Learning to program: from problems to code

Paul Piwek, Michel Wermelinger, Robin Laney and Richard Walker
School of Computing and Communications
The Open University
Milton Keynes, UK
{paul.piwek, michel.wermelinger, robin.laney, r.c.walker}@open.ac.uk

ABSTRACT

This paper introduces the approach to teaching problem-solving and text-based programming that has been adopted in a large, post-18, undergraduate, key introductory module (L4 FHEQ) on Computing and Information Technology at the Open University (UK). We describe how students are equipped with programming, but foremost problem-solving skills. Key ingredients of the approach are interleaving of skills, explicit worked examples of decomposition, formulation of algorithms (with the help of patterns for recurring problems) and translation to code. Preliminary results are encouraging: students' average course work scores increase as they progress through the course.

CCS CONCEPTS

Education • Distance Learning • E-Learning

KEYWORDS

Problem solving, Python, programming, patterns, algorithms, problem decomposition

1 Introduction

Learning to program, especially in a text-based programming language, is often viewed as difficult by students [1, 2], and consequently students can easily lose motivation. Also, mastery of the constructs of a programming language does not automatically translate into the ability to solve new programming problems [3]. These difficulties are compounded when programming is learned in a distance or blended learning context. This paper describes how text-based programming is introduced as part of problem solving in TM112 'Introduction to computing and information technology 2', a large key introductory distance learning module at the Open University (UK). We describe both the approach and preliminary results with the first cohort (April – September 2018).

Our Faculty offers several qualifications in Computing and Information Technology. Most of the pathways that lead to these qualifications start with our key introductory modules, TM111 and TM112.

Both are 30 credit, post-18, undergraduate modules at Level 4 FHEQ (Scottish Level 7). The modules serve several purposes: equip students with study skills for their further studies, introduce a range of Computing and IT topics and prepare students for problem solving and programming in subsequent modules.

Both modules are taught over 21 weeks (approximately 14 hours of student workload per week). The number of students per cohort exceeds 1500. Student backgrounds vary from no prior computing experience to professionals who need a qualification.

Students study in groups of about 20, under the guidance of a tutor who provides online and face-to-face tuition, and feedback through marking of course work. There are module-wide and tutor group-based online discussion forums for peer-to-peer support.

Students are advised to study TM112 immediately following completion of TM111. TM111 introduces basic study skills, employability and personal development planning, computing and information technologies and programming in a visual programming language (a variant of MIT's Scratch [4]). TM112 builds on the skills from TM111. At the core of TM112 are the three themes shown in Table 1.

Table 1: Descriptions of the TM112 Themes

Theme	*Description*
Essential information technologies	This theme introduces you to information technologies, including basic computer architecture, the cloud and mobile computing. At the same time, you'll work to improve your numerical skills.
Problem solving with Python	This theme helps you develop your problem-solving skills as you get familiar with the Python programming language, analyse real-world data and carry out programming projects.
Information technologies in the wild	This theme allows you to practise your communication and analytical skills as you explore the profound legal, social, ethical and security challenges posed by information technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CEP '19, January 9, 2019, Durham, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6631-1/19/01...$15.00
https://doi.org/10.1145/3294016.3294024

Each theme covers specific topics and allows students to practise a range of skills. Though each theme addresses several skills, as shown in Table 1, there is one core skill that each theme focuses on.

The three themes are interleaved, rather than delivered in succession, to allow each of the skills to bed in over a longer period, cf. [5]. This was deemed important especially for the problem solving/programming skills. It also means that the skills can be assessed (with feedback for learning) at more than one point during the course.

The problem solving with Python theme is delivered over 6 weeks as follows:

- Week 2: *Introduction to problem solving in Python*: Sequence, selection, variables, lists and (nested) iteration, mostly demonstrated using the Python Turtles library.
- Week 4: *Patterns, algorithms and programs 1*: Formula problems, case analysis (and Booleans), testing and documentation, pattern for generating a sequence.
- Week 7: *Patterns, algorithms and programs 2*: Generating lists, reduce (count and aggregate), search (finding a value/the best value), combining patterns.
- Week 9: *Organising your Python code and data*: Introduction to Python functions (and automated testing of functions with `assert`) and Python objects and names.
- Week 10: *Diving into Data*: worked example of analysis, with Python, of Office for National Statistics (ONS) health and wellbeing data.
- Week 15: *My Python project*: worked example implementing a flashcard program that makes use of the module's electronic glossary. This week also introduces Python dictionaries, interactive loops and the `random` library.

These 6 weeks (~84 study hours) constitute about 35% of the entire module content, with the remaining 15 weeks dedicated to the other two themes and separate assessment weeks. Each week is supported by one chapter in a collection of three printed books (developed specifically for this module) and online study and programming activities. This is complemented with online automatically marked formative quizzes - including CodeRunner [6] quiz questions for coding questions.

From October 2017, the two 30-credit modules TM111 and TM112 replaced the 60-credit module TU100. TU100 made use of another variant of MIT's Scratch in combination with a physical sensor board. However, the absence of text-based programming at L4 FHEQ was cause for concern, especially with several students struggling with text-based programming at L5. Additionally, the focus in TU100 was on the programming constructs and student experimentation, with little explicit guidance on problem solving techniques and heuristics. To prepare students better for problem solving and programming at L5 (both in Java and Python) TU100 was divided in two: TM111 using a visual language to ease students into programming with engaging games-oriented programming tasks and TM112 using Python and focusing on problem solving and worked examples with real-world data.

2 From problems to code via patterns

Robins et al. observe that "A major recommendation to emerge from the literature is that instruction should focus not only on the learning of new language features, but also on the combination and use of those features, especially the underlying issue of basic program design." [3] On TU100, "many students were reportedly daunted by the sophisticated programs they were presented with (which they were asked to modify) and tutors felt it would be better to ask students to build up their own program from a simpler base, in a more stepwise fashion." [2]

The use of programming and design patterns is well grounded in cognitive theories in how knowledge is constructed and organized and how people become experts in problem solving. Muller et al. [7] introduced 30 programming patterns and showed that they improve the students' ability to correctly solve programming problems. Some of their patterns are specific, e.g. "extreme value computation", or very small, e.g. "traverse successive elements". Our patterns are more generic and correspond to complete sub-problems, e.g. "find best value". This reduces the students' cognitive load as there are fewer patterns to learn.

2.1 Decomposition, patterns, algorithms and code

To help students construct programs in response to problem statements, TM112 is packed with worked examples and activities that guide students from a problem statement to code. We begin with the following simple workflow:

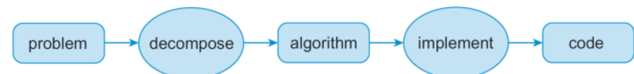

Figure 1: Simple workflow for the problem-solving process © The Open University

Imagine we want to draw the letter L with the Python turtle. We can decompose this problem as follows:

```
> Draw L
point turtle downwards
move forward by 50 units
turn left by 90 degrees
move forward by 30 units
```

Our first line uses the chevron ('>' symbol), which shows that the first line is a heading: '> Draw L' tells us what we want to do. It describes the problem we are solving. The next four lines are a decomposition of the heading line above. These four lines achieve the task set out in the heading.

And this can be translated into Python code:

```
# Draw L
from turtle import *
right(90)
forward(50)
left(90)
forward(30)
```

Subsequently, for problems with subproblems, the workflow is extended as shown next.

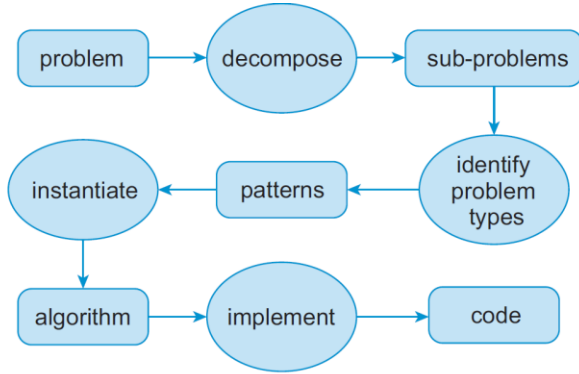

Figure 2: Full workflow for the problem-solving process © The Open University

The rationale is that the most difficult part is problem solving, not coding, but fortunately there are recurring problem types with boilerplate solution templates. The process thus becomes:

1. Recognise the type of problem.
2. Get the corresponding solution pattern.
3. Instantiate the pattern to get an algorithm for the problem at hand.
4. Translate (largely 'automatically') the algorithm to code.

In UK schools, computational thinking is taught as comprising decomposition, abstraction, generalization (patterns), algorithmic thinking and evaluation [8]. The process above starts with the decomposition step, but instead of asking students to do the hard step of abstraction, we do it for them, providing the patterns that they need to match to the problem at hand, to remove a potential stumbling block towards obtaining a solution. Overall, TM112 students are introduced to 14 distinct patterns for common problem types, including generating, searching, and filtering lists.

The patterns help students realise how problem types are related. For example, filtering a list is a search that retrieves *all* items satisfying some condition, and the solution pattern is a special case of the list transformation pattern with the identity transformation. Patterns also make it clearer that variables play particular roles, e.g. the container (a list), the iterator (each item processed), the accumulator (the resulting item or list and its intermediate values). Patterns are a thinking scaffold that guides algorithm development: students are forced to think how to initialise and update variables to instantiate a given pattern.

We describe patterns and algorithms in plain English, with variables in italics and numbered lists of steps, *not* pseudo-code. In a L5 module that uses a form of pseudo-code we observed that for some students, pseudo-code is a barrier: they perceive it as yet another language to learn and fret over its syntax. We thus use plain English, albeit in a formulaic way (see examples below) but without drawing attention to it. The formatting conventions followed in English (starting itemized lists with a colon and indenting them) map directly to Python, easing the translation of the English algorithm to code.

Consider the problem of computing the volume of a brick, given its width, length and height. We perform an initial decomposition into the follow subproblems:

```
> Compute the volume, given the width, length and height
>> Compute the base area, given the width and the length
>> Compute the volume, given the base area and the height
```

Both sub-problems are of the same form, so there is only one problem type to identify. We refer to these type of problem as 'formula problems'. They are solved by the following pattern:

Line	Instruction
1	initialise the input variables
2	set the output variable to the value of the formula applied to the inputs
3	print the output variable

Note that the pattern is not an algorithm, it is a template that needs to be 'filled in' (with the variables and values to be used for the problem at hand) to become an algorithm. Via several intermediate steps (not shown here), the pattern is instantiated to the following algorithm.

1	> Compute the volume, given the width, length and height
2	>> Compute the base area, given the width and the length
3	set *width* to 2
4	set *length* to 3
5	set *area* to *width* * *length*
6	print *area*
7	>> Compute the volume, given the base area and the height
8	initialise the input variables
9	set the output variable to the value of the formula applied to the inputs
10	print the output variable

Note that we keep the headings of the sub-problems to help structure the algorithm.

Finally, the algorithm is translated into code, rather mechanically. The sub-problem headings become comments.

```
# Compute the volume of a brick, given its dimensions
# Compute the base area from the width and length

# Input: the width, a positive float in any distance unit
width = 2

# Input: the length, a positive float in the same unit
length = 3

area = width * length
print('The base area is', area)

# Compute the volume from the base area and height

# Input: the height, a positive float in the same unit
height = 4

# Output: the volume, in cubic distance units
volume = area * height
print('The volume is', volume)
```

Further examples of patterns are available online [10].

2.2 Worked examples

The final two weeks consist of extended worked examples/activities. Student learning is known to benefit from worked examples, see e.g. [9]. According to [1], students learn programming best when given assignments that inspire. Assignments need to connect with student interests, see also [9]. In these two weeks students apply their problem-solving skills in extended realistic scenarios chosen to capture their imagination. They apply the problem-solving strategies and patterns they have learned to authentic problems and meet some further strategies. They are also encouraged to take a reflective approach and to begin the habit of keeping a journal as they work on problems.

3 Assessment and Module Evaluation

Apart from the formative assessment with quizzes, TM112 has three summative assignments, each equivalent to about 10 hours of student workload. Each assignment consists of several questions, with only some of these specifically about problem solving and programming, as follows:

Assignment 1 *Question 3:* Turtles, problem decomposition, nested iteration. *Question 5:* Inputs and outputs, admissible values, tests, borderline values, patterns, algorithms and code.
Assignment 2 *Question 3:* Admissible inputs and outputs, writing tests, decomposition into subproblems, identifying problem types and patterns, writing code. *Question 5*: Python objects, decomposition, algorithm, code with a Python function. *Question 6*: Data analysis using Python functions that are provided.
Assignment 3 *Question 3*: Flashcard programming project extension. Amending an algorithm for a function. Write Python code with amended function, testing and documentation of code, use of a notebook to track progress (synoptic).

The mean of scores on the programming questions progresses from Assignment 1 (at 71.1%) to Assignment 2 (at 75%) and Assignment 3 (at 88.6%).

In terms of overall marks, those studying TM112 as part of an Open Degree (where students are free to choose the modules they study) or as stand-alone module not linked to any qualification perform better than Computing students, but those studying TM112 as part of a Computing with a second subject qualification perform slightly worse (~5% lower pass rate than Computing-only students). This suggests that the material is appropriate both for Computing and non-Computing students.

4 Conclusion and further work

The progressively increasing mean programming scores are encouraging in that they are consistent with the intentions behind the design of the summative assessment: to help students develop skills in lightweight Assignments 1 and 2 (contributing 15% and 35% of overall module score, respectively), so they are prepared for the more synoptic application of these skills in Assignment 3 (weighted at 50% of the module score; additionally there is a 30% threshold on Assignment 3). The first cohort has just finished the course and as such we don't yet have the students' end-of-course survey data, but we will analyse it soon. Eventually, we would like to do follow-up studies with students that have progressed to L5, to determine to what extent they have benefitted from the approach.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Institute of Coding, and thank the anonymous reviewers for their comments.

REFERENCES

[1] Jenkins, T. (2002). 'On the Difficulty of Learning to Program', *Proc 3rd Annual HEA Conference for the ICS Learning and Teaching Support Network*, pp. 1-8.
[2] Chetwynd, F. and C. Dobbyn (2014). 'Transforming retention and progression in a new Level 1 course', *eSTEeM project Final report*, The Open University, Milton Keynes.
[3] Robins, A., Rountree, J. and Rountree, N. (2003). 'Learning and Teaching Programming: A Review and Discussion', *Computer Science Education*, 13(2), pp.137-172.
[4] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Y. Kafai. (2009). 'Scratch: programming for all', Commun. ACM, 52(11) (November 2009), 60-67. DOI: https://doi.org/10.1145/1592761.1592779
[5] Brown, P.C., Roediger III, H.L. and M.A. McDaniel (2014). *Make it stick: The science of successful learning*. Harvard University Press, Cambridge, Massachusetts.
[6] Lobb, R. and J. Harlow (2016). 'Coderunner: A Tool for Assessing Computer Programming Skills'. *ACM Inroads*, 7(1).
[7] Muller, O., Haberman, B., Ginat, D. (2007). 'Pattern-oriented instruction and its influence on problem decomposition and solution construction'. Proc. ITiCSE, pp. 151-155, ACM.
[8] Czismadia, A. et al. (2015), Computational thinking: A guide for teachers. Computing at Schools, part of BCS.
[9] Merrill, M. D. (2002). 'First principles of instruction'. *Educational Technology Research and Development*, 50(3), 43-59.
[10] Wermelinger, M. (2018). 'From problems to programs'. Available at: https://community.computingatschool.org.uk/resources/5691