**Guidance for the final course synthesis essay**

|  |
| --- |
| * **Reflect on the nature of research and how the course changed your thinking**, make connections within the course, to other courses, to practice. * Give a coherent picture of what you learned and how you are going to use what you learned. * Demonstrate your knowledge of specific research methods that you can adduce as examples. * The essay needs to be a reflection on concepts and ideas with the option of mixing in a bit of your personal stance on issues in research methods. * Think of the essay as explaining to another student what you learned in the course.   **Specifically**   * Distill major concepts and skills you learned (what you got out of the course)  and identify general insights. Make connections to other courses and subjects. * When five years hence you ask “What did I learn in UBLIS 575DS Research Methods?” the essay should give you an answer. * Thinking about your career and journey through life, reflect on how you see yourself using what you have learned.   You might want to consider some of the following questions (similar to the learning blog questions). These are just suggestions to think about; you may have different ideas on what to cover in this essay and how to organize it.  1 What have I learned, what was most important, what was most interesting, what was extraneous; what helps me in my (future) work? How?  2 How does a course idea support better service to users, directly or indirectly?  3 How does a course idea relate to other ideas in this course and/or to other courses?  4 Going forward, what more do you want to learn about research methods? How?  5 Can any of the ideas you acquired be transferred for use in other contexts?  Keeping a learning blog can greatly help you in producing this essay  Note: I welcome critique. It is the quality of the thinking that counts.  **4,000 - 5,000 words.** This is an indication of the substance expected in the reflection. It s not a length limitation (as you might find, for example, in specifications for a grant proposal). Whatever you write I will read. On the other hand, if your essay is below 3,500 words, you may be missing some substance.  May include references, but they do not count toward the length.  Assigned at the beginning of the course |

|  |
| --- |
| **Evaluation criteria** |
| To what extent does the essay  1 demonstrate an understanding of the broad nature of research as an approach to gaining knowledge?  2 demonstrate specific understanding of how research studies can be characterized?  3 demonstrate specific understanding of some research methods.  4 make connections between concepts in the course?  5 make connections to concepts in other courses and to ways the student might apply knowledge of research methods in her or his professional work?  Overall:   * Is the essay as whole and each section well written? * Are points well argued and, where applicable, supported with evidence? * Is the essay as a whole and each section coherent ? Does it flow in a logical and meaningful sequence? Does it make sense for the reader? |

**Further advice**

**What the essay should not be**

* The essay should not be a personal narrative how you progressed through the course.
* The essay should not be a listing of textbook chapters with a brief abstract/critique/review and/or a summary of what you got out of each. Such comments belong in the leaning blog chapter comments. You can draw on the book and the lectures and the readings and any other source that helped you understand research methods, then synthesize the ideas and discuss how they can be used.
* It is not sufficient to repeat or rephrase what you read in the textbooks or other readings. You must put this in a framework of your own thought, making your own connections.
* The essay should not be a discussion of your research topic or proposal except as you illustrate a concept or use of a specific research method by an example from your research proposal.

**What the essay should be**

* The essay should present your own deeper insights, comparisons, more general themes, going beyond what you read. It should make connections as mentioned above.
* In that context, it is fine to include critique or appreciation of a textbook chapter or a study you read if you use it to illustrate your own insights concerning a principle or approach in research methods. Likewise it is ok to use an issue in your research proposal as an example to illustrate a principle.
* The essay should discuss how a general principle applies to different research methods. Choose research methods that you studied in a little more depth. (Remember that in this course you cannot become an expert in all research methods covered in Wildemuth. Rather, you should get an overview of the research methods landscape, have an idea of what research methods exist, and learn in more depth about some of these methods, especially those important for your research study.)
* The essay needs to have a good structure. **Include an introductory paragraph that explains the structure.** Use section headings in the essay to communicate the structure.  
  Organize the essay around ideas, not the chronology of the course. However, as a stepping stone to a synthesis you might "go through the course in order and note my observations, suggestions and musings" (as one student put it), elaborating on your learning blogs. But then you would take the ideas generated through this exercise and weave them together using a broader perspective.
* There are many good ways to write such an essay. I do not give specific examples of ideas or connections from UBLIS 575DS because that would channel the thinking of some students into a certain direction. Leaving the field wide open is more conducive to creativity and coming up with new perspectives.

**Advice on how to approach writing the essay**

* To plan and write the essay, you can do the following:
* Think about all the points you want to make in the essay, then write them down in an abstract that will tell the reader what to expect.
* Create an outline for the essay (you may change this as you write). In the final version use section headings.
* If you have notes you want to incorporate, paste them in the appropriate section.
* Write the sections
* If you kept learning blogs, use them a source for ideas.
* Looking at the course learning objectives may be helpful in thinking about the content and structure of the essay.
* On style: Use a concise, clear style, not a chatty narrative. The Hemingway app can be helpful for cleaning up your prose and tighten up your writing.

(1) Do not to repeat a thought. You have a lot of repetition

(2) Do not say the obvious.

(3) "Omit needless words".

In previous semesters I have seen many examples where a student did not follow the advice in  
Box\575DS\Week00\SoergelMindfulMicroInformationArchitectureV8.docx

Please (re-) read this document and follow the advice unless there are good reasons not to.

**Starting on the next page is a good essay from LIS503PDS Computer Programming.   
You need to read just Part 1. Reflection.**

**Attachment**

**A sample Final Course Synthesis Essay**   
from UBLIS 503PDS Computer Programming for Information Science

For this course, students wrote a two-part essay. **You need to read just Part 1. Reflection.**

In the sample essay,   
Part 1 takes a general principle, the importance of precision and accuracy, and discusses it for several contexts. For UBLIS 575DS, the whole essay is reflection.

Part 2 demonstrate knowledge of programming concepts, specifically as implemented in the programming language Python. For LIS, you must demonstrate knowledge of some research methods of your choice, but you should weave this demonstration into your entire essay.

**Part 1: Reflection**

# Precision and the complexity of information

One thing that quickly became apparent to me this semester is the importance of precision and accuracy when writing Python code. There are many rules that must be followed exactly in order for the code to work, yet these are not immediately obvious to the average human. A frequent issue that came up for me was keeping track of data types. I generally only worked with strings, integers, and Boolean types, and many functions work well with the type I would naturally use, so it was often not something I had to think about. In other cases, however, data types would take me by surprise. For example, the print() function can print anything, but when concatenating strings, it is necessary to make sure everything being printed is a string. Thus, print ('There are this many items in the list: ' + len(litotes)) does not work because len() produces an integer and integers cannot be concatenated with strings. I frequently encountered a similar issue trying to concatenate lists of strings.

I think the reason that problems like this always seemed to surprise me is that human brains do not work like computer programs. A human has no problems putting a word and a number in the same sentence, but for a computer to do this it needs extra help. The computer typically does different things with words and with numbers, and therefore needs to store them in different categories. The more I have thought about the complexities involved in making the computer do a simple task, the more I have come to appreciate the computing power of the human mind.

There are several methods I used for keeping details like data types straight. A feature of Python that does not help with data types is that it is possible to assign any kind of data to any variable, and even to change a variable's type, without the program complaining. The resulting mushiness is the reason why we were instructed to label our variables with their intended type. This practice helped to remind me to consider the data type each time I created a new variable. Another useful feature is that error messages resulting from incorrect data types are usually easy to understand, so when problems such as my print issue came up, I was usually able to resolve them quickly.

The precision and accuracy needed to translate human thought into a logical system has come up in other areas of my studies and work in libraries as well. In my project for this class I have been parsing bibliography entries to separate out the different kinds of information they contain. A human has no trouble looking at a bibliography entry and knowing which words correspond to people's names, to the date of publication, or to a URL. We do require cues like quotation marks or italics to distinguish, e.g., the title of a chapter from the title of the book containing it, but we usually cope fairly well with variations in format. Getting the computer to do this, on the other hand, proved to be a complex task. Often I ended up relying on the precise format set by the style guide (in this case, the *Chicago Manual of Style*) but I would have preferred not to, since people deviate from these instructions frequently. In one case I was not able to distinguish between fields: if a bibliography entry included a translator or editor before the place of publication, I had no way to tell the computer where to separate these words. I could look for the words "translated by" or "edited by", and assume a name came after, but I could not think of how to tell the computer where the name ended. Relying on a period would not work—if the name contained a middle initial with a period, or an abbreviated honorific, stopping at the first period would be incorrect. I solved this problem by relegating editors and translators, along with the place of publication, to an "other information" category. Usually this would contain only the place of publication—a fact that is rarely useful to the average user.

The above example illustrates a way in which human knowledge (the information contained within books and articles, for example) cannot always fit perfectly into a simple logical system. The publishing industry is in essence an attempt to codify and standardize knowledge for wide distribution, yet my work in libraries and in LIS courses has shown me that there must always be exceptions and irregularities in any system meant to organize information. Otherwise the system ends up being so generalized as to be useless. At the library we use notes fields when we need to know something about an item that does not fit the expected pattern or does not fit anywhere in the record. I have learned to be very vigilant of the notes field; confusions are often resolved by reading the notes. This is a similar level of attention to detail, precision, and procedural complexity that I have come to appreciate in computer programming. In this sense, I think that studying Python has helped me understand more intimately the major, intrinsic challenges of information science. Even in the rudimentary tasks I completed in this class, the needs of the system and the expectations of the human user often clashed. I have realized that this problem permeates the field of information science.

# Thinking of the user

This clash between user and system is something that came up often throughout the semester—not as an inherent quality of Python, but as something that the programmer must attend to. Python, of course, does not care about what the human user wants it to do; it can only follow its instructions. Therefore, it is up to the programmer to anticipate how the user might act and account for a range of possibilities. I found it interesting that this concern with the user's needs often went hand-in-hand with a particular code structure. Loops, especially while loops in my experience, allow the user to continue trying inputs until they find one that works. This structure can work well for obvious input rules (for example, entering a number using numerals only) but can become cumbersome if it is difficult for the user to know exactly what inputs will work. Therefore it is also necessary for the programmer to anticipate the many ways another human would enter the information asked and allow for a variety of inputs. For example, in my bibliography program I accounted for abbreviations as well as complete words (e.g., trans. and translator), assuming that people would not always follow the style manual's protocol in cases like this. Although it did not come up in our introductory work in class, I can imagine that at a certain level programmers may find it difficult to know how the non-programmers will approach the program. If one is involved in developing a program for months or years, its intricacies become second-nature.

The same problem arises in libraries all the time, and has been covered in several of my LIS courses. Decades ago, library systems were designed to prioritize collections and users were expected to learn how to use them. In a non-digital age, tailoring systems primarily to user needs was perhaps more difficult. By now, however, focus has shifted to prioritizing users above all. This extends to thinking about how users will prefer to interact with the library. We cannot expect users to come in knowing specialized terms like "stacks", "reference", etc., so we try to use terminology that is more readily-understood, and to design systems that are accessible at all levels of knowledge.

# Information overload

Other lessons I took away from this course have less to do with Python or coding specifically, but with the process of learning to code. The concept seemed a bit daunting at first because what I was looking at was so information-dense. Even comments written in the code (marked by a #) were confusing at first. For some reason the fact that they were in a different color and preceded by an odd symbol seemed to make my brain doubt that they would be in plain English. The result was that I tended to gloss over entire chunks of code in confusion rather than attempting to look at single lines and make some sense of them.

While at this point I still have to refer to a book or website when I write code, I no longer find it so mystifying to read through. In the end, it took going through the language, concept by concept, to become familiar with it gradually and feel comfortable looking at blocks of code. My initial encounters with code could be described as information overload, and the exercise helped me truly appreciate the dangers of this condition.

It is important to be able to understand the feeling of information overload, because information science professionals should always be working to avoid it. A librarian who knows a multitude of ways to get users the information they seek can be tempted to present the user with all of these suggestions at once, not realizing that giving too much information at one time can actually render it unintelligible to the user. Just as I found with learning Python, it is better to take things one at a time, to give the brain time to adjust to a new way of thinking.

# Where to look for help

A final important and applicable lesson that I found in this course is the importance of knowing where and when to find answers from an outside source. What I most enjoyed about learning Python, in fact, is that it is not necessary to memorize each “word” as you would for a spoken language. There is no reason to do this, as it is always possible to look up a function or method as needed. I soon became comfortable with the best online sources for my questions. For general reference and documentation on built-in functions, I consulted python.org. However, sometimes this was a bit difficult to understand, so websites devoted to teaching (w3schools.com is one I used often) served to give more in-depth examples. When I was trying to find a way to solve something especially tricky, a Google search generally led me to the Stack Exchange forums, where someone else had always encountered my problem before me, and a community of programmers offered solutions and advice. Something I learned on Stack Exchange is that there are usually multiple ways of solving the same problem.

I appreciated this kind of learning, which was very different from what is usually found in academic work. An information professional such as a librarian does not have to have all the answers memorized either. By the very nature of the job, they can always look up what they need to know in the moment. This does not mean, of course, that no training or education is required. After all, if I had attempted to write a Python program like my bibliography tool with no prior instruction, relying on Google searches, I probably would have given up quickly. Yet, the skill of knowing where and when to look up what I do not know served me well in this course as I know it will in my future career.

**Part 2: Programming concepts**

# For the computer programming course, students were required to demonstrate knowledge on programming concepts, specifically as implemented in the programming language Python, in a separate part of the essay. For the research methods course, you must demonstrate knowledge of some research methods of your choice, but you should weave this demonstration into your entire essay.

*Note: From Part 2 I learned a lot about Python I did not know before. We are all part of a learning community.*

# Data types

The most common data types in Python, at least in my experience, are strings, integers, floating-point numbers (floats), and Booleans. Strings are sequences of characters and are denoted with quotation marks. Integers and floats are both numbers; floats have decimal points, while integers are equivalent to mathematical integers. Booleans are a limited set of conditions—True and False. It is important to keep track of the data types assigned to variables because some functions only take certain data types or produce certain data types. Additionally, errors can occur when mixing data types. Thus,

print(5+5)

evaluates to 10, while

print('five' + 'five')

evaluates to 'fivefive'. "Adding" two strings like this is called concatenation. Python cannot concatenate a string with a non-string, however, so

print('five'+5)

produces an error:

TypeError: can only concatenate str (not "int") to str

In order to print the string 'five5', we would have to put quotations marks around the 5, making it a string:

print('five'+'5')

Using quotation marks to make something a string only works with a literal, i.e., the literal representation of the data—the word (or number) itself. If the data in question is represented by a variable, certain functions convert between one type and another. This is handy if a method or function produces an integer that must later be treated as a string. For example, len() returns the length (number of characters) of a string. It returns an integer, but to concatenate it with a string we would have to convert it into string form:

name = 'Sophia'

print('Your name has ' + str(len(name)) + ' letters.')

Conversely, we could join the strings and number with commas, for a more elegant solution:

print('Your name has ', len(name), ' letters.')

# Data structures

Data structures include lists, tuples, dictionaries, and sets. These offer different ways of structuring multiple units of data to work together.

## Lists

Lists are ordered, changeable collections of data. Lists are denoted by square brackets []. To create a list, we can simply type the data into the brackets:

myList = ['first thing', 'second thing', 'third thing']

To retrieve an item from a list in a known position, we put the position number (i.e., index) in square brackets (the first item is at position 0):

print(myList[1])

evaluates to 'second thing'. The index() function finds the position of the first occurrence of an item in a list, so that

print(index('second thing'))

evaluates to 1.

Often the items that will populate a list are not known at the outset, or to type them all in would be tedious. In this case, the append method is used to add an item onto the end of an existing list. First a blank list is created:

myList = []

And then something can be appended:

myList.append('first thing')

Usually, the item appended will not be a literal, but the result of some other function. As an example, here is a program that creates a list of all the short words (three letters or fewer) in a block of text:

sText = 'The Piglet lived in a very grand house in the middle of a beech-tree, and the beech-tree was in the middle of the forest, and the Piglet lived in the middle of the house. '

First, the text is broken up into a list of words by using the split method:

wordList = sText.split()

Now, using a for loop, the program runs through the list of words and adds to the list only the words whose length is less than four:

listOfShortWords = []

for word in wordList:

if len(word) < 4:

listOfShortWords.append(word)

To see the list, we must print it:

print(listOfShortWords)

['The', 'in', 'a', 'in', 'the', 'of', 'a', 'and', 'the', 'was', 'in', 'the', 'of', 'the', 'and', 'the', 'in', 'the', 'of', 'the']

Many functions and methods return lists. The above example uses the split method, one of the built-in string methods, to create a list of words from a longer string. Functions that are written to return lists will do so even if the list contains only one item. Therefore, it is important to pay attention to what data structures are returned by functions. Errors could turn up if, for example, the list resulting from a function were treated as a string in a subsequent function.

A list of words can be converted to a single string again with the join method, which also allows the programmer to insert a string of their choice between each of the joined list items. Often only a blank space is required. Thus we can use join on the word list from the above example to return it to its original form:

print(' '.join(wordList))

The Piglet lived in a very grand house in the middle of a beech-tree, and the beech-tree was in the middle of the forest, and the Piglet lived in the middle of the house.

## Tuples

Tuples are ordered but unchangeable collections of data. Tuples are denoted by parentheses (). Because they are unchangeable (or immutable), once a tuple is created, items cannot be appended or removed as with lists. However, many of the same operations that can be done on lists also work with tuples. For example, it is possible to pull out a specific item from a tuple given its index:

myTuple = ('first thing', 'second thing', 'third thing')

print(myTuple[1])

second thing

Tuples can be pulled apart by assigning separate variables to each item. For example:

a, b, c = myTuple

Now a has been assigned the first item in myTuple, b has the second item, and so on. We can confirm this with print:

print(a)

'first thing'

print(b)

'second thing'

print(c)

'third thing'

This ability of tuples can cause confusion if the number of variables to be assigned does not match the number of values within the tuple. This can occur when calling functions that return tuples. When a function returns more than one object, it actually returns them in the form of a tuple. Typically, when the function is called, however, it is most useful to be able to manipulate the objects independently, so they are each assigned to a variable. As an example, we can imagine a function that takes a list of words and returns the number of short words (three letters or fewer) and the number of longer words (over three letters). We can call it countWords(). Normally, to call the function we would assign it to two variables:

iShort, iLong = countWords(wordList)

where iShort is the number of shorter words and iLong is the number of longer words.

We can then print these two variables to verify that it worked:

print(iShort)

20

print(iLong)

14

However, if we were to forget that countWords returns two values and attempted to assign it to a single variable, the variable itself would become a tuple.

iShort = countWords(wordList)

print(iShort)

(20, 14)

This could cause problems if the variable were later treated as, for example, an integer, but in general it still works. If the function returns three objects and we try to assign it to two variables, the program cannot know which objects to assign to what variable and it produces an error. If countWords also returns the total number of words in the string (e.g., iTotal), but we forget about this feature, we could end up with this:

iShort, iLong = countWords(wordList)

…

ValueError: too many values to unpack (expected 2)

## Dictionaries

Dictionaries are collections of objects with two special features: the objects are unordered (unlike lists and tuples), and the objects consist of a key and a value, both of which are created by the programmer. Dictionaries are denoted by curled brackets {}. Within the brackets, a colon separates each key from its value, and commas separate key-value pairs from one another. A dictionary can be created by typing it out:

dWinnieThePooh = {

'author' : 'A. A. Milne',

'country' : 'United Kingdom',

'sequel' : 'The House at Pooh Corner'}

Writing each key-value pair on a new line helps readability. Another way to create a dictionary is to use the dict() function:

dAnneOfGreenGables = dict(author = 'L. M. Montgomery', country = 'Canada')

Printing the dictionary shows it with its brackets and colons:

print(dAnneOfGreenGables)

{'author': 'L. M. Montgomery', 'country': 'Canada'}

With lists it is possible to produce a single item by using its index. Since dictionaries are unordered, they do not have numbered indices. Instead, the index is the key. If we forget the sequel to *Winnie-the-Pooh* we can retrieve it:

print(dWinnieThePooh['sequel'])

The House at Pooh Corner

Attempting the same with *Anne of Green Gables* does not work because the dictionary does not contain a ‘sequel’ key:

print(dAnneOfGreenGables['sequel'])

…

KeyError: 'sequel'

There is another way of retrieving values from dictionaries that avoids this issue. The get() method can specify a default value to return in case the key does not exist:

print(dAnneOfGreenGables.get('sequel', 'unknown'))

unknown

It is also possible to add a key and value to a dictionary:

dAnneOfGreenGables['sequel'] = 'Anne of the Island'

print(dAnneOfGreenGables['sequel'])

Anne of the Island

When we realize that *Anne of the Island* is not the direct sequel to *Anne of Green Gables*, we can also change this value:

dAnneOfGreenGables['sequel'] = 'Anne of Avonlea'

print(dAnneOfGreenGables['sequel'])

Anne of Avonlea

The keys are not as easily changeable as the values. Practically speaking, re-naming a key would involve deleting the key-value pair and appending a new one. Deleting can be done with the pop() method, which also works for lists.

## Sets

Sets are the simplest collection of values. They are unordered, and values cannot be repeated. Sets are denoted by curly brackets, as with dictionaries. Unlike dictionaries, the items in a set have no index with which to retrieve them individually. A set can be created by typing it into curly brackets:

anneBooks = {'Anne of Green Gables', 'Anne of Avonlea', 'Anne of the Island'}

Because the items have no order, printing the set may return the items in a different order than they were entered. Indeed, this is the case here:

print(anneBooks)

{'Anne of the Island', 'Anne of Green Gables', 'Anne of Avonlea'}

More items can be added to the set by using the add() method:

anneBooks.add('Anne of Windy Poplars')

print (anneBooks)

{'Anne of Windy Poplars', 'Anne of the Island', 'Anne of Green Gables', 'Anne of Avonlea'}

Items can be deleted in a similar fashion with remove(). However, as with the keys in lists, items cannot be changed. If one of the books in our list were misspelled, we would have to remove it and add it again with the correct spelling. An alternative would be to convert the set to a list using the list() function, altering the list as needed, and converting back to a set.

A list or tuple can be converted into a set using the set() function. The set may end up containing fewer items than the original list or tuple because only one instance of any duplicate items is retained. Each item in a set must be unique. For an example, we can create a set from our earlier wordList:

wordSet = set(wordList)

print(wordSet)

{'the', 'house', 'a', 'of', 'very', 'was', 'forest,', 'Piglet', 'beech-tree,', 'The', 'in', 'and', 'house.', 'beech-tree', 'middle', 'lived', 'grand'}

Using len() we can see that the set is indeed shorter than the list:

print(len(wordList))

34

print(len(wordSet))

17

At first, the limitations of sets (lack of order, lack of index, lack of duplicates) may make them seem less useful than lists. The advantage of sets comes in the ability to perform mathematical operations (that is, apply set theory) with them. It is possible to generate the union of two sets (a OR b), the intersection between them (a AND b), and perform a number of other related methods. In the example of lists of words, sets allow for some rudimentary analysis and comparison between texts. Our initial word list is derived from the opening sentence of Chapter 3 of *Winnie-the-Pooh*. A literary scholar may wish to compare, for example, the vocabulary of *Winnie-the-Pooh* with that of *Anne of Green Gables*. A more fruitful comparison would involve the entire length of each novel, but for this example we will use a sentence from each. First we rename the original set of words to be more descriptive:

poohSet = wordSet

Then we generate another set from a sentence from *Anne*:

anneText = 'A huge cherry-tree grew outside, so close that its boughs tapped against the house, and it was so thick-set with blossoms that hardly a leaf was to be seen.'

anneList = anneText.split()

anneSet = set(anneList)

print(anneSet)

{'leaf', 'a', 'blossoms', 'grew', 'boughs', 'seen.', 'it', 'was', 'and', 'house,', 'to', 'thick-set', 'tapped', 'so', 'be', 'huge', 'close', 'cherry-tree', 'that', 'hardly', 'outside,', 'with', 'A', 'the', 'its', 'against'}

If we wanted to know which words occur in both sets, we need the intersection of the two:

print(poohSet.intersection(anneSet))

{'the', 'a', 'and', 'was'}

Using lists would be less helpful in this instance. Our imaginary literary scholar is not interested in the number of times each word occurs in a sentence, she merely wants to look at the set of all words that the two sentences have in common. Counting each instance of “the” or “and” would be useless here.

It is obvious, however, that the above example is flawed. The word “house” appears in both sentences, but the program did not catch this because one of the instances had a comma at the end. In order to perfect this vocabulary comparison tool, we will have to delve into more features of Python.

# if/else

In an above example, the function wordCount counted how many words in a list were of different lengths. An if statement is needed to have a program act a certain way only if a condition is met. In this case, the program does the following:

if len(word) < 4:

listShort.append(word)

else:

listLong.append(word)

The initial if is followed by a condition (that the length of a word is less than four), and a colon. The action that the program will take if the condition is met is on the next line, and this line must be indented. Since we want the program to do a different action if the condition is *not* met, we include an else statement. As with if, it is followed by a colon and a second, indented line. However, since the condition for else is simply that the condition for if is not met, there is no need to add any more explanation to the else line. If we wanted the program to carry out even more tasks, we could use elif, an abbreviation of else if. Elif follows the if statement and describes a second condition to be met if the first is not met. There can be many elif statements in a row, and all can be followed by a final else statement, for the case where none of the above conditions are met. Of course, it is not necessary to include and elif or else statements at all; this would mean that the computer does nothing if the if condition is not met.

# for loops

The if/else construction above works only for a single variable “word”. We could write:

listShort = []

listLong = []

word = 'house'

if len(word) < 4:

listShort.append(word)

else:

listLong.append(word)

print(listShort)

print(listLong)

And the output would be an empty list for listShort and a list containing ‘house’ for listLong. This is a fairly useless task as we could have counted the letters in ‘house’ ourselves. The if/else construction becomes more useful when iterated over many items quickly with a for loop. A for loop performs an action on every item within a collection, such as a list. Below is an example of a more complete wordCount program:

for word in wordList:

if len(word) <4:

listShort.append(word)

else:

listLong.append(word)

This program checks the word length of every word in wordList. We do this by beginning a statement with “for”, then assigning a variable to represent an individual item in a collection, then “in”, and lastly the name of the collection (in this case, a list). Again, the statement is followed by a colon and the subsequent statement(s) must be indented. An interesting feature of the for loop is that it was not necessary previously to specify that the items in wordList would each individually be called “word”. Furthermore, we could have chosen any variable name.

for heffalump in wordList:

would work just as well, as long as we used “heffalump” again further down when we tell the program what to do with each item. The for statement itself is assigning a variable name to each list item.

This program then looks at each word one at a time, checks to see if it is less than four letters long, in which case it puts it in one list. If this condition is not met, it puts the word in another list. Printing the lengths of the lists confirms that they are now populated with words, and gives us a breakdown of the word lengths from the original sentence.

print(len(listShort))

20

print(len(listLong))

14

# Built-in string methods

To return to our task of comparing the vocabulary from two different books, it is necessary not just to count words, but to modify them to remove the punctuation and capitalizations that make the computer think two identical words are actually different. This can be accomplished with the string methods that are built in to the basic Python language. The string method split() has already been used to create our lists of words in the first place. Split() splits up a string at every occurrence of a specified character or string, returning a list of strings. The default string with which to split is an empty space, which is what we used for the word list. We can also use split() to turn the hyphenated words in our list into two separate words. To save time, we will iterated it over the whole list. We could do the following:

poohList = wordList

(we’ll start be renaming wordList to something more descriptive)

betterPoohList = []

(this is a new list where words do not contain hyphens)

for word in poohList:

betterPoohList.extend (word.split('-'))

Extend() is similar to append(), but it can add the contents of one list to another. If we used append, we would end up with a list of lists, because split() returns a list and not a string. If we print betterPoohList we see that the hyphen problem has been fixed:

print(betterPoohList)

['The', 'Piglet', 'lived', 'in', 'a', 'very', 'grand', 'house', 'in', 'the', 'middle', 'of', 'a', 'beech', 'tree,', 'and', 'the', 'beech', 'tree', 'was', 'in', 'the', 'middle', 'of', 'the', 'forest,', 'and', 'the', 'Piglet', 'lived', 'in', 'the', 'middle', 'of', 'the', 'house.']

Next, the lower() method converts all letters to lowercase. This eliminates the problems with the word “the”. As is, the program would see “the” and “The” as two different words. At the same time, the strip() method can be used to remove punctuation from the ends of words. Strip() defaults to removing blank spaces from the beginning and end of a string. By adding a string of characters within the parentheses, we can customize which characters are stripped. Since none of our words have punctuation at the beginning, we can use rstrip() which removes from the right side only. Combining the lower() and strip() actions into one program will save time:

bestPoohList = []

for word in betterPoohList:

bestPoohList.append(word.lower().strip(' ., '))

print(bestPoohList)

['the', 'piglet', 'lived', 'in', 'a', 'very', 'grand', 'house', 'in', 'the', 'middle', 'of', 'a', 'beech', 'tree', 'and', 'the', 'beech', 'tree', 'was', 'in', 'the', 'middle', 'of', 'the', 'forest', 'and', 'the', 'piglet', 'lived', 'in', 'the', 'middle', 'of', 'the', 'house']

Now the list is ready for comparison to the *Anne* list. First we can convert it to a set once more:

poohSet = set(bestPoohList)

Performing the same manipulations on the *Anne* word list yields the following set:

{'leaf', 'a', 'blossoms', 'grew', 'boughs', 'seen', 'it', 'was', 'and', 'house', 'to', 'thick', 'set', 'tapped', 'so', 'be', 'huge', 'close', 'cherry', 'tree', 'that', 'hardly', 'outside', 'with', 'the', 'its', 'against'}

Finally, the intersection of our improved word sets yields a more accurate result:

print(poohSet.intersection(anneSet))

{'the', 'a', 'house', 'tree', 'and', 'was'}