Linguistic-first approach to learning Python for natural language generation: Problem breakdown to pseudocode

FREE

John Blake ✉; Kazuma Tamura; Evgeny Pyshkin

Check for updates

View Online   Export Citation   CrossMark

02 December 2023 08:02:39

Linguistic-first approach to learning Python for natural language generation: Problem breakdown to Pseudocode

John Blake,[a)] Kazuma Tamura, and Evgeny Pyshkin

University of Aizu, Aizuwakamatsu, Japan

[a)]Corresponding author: jblake@u-aizu.ac.jp

Abstract. Participants in an elective course on natural language generation developed their ability to program in Python through a linguistic-first, problem-based approach. The primary aim of the course was to create a text generation program, but in doing so, students achieved the secondary aim of increasing their mastery of Python. The course started with a thorough linguistic analysis of the genre of the target language, using both top-down and bottom-up approaches. This served as the basis for the development of a set of guiding principles. These principles were then used to develop pseudocode, which, in turn, served as the foundation for the initial draft of the program. Lessons learned include the importance of aligning aims and assessment criteria, and providing learners with space to struggle.

INTRODUCTION

This paper describes a linguistic-first approach to learning Python for natural language generation that was used in an elective course for computer science majors studying at a public university in rural Japan. In introductory programming courses for Python, the problems and exercises set tend to be designed to enable learners to develop particular predetermined skills and apply various algorithms. This serialistic sequential approach is well-founded in the literature, and closely follows the ideas of Bloom [1] and his ideas of mastery learning [2]. One downside to such approaches, however, is that learners may begin at different starting points in terms of both knowledge and skills, progress at different rates, and struggle with different aspects [3]. Courses tend to be designed as one-size fits all with learners having to cope with the generic design.

Problem-based approaches have been used to increase motivation and focus learners on computational thinking. The linguistic-first approach is a language-orientated problem-based approach. In this course, learners are focused on solving a linguistic problem rather than on mastering any particular aspects of programming. Although the course adopts a non-prescriptive approach to programming [4], learners are introduced to features that may prove useful, such as regular expressions and dictionaries. There is no necessity to use any of the features introduced, though. The problem-focused nature of the course releases students from the pressure of following a strict syllabus, and puts the students in charge of deciding what they need to learn in order to solve the problem. For example, novice programmers may harness string processing to solve the problem while more advanced programmers may make use of more sophisticated semantic and syntactic analyses using, for example, statechart and conceptual graph models [5].

The undergraduate students taking this course are Japanese, but the medium of instruction is English as the course is taught using Content and Language Integrated Learning (CLIL), in which students learn the subject content and language simultaneously. This credit-bearing elective course is offered within a task-based curriculum [6]. Students enrolled have already completed compulsory introductory courses in programming in C and Java. Based on the user profiles of participants enrolled in previous elective courses, it was expected that most students would have little to no experience in Python. Some may have already studied an elective course in C++, and some may have taught themselves how to program in Python. At the outset of the course, students classified themselves as beginner, intermediate or advanced users, with self-declared beginners comprising approximately 80% of participants.

This paper shows how a thorough linguistic analysis using both top-down and bottom-up approaches [7] served as the basis for the development of a set of guiding principles. Indeed, from their programming classes, the students know that in the process of problem breakdown, top-down and bottom-up designed strategies are rarely used in their pure form, but in combination as Figure 1 illustrates. Real-world software problems usually have many unique elements to be elicited during top-down decomposition; however, even small-scale software is still based on many reusable components such as external libraries, successful algorithms, or design patterns.

To expedite the process, results of linguistic analyses were presented in summary forms to the learners. Based on the results of the analyses, guiding principles were drafted. These principles were then expanded into instructions. Some students then developed the instructions into pseudocode and then a Python program while others omitted the pseudocode step and opted to create the Python program directly. Although described sequentially in this paper, the process from linguistic analysis to writing the Python program was cyclical.
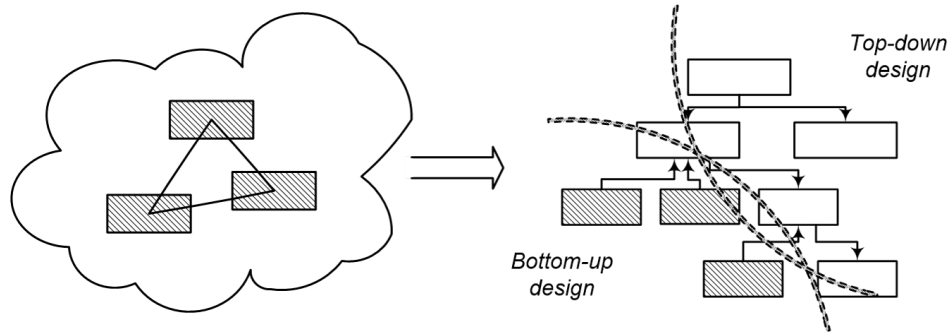
FIGURE 1. Problem breakdown is usually achieved by a combination of top-down and bottom-up approaches.

The remainder of this paper is organized as follows. First, a general introduction to the multifarious approaches to teaching programming is provided. The next section introduces trend descriptions that accompany time series data, and the automatic generation of trend descriptions. The importance of the learning environment is the focus of the subsequent section. The problem breakdown, guiding principles and pseudocode are then described in turn. Annotated examples of snippets of initial drafts of Python code and their revisions are provided in the following section. The paper closes with a brief conclusion and details seven lessons learned.

APPROACHES TO PROGRAMMING

Python has become the *de facto* standard language for natural language programming (NLP) [8]. Based on the PYPL PopularitY of Programming Language Index [9], Python is the most popular language, with approximately 30% of all programming tutorial viewers opting to study Python. It is considered relatively easy to learn due to its comparatively simple syntax and transparent semantics [10]. Python is particularly versatile with an outstanding collection of NLP tools and libraries, which enable programmers to reuse code to perform tasks, such as part-of-speech tagging, sentiment analysis and topic modeling. Libraries that can work with Python include the Natural Language Toolkit (NLTK) [11], numPy [12], CoreNLP [13], TensorFlow [14], scikit-learn [15] and many more.

A survey of the literature on teaching programming in 2011 revealed four individual approaches, namely (1) code analysis, (2) building blocks, (3) simple units and (4) full systems [16]. More recently, a literature review on innovative approaches to teaching programming found that programming was taught through specific scenarios, environments or conditions. Problem-based learning and cognitive apprenticeship were the most common categories [17]. Learning linguistics can form a symbiotic relationship with learning programming, the latter being a kind of linguistic system, where design problems assume a large variety of valid solutions [18], which may be evaluated and contrasted against each other.

Learning the Python ecosystem through the prism of natural language generation not only follows one of the most developed Python usage scenarios (thus, addressing the skill-based education goals), but also creates the rationale for bridging one of the known gaps in software development education, namely: the lack of connection between programming classes and the present-day research and technology agenda. It also contributes to developing engineering skills and to finding a good balance between students' individual efforts and software reuse. University courses that aim to develop Python programming skills fall into two broad categories: those that teach Python directly, and those that use Python to create something. This course falls into the latter category as the focus is on generating the best quality trend descriptions rather than mastering any specific skills using Python.

TREND DESCRIPTIONS AND THEIR GENERATION

Time-series graphs are used to represent changes in data over time. Graphs are visual representations of data, which help provide a holistic overview of trends or patterns. Such graphs are usually accompanied by written trend descrip-

tions. Descriptions of graphs may be found in *inter alia* scientific reports, academic theses, business reports and the financial press. Language proficiency examinations, such as IELTS, TOEFL and TOEIC, include trend descriptions in both the receptive and productive skills sections of their examinations.

A time-series graph is a line graph that shows data such as measurements, sales or frequencies over a given time period. They can be used to show a pattern or trend in the data and are useful for making predictions about the future, such as forecasting product sales or financial growth.

Trend descriptions, like other genres and sub-genres, contain predictable rhetorical moves and genre-specific lexis. Rhetorical moves include (1) referring to visuals, (2) describing overall trends, and (3) predicting future trends. Genre-specific lexis includes vocabulary that describes the directionality of change, e.g. *rise, fall* and *remain stable*, the magnitude of change, e.g. *slightly, moderately* and *substantially*, and the rate of change, e.g. *gradually* and *steadily*.

Although some researchers in the field of natural language processing focus on generating graphs by extracting information from textual descriptions of trends [19], there is a paucity of research on generating textual descriptions from data values. A core requirement is the provision of grammatically accurate and generically appropriate descriptions, so the generated text may be used as is without the need for human editors. Such descriptions need to be generated from a title and set of data values. In this paper, the input data used is not the graph itself, but the underlying data values that are used to create the graph. Populating predetermined template sentences showing trend changes is a relatively straightforward task, but the generation of natural language descriptions that readers would not suspect of being produced automatically is a non-trivial task.

Such descriptions could be used for multiple purposes, such as to accompany real-time graphs displayed online. The particular use case that is the focus of this course is for the generated descriptions to serve as models that learners of English can analyze to understand how to describe graphs appropriately. In order to serve as good models, the descriptions should not contain any "tortured phrases" [20, 21], that is phrases that belie automatic generation. Language used in the models should reflect prototypical usage, adopt typical rhetorical organisation structures, follow accepted grammatical constructions and read naturally, which means that the lexical and grammatical choices are unmarked [22, 23].

Text generation models, such as Generative Pre-Training model (GPT) [24] and the GPT-2 model [25], may be able to automatically generate descriptive texts. However, given the black-box nature of such models, it is not possible to control the output. Large language models may produce grammatically accurate and appropriate texts, but the software developer has little control of what the final output will look like and whether this output will be able to serve as comprehensible input [26] for learners and provide an appropriate pedagogic model.

In contrast, harnessing string manipulation enables the software developer to control the program output more precisely and systematically, and ensure that learners are provided with predetermined model texts based on teachable principles.

LEARNING ENVIRONMENT

Highly motivated learners can master a new programming language with few resources, little help and no teacher. Conversely, learners with little to no motivation may still fail to master a language despite abundant resources, ample help and a personal tutor. Motivation is therefore paramount. In an effort to promote instrumental motivation, at the outset of the course, job advertisements for top-tier companies which included the requirement for proficiency in Python were shown to students.

Although socially-inactive learners may not enjoy working in teams, many learners enjoy doing so. Teamwork activities also serve as good preparation for the transition from education to employment [27]. Teamwork activities help develop transferable skills, such as communication and leadership. Despite the importance of teamwork, learners with severe social anxiety were allowed to form teams of one, provided that they were able to show sufficient mastery of or commitment to learning Python and were able to communicate directly with the class tutor.

The remit for the end-of-course assignment was to develop a program to automatically generate descriptions to accompany time-series graphs. This non-trivial task provides ample opportunity for learners to develop their programming skills, regardless of their level at the start of the course. To provide sufficient extrinsic motivation, assessment criteria were designed so that learners could achieve grade A based solely on the quality of their Python program, or based on the quality of their Python program and the quality of their demonstration of the program. Students were allowed to submit a videoed demonstration [28] or give a live in-person demonstration.

Factors impacting the learning environment, learning and learners are multifarious; but here we focus on three core factors: fun, fast and flow. Through this linguistic-first approach, learners are provided with a clear goal of what they

need to achieve. However, decisions on the specific details of the output (e.g. the organisation of the generated text, the vocabulary and the grammar that need to be incorporated) are left to the student teams. Providing students use Python as their main programming language and avoid plagiarism through appropriate accreditation of reuse of source code, there are no restrictions placed on the program itself. This is designed to allow students to take ownership of their projects rather than simply follow instructions. The course is designed to enable learners to have fun, be fast and (hopefully) experience flow.

The maxim "laughter lubricates learning" serves as a useful reminder of the importance of injecting humour into programming classes. Fun is somewhat subjective, but fun may arise due to the teacher, content, materials and/or participants. Creating an environment conducive to having fun is paramount. Fun is an essential component of the *teaching as entertainment* metaphor, but fun requires good preparation [29]. Fun may occur during the creation of the code, i.e. the process. Alternatively fun may stem from the feeling of satisfaction of solving a problem, i.e. delivering the product. Setting clear criteria helps learners know when they are able to celebrate their programming victories.

Learners may find it difficult to set up an appropriate environment, which may negatively impact motivation prior to starting to learn Python. This can be ameliorated by providing learners with easy set-up options, which may be as simple as directing learners to use an online integrated development environment [30]. Enabling learners to start solving the problems quickly helps learners to time-box their learning journey since they only need to take into account the time that they expect to spend on developing code. The ideal learning environment is one in which learners are sufficiently challenged and sufficiently supported. Too much challenge or too much support throws the learning environment out of balance. Learners are said to experience the psychological state of *flow* [31] when they are entirely immersed in and paying total attention to an activity.

PROBLEM BREAKDOWN

To work out how to solve a problem, axiomatically it is necessary to fully understand the problem. This can be carried out through a formal or informal problem breakdown. Problem breakdown covers four main steps: identification of the problem to be addressed by the program, gathering relevant information that may inform the development of the program, iterating potential solutions to identify promising options and finally testing the solutions. This section focuses on the first two steps.

Relevant information was gathered via comprehensive linguistic analysis. Both top-down and bottom-up approaches [32] were adopted. Top-down approaches comprised genre and move analysis [33] while the bottom-up approach consisted of discourse and corpus analysis. A small web-crawled corpus of data-series descriptions of trends was created using appropriate seed words in WebBootCaT, a feature available in Sketch Engine [34]. The data-series trend description (DTD) corpus was cleaned using tailor-made scripts. Genre analysis was used to identify trend descriptions within articles in financial and business articles. These descriptions were then analyzed for rhetorical moves [33]. The main moves identified are listed in Table 1.

TABLE 1. Rhetorical moves identified in the Data-series Trend Description corpus

Rhetorical move	Example
Referring to visuals	Figure 1 shows the share price from ....
Describing overall trends	Footfall fluctuated throughout the year with...
Describing specific trends	Enrolments held steady in the second quarter.
Describing turning points	Sales peaked at ... on Christmas Eve before ...
Proposing explanations	The plummet in prices may have been caused by ...
Predicting future trends	The volume is expected to continue to ...

The prototypical moves identified included referring to visuals, description of general trends, description and explanation of specific trends, and predicting future trends. The moves of prediction and explanation were excluded from this project, and so the moves included were limited to referring to visuals and the description of general and specific trends.

Discourse analysis was conducted at clausal and phrasal levels to establish the structural units that function as the building blocks in trend descriptions. Various functionalities within the corpus tool, *AntConc Version 4.1.4* [35]. The functions used to investigate the language used in the DTD corpus included Key Word In Context (KWIC), keyness [36] and dispersion plots.

Students conducted some of the linguistics analyses themselves, but to ensure that there was sufficient time to develop a program, some analytic results were presented to the whole class or student teams in summary forms.

GUIDING PRINCIPLES

By using more general guiding principles rather than fixed rules, there is more opportunity for flexible implementation. This is a concept that has been implemented successfully in customer service quality [37] but has not been reported in the software engineering literature. The thorough linguistic analysis provided the empirical foundation for the creation of a set of guiding principles to inform the development of the codebase. Guiding principles include producing output that is concise, corpus-informed, and maintains expectations of collocation and colligation. Adhering to conventions of collocation and colligation involves multiple challenges, which can also be described using guiding principles, such as minimizing repetition of content words and sentence patterns. The guiding principles were informed by analyses of the DTD corpus. These principles may be used by both developers of trend description generation software and by students who need practice to write trend descriptions. The descriptions are worded with software developers in mind, and so some minor alterations when using the principles with learners are needed. The guiding principles are not designed to be mapped directly to specific instructions or snippets of code, but are there to provide a general direction in which the codebase should be steered. A selection of the guiding principles is reproduced below:

1. Generate a simple sentence for each set of subsequent values.

2. Merge subsequent sentences into single simple sentences when the direction of change is the same.

3. Use verb showing the direction of change in at least a half of all sentences

4. Use noun showing the direction of change for a third to a quarter of all sentences.

5. Use different grammatical subjects in subsequent simple sentences.

6. Append prepositional phrases describing values in the following order: initial value, the value of change and final value (e.g. from *X* by *Y* to *Z*).

7. Append prepositional phrases describing time periods in the following order: initial period and final period (e.g. from *PI* to *P*2).

8. When the value remains constant in subsequent sentences omit one value.

9. Place prepositional phrase describing value before prepositional phrase describing time period (e.g. from 32 in January to 48 in February).

10. Include no more than two prepositional phrases describing values in one clause

11. Do not repeat identical prepositional phrases in subsequent sentences.

PSEUDOCODE

Learners are introduced to pseudocode in *Algorithms I*, a compulsory core course that most students in this university complete in their first year of study. Pseudocode is a way to represent computational algorithms in a readable format using plain language [38]. Pseudocode can be viewed as a halfway point between guiding instructions written in natural language and program code. The format of the pseudocode resembles a program in terms of indentation and hierarchy, but is written using plain language. Some rules for drafting pseudocode are given in Table 2.

Table 3 shows an example of pseudocode that describes the steps in a function to assign an appropriate verb based on the comparison of two subsequent values. The use of two if-statements followed by otherwise, closely parallels, the `if elif else` constructions in Python, and so transforming this simple pseudocode into Python should be relatively straightforward.

TABLE 2. Some rules for drafting pseudocode

Rules
Write one statement for one action.
Write one statement per line.
Start statements with keyword.
Capitalize keywords.
Indent to show hierarchy.
End multiline structures.

TABLE 3. Example of student-created pseudocode

Function: To describe difference in values

COMPARE *x* with *y*.
  IF *y* is greater than *x*,
        USE *decrease*.
  IF *x* is equal to *y*,
        USE *remain the same*.
  OTHERWISE,
        USE *increase*.
END

PYTHON SOFTWARE IMPLEMENTATION

The following snippets of a Python program show how one student revised his code from the initial draft to the final draft. The pseudocode for this function is provided in the algorithm in the previous section. The initial draft (version 1) of the function is given below.

```python
1   # Version 1
2   def describe_difference(num_before: int, num_after: int) -> str:
3
4       DESCRIPTIONS = ("increase", "remain the same", "decrease")
5       description: str = ""
6       if num_after > num_before:
7           description = DESCRIPTIONS[0]
8       elif num_after == num_before:
9           description = DESCRIPTIONS[1]
10      else:
11          description = DESCRIPTIONS[2]
12
13      return description
```

This draft shows that the student was able to select the verb showing the appropriate trend by comparing subsequent data values. However, it was noted that the number of verbs was limited, and there was no variety in the type of expression created. The student developer noted that he did not know which expressions were typically used in real-life examples, and so was worried that his generated declarative statement was inappropriate. Following feedback on the output of the initial program, the student revised his code to:

```python
1   # Version 2
2   VERB_GROUPS: tuple = (
3       ("increase", "remain the same", "decrease"),
4       ("rise", "stay the same", "fall"),
5   )
6
7   def describe_difference(
8       num_before: int, num_after: int, verb_group_id: int
9   ) -> tuple[str,int]:
```

```
11          description: str = ""
12          if num_after > num_before:
13              description =
14                  VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][0]
15          elif num_after == num_before:
16              description =
17                  VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][1]
18          else:
19              description =
20                  VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][2]
21
22          verb_group_id += 1
23          return description, verb_group_id
```

In the revised program (Version 2), the student now focuses more on verb groups and has doubled the number of verbs that the function can generate, reducing repetition. However, the number of verbs that the program can select from is still very limited. After another cycle of feedback, the student revised the function again to the following Python code:

```
1      # Version 3
2      VERB_GROUPS: tuple = (
3          ("increase", "remain the same", "decrease"),
4          ("rise", "stay the same", "fall"),
5          ("go up", "stay flat", "go down"),
6          ("grow","remain unchanged","shrink")
7      )
8
9      def describe_difference(num_before: float, num_after: float, \
10     verb_group_id: int) -> str:
11         if num_after > num_before:
12             return VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][0]
13         if num_after == num_before:
14             return VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][1]
15
16         return VERB_GROUPS[verb_group_id % len(VERB_GROUPS)][2]
17
18     def reinitialize_group_id(group_id: int, group: Sized) -> int:
19         return (group_id + 1) % len(group)
20
21     def check_and_append_verb(
22         num_before: float,
23         num_after: float,
24         difference_before: int,
25         difference_after: int,
26         verb_group_id: int,
27     ) -> tuple[str, int]:
28         if difference_before == difference_after:
29             new_verb_group_id = reinitialize_group_id(verb_group_id, \
30             VERB_GROUPS)
31             return describe_difference(num_before, num_after, \
32             new_verb_group_id), new_verb_group_id
33
34         return (
35             describe_difference(num_before, num_after, verb_group_id),
36             reinitialize_group_id(verb_group_id, VERB_GROUPS),
37             )
```

In Version 3, verbs describing trends are paired by collocation (e.g. increase/decrease, rise/fall and grow/shrink). What began as a single function is now separated into different functions, making it easier to reuse them in different parts of the program. At this stage, the student commented that he is considering using classes.

Having created programs that generate trend descriptions, some student teams were able to deploy their natural language generation pipeline online. Figure 2 shows an example of a graphical user interface (GUI) created using the Flask framework to enable the Python program to work online.
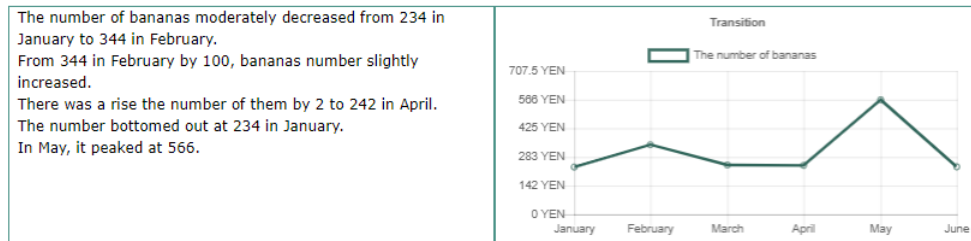

FIGURE 2. GUI for single-line graph and trend description generator

Although the output text in Figure 2 is far from ideal, it displays a variety of sentence structures and draws on a range of vocabulary, including the terms *peaked* and *bottomed out* to describe the highest and lowest points. This web application provides users with the choice of uploading data values using a CSV file or inputting data values manually. The software also generates a single-line graph and a short description. Figure 3 shows a different graphical user interface (GUI). The student-created web application enables users to input pairs of data values for each point. The program generates a bar chart and accompanying description. This description is more sophisticated and contains fewer instances of marked constructions.

FIGURE 3. GUI for bar chart and trend description generator

LESSONS LEARNED

Some key takeaways include the necessity to streamline the linguistic analysis and provide students with easily digestible linguistic guidelines and access to data-driven tools to investigate target language features empirically. Seven lessons learned from this linguistic-first problem-based approach are detailed below.

1. **Alignment of aims and assessment criteria.** Close alignment of the aims and assessment criteria help keep learners on track, working to complete their assignment, which serves as a vehicle for them to display that they have been able to achieve the set aims.

2. **Clarity of remit.** When learners are given unambiguous directions and instructions, there is less chance of misunderstanding the task at hand.

3. **Generative scenario.** The problem set must provide sufficient challenge for learners. The ideal problem is one that can be addressed at multiple levels. This may be achieved by varying the depth, breadth, scope or granularity of the scenario/problem.

4. **Student-selected teams.** Although teamwork is a valuable skill, invariably the balance of work within a team is uneven. Learners have the opportunity to develop their communication skills, work together to solve problems, and hopefully enjoy the process.

5. **Linguistic-first but linguistic-lite.** By initially focusing learners on language analysis, learners are able to see how programming can be used to solve real-world problems. Putting language first may also be a pleasant change for the learners who are less interested in programming *per se*. Linguistic analysis can become time-consuming and so in order to allow sufficient time for teams to develop programs, support can be provided to ensure timely completion.

6. **Process vs. product.** Although the aim of the course is to create a product, namely a program that generates a data-series trend description; learners are expected to learn from the process.

7. **Space to struggle and flourish.** By allowing space for teams to work on the problems themselves, they have the opportunity to struggle and solve the problems and, with luck, flourish. The teacher, however, needs to monitor discretely at a distance, to be able to provide guidance to any team or individual who may not be able to rise to the challenge without additional assistance.

ACKNOWLEDGEMENT

REFERENCES

1. Motoei Azuma, François Coallier, and Juan Garbajosa. How to apply the Bloom taxonomy to software engineering. In *Eleventh annual international workshop on software technology and engineering practice*, pages 117–122. IEEE, 2003.
2. Daniel B Oerther. Using modified mastery learning to teach sustainability and life-cycle principles as part of modeling and design. *Environmental Engineering Science*, 39(9):784–795, 2022.
3. Moula Husain, Neha Tarannum, and Nirmala Patil. Teaching programming course elective: A new teaching and learning experience. In *2013 IEEE International Conference in MOOC, Innovation and Technology in Education (MITE)*, pages 275–279. IEEE, 2013.
4. Jackie O'Kelly and J Paul Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 217–221, 2006.
5. Evgeny Pyshkin and John Blake. Bringing linguistics to a programming class: A problem of automatic text generation for describing data series. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*, pages 621–630. IOS Press, 2022.
6. Emiko Kaneko, Moonyoung Park, Ian Wilson, Younghyon Heo, Debopriyo Roy, Takako Yasuta, Allan Nicholas, and John Blake. English curriculum innovation for computer science majors in the japanese EFL context: From needs to tasks. In *2018 IEEE International Professional Communication Conference (ProComm)*, pages 84–89. IEEE, 2018.

7. Angvarrah Lieungnapar and Richard Watson Todd. Top-down versus bottom-up approaches toward move analysis in ESP. In *Proceedings of the International Conference on Doing Research in Applied Linguistics, King Mongkut's University of Technology Thonburi*, pages 21–22, 2011.

8. Lesia Ivashkevych. Teaching programming with python for linguists: Whys and how-tos. *Advanced Linguistics*, 3:4–12, 2019.

9. PYPLI. Popularity of programming language index, 2023.

10. Dennis G Balreira, Thiago LT da Silveira, and Juliano A Wickboldt. Investigating the impact of adopting python and c languages for introductory engineering programming courses. *Computer Applications in Engineering Education*, 31(1):47–62, 2023.

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

12. Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

13. Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

14. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

15. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

16. Cynthia Selby. Four approaches to teaching programming. In *Learning, Media and Technology: A doctoral conference*, London, 2011.

17. Simone C Santos, Patricia Azevedo Tedesco, Matheus Borba, and Matheus Brito. Innovative approaches in teaching programming: A systematic literature review. In *Proceedings of the 12th International Conference on Computer Supported Education*, volume 1, pages 205–214, 2020.

18. Evgeny Pyshkin. On programming classes under constraints of distant learning. In *2020 The 4th International Conference on Software and e-Business*, pages 14–19, 2020.

19. Siddhesh Kandarkar. Computer assisted natural language description of trends and patterns in time series data. Master thesis, Technishe Universitaet Muenchen, 2020.

20. Guillaume Cabanac, Cyril Labbé, and Alexander Magazinov. Tortured phrases: A dubious writing style emerging in science. evidence of critical issues affecting established journals. *arXiv preprint arXiv:2107.06751*, 2021.

21. Holly Else. 'tortured phrases' give away fabricated research papers. *Nature*, 596, 2021.

22. John Blake. Incorporating information structure in the EAP curriculum. In *Conference proceedings of 2nd International Symposium on Innovative Teaching and Research in ESP*, 2015.

23. John Blake, Evgeny Pyshkin, and Simon Pavlic. Automatic detection and visualization of information structure in english. In *Proceedings of the 6th International Conference on Natural Language Processing and Information Retrieval*, pages 200–204. ACM, 2023.

24. Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training, 2018.

25. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

26. Stephen D Krashen. *The input hypothesis: Issues and implications*. Addison-Wesley Longman Limited, 1985.

27. John Blake. Real-world simulation: Software development. *Applied Degree Education and the Future of Work: Education 4.0*, pages 303–317, 2020.

28. Christa Chewar and Suzanne J Matthews. Lights, camera, action! video deliverables for programming projects. *Journal of Computing Sciences in Colleges*, 31(3):8–17, 2016.

29. Martin Cortazzi and Lixian Jin. Bridges to learning: Metaphors of teaching, learning and language. *Researching and applying metaphor*, 149:176, 1999.

30. Michael Kölling. The problem of teaching object-oriented programming, part 2: Environments. *Journal of Object-Oriented Programming*, 11(9):6–12, 1999.

31. Mihaly Csikszentmihalyi. *Flow: The psychology of optimal experience*. Harper & Row New York, 1990.

32. Maggie Charles. Reconciling top-down and bottom-up approaches to graduate writing: Using a corpus to teach rhetorical functions. *Journal of English for Academic Purposes*, 6(4):289–302, 2007.

33. John Swales. *Genre analysis: English in academic and research settings*. Cambridge University Press, 1990.

34. Marco Baroni, Adam Kilgarriff, Jan Pomikálek, and Pavel Rychlỳ. WebBootCaT. instant domain-specific corpora to support human translators. In *Proceedings of the 11th Annual Conference of the European Association for Machine Translation*, 2006.

35. Laurence Anthony. Antconc (version 4.1.4) [computer software], 2022.

36. Marina Bondi and Mike Scott. *Keyness in texts*. John Benjamins Publishing, 2010.

37. Robert Spector and Patrick D McCarthy. *The Nordstrom way to customer service excellence: The handbook for becoming the "Nordstrom" of your industry*. John Wiley & Sons, 2012.

38. Geoffrey G Roy. Designing and explaining programs with aliterate pseudocode. *Journal on Educational Resources in Computing,* 6(1):1–es, 2006.