where the script is saved and enter the command *python whatever.py* and the script will run.

The introduction below will discuss a set of sample scripts and data that are available on Github. You can download these files from either from the command line if you are comfortable with git, or just by downloading them as a zip file from the Python4Archives Github page[16]. Then you can just cdinto the folder where you cloned or extracted the files, and run python *test.py* to get a result. Open the file in a text editor to look at the source code.

Basic Syntax

The key difference between Python code and most other languages is that whitespace counts. In fact, indents and new lines define different parts of the code. For example, in XML or HTML, using indentation to show children is a good idea and makes the code readable, but it does not really count as code. The computer will still know which a child is and which a parent is. This is not the case with Python. Each line and each indent matters just as much as any other character. Unlike other languages that ignore whitespace, Python does not have to use sets of brackets, parenthesis, or semicolons as much. This makes the code cleaner, less verbose, and simpler to the eye.

There are just a handful of basic statements that you will use for just about every script. This is really all there is to it:

```
#comments don't matter

print(something)

something = "A string of text"

for something in manyThings:

if something == anotherThing:
```

Comments are the simplest unit of Python. They exist to help you read and structure your code. Essentially Python ignores anything to the right of a pound sign (#) or everything between sets of three double quotes (""") for multi-line comments.

print simply displays whatever you put in the parenthesis on the command line. Just note that this is very different than writing your data as output. It is helpful to think of print as merely a tool to give you feedback on what the script is doing, not to produce the output itself. We will talk about producing output later.

The equals sign (=) is used to assign things like integers and text strings to variables. You can make the variable *something* mean some text and then print it. You can also add together strings using the plus (+) operator.

```
something = "A string of text"
anotherThing = something + ", and more text"
print(anotherThing)

> A string of text, and more text
```

You can also do math with variables and integers using operators (+, -, *, /):

```
something = 5
print(something + 15)


> 20
```

You can experiment with variables by running the sample script equals.py from the Github page. Open the file in a text editor and change some of the text between the quotes. You can then save it and just run *python equals.py* in the command line from the directory where you extracted or cloned the sample scripts. If you get an error you can just use the undo option in text editors like Notepad++ or GEdit to go back to something that works, save the file, and try again. Experiment and see what results you get. While crude, this can be a very effective way to learn code.

for is a loop that runs through a group of things in turn so you can perform the same actions on each individual thing. You can loop through lines in a CSV file or a table, lines or characters in text, items in a list, children of a parent in XML, or files in a directory. If manyThings is a group of things, you can write:

```
4| for number in manyThings:
5|    print(number)
```

Think of it as: "for each number in manyThings." Then, you can do perform actions on the number variable that will be performed by everything in the manyThings variable in turn. Take a look at for.py in the sample scripts to see how this works. Notice the indents. The indented code underneath the for loop tells the computer what to run during the loop. Code that is not indented will then be run after the loop has fully completed.

A list is a good thing to start experimenting with loops. This is any set of things, like numbers or strings of text in quotes, separated by commas between brackets.

```
14| listOfIntegers = [5, 12324, 23, 657, 177]
15| listOfStrings = ["a thing", "another thing", "a 3rd thing"]
16|
17| listOfStrings.append(10)
18|
19| print(listOfStrings)


> ["a thing", "another thing", "a 3rd thing", 10]
```

Lists can also contain a mix of both strings and integers, and they can also contain other lists, so you can have lists of lists of lists. To add things to list you can use the .append() function like above.

```
27| listOfLists = [[01, 02, 03], ["January", "February", "March"]]
```

Reading and Writing Text to Files

Another major hurdle to working with Python is reading and writing data with serialized text formats like XML, JSON, CSV, or just plain text files. First we have to point our script to a text file we want to read. The simplest way to do this is to just enter a path into our script as a string. Windows and Unix paths are a bit different, and this is complicated further by Windows's use of backslashes (\) which are special characters in Python used to escape other characters. In practice, this just means that we have to use two backslashes for Windows paths instead of one. First, assign the explicit path to the

sample scripts as a variable. Here are two examples of what your path could be in Windows and then Unix:

```
2| windowsPath = "C:\\Users\\[USERNAME]\\python4archivists\\example.txt"
3| unixPath = "/home/[USERNAME]/python4archivist/example.txt"
```

Here is also an example of a Windows UNC path for a Windows server that is not mounted:

```
uncPath = "\\\\myserver\directory\\python4archivist\\example.txt"
```

Once this is saved correctly as a string, you can read the text from the file into a string like this:

```
fileObject = open(windowsPath, "r")
#do stuff to fileObject
fileObject.close()
```

The "r" is the mode you are opening the file in, this tells the computer that we are just reading from the file and not modifying it. We can now do things like *print()* or loop though file with *for*. Closing the file with *close()* after you are done with it is always a good idea. Once you have a file object, you can turn it into a string of text with *.read(),* and work with it as you would any other string. In the example readWrite.py script on Github, you can experiment with adding new text to files. In this script, we change the *fileObject* variable into a text string called *textString* with the *read()* function. On line 8 we print the string and then add an additional line to it on line 11. Remember that we have to use two backslashes in Windows paths? That is because a backslash is used to denote additional characters. Here \n means a carriage return or new line.

```
 5| fileObject = open(myPath, "r")
 6|
 7| textString = fileObject.read()
 8| print(textString)
 9|

10| #make the textString variable equal the same string plus more text
11| textString = textString + "\nAdd another line here!"
12|
13| #print the new string
14| print(textString)
15|
16| #remember to close the file
17| fileObject.close()
```

To run this script you have to open readWrite.py in a text editor and change the path to where you downloaded the example.txt file. You can edit the Windows path on line 2 or the Mac or Linux friendly one on line 3. Just remember to change the variable on line 5 if you use *unixPath*. Then you can just cd into the correct directory and run the command python readWrite.py. You can experiment by adding additional text to the *textString* variable on line 11 before printing it.

If you open the example.txt file after running the readWrite.py script, you will notice that it remains the same even though you modified the text in the script and saw the additions printed to the command line. This is because python loaded the text from the file into memory, and even though you can modify it and print it, this data only stays in memory while the script itself is running. Printing to the command line is very different from writing data back to a file on your hard drive. Writing text to a file is a type of serialization, this means the data is written back to the file system for storage. If

you never write back to a file explicitly, you can work with important data—like a directory of EAD files—without unknowingly modifying that data.

You may want to write the new text you modified back to a file. You can do this by removing the two sets of double quotes (""") on lines 19 and 24 in readWrite.py. This loads the path to the existing or new file you want to write to again with *open()*, but uses "w" for write as the mode instead of "r". This will overwrite any existing file or create one if none exists. If you want to add text to the end of an existing file there is another mode, "a," for append[17]. Finally, you must write the text string to the file object using *.write()* before closing it.

```
20| saveDir = "newFile.txt"
21| file2 = open(saveDir, "w")
22| file2.write(textString)
23| file2.close()
```

Remove the triple-quotes and run *python readWrite.py* to write these changes back to the file. Open *newFile.txt* in a text editor to see what happened.

Reading and writing textual metadata to and from files can be an important skill for archivists. While we should not have to write or develop software to manage our data, it is becoming more and more likely that archivists will play a major role in managing the movement of data between different systems – particularly though migrations. These processes can involve temporarily writing data to files while making alterations or joining data from multiple systems. Working with serialized data can be an important skill to speed up these complex challenges, and give archivists a more hands-on role, rather than sending this work to a technologists who may not fully understand the eccentricities of archives metadata.

Working with XML

XML is very familiar to archivists because of the proliferation of EAD, so it may be a comfortable format to start working with data in Python – particularly for those who are familiar with XPATH. ElementTree is the default Python library for working with XML, but lxml is much superior. The syntax of ElementTree is essentially the same as lxml, but it lacks things like full XPATH support and handy functions like .getparent(), so if you work with XML, you will find that it is worth the process of installing the additional lxml library.[18]

Installing lxml on OSX and most Linux machines should be easy, and it may even be installed by default. Newer versions of Python come with pip, Python's own package manager for downloading and installing libraries. You just may need to run the command *pip install lxml*.

On Windows, installing can be more difficult as it requires some dependencies that do not install when you use pip. The easiest way is to download and run the executable installer that is compatible for your version of Python[19]. As of writing the latest installers were for lxml 3.6.0 which only works with Python 3.4.x and below. Make sure you choose the correct 64-bit or 32-bit version that matches your Python install. Once you download and run the installer you should have lxml up and running.

For all operating systems, the simplest way to check if lxml is installed correctly is to enter *python* in the command line. This will give you a Python prompt where you can enter *import lxml*. If you do not get an error, lxml should be installed and working fine[20]. Run *exit()* to get out of the Python prompt and back to the basic command line.

After you have lxml installed, you can import the library to script by stating *from lxml import etree as ET* at the top. If you have trouble installing lxml, you can use ElementTree to start out. Just use *import xml.etree.ElementTree as ET* as the import

line. Once the library is imported, you need to parse the XML file to an object that Python can understand using *ET.parse(xmlPath)*. Finally, you need to get the root element as a variable by using *.getroot().*

```
1| from lxml import etree as ET
2|
3| #enter a path to the XML file here
4| windowsPath = "C:\\Users\\[USERNAME]\\python4archivists\\apap159.xml"
5| unixPath = "/home/[USERNAME]/python4archivists/example.txt"
6| xmlObject = ET.parse(windowsPath)
7|
8| #root is the top-level  tag
9| root = xmlObject.getroot()
```

Now that you have assigned the root element to a variable, there are a basic set of methods you can use to get information for every element – from the root down to the smallest child.

- *root.find()* will navigate down the document tree to find children
- *root.xpath()* will return a matching element using an XPATH query as a sting
- *root.tag* will show the element's name
- *root.text* will show any text contained by the element
- *root.attrib* will provide an dictionary of an element's attributes
    - To get an attribute, use *root.attib["attributeName"]*
- *root.tail* will show any text after the element for mixed content

We can now navigate and read any XML document in complex ways using just these few methods along with Python's basic *for* and *if* statements. There is a working example of reading a sample EAD file in the sample scripts called editXML.py. Here we can load the root element as an XML Object like we did above, and print its tag name and ID attribute using *root.tag* and *root.attrib["id"]*. In the next block of code, we find

the collection-level element and a assign its .text to a variable called *collectionName* and then print it on the next line.

```
13| #lets find out which collection this is
14| collectionName = root.find("archdesc/did/unittitle").text
15| print("We are working with " + collectionName)
```

Next, starting on line 19, we can loop though each child of /, and assign each child element to the variable *series*. Now, in EAD not all children of must be series components, so indented within the *for* loop, we use an *if* statement on line 21 to make sure that everything that is indented underneath the if statement is limited to elements that have a tag name equal to "c01." Notice that we use a double equal sign (==) as the operator here. This is because a single equal sign is used for assigning variables while a double equal sign is used to compare two things. Finally we use the *.find()* method to print the *.text* of the for each *series*.

```
19| for series in root.find("archdesc/dsc"):
20|  if series.tag == "c01":
21|   print(series.find("did/unittitle").text)
```

While we are still looping through each series, we can change anything in the XML file itself. As long as we stay at the same indentation level under the if statement, we can keep acting upon the *series* variable. On line 26 we use the *.set()* method to change the "level" attribute to "series." This will either change the existing series attribute, or add a new one.

```
26|  series.set("level", "series")
```

Perhaps we want to add an additional element for each series to state that the series is located offsite. In the same loop we have been working in, we can assign a new to a variable by using *ET.Element("physloc")*. On line 30 we add a string of text to that element using *.text*. Finally, we have to place that element as a child *of* , so we use *.find()* to assign it to the variable did, and then use the *.append()* method to place the element as its last child.

```
29|  newElement = ET.Element("physloc")
30|  newElement.text = "This series is located in..."
31|  did = series.find("did")
32|  did.append(newElement)
```

By returning to the outer indent level in line 34 we are telling the computer that we want the next set of code to execute after our *for* loop has finished and completed everything we discussed so far. In this block of code we loop though each series again, while also limiting our actions to "c01" elements with an *if* statement. On line 39 we find the text of each series element and print it to the command line within other text strings to make the output readable. Next we use the comparison operator (==) check to see if we can find an element with for each series on line 39, and if that element has text on line 41. Finally we use *print()* to give us feedback on the command line.

```
36| for series in root.find("archdesc/dsc"):
37|    if series.tag == "c01":
38|   print("\nDoes " + series.find("did/unittitle").text + " have a ?")
39|   if series.find("did/physloc") == None:
40|     print("No! its does not, we made a mistake...")
41|   elif series.find("did/physloc").text == None:
42|     print("Its there, but it doesn't have any text.")
```

```
43|   else:
44|     print("Yes it does!")
```

After running editXML.py, you will again notice that, although we modified our root element while the script was running, the apap159.xml file remains unchanged because we never wrote the new data back to the hard disk. To do this, we first need to convert our root XML object back to a string using *ET.tostring()*. On line 49 we assign the string to a variable called *xmlString* using the *root* XML object, and some other options that will place an XML declaration on the top of the file.

```
49| xmlString = ET.tostring(root, pretty_print=True, xml_declaration=True,
encoding="utf-8", doctype="")
```

Finally, to write the new data to a file we use the same code we used in readWrite.py to write the text to a new XML file. Remove the two sets of triple quotes (""") in editXML.py, and then save and run the script again to see how this works.

A large percentage of archives metadata is still stored in the EAD format as XML files. Python gives us an easy way to edit and manipulate this data at scale. This could mean creating, editing, or reviewing EAD files, or converting EAD data to another format. EAD is likely to stick around in archives for a while, even as more effective tools become available, so being able to automate the management of XML files at scale is a key skill for archivists.

Working with JSON

JavaScript Object Notation (JSON) has moved past XML to become the most common format for serialized data on the web. Archivists are becoming more and more likely to work with JSON, and Python provides an easy way to manage data in JSON and

move data between JSON and other formats and systems with the json library, which is included in Python's default installation.

Remember when we talked about lists while introducing loops in the Basic Syntax section? Another useful data type in Python is dictionaries. Dictionaries are essentially a list of key-value pairs that you can loop through with *for* and get values by calling the appropriate key. Dictionaries are held in curly brackets ({}), and key-value pairs are listed with commas (just like a list), and separated by a colon (:). In the sample scripts, dict.py shows a basic dictionary on line 4.

```
3| #a simple dictionary
4| dict = {"key1": "value", "key2": 1}
```

Dictionaries are very useful for archivists, as they are ideal for holding data with description, and key value pairs will be familiar to all archivists who work with metadata. The dict.py sample script provides an example of a useful calendar dictionary on line 7. You can get values from a dictionary by calling keys with brackets. Here we get the value of the "04" key from the calendar dictionary just like in line 10.

```
6| #a useful dictionary
7| calendar = {"01": "January", "02": "February", "03": "March", "04": "April", "05":
"May", "06": "June", "07": "July", "08": "August", "09": "September", "10":
"October", "11": "November", "12": "December"}
8|
9| #print the value for the key "04"
10| print (calendar["04"])

 > April
```

Next, we see that both list and dictionaries can be nested within one another. You can have a list of dictionaries, or dictionaries that contain lists as values. On lines 12-17 we set up a set of dictionaries that contain the keys: title, dates, and extent. The first two keys contain strings and the extent key contains a list with both an integer and a string. Finally, we add all of these dictionaries to a single list called *fonds* on line 20.

```
12| #both lists and dictionaries can be nested
13| collection = {"title": "Hugo A. Bedau Papers", "dates": "1957-2003 ", "extent":
[36, "cubic ft."]}
14| series1 = {"title": "Scholarship and Teaching", "dates": "1958-2002", "extent":
[6, "cubic ft."]}
15| series2 = {"title": "Correspondence", "dates": "1957-2003", "extent": [4, "cubic
ft."]}
16| series3 = {"title": "Advocacy Organization", "dates": "1958-2003", "extent":
[10, "cubic ft."]}
17| series4 = {"title": "Subject Files", "dates": "1955-2003", "extent": [16.79, "cubic
ft."]}
18|
19| #here is a list of the dictionaries above
20| fonds = [collection, series1, series2, series3, series4]
```

We can loop though this list with *for* and print values by calling the relevant keys. On lines 22-23 we print the title and dates keys for each dictionary in the *fonds* list. On lines 26-27 we loop through the extent lists and print each item after turning it into a string with *str()*.

```
22| for resource in fonds:
23|    print (resource["title"] + ", " + resource["dates"])
24|
```

```
25|  #the value of extent is a list
26|    for extent in resource["extent"]:
27|   print ("  " + str(extent))
```

If you are wondering why we are experimenting with lists and dictionaries instead of JSON, the reason for this is that Python's json library reads a JSON file and essentially turns it into sets of nested lists and dictionaries. You can work with JSON just like we did above. Open the sample script editJSON.py in a text editor for an example of how to work with JSON data in just this way. You can see we imported the json library on line 1, and we also imported a function on line 2 that we will use later. Change the path to the sample scripts, where there should be a sample json file called resource.json. On lines 9-11 we load the file and use the json library's *.load()* function to read the data as sets of lists and dictionaries.

```
9| jsonFile = open(windowsPath, "r")
10| jsonData = json.load(jsonFile)
11| jsonFile.close()
```

This is a fairly large and complex JSON file, so on lines 13-16 we first loop though the *jsonData* object and print all the keys so we can see what kinds of data we have. Run the script with *python editJSON.py* to take a look at the keys.

```
13| #lets take a look at what keys are in this file
14| print ("Here are the keys:")
15| for key in jsonData:
16|   print ("  " + key)
17|
18| #looks like there is a "level" key and a "title" key
19| print ("\n\n" + jsonData["level"] + ": "+ jsonData["title"])
```

```
20|
21| #there is also a notes key too, remove the # below to uncomment the line
22| #print (jsonData["notes"])
```

There is a "level" and a "title" key, so we print those values on line 19, and we find that this is JSON data for the collection, Environmental Advocates of New York Records. A good amount of descriptive metadata is under the "notes" key. You can uncomment line 22 by removing the # from the start of the line, and when you run the script again a large amount of data is printed to the command line. Most of this is hard to read in this format, so if you comment or erase line 22, and uncomment lines 24-30 by removing the triple quotes (""") on each end, we can loop though the notes to read only the data we are looking for. On line 26 we loop though the notes and print the "type" key on line 27. Within this same loop we use an if statement to limit notes that have a type that equals the string "abstract." Finally we print the "content" key for this note. As you can see if you run the script again, the content key also contains a list of a single string with the collection's abstract.

```
25| print ("\nHere are the notes: ")
26| for note in jsonData["notes"]:
27|    print (note["type"])
28|  if note["type"] == "abstract":
29|   print (note["content"])
```

We can also modify JSON data just like any other type of serialized data. If you uncomment lines 34-46, you can see how we can loop though the "dates" key and get the "begin" date and "end" date. Here we are also using the handy .split() method to get only the characters in the timestamp before the "T" character. On line 38 we combine the "begin" and "end" dates with a slash (/) in between to get the normalized ISO date format archivists have commonly used in EAD XML files. On line 39 we use that magic

function we imported on line 2 called *iso2DACS()* to reformat the date range in the common DACS display date format, and assign that new date to the "expression" key. Outside the loop on line 40 we call that "expression" key from the 0th item in the "dates" list and print it. Finally, on lines 43-45, we use the json library's *.dump()* method to turn our dictionaries and lists back into a big string, and write it to a new file called newResource .json. If you run the script again without these lines commented, it will show the reformatted date and create the new output file.

```
35| for date in jsonData["dates"]:
36|    beginDate = date["begin"].split("T")[0]
37|  endDate = date["end"].split("T")[0]
38|  isoDate = beginDate + "/" + endDate
39|  date["expression"] = iso2DACS(isoDate)
40| print (jsonData["dates"][0]["expression"])
41|
42| #write the data back to a JSON file
43| output = open("newResource.json", "w")
44| json.dump(jsonData, output)
45| output.close()
```

Working with CSV

Comma-separated values are a very simple way to serialize tabular data that relies solely on text encoding for its structure. Although it has some flaws, the independence of CSV is useful for moving data between systems. Also, since CSV is a tabular format, it can be a good way to make data human-readable by just opening it in a spreadsheet. Many archives workflows involve viewing data in a spreadsheet for editing or cleanup, making CSV a core tool in an archivist's toolkit.

Python readily works with CSV through its default csv library. Unlike lxml, this library is included with Python and does not require an additional installation. All you have to do is state *import csv* on the top of your script and you can easily manipulate CSV files.

Remember when we discussed lists in the syntax section? Python essentially views CSV files as a list of lists. There is an example called editCSV.py in the Github sample scripts that introduces manipulating CSV files with Python, and some sample data called executedJuveniles.csv. We have to *open()* a CSV file from a path just like other file formats, but then we can easily use *csv.reader()* on the open file object to turn the data into a list of lists on line 8. Then we can use our familiar for statement to loop though the *csvObject* and act upon each row. In this example, each row variable is a list of each field that we can access by an index number – you just have to remember that the index starts at 0 instead of 1. So by stating *row[2]*, we are calling the third column of each row and printing it to the command line on line 10. If we run the editCSV.py script, we should see a long list of names that is from the third column of the example CSV file. Remember to edit the path to the CSV file on lines 4 or 5 to point to the directory where you cloned or downloaded the scripts.

```
7| csvFile = open(windowsPath, "r")
8| csvObject = csv.reader(csvFile)
9|
10| #loop though the CSV file
11| for row in csvObject:
12|  print(row[2])
```

Next we can take the data we have in that same CSV file, change it, and write it to a different file. If we remove the two sets of three quotes (""") on lines 16 and 46 that change all the rest of the code to a comment, we can see how this works. First, we have

to create some new variables. *newList* is an empty list that we will write to our new CSV file after we add rows to it. *rowCount* is an integer that will let us know which row of the CSV we are on during our loop. On lines 22 and 23 we open and read our original CSV file again just like we discussed above (remember to change the path variable on line 22 for Unix paths). Then, on line 25 we begin our for loop that lets us read each row in turn. If you take a look at the sample CSV file, you will notice that the first row is a header that describes the content of each column. We want to omit this row, which we can do by counting through our loop. Line 27 takes our *rowCount* variable and adds 1 to it. Since it starts at 0, the *rowCount* variable will equal 1 for the first row. For subsequent rows, this line will continue to add 1 to the variable, so for the second row, *rowCount* will be 2, 3 for the third, etc. This lets us know what row we are on during our loop.

```
17| #set and empty list and initialize the count at 0
18| newList = []
19| rowCount = 0
20|
21| #open the original CSV again and read it
22| csvFile = open(windowsPath, "r")
23| csvObject = csv.reader(csvFile)
24|
25| for row in csvObject:
26|   #count the number of times though the loop
27|   rowCount = rowCount + 1
```

We can skip the first row by using a simple *if* statement on line 29. Now, the code block indented underneath it will only be executed for rows 2 and above, omitting the header row. Next, on line 31 we make a new list called *rowList* that will represent each row in our new CSV file that we will make. In our new file we will make three columns: a count of each row, the names from each row in our original CSV, and a DACS-

compatible date. We do not want our count to start at two, so we will subtract one from our rowCount for the first column. The second column is easy: we will just use the index of the original rows to get the names in the third of the original CSV. Next, we use the *iso2DACS()* function that we imported on line 2 to convert the forth column to the new date format.

Now, for each row of our original CSV file we have a new list of our reformatted data. During this same loop, we can *.append()* this list to our *newList* variable which will become a big list of these smaller lists.

```
30|   #make a list of the rowCount, the 3rd column, and the 4th column using the
iso2DACS function
31|   rowList = [rowCount - 1, row[2], DACS(row[3])]
32|   #append that list to the newList - its just a list of lists
33|   newList.append(rowList)
```

Finally, we have to write this big list of lists to a new CSV file. We can close the original CSV file on line 36, as we have all the data we need now in our *newList* variable. Remember to write a correct path to our new file on lines 39 for Windows or move the comment from line 40 to line 39 to use the Unix path. This is where we will write this data back to the file system. We can then use *open()* to open our new file in write mode, but also add *newline="* to format our CSV correctly. Python's CSV library has a simple *.writerows()* method that lets us format a list of lists as a CSV file. After that, we just have to close the new file on line 45 and Python will create a new CSV file. Just be sure to remove the two sets of three quotes (""") and enter a correct path before running editCSV.py again, and you should have a new CSV file called csvDACS.csv. Open it up in a spreadsheet program or text editor to see what it looks like.

```
39| newCSV = "C:\\Users\\[USERNAME]\\python4archivists\\csvDACS.csv"
40| #newCSV = "/home/[USERNAME]/python4archivists/csvDACS.csv"
41|
42| newFile = open(newCSV, "wb", newline='')
43| newCSV = csv.writer(newFile, delimiter=',')
44| newCSV.writerows(newList)
45| newFile.close()
```

Currently, we plan to use Python to manage the migration of accession data from our old CMS database into ArchivesSpace. Our database developer will export the important data from the old database as CSV files, but since the fields in the old system were problematic, they have to be mapped onto the correct ArchivesSpace fields. Over the years, the old fields were used inconsistently, and even the same fields will need to be mapped differently, depending on how they were used. It is much more efficient for archivists to manage this process from CSV files, rather than meeting multiple times with technologists who would need to understand unfamiliar archival practices. In migrations like these, the intellectual and technical challenges are so intertwined, that separating them among multiple staff members can often result in miscommunication and drawn-out processes.

Working with File Systems

So far we have worked with serialized text formats like XML, JSON, CSV, and plain text, but one of the major benefits to using Python on a desktop is that you can read and write to the file system directly. This means Python can read and write directories and files right on your hard drive using the default os library, all you need to do is state *import os* on the top of your Python script.

The fileSystem.py file provides a brief introduction to the kinds of things we can do with Python's os library. You can see we imported the os library on line 1, and you

can ignore lines 3-5 for now, as they just define a date function we will use later. Lines 7 and 8 demonstrate how we can use Python to work with all types of operating systems. The *os.name* method lets us know what type of system Python is running on – "nt" for Windows, and "posix" for a Unix-based system like Mac OSX or Linux. If you run fileSystem.py, you should see that Python scripts can run on any number of platforms. The script should let you know what operating system you are running.

Remember how we wrote in paths manually in other scripts? Well, Python's os library can do that for us. On line 10 we get the path to where this script is saved. This should work on almost all platforms. You can see this path by printing it on line 11. Next, we can loop through that directory using the *os.listdir()* method on line 17 which creates a list of everything in the directory. We could just print each item, but there are probably both files and folders in this directory, so we can count each type with our two variables from lines 14 and 15. Python has two methods that identify files and folders, *os.path.isfile()* and *os.path.isdir()*. We can use these to add 1 to the count for each type and then print our final count outside the loop on line 26. This will tell us how many files and how many folders are in the directory.

```
13| #start our counting variables
14| folderCount = 0
15| fileCount = 0
16| #loop though everything in a directory with os.listdir()
17| for item in os.listdir(directory):
18|
19|  #count each folder and file
20|  if os.path.isdir(item):
21|    folderCount = folderCount + 1
22|  elif os.path.isfile(item):
23|    fileCount = fileCount + 1
24|
```

```
25| #print what we found to the command line
26| print("Found " + str(folderCount) + " folders and " + str(fileCount) + " files
here")
```

Note that we used another method while printing these results to the command line. Our original variables were integers so we could add and subtract from them, but you cannot combine integers and strings together, so we used *str()* to change our final integers in to a text string of the same number – say the string "13", rather than the integer 13. We could do the opposite using *int().*

We can also use *os.listdir()* with what we learned about working with XML and CSV files above. In many cases it could be helpful to loop though a large number of EAD XML files to evaluate them or reformat some of the data. We can just use *os.listdir()* on the directory to make a list of all the files and then write a for loop that acts upon each of them using the lxml library within that one big loop. If you have a large directory of XML files, you could easily use a python script with the os library and lxml to extract all of the extents.

Python's os library allows us to fully manipulate and extract information from the ordinary files and folders on our hard drive. If you remove the two sets of three quotes (""") on lines 28 and 54, we can see how to make folders and get details on each file from filesystems. In the example scripts from Github there is an example directory called "exampleDir" that contains some subdirectories and small text files. We can loop through that folder using *os.listdir()* and find "exampleDir" using an if statement in line 31.

We can make a new directory here using Python's *os.mkdir()* method, but first we need a complete path for this new folder. Remember how paths are different in Windows and Unix-based systems? The os library accounts for this using

the *os.path.join()* method. We can combine paths by just listing them separated by commas and Python will make sure they will work on all operating systems. Here, we combine *directory* (the original path from line 10), item ("exampleDir" from line 30 and 31), and our new folder as the string "WeMadeThis." If we try to create this directory, and it already exists, the script will return an error, so we can use a simple *if not* statement and the *os.path.exists()*method on line 35 to only call *os.mkdir()* on the new path if it does not already exist. After we run fileSystem.py, you should see the new folder appear.

```
33|   #combine the original directory with this item and "WeMadeThis" into a path
34|   newFolderPath = os.path.join(directory, item, "WeMadeThis")
35|   #check if this new path exists and then make it
36|   if not os.path.exists():
37|     os.mkdir(newFolderPath)
```

There are many tools in the os library that are useful to archivists. One of the most powerful is *os.walk()* which can be used to navigate a large directory tree and provide a list of folders and a list of files that it finds throughout. Unlike a typical for loop, *os.walk()* assigns three different variables that you can see on line 40. Here, *rootDir* is a string for the explicit path. We can use this later in the loop to get the full path of files deep within the directory structure. The second variable is a list of subfolders we assign to *folders* and the third is a list of all the files from all lower levels of the directory tree that we assign to *files*. This gives us the root path, a list of all folders from all child levels, and a list of all files from all child levels.

```
39|   #walk the directory to get lists of files and folders
40|   for rootDir, folders, files in os.walk(item):
41|     #loop through the list of files
42|     for file in files:
```

```
43|     #print the file name and full path
44|     print("filename: " + file)
45|     fullPath = os.path.join(rootDir, file)
46|     print("  path: " + fullPath)
```

On line 42 we loop though the file list, so we can act upon each one of these disparate files in turn. First, we print the filename (which is *file*) on line 44, then we use *os.path.join()* to combine *rootDir* and *file* to get the full explicit path to that file and print that as well.

Now that we have a full path for each file, we can use some of the additional os.path methods to get information on each file. *os.path.getsize()* provides us with the file's size in bytes. We can do some basic math to make the size more readable for larger files, but we omitted this to keep things simple. On line 49 we get the size of each file, turn it into a text string, and print it to the command line.

```
49|    print("  size: " + str(os.path.getsize(fullPath)))
```

The os library also includes a set of methods to read the timestamps that are stored in the file system for each file. Different operating systems store these timestamps differently and consider creation time to be different, but Python provides a cross-platform way to get this important information. These methods are *os.path.getmtime()* for last modified, *os.path.getatime()* for last accessed, and *os.path.getctime()* for creation date. On lines 51-53 we use these methods and the small *humanTime()* function we defined on lines 4 and 5, to print human-readable versions of these timestamps to the command line.

```
51|     print("  Last Modified: " + humanTime(os.path.getmtime(fullPath)))
52|     print("  Last Accessed: " + humanTime(os.path.getatime(fullPath)))
53|     print("  Created: " + humanTime(os.path.getatime(fullPath)))
```

Python's ability to easily work with the filesystems of a number of different operating systems archivists work with makes it very useful for born-digital material. Files systems are not designed to store information about records for the long-term, and the ability to get important provenance information on files and move that information to more stable formats can be extremely valuable. Even though the files from Github serve only as a basic example, you can probably learn something about them by running fileSystem.py and looking at the timestamps the script shows. When were these files created? What can you tell about how Github handles metadata for files?

Working with APIs

As I have demonstrated above, once you get a basic environment set up and understand the basics of writing scripts, Python is a powerful tool for editing, cleaning, transforming, and migrating different formats of serialized data and file systems. Yet, the real strength of Python is its ability to work both with serialized data and large systems through APIs. The most obvious benefit for archivists is to read and write data to ArchivesSpace at scale using its backend HTTP API. This tool is one of the biggest benefits to ArchivesSpace. It is a large, complex, and stable system for archival data that is governed by a strict data model that also allows non-technical archivists to manage data through a web browser. Through the API, we also have the granular control to make large scale, iterative changes to our data just like we can with EAD data in XML files. Unlike working in XML, the ArchivesSpace API will also prevent us from making damaging changes to our system of record, and will return errors instead.

Here we can easily take data from XML, JSON, CSV, and file systems to update and make large-scale changes to data held in ArchivesSpace with Python's requests library. This is an add-on library that is not included in the default install, but you should be able to just enter *pip install requests* on the command line to install the library with

pip. Once this is installed, open editAPIs.py in a text editor to see how we can edit ArchivesSpace data on the backend.

Remember the JSON data we learned about previously? That sample file, resource.json is an example of an ArchivesSpace resource record from the API. In this script, we will use the API to get a resource record, create the same date expression from the JSON section by working with lists and dictionaries, and post the updated record back to ArchivesSpace. Here, lines 1-3 import the libraries we need, and lines 6-10 contain the important information ArchivesSpace requires to login via the API. If you have a local or development instance of ArchivesSpace, you can try to add in your own information, but you should not use this script on a production instance [21]. We also need the repository number and the resource number for the collection we wish to update. Edit these accordingly to try it yourself.

On line 13 we use *requests.post()* with the correct username and password and ArchivesSpace will give us a response as the *connectASpace* variable. If the connection is successful, *connectASpace.status_code* will be equal to 200. On line 15 we check for this and then print a confirmation while continuing the script. If the connection failed, the whole response will be printed from line 38. We can use the *.json()* method to get the date included in the response. This has a session key that gives us access to make changes for a set period of time, but we have to include that session key for each additional request we make, so we save it in the format ArchivesSpace likes as headers on line 19.

```
12| #inital request for session
13| connectASpace = requests.post(backendURL + "/users/" + user + "/login",
data = {"password":pw})
14|
15| if connectASpace.status_code == 200:
```

```
16|  print ("Connection Successful")
17|
18|  sessionID = connectASpace.json()["session"]
19|  headers = {'X-ArchivesSpace-Session': sessionID}
20|
21|  resource = requests.get(backendURL + "/repositories/" + repoID +
"/resources/" + resourceNumber,  headers=headers).json()
```

Now that we have permission to make changes, we make an additional API request for the resource we want on line 21, including the *headers* variable.

```
21|  resource = requests.get(backendURL + "/repositories/" + repoID +
"/resources/" + resourceNumber,  headers=headers).json()
```

ArchivesSpace then gives us the same JSON object we worked with before as the *resource* variable, so on lines 23-27 we make the same date expression change as in the JSON section above, and print the result on line 28. Even though we are not writing this data to a file, we still have to convert the sets of lists and dictionaries to a big string before posting it back to ArchivesSpace, which we do using the *.dumps()* method from the json library on line 31. Finally, on line 33 we post this updated record back to ArchivesSpace with another API request, and print the result on line 34.

```
31|  jsonData = json.dumps(resource)
32|
33|  postResource = requests.post(backendURL + "/repositories/" + repoID +
"/resources/" + resourceNumber,  headers=headers, data=jsonData)
34|  print (postResource.json()["status"])
```

We can make many kinds of large scale updates with the ArchivesSpace API and Python. The API documentation details each type of request we can make[22]. The request module is just managing HTTP URLs for us, somewhat like you would type in to a web

browser, but with Python we can link a number of API requests together to make complex changes and automate multi-step workflows. With a bit more coding, we can loop through many—or even all—of the resources held within ArchivesSpace, and run the same date expression update to each one. We could also make further requests to get child archival object or digital object records and apply the same code there. The API is so complete, you can update just about any type of data held within ArchivesSpace at scale. The API makes these scripts relatively stable, so they are easier to maintain over time then working with XML or other serialized data. Additionally, Python scripts can be scheduled to run at certain times, so once they are written, these automated processes can run on their own without any human intervention.

The ArchivesSpace API can also be used for exporting or displaying data as well. This is a much more consistent method than working with EAD exports. An effective example of using the ArchivesSpace API to export collection data for display is Hillel Arnold's StaticAid test case[23]. This example uses Python scripts and Jekyll to periodically create static HTML pages with collection data from the ArchivesSpace API. The result is a limited, but reasonably effective way of enabling access to archival collections with just ArchivesSpace and a web server. A similar, in-production example is North Carolina State University Libraries' Collection Guides system[24]. This is a Ruby on Rails web application that also relies on exported data from the ArchivesSpace API.

Conclusion

Python's independence and wide use make it a great tool for developing automated workflows between larger systems. If environments are set up effectively, Python scripts can be true cross-platform tools to manage data between essentially every open system. Python's prolific use with research data and general software development means is has extremely good publically-available documentation and wide variety of useful libraries. While working with Python, archivists are likely to find that

many of their problems have already been solved and that others have written code samples and made them available for use or modification.

The barriers to using Python to solve archives problems are steep and just getting started can be challenging. Although the syntax itself if not difficult, its independence from any sort of system or structure makes it particularly challenging for archivists that are just starting to work with code or on the command line. Just getting a working environment set up requires tackling some significant challenges, particularly on Windows systems.

However, once archivists become comfortable with a Python environment and start running code and getting feedback, learning Python to perform basic reformatting and data exchange can be much more accessible. The syntax of Python is concise and intuitive, particularly comparted to other technologies that archivists have learned to work with – whether that is XSLT, MARC, or odd "expert search" parameters in catalogs. Python syntax is much simpler that these tools. While archivists do not need to become software developers, many already perform complex and intellectually-challenging work with data that can be made easier by some basic Python scripts.

About the author

Gregory Wiedeman is the University Archivist in the M.E. Grenander Department of Special Collections & Archives at the University at Albany, SUNY. Here he manages preservation and access for permanent public records and develops digital collecting practices.

Notes:

1. "Bagit-python." Github Pages, Library of Congress. Accessed October 29, 2016. https://libraryofcongress.github.io/bagit-python/.

2. Heidi Frank, "Augmenting the Cataloger's Bag of Tricks: Using MarcEdit, Python, and PyMARC for Batch-Processing MARC Records Generated From the Archivists' Toolkit," Code4Lib Journal Issue 20 (2013). (Emphasis in original). Accessed October 30, 2016. http://journal.code4lib.org/articles/8336.

3. Jacob L. Nash and Jonathan Wheeler, "Desktop Batch Import Workflow for Ingesting Heterogeneous Collections: A Case Study with DSpace 5," D-Lib Magazine 22:1/2 (January/February 2016). Accessed October 30, 2016. http://www.dlib.org/dlib/january16/nash/01nash.html.

4. Amy Berish, "Learning Python as a Processing Archivist: A Reflection," Bits & Bytes: News from Rockefeller Archive Center's Digital Team (April 6, 2016). Accessed October 30, 2016. http://blog.rockarch.org/?p=1483.

5. There are a number of free and paid tools to learn Python for different types of learners. I recommend Codecademy (https://www.codecademy.com/) and Learn Python the Hard Way (https://learnpythonthehardway.org/book/), but others may have success with other methods.

6. "Python FAQ: Why Should I Use Python 3? / Fuzzy Notepad." Accessed October 21, 2016. https://eev.ee/blog/2016/07/31/python-faq-why-should-i-use-python-3/.

7. A good introduction to Bash is available at The Programming Historian. Ian Milligan and James Baker, "Introduction to the Bash Command Line," The Programming Historian. http://programminghistorian.org/lessons/intro-to-bash. Accessed September 20, 2016.

8. "Python Releases for Mac OS X | Python.org." Accessed October 21, 2016. https://www.python.org/downloads/mac-osx/.

9. Using Python on a Macintosh – Python 3.5.1 Documentation." Accessed October 21, 2016. https://docs.python.org/3/using/mac.html.

10. Ethan Gates, "Windows Subsystem for Linux – What's the Deal?" The Patch Bay (2016). Accessed October 30, 2016. https://patchbaynyu.wordpress.com/2016/10/19/windows-subsystem-for-linux-whats-the-deal/.

11. "Python Releases for Windows | Python.org." Accessed October 21, 2016. https://www.python.org/downloads/windows/.

12. "Lxml – Processing XML and HTML with Python." Accessed October 21, 2016. http://lxml.de/.

13. "Lxml – PyPI – the Python Package Index." Accessed October 21, 2016. https://pypi.python.org/pypi/lxml/.

14. "Lxml 3.6.0 – PyPI – the Python Package Index." Accessed October 21, 2016. https://pypi.python.org/pypi/lxml/3.6.0.

15. There are more robust ways to set up your Python environment using virtualenv, but the goal here is to get readers up and running code in the fastest and simplest possible way.

16. "python4archvists" Github. https://github.com/gwiedeman/python4archivists. Accessed September 20, 2016.

17. There is a good chart on the different I/O modes here. "Python Files I/O," Tutorials Point. http://www.tutorialspoint.com/python/python_files_io.htm. Accessed September 22, 2016.

18. There is a great introduction to using lxml with EAD files on the Bentley Historical Library's Archival Integration Blog. "Tools for the programming archivist: ead manipulation with python and lxml." Bridging Technologies to Efficiently Arrange and Describe Digital Archives, October 5th, 2015. http://archival-integration.blogspot.com/2015/10/tools-for-programming-archivist-ead.html. Accessed September 21, 2016.

19. "Lxml – PyPI – the Python Package Index." Accessed October 21, 2016. https://pypi.python.org/pypi/lxml/.

20. If you are unable to install lxml you can use the Element Tree library which is included with Python. You just have to change every line that reads: from lxml import etree as ET to import xml.etree.ElementTree as ET

21. The code itself will only create date expressions from dates you have already entered, but it will overwrite the date expression field. The script also requires a plain text password which is not a good idea to use with a production instance.

22. "ArchivesSpace API Reference," Github Pages. Accessed October 31, 2016. http://archivesspace.github.io/archivesspace/api/.

23. Hillel Arnold, "Introducing staticAid: A Static Site Generator for Archival Description." Accessed November 1, 2016. http://hillelarnold.com/blog/2016/02/a-static-html-site-generator-for-archival-description/. Test site available at http://hillelarnold.com/staticAid/.

24. "Collection Guides" NCSU Libraries. Accessed November 1, 2016. http://www.lib.ncsu.edu/findingaids.