# Learning Latent Student Design Decisions in Python Programming Classes

**Elena Glassman** ELG@MIT.EDU
MIT CSAIL, 32 Vassar St., Cambridge, MA 02139 USA

## 1. Introduction

Programming courses produce a collection of student solutions for each programming exercise. The solutions to each exercise vary along many dimensions, including bugs, naming, syntax, and semantics. The distribution of solutions along these dimensions reflect student prior knowledge, teacher explanations, and common misconceptions.

Solutions can be thought of as a collection of design decisions along the various dimensions of solution variation. Some design decisions are mutually exclusive, e.g. looping over a particular array with a for or a while, some decisions are correlated with one another, and some decisions are completely independent. Each new solution submitted by a student may introduce a novel design decision into the collection. Every possible combination of design decisions (that arent mutually exclusive) may be observed as new student solutions are submitted.

Hundreds or thousands of student solutions become painful or prohibitively exhausting to review by hand. Automated testing is a noisy predictor of code quality. Students get little or no expert feedback on the design decisions they made in their solutions.

The paper presents a potential resolution to this problem by describing how to factorize student solutions into combinations of student design decisions. Rather than reviewing individual solutions from the collection, teachers could review student design decisions, and written feedback could be propagated back to all the students which express that design decision.

## 2. Dataset

The dataset of solutions is collected from 6.00x, an introductory programming course in Python that was offered on edX in fall 2012. This paper focuses on Python solutions from one exercise problem, which will be the running example throughout this paper. Specifically, it is a collection of 3875 student solutions to the problem of writing a function `iterPower(base,exp)` that iteratively exponentiates base to an exponent exp. Figures 1, 2, and 3 provide some concrete examples of the `iterPower` solutions.

```
def iterPower(base, exp):
    '''
    base: int or float.
    exp: int >= 0

    returns: int or float, base^exp
    '''
    # Your code here
    if exp <= 0:
        return 1
    else:
        return base * iterPower(base, exp - 1)
```

*Figure 1.* Example of a recursive student solution.

```
def iterPower(base, exp):
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result
```

*Figure 2.* Example of a `while`-based student solution.

In early pilot studies, python programming teachers given the same set of solutions partitioned the space of solutions in a variety of ways, producing different numbers and/or compositions of clusters. Since teachers found multiple reasonable clusterings, the pilot results can be explained by one or both of the following reasons: (1) Teachers have different internal clustering metrics. (2) Solutions can reasonably belong to multiple clusters. While (1) is probably true as well, this paper focuses on addressing (2).

## 3. Methods

### 3.1. Features

Learning latent variables has been applied in a variety of contexts. For processing text, it is common to treat documents as exchangeable as well as treat the words within each document as exchangeable. This matches the generative model of Latent Dirichlet Allocation (LDA) (Blei et al., 2003). Topics, i.e., distributions over words, are learned, as well as the distributions over topics found in each document.

A similar process could be applied to programs. Consider each Python program as a separate "document", and the "words" in each document are the tokens that a Python

```
def iterPower(base, exp):
    iter = exp
    result = 1
    while iter > 0:
        result = result * base
        iter = iter - 1
    return result
```

*Figure 3.* Example of a `while`-based student solution, where the student has not modified any input arguments, i.e., better programming style.

parser would produce, i.e., the nodes of an Abstract Syntax Tree including language keywords, operators, variable names, the values of primitive times, and the names of functions called. However, given that the datasets currently available for this work range from hundreds to a few thousands of solutions instead of millions of documents, there may not be enough data to infer reasonable cross-program topics from this token-level data.

Fortunately, programs are also executable. Features from dynamic analysis can complement static token-level features, as was demonstrated in (Kim et al., 2015). Of particular interest in this work are the features produced by the OverCode program analysis method (Glassman et al., 2015).

**Common Variables** The OverCode (Glassman et al., 2015) analysis pipeline was initially developed to deduplicating Python programs that differ only by variable names, statement order, formatting, and comments. In the process of this deduplication process, OverCode executes all programs on a common set of test cases and records the sequence of values taken on by each variable in the program.

Sequence of variable values observed while executing `iterPower(5,3)` as defined in the following figures:

**Figure 1**
- `exp: 3, 2, 1, 0, 1, 2, 3`
- `base: 5`

**Figure 2**
- `exp: 3, 2, 1, 0`
- `base: 5`
- `result: 1, 5, 25, 125`

**Figure 3**
- `exp: 3`
- `base: 5`
- `iter: 3, 2, 1, 0`
- `result: 1, 5, 25, 125`

This sequence of values taken on by each variable in each program becomes a signature, i.e., the key to recognizing semantically equivalent variables across programs. OverCode assumes that variables in different programs that transition through the same sequence of values on the same test cases are in fact fulfilling the same role in the program and can therefore be considered the same **common variable** uniquely defined by that sequence of values.

*Table 1.* Variable-by-Solution Matrix for Programs, where variables are uniquely identified by their sequence of values while run on a set of test case(s)

| PROGRAM | 5 | 1,5,... 25,125 | 3,2,1,0, ...1,2,3 | 3,2,1,0 | 3 |
|---|---|---|---|---|---|
| FIG. 1 | 1 | 1 | 1 | 0 | 0 |
| FIG. 2 | 1 | 1 | 0 | 1 | 0 |
| FIG. 3 | 1 | 1 | 0 | 1 | 1 |

In the previous examples, the input argument `base` would be considered a common variable found in all three programs, but the input argument `exp` would not be. The variable `result` would also be considered a common variable shared across just the definitions in Figure 2 and Figure 3. This allows us to distinguish between programs that calculate the answer in distinct ways, without getting bogged down in discrimating between the low-level syntax-based design decisions.

In order to focus on identifing higher-level design decisions about how to compute an answer, the programs in the dataset were represented as bags of variables instead of bags of tokens. Textual documents can be represented as a $W \times N$ term-by-document matrix of counts, where $W$ is the vocabulary size across all documents and $N$ is the number of documents. This is one way to represent a document as a set of word counts, i.e., how many times each word appears in each document. The corresponding matrix for a bag-of-variables approach to the programs in Figures 1 through 3 produced by executing `iterPower(5,3)` is shown in Table 1.

Note that, while the entries in Table 1 only take on values 0 or 1, more complicated definitions may have $n$ instances of, e.g., a variable that takes on the sequence of values `3,2,1,0`. In that case, there would be an $n$ in the `3,2,1,0` for the row corresponding to that solution. In other words, true to the assumptions made by LDA, these are occurrence counts, not binary indicators.

### 3.2. Model Choice

As new solutions are submitted by students, the number of observed distinct student design decisions may grow without bound, and each solution may be a mixture of multiple design decisions. Therefore, a Hierarchical Dirichlet Process (Teh et al., 2012) for learning these underlying design decisions may be a good model choice for this data. However, since some design decisions may be correlated with each other, it may be even more accurate to replace the (conveniently conjugate) Dirichlet prior with a (non-

conjugate) logistic normal prior that can model correlated topics in addition to uncorrelated topics. This would be a non-parametric version of the Correlated Topic Model (Blei & Lafferty, 2006). If, as is likely, the chosen features can belong to multiple design decisions, it may be useful to add in elements of the Indian Buffett Process (Ghahramani & Griffiths, 2005).

While these models' assumptions may better match the reality of this dataset of student programs, LDA (Blei et al., 2003) as implemented in the Gensim toolbox (Řehůřek & Sojka, 2010) was chosen as the model to evaluate. The performance of the more sophisticated models will be explored in future work. Simpler models and transformations, like TF-IDF and pLSA/pLSI (Hofmann, 2000) were not considered because they reveal little in the way of inter- or intra-solution statistical structure and are not a proper Bayesian models, respectively.

### 3.3. Procedure

All 3875 student solutions were run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produced a set of 977 deduplicated Python programs and a set of features for each program, including which variable sequences were observed during execution. Another script turned this output into a variable-by-solution matrix for the 977 deduplicated programs, which were then fed into various models for analysis. The learned latent variables, e.g., LDA "topics," were then inspected, since perplexity and held-out likelihoods are not necessarily good proxities for human interpretability (Chang et al., 2009). Interpretability is the only end goal, because the end result will be directly visualized within a teacher-facing user interface.

## 4. Results

Since LDA does not infer the number of topics $K$ from the data itself, the behavior of LDA on the data was collected for two values of $K$: 25 and 100.

### 4.1. Fitting 100 LDA Topics

As shown in Figure 5, a small number of topics were inferred to be present in many dedpulicated solutions when K is at a larger setting, i.e., $K = 100$. Here are solutions associated with some of these "popular" topics in the dataset:

($K = 100$) Deduplicated solutions containing a popular topic:

```
def iterPower(base, exp):
    result = base
    if exp == 0:
        result = 1
    while exp > 1:
        result *=base
        exp -=1
    return result

def iterPower(base, exp):
    result = base
    if exp == 1:
        result = base
    if exp == 0:
        result = 1
    while exp > 1:
        result *= base
        exp -= 1
    return result
```

($K = 100$) Deduplicated solutions containing a different popular topic:

```
def iterPower(base, exp):
    result = 1
    while exp > 0:
       result = result * base
       exp = exp - 1
    return result

def iterPower(base,exp):
    result=1
    while exp>0:
        exp-=1
        result=base*result
    return result
```

The two topics above correctly tease apart the difference between `while`-based solutions with extra (unnecessary) conditional statements from those without them.

### 4.2. Fitting 25 LDA Topics

($K = 25$) Solutions containing four different popular topics:

```
def iterPower(base, exp):
    result = 1
    while exp > 0:
       result = result * base
       exp = exp - 1
    return result

def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

These two deduplicated solutions are examples of the many solutions associated with all the first four most popular topics. Popularity, in this case, was determined by whether the topic was associated with at least 500 deduplicated solutions (see Figure 4 for details). The first two of the popular topics overlap to the point that they are assigned to the same deduplicated solutions 578 times; they are not assigned to the same deduplicated solution only 101 times.

For additional context, when $K = 25$, the first recorded run of LDA inferred that each solution was a mixture of 5 different topics, on average.[1] For comparison, when $K = 100$, each deduplicated solution was found to be a mixture of only 1.2 topics, on average.

($K = 25$) Solution containing a different popular topic:

```
def iterPower(base,exp):
    result=1
    i=0
    while i<exp:
        result*=base
        i+=1
    return result
```

```
def iterPower(base,exp):
    if exp==0:
        return 1.0
    b=exp
    result=base
    while b>1:
        result*=base
        b-=1
    return result
```

These two groups of solutions above containing some different popular topics correctly tease apart the difference between using the input argument `exp` as a counter and creating a separate variable as a counter, so as not to modify function inputs. This can be considered a superior software development choice.

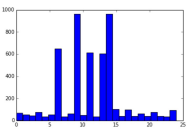

*Figure 4.* Histogram of topic assignments for 977 deduplicated solutions, when LDA's maximum number of topics is set to 25.

[1] Subsequent runs show that this average number of topics associated with each deduplicated solution, and the concentration of deduplicated solutions per topic, is highly variable.

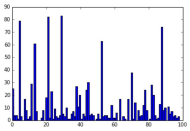

*Figure 5.* Histogram of topic assignments for 977 deduplicated solutions, when LDA's maximum number of topics is set to 100.

## 5. Discussion

LDA applied to a variable-by-solution matrix is a promising method for factoring student solutions into their constituent design decisions. LDA is not the most correct model for this dataset, e.g., it cannot model interdependent (correlated) decisions that are very likely to occur in Python solutions, but inspection of the latent topics clearly reflects the design decisions teachers might want to comment on in this particular dataset. However, the choice of topics has non-trivial consequences in terms of topic clarity. This is evidence that the additional effort of finding or implementing HDP is worth exploring.

In the future, a generative model specifically for this kind of dataset may be worth designing and building around. Encoding solutions in a variable-by-solution matrix only allows the mixture model to separate solutions based on variable behavior. Variable behavior carries a lot of information about how a student approached computing the solution, but this representation ignores the important differences between solutions using, e.g., idiomatic Python syntax vs. tangled syntax that, by some "miracle," still works. A pedagogical tool that supports teachers at both levels would be ideal. Consider the plate model of LDA. If one adds one more level of hierarchy, as illustrated in Figure 6, then $\theta$ can be the distribution over combinations of variables in a solution and $\psi$ can be the distribution over the possible syntax to create the execution behavior that defined those variables. Finally, what is observed, $w$, is a set of lines of syntax that manipulate the variables described by $\theta$. Future work includes exploring whether inference based on this generative model is tractable.

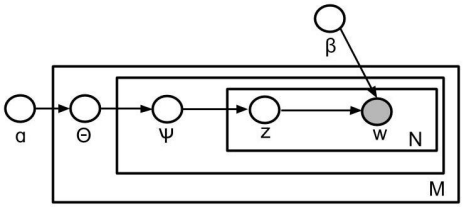


*Figure 6.* Alternative plate model for modeling Python solutions

## Acknowledgments

The author is grateful for the clarifications provided by both Finale Doshi-Velez and Tamara Broderick.

## References

Blei, David and Lafferty, John. Correlated topic models. *Advances in neural information processing systems*, 18: 147, 2006.

Blei, David M, Ng, Andrew Y, and Jordan, Michael I. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

Chang, Jonathan, Gerrish, Sean, Wang, Chong, Boyd-Graber, Jordan L, and Blei, David M. Reading tea leaves: How humans interpret topic models. In *Advances in neural information processing systems*, pp. 288–296, 2009.

Ghahramani, Zoubin and Griffiths, Thomas L. Infinite latent feature models and the indian buffet process. In *Advances in neural information processing systems*, pp. 475–482, 2005.

Glassman, Elena L, Scott, Jeremy, Singh, Rishabh, Guo, Philip, and Miller, Robert C. Overcode: visualizing variation in student solutions to programming problems at scale. *Transactions on Computer-Human Interaction*, 2015.

Hofmann, Thomas. Learning the similarity of documents: An information-geometric approach to document retrieval and categorization. 2000.

Kim, Been, Glassman, Elena, Johnson, Brittney, and Shah, Julie. ibcm: Interactive bayesian case model empowering humans via intuitive interaction. 2015.

Řehůřek, Radim and Sojka, Petr. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

Teh, Yee Whye, Jordan, Michael I, Beal, Matthew J, and Blei, David M. Hierarchical dirichlet processes. *Journal of the american statistical association*, 2012.