AIMS

Craig'n'Dave programming resources contain over 118 programs, including over 30 sample programs to try and investigate, 60 comprehension questions and over 80 differentiated problems to explore and solve.

The aim of these resources is not to cover every theoretical aspect of the specification, but instead to provide students with an independent self-study guide to learning the fundamentals of programming to become proficient at the basics.

Once students have mastered the fundamentals, they can then apply this knowledge to other areas of the specification. The algorithms for searching, sorting, optimisation and higher-order data structures are covered as practical activities in Craig'n'Dave theory resources.

PEDAGOGY

The Craig'n'Dave approach to teaching programming is revolutionary. Not because it is a new pedagogy, but because it is the first set of resources to truly unify the proven pedagogies for teaching programming effectively. Wrapped into one scheme of learning for teaching and learning the fundamentals of programming in Python and C#, our resources are deeply rooted in these methodologies:

- ## Barak Rosenshine's principles

  Introducing new material in small steps, asking questions, providing models of code and guiding student practice with progressive, logical learning. Checking understanding with an emphasis on teacher-student review, oral feedback and obtaining a high success rate with points-based challenges. Independent practice is central to these resources where students learn programming for themselves.

- ## Sue Sentence's PRIMM model

  Predicting, running, investigating, making and modifying programs as a practical approach to learning and studying code has been adopted and streamlined to "T.I.M.E." in these resources, because becoming proficient at programming takes time and that acronym just makes more sense!

- ## Carsten Schulte's block model of program comprehension

  Comprehending how code works by identifying key programming terminology, recognising structures and syntax, describing the purpose of algorithms, and considering alternative approaches. We have distilled Schulte's work on the duality of structure and function with atoms, blocks, relations, and macro structures in code to something secondary students can understand. Essentially studying program source code and asking structured questions about it.

- ## Dale Parson's code sorting

  Supporting weaker programmers by providing them with all the code they need to create a working program but jumbling the statements up. Creating a working program becomes a card sorting exercise. Scaffolding learning, reducing the comprehension required and enabling a trial and error approach.

- ## Alan O'Donohoe's stepped challenges

  Providing students with the desired output instead of the input. Algorithms can be created in many ways. Encouraging students to adopt paired programming techniques to discuss and justify their approaches with each other before learning from other pairs. Using differentiation by outcome to uncover deeper theoretical conversations about the merits of the different approaches taken.

- ## Richard Pawson and the functional programming paradigm

  Independent routines are as critical as sensible variables names and comments for creating well structured, modular programs. All programmers should start by learning to make subroutines from the very beginning of the course. Understanding the necessity for reusable components and parameter passing, not as an advanced topic, but one that is essential to the structure of all programs.

- ## William Lau's little book of algorithms

  Building fluency in programming by giving students many different and differentiated challenges. Recognising that repetition reduces the cognitive load by committing fundamental concepts of programming and keywords to long term memory. Working on small, standard algorithms that repeatedly demand sequence, selection and iteration creates more confident programmers.

- ## Craig and Dave's design by doing

  The traditional software engineer understands the system life cycle. Analyse the problem and design a solution before you begin to write code. While essential in many historical contexts of programming, rapid and iterative design practices today negate the need for this approach. Instead students learn best by writing real programs. Benefitting from instant feedback modern compilers and run-time environments provide.

  Programming taught from the front of the class by a teacher only benefits at best a third of the class. Some can already progress further and some already need additional support. Allowing students to learn independently at their own pace and choosing their own challenges allows greater flexibility for the teacher to stretch and support individual students.

A SELF-STUDY APPROACH TO LEARNING

The intention with these resources is that students make progress independently. New concepts are introduced through practical activities by engaging with sample code. Open-ended problems are sufficiently differentiated with a points system to enable all learners to make progress independently at their level. There is no need for a teacher to stand at the front of the class and teach the keywords and concepts in each objective in a traditional way. Students should make progress through the resources on their own, at their own pace. The role of the teacher is to maintain that pace, provide individual interventions when students are stuck, review completed objectives and track progress.

Using these resources, students that are absent from class are not disadvantaged and can even continue their work at home. If a question is too difficult to answer, or a problem is too difficult to solve these can be left to be discussed with the teacher. The student can still move on to the next objective or problem.

FUNCTIONS FIRST

The use of comments and sensible variable names is expected from students from the outset. It therefore seems odd not to also introduce structured programming using subroutines from the start too.

This does undoubtably mean the initial small programs students write become unnecessarily complex, but good habits introduced early are not so easily forgotten.

Although avoiding input and output within a function is desirable, for simplicity this is sometimes used. It is also regularly seen in exam mark schemes.

PRIOR KNOWLEDGE

Craig'n'Dave resources assume no prior knowledge, so they are suitable for all learners regardless of their programming experience. Students that have studied a text-based language at Key Stage 3 and/or GCSE will no doubt find the objectives familiar, although it is less likely they have adopted a functions first approach.

The Python resources are written with GCSE in mind, with C# being more suitable for A level. It is extremely unlikely that a GCSE student will complete every problem presented in these resources due to the volume of work and because of the differentiated approach. At A level it is suggested students use the same resources in an unfamiliar language, perhaps undertaking some of the problems they did not solve at GCSE.

.

NAMING CONVENTIONS

Although experienced programmers use different naming conventions for different purposes within a program, such as PascalCase, camelCase, snake_case and kebab-case, Craig'n'Dave resources use PascalCase throughout for consistency.

With PascalCase, each letter of a new word in an identifier is uppercase.

BREAKS AND OPTIMAL SOLUTIONS

Although commands exist in many languages to break out of iterations or jump to new sequences, these are not considered good practice for creating structured code. Therefore break, goto and equivalent commands are never used.

Many programs can be written more elegantly with modern functions and frameworks. However, this is an introduction to programming that also prepares students for the algorithms they will see in exams. Therefore, logical instead of optimal code is often used. E.g. using .sort() in answer to a question about sorting algorithms gains no marks!

OLD AND NEW APPROACHES

It is recognised that different exam boards exemplify slightly different keywords and approaches that students could be using when coding. An example is operator overload concatenation or string formatting.

```
Sol = 299792458
OUTPUT("The speed of light is " + Sol + "m/s")
```

older approach and less performant

Could also be written like this:

```
Sol = 299792458
OUTPUT("The speed of light is {0} m/s",Sol)
```

contemporary approach and more performant

The Craig'n'Dave approach is to use a coding style that primarily matches exam board expectations but with more modern approaches being a consideration too. These are often not easy decisions to make when there are many possible ways to write a program! We introduce students to both operator overloading concatenation (in objective 1) and string formatting (in objective 3).

KEY TERMINOLOGY

"The syntax error was because of a missing string qualifier before the operator in the statement that concatenates the two variables."

Students become far better programmers if they understand what statements like this mean. In teaching we are encouraged to consider reading age and simplifying language for comprehension. With programming, by having a commanding vocabulary of the words associated with code, not only does it enable students to understand other programmers and articulate their approaches, but it also helps to understand new concepts. For example, if you know that an operator can add two integer variables, then understanding overloading operators becomes easier too.

Throughout the resources each new term is explained and summarised.

10 OBJECTIVES

Students are introduced sequentially to each new programming construct. We call these stages "objectives" so that the language of learning is familiar.

Each objective builds-on and uses the keywords only introduced in the current and previous objectives to ensure a smooth and gradual learning transition between the objectives for students.

With a functions first approach, inputs are introduced much later than you might expect in favour of structured programming, arguments, and parameters.

Objectives:

1. Learn how to write structured programs.
2. Learn how to use selection.
3. Learn how to use number data types.
4. Learn how to use string data types.
5. Learn how to use counter-controlled iterations.
6. Learn how to use condition-controlled iterations.
7. Learn how to handle user inputs.
8. Learn how to use arrays and lists.
9. Learn how to use serial files.
10. Learn how to master the basics.

Objective 10 is an ever-increasing set of problems with a new one being released periodically.

Students will probably have studied most of these concepts at Key Stage 3, but these resources will not be too easy for them! The level of rigour is higher than that expected at Key Stage 3, especially with a functions first approach and validation of inputs. Repeating known concepts and consolidating knowledge firmly is extremely helpful. Even if students used these resources at GCSE, at A level they should attempt them again from objective 1 using a different language. E.g. Python at GCSE and C# at A level. Understanding the similarities and differences between different procedural languages is beneficial to learners wanting to study Computer Science beyond school.

4 STAGES IN EACH OBJECTIVE

Each objective has four stages for students to complete.

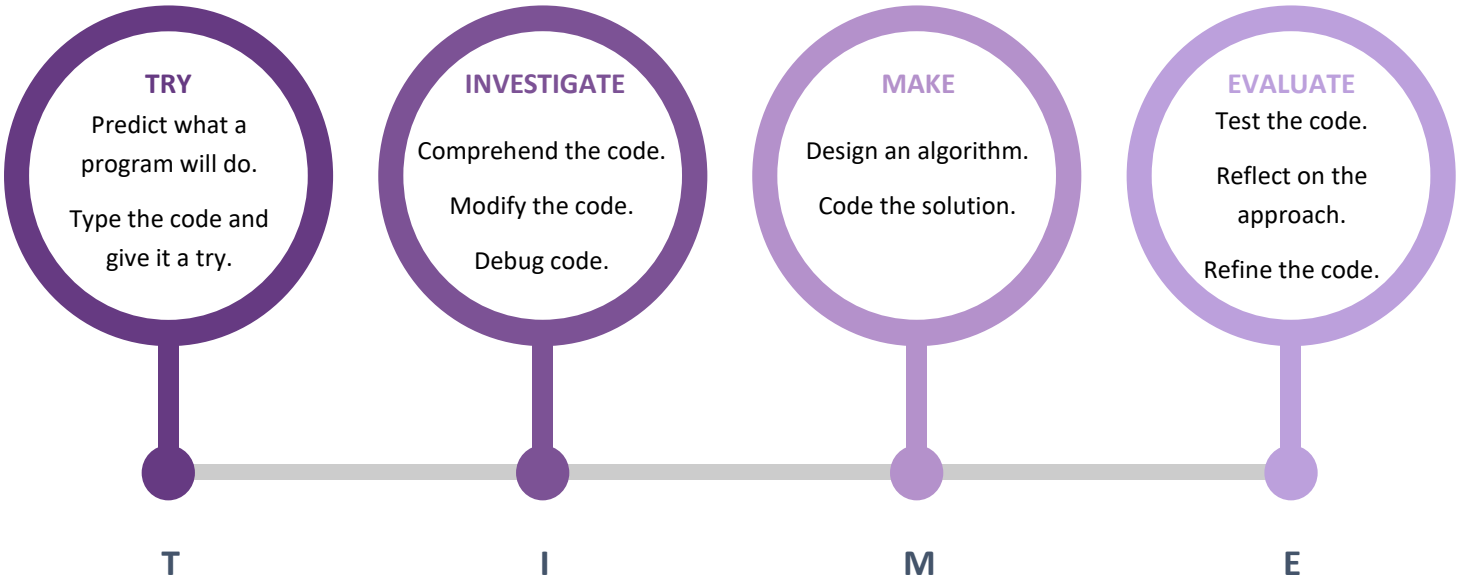
T - Try

I - Investigate

M - Make

E - Evaluate

Craig'n'Dave call this their "TIME" approach to programming. It is based on the PRIMM model proposed by Sue Sentence.



WORKBOOKS

Each objective is presented in a PowerPoint student workbook.

Activities for the students are written in the notes section of each slide.

Using PowerPoint enables the teacher to project slides if that is useful and output them in a variety of formats. They also work well with learning platforms, e.g. Google Classroom.

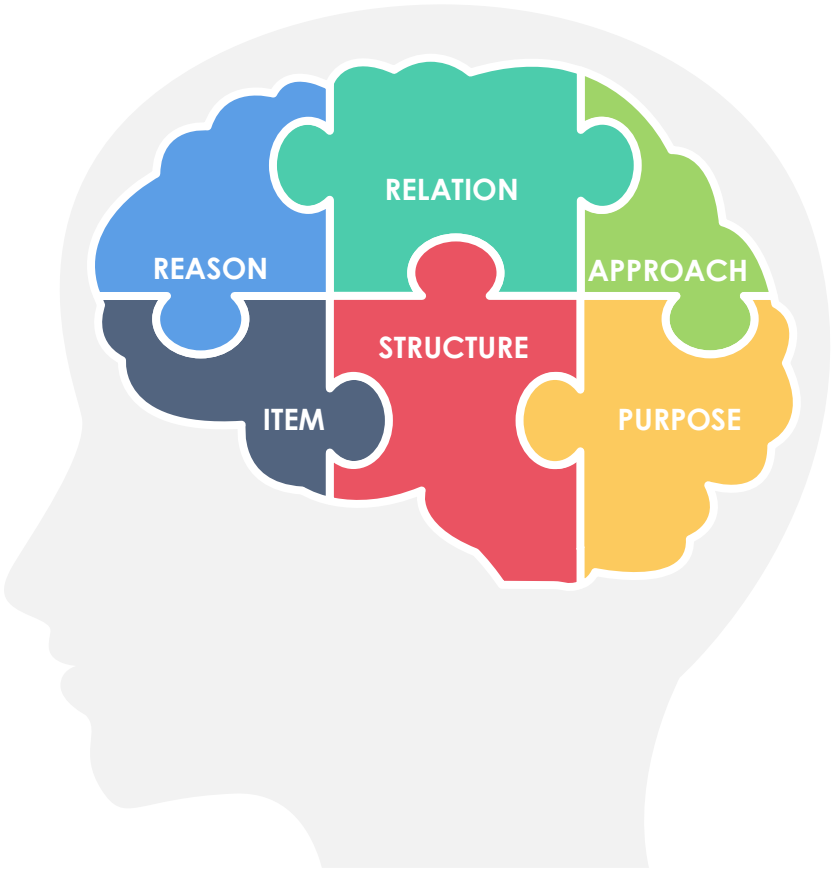
Try stage

Students look at a coded solution and predict what the output will be. Students type in the code to see if they are correct. This approach aims to simulate how self-taught programmers in the 1980s used code listings in magazines and programming manuals to learn to code. Not starting from a blank screen is less daunting for students, provides models and guides student practice.

Investigate stage

Students learn how the sample programs work, understanding the new commands introduced. A set of small tasks instruct students to modify the sample programs they have been given. At the end of the stage keywords and learning points are summarised as a handy reference. The syntax of related additional commands is also presented that were not needed in the sample programs but could be useful to students when writing their own programs. These also include all the commands listed in examination board specifications.

A set of progressive comprehension questions also provide students with an opportunity to consider what terms mean, what commands or blocks of code achieve, how they work and why they are used. A deeper knowledge of syntax and programming constructs allows students to adapt to new commands more easily. This is based on a simplified student-friendly version of the block model of program comprehension proposed by Carsten Schulte.



ITEM

Programming terminology and keywords.

STRUCTURE

Syntax of lines and blocks of code.

PURPOSE

What the item or structure achieves, returns, or outputs.

REASON

The reason why an item or structure is used.

RELATION

How items or structures relate to each other. The wider implication for the program or computer system.

APPROACH

How a section of code can be modified to achieve a greater purpose.

Program comprehension questions

ITEM	Programming terminology and keywords. Delimiter, qualifier, identifier, literal, operator, variable, constant, reserved word etc.	Q: In the command print("hello world") what is hello world known as? A: String. Q: In the command print("hello world") identify the string. A: hello world.
STRUCTURE	Identifying a block of code, syntax of an item or providing an example of a line of code.	Q: What character is used to identify (or qualify) the start and end of a string? A: double quote (sometimes a single quote).
PURPOSE	What the item or structure achieves, returns or outputs.	Q: What does the command print("hello world") do? A: Prints the words hello world to the screen.
REASON	Why an item or structure is used.	Q: Explain why the " character must be used. A: Without a string qualifier the compiler will assume hello is a variable, generating a syntax error.
RELATION	How items or structures relate to each other and the wider implication for the program or computer system.	Q: What are the advantages and disadvantages of using the command print("hello world") instead of using: txt = "hello world" : print(txt) A: There are less FDE cycles used but the string cannot be used later with other commands because it is not stored in memory. Q: What is the implication of changing the order of the commands to: print(txt) : txt = "hello world“ A: Whatever is currently stored in txt will be output or an error may occur.
APPROACH	How a section of code can be modified to achieve a greater purpose.	Q: Explain how the string can be sanitised to eliminate any spaces. A: By using a loop to iterate over each character in the string, concatenating it to a new string if it is not a space.

Make stage

Problems are presented in a mixture of written English, flowcharts, pseudocode, Parsons problems and output focused. Problems increase in difficulty indicated by the number of points. Students are encouraged to use whitespace, indentation, sensible variable names, and subroutines from their very first program.

Students should be using comments throughout their programs to explain the purpose of subroutines, variables, selections, and iterations. This will provide adequate problem decomposition making the explicit design of an algorithm using pseudocode redundant.

Back in the history of Computer Science when IDEs did not exist, programs were written on tape or card and it took a day to execute within a queue of other programs, it made sense to design algorithms to ensure they were robust! This is no longer necessary to become a good programmer. The small bite-sized programs used to teach programming are not large enough to warrant a design stage.

If students struggle to see solutions to the problems and therefore struggle with problem decomposition, they could produce all their comments first and fill in the code required between the comments afterwards, much like using pseudocode.

We suggest students choose the problems they want to solve, aiming for a total of 6 points in each objective. This provides for student choice and differentiation. Students who find programming more difficult could achieve 6 points from: 1 + 1 + 2 + 2-point problems. More able students could achieve 6 points from: 3 + 3-point problems.

Evaluate stage

Objectives will frequently require students to create and use test tables to test their solutions to the problems. This encourages good practice and teaches the importance of robust code.

Once students have finished an objective, they should alert their teacher. This is an opportunity to have a conversation with the student and give oral feedback on the problems attempted. Immediate oral feedback will be far more useful to the student than written feedback.

See overleaf for a framework for feedback conversations.

In oral feedback conversations, consider with the student:

Comprehension

To what extent does the student understand the code they have written?

- Review the investigate slides in their workbook.
- Ask questions about lines of code they have written to solve problems, getting them to explain how and why their algorithms work.

Maintainability

To what extent and how consistently has the student used best practices in creating readable code?

- The use of comments, subroutines, sensible identifier names and whitespace.
- Using code structures that are easy to understand.
- The use of the most appropriate iteration: counter or condition (from objective 6).

Scalability

To what extent could subroutines be used in other programs later and how well would the program perform if the data set it uses is increased significantly?

- Using subroutines and iterations instead of repeating blocks of code.
- Using self-contained subroutines with local variables and functions that return values.
- Using arrays and lists instead of multiple variables (from objective 8).

Robustness

To what extent can the program easily crash?

- Using validation (from objective 7).
- Using exception handling (from objective 9).

Approach

To what extent is the code the best algorithm for solving the problem?

- Creating time efficient algorithms (minimising the CPU cycles).
- Creating space efficient algorithms (minimising the use of memory) including using global variables only when it makes sense to do so.
- Alternative algorithms may also be considered even though they do not gain any significant advantage to appreciate the different approaches programmers might take and why.

OVERVIEW SCHEME OF LEARNING

	Objective	Content	Try/Investigate programs	Make 1-point problems	Make 2-point problems	Make 3-point problems
1	Learn how to write structured programs	Functions Parameters Variables Constants Concatenation	Hello World Discount Flow rate Square number	Dice face 5 Temperature converter Characters	Fish tank volume Microscopy Carpet cost	Energy bill calculator Circle properties Ball pit
2	Learn how to use selection	If Else Switch	Check age Valid month Key Stage Sample rate	Driving test Max States of water	Career quote Currency converter Nitrate	Exam grade Periodic table Day format
3	Learn how to use number data types	Integers & decimals Casting Random Output formatting Mod Div	VAT Roll Dice Odd or even Operators	Save the change Polyhedral dice Clamp	Leap year Hours in a day Dice game	Divisible Dogs life Electric car
4	Learn how to use string data types	String manipulation functions	Uppercase Length of name Wolf in the forest Find string Replace string	Tweet Initial & surname Inventory	Airline ticket Teacher code Valid address	Name separator Naming conventions ASCII to EBCDIC

5	Learn how to use counter-controlled iterations	For Foreach in Step	Repeating code Letters in string Countdown	Times table Factorial Ten green bottles	ASCII art FizzBuzz Scrabble	Passcode Cassini Prime number
6	Learn how to use condition-controlled iterations	While Do	Roll a six LCD Model virus Infinite loops Denominator	Compound interest Car value Discount counter	Lottery Cashpoint Square root	Denary to binary Happy numbers Predator-prey
7	Learn how to handle user inputs	Input Sanitisation & validation	Menu choice Validation Dice roll Metric-Imperial	Username Automatic feeder Guess the number	Conversion utility PIN Adder	Password Car park Rock paper scissors
8	Learn how to use arrays and lists	Array List	The quick brown fox Product database Pocket Letter grid	Quote of the day RPG inventory Notebook	Proc gen Underground Days of Christmas	Maths test Tanks Strong numbers
9	Learn how to use serial files	Read, Write Try Catch Strip	Write a line of data Read a line of data Read multiple lines Perf Counters	Div zero Cookie Attributes	ROT13 Time sheet Gamertag	Shopping list Vending machine Amino acids

TRACKING PROGRESS

A spreadsheet checklist to track problems students have attempted is included. We recommend entering the number of points achieved when a problem is solved so that students can see a running total of the points they have accumulated. This can make for some healthy in-class competition too!

We do not track progress on the try and investigate tasks. These are simply for the students to learn and experiment with the keywords introduced in the objective.

MORE FROM CRAIG'N'DAVE

Increasing number of challenges

Objective 10 will continue to be adapted to include more challenges for students to attempt. It is worth downloading a new copy of Craig'n'Dave resources every year to ensure you have the most up-to-date version.

Different languages

These resources are available for Python version 3.x and C# (Visual Studio 2015 onwards).

Visual Basic and Java will be available in the future.

Object-oriented programming

Use Craig'n'Dave Defold tutorials for Lua to teach object-oriented approaches with 2D games.

Event-driven & GUI programming

In the future Craig'n'Dave will create a Tkinter and Windows Forms extension to these resources.