PYTHON IN A WEEK — CONCEPTUAL TESTS FOR LEARNING AND COURSE DEVELOPMENT

Christopher Blöcker[1], Thomas Mejtoft[2], Nina Norgren[3,4]

[1]Integrated Science Lab, Department of Physics,
Umeå University, Umeå, Sweden

[2]Department of Applied Physics and Electronics,
Umeå University, Umeå, Sweden

[3]Department of Molecular Biology, Training Hub,
Science for Life Laboratory, Umeå University, Umeå, Sweden

[4]Department of Molecular Biology, National Bioinformatics Infrastructure Sweden,
Science for Life Laboratory, Umeå University, Umeå, Sweden

ABSTRACT

Programming has gradually become an essential skill for engineers and scientists across disciplines and is an important part of the CDIO Syllabus covering fundamental knowledge and reasoning. Recently, there has been a shift away from introductory programming languages like C and Java towards Python, especially in programs where the focus lies on handling and analysing large quantities of data, such as energy technology, biotechnology, and bioinformatics. This paper illustrates the successful setup of a one-week-long introductory Python programming course with a hands-on approach. Given the limited time, a challenge is how to effectively teach students a meaningful set of skills that enables them to self-guide their future learning. Moreover, since the course does not include any summative assessment, we need other means of measuring students' learning and guiding course development. We address these challenges by coupling short lectures with short quizzes for formative assessment, adding another learning activity to the course. We find that, in the absence of summative assessment, short, frequent quizzes with immediate feedback are an excellent tool to track the learning of a class as a whole. Students report that the quizzes, albeit challenging, improved their understanding of programming concepts, made them aware of potential mistakes, and were a fun learning experience. Furthermore, the results from this paper illustrate how a new programming language can be taught to students without prior programming skills in a short period of time. We summarise our lessons learnt for designing and integrating quizzes in short-format programming courses.

KEYWORDS

Python programming, conceptual test, formative assessment, Standards: 2, 4, 7, 8, 10, 11

INTRODUCTION

The CDIO framework (Crawley, Malmqvist, Östlund, & Brodeur, 2007) mentions knowledge and skills that need to be part of the syllabus for our students to enable them to conceive, design, implement, and operate in their future professions. In the CDIO syllabus 3.0 (Malmqvist et al., 2022), *Personal and professional skills and attributes* (2) are emphasised, including everything from fundamental skills to creativity and ethical responsibility. Under *Analytical reasoning and problem-solving* (2.1) and *Experimentation, investigation and knowledge discovery* (2.2), there are many important parts that focus on modelling, analysis, and other aspects requiring good skills within data manipulation, that is, programming skills, for providing high-quality results and solutions. Consequently, not only the ability to use computers defines successful engineers but the ability to program computers. Having basic skills within programming provides a better foundation for engineers across disciplines to solve complex problems (e.g. Ball & Zorn, 2015).

Today, many different programming languages are taught to students, chosen depending on what skills are required within their discipline. For example, computer scientists need to be able to implement highly-efficient programs for complex real-world business applications and often learn C/C++ or Java, both of which are widely adopted in the industry. Physicists and engineers create models for running simulations and computations, typically using languages like C++ for efficient implementations, or MATLAB because it is specialised for numerical computations. Common tasks for data scientists include pooling data from various sources, performing statistical analyses, automating workflows, and visualising results, often done in languages such as R and Python. During recent years, academia has started to shift away from "traditional" introductory programming languages towards Python. Reasons for this shift include Python's simple syntax which makes it easy to learn and its increasing relevance in industry (Bogdanchikov, Zhaparov, & Suliyev, 2013; Cheng, Jayasuriya, & Lim, 2010; Jayal, Lauria, Tucker, & Swift, 2011; Leping et al., 2009; Mannila, Peltomäki, & Salakoski, 2006). Moreover, the abundance of available Python libraries for statistical analyses, machine learning, and visualisations has made Python a common tool across disciplines and a de-facto standard for data scientists.

In this paper, we describe how we have designed short quizzes and incorporated them into the setup of the one-week long course *Introduction to Python — with Applications in Bioinformatics* offered by the National Bioinformatics Infrastructure Sweden (NBIS) to the Swedish research community. Our quizzes are tightly coupled with the lectures' topics and inspired by the principles behind concept inventories (Taylor et al., 2014); their purpose is to help teachers identify and address students' misconceptions to improve their conceptual programming understanding. The first version of our quiz, which we made available on GitHub, contains 21 questions. We used the learning management system Canvas to run the quizzes and provide immediate feedback to the students, explaining why their answers were correct or incorrect and discussing the remaining open questions in class. Using the quizzes for the first time in 2022, we found that they work well as a learning activity, help improve students' conceptual programming understanding, and provide insights for improving teaching material.

RELATED WORK

According to Robins, Rountree, and Rountree (2003), teaching programming involves programming-language-specific knowledge, problem-solving strategies, and mental models of the problem and program domain to enable students to design, implement, and evaluate solutions. Traditional programming courses are often knowledge-driven and use textbooks focusing on presenting syntactic and semantic knowledge, supported with examples and exercises. However, problem-based instruction has been found to improve student learning. For example, Cheng et al. (2010) suggest focusing on analysing, decomposing, and solving problems instead of merely memorising programming syntax. They approach teaching programming from a constructivist approach, letting students learn and draw their own conclusions through experimentation. Vial and Negoita (2018) emphasise that learning programming is a social activity, which has implications for choosing teaching strategies and assessments. They propose a course setup with Python, Jupyter notebooks, and GitHub to facilitate collaboration and make programming an active engagement with others.

Vial and Negoita (2018) argue that teaching programming to non-computer science students removes certain constraints: Theoretical foundations that are part of traditional computer science curricula need not be covered as rigorously. They suggest focusing on solving problems in a specific domain instead of, as commonly done, teaching programming without considering an application context. Mironova et al. (2015) agree that problems should be selected based on the students' discipline. Vial and Negoita (2018) emphasise that the main objective of teaching programming to non-computer science students is not to educate future programmers, but rather teach students to think like programmers and develop computational thinking skills. However, different from our situation, programming courses for non-computer science students often aim at first-year students who are not yet domain experts in their field of study and typically span a whole semester (Cheng et al., 2010; Mironova, Amitan, Vendelin, Vilipõld, & Saar, 2016; Mironova et al., 2015; Vial & Negoita, 2018).

Concept inventories are tools that help teachers identify students' misconceptions through a set of open or closed-ended questions and problems (Taylor et al., 2014). They are well-established in physics education to assess students' understanding of concepts such as force (Hestenes, Wells, & Swackhamer, 1992), mechanical waves (Caleon & Subramaniam, 2010), or electricity and magnetism (Maloney, O'Kuma, Hieggelke, & Van Heuvelen, 2001), but have also been systematically studied and applied for computer science education in general, and, more specifically, for teaching Python (Johnson, McQuistin, & O'Donnell, 2020; Kaczmarczyk, Petrick, East, & Herman, 2010; Taylor et al., 2014). Independent of the subject, misconceptions can cause a significant challenge for students and hinder their ability to understand and apply concepts and principles. Therefore, identifying and addressing misconceptions is critical for ensuring that students have a strong foundation and are able to apply their knowledge effectively. In programming, misconceptions can arise due to a variety of factors, including differences in the semantics of the same word in programming and natural language, prior math knowledge, flawed mental models regarding how a computer executes code, inadequate problem-solving strategies, or, more generally and from a constructivist point of view, the entirety of students' previous experience (Qian & Lehman, 2017; Robins et al., 2003).

COURSE SETUP

Introduction to Python — with Applications to Bioinformatics is a one-week-long course offered to the research community in Sweden by NBIS, and aimed towards bioinformatics and data science, thus focusing mostly on usage and understanding of code, rather than an in-depth un-

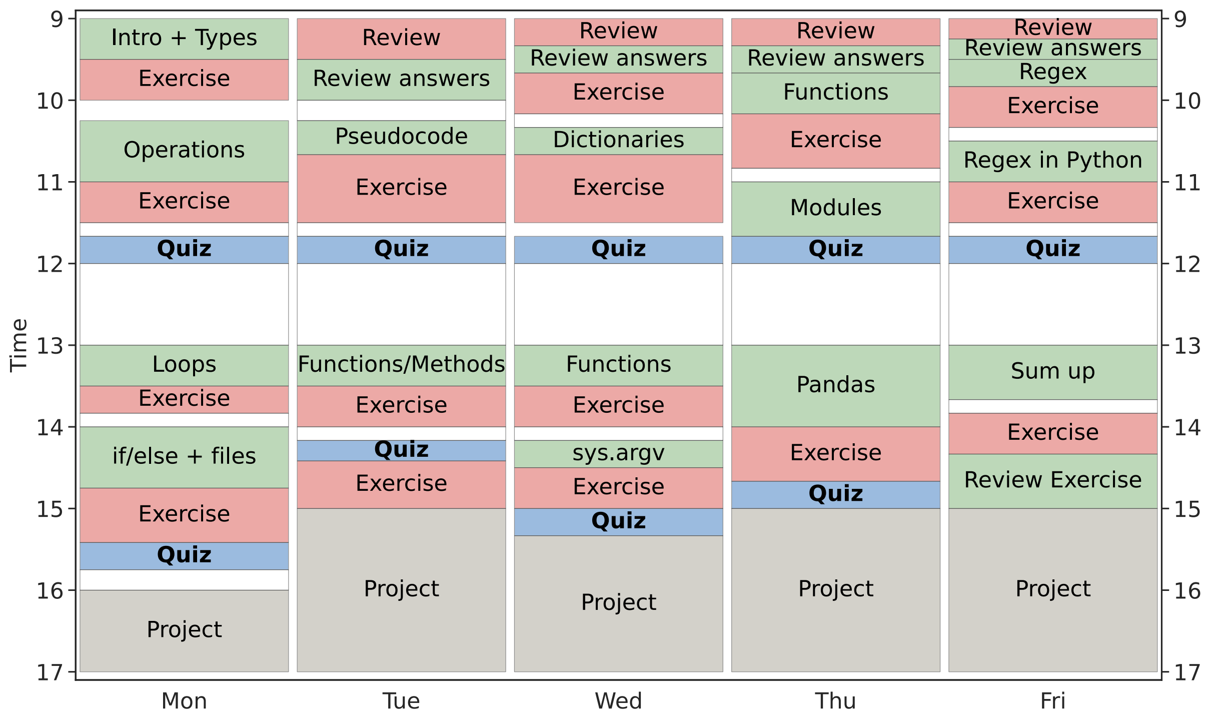

Figure 1. Course schedule where lectures are shown in green, exercises in red, quizzes in blue, and project sessions in grey. Each quiz session is thematically coupled with the previous lecture. Between lectures and quiz sessions, students have the opportunity to revisit and practice new material in the exercises.

derstanding of underlying computer science principles. The course assumes no previous programming knowledge and aims to bring students' knowledge to a level where they can directly apply Python and continue learning Python on their own. Achieving this level of understanding in one week is challenging and requires an effective course setup: we group short informative lectures together with practical exercises aimed at solidifying the students' new knowledge. In a hands-on project that spans the entire week, students work on an open-ended real bioinformatics problem, albeit simplified to fit the course's time frame, and put their newly learnt skills to work. To continuously monitor the effectiveness of this setup as well as students' learning, we have designed short formative quizzes that are thematically coupled with the lectures, and that the students answer in two quiz sessions per day (Figure 1). The learning outcomes for *Introduction to Python — with Applications to Bioinformatics* are listed in Figure 2.

NBIS, which is part of the Science for Life Laboratory (SciLifeLab), has several learning paths for becoming an expert bioinformatician or data scientist and offers a range of courses including *Neural Networks and Deep Learning*, *Omics Integration and Systems Biology*, *Single Cell RNAseq Data Analysis*, and *Advanced Python*. NBIS' courses contribute to life-long learning for researchers at all career stages, targeting mainly PhD students and postdoctoral researchers, and are a continuation rather than a part of formal education; therefore they do not contain any formal assessments. Because Python has become an important foundation in data science, all of the above courses use Python and build on *Introduction to Python — with Applications to Bioinformatics* or equivalent knowledge as a prerequisite (Figure 2).

- Use variables and explain how operators work
- Process data using loops
- Separate data using if/else statements
- Use functions to read and write to files
- Describe their own approach to a coding task
- Understand the difference between functions and methods
- Be able to read the documentation for built-in functions/methods
- Give examples of use cases for dictionaries
- Write data to a simple dictionary
- Understand concept and syntax of functions
- Write basic functions for processing data
- Describe pandas dataframes
- Give examples of how to use pandas for processing data
- Explain how regular expressions can be used
- Define Python syntax for regular expressions
- Combine basic concepts to create functional stand-alone programs to process data
- Write file processing programs that produce output to the terminal and/or external files
- Explain how to debug and further develop your skills in Python after the course

Figure 2. Learning outcomes for *Introduction to Python — with Applications to Bioinformatics*.

QUIZ DESIGN

Creating a concept inventory typically involves four steps: setting the scope, identifying misconceptions, developing questions, and validation (Goldman et al., 2010). In our case, we use the course's learning outcomes to set the scope. We have identified misconceptions and developed questions connected to each lecture topic with the purpose to improve students' learning by confronting them with related, but slightly more advanced situations. Our questions aim to prepare students for typical programming challenges and common mistakes they are likely to face when applying their programming knowledge in day-to-day work. To achieve this, we designed our quizzes based on three sub-goals: they should (i) test higher-level cognitive processes according to Bloom's revised taxonomy (Krathwohl, 2002), (ii) help identify students' programming misconceptions, and (iii) provide insights for improving the course.

First, we aimed to test students' higher-level cognitive abilities according to Bloom's revised taxonomy, more specifically their ability to *analyse* and *evaluate* Python code and to make predictions about the result that a piece of code produces. We did not cover the *create* level because it is addressed by the hands-on project; we regard the quizzes as an additional learning activity that helps prepare students for applying the learnt programming knowledge in real situations. To ensure that the quizzes test the intended Bloom's level, we purposefully designed questions that go beyond the material discussed in the lectures. Otherwise, students could answer the questions by simply recalling the respective information without activating higher-level cognitive processes. Instead, we require students to combine several concepts that were discussed in the lectures in a new way. For example, Figure 3 shows a question that tests students' understanding of variable scopes. Before answering this question, the students had learned about variables, functions, and scoping rules.

Second, we wanted to use the quizzes in a similar manner as concept inventories are used, that is, as a tool that helps teachers identify students' misconceptions, and address them in a timely manner. Therefore, we have coupled the quizzes with the lectures and run quiz sessions twice per day, one in the morning and one in the afternoon (Figure 1). However, between the lectures

```
x = 1
y = 2

def my_function(x):
   x *= 2
   return x*y

y = 4
z = my_function(2)
print(x,y,z)
```

What does the code print?	Students answered
(a) 1,2,4	(a) 0%
(b) 1,4,8	(b) 31%
(c) 1,4,16	(c) 38%
(d) 2,4,4	(d) 0%
(e) 2,4,8	(e) 8%
(f) 2,4,16	(f) 23%

Figure 3. Scopes of local and global variables. A code snippet is shown on the left, possible outputs in the middle with the correct answer underlined, and students' answers on the right.

and quizzes, students have time to revisit and practice new material in short exercises. We designed the possible quiz answers so that they all appear plausible while incorrect answers point out what misconception a student holds. For implementing the quizzes in practice, we used the learning management system Canvas because it enables providing immediate feedback. The immediate feedback helps students understand their misconceptions, allowing them to refine their mental models if necessary, which plays an important role in making the quizzes an effective learning experience.

Third, we intended the quizzes as a way to collect feedback for improving lecture and exercise content to teach programming concepts more effectively. Collecting answer statistics through Canvas provides a basis on which we can identify the most common misconceptions to develop our teaching material accordingly.

In total, we designed a quiz with 21 questions that we split up into 9 quiz sessions. The complete set of questions is available online[1]. To summarise, our quiz addresses higher-level cognitive processes by requiring students to combine learnt knowledge in new ways, helps identify misconceptions and refine mental models, and provides a basis for improving course content.

USING THE QUIZ AND STUDENTS' RESULTS

Introduction to Python — with Applications to Bioinformatics has been running for several years, but 2022 was the first time we used our quizzes as a learning activity. In 2022, there were 24 students from all over Sweden who took the course, mostly PhD students and postdocs. Their prior knowledge ranged from never having done any programming to knowing another programming language. On average, students answered 52% of the questions correctly (Figure 4a).

For example, 38% answered the scoping question (Figure 3) correctly. However, 39% answered that z has value 8 when printed, indicating that they have a misconception regarding when the value for y is accessed: at the time when my_function is defined, y holds the value 4, which would indeed result in setting z to 8. But at the time my_function is executed, the global variable y holds the value 4, which is used when assigning a value to z. 31% answered that x has value

[1]https://github.com/chrisbloecker/python-in-a-week-quiz

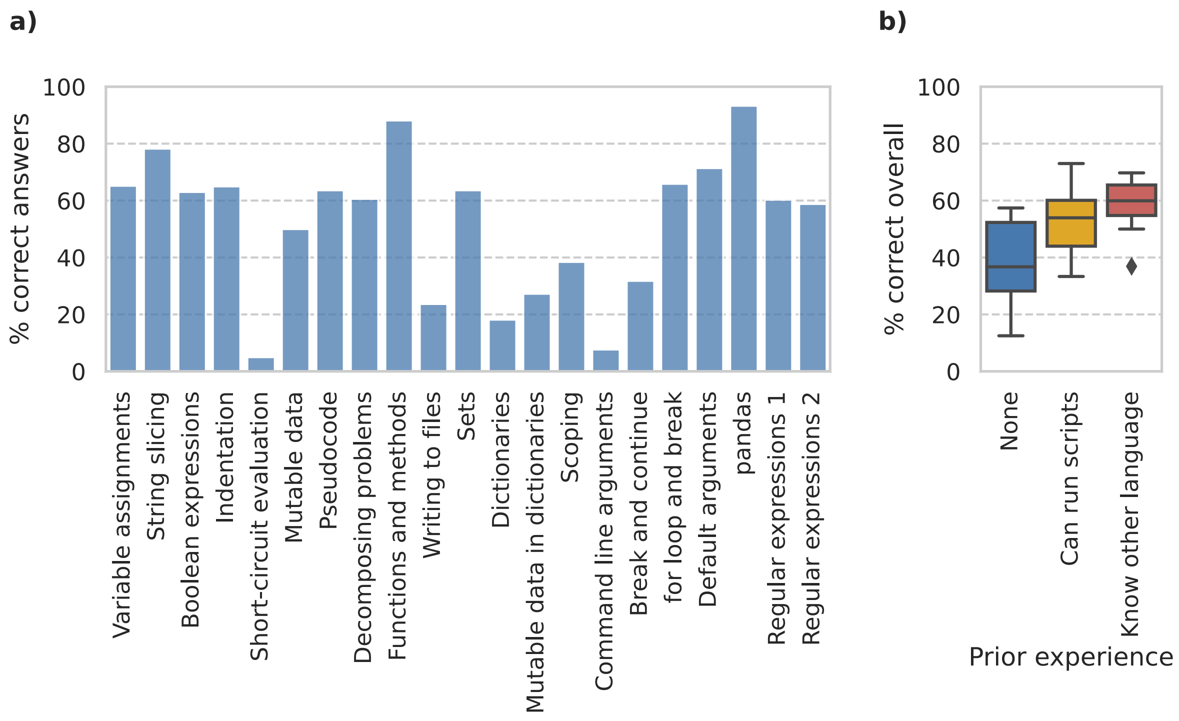

Figure 4. Quiz results and prior experience. **a)** Percentage of students answering the questions correctly. Questions are arranged in the same order as they are presented to the students during the course. **b)** Overall quiz results per student, separated by self-reported prior knowledge.

2 when printed, indicating a misconception regarding variable shadowing: the local variable x in my_functions's local scope shadows the global variable x, which remains unchanged.

The most challenging question, according to the overall results, was related to *short-circuit evaluation* (Figure 5). A possible misconception source may be the difference between how the logical conjunctions and and or are used in natural language and formal logic. However, the reason why this question was more challenging is probably how we have formulated the question: we are not interested in a variable assignment for which the condition evaluates to True. Instead, we ask for an assignment that prevents the code from crashing. Despite our hint that "not all variables are defined", students seem to forget about the objective and select the variable assignment that evaluates the condition to True, happy that ok will be printed. Nevertheless, 70% chose answer (d), indicating a good understanding of short-circuit evaluation, highlighting that it is important to be aware of which answers reveal which misconceptions.

From a teaching perspective, the quizzes provided timely insights into students' learning and revealed what parts of the introduced material were challenging. This allowed us to select the most relevant concepts for discussion after each quiz session for clarification. Moreover, the results showed what parts of the lectures require revision for more effective learning. Setting up the quizzes with detailed feedback in Canvas took some time, however, we find that this was time well spent because it allowed giving meaningful feedback to the students. Moreover, once

What variable assignment will prevent this code from crashing? Note that not all variables are defined in all cases.

```
if (a and b) or (not a and c) and (d != e):
   print(ok)
```

(a) a=False, b=False, d=False, e=True (10%)

(b) a=True, c=False, d=True, e=False (15%)

(c) a=False, c=False, d=False, e=True (5%)

(d) a=False, c=True, d=False, e=True (70%)

Figure 5. Short-circuit evaluation of Boolean expressions.

set up, the quizzes require no additional time from the teachers, give instant feedback to both students and teachers when students take the quizzes, and can easily be re-used.

On course signup, students reported their prior programming knowledge, choosing between

- I have never written any code before,
- I can run scripts written by others,
- I know another programming language (for example Perl, Java, R, etc.).

The correlation between prior programming knowledge and overall quiz result suggests that a higher level of prior knowledge tends to lead to a better quiz outcome (Figure 4b). However, because data is sparse, we can only report the results for this particular course instance and drawing general conclusions is not warranted.

In the course evaluation, we asked the students to describe how they experienced the quizzes and their contribution to their learning experience. Overall, they appreciated the quizzes as an activity that enhanced their learning. One student commented that they learned "A lot, it was a good way to practice and digest the info.". Another student said about the quizzes "They certainly helped me a lot and made me aware of details that I would have otherwise missed.".

CONCLUSION AND FUTURE WORK

We have designed a basic Python programming quiz consisting of 21 questions and integrated it into NBIS' introductory Python programming course, *Introduction to Python — with Application to Bioinformatics*. The quiz serves three purposes: (i) it tests students' higher-level cognitive skills by requiring a combination of several programming concepts, (ii) it helps identify students' programming misconceptions, and (iii) it provides data for improving the course. We used the quiz for the first time in 2022 when 24 students participated in the course. Despite being challenging, the students reported that the quizzes were a good learning activity and helped them understand programming concepts better.

We summarise our lessons learnt for designing and integrating conceptual tests in the form of a quiz in a short-format programming course:

- It is important to take the time to set up the questions, including detailed feedback on both the right and wrong answers.
- Emphasise for the students that the quizzes are not to be understood as summative assessments, but rather as learning activities. Therefore, they should not feel bad for not answering everything correctly, but rather learn from their mistakes and take the opportunity to improve their understanding.

- Since taking the quizzes was not mandatory, some students did not answer all questions. Make sure to explain the importance to the students, and allocate enough time to finish the quizzes.
- Clarify that the quizzes are supposed to be solved using pen and paper, and running the code through the Python interpreter to get the right answer defies the purpose of the learning activity.

Even though this pilot test has been carried out on PhD students and postdoctoral researchers, we believe that the results are interesting for engineering education. Giving engineering students the possibility to quickly learn basics in a new programming language will create new opportunities to give students more complex tasks in terms of, for example, data manipulation. Furthermore, letting students learn and practice basics in several programming languages during their studies will lower the bar for using those programming languages in their future profession and increase their confidence. Hence, to further extend the results from this study, this concept should be introduced in undergraduate education programs in the immediate vicinity of tasks that benefit from using Python.

Future work that remains to be done is to revise our lecture material based on the students' quiz outcomes and validate the quizzes over a longer period of time with each revision. Long-term follow-up of the students, and how, in their experience, the quizzes have contributed to their knowledge should be done, ideally around 6-12 months after the course is finished.

ACKNOWLEDGEMENTS

We would like to thank Dimitrios Bampalikis, Nanjiang Shu, Jeanette Tångrot, and all TAs who have been involved in teaching the course.

FINANCIAL SUPPORT ACKNOWLEDGEMENTS

The authors received no financial support for this work.

REFERENCES

Ball, T., & Zorn, B. (2015). Teach Foundational Language Principles. *Communications of the ACM*, *58*(5), 30–31.

Bogdanchikov, A., Zhaparov, M., & Suliyev, R. (2013, apr). Python to learn programming. *Journal of Physics: Conference Series*, *423*(1), 012027.

Caleon, I. S., & Subramaniam, R. (2010). Do Students Know What They Know and What They Don't Know? Using a Four-Tier Diagnostic Test to Assess the Nature of Students' Alternative Conceptions. *Research in Science Education*, *40*(3), 313–337.

Cheng, T. K., Jayasuriya, M., & Lim, J. (2010). Removing the fear factor in Programming. *The Python Papers Monograph*, *2*, 1–9.

Crawley, E. F., Malmqvist, J., Östlund, S., & Brodeur, D. R. (2007). *Rethinking Engineering Education: The CDIO Approach*. Boston, MA: Springer.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010, jun). Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Trans. Comput. Educ.*, *10*(2).

Hestenes, D., Wells, M., & Swackhamer, G. (1992). Force concept inventory. *The Physics Teacher*, *30*(3), 141–158.

Jayal, A., Lauria, S., Tucker, A., & Swift, S. (2011). Python for Teaching Introductory Programming: A Quantitative Evaluation. *Innovation in Teaching and Learning in Information and Computer Sciences*, *10*(1), 86–90.

Johnson, F., McQuistin, S., & O'Donnell, J. (2020). Analysis of Student Misconceptions Using Python as an Introductory Programming Language. In *Proceedings of the 4th conference on computing education practice 2020.* New York, NY, USA: Association for Computing Machinery.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying Student Misconceptions of Programming. In *Proceedings of the 41st acm technical symposium on computer science education* (pp. 107–111). New York, NY, USA: Association for Computing Machinery.

Krathwohl, D. R. (2002). A Revision of Bloom's Taxonomy: An Overview. *Theory Into Practice*, *41*(4), 212–218.

Leping, V., Lepp, M., Niitsoo, M., Tõnisson, E., Vene, V., & Villems, A. (2009). Python Prevails. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing.* New York, NY, USA: Association for Computing Machinery.

Malmqvist, J., Lundqvist, U., Rosén, A., Edström, K., Gupta, R., Leong, H., … Spooner, D. (2022). The CDIO syllabus v3.0 - An updated statement of goals. In M. S. Gudjonsdottir et al. (Eds.), *Proceedings of the 18th International CDIO Conference* (pp. 18–36). Reykjavik University/CDIO Initiative.

Maloney, D. P., O'Kuma, T. L., Hieggelke, C. J., & Van Heuvelen, A. (2001). Surveying students' conceptual knowledge of electricity and magnetism. *American Journal of Physics*, *69*(S1), S12–S23.

Mannila, L., Peltomäki, M., & Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education*, *16*(3), 211–227.

Mironova, O., Amitan, I., Vendelin, J., Vilipõld, J., & Saar, M. (2016). Teaching programming basics for first year non-IT students. In *2016 IEEE Global Engineering Education Conference* (p. 15-19).

Mironova, O., Vendelin, J., Amitan, I., Vilipõld, J., Saar, M., & Rüütmann, T. (2015). Teaching computing for non-IT students experience of Tallinn University of Technology. In *2015 IEEE Global Engineering Education Conference* (p. 305-309).

Qian, Y., & Lehman, J. (2017, oct). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.*, *18*(1).

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Comput. Sci. Educ.*, *13*(2), 137-172.

Taylor, C., Zingaro, D., Porter, L., Webb, K., Lee, C., & Clancy, M. (2014). Computer science concept inventories: past and future. *Computer Science Education*, *24*(4), 253–276.

Vial, G., & Negoita, B. (2018). Teaching Programming to Non-Programmers: The Case of Python and Jupyter Notebooks. In *International conference on interaction sciences.*

BIOGRAPHICAL INFORMATION

Christopher Blöcker is a Senior Research Engineer at the Integrated Science Lab, Department of Physics at Umeå University. He holds a PhD in Computational Science and Engineering from Umeå University and has a background in computer science. He has worked as a Research Associate in Bioinformatics at Duke-NUS Medical School in Singapore and as a Software Engineer in an industrial setting in Germany. His research is focused on Community Detection in Complex Networks, and his teaching includes courses about mathematical modelling with networks, information theory, and technology for social media.

Thomas Mejtoft is an Associate Professor at the Department of Applied Physics and Electronics at Umeå University. He holds a PhD from the Royal Institute of Technology (KTH) in Stockholm and since 2011 acting as the director of the five-year integrated Master of Science study program in Interaction Technology and Design at Umeå University. His teaching includes interaction technology, interaction design, technology for social media, and business development.

Nina Norgren is a Training Manager for the Science for Life Laboratory (SciLifeLab) Training Hub, and a Training Coordinator at the National Bioinformatics Infrastructure Sweden (NBIS), SciLifeLab. She holds a PhD in Medical Genetics from Umeå University. Her research interests include pedagogics and life-long learning, with a specific interest in bioinformatics and data science. She is also the responsible course leader for the Python course.

Corresponding author

Nina Norgren
Department of Molecular Biology
National Bioinformatics Infrastructure Sweden
Science for Life Laboratory
Umeå University
SE-901 87, Umeå, Sweden
nina.norgren@umu.se