XIII Congresso Brasileiro de Informática na Educação (CBIE 2024)
XXXV Simpósio Brasileiro de Informática na Educação (SBIE 2024)

Explorable Discrete Mathematics: a Python-based undergraduate-level teaching approach

Marcelo de Gomensoro Malheiros[1], **Claus Haetinger**[2]

[1]Center for Computational Sciences (C3) – Federal University of Rio Grande (FURG)

[2]Institute of Physics and Mathematics (IFM) – Federal University of Pelotas (UFPEL)

mgm@furg.br, claus.haetinger@gmail.com

Abstract. *Discrete Mathematics is a foundational yet demanding introductory subject for Computer Science curricula, where undergraduate students often have difficulties grasping its concepts and applications. In this work, we describe our teaching approach, using a programming language in tandem with math notation, inviting students to explore and learn its main concepts. Our goal is to build intuition through experimentation: first, evaluating expressions and running code snippets, then linking the programming constructs to mathematical notation, and finally, formalizing the underlying theory. For this, we detail our methodology and course design covering four major topics: set theory, counting, combinatorics, and propositional logic, presented as a continuous progression. Compared to the traditional teaching of Discrete Mathematics, we have quantitatively measured an increase in grades and a reduction of students dropping out of classes while perceiving an overall smooth learning curve.*

1. Introduction

Discrete Mathematics (DM) is a fundamental theoretical topic for Computer Science (CS) and is usually taught in the first semester of CS undergraduate programs.

Traditional topics like Set Theory or Propositional Logic are presented early in CS curricula so that higher-level concepts can be built upon later. However, just like Calculus, it demands an abstraction capacity students may not yet be prepared for [Mirolo et al. 2022]. This means that students often have difficulties grasping new concepts, abstractions, and formal notation while failing to see a direct link to applications. Moreover, Discrete Mathematics is traditionally taught within a standard class setting, with pen, paper, and lots of drill exercises.

In this work, we describe our experience of teaching Discrete Mathematics with a distinct approach, matured along several editions, where by using an accessible programming environment and gradually increasing language syntax we invite students to play and explore basic DM concepts[1]. We aim to build an intuition of how DM "objects" interact, first exercising the mechanics, then linking the programming constructs to formal notation, and finally formalizing those concepts.

We have observed several advantages of this approach. First, we tap into the concurrent teaching of Algorithms (using Python in a parallel class), reinforcing the concepts

[1]We made available our course plan and several developed materials, many of them in Jupyter notebook format, at https://github.com/mgmalheiros/discrete-math/

DOI: 10.5753/sbie.2024.242630

shown there, but from a slightly different angle of problem-solving. This provides another opportunity where first-semester students can play with programming, which is a key motivation. Also, by using a REPL[2] environment, we provide instant feedback for student exploration. Finally, by carefully planning the topic progression and slowly adding new programming language syntax, we have observed a smoother learning curve when compared to the standard way of teaching Discrete Mathematics.

Here we lay down our overall course design, and cover in detail four topics: Set Theory, Counting Principles, Combinatorics, and Propositional Logic. We believe that this course proposal points to important directions for undergraduate programs, having a more integrated and gradual introduction to Computer Science theory in the first years.

2. Related work

The disciplines involving Mathematics in Higher Education require a significant amount of abstractions, as well as knowledge of notations that allow the student to communicate through math language. Such obstacles are projected over the semesters, hindering their success in subsequent disciplines, and causing retention and evasion in undergraduate programs. Moreover, loss of motivation and difficulty in understanding directly affect student engagement [Pfitscher et al. 2023]. A study by [Giannakos et al. 2017] identified gaps in knowledge from basic education, with insufficient hierarchy of concepts, difficulties in understanding and solving problems, and lack of attention and motivation as important causes for low performance in courses in Computer Science courses.

In [Benton et al. 2018] an alternative teaching methodology was recommend, using programming to produce pedagogical environments favorable to the development of learning. The authors highlight the use of intuitive environments and simple programming languages to build relationships between mathematical concepts and the real world. In [Zampirolli et al. 2020], programming is used as a tool to support personalized learning, adapted to the objectives and pace of students.

The idea of using Python as a tool for teaching mathematics is not new. In [Lockwood 2021] the role of computational activity in the development of mathematical thinking is investigated, specifically examining the generalizing activity of undergraduate students when solving combinatorial problems using Python. The authors argue programming provided new opportunities and facilitated learning. The teaching of fundamental calculus concepts through the use of programming is analyzed in [Kado 2022], which measured significant gains for high-school students who performed coding activities in contrast with a math-only control group. And for teaching Statistics with Python, in particular, there are plenty of initiatives and open books [Duchesnay et al. 2021, Suzuki and Suzuki 2021, Kenett et al. 2022], to cite a few.

A comprehensive survey of works on teaching discrete structures is presented in [Power et al. 2011]. Many of the challenges of teaching Discrete Mathematics are addressed there, like presenting its contents at once and early in the undergraduate course or splitting its topics into smaller sizes, shown later together with applied computing subjects. Concerning programming, the majority of works suggested that the implementation

[2]Read, Eval, and Print Loop, named after the steps of an interactive prompt for an interpreted programming language.

of DM topics through coding is helpful, but it implied having programming proficiency first. Other works draw parallels to particular languages and DM topics, like functions and functional languages, albeit at the time of the survey Python was yet a niche language.

More recently, [Liu and Castellana 2021] presented an approach for teaching Discrete Mathematics, adding a language extension to Python which provided additional tools, with a specific focus on predicate logic. According to the authors, it allowed students to understand the concepts precisely, write them rigorously in specifications, and use them directly in executions. Also, in [Vasconcelos and Guerra 2023] a new approach is proposed for covering Computing Theory topics through the use of Python libraries in a Jupyter Notebook environment. The topic of Turing Machines is detailed in the work, drawing state machines directly derived from the formal definitions.

Differently from early works, in this paper, we focus on a course design that explicitly matches the gradually learned programming skills to similar math notation, building a coherent and explorable progression from Discrete Mathematics topics.

3. Course design

In this section we give an overview of our course organization, highlighting our three major strategies: concurrently teaching Discrete Mathematics and Algorithms (in another course), careful choice of the programming language and the supporting software tools, and smooth topic progression following a natural order with increasing complexity.

3.1. Concurrent teaching with Algorithms

In the particular setting of our Discrete Mathematics undergraduate course, only a few traditional Computer Science topics are covered. The introduction to algorithms and programming with Python is done in another course, which occurs simultaneously in the same first semester. Also, students will continue to use Python in the following semesters.

We have designed our approach to Discrete Mathematics to closely follow the progression of programming concepts in the Algorithms course, such that as soon as they were practiced by the same group of students (undertaking both courses), we could use them as a tool to practice and explore DM concepts. The concise sequence of associated subjects in the two courses is listed in Table 1.

Table 1. Our subject association between Algorithms and Discrete Mathematics.

Algorithms	Discrete Mathematics
types and expressions	set pertinence and equality
variables and output	set cardinality, inclusion, and operations
input and conditional structure	universe, complement, and set properties
loop structure	counting principles
sequences and iteration over sequences	combinatorics
function definition	propositional logic

This is the most important strategy used in our course, as we could at the same time rely on new programming constructs to build more complex examples and also provide students with increasingly powerful tools for exploring and solving the problems posed in each DM subject.

3.2. Choice of programming language and tools

Of course, the choice of Python as the underlying programming language is also crucial for our teaching goals.

Python is a modern and mature programming language that has a strong emphasis on both readability and coherence. In fact, of the major programming languages in use today [Ramalho 2022], Python is one with minimum boilerplate code (not needing initialization constructs or explicit class declaration for simple programs, for instance). This alone makes Python very friendly to newcomers in programming. Moreover, the actual design of programming constructs from Python heavily borrows from Mathematics notation, which was an explicit goal of its creator Guido van Rossum, a mathematician.

For example, Python allows Boolean expressions with multiple comparisons in a manner that is consistent with general use in Mathematics, like in `a < b < c`, testing whether `a` is less than `b` and `b` is less than `c`. Another distinct design choice as a programming language is the use of arbitrary-precision arithmetic for all integer operations. Due to Python's extensive Mathematics library (`math`) and Linear Algebra third-party libraries (like NumPy), Python became very popular as a scientific scripting language.

Moreover, due to the recent popularity of Python in many programming fields [Nagpal and Gabrani 2019], it has been established as one of the leading programming languages, which is important as the concepts and practices done in the DM course straightforwardly add up to the programming and problem-solving skills for students.

Another important choice was the programming tools used. For the first half of the course, we used only Google Colaboratory[3], as a simple, distraction-free notebook environment with zero setup time. The main gain of Colaboratory (also used at the beginning of the Algorithms course) is to provide a Read-Eval-Print-Loop (REPL) interface focused on the interactive execution of small snippets of code. Instead of a full-fledged IDE like Visual Studio Code, which could be overwhelming in terms of both user interface and amount of functionalities for novice students, Colaboratory is very spartan. Additionally, the REPL provides the means to evaluate simple expressions, which is the ideal mode for experimenting with set relations. By not being file-oriented (as traditional IDEs), but cell-evaluation oriented, a fast exploratory experience is possible.

In the notebook approach (which can be locally executed using Jupyter[4]) a single document can contain all the experimentation done in a class, in chronological order, without scattering all that was produced by students over several independent **.py** files. Additionally, the integrated support of Markdown provides the opportunity to write longer texts mixed with code, including a much-needed LaTeX support for math notation.

As the semester progresses, and again timed together with the Algorithms course, we switch to the local Thonny[5] IDE, now providing both an interactive prompt and a file-oriented program execution model. This is particularly helpful as the student code gets complex, we can debug and single-step programs. This helps in comprehending the actual working of more complex programming language constructs, like sequences or functions. Thonny, being a novice-oriented IDE, is a perfect choice for that, having an

[3] `https://colab.research.google.com/`
[4] `https://jupyter.org/`
[5] `https://thonny.org/`

uncluttered interface and many helpful features for beginners. For example, it includes a Variables panel that tracks the values of the current program variables as it is executed. Another useful feature is a stacked window visualization for function calls, mimicking the underlying stack frame for nested calls, which helps students understand both the call mechanics and the scope of parameters and local variables.

3.3. Topic progression

A third important point for our course design is how each of the main topics (Set Theory, Counting Principles, Combinatorics, and Propositional Logic) are tied together when presented in this specific order.

We have very carefully laid out strategies for a smooth transition between these topics, building new concepts with the explicit usage of previously learned mathematical concepts and programming structures. We will detail the particular linkage between topics as we describe their organization in Section 4. For each topic, we introduced each subject following the following approach:

1. Build intuition through manipulation and experimentation of Python code, with simple exercises;
2. Connect Python with math notation, with drill exercises;
3. Define each concept formally; and
4. Exercise the whole with more traditional textbook problems, still using Python to help solve them.

For example, to start building intuition about set inclusion relations, we first presented students with several variable definitions as in Listing 1. As we have previously established the concept of element pertinence to sets, we can show new relations, expressed by the overloaded operators `<` and `<=` for sets. Then we let students experiment which combinations with variables `a`, `A`, and, `B` are valid, and for the valid ones, which yield truth values, and why.

Listing 1. Python example

```
a = 1
A = {1, 2, 3}
B = {1, 3}
```

In this particular case, we used the similarity with numeric comparison to establish what kind of relation the inclusion operators ensues. Then the mathematical notation of $\subset$ and $\subseteq$ were shown. After a few drill exercises, we then propose new problems in mathematical notation, still to be solved through experimentation using programming.

Only then the formal definition of set inclusion is presented, followed by the terminology of subsets, proper subsets, supersets, and proper supersets. Further exercises follow, trying to address edge cases like "what happens for $A \subset A$" or the relations with the empty set ($\emptyset$). Finally, we conclude with textbook exercises, again to be solved with the help of programming but formalized into written form with adequate math notation.

4. Topics covered

Particular care was made when designing the progression of topics, both regarding the use of gradually available programming language structures and the provision of strong conceptual links between subjects.

A brief rationale of our proposed order is given. Namely, Set Theory, Counting Principles, Combinatorics, and Propositional Logic. We also detail the matching between mathematical notation and Python syntax, noting some specific discrepancies.

4.1. Set Theory

When introducing Set Theory we first go back to the basics of Number Theory, reviewing with students how they learned about $\mathbb{N}$, the natural numbers. We then go directly to the Colaboratory environment and use it as a simple calculator, suggesting a few trivial addition and multiplication operations, noting that if the operands are natural, the results are always within $\mathbb{N}$. We also remember students of the peculiar cases of both zero and one, being the neutral elements for addition and multiplication, respectively. Then we give a brief overview of the historical time frame for the concept of natural numbers, and how they cover only a very limited set of human activities, therefore introducing the subtraction operator.

The strategy, encompassing the full course, is to informally present most topics from Discrete Mathematics as *algebras*, in the sense of being a "small contained world with elements and operations to be played with". As we add the new operation of subtraction, the "world" changes and we have newer elements, thus arriving at the $\mathbb{Z}$, the set of integers. We then continue revising the effects of the inclusion of the division operators and then proceed to the creation of $\mathbb{Q}$, the set of rational numbers.

Python is particularly useful as we have the dedicated $**$ operator for exponentiation so that we can delve into positive, negative, and fractional exponents, giving rise to irrational numbers. After a brief explanation, we arrive at the real numbers, $\mathbb{R}$. From this starting point, we progress into the following list of subjects, in this specific order:

1. Simple non-empty sets and pertinence, initially using just natural numbers as elements and using Python expressions like `1 in {1, 2}` as a direct equivalent to $1 \in \{1, 2\}$. Only the evaluation of Python expressions is needed at this time, which can be done in Colaboratory with a single expression per cell. By now sets are defined only by extension, that is, by enumerating all of its elements.
2. Set equality and inequality, using the Python operators `==` and `!=` as direct equivalents to the $=$ and $\neq$ math counterparts.
3. Inclusion relations, using `<`, `<=`, `>`, and `>=` as the direct equivalents to $\subset$, $\subseteq$, $\supset$ and $\supseteq$. Now Python variables (and therefore assignments) are needed to create a few sets with names like `A` or `B`, enabling the testing of many relations between pairs of these sets. Now more complex output using **`print`** can be useful too, producing one or more output lines per Colaboratory cell.
4. Empty set $\emptyset$, which unfortunately has to be created as **`set()`**, as Python reserves the `{}` construct for an empty dictionary. We reduced the notation inconsistency by creating a single variable to hold an empty set, appropriately called `empty` to highlight it as a special set, stimulating students to understand the peculiar relations involving it.
5. Cardinality can then be introduced, juxtaposing the $|A|$ math notation to the **`len`**`(A)` Python syntax.
6. Operations like union, intersection, and difference can now be presented, formalizing the concept of disjoint sets. Moreover, a direct visualization within Colab-

oratory for Venn diagrams can be made using the `matplotlib-venn`[6] package. There is a direct mapping from overloaded Python operators over sets (`|`, `&`, and `-`) to the math counterparts ($\cap$, $\cup$, and $-$). We avoid discussing symmetric difference as Python does not have a specialized operator for it; later on, it can be explicitly built as a Python function.

7. Universe and complement can be exercised by using a finite universe set and the difference operator to construct the complement set $\bar{A}$ of a given set A. Now is a proper time to present set properties like commutativity, associativity, and distributivity, for example. These properties can be directly evaluated and inferred by students using conditional Python conditional **`if`** and **`else`** statements.
8. Important properties like De Morgan's Laws or the Principle of Inclusion and Exclusion ($|A \cap B| = |A| + |B| - |A \cup B|$) can be derived empirically by experimentation from several pairs of sets, which provides an interesting class dynamic.
9. Sets of sets is the final subject explored, which suffers a bit from the Python constraint where sets can only contain immutable values. This is a bit problematic as normal Python sets are mutable (yet, we avoid showing the explicit set methods for adding or removing elements). Therefore we need to explicitly convert a normal set `A` to its immutable counterpart with **`frozenset`**`(A)`, which is then able to be added to another set. Still, experimentation over sets of sets is very important, to develop an intuition of why $\{2\} \in \{1, \{2\}\}$ is true while $2 \in \{1, \{2\}\}$ isn't.

4.2. Counting Principles

We then proceed to the principles of counting, with an early and informal presentation of the multiplicative counting principle (also called the product rule). For that, we connect with Set Theory by introducing the Cartesian product of two sets. In this topic, the sequence of subjects is presented in the following order:

1. The multiplicative counting principle is visualized by using the `product` function from the `itertools` package. As the returned value is a Python generator, the simplest approach is to explicitly convert it into a set, as shown in Listing 2, which creates the expected Cartesian product: $C = \{(2, 3), (2, 4), (1, 3), (1, 4)\}$.

Listing 2. Python example

```
import itertools as it

A = {1, 2}
B = {3, 4}
C = set(it.product(A, B))
```

2. Ordered pairs are then introduced as a special case of tuples. As in Python, the matching syntax of `(1, 2, 3)` is used, and only pertinence, cardinality, and equality are discussed for tuples. We avoid showing tuple comparison as the relational operators have very distinct meanings in Python: whereas they denote inclusion for sets, when applied for tuples they provide lexicographic order, which we believe could cause student confusion.

[6] `https://github.com/konstantint/matplotlib-venn`

3. Provided students already understand the concept of loops in programming (even when they have just seen **`while`** at this point), now it is a good moment to introduce the more complex way of specifying sets, by comprehension. That is, now we can define a new set like `B = {2 * e for e in A}`. This provides a fit introduction to **`for`** as the mechanism to iterate over elements of a given set. Likewise, the equivalence of loops defined by **`for`** `i` **`in`** **`range`**`(1, 4)` and **`for`** `i` **`in`** `{1, 2, 3}` can be practiced.
4. The multiplicative counting principle can again be addressed, now using nested loops. Because of the complex syntax, at the teacher's discretion, a more involved comprehension can be shown to also produce a Cartesian product, as in `C = {(a, b)` **`for`** `a` **`in`** `A` **`for`** `b` **`in`** `B}`.
5. The remaining subjects are the other fundamental principles: additive, subtractive, and again the principle of inclusion and exclusion. All of them can be exercised by explicitly generating all the possible decisions, which results in sets of tuples, and using set operations to provide the answer set, which can then be directly counted using **`len`**`()`.

4.3. Combinatorics

The topic of Combinatorics can be directly jumpstarted from the knowledge built previously using sets and counting principles.

The traditional subjects of arrangements with repetition, permutations, arrangements without repetition, and combinations can all be produced by specialized functions from the `itertools` module. Respectively, `it.product()`, `it.permutations()`, `it.permutations(r)`, and `it.combinations(r)`, where the `r` parameter provides the length of the sequences produced.

With the help of the programming concepts of sequences and indexing, now we can test particular elements inside a given tuple, being able to filter out individual possibilities in which we are not interested. This opens up many exploration possibilities, like detecting and dropping all tuples from a Cartesian product that have duplicated elements, thus producing permutations and arrangements without repetition.

We can also show the similarity and differences between **`set`**, **`tuple`**, and **`list`** in Python, which provides tools to students to make comparisons where the order is not important (when converting tuples to sets), and thus making it possible to explicitly construct combinations.

Finally, Python also provides the `factorial` function in the `math` module, which can be used to explicitly construct the binomial coefficient $\binom{n}{k}$, to numerically calculate all possibilities.

4.4. Propositional Logic

Historically we have perceived that many Discrete Mathematics textbooks, like [Rosen 2007], start with propositional logic. The rationale is to lay its foundations so that formalisms and theorem proving can be established earlier on. In our experience, this topic is better suited to be addressed later in the course, so we introduce propositional logic only after covering Set Theory, Counting Principles, and Combinatorics.

We even do not go straight up into Propositional Logic, but start with Boolean Algebra. Again, we intend to recover the concept of algebra as a playground for specific types of objects and their operations. So the introduction to this topic is done only by establishing the Python `True` and `False` constants, together with the operators **`not`**, **`and`**, and **`or`**.

A few hours are devoted to evaluating Boolean expressions in Colaboratory, manually creating truth tables on paper, and also showing applications like digital circuits. Students showed to be particularly interested in logic gate simulators, which can also run in a web browser and provide quick feedback for simple circuits[7] Our goal is to motivate students and develop intuition over the versatility of such a small set of values and operations.

As before, the bridge to previous topics must be explicitly made, for example, using a Cartesian product to provide all possible variable values for the evaluation of a Boolean expression.

As we address the specific notation for Propositional Logic, the already known operators are presented: negation $\neg$, conjunction $\wedge$, and disjunction $\vee$.

For the remaining operators, namely implication $\rightarrow$, equivalence $\leftrightarrow$, and exclusive disjunction $\otimes$, we need another programming language construct: function definitions using the **`def`** keyword. Therefore we build along students the functions `implication(p, q)`, `equivalence(p, q)`, and `xor(p, q)`. For the name of the last one, we opted to make a direct reference to logic circuits.

More importantly, now we can express a complex logic proposition as a single function in Python, having as parameters all the needed terms, like `p`, `q`, and `r`. This makes it possible to interactively test for a given set of Boolean values the result of a proposition. And we can again use Python to automate some tasks, like building the truth table for a given proposition when in functional form.

Likewise, we can use automation to store into a Python **`list`** all the results when testing a single proposition, combining them into a single value with the **`all`** function, which is an adequate test for tautologies. In other words, checking whether that proposition is `True` under all circumstances.

For example, the expression **`all`**`([True, False, False, True])` evaluates to `False`, meaning it is not a tautology. Again, we can test for contradictions by collecting all results and applying the function **`any`**, verifying whether it is `False` in all cases. For example, **`any`**`([False, False, False, False])` will return `False`, attesting that the proposition is indeed a contradiction.

As the topic progresses, the same strategy of generating all possible results and comparing them can be used to attest logical equivalence and logical implication between propositions. For example, the code shown in Listing 3 verifies the first De Morgan's law, $\neg(p \wedge q) \iff (\neg p \vee \neg q)$, expectedly outputting `True`.

[7]For example, `https://logic.ly/demo/`.

Listing 3. Python example

```
import itertools as it

def equivalent(p, q):
    return p == q
def left(p, q):
    return not (p and q)
def right(p, q):
    return (not p) or (not q)

results = []
for p, q in it.product({False, True}, repeat=2):
    r = equivalent(left(p, q), right(p, q))
    results.append(r)
print(all(results))
```

5. Evaluation

We have applied our methodology in two settings: normal in-person classes and remote teaching during the COVID-19 pandemic. As these settings are very distinct in terms of social interaction, we will address them separately.

During the COVID-19 pandemic, we had 11 distinct classes of Discrete Mathematics, all through remote teaching. Those classes involved three programs, Computer Engineering, Information Systems, and Automation Engineering, all provided for novice students in their first year. We had 356 students registered in total, with six teachers responsible for those 11 classes. With the flexibilization of teaching methods (either synchronous or asynchronous lessons) and not requiring explicit participation, we have analyzed only the final grade results for those classes. The covered topics were the same, although teachers had freedom to organize their teaching plans and evaluation procedures.

The overall result during the pandemic is a 40.4% approval rate, with a large standard deviation of 20.7 over the individual class approval rates. Within those 11 remote classes, two used the approach described in this paper, both given by a single teacher, with approval rates of 44.0% (in 2020, for 25 students) and 21.3% (in 2021, for 47 students). Albeit the approval rate of the first class is similar to the overall mean, we believe that the method was far from being used successfully, as the class interaction was not done at the same time and within a laboratory setting, depending on each student motivation to play and explore by themselves in their homes. By not requiring explicit participation we do not have class drop-out statistics, however for those two years the overall course abandon was unusually high, with 54 other students canceling their classes in the first few weeks.

For the in-person classes, we have analyzed the results of six classes post-pandemic (in 2022) and 12 classes pre-pandemic (from 2017 to 2019), for the same programs (Computer Engineering, Information Systems, and Automation Engineering). Here we have information on whether students failed based on non-attendance or by final grade, therefore we can assess the individual class drop-out numbers. The drop-out gives a more precise measure of the effects of simply quitting classes, whereas the reproval rates apply only to those students normally attending classes and failing the exams.

The overall result during in-person attendance is of 38.8% approval rate, with a standard deviation of 17.7 over the individual class approval rates. Within those classes, three used the approach described in this paper, given by two teachers, with approval rates of 55.4% (in 2022, for 74 students). The other 15 classes, with more traditional teaching methods, the approval average is 37.1% (covering 728 students).

However, if we do not count the students who dropped out of classes during those periods, we can measure a more precise reproval rate. That is, of those students who followed the classes until the end, how many indeed were not able to successfully pass the exams? For the three in-person classes following our approach, we got an average reproval rate of only 6.8% (3 students out of 44 that concluded). For all other classes with more traditional methods, we have an average reproval rate of 28.6% (108 students out of 378 that concluded).

We have measured only a slight decrease in the actual drop-out rates for the three classes (40.5%) against the other 15 classes (48.1%). We suppose that the similar numbers are caused by a broader phenomenon of high abandon during the first year, as there is a strong correlation to similar drop-out for introductory courses. More precisely, the drop-out rates calculated over the same period (2017, 2018, 2019, and 2022) are Algorithms (46.9%), Calculus (42.8%), and Linear Algebra and Analytic Geometry (46.1%).

Therefore, we believe a small decrease in drop-out rates and a significant increase in approval rates can be attributed to a more interesting and significant student experience using the proposed teaching methodology for Discrete Mathematics.

6. Conclusions

This work aims to use the Python language as a useful tool for Computer Science students when learning Discrete Mathematics concepts. Here we describe in detail our course design, focused on exploration and interactivity by using an actual programming language during classes. We have also analyzed quantitative evidence, which points to significant gains in approval rates and a slight reduction in drop-out rates.

One of our contributions is the use of progressively complex programming language constructs, employed to help the development of Discrete Mathematics topics as their abstraction increases. We also restate the importance of organizing the major topics in a fluid and linear sequence, where each new concept is built on top of previous subjects, formalizing and introducing new notations along the way.

Here we covered a large part of our Discrete Mathematics program, but we have not found a simple way to use Python with predicate logic and demonstration techniques. We evaluated a few Python-based theorem provers but found their syntax too distant from the math notation, which would bring more difficulties for students in already complex topics. Thus we opted for a traditional presentation with paper-based exercises.

In future work, we plan to perform a more extensive qualitative evaluation of the learning gains of this approach, measuring engagement, long-term memory, comprehension, and practical usage of the mathematical concepts. We also plan to incorporate automated grading tools for self-study, which can provide additional feedback for students. Another possibility is the algorithmic generation of drill exercises, which could also provide the respective solutions for a grading tool.

References

Benton, L., Saunders, P., Kalas, I., Hoyles, C., and Noss, R. (2018). Designing for learning mathematics through programming: A case study of pupils engaging with place value. *International journal of child-computer interaction*, 16:68–76.

Duchesnay, E., Lofstedt, T., and Younes, F. (2021). *Statistics and Machine Learning in Python*. Université Paris-Saclay, France.

Giannakos, M. N., Pappas, I. O., Jaccheri, L., and Sampson, D. G. (2017). Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness. *Education and Information Technologies*, 22:2365–2382.

Kado, K. (2022). A teaching and learning the fundamental of calculus through python-based coding. *International Journal of Didactical Studies*, 3(1):15006.

Kenett, R. S., Zacks, S., and Gedeck, P. (2022). *Modern statistics: a computer-based approach with python*. Springer.

Liu, Y. A. and Castellana, M. (2021). Discrete math with programming: A principled approach. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1156–1162, New York, NY, USA. ACM.

Lockwood, E. (2021). Investigating undergraduate students' generalizing activity in a computational setting. In *Proceedings of the 42nd Meeting of the North American Chapter of the International Group for the Psychology of Mathematics Education*.

Mirolo, C., Izu, C., Lonati, V., and Scapin, E. (2022). Abstraction in computer science education: An overview. *Informatics in Education*, 20(4):615–639.

Nagpal, A. and Gabrani, G. (2019). Python for data analytics, scientific and technical applications. In *2019 Amity international conference on artificial intelligence (AICAI)*, pages 140–145. IEEE.

Pfitscher, R. J., Camargo, L. C., Moreira, B. G., Wang, C., Zedral, R., and Garcia, T. R. (2023). Análise de sentimentos em turmas de programação com vistas ao apoio à permanência estudantil. In *Anais do XXXIV Simpósio Brasileiro de Informática na Educação*, pages 1329–1340. SBC.

Power, J. F., Whelan, T., and Bergin, S. (2011). Teaching discrete structures: a systematic review of the literature. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 275–280.

Ramalho, L. (2022). *Fluent python*. O'Reilly Media, Inc.

Rosen, K. H. (2007). *Discrete mathematics and its applications*. McGraw Hill.

Suzuki, J. and Suzuki, J. (2021). *Statistical Learning with Math and Python*. Springer.

Vasconcelos, D. R. and Guerra, P. T. (2023). Ensinando teoria da computação com jupyter notebook. In *Anais do XXXI Workshop sobre Educação em Computação*, pages 9–19. SBC.

Zampirolli, F. d. A., Pisani, P. H., Josko, J. M., Kobayashi, G., Fraga, F., Goya, D., and Savegnago, H. R. (2020). Parameterized and automated assessment on an introductory programming course. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação*, pages 1573–1582. SBC.