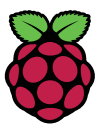
Computational thinking framework

Version 1.0, July 2020



Raspberry Pi

Computational thinking framework

Version 1.0, July 2020

Published in July 2020
by the Raspberry Pi Foundation

www.raspberrypi.org

ISSN 2514-586X

Contents

Introduction

Our mission at the Raspberry Pi Foundation is to put the power of computing and digital making into the hands of people all over the world. A large part of our work is supporting people to learn and develop computing skills, and knowledge of how computers function and how to harness them to create projects and solve problems. We promote educational approaches that enable young people to learn through making and to explore their own interests, because we see this as the most engaging and relevant way for them to learn. Our educational resources are underpinned by a rigorous understanding of computing and computer science, and they include key learning objectives and progression.

To support all of this work, we have collaborated with experts and experienced educators to develop a framework of computational thinking (CT) skills.

What is computational thinking?

Computational thinking comprises a set of ideas and thinking skills that people can apply to design systems that a computer or computational agent can enact; part of CT is expressing problems in such a way that computing can be used to solve them. The term 'computational thinking' was originally used by Seymor Papert in the 1980s[1] in his work on encouraging exploratory learning using computers. The ideas that are part of CT have been refined over many years, in conjunction with the development of computing. More recently, they have been made part of the key principles of the curriculum subjects computing and computer science. This happened following discussions in the field prompted by Jeanette Wing[2]. She argued that CT is fundamental and could be widely used across areas of our everyday lives, rather than only being reserved for specialists. However, some researchers have reservations about how widely CT is applicable outside of computer science; notably, Tedre and Denning point out drawbacks highlighted by historic attempts to apply computing skills to other domains[3].

A number of resources for CT are used in various educational initiatives. For example:

- Computing At School in the UK has created a guide to CT for teachers, which sets out the areas of CT and tips for developing them in the classroom. This guide has been used to structure the widely used Barefoot Computing resources[5].
- The Massachusetts Department of Elementary and Secondary Education's Digital Literacy and Computer Science Framework sets out detailed objectives, sorted by age range, covering CT as well as many other aspects of digital literacy[6].
- The Computer Science Teachers' Association published a set of examples of what computational thinking could look like in the classroom[7].

How did we develop this framework?

We wanted to assemble the existing documents into a framework that captures the big picture and powerful ideas of CT, but is also detailed enough to allow educators to build these ideas into learning activities and resources, and even use them to assess students.

In order to do this, we brought together a group of experts in computing and computing education, including academics, educational resource developers, and experienced and practising teachers. The group considered many different perspectives on CT, as well as experiences in the classroom, to come up with a set of themed learning objectives that represent what we believe learners can work towards as their CT skills develop.

We think we have reached a practical definition of the term 'computational thinking' that is open enough to provide scope for exploration but specific enough to allow us (and others) to systematically incorporate learning experiences designed to develop CT skills into learning resources. The resultant CT framework is now being used in the development of learning resources at the Raspberry Pi Foundation.

We see this framework as a first iteration, which we will review and revise in the future based on experience and feedback.

Why are we sharing the framework?

We're sharing our CT framework as part of our work to make visible the tools we are using to inform learning experiences in our educational work, and in the hope others will find it useful for their own work on CT and computing education.

We invite discussion and comments from others working in this area, whether in academia and research or more practically in education and supporting young people. If you have feedback for us, please do get in touch by emailing research@raspberrypi.org.

Acknowledgements

Document authored by Oliver Quinlan with Rik Cross.

We would like to thank the expert academics, educators, and teachers who spent time working with us to develop this framework, face to face, online, and by reviewing drafts.

Thank you to the following people for their ideas, input, and critique, all of which contributed to this work:

Phil Bagge, Miles Berry, Helen Caldwell, Lynda Chinaka, Prof. Paul Curzon, Prof Quintin Cutts, Catherine Elliot, Dave Gibbs, Sway Grantham, Amanda Haughs, Dawn Hewitson, Simon Humphreys, Peter Kemp, William Lau, Chris Roffey, Sue Sentance, Peter Strawn, Christine Swan, Jane Waite, Sarah Zaman

---

[1] Papert, S. (1980). *Mindstorms: Children, computers and powerful ideas.*

[2] Wing, J. (2006) 'Computational Thinking', in *Communications of the ACM*, March 2006. Available at: https://doi.org/10.1145/1118178.1118215

[3] Tedre, M. and Denning, P.J. (2016) The Long Quest for Computational Thinking. *Proceedings of the 16th Koli Calling Conference on Computing Education Research*, November 24-27, 2016, Koli, Finland: pp. 120-129. Available at: http://denninginstitute.com/pjd/PUBS/long-quest-ct.pdf

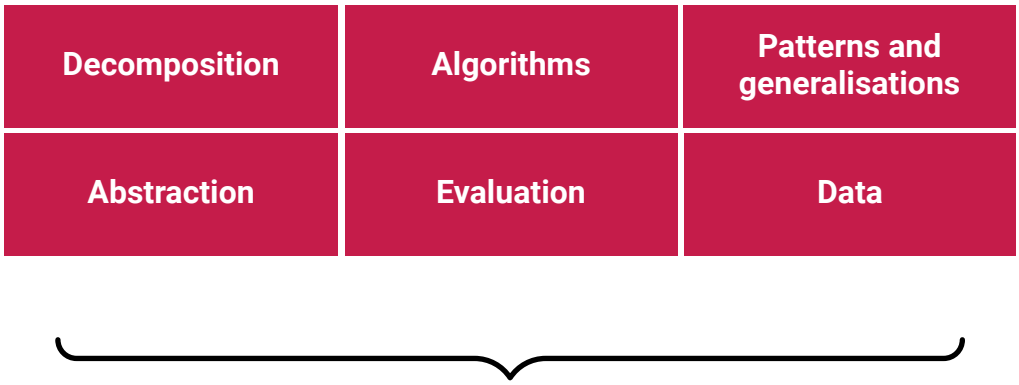
[4] Csizmadia, A., Curzon, C., Dorling, M., Humphreys, S., Ng, T., Selby, S., Woolard J. (2015). *Computational thinking: A guide for teachers*. Available at: https://community.computingatschool.org.uk/resources/2324/single

[5] Barefoot: Building Skills for Tomorrow. https://www.barefootcomputing.org

[6] Massachusetts Department of Elementary and Secondary Education (2016). *Digital Literacy and Computer Science: Grades Kindergarten to 12*. Available at: http://www.doe.mass.edu/stem/dlcs/?section=resources#standards

[7] Computer Science Teachers Association (CSTA), *CT examples from across the curriculum*. This document is no longer available online. Reach out to CSTA if you're interested in accessing it.

Computational thinking comprises a set of ideas and thinking skills that people can apply to design solutions or systems that a computer or computational agent can enact.



1: Decomposition	
Theme	**Learning objective**
	a. Know what decomposition is and/or understand when it can be useful
Breaking a problem into smaller parts	b. Identify when a problem needs to be broken down
	c. Break down instructions or systems into parts to make them easier to work with
	d. Break down a problem into simpler versions of the same problem that can be solved in the same way (recursive and divide and conquer strategies)
Recombining solutions	e. Understand how individual components of systems are combined, and how data flows between them (e.g. sensors, output devices, etc.)

2. Algorithms	
Theme	**Learning objective**
	a. Know what decomposition is and/or understand when it can be useful
Breaking a problem into smaller parts	b. Identify when a problem needs to be broken down
	c. Break down instructions or systems into parts to make them easier to work with
Identifying steps and designing algorithms	d. Break down a problem into simpler versions of the same problem that can be solved in the same way (recursive and divide and conquer strategies)
	e. Understand how individual components of systems are combined, and how data flows between them (e.g. sensors, output devices, etc.)
	f. Design steps to be followed in a given order (a simple sequence) or in parallel
	g. Design instructions that use arithmetic and logical operators
	h. Design sequences of instructions that store, move, and manipulate data (variables and assignment)
	i. Design instructions that choose between different instructions (selection)
	j. Design instructions that repeat groups of instructions (loops/iteration)
	k. Group and name a collection of instructions that do a well-defined task to make a new instruction (subroutines, procedures, functions, methods)

2. Algorithms cont.	
	l. Design instructions that involve subroutines that use copies of themselves (recursion)
	m. Design sets of instructions that can be followed at the same time by different agents (computers/people, parallel thinking and processing, concurrency)
	n. Design a set of declarative rules (such as coding in a database query language)
	o. Use notation to represent algorithms (e.g. flow charts, informal diagrams, or pseudocode)
Boolean logic	p. Understand Boolean operators
	q. Apply laws of Boolean logic to simplify statements
	r. Appreciate that Boolean logic can be used to control the flow of a program
	s. Understand the relevance and applications of Boolean logic in computer system
	t. Recognise that boundaries need to be taken into account for an algorithm to produce correct results
	u. Describe that there are ways to characterise how well algorithms perform and that two algorithms can perform differently for the same task
	v. Understand that algorithms can be expressed as sets of rules as well as sequences of steps
Data	w. Identify a range of test data for an algorithm i.e. valid, invalid, erroneous, boundary, and extreme, and predict what the program will do when it receives invalid, erroneous, or extreme data
	x. Identify the flow and control of data in an algorithm

3. Patterns and generalisation	
Theme	**Learning objective**
Identifying, adapting, and reusing solutions	a. Identify patterns and commonalities
	b. Recognise that one problem can have multiple or different possible solutions
	c. Adapt solutions, or parts of solutions, so they apply to a whole class of similar problems
	d. Identify common problems and the related common solutions
	e. Identify differences in problems and understand which solutions are not appropriate for applying/reusing
Predicting	f. Predict the outcome of an algorithm or process drawing on prior knowledge of similar programs and blocks of code
	g. Predict the outcome of an algorithm or process using clues and tracing code
	h. Understand the limits and drawbacks of prediction strategies and use to inform use of them
Explaining	i. Explain the generalisations or patterns used in a program/solution
	j. Transfer ideas and solutions from one problem area to another

4. Abstraction	
Theme	**Learning objective**
Abstracting problems	a. Recognise what is important in a solution and focus on only that
	b. Reduce complexity by removing unnecessary detail
	c. Hide the full complexity of instructions or systems (hiding functional complexity)
	d. Choose a way to represent an artefact, to allow it to be manipulated in useful ways
	e. Hide complexity in data, e.g. by using data structures
	f. Identify relationships between abstractions
	g. Filter information when developing solutions
	h. Use decomposition to define and apply a hierarchical classification scheme to a complex system
Modelling	i. Discuss and give an example of the value of generalising and decomposing aspects of a problem in order to solve it more effectively
	j. Modelling the behaviour of a system (the rules)
	k. Create a notation or model of a scenario
	l. Know that how things are represented is often not how they really are

5. Evaluation	
Theme	**Learning objective**
Abstracting problems	a. Find and define problems
	b. Assess that a solution or system is fit for the purpose and the needs of the user
	c. Assess whether a solution or system does the right thing (functional correctness)
	d. Assess whether a product meets general performance criteria (heuristics)
	e. Design plans to test a range of data and interpret the results (testing)
	f. Assess whether the performance of a solution or system is good enough (utility: effectiveness and efficiency)
	g. Compare the performance of solutions or systems that do the same thing
	h. Step through processes or algorithms/code step by step to work out what they do (dry run/tracing), and recognise when they don't do as intended
Modelling	i. Use validation to decide the appropriateness of algorithms or processes
	j. Use rigorous argument to justify that an algorithm works (proof)
	k. Use rigorous argument to check the usability or performance of an artefact (analytical evaluation)
	l. Use methods involving observing an artefact in use to assess its usability (empirical evaluation)

5. Evaluation cont.	
Exploring alternatives	m. Recognise that a program may be written in different ways but achieve the same outcome
	n. Regularly look for a 'better way' to solve the same problem
	o. Compare two programs that solve the same problem, in terms of: • Efficiency • User experience • Use of computer resources
Social and ethical norms, user experience	p. Know the social and ethical issues around the creation/use of computational products
	q. Consider the specific needs and limitations of a range of potential and actual users of systems and software
	r. Assess whether a solution or system is easy for people to use (usability)
	s. Assess whether a solution or system gives an appropriately positive experience when used (user experience)
	t. Make trade-offs between conflicting demands
	u. Understand how systems impact privacy and other human rights, through intended or unintended consequences

6. Data	
Theme	**Learning objective**
Organising data	a. Find and define problems
	b. Assess that a solution or system is fit for the purpose and the needs of the user
	c. Assess whether a solution or system does the right thing (functional correctness)
	d. Assess whether a product meets general performance criteria (heuristics)
	e. Design plans to test a range of data and interpret the results (testing)
	f. Assess whether the performance of a solution or system is good enough (utility: effectiveness and efficiency)
	g. Compare the performance of solutions or systems that do the same thing
	h. Step through processes or algorithms/code step by step to work out what they do (dry run/tracing), and recognise when they don't do as intended
Modelling	i. Use validation to decide the appropriateness of algorithms or processes
	j. Use rigorous argument to justify that an algorithm works (proof)
	k. Use rigorous argument to check the usability or performance of an artefact (analytical evaluation)
	l. Use methods involving observing an artefact in use to assess its usability (empirical evaluation)

ISSN 2514-586X
www.raspberrypi.org

Raspberry Pi