The Many Authors of *The Several Houses of Brian, Spencer, Liam, Victoria, Brayden, Vincent, and Alex*: Authorship, Agency, and Appropriation

Zach Whalen
University of Mary Washington
zwhalen@umw.edu

*The Several Houses of Brian, Spencer, Liam, Victoria, Brayden, Vincent, and Alex* is a computer-generated children’s book of 53,651 words and 350 unique illustrations arranged over 800 pages. The text is a cumulative poem in the style of the nursery rhyme “This is the House that Jack Built,” but with a house for each of the eponymous seven individuals, and with each of their houses containing many more types of things. These houses, these things, and these words were chosen by a Python script that I wrote, and the resulting novel -- which can be viewed on my Github repository -- is the outcome of a specific execution of this script that I carried out on December 16, 2017. I completed this novel for NaNoGenMo (National Novel Generation Month), where participants spend November writing code that will generate a novel of at least 50,000 words, then share a sample and their source code. My 2017 entry was late, but I still consider it a success.1

In this artist’s statement, I will reflect on the context for creating this book, and use it as an illustration of the challenges that so-called computer-generated text pose within the scope of digital creative writing studies and practice. I will also comment on some of the conceptual, ethical, and aesthetic issues involved in declaring ownership (authorship) of this kind of work, and I reflect on the pedagogical implementation of this kind of work in a creating coding classroom.

This sample from the first few pages of Chapter 3, “Liam and the cheetah that ran faster than a person,” demonstrate the concept behind the book’s structure:

p. 229:

This is Liam

1 The source code and a complete sample are available in this Github repository: https://github.com/zachwhalen/nngm17
This is the NEW HOUSE that Liam built

This is the FAMILY that moved to the NEW HOUSE that Liam built.

This is the SOLDIER that waved to the FAMILY that moved to the NEW HOUSE that Liam built.

This continues for some time until the chapter ends with the following cumulative sentence:

This is the CHEETAH that ran faster than the PERSON that lost consciousness in the HOUSE FIRE that hurt the PERSON that pushed the CAR that killed the PERSON that bought the CAR that moved the PERSON that arrived at the PLACE that went in to see the DOCTOR that healed the SICK PERSON that kissed the DOG that guided the BLIND PERSON that propelled herself through the WATER that doused the FIRE that burnt the CARELESS PERSON that quieted the DOG that followed the PERSON that bought the FOOD that tastes better with the SALT that seasoned bland FOOD that pleased the HUNGRY PERSON that waved to the FRIEND that shouldered the PERSON that mailed the FRIEND that needled the PERSON that owned the CAR that killed the CAT that fought the DOG that guided the BLIND HUMAN that had the MENTAL ILLNESS that subjected one to much PAIN that provided information to the PERSON that sealed something with the WAX that coated the CAR that killed the DOG that caused the ACCIDENT that happened to the CLUMSY PERSON that gave the GIFT that pleased the PERSON that fed the CAT that hid under the BED that rested the PERSON that destroyed the COMPUTER that taught lessons to the STUDENT that failed the TEST that placed the STUDENT that thanked the TEACHER that listened to the STUDENT that proved the TEACHER that spanked the STUDENT that waited for the BUS that took you across the COUNTRY that honored the SOLDIER that waved to the FAMILY that moved to the NEW HOUSE that Liam built.

The items and relationships in this sentence have been selected and arranged randomly by a computer program, but many of these statements seem plausible, for example, “the CAR that killed the CAT that fought the DOG,” because these are entries in the ConceptNet.io taxonomy, which is a system that encodes relationships among things. The entry for “cat” includes “fight a dog” among the list of things a cat is capable of, and the entry for “car” includes “kill a cat” among the list of things a car is capable of and so on. The computer program I wrote constructs sequences out of
these chains of causal association, but the formula for the poem follows a pattern with a long prece-
dent.

In other words, the story that my computer generates gains its richness and complexity by fol-
lowing a pattern that I have specified for it, but the cumulative pattern that this poem follows comes
from a folktale or nursery rhyme rooted in oral storytelling where the pattern assists the story-
teller’s memorization. The objects and characters join the story through plausible associations, and
then each verse is otherwise preoccupied with repeating each previous verse. As such, the bulk of
the story is deterministic and repetitive, but that feature is characteristic of this specific genre and
not a consequence of the limitations of the programming.

Like many folktales and nursery rhymes, the specific origins of the “House that Jack Built”
is unknown, but the taxonomic efforts of folkoristics might provide some useful context for the
purposes of this kind of story. That said, the Aarne-Thompson Motif-Index classifies “The House
that Jack Built” under the under-specified “Z” heading for “Miscellaneous Groups of Motifs,” and
in placing the story in the Tale-Type Index with number “ATU2035,” Aarne-Thompson (and now
Uther) relegate this story to the similarly generic class of “Formula Tales” with the subtype “Cumu-
licative Tales,” alongside other sequential narratives like “The Fleeing Pancake” (type ATU2025, of
which the “Gingerbread Man” is a familiar example in North America) and the macabre “Climax
of Horrors,” or type ATU2040 (Uther 525-37).

What stands out in each of these classifications is that, in each, it is the structure or syntagmatic
pattern of the story that communicates the most salient feature. This expository, cumulative syntax
of the story formula matters and makes these stories similar -- not the setting, characters, themes, or
even plot. In this way, the original “House that Jack Built,” is, therefore, not a story about Jack, the
“cat that ate the malt”, or even the “man all tattered and torn,” as much as it is a demonstration of a
storytelling technique based on association and memorization. This semantic paucity may be why
there appears to be so little scholarship about what the “House that Jack Built” story really means,
or even its exact cultural origins.

In describing a convenient paradigm for understanding combinatory poetics in the context of
electronic literature, Scott Rettberg has offered a convenient generalization by pointing out that
many such works are about creating interesting relationships between a database and an algorithm
(Rettberg 20). In a way, “The House that Jack Built” is also an example of how a database (this list
of people, animals and objects) and an algorithm (a sequence plausible subject/direct object pairs)
can express a story. Likewise, my work, *The Several Houses...*, is also an example of folktale type
ATU2035 because is constrained by the folkloristic definition of the formulaic folktale. Through
the power of the taxonomy at ConceptNet.io, the database is much larger, but the algorithm is
essentially the same, which is all the complexity necessary to create a novel that would fulfill the 50,000-word constraint of NaNoGenMo. In these ways, the complementary formulae of ATU2035 and NaNoGenMo, and the conceptual taxonomy of ConceptNet.io each stand against my ideational subjectivity and expressive humanity, and yet, I insist, the work of writing code that produces this work is a meaningful creative act.

“Computer-Generated” Texts and Attribution

To write or speak of computation in the service of generating text is to invoke and question the nature of authorship, and as the nominal author of The Several Houses... and the eponymous “artist” of this essay’s “Artist Statement”, it is hard to avoid the sense of feeling like a fraud. I confess I feel like an imposter in a journal on creative writing studies: I don’t teach creative writing classes, I don’t publish fiction, and I don’t routinely think of myself as a writer at all. I’m an Associate Professor in an English department, true, but I teach classes in an emerging area called “Digital Studies” that includes many more ways of thinking about working creatively with digital tools than would normally be considered “writing.” In this essay’s opening sentence, and a few times since, I’ve described The Several Houses... as a “computer-generated” book, not a “me-generated” book.

Of course, “computer-generated” is also a misleading phrase because humans have indeed created the words, the relationships among them, and the images that illustrate them as they appear in my book, but here again, most of these creators are, like my program, also not me. They are the users of Flickr, The Noun Project, and most importantly, ConceptNet.io, whose licensed contributions² are acknowledged in the lengthy appendix to The Several Houses.... All told, the work of some 431 individuals complete this book, not counting the programmers and administrators who create and make possible the Flickr, ConceptNet, Noun Project, Corpora and Wordnik APIs which are all necessary for the program that creates the book to execute completely, to say nothing of Github or the community that develops the Python programming language.

And even though my name is on the title page of this book -- where it says, “by Zach Whalen” -- my chief contribution isn’t the book itself, but rather the script which I wrote for my computer.

² ConceptNet is offered under a CC BY-SA. Its creators suggest the following attribution:

This work includes data from ConceptNet 5, which was compiled by the Commonsense Computing Initiative. ConceptNet 5 is freely available under the Creative Commons Attribution-ShareAlike license (CC BY SA 4.0) from http://conceptnet.io. The included data was created by contributors to Commonsense Computing projects, contributors to told, the work of some 431 individuals complete this book, not counting the programmers and administrators who create and make possible the Flickr, ConceptNet, Noun Project, Corpora and Wordnik APIs which are all necessary for the program that creates the book to execute completely, to say nothing of Github or the community that develops the Python programming language.
In carrying out that script -- 700 or so lines of Python -- a computer follows my instructions to download, process, select, and arrange all of the information required to create a PDF with text like the sample quoted above and included in my project’s GitHub repository. Still, it’s not as if that code sprang from my mind in a frenzy of romantic epiphany, and much of it is adapted from examples in documentation or copied and pasted from solutions posted on StackExchange.com.

My point is that digital creative writing or programmed text generation is fundamentally but often tacitly collaborative; it’s an arrangement and selection of elements -- phrases, words, symbols -- from other sources in a way that is a literal application of what D.F. McKenzie theorized as the sociology of texts, but sometimes, participating in that kind of authorship network feels like perpetrating a literary hoax.

Perhaps this complex social reality is why audiences are drawn to the “ghost in the machine” of attributing authorship to the computer. The closer one gets to the code that generates procedural creative work, the less one is held in thrall to the emergent, spectral “I” of the machine that prints out “I think therefore I am” in the Harlan Ellison short story. But the ease of attributing authorship to computer processes is a well-worn path in popular discourse about artificial intelligence. In writing about Joseph Weizenbaum’s ELIZA/DOCTOR (1966), Noah Wardrip-Fruin defines the Eliza-effect as the “phenomenon in which audience expectations allow a digital media system to appear much more complex on its surface than is supported by its underlying structure” (15). This is complementary to other epistemological patterns that are based on abstractions from complexity like what Clay Shirky wrote of in 2009 as “algorithmic authority,” defining it as “the decision to regard as authoritative an unmanaged process of extracting value from diverse, untrustworthy sources, without any human standing beside the result saying ‘Trust this because you trust me’” (Shirky n.p.).

As Leah Henrickson argues, this tendency to attribute agency or authority to non-human actors is a way of giving power to the computer by allowing it to enter into a system of linguistic discourse where to speak (or to write) accords with the way we (humans) rely on conventions as a way to facilitate intersubjective identification. Just as any speech act connects us with a speaker through some measure of interpretation, assumption, and inference, we may also find ourselves making the same assumptions of a computer when, “If users ascribe beliefs and desires to a system, and act accordingly, the system therefore becomes a social agent when it completes its assigned tasks because the computer has been given power by the user” (Henrickson “Tool vs. Agent” 188).

Through this semantic lens, the procedurally-rhetorical claims baked into the Eliza effect and
algorithmic authority find their consequences in socio-cultural reality. In another essay, Henrickson links the reception of natural-language generation systems to the role of postmodern and conceptual writing in 20th- and 21st-century literature. By focusing on intention and power as key determinants of the hermeneutic contract of authorship, Henrickson aligns Darius Kazemi’s seminal NaNoGenMo contribution, *Teens Wander around a House* with the disruptive and frequently boring aesthetics of works like, Claude Closky’s ‘The First Thousand Numbers Classified in Alphabetical Order’ and Samuel Beckett’s *How it Goes* (Henrickson “Computer-Generated Fiction” 59). Focusing on the way that both *Teens* and *How it goes* make latent authorial intention more visible by starving the work itself of the kind of clues that allow readers to influence their own interpretation (Kazemi’s work does not, within itself, explain the origins of its text fragments and Beckett’s work does not use punctuation), Henrickson locates evidence of context in the aesthetic features of the generated texts. But there is a different similarity at work in relating Closky’s poem to computer-generated texts that worth unpacking. As Henrickson writes, ‘‘The First Thousand Numbers’ is a product uniquely of Closky’s mind. By instilling boredom and frustration in his readers, Closky asserts authorial dominance...considered in light of the hermeneutic contract, [it] is neither interesting nor understandable: it is an explicit deviation” (Henrickson “Computer-Generated Fiction” 59). The fact that Closky insists upon this deviation, and that Craig Dworkin gives license to this deviation by including Closky’s work in an anthology of conceptual writing (*Against Expression*), is a good example of how conceptual writing, as I will discuss in the next section, always reconnects with the lived social realities of the human subjects who carry it into existence and the audiences to whom its works are enacted.

Attributing authorship to a computer is one way to call reader’s attention to the material and cultural contexts of computer-generated literary production, although this attribution also becomes a misleading, click-bait tease that pre-conditions audiences to perceive nonsensical or hilariously absurd text as all the more hilarious because it was produced by a robot. Racter, the putative computer author of *The Policeman’s Beard is Half Constructed* – published in 1984 with the subtitle “The First Book Ever Written by a Computer” – provides one influential example where the irony in lines like “Blue potatoes are ungainly things / as are red and purple lamb chops” is entertaining as an example of a computer’s serendipitous alleged insight. More recent examples include a chapter of *Harry Potter and the Sorcerer’s Stone* generated Botnik Studios in late 2017 (Hathaway n.p.), and a Twitter meme a few months later in which people offer variations on, “I forced a bot to watch 1000 hours of <something>. This is the script it produced” (Matt and Kim n.p.). That the *Harry Potter* book was produced by a collectively controlled predictive keyboard directed by comedy

---

4 Ian Bogost defines “procedural rhetoric” as the “practice of using processes [like those found in computer programs and video games] persuasively” (3), and I view the Eliza effect as a related phenomenon in which users are influenced or persuaded toward a specific belief about a computer process by a computer process.
writers, or that the “forced a bot” memes are almost certainly fake (Shane) does not undermine their status as representing the association of a particular style with computer generated aesthetic works, which has been trained by the striking and well-known aesthetics of AI projects like Google’s Deep Dream. As Jay Hathaway writes of Botnik’s work, “It has the slightly ‘off’ elements we associate with machine-generated text, but it also has the human touch that understands why that ‘off-ness’ is so funny” (n.p.), and I would argue that the same analysis holds for The Policeman’s Beard is Half Constructed.

Lamenting mainstream coverage of another story generator, Brutus, created in the late 1990s by Selmer Bringsjord and David Ferrucci, Noah Wardrip-Fruin notes the tendency of the authors, and of press coverage, to attribute creative authority to the Brutus program, despite the fact that the story-generation model is rigidly deterministic. “Each piece can only fit one place, in a manner determined by the authors before the system is set running, and no internal model of betrayal (or any other literary concept) could possibly influence the process” (Wardrip-Fruin 258). But while there does seem to be something incorrect in the claim that Brutus simulates human creative behavior, it could also be said that Closky, in writing “First Thousand Numbers,” or any authors of works derivative of “The House That Jack Built,” likewise follow a rigidly deterministic script. What makes a difference for many of these works seems to be how and when the computer-as-agent enters into the hermeneutic circle.

Some authors are very explicit about their role vis-a-vis the computer. Charles O. Hartman is consistent in thinking of his relationship with the programs he writes as occasions for “collaboration,” where computational constrained and randomly-selected contributions to a text reorient or disrupt Hartman’s hermeneutic circle: “If using randomness makes me a little more passive -- a little more obviously a judge than a creator -- that’s another similarity between me and the reader and another point of contact for the poem” (Hartman 35). Allison Parrish, by contrast, leaves the circle a bit more vague in her introduction to Articulations through the rigorous use of passive voice when describing the processes that produced the poetry in the book -- e.g. “The poems in this volume are constructed ... are processed .. then represented” until “Lines with vectors that are similar are considered to be closer to one another...” (Parrish v). This consideration, like “judgment” in Hartman’s phrase, could be the provenance of Parrish’s program which evaluates distance through cosine similarity, or it could be the contemplation of the poet who decides that these mathematical relationships express poetic value. The reader’s consideration is evoked only obliquely as the hypothetical who “may think of these lines of poetry as existing in a continuous space” or that may realize the demonstrations of “an intuitive coherence found outside the bounds of intentional semantic constraints” (publisher’s blurb, Articulations, back cover). The ironic grammatical ambivalence of these statements contrasts the elegance, wit, beauty and occasional horror one finds in the words of
Articulations, which are clearly and emphatically the work product of Parrish’s creativity, despite the fact that those words are also the words of two million lines of public domain poetry scraped from texts in Project Gutenberg.

If The Several Houses of Brian, Spencer, Liam, Victoria, Brayden, Vincent, and Alex have anything of me in them, what is my ethical responsibility to those other voices whose words I have rearranged with my computer program? How does my identity situate the ideological and political implications of appropriation?

**Authorship and Appropriation**

Kenneth Goldsmith’s 2011 book *Uncreative Writing: Managing Language in the Digital Age* describes a way of thinking about text beyond authorship and provides several examples of Goldsmith’s conceptual poetics wherein he appropriates every day texts like traffic reports and newspapers to render them poetic through the endorsement of his affordance as a poet of the avant-garde. Goldsmith’s works like *Fidget, Day* and *Traffic* are challenging, precocious, and controversial, and when I encountered *Uncreative Writing* in 2013, just as I was beginning to experiment with Twitter Bots and other forms of computational creativity, I latched onto *Uncreative Writing* as a permission slip: not because it authorizes appropriation as a valid poetic technique, but because it exploded (at least for me, unversed as I am in contemporary poetics) the idea of received formal conventions or filters for poetry.

The “why not?” ethos advocated in the book led me to create one of my first successful Twitter bots, @ROM_TXT, which scans videogame ROM files for strings of bytes that are probably (though not always) human-readable text, and simply tweets 140 characters of what it finds. My method for this bot came about as the consequence of a long-dormant research project, but tweeting random snippets of this unusual corpus and calling its output “poetry,” as I do in the bot’s bio, is my recognition that sharing this text in its barely-cooked and minimally filtered form is good enough. I didn’t need to fuss about form and rhyme or Markov-chain algorithms like I had done with other bots -- Goldsmith’s book permitted, in its way, publishing the text as it was and calling that poetry. Naturally, that warrant to publish what I call poetry on Twitter doesn’t come from Goldsmith, but Goldsmith’s writing helped me see that there was nothing stopping me.

I personalize this justification because the “me” in “there was nothing stopping me” is not some universal or generic author-self. Instead, as I have come to realize, my status as a white, cisgendered, heterosexual male professor with tenure means that I face little risk in publishing creative ideas that disrupt a status quo of poetics, at least not the same risks that anyone without any of those
identities might face. The systems in which my identities intersect are in many ways built to justify “not stopping me,” so I have tried to become more aware of this justification when reflecting on my own work, especially when that work builds off of or appropriates the works of others.

There is an ethics to appropriation, a line that can be crossed when presenting something as poetry that reveals that ideology persists in this form as well. In March of 2015, Goldsmith’s work, Michael Brown’s Body crossed that line for many people -- or at least the idea of the work crossed that line, since it was performed only once at an event at Brown University, and not recorded or broadcast in any way. While individuals in that audience have had different personal responses to the event, in which Goldsmith performed the text of Michael Brown’s autopsy report, those variations are clearly not sufficient defenses of the work, which many saw as appropriating Michael Brown’s trauma as well as the ongoing racial conflict sparked by Brown’s death. That is, where in many other cases of controversial creative work, it would be reasonable to ask that critics focus on the work itself and not their preconceived ideas about it, this is conceptual poetry, so the concept is all. The poet must own the consequences of the concept.

What this event illustrates and helps clarify for me is the inherent risk in conceptualism of what Chuck Ryback has called “avant privilege” (Ryback n.p.) and what Lillian-Yvonne Bertram, blogging for the Poetry Foundation, aligns with white privilege when she writes “Whiteness presumes accessibility to all spaces and people. That whiteness presumes transcendence into a pure and untouched imaginative or conceptual space is perhaps its most obvious and potent operational aspect” (Bertram n.p.). In other words, if Goldsmith’s conceptualism seeks to free others’ language of its social reality by calling it poetry, then it dismisses the social, embodied realities of those who generated that language in the first place.

The pedagogical energy of conceptualism’s permissiveness is clear, once students overcome their well-ingrained resistance to copying and pasting. But if algorithmic authorship is a kind of conceptualism that depends on appropriation, how does a teacher of creative coding help students understand their responsibilities and privileges when creating their own work that builds off of the labor of others?

**CONCLUSION, AND A FIRST ATTEMPT AT TEACHING CREATIVE CODING**

If, in writing this artist’s statement as an “artist,” I’ve had to confront my sense of feeling like an imposter, that is doubly true for any sense in which I can claim meaningful authority in teaching creative writing.

Earlier in this essay, I alluded to a sense of impostor syndrome with respect to me “authorship”
of *The Several Houses*..., and this anxiety also extends to my role as an instructor of a “Creative Coding” class, something I got to experience for the first time in Fall 2018.\(^5\) I have been teaching for over 16 years, but while my classes do almost always include at least one creative assignment, this was the first class I have taught with an explicit focus on creative work. My students, many of whom were coding for the first time, dealt with different anxieties, but sharing my anxieties with students helped us form a community based on mutual trust. But in reflecting on how I taught this class, especially as I think about how the class attempted to (or, in some cases, failed) to engage some of my ethical concerns about conceptualism more generally, I think it’s important to underscore some patterns I observed in my students’ creative coding processes.

In terms of programming, I had some real beginners as well as some who were already more comfortable with code in general. This diversity of experience made it hard to find projects that were the right balance for those differing skill levels, but because some students were such beginners, much of my time in class fell to helping those students debug their projects or get something that actually carried out the instructions they intended to impart to their computers. The majority of my students had no experience with Python, so one early project that required Python became a matter of copying and pasting my example code and then switching out the vocabulary. This pattern gave us the raw material for some good conversations about the nature of authorship -- is the student, as the one who selected the words, the author of their poem, or is it I who wrote the basic algorithm and syntax? This echoes the question I raised earlier about my authorial status regarding *The Several Houses*..., with one important difference: what is trivial for me (writing a short Python program to arrange words by part of speech) is an epiphany for the first-time programmer, so their sense of pride in accomplishing this task is an outcome that no philosophical dithering about agency should undermine. The way in which I worked with the ATU2035 storytelling formula and the crowd-sourced terminology from ConceptNet is similar to the way my students worked with my code examples and their respective personal inspirations and influences.

That said, teaching-as-tech support meant that I wasn’t able to lead as many high-level discussions about poetics as I’d hoped for, but it did help me realize something more important about myself. As a teacher, I try to pay more attention to which moments will become potential pivotal points for an individual student or for the whole class. In this class on Creative Coding, I noticed how excited I got when a student approached me with a project idea, but started their explanation with something like, “I have no idea if this is possible, but ....” This overture usually indicated a student with an idea worth figuring out or a problem worth solving, and in helping them go to the next steps of execution, I found how much I loved helping students solve the problems that they’ve

---

\(^5\) I can’t claim that what I did in this class was uniformly effective for my students, or that it will be at all helpful to others teaching similar courses (although most of the course materials are available on Github, just in case it is helpful)
come up with, as opposed to watching them work through the “problem” of how to adequately complete whatever they perceive I’ve set out for them in an assignment so that they can receive an appropriate grade.

Perhaps this revelation is common knowledge. This kind of relationship might be something taken as a given in more programming-intensive or maker-oriented classes, but the core of this insight revealed to me something that gets to the core of all my teaching: my job is to show students what’s possible and then help them solve the problems that stand in the way of achieving that possibility.

If *The Several Houses*... is to have any value, maybe it’s that it helped me help students believe in their own permission to be creative, not because my (or their) privileged status allows it, but because the way that *The Several Houses* extends a readymade formula exploits a creative tendency inherent in many creative arts. In this way, creative coding has provided for me and my students a window into the endless supply of readymade formulae built into language, culture, and especially digital technology, and by calling ourselves authors, we leverage creative writing-as-making so that we can better understand the ecology of content and networks of influence that invisibly inform and enable digitally mediated culture.

**Works Cited**


**Biography**

Zach Whalen is an Associate Professor at the University of Mary Washington where he teaches courses in Digital Studies. In this role, he has developed courses on electronic literature, game studies, graphic novels, transmedia fiction, and creative coding, and published several articles and book chapters in these fields. He is the co-editor of Disability in Comic Books and Graphic
Narratives (2016) with Chris Foss and Jonathan W. Gray and Playing the Past: History and Nostalgia in Video Games (2008) with Laurie N. Taylor. He is also responsible for several Twitter bots including @ROM_TXT, @HouseBudgets, and @AutoImagist.

**Coda: Source Code**

What follows is a version of the Python script that I used to generate The Several Houses of Brian, Spencer, Liam, Victoria, Brayden, Vincent, and Alex in December 2017. The original version is available in Github, and I have edited the version that appears here to include comments explaining key aspects.

```python
# -*- coding: utf-8 -*-

# This first section imports all of the many third-party Python libraries that will be necessary to create this book

import requests
import random
import re
import os
import json
import glob
import urllib
from requests_oauthlib import OAuth1
import flickrapi
import pycorpora
from textblob import TextBlob
from textblob.tokenizers import SentenceTokenizer
```
from pattern.en import conjugate, lemma, lexeme
from pattern.en import tenses, PAST, PL, parse, pluralize, singularize, quantify

from weasyprint import HTML

from PIL import Image

# These dictionary objects create the placeholders for the book’s metadata, which will be used when generating the final PDF.
concept_cache = {}
credits = {
    'nouns':[],
    'houses':[],
    'characters':[],
    'concepts':[],
    'raw_txt':'',
    'chapter_count':2,
    'chapter_titles':[],
    'character_icons':[]
}

# This function initializes the list of contributors which will be added to as the script selects concepts, relationships, and images to add to the book it is generating.
def contrib(sources):
contributors = []
for c in sources:
    contributors.append(c['contributor'].split('/')[1][-1])
return contributors

# This function creates a list of color palettes, one of which will be
# selected randomly for each chapter.
def pal():
    palettes = [
        "cf3e27","0d8a89","066598","f9800","e68900"],
        "25CAF7","BEF272","f9fa96","F89573","F44750"],
        "EA9155","EFB53A","C54400","B0946F","5D443F"],
        "820434","C84622","F6871E","F3B778","2D132E"],
        "1E2C45","A1531B","D9B32B","A11B1B","250433"
    ]

    return random.choice(palettes)

# This function loads specific API credentials like access tokens, which
# are stored in a separate file for security reasons.
def cred():
    cred = {}
    f = open("credentials")
    lines = f.readlines()
    for l in lines:
        l = l.rstrip()
cred[l.split(" = ")[0]] = l.split(" = ")[1]

return cred

# This is a general-purpose function for replacing tokens in an HTML template with values specific to the book being generated.

def tpl(template_file,outfile,data):

    if os.path.isfile(outfile):
        os.unlink(outfile)
    f = open(template_file)
    lines = f.readlines()
    f.close()

    for l in lines:
        for token in data:
            l = l.replace("<!-- " + token[0] + " -->",token[1])
    g = open(outfile,"a")
    g.write(l)
    f.close()

    return True

# This function simplifies some of the text that arrives from ConceptNet by removing leading articles.

def simpler(word):

```python
return re.sub(r“^an? |^the |^your “,”””, word, flags=re.IGNORECASE)

# This function transforms phrases into past-tense. It uses TextBlob to
# identify parts of speech and Pattern to conjugate verbs.

# Several exceptions are hard-coded because conjugation in English is
# not consistent so, for example, Pattern thinks the past tense of “lose” is
# “losed” instead of “lost.”

def pastify(phrase):
    fixed = ‘
    found = 0
    blob = TextBlob(“I would like to “ + phrase)
    for w in blob.tags[4:]
        if (“VB” in w[1] and found is 0):
            if (“heat” in w[0]):
                fixed += “heated up “
            elif (“leave” in w[0]):
                fixed += “left “
            elif (“lose” in w[0]):
                fixed += “lost “
            elif (“lie” in w[0]):
                fixed += “lay “
            elif (“bite” in w[0]):
                fixed += “bit “
            else:
                fixed += conjugate(w[0], tense=PAST) + “ “
        found = 1
```

The Many Authors of *The Several Houses of Brian, Spencer, Liam*...
else:

    fixed += w[0] + " "

return fixed

# This function transforms some of the phrases from ConceptNet to make them more specific with the definite article “the”. This reads better and follows the House that Jack Built formula more closely.

#

def specify(phrase):

    phrase = re.sub(r” an? | their | your | its | her | his “,” the “,phrase)
    do = ‘’

    if (“ the “ not in phrase):
        # check for the direct object
        chunks = TextBlob(“this is the thing that “ + phrase)
        for word in chunks.tags:

            if (“NN” in word[1]):
                do = word[0]

        return phrase.replace(do,”the “ + do,1)
    phrase = re.sub(do,”the “ + do,phrase)

    return phrase
# This is a function for retrieving some concepts from ConceptNet. It expects to have a starting word, a type of relationship, and a direction (a sense of whether this word is acting or being acted upon, in my case).

```python
def get_some(word, rel, direction, number):
    global concept_cache

    if word in concept_cache.keys():
        return concept_cache[word]
    else:
        url = 'http://api.conceptnet.io/query?node=/c/en/' + word + '&rel=/r/' + rel + '&limit=' + number
        objects = requests.get(url).json()

        things = []
        for thing in objects['edges']:
            directions = simpler(thing[direction]['label']).lower()
            if (simpler(word) not in directions):
                contributors = contrib(thing['sources'])
                things.append(((directions, contributors)) # make this a tuple

        concept_cache[word] = things

    return things
```

The Many Authors of The Several Houses of Brian, Spencer, Liam... 19
# This function contains the algorithm that is the engine driving the whole thing. I have explained this in more detail in a blog post: http://www.zachwhalen.net/posts/a-python-script-that-writes-800-page-childrens-books/

# In short, this works by building plausible relational chains of causality among concepts, moving from word to word by looking for plausible words to add, then with each of those plausible words, looking for plausible connections, and so on, until at least one chain reaches the specified “depth” variable.

def stack(seed, depth):
    
    db = []
    completed = []
    og = seed

    for d in range(depth):

        print "Depth is now " + str(d) + " and the db is at " + str(len(db))

        if (d is 0):
            actions = get_some(seed,"CapableOf","end","1000")

            # get some actions
            for act in actions:
                blob = TextBlob(seed + " can " + act[0]) # adjust for tuple here

                # make sure that there’s a verb besides “is"
                pos = []
for w in blob.tags:
    if (not re.match("be", lemma(w[0]))):
        pos.append(w[1])

if (re.match(blob.tags[-1][1], "NN") and "VB" in pos and act[0].split(" ")[0] not in ['to', 'near']):
    # print "Adding " + act
    db.append([act])
    #print db
else:

    threadsnap = len(db)

    #print "DB length is " + str(threadsnap)
    #print db
    for s in range(threadsnap):
        #row = db[s]
        thread = db[s][d - 1]
        #print "Working on " + str(thread)

        seed = thread[0].split(" ")[1][-1]

        new_tails = []

        if (len(seed) > 0 and seed not in 'house'):
actions = get_some(seed,"CapableOf","end","1000")

for act in actions:
    blob = TextBlob(seed + " can " + act[0]) # adjust

pos = []
    for w in blob.tags:
        if (not re.match("be",lemma(w[0]))):
            pos.append(w[1])

    if (re.match(blob.tags[-1][1],"NN") and "VB" in pos and act not in db[s] and act[0].split(" ")[0] not in ['to','near'] ):

        # add it
        #print "A new tail action: " + act
        new_tails.append(act)

if (len(new_tails) > 0):
    row = list(db[s])

    for n in range(len(new_tails)):
        if (n > 0):
            # the new phrase is new_tails[n]

            new_row = list(row)
            new_row.append(new_tails[n])
            db.append(new_row)

else:
db[s].append(new_tails[0])

else:
    if (len(db[s]) <= d):
        db[s].append('”')
    else:
        db[s][d] = ‘”'

print “Before cleanup, db is at “ + str(len(db))

cull = []

for c in range(len(db)):
    if (len(db[c][-1]) > 0):
        # save any that end in houses
        if (“house” in db[c][-1][0].split(“ “)[0][-1]):
            completed.append(db[c])
            cull.append(c)
    else:
        cull.append(c)

db = [v for i, v in enumerate(db) if i not in cull]

trim = []

for r in range(len(db) - 50):

The Many Authors of The Several Houses of Brian, Spencer, Liam...
```python
totrim = random.randrange(0, len(db))
while (totrim in trim):
    totrim = random.randrange(0, len(db))
    trim.append(totrim)

db = [v for i, v in enumerate(db) if i not in trim]

print "After cleanup, db is at " + str(len(db))

# pick one of the longest complete chains, prepend the seed, and return
if (len(db) > 0 and len(completed) > 0):
    content = completed[-1]

    # add to the credits here with information from tuple
    for c in content:
        credits['concepts'].append(c[1])
    content.insert(0, str(og))
    return content
else:
    return 0

# This function searches the Noun Project for icons related to each noun in the chain that has been selected to build this particular book.
def get_icons(noun, fallback):
    save_as = ""
```

keyword = re.sub(r" \$","",noun)

# first check the cache
if (os.path.isfile("icons/" + keyword.replace(" ","+") + ".json")):
    print "Loading from file"
    # proceed to load it from file

    icon_file = open("icons/" + keyword.replace(" ","+") + ".json").read()
    icons = json.loads(icon_file)
    #icon_file.close()
    save_as = keyword

elif (os.path.isfile("icons/" + keyword.split(" ")[\-1] + ".json")):
    icon_file = open("icons/" + keyword.split(" ")[\-1] + ".json").read()
    icons = json.loads(icon_file)
    save_as = keyword.split(" ")[\-1]
else:
    print "Getting from API"
    # set up nounproject auth
    credentials = cred()
    auth = OAuth1(credentials[‘noun_key’],credentials[‘noun_secret’])

    # the api endpoint
    endpoint = “http://api.thenounproject.com/icons/” + urllib.quote(keyword)
response = requests.get(endpoint, auth=auth)

print response.status_code

if (“404” not in str(response.status_code)):
    icons = response.json()
    save_as = keyword

elif (len(keyword.split(“ “)) > 1):
    # try again
    # should put a file check in here
    endpoint = “http://api.thenounproject.com/icons/” + urllib.quote(keyword.split(“ “)[-1])
    response = requests.get(endpoint, auth=auth)

if (“404” not in str(response.status_code)):
    icons = response.json()
    save_as = keyword.split(“ “)[-1]
else:
    icons = json.loads(“icons/object.json”)
    save_as = “object”

else:
    # icons = json.loads(“icons/thing.json”)
    # save_as = “thing”

    icons = get_icons(fallback, fallback)
    save_as = fallback
with open("icons/" + save_as.replace(“,,”, “”) + ".json”, "w") as outfile:
    json.dump(icons, outfile)

    return icons

# This function finds an appropriately-licensed image from Flickr using the keyword “house”, and applies some Fred Weinhaus’s “watercolor” Imagemagick script. (See http://www.fmwconcepts.com/imagemagick/index.php)
def get_flickr_image(keyword, chapter_number):

    new_fn = ‘images/’ + keyword + ‘-’ + str(chapter_number) + ‘-watercolor.jpg’

    if (not os.path.isfile(new_fn)):

        credentials = cred()
        api_key = credentials[‘flickr_key’]
        api_secret = credentials[‘flickr_secret’]

        flickr = flickrapi.FlickrAPI(api_key, api_secret, format="parsed-json")

        photos = flickr.photos.search(text=keyword, license=’4,5,9,10’, sort=’relevance’, per_page=’300’)  

        photo = random.choice(photos[‘photos’][‘photo’])
        photo_info = flickr.photos.getInfo(photo_id=photo[‘id’])

        #license = flickr.photos.getInfo(photo_id=photo[‘id’])[‘photo’]

The Many Authors of The Several Houses of Brian, Spencer, Liam...
['license']

owner = flickr.photos.getInfo(photo_id=photo['id'])['photo']
[‘owner’][‘username’]

print photo_info

if(len(photo_info['photo'][‘owner’][‘realname’]) > 0):
    photog = photo_info['photo'][‘owner’][‘realname’]
else:
    photog = photo_info['photo'][‘owner’][‘username’]

credits[‘houses’].append((photo[‘id’],photo_info[‘photo’][‘license’],photog,photo_info[‘photo’][‘title’][‘_content’],photo_info[‘photo’][‘urls’][‘url’][0][‘_content’])

fullsize = flickr.photos.getSizes(photo_id=photo[‘id’])['sizes'][‘size’][-1][‘source’]

fn = fullsize.split(“/”)[-1]
iid = fn.split(“.”)[0]

#new_fn = fn.split(“.”)[0] + “-watercolor.jpg”

os.system("wget -P images “ + fullsize)  

# convert images/larger-flickr.jpg -resize “800x800^” -gravity center -crop 800x800+0+0 +repage -brightness-contrast 50x0 images/smaller-flickr.jpg

os.system("convert images/" + fn + “ -resize “800x800\” -gravity center -crop 800x800+0+0 +repage -brightness-contrast 50x0 images/” + fn)

os.system("/watercolor -s 25 -e 5 -m 50 -c 0 images/" + fn + “ “ + new_fn)
return new_fn

# This function finds a particular icon, colors it, and applies appropriate CSS to insert the icon into the appropriate place in the book.

def get_icon(keyword, color, fallback="thing"):  
    global credits  
    icons = get_icons(keyword, fallback)  
    #print icons

    icon_url = 0
    while(not icon_url):
        icon = random.choice(icons['icons'])

        if (“attribution_preview_url” in icon):
            print “has png”
            icon_url = icon[‘attribution_preview_url’]

            credits[‘nouns’].append((icon[‘id’],icon[‘attribution’]))

        # do I already have this one?
        if (not os.path.isfile(“images/” + icon[‘id’] + “.png”)):
            # save it
            get = os.system(“wget -P images/ -O images/” + icon[‘id’] + “.png “ + icon_url)
            #print “get = “ + get
            # crop it

The Many Authors of The Several Houses of Brian, Spencer, Liam...
```python
img = Image.open("images/" + icon['id'] + ".png")
t = img.crop((0,0,700,600))
t.save("images/" + icon['id'] + ".png")

# do I have the right color?
color_fn = "images/" + icon['id'] + "-" + color + ".png"
print "The colorized file name should be " + color_fn
if (not os.path.isfile(color_fn)):
    # convert 8668.png -fuzz 100% -fill "#e68900" -opaque
    black test.png
    print "coloring the icon "
    os.system("convert images/" + icon['id'] + ".png " + "-fuzz 90% -fill \""#""\" + color + "\" -opaque black " + color_fn)

    top = "%.1f" % random.uniform(-0.5,7.0)
    left = "%.1f" % random.uniform(-0.5,7.0)
    width = "%.1f" % random.uniform(0.2,2.8)
    rotation = "%.1fdeg" % random.uniform(-30,30)

    position = "top:" + top + "in;" + "left:" + left + "in;"

    return u'<img src="../' + color_fn + '\" class="' + keyword + '\" style="transform:rotate(' + rotation + ');position:absolute;width:' + width + 'in;" + position + '\" />'

    # This function converts definite articles into the appropriate indefinite article, when necessary.
```

# This function converts definite articles into the appropriate indefinite article, when necessary.
def a(phrase):
    if (re.search(r" the [aeiou]",phrase)):
        new_phrase = phrase.replace(" the ", " an ").replace(" for an ", " for ")
    else:
        new_phrase = phrase.replace(" the ", " a ").replace(" for a ", " for ").replace(" a water", " water").replace(" a sun", " the sun")
    return new_phrase

# This function generates the book’s title by quantifying and listing the proprietors of each chapter’s house.

def book_title ():
    global credits
    people = list(credits['characters'])
    people[-1] = "and " + people[-1]
    return "The " + quantify("house",amount=len(credits['chapter_titles'])) + " of " + " , ".join(people)

# This function is, in addition to stack(), the other main workhorse of generating this book. It makes several key decisions and generates the HTML templates which will later be converted into PDF.

def prepare_chapter(content,startpage,chapter_number):

    global credits
print content

# put the list from stack() in reverse
ordered = list(content[::-1])

# this will hold the chapter text as it accumulates
chapter = ''

# this chooses a color palette for this chapter
colors = pal()

# pick a name for our main character
jack = random.choice(pycorpora.get_file("humans","firstnames")['first-
Names'])
credits['characters'].append(jack)

# get an image for this character
jack_icon_first = get_icon(jack,random.choice(colors),random.choice(["m
an","girl","boy","woman","baby","child","grandmother","dude"]))

jack_icon = re.sub(r"style=".+?",","\n",jack_icon_first)

credits["character_icons"].append(jack_icon_first)

# chapter object
chapter_object = content[0]
# make a chapter title
# this will make a title for the chapter
chapter_title = str( jack + 
  " and the " + 
  str(chapter_object) + 
  " that " + 
  a(specify(pastify(content[1][0]))) )

credits[‘chapter_titles’].append((chapter_title, startpage + 3))

# loop through each concept in the stack
for page in range(len(ordered)):

  print "Working on page " + str(page)


# the page number
page_number = str( startpage + ((page * 2) + 2) )

# a color for this concept
color = random.choice(colors)

# I don’t remember what this does but it’s probably important later
if (page == len(ordered) - 1):
the_concept = chapter_object

else:
    the_concept = pastify(specify(ordered[page][0]))

# isolate the current object
the_thing = the_concept.split(“ the “)[-1]

# find an icon. This returns an <img> tag for the icon
the_icon_first = get_icon(the_thing,color)
the_icon = re.sub(r"style=".+?"","",the_icon_first)

if (page == 0): # at the beginning of the chapter

    # make the blank page
    tpl("templates/template.html", "pages/" + str(startpage).zfill(5) + "1.html",[])

    # make the chapter title page
    print “Making the title page “
    tpl("templates/chaptertitlepage.html",
        "pages/" + str(startpage + 1).zfill(5) + "r.html",
        [("chapter_number",str(chapter_number)),("chapter_title",chapter_title),("character_name",jack),("character_icon",str(jack_icon))])
next_concept = ""

# start the chapter string
chapter = "that " + jack + " " + random.choice(["built.","built.","built.","built.","built.","built of brick.","divided into several rooms.","found in a neighborhood.","located on an estate."])}

else:
    # I don't know why this is going backwards?
    next_concept = "<span> that " + pastify(specify(ordered[page - 1][0])).split(" the ")[0] + " the </span>"

page_content = [
    ("pn",str(int(page_number) - 6)),
    ("chapter_content",chapter),
    ("icon",the_icon),
    ("first_line","This is the <span class='page-object' style='color:"") + color + "'>" + the_thing + "</span>" + next_concept)
]

tpl("templates/template.html","pages/" + page_number.zfill(5) + "1.html",page_content)

if (page == 0):
    if (not os.path.isfile("images/house-" + str(chapter_number) + "-watercolor.jpg")):
        get_flickr_image("house",chapter_number)
# This function is a controller for prepare_chapter() and other key functions. As my original comment notes, this function “actually makes the thing.”

```python
def assemble():
    # actually make the thing
    global credits
    credits = {
        'nouns': [],
        'houses': []
    }  
```
page_counter = 6

# first, make the chapters
animals = pycorpora.get_file(“animals”,”common”)[‘animals’]

for chapter_counter in range(credits[‘chapter_count’]):
    result = 0
    while (result == 0):
        animal = random.randrange(0,len(animals))
        result = stack(animals[animal], 55)
        del animals[animal]

    a_chapter = prepare_chapter(result,int(page_counter) + 2,int(chapter_counter) + 1)
    page_counter = a_chapter
# prepare the frontmatter

# 1r = frontcover
# 2l = blank
# 3r = title page
# 4l = dedication
# 5r = toc
# 6l = blank
# 7r = introduction
# 8l = blank, chapter 1 page 0

booktitle = book_title()

print "This book is called " + booktitle

tpl("templates/frontcover.html","pages/00001r.html",["book_title",booktitle])

# blank page

tpl("templates/template.html","pages/00002l.html",["",""])

# title page

tpl("templates/titlepage.html","pages/00003r.html",["book_title",booktitle])

# dedication page
kid_icons = '<div id="kids">
for kid in ["Cecily","Daniel","Serena","Wendy"]:  
a_kid = re.sub(r"style=".+?",","");get_icon(kid,"333333")

kid_icons += a_kid
kid_icons += '</div>'
tpl("templates/dedication.html","pages/00004l.html",["kids",kid_icons])

# toc -- this one has to be a bit more manual
toc = ''
for ch in range(len(credits["chapter_titles"])):
    toc_string = '<div class="toc-entry"><span>Chapter ' + str(ch + 1) + '</span><span>' + credits["chapter_titles"][ch][0] + '</span><span>' + str(credits["chapter_titles"][ch][1] - 8) + '</span></div>

toc += toc_string

toc += '<div class="toc-entry"><span>&nbsp;</span><span>Credits</span><span>' + str(int(page_counter) -3) + '</span></div>'
tpl("templates/toc.html","pages/00005r.html",["toc",toc])

# blank page
tpl("templates/template.html","pages/00006l.html",["",""])

# introduction
housecount = quantify(“house”,amount=len(credits[“chapter_titles”]))
peoplecount = quantify(“person”,amount=len(credits[“characters”]))
# make a The End

tpl("templates/last_page.html","pages/" + str(int(page_counter) + 2).zfill(5) + ".html",[("character_names", "\n".join(list(credits['characters'])))])

tpl("templates/the_end.html","pages/" + str(int(page_counter) + 3).zfill(5) + "html",["character_icons","\n".join(list(credits['character_icons'])))])

# make the credits

make_credits(credits,str(int(page_counter) + 2))

print "Generation complete"

# This function makes and appends credits.

def make_credits(credits,page_number):

    # actually first make a blank verso page
    tpl("templates/credits.html","pages/" + str(int(page_number) + 2).zfill(5) + "html",["\n","\n"])

    license = {
'4': 'CC BY',
'5': 'CC BY-SA',
'9': 'CC 0',
'10': 'Public Domain'
}
house_cred = ''

# first the houses
for h in credits['houses']:
    house_cred += "<div class='housecred'>" + h[3] + "" by Flickr User " + h[2] + " " + license[str(h[1])] + "</u></div>"

house_credit_content = [
    ('credits_header', '<h2>Credits</h2>'),
    ('credits_content', '<h3>Houses</h3>' + house_cred.encode('utf-8')),
    ('page_number', str(int(page_number) - 5))
]

tpl('templates/credits.html', 'pages/' + str(int(page_number) + 3).zfill(5) + '.html', house_credit_content)

# then the noun icons in pages with two 25-item columns
# each is a tuple of an id and a attribution line
nounlist = credits['nouns']
chunked = [nounlist[i:i+50] for i in xrange(0,len(nounlist),50)]
for chunk in range(len(chunked)):
    chunk_counter = 1
    chunk_string = ''
    for noun in chunked[chunk]:
        noun_string = '<div class="noun-credit"><img src="../images/" + noun[0].encode('utf-8') + ".png" /><span> " + noun[1].encode('utf-8') + "</span></div>

        chunk_string += noun_string

    this_header = "<h3>Icons</h3>
    if (chunk >= 1):
        this_header = "<h3>Icons (continued)</h3>

    tpl("templates/credits.html","pages/" + str(int(page_number) + chunk + 4).zfill(5) + ".html",["credits_header",this_header],"credits_content",chunk_string),["page_number",str(int(page_number) + chunk - 2)])

# finally all the concepts
contributors = {}
for c in credits['concepts']:
    for contributor in c:
        contributors[contributor] = ' '  

con_string = "", ",join(contributors.keys())

    tpl("templates/credits.html","pages/" + str(int(page_number) + chunk + 5).zfill(5) + ".html",["page_number",str(int(page_number))}
# This function simplifies the list of contributors so that each is only listed once.

def chunk(thelist, size):
    nouncredits = {}
    for n in thenouns:
        nouncredits[n[0]] = n[1]
    for i in range(0, len(thelist), 50):
        yield l[i:i + n]

# This function runs weasyprint to generate a PDF out of all of the HTML templates.

def make_pdf():
    print "Making PDF "
    pages = glob.glob("pages/*.html")

    print "Converting individual pages "

    for p in pages:
        pn = p.split("/")[1].split(".")[0]
        nf = "pdfs/"+ pn.zfill(5) + ".pdf"

        HTML(p).write_pdf(nf)
print "Combining them all"

os.system("pdfk pdfs/*.pdf cat output output.pdf")

print "I think it’s done!"

assemble()
make_pdf()