Programming

What is programming?
Programming

WHAT IS PROGRAMMING?

Programming is the process of designing and writing a set of instructions (a program) for a computer in a language it can understand.

This can be really simple, such as the program to make a robot toy trace out a square; or it can be incredibly sophisticated, such as the software used to forecast the weather or to generate a set of ranked search results.

Programming is a two-step process:

- First, you need to analyse the problem and design a solution. This process will use logical reasoning, decomposition and generalisation to develop computational abstractions which capture the right level of detail about the state and behaviour of the system as well as algorithms that can solve the problem correctly and efficiently.

- Secondly, you need to express these ideas in a particular programming language on a computer, making use of data structures so the program can manipulate information. This might sometimes be called coding, and we can refer to the set of instructions that make up the program as ‘code’.

Coding provides the motivation for learning computer science – there’s a great sense of achievement when a computer does just what you ask it, because you’ve written the precise set of instructions necessary to make something happen. Coding also provides the opportunity to test out ideas and get immediate feedback on whether something works or not.

What should programming be like in schools?

It is possible to teach computational thinking without coding and vice versa, but the two seem to work best hand-in-hand.

Teaching computational thinking without giving pupils the opportunity to try out their ideas as code on a computer is like teaching science without doing any experiments. Similarly, teaching coding without helping pupils to understand the underlying processes of computational thinking is like doing experiments in science without any attempt to teach pupils the theories which underpin them.

This relationship is reflected in the new computing curriculum, which states that pupils should not only know the principles of information and computation, but should also be able to put this knowledge to use through programming. One of the aims of the national curriculum for computing is that pupils can:

analyse problems in computational terms, and have repeated practical experience of writing computer programs in order to solve such problems.

At primary school, pupils will have been taught how simple algorithms are implemented as programs on digital devices, from floor turtles to distant webservers. They will have had the opportunity to create and debug their own programs, as well as predicting what a program will do. They will have been taught to design and write programs that accomplish specific goals, which should include controlling or simulating physical systems. They should have learnt to use sequence, selection and repetition in their programs, as well as variables to store data. They should have learnt to use logical reasoning to detect and fix the errors in their programs. All this is likely to have been in the context of device-specific languages for programmable toys such as the Bee Bot and then visual, block-based programming languages such as Scratch.

The curriculum requirements at Key Stage 3 are more ambitious, as they were written with progression from Key Stage 2 in mind. In the early days of the new curriculum this led to some challenges in implementing computing at Key Stage 3, but secondary schools can now expect their new Year 7s to have learnt a fair amount of programming already, typically through Scratch.

Programming at Key Stage 3 should include working with real-world problems and physical systems, with an emphasis on teaching pupils how to develop computational abstractions which model the state and behaviour of such systems. Pupils are also expected to solve a variety of computational problems: look to provide as diverse a range as possible of contexts here for pupils’ programming.
including cross-curricular opportunities arising out of the other subjects pupils are studying.

Moving beyond the visual programming pupils will have studied at primary school, in Key Stage 3 they are taught at least two programming languages, at least one of which should be text based see pages 45 - 54 for some thoughts on the choice of language.

It’s fine to continue to work in Scratch for much of Key Stage 3, although anecdotal evidence suggests that some pupils have become somewhat bored with Scratch after much focus on this in primary computing lessons, however other visual languages are available which extend Scratch’s functionality, such as Snap! or the semi-visual TouchDevelop.

There seems evidence to suggest that it is effective to introduce a text-based language alongside a visual language at the beginning of Key Stage 3 (Dorling and White, 2015), using the familiar blocks to scaffold pupils understanding of the semantics of their programming whilst they become increasingly fluent in the syntax of the text-based language (Shneiderman and Mayer, 1979). There are very few things in the expectations for programming or computational thinking at Key Stage 3 that cannot be accomplished in a visual-programming language, so a sensible approach to take would be not to rush to take on the additional cognitive load of programming in a text-based language until pupils’ understanding of the underpinning ideas is very sound (qv Robins, 2010).

Strange as it may sound, teaching pupils to program need not always involve them programming. It is vital that pupils think about their solution before they start coding it, perhaps documenting their solution as pseudocode or a flowchart. Evidence from higher education and emerging models of effective practice in schools suggest that pupils need to be able to understand code before they can write code – give them programs in block – or text-based languages and ask them to trace them through, using logical reasoning to predict what would happen when the code was run. It is often less daunting for pupils to be given skeleton code to edit or buggy code to fix than to have to start from a blank screen. Pair programming is a proven, effective development methodology in the software industry and is likely to have its place in the classroom too. Similarly, reviewing code written by their peers helps develop evaluation skills and gives pupils more experience of reasoning about code (see Grover, 2016).

In his guide to Decoding the programmes of study for computing document, Simon Peyton Jones suggests the following approaches to teaching programming (Peyton Jones, 2014):

- **Simply experimenting with the medium.** Programming environments like Scratch and Kodu make it easy to try things out in a playful, exploratory way: ‘I wonder what happens if I press that button/drag that shape?’. At this stage the goal is to experiment, gain confidence that nothing bad will happen, and to gain intuition about what happens. It’s rather like a toddler playing with building bricks.

- **Simply copy an existing program, run it and then start making small changes to it.** The program solves the ‘blank sheet of paper’ problem. Some changes are limited but fun (for example, change the colour of the monster). As confidence builds, pupils will become more ambitious (for example, can we have more than one monster?).

- **Start to predict what a change will do.** One important aspect of computational thinking is to be able to predict what a program will do or what effect a change to the program will have. For simple, straight-line programs (that is, a simple sequence of instructions) this is pretty easy; the more complicated the program, the harder it gets. But at every level the ability to reason logically about the program is key. (The phrase ‘reason logically’ about programs is used in Key Stage 1, 2, and 3 of the Programmes of Study.)

- **Debug a program that is not working properly.** For example, if you want to draw a square with a floor turtle, you might forget to put the pen down so the turtle crawls around but doesn’t draw anything. Debugging always involves coming up with a guess (or hypothesis) about what is going wrong, performing experiments to confirm the guess and making a change that you predict will fix it. (1)

- **Explain to someone else how/why your program works.** The simple act of explaining often reveals latent bugs in your program or potential simplifications to your code.

---

1 Pupils often try to debug programs by making random changes, unsupported by any reasoning, and running the changed program to see if it behaves better. This isn’t computational thinking; it’s simply guesswork.
● **Read a program and figure out its purpose.** For example

```
T := 0
for I = 1..N { T := T + I }
```

You could talk about loops and variables, but an experienced programmer would say ‘oh, that just adds up the numbers between 1 and N, and puts the total in T’. That is, she has worked out the **purpose** of the code, rather than just following the individual steps it takes.

● **Starting from an idea of what you want your program to do, write a program from scratch to do it.**

---

**Classroom activity ideas**

● Get pupils to ‘reverse engineer’ some of the programs they use. For example, they might think through the different states a simple system such as a smart TV remote or a digital microwave oven can be in, and how one links to another (an example of a finite state machine, see page 26). Or they could think through what algorithms have been coded into simple or more complex games they play. Or pupils could think about how complicated the code would have to be for familiar application software such as Word or PowerPoint. Draw their attention to how the model-view-controller design pattern has been applied to many of these examples.

● Look for (or create) some simple programming exercises which focus on particular learning objectives. For example, when teaching pupils how a sequence of steps in an algorithm can be translated into code, give pupils a simple algorithm (for example to draw a regular pentagon) and set them the challenge of implementing this as code.

● Set some extended programming projects in which pupils can work through the process of software development, from original design through writing code to testing and debugging their programs.

● Here are some ideas for extended programming projects:

  » **Year 7:** turtle graphics in two languages; creating an animated dance routine; developing a game for the BBC micro:bit; developing a maths quiz for primary pupils in a feeder school.

  » **Year 8:** write a program to encrypt and decrypt text using a shared key; program a robot to find a path through a maze; create a simple chatbot; investigating recursion (for example fractals using turtle graphics).

  » **Year 9:** composing music, storing and retrieving information in a database, analysing a large public data set, mobile phone app development. (Dorling and Rouse, 2014)

---

**Further resources**


BBC Bitesize (n.d.) *Controlling physical systems*. Available from [www.bbc.co.uk/guides/zxjsfg8](http://www.bbc.co.uk/guides/zxjsfg8)

BBC Cracking the Code (n.d.) For examples of source code for complex software systems such as robot footballers and a racing car simulator. Available from [www.bbc.co.uk/programmes/p01661pj](http://www.bbc.co.uk/programmes/p01661pj).

Computing at School (2016.) CAS Chair, Prof. Simon Peyton Jones’ explanation of some of the computer science that forms the basis for the computing curriculum. Available from [http://community.computingatschool.org.uk/resources/2936](http://community.computingatschool.org.uk/resources/2936)


How do you Program a Computer?

The code is the set of instructions needed to make the computer do what you want, and this has to be written in a programming language which the computer understands. There are many languages to choose from. Some of these will work on only particular devices, or are only intended for a small set of particular purposes. Some languages have been developed specifically with children or other new programmers in mind, whereas others would be best left to professional software developers working on complex projects (although some more confident and motivated Key Stage 3 pupils might relish the challenge of mastering such a language). In some languages you write a program as a sequence of commands for the computer to execute, in others you might create classes of objects with particular properties that can interact with one another, or sets of functions, each of which will produce certain output from given input.

Programming languages are formal and have to be used in precise ways. Programs are made up of precise, unambiguous instructions – there’s no room for interpretation or debate about the meaning of a particular line of computer code. You are only able to write code using the clearly defined vocabulary and grammar of the language, but typically you do so using words taken from English, so code is something which people can write and understand but the computer can also follow. Each programming language will have its own compiler or interpreter, written, or at least customised, for the particular system on which it runs. This takes the code written in the programming language and converts that, either in one go or a bit at a time, into the sort of instructions which the computer’s central processing unit (CPU) can follow. We call these instructions ‘machine code’: the commands here are very simple ones but modern processors can execute these at very, very high speed.

At Key Stage 3, the details of compilers and interpreters are unlikely to feature significantly, and often other layers of abstraction are present between the programming language itself and the CPU – Scratch programs run inside a Flash runtime environment within the web browser and Snap!, TouchDevelop and the Trinket version of Python all are interpreted as JavaScript, itself being executed within the web browser that runs on the CPU itself.

Programming the BBC micro:bit perhaps gives pupils more of an insight into the detail here, with programs in TouchDevelop, Code Kingdoms and the blocks editor being converted to runtime machine code (the .hex file) by a JavaScript compiler running in the web browser. Python on the micro:bit works a little differently: it flashes a micro-Python interpreter onto the micro:bit, which then reads and interprets the Python source code that’s flashed to the micro:bit.

How are instructions stored and executed?

You can get a feel for what machine code is like through emulators such as the Little Man Computer (LMC). This abstraction captures the fundamental architecture of modern computers well, in which data and instructions are stored side by side in main memory, with a central processor fetching instructions, executing them and then receiving or outputting data depending on what those instructions are (called Von Neumann architecture, after the designer of ENIAC (Electronic Numerical Integrator And Computer), one of the first electronic stored program computers which, like most computers after it, adopted this approach).

---

3 From [www.microbit.co.uk/offline](http://www.microbit.co.uk/offline)
The LMC abstraction models a computer as if it were a ‘little man’ in a closed room with just 100 mailboxes (the memory) at one end of the room and two further mailboxes, input and output at the other end. In the middle of the room, a calculator (the accumulator) allows addition and subtraction to be done, and there’s a resettable counter which points to the mailbox where the next instruction will come from. Normally, the counter increases by one each time, so instructions are worked through in sequence, although branch and repeat instructions can change this. Programming the LMC involves putting a sequence of instructions and the data into the mailboxes, setting the counter to zero and letting the LMC work through the instructions given.(4) A number of online LMC implementations are available.(5)

Modern processors are somewhat more complex than this and obviously run significantly faster, but the essential ideas of data and instructions being stored in memory, and simple instructions being executed one after another remain the same. Between fetching and executing instructions, the instructions are decoded, routing the data that follows to one of the many logic circuits which make up the CPU. Neil Brown puts it like this:

The processor runs a fetch-execute cycle. It fetches a single instruction from memory, which is then executed. For example, a LOAD instruction loads a value from memory into a processor register, an ADD instruction adds two registers, and a STORE instruction stores a value from a register back into memory. Once an instruction has been executed, the next instruction is fetched and executed. The number of instructions that can be executed in a second is known as the clock speed, so 1 Mhz is 1 million instructions per second.(6)

Although he goes on to explain that modern processors are in practice somewhat more complex. Apart from the simplest microprocessor controlled devices, modern computers are multi-core devices, with a number of CPUs each able to execute instructions independently of and in parallel with the others. Parallel computing, using many CPU cores simultaneously, can be used for applications such as graphics rendering, search and even simulating the brain.(7)

What programming languages should you use?

There are many languages to choose from. The majority are more complex than necessary for those still getting to grips with the ideas of programming but there are plenty of simple, well supported, general purpose languages that can be used very effectively in the lower secondary classroom. Try to pick a language that you will find easy to learn or, better still, know already.

Consider these points when choosing a programming language:

- Not all languages run on all computer systems.
- Choose a language that is suitable for your pupils. There are computer languages that are readily accessible to lower secondary pupils – in some cases this will mean one that has been written with pupils in mind or at least adapted to make it easier to learn, but well constructed general purpose languages should not be ruled out.
- Choose a language supported by a good range of learning resources. It’s better still if it has online support communities available, both for those who are teaching the language and those who are learning it.
- It is beneficial to the pupils if they can continue working in the language on their home computer or, even better, if they can easily continue to work on the same project via the internet.
- Think carefully about primary to secondary transition. If many of your pupils have been learning, for example, Scratch, whilst at primary school, then they can hit the ground running with some ambitious projects for Year 7 without having to learn the commands and interface of a new language. On the other hand, they might perhaps have become somewhat jaded with Scratch and be ready for something different now they are at big school.
Aim for depth rather than breadth – the computing curriculum is about learning the principles of computer science through practical programming rather than learning lots of different languages. Mastering one or two languages will mean pupils can start to tackle authentic problems, perhaps from elsewhere in the curriculum, with a degree of independence. Fluency in a couple of languages is far more useful than a vague familiarity with a dozen.

There is value in learning multiple languages and particularly different language paradigms, but there’s no need to rush into this at Key Stage 3 or even Key Stage 4: few of your pupils will become professional software developers but all ought to understand the basics of programming.

Mark Guzdial recently blogged about his own set of principles for choosing a programming language for teaching:

1. Connect to what learners know.
2. Keep cognitive load low.
3. Be honest.
4. Be generative and productive.
5. Test, don’t trust.

There’s a view that some languages are better at developing good programming ‘habits’ than others. Good teaching, in which computational thinking is emphasised alongside coding, through an emphasis on planning and reasoning about code, should help to prevent pupils developing bad coding habits at this stage.

---


9 Online at http://cacm.acm.org/blogs/blog-cacm/203554-five-principles-for-programming-languages-for-learners/fulltext; qv
https://computinged.wordpress.com/2016/06/20/how-to-choose-programming-languages-for-learners/
Visual Programming Languages

There are a number of graphical programming toolkits available: these make learning to code easier than it’s ever been. In most of these, programs are developed by dragging or selecting blocks or icons which represent particular instructions in the programming language. These can normally only fit together in ways that make sense and the amount of typing, and thus the potential for spelling or punctuation (syntax) errors, is kept to an absolute minimum.

With toolkits like these it’s easy to experiment with creating code. By letting the programmer focus on the ideas of their algorithm rather than the particular vocabulary and grammar of the programming language, programming and learning to program becomes easier and often needs less teacher input.

Kodu

Microsoft’s Kodu (Figure 2.3) is a rich, graphical toolkit for developing simple, interactive 3D games.

Each object in the Kodu game world can have its own program. These programs are ‘event driven’: they are made up of sets of ‘when [this happens], do [that]’ conditions, so that particular actions are triggered when certain things happen, such as a key being pressed, one object hitting another or the score reaching a certain level.

Figure 2.3 Kodu interface

Programmers can share their games with others in the Kodu community, which facilitates informal and independent learning. There’s also plenty of scope for pupils to download and modify games developed by others, which many find quite an effective way to learn the craft of programming. This can also offer pupils a sense of creating games with an audience and purpose in mind.

Scratch

In Massachusetts Institute of Technology’s (MIT) Scratch (Figure 2.4), the programmer can create their own graphical objects, including the stage background on which the action of a Scratch program happens and a number of moving objects, or sprites, such as the characters in an animation or game.

Figure 2.4 Screenshot of a Scratch program
Each object can have one or more scripts, built up using the building blocks of the Scratch language. To program an object in Scratch, you drag the colour-coded block you want from the different palettes of blocks and snap this into place with other blocks to form a script. Scripts can run in parallel with one another or be triggered by particular events, as in Kodu.

A number of other projects use Scratch as a starting point for their own platforms, for example ScratchJr is a tablet app designed for young programmers (Key Stage 1). There’s a great online community for Scratch developers to download and share projects globally, making it easier for pupils to pursue programming in Scratch far beyond what’s needed for the national curriculum. There’s also a supportive educator community, which has developed and shared high-quality curriculum materials.

The current version of Scratch (2.0) allows users to create their own new blocks built from Scratch’s commands, and allows parameters to be passed to these blocks. These custom blocks can display text on screen, change sprite or global variables and so on, but they cannot return values — that is, they are procedures rather than functions, this can be a limiting factor with Scratch.

Scratch is available as a free web-based editor or as a standalone desktop application. Files can be moved between online and offline versions. There’s some support for interfacing with hardware components, including webcams, and a way to extend Scratch functionality via an application program interface (API).¹⁰

Snap!¹¹ started life as build your own blocks (BYOB) – a fork of the Scratch code which, unsurprisingly, allowed users to create their own blocks. Scratch now has this functionality, albeit in a somewhat limited way, but Snap! has continued development, focussing on implementing some more sophisticated computer science ideas in a block-based language: perhaps the most immediate is offering proper functions that can return values to the program or function that called them (see Figure 2.5).

Snap!’s functions though are ‘first class’ citizens, that is, functions can be passed as arguments to other functions, so it’s relatively easy in Snap! to implement new control structures, such as ‘for’ loops or functional programming ideas such as a map function which applies the same user specified function to each of a list of elements. Snap! also supports anonymous ‘lambda’ functions. Snap! has better support for lists too, which can be useful when teaching data structures, including the ability to have lists of lists.

Snap! is implemented in JavaScript so runs in any browser, including those without Flash support such as on tablet computers. Like Scratch, there are a number of extensions and modifications available, including tools to import (but not export) Scratch projects, an exporter to create standalone applications and Edgy,¹² a version of Snap! designed for programming with graphs. On the downside, there is nothing like the vibrant, global user community of Scratch and far fewer teaching resources available. However, it is used as the teaching language for Berkeley’s Beauty and Joy of Computing course, which is also offered to US high school students as an AP CS (advance placement computer science) Principles course.¹³

Figure 2.5 Recursive implementation of quicksort in Snap!

¹⁰ http://scratchx.org/
¹¹ Introduction by John Stout for CAS TV at www.youtube.com/watch?v=7tjNnF4fAgl
¹² http://snapapps.github.io/
¹³ http://bjc.berkeley.edu/
Classroom activity ideas

- Pupils could develop a game in Kodu, taking inspiration from some of the games on the Kodu community site. As a starting point, tell them to create a game in which Kodu (the player’s avatar in the game) is guided around the landscape bumping into (or shooting) enemies.
- Pupils could take photographs of one another in a variety of dance poses then use Scratch to create a program which animates these to choreograph a simple (or complex) dance routine. They could add music to their program, either by importing an MP3 or by composing music in Scratch.
- Scratch lends itself to game programming and it can be a good platform for pupils to work on projects like this quite independently. Start by asking pairs or groups to plan their games very carefully, thinking through the rules of their games, which will be the algorithms used as the basis for their programs. As well as working creatively to design the media used in their games, pupils will need to think through how the user’s interaction with the game will work, tweaking their game to provide just the right level of challenge to the player. Games based on classic arcade games such as Pong, Pacman and Duck Shoot can be programmed in Scratch without too much difficulty.
- Pupils could implement fractions arithmetic in Snap!, treating fractions as lists with just two members. They would need to write helper functions to find highest common factors and simplify fractions, which could then be used in functions to implement addition, subtraction, multiplication and division.

Further resources


*Kodu Game Lab Community* (n.d.) Available from www.kodugamelab.com/


*Scratch* (n.d.) Available from http://scratch.mit.edu/


*Snap!* (n.d.) Available from http://snap.berkeley.edu/

Text-Based Programming Languages

Most software development in academia and industry takes place using text-based languages, where programs are constructed by typing the commands from the programming language at a keyboard.

Historically, text-based programming has been a real barrier for children when learning to code, and there’s no need to rush into text-based programming at the beginning of Key Stage 3. Python is by far the most common text-based programming language in secondary schools at the moment and this offers many advantages, as it is relatively easy to learn and sufficiently flexible to be used for general purpose, real-world development, but it’s worth looking at some of the alternatives as well.

Logo

Logo was developed by Seymour Papert and others at MIT as an introductory programming language for children. It’s probably best known for its use of
‘turtle graphics’ – an approach to creating images in which a ‘turtle’ (either a robot or a representation on screen) is given instructions for drawing a shape, such as:

```plaintext
REPEAT 4 [
    FORWARD 100
    RIGHT 90
]
```

Repetition can be nested, allowing relatively complex figures to be programmed quite easily:

```plaintext
REPEAT 10 [
    REPEAT 5 [
        FD 100
        RT 72
    ]
    RT 36
]
```

Figure 2.6

Papert saw Logo as a tool for children to think with, just as programming is both the means to and motivation for computational thinking.

In Logo programming, more complex programs are built up by ‘teaching’ the computer new words. These are called procedures. For example you could define a procedure to draw a square of a certain size using the key words of the language. Once this is defined typing it in will then result in the turtle drawing a square.

```plaintext
TO SQUARE :SIDE
    REPEAT 4 [
        FORWARD :SIDE
        RIGHT 90
    ]
END
SQUARE 50
```

Many associate Logo with these sort of turtle graphics programs. Turtle graphics are supported by most programming languages, including Scratch, Snap!, TouchDevelop, Small Basic and Python. Logo is, however, capable of more general programming, so for example factorials (the product of the integers up to and including a number, for example 5!=1x2x3x4x5=120) can be calculated using a recursive function in Logo:

```plaintext
TO FACTORIAL :NUMBER
    IF :NUMBER = 1 [OUTPUT 1]
    OUTPUT :NUMBER * FACTORIAL :NUMBER - 1
END
```

Logo’s original development grew out of Lisp, and thus it also has good support for lists.(14)

Whilst not a popular choice at present, the text-based programming requirements of the national curriculum could be met through Logo.

Microsoft Small Basic

Microsoft Small Basic is a simplified version of Microsoft’s Visual Basic programming language and associated environments, and indeed Small Basic programs can be exported to form the basis of more complex Visual Basic code. The language is text based, but with a development studio designed to help with many of the difficulties of text-based programming: thus there’s a built-in visual environment in which programs can be run and ‘IntelliSense’ is used to help suggest and complete the keywords of the language as you type. The language is kept deliberately small, with just 14 keywords, although these are supplemented through an extensive standard library, with support for turtle graphics as in Logo as well as external resources such as Flickr. For example, the program to draw a square in Small Basic would look something like:

```plaintext
Turtle.Show()
For i = 1 to 4
    Turtle.Move(100)
    Turtle.TurnRight(90)
EndFor
```

As with Kodu and Scratch, there’s an online gallery in which programmers can share the source code for their programs, with others using these as a starting point for their own work.

---

14 The classic introduction to programming in Logo beyond the realm of turtle graphics is Harvey (1997).
Typing code on a tablet computer or a smartphone is not easy and this can be problematic for schools that use these devices extensively.

Developed by Microsoft Research, Touch Develop is a programming language and environment, which takes into account both the challenges posed and the opportunities offered by touch-based interfaces such as those on tablets and smartphones.

Touch Develop makes it quite easy to develop an app for a smartphone or tablet on the smartphone or tablet itself.

Although Touch Develop is a text-based language, programmes aren’t typed but are created by choosing commands from the options displayed in a menu system. In this way, Touch Develop is a half-way house between graphical and text-based programming. Those who have become familiar with drag and drop or keyboard-based programming sometimes find it hard to adapt to the touch-optimised interface of Touch Develop.

As with Logo, turtle graphics commands are available as standard. On some platforms Touch Develop can also access some of the additional hardware built into the device, such as the accelerometer or global positioning system (GPS) location, allowing more complex apps to be developed: these can be hosted online as web-based apps or installed directly on the device if it’s a Windows phone. Touch Develop is one of the program editors provided for the BBC micro:bit (Figure 2.7).

A particularly nice feature of Touch Develop is the use of interactive tutorials to scaffold pupils’ learning of the language.

Whilst few would immediately think of it as a programming language, the Excel spreadsheet package is a text-based programming language, although admittedly a rather strange one (Peyton Jones et al., 2003). In Excel, rather than creating a sequence of instructions, you write code (Excel formulae) to create a system of interlinked functions, which take values in the spreadsheet cells, and return the results of performing computation on those functions. This provides some introduction to functional programming, as well as being really useful in all sorts of situations where a large amount of numerical data needs to be processed or where mathematical functions provide a good way to model a complex real-world problem. The way in which an Excel spreadsheet shows the values in each cell can help pupils visualise how this sort of computation is performed.

Computational thinking processes such as logical reasoning, abstraction, decomposition and generalisation apply just as much to developing a spreadsheet in Excel as they do to writing imperative programs in Scratch, Small Basic or the other languages discussed here. There are many real-world problems for which a spreadsheet may be the most efficient solution. It would be a shame for pupils to miss out on developing some fluency with this approach to solving computational problems.

For many secondary teachers and their pupils, Python seems a great introductory, text-based programming language. Papert argued that a good teaching language should have ‘low floors, wide walls and high ceilings’, and Python seems to offer all three:

- Whilst text-based programming inevitably introduces additional cognitive load over graphical languages, Python allows pupils to write programs with a similar structure to what they would write in Scratch, at least in the case of
Scratch programs made up of just one script and some custom blocks – support for multithreading, which is easy in Scratch, is much less straightforward in Python.

- Python has a good set of standard libraries which extend the functionality of the language, including a great implementation of turtle graphics,[15] game libraries[16] and support for developing programs with graphical user interfaces. There are many other libraries available publicly via the internet, together with tools to install these without too much difficulty. There is very good support for science, maths and statistics, plus libraries for natural-language processing, working with graphs and many, many other specialised areas. Python is installed as standard on the Raspberry Pi and macs, it can be run on Android phones and the BBC micro:bit. There’s also a good culture of folk sharing Python programs for a wide range of applications via Github.

- Python is a proper, grown-up programming language used for real-world software development in a range of domains. Whilst not an entirely functional programming language, Python supports functional programming. Whilst not an entirely object-oriented language, Python supports object-oriented programming. There is lots of interest in it in academic computing, including as a teaching language for computer science degrees as well as for science and humanities. It can be used for developing server-driven web applications, and is used by Google, Facebook, Yahoo, EventBrite, Reddit and NASA. Python programming skills are in demand for jobs in software development.

Here’s an example of a simple Python program to do a drill and practice tables test, which illustrates some of the features of the language:

```python
import random
for i in range(10):
    a = random.randint(1,12)
    b = random.randint(1,12)
    question="What is "+str(a)+" x "+str(b)+"? "
    answer = int(input(question))
    if answer == a*b:
        print("Well done!")
    else:
        print("No.")
```

If you’ve never seen Python code before this might be a bit daunting, but spend a couple of minutes reading through the code and you should get a reasonable feel of what’s going on here.

A few things to mention:

- The nice, indented layout here is a feature of the language – repeated code in the for loop is indented, as are the different bits of code that get executed in the if … else selection statement. Similarly the : which precedes these indented blocks are part of the language.
- The `randint` command (which picks a random integer from the given range) isn’t part of the Python language itself, and so needs to be imported as part of the `random` library.
- Variables in Python have implied types – a and b are integers, question is a string: functions allow variables to be converted between one type and another.
- In the `print` command the text to be printed is inside ( brackets ). This was one of the significant changes from Python 2 to Python 3, so it’s worth double checking which version of Python is running on your computer.

Don’t be too ambitious as you introduce pupils to programming in Python: learning any text-based language demands concentration and attention to detail, and this makes it hard for pupils to give lots of attention to mastering complex algorithmic ideas at the same time.

Starting with something familiar, such as turtle graphics, offers a nice way in. The Logo program on page 50 can be implemented in Python very easily, using the standard Turtle library:

```python
from turtle import *
for i in range(10):
    for j in range(5):
        forward(100)
        right(72)
        right(36)
done()
```

This is very similar to the same program in Scratch (Figure 2.8):

---

15 https://docs.python.org/3/library/turtle.html
And many pupils will be able to see the connections between the algorithm for their pattern and its expression as code in the two languages.

Another good introductory project is to get pupils to recreate the choose your own adventure games of old (Jackson and Livingstone, 1982), perhaps providing pupils with the skeleton code of a procedure for an individual ‘room’ and allowing them to adapt and expand on this:

```python
def room0():
    print ("You are in room 0.
There are exits here to room 1 and room 2.")
    choice = input ("Choose 1 or 2: ")
    if choice == "1":
        room1()
    elif choice == "2":
        room2()
    else:
        print("That's not one of the choices!")
        room0()
```

Adventure games are based on a binary tree graph as the computational abstraction, with the nodes being the rooms (the states), the edges the choices between them (the behaviour) at each point. With minor adjustments the same program could be turned into a ‘branching database’ classification key for a group of plants or animals.

### Some practicalities

Python is a free download and can be installed on Windows and Linux systems; Python is pre-installed on OS X. Python itself does not introduce any security risks to a properly configured system or network – it doesn’t need to run with administrator or root permission and thus can only modify files or directories to which the user already has write-access. An alternative to installing Python locally is to access a Python interpreter and editor via the web; this is useful for learning the language, but can make it difficult to access particular libraries or develop more complex software.

Downloading and installing Python brings you the Python interpreter itself and a simple integrated development environment (IDE) called IDLE. You can use IDLE to write, save and edit Python programs and to run them – the output of the code appears in another window (Figure 2.9).

<table>
<thead>
<tr>
<th>Figure 2.9 IDLE showing program editor, console and turtle graphics output on a Mac</th>
</tr>
</thead>
</table>

You don’t have to use IDLE to use Python. You can write Python code in any text editor, running the program you save at the command prompt or shell, or you can use more sophisticated IDEs. You can also use Python in interactive mode, either in IDLE’s console or at the command prompt/shell after just typing python. This can be useful for just experimenting with the syntax of the language rather than for writing programs.

---

17 See [http://community.computingatschool.org.uk/resources/446](http://community.computingatschool.org.uk/resources/446) for a discussion of some of the school infrastructure issues associated with providing pupils with access to programming tools.


19 For example Notepad ++, Atom or Mu.

20 Integrated development environments, for example PyCharm Edu or Visual Studio Python Tools for Windows.

21 Or using the Jupyter/iPython interactive notebook package.
Other programming languages are available, which could be used as introductory text-based languages in secondary schools, for example: JavaScript, Ruby, Pyret, Visual Basic, Swift (OS X only) and Java (perhaps using Greenfoot).

Classroom activity ideas

- Revisit the turtle graphics activities you might have been using for programming in the old information and communication technology (ICT) curriculum. Whilst these can be accomplished using the motion and pen commands in Scratch, projects such as drawing regular polygons, a simple house or complex repeating patterns such as ‘crystal flowers’ are usually well enough understood for pupils to get to grips with the additional challenges of text-based languages – there’s evidence that it can be effective for pupils to work in a visual- and a text-based language side by side for this (Dorling and White, 2015).

- Explore some of the commands and functions available in these languages for working with text. For example, can pupils write a program which takes any sentence and converts it into capital letter, or reverses the sentence, or removes all the vowels from the sentence, or reverses each word in the sentence?

- Explore how one or more of these programming languages could be used to simulate dice being rolled. In Excel, could pupils simulate rolling 100 dice at the same time and then draw a bar chart of the results? Ask them to think how they would do that in Scratch. Or can pupils create an app in Touch Develop which simulates rolling a dice when the phone, tablet or micro:bit is shaken? Ask pupils to think about how deterministic computers can simulate random events such as these.

- On the Raspberry Pi, Python can be used as a scripting language for Minecraft or for simple physical computing activities using the Raspberry Pi’s general purpose input/output (GPIO) pins.

Further resources


Python (n.d.) Available from www.python.org/ and online via trinket.io


Small Basic (n.d.) Available from www.smallbasic.com/

Touch Develop from Microsoft Research (n.d.) Available from www.Touchdevelop.com

What’s inside a Program?

Whilst the detail will vary from one language to another, there are some common structures and ideas which programmers use over and over again from one language to another and from one problem to another:

- **Sequence:** running instructions in order (see below and page 55).
- **Selection:** running one set of instructions or another, depending on what happens (see page 57).
- **Repetition:** running some instructions several times (see page 59).
- **Modularity:** building programs from smaller, independent blocks of code that return values or do specific things (see page 63).
- **Data structures:** organising data so that it can be stored and retrieved from the computer’s memory (see page 69).

These are so useful that it’s important to make sure all pupils learn these. Sequence, selection and repetition are introduced in Key Stage 2 of the computing curriculum, where pupils also learn about variables, simple data structures that handle just one piece of information.

This Scratch script (Figure 2.10) shows sequence, selection, repetition and variables. Can you work out which bit is which before we look at these ideas in detail?

```python
import random
for i in range(10):
    a = random.randint(1,12)
    b = random.randint(1,12)
    question="What is "+str(a)+" x "+str(b)+"? 
    answer = int(input(question))
    if answer == a*b: 
        print("Well done")
    else:
        print("Think again")
```

Further resources

**BBC Bitesize** (n.d.) How do we get computers to do what we want? (Covering sequence, selection and repetition). Available from [www.bbc.co.uk/guides/z23q7ty](http://www.bbc.co.uk/guides/z23q7ty)


Berry, M. (2016) Tables test. Python program, available from [https://trinket.io/library/trinkets/d15c8f972b](https://trinket.io/library/trinkets/d15c8f972b) [29/12/16]

**Bitesize programming materials** at [www.bbc.co.uk/education/topics/zhy39j6](http://www.bbc.co.uk/education/topics/zhy39j6)

**Cracking the Code clip** (n.d.) Available from [www.bbc.co.uk/programmes/p016j4g5](http://www.bbc.co.uk/programmes/p016j4g5)

Sequence

Programs are built up of sequences of instructions. In EYFS (early years foundation stage) or Key Stage 1, when pupils start programming with floor turtles, their programs consist entirely of sequences of instructions, built up as the stored sequence of button presses for what the floor turtle should do. As with any program, these instructions are precise and unambiguous, and the floor turtle will simply take each instruction (the stored button presses) and turn that into signals for the motors driving its wheels.

---

22 In imperative programming languages such as those discussed here. Declarative languages such as Haskell, F# and Excel work rather differently.
Pupils' first Scratch or Python programs are also likely to be made up of simple sequences of instructions. Again, these need to be precise and unambiguous, and of course the order of the instructions matters. In developing their algorithms, pupils have to work out exactly what order to put the steps in to complete a task. In more complex programs involving variables or other data structures, they will need to think through how the steps in their programs change the data stored.

Figure 2.11 A simple music program in Scratch

```python
when green flag clicked
set volume to 100%
set tempo to 72 bpm
set instrument to 1
play note 64 for 0.5 beats
play note 64 for 0.5 beats
play note 65 for 0.5 beats
play note 67 for 0.5 beats
play note 65 for 0.5 beats
play note 67 for 0.5 beats
play note 67 for 0.5 beats
play note 64 for 0.75 beats
play note 62 for 0.25 beats
```

A simple chat bot in Python.

```python
print("Hello!")
name = input("What is your name?")
print("It's a pleasure to meet you, " + name + ".")
print("What odd weather it's been of late.")
today = input("What have you been doing today?")
print("What a coincidence! I've been " + today.lower() + "+ too.")
```

Classroom activity ideas

- Ask pupils to experiment with programming Scratch to play music as in Figure 2.11 above. Take a simple, familiar melody, perhaps in score notation or just as a list of notes and have pupils translate this into a sequence of Scratch commands. Can pupils tell by ear where there are mistakes in their code? Pupils could do a similar exercise using Python's winsound library for Windows or Sonic Pi.

- Ask pupils to design, plan and code scripted animations in Scratch, perhaps using a timeline or storyboard to work out their algorithm before converting this into instructions for sprites in Scratch. Animations could be based on historical events, scenes from a reading book or dialogue in a foreign language pupils are studying. Scratch has support for recording and playing back audio. Pupils might enter their animation for the UK Schools Computer Animation Competition.

- Pupils could take the chat bot idea above and develop this further, either in Python as shown or Scratch, or perhaps both languages side by side.

Further resources


Selection

Selection is the programming structure through which a computer executes one or other set of instructions according to whether a particular condition is met or not. This ability to do different things depending on what happens in the computer as the program is run or out in the real world lies at the heart of what makes programming such a powerful tool.

Selection is an important part of creating a game in Kodu. An object’s behaviour in a game is determined by a set of conditions, for example: WHEN the left arrow is pressed, the object will move left. Similarly, interaction with other objects, variables and environments in Kodu are programmed as a set of WHEN … DO … conditions. For example, WHEN I bump the apple DO eat it AND add 2 points to score. Scratch can also be programmed in this event-driven way (Figure 2.12):

```
answer = input("Type a word ")
if len(answer)>5:
    print("That's a long word!")
```

**Figure 2.13**

Many apps and other programs include this sort of event-driven programming for implementing the user interface: tapping this button or clicking that icon causes the program to respond in a particular way, perhaps changing the stored data and the user’s view of it.

In Scratch, Python and other languages you can build selection into a sequence of instructions, allowing the computer to run different instructions depending on whether a condition is met. For example, this program tests whether a word has more than five letters (Figure 2.13):

```
answer = input("Type a word ")
if len(answer)>5:
    print("That's a long word!")
```

**Figure 2.13**

Word length tests in Scratch and Python

Notice that the thing which determines whether ‘That’s a long word!’ gets displayed is a test (a ‘condition’) which is either true or false in the Boolean sense. If it’s true then the next bit of code (here say… or print…) gets executed, otherwise it doesn’t.

We can use more complex Boolean conditions, for example the somewhat contrived (Figure 2.14):

```
answer = int(input("Give me a number! "))
if (answer % 3 == 0) and (not (answer < 1000)):
    print("That's a multiple of three that's not less than a thousand!")
```

**Figure 2.14**

Boolean selection in Scratch and Python

Selection statements such as these are at the core of most game programs too, for example (Figure 2.15):

```
If touching Ghost 
then
    say Game Over!
for 2 secs
stop all
```

**Figure 2.15**
It’s worth noting that selection statements can be nested inside one another, allowing more complex sets of conditions to be used to determine what happens in a program. Look at the way some if blocks here are inside others in the following script to model a clock in Scratch, which also uses repetition and three variables for the seconds, minutes and hours of the time (see Figure 2.16):

```python
from time import sleep
hours=7
minutes=50
seconds=48
while True:
    sleep(1)
    seconds=seconds+1
    if seconds==60:
        seconds=0
        minutes=minutes+1
    if minutes==60:
        minutes=0
        hours=hours+1
    if hours==24:
        hours=0
    print(hours,minutes,seconds)
```

Simple clock programs in Scratch and Python

Notice that in the Python code here and above we use a double `==` to check for equality; a single `=` is used to assign a value to a variable.

Selection statements in programming languages typically also include the ability to say what should happen if the condition is false. The usual structure for this is:

```python
if <some condition> then:
    <do something>
else:
    <do a different thing>
```

At the core of many educational games are selection statements like this: if the answer is right then give a reward, else say the answer is wrong (see Figure 2.17).

```python
answer=input(“what is 7x8? ”)
if answer==”56”:
    print(“Well done!”)
else:
    print(“Think again!”)
```

Tables question in Scratch and Python

See also the Scratch and Python programs for the times tables game on page 71.

Some programming languages, including Python, allow multiple conditions to be combined into a single selection statement, with only the code for the first condition that’s true being executed:

```python
answer = int(input(“What was your mark? ”))
if answer >= 70:
    print(“You get an A”)
elif answer >= 60:
    print(“You get a B”)
elif answer >= 50:
    print(“You get a C”)
else:
    print(“You fail!”)
```

Grading program in Python
Other languages, such as the functional programming language Haskell, implement something similar through pattern matching:

```haskell
grade :: Integer -> String
grade mark
| mark >= 70 = "A"
| mark >= 60 = "B"
| mark >= 50 = "C"
| otherwise = "fail"
```

Grading function in Haskell

### Classroom activity ideas

- Encourage pupils to explore the different conditions which the character in Kodu can respond to in its event-driven programming. Get pupils to think creatively about how they might use these when developing a game of their own. Give them time to design their game, thinking carefully about the algorithm, that is, the rules, they are using.

- Ask pupils to design simple question and answer games in Scratch. Encourage them to first think about the overall algorithm for their game before coding this and then working to develop the user interface, making this more engaging than just a cat asking lots of questions. It’s helpful if pupils have a target audience in mind for software like this.

- Selection can also be used to design computer simulations for real-world systems. Pupils could use Scratch to model the outbreak of a disease, using colours to represent whether a sprite is infected or not, and then nested selection statements to determine if a sprite becomes infected when it touches another carrying the disease. The simulation can be made more sophisticated by adding in further selection criteria, such as natural immunity or whether a sprite has been vaccinated. This approach to simulating complex systems is called agent-based modelling. A more sophisticated approach could use Python’s Mesa library [23] or Star Logo TNG [24].

### Further resources


### Repetition

Repetition in programming means to repeat the execution of certain instructions. This can make a long sequence of instructions much shorter, and typically easier to understand.

Using repetition in programming usually involves spotting that some of the instructions you want the computer to follow are the same, or very similar, and therefore draws on the computational thinking process of pattern recognition/generalisation. You will sometimes hear the repeating block of code referred to as a loop, that is, the computer keeps looping through the commands one at a time as they are executed (carried out).

Think about a simple turtle graphics program for a square (Figure 2.18):

---

23 [https://pypi.python.org/pypi/Mesa/](https://pypi.python.org/pypi/Mesa/)
from turtle import *
for i in range(4):
    forward(100)
    right(90)

Squares in Scratch and Python

Using repetition reduces the amount of typing and makes the program reflect the underlying algorithm more clearly.

Notice that in Python we use a variable to keep track of how many times we’ve been round the loop. The Python function `range(4)` is shorthand for the list of numbers 0, 1, 2, 3. The iterator variable `i` takes each of these values in turn. Thus the program:

```python
for i in range(12):
    print(7*i)
```

Prints 0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77 on screen.

To do something similar in Scratch, we’d need to keep track of this ourselves (Figure 2.20):

```
set i \text{ to } 0
repeat 12
say 7 \times i \text{ for 2 secs}
change i \text{ by 1}
```

Figure 2.20

In Snap! this is easier as we have a `for` loop available in the standard tools library (Figure 2.21):

```
for i = 0 to 11
say 7 \times i \text{ for 2 secs}
```

Figure 2.21
In Python and Snap! the list for the iteration doesn’t need to be a sequence of numbers: any list will do. For example (Figure 2.22):

```python
for day in ["Monday","Tuesday","Wednesday"]:
    print day
```

Figure 2.22 Snap! Using standard tools

In the examples above, the repeated code is run a fixed number of times, which is the best way to introduce the idea. You can also repeat code forever (Figure 2.23):

```python
from turtle import *
from random import randint

while True:
    forward(10)
    right(randint(0,3) * 90)
```

Figure 2.23

Random walks in Scratch and Python

Notice that the Python code here is a particular version of a while loop (see below), where the condition is always true, so the code inside the loop runs forever.

This can be useful in real-world systems, such as a control program for a digital thermostat, which would continually check the temperature of a room, sending a signal to turn the heating on when this dropped below a certain value. This is a common technique in game programming. For example, the following Scratch code (Figure 2.24) would make a sprite continually chase another around the screen:

```scratch
pen down
forever
  move 10 steps
  turn (pick random 0 to 3) * 90 degrees
```

Figure 2.24

In event-driven applications, such as a game programmed in Kodu, you can think of all the different event conditions as sitting inside one big ‘repeat forever’ loop. It’s easy to program the same idea in Scratch, as in the following example (Figure 2.25) which uses the W, A, S and D keys to move a sprite around the screen.

```scratch
forever
  if key w pressed? then
    change y by 10
  if key s pressed? then
    change y by -10
  if key d pressed? then
    change x by 10
  if key a pressed? then
    change x by -10
```

Figure 2.25

Much the same thing happens in PyGame Zero’s main game loop in Python:

```python
while game_has_not_ended():
    process_input()
    update()
    draw()
```

You can nest one repeating block of code inside another. The ‘crystal flower’ programs in Logo use this idea. For example (Figure 2.26):

Repetition can be combined with selection, so that a repeating block of code is repeated as many times as necessary until a certain condition is true or while a certain condition is true. There's a subtle but important distinction here. The `<code>` in a repeat until `<condition>` `<code>` loop is executed when the `<condition>` is false, but the code in the while `<condition>` `<code>` loop is executed when the `<condition>` is true.

Compare (Figure 2.27):

```python
sentence = input('Give me a sentence: '
firstword = ''
for letter in sentence:
    if letter== ' ':
        break
    firstword = firstword + letter
print ('The first word was ' + firstword)
```

Finding the first word of a sentence in Python using break

The continue on the other hand skips the rest of the code inside the loop but then goes back to beginning of the repeating loop with the iterator moved on one point:

```python
sentence = input('Give me a sentence: '
nospaces = ''
for letter in sentence:
    if letter== ' ':
        continue
    nospaces = nospaces + letter
print ('Without spaces, you get ' + nospaces)
```

Stripping the spaces from a sentence in Python using continue

26 Note that this only works correctly the first time it is run. Can you work out why?
Classroom activity ideas

- Ask pupils to use simple repetition commands to produce a ‘fish tank’ animation in Scratch, with a number of different sprites each running their own set of repeating motion instructions. This can be made more complex by including some selection commands to change the behaviour of sprites as they touch one another.

- Encourage pupils to experiment with ‘crystal flower’ programs in Scratch, Logo, Python or other languages that support turtle graphics, and investigate the effect of changing the number of times a loop repeats as well as the parameters for the commands inside the loop. What combination of numbers produce complete, symmetrical ‘flowers’? What numbers produce particularly aesthetically pleasing images? There are some great opportunities to link computing with spiritual, social and cultural education here, noting that traditional Islamic art uses repeated geometric patterns.

- Simple game programming, in Scratch, Kodu or PyGame Zero, will often use a combination of repetition and selection. Pupils could program a simple, one-player squash game by writing scripts for the ball which made this move repeatedly around the court until hitting the racquet or the back wall. The racquet script could use the event-driven loop above, but restrict movement to just up and down. Scratch has a built in *(if on edge bounce)* command, so the trickiest thing here is determining what should happen to the ball when it hits the bat. Once pupils have a game like this working, they could adapt it to make a two-player version like the classic Pong video game.

Further resources


Modularity

The above ideas of sequence, selection and repetition are covered at Key Stage 2 in the national curriculum, but remain conceptually and practically important as pupils continue programming at Key Stage 3 and beyond. At Key Stage 3, pupils should be introduced to modularity in their programming, making use of ideas such as procedures and functions to bring computational thinking concepts such as decomposition and abstraction into their coding as well as their planning.

Procedures, functions (and other modular ideas such as classes) allow programs to be written with a far clearer structure, better reflecting the decomposition and abstraction that went into their design: just as we use decomposition to break a problem down into smaller problems, so modularity allows us to build programs up out of smaller parts. Similarly, as abstraction allows us to set to one side details, so modularity means that we can hide the details of specific implementation within procedure, function or class definitions.

Modularity allows for better generalisation too: often someone else might have written a function that solves part of a problem – typically we can simply call that function, perhaps part of a standard library, from our program, without generally concerning ourselves with how they implemented this. Useful as it is to know algorithms for search and sort, most software developers will just take those as given. Most of the time, sorting a list in Python involves simply calling the built-in function sorted, not writing your own code to implement bubble sort, quicksort or one of the other algorithms.
sorted ([31, 41, 59, 26, 53, 58, 97])

To check if a number is prime or not, you can write your own function, or you can just use the isprime function from the number theory module in SymPy:

```python
from sympy.ntheory import isprime
print(isprime(1301))
```

Modularity also makes it easy for a programmer to reuse her own code between different projects, as well constructed functions and classes can be moved between programs quite easily.

Modularity is important for collaborative software development, as it makes it easy to share out the work of writing software across a team, with individuals (or pairs) taking responsibility for implementing the detail of particular elements according to agreed specifications.

Modularity helps with testing and debugging too, as each function, procedure or class can be tested independently of the others, making sure it does exactly what it’s supposed to do. For this to work, it’s important that the modular code doesn’t introduce unpredictable side-effects affecting how other procedures or the main program itself operates.

Modularity also makes it easy to maintain a program, gradually or dramatically, improving efficiency. For example, a function which returns the highest common factor of a couple of numbers can be replaced by a more efficient one – the outside program calling this function will work, irrespective of which version of the function is used but one version is (very much) faster than the other.

```python
def hcf(a,b):
    bestsofar=1
    test=1
    while test <= a:
        if (a % test == 0) and (b % test == 0):
            bestsofar = test
        test = test + 1
    return bestsofar
```

Deficient algorithm to find highest common factors in Snap! and Python

```python
def hcf(a,b):
    if a == 0:
        return b
    else:
        return hcf (b % a, a)
```

Euclid’s (recursive) algorithm to find highest common factor in Snap! and Python

**Procedures**

Procedures are a simple way of using modularity. We group together code with a particular purpose and give it a name. Then, rather than having to type the code each time we want to use it, we simply call the name we’ve given it.

To create a procedure in Scratch we use the Make a Block button; in Python, we write code to define the procedure:

```python
def procedure:
    <procedure code goes here>
```

27 A Python library for symbolic mathematics, [www.sympy.org/](http://www.sympy.org/)
Take for example a procedure to draw a square (Figure 2.30):

![Image of Scratch block for square](image1)

```python
from turtle import *

def square():
    for i in range(4):
        forward(50)
        right(90)

Procedure for a turtle graphics square in Scratch and Python
```

Then code to draw a pattern of squares becomes easier to write, and to understand (Figure 2.31):

![Image of Scratch block for square pattern](image2)

```python
setx(-220)
clear()
for i in range(5):
    pendown()
    square()
    penup()
    forward(100)

Scratch and Python code for drawing five squares (Figure 2.32) using the above procedure
```

**Parameters**

You can pass values (numbers or other data such as strings and lists) to procedures, using generalisation to make procedures much more flexible. We call these values parameters. For example, we can generalise our square procedure to make a procedure which draws a regular polygon with sides of any given length (Figure 2.33):

![Image of Scratch block for regular polygon](image3)

```python
def polygon(sides, length):
    for i in range(sides):
        forward(length)
        right(360./sides)

Procedure for a turtle graphics regular polygon in Scratch and Python
```

We can then use this procedure to make more complex patterns (Figure 2.34):

![Image of Scratch block for complex pattern](image4)
Programming

clear()
edge = 20
for i in range(5):
    polygon(6, edge)
    edge = edge + 20

Scratch and Python code to draw a nest of hexagons (Figure 2.35) using the above procedure

Figure 2.35

Functions

Simply, functions are procedures that return values to the code that called them, which might include another function. Whereas procedures help with structuring what a program does, functions come into their own for decomposing and abstracting the computation that the program performs.

Thus to print the average (mean) of a list of numbers with a procedure, we might do:

def mean(list):
    total = 0.
    for item in list:
        total = total + item
    average = total / len(list)
    print('the mean is ' + str(average))

mean([3, 4, 5, 9])

Whereas a function would simply work out the average and then let the outside program decide what to do with that, which could be printing, or it could be updating a record or using it in another calculation or whatever, which provides much more flexibility.

def mean(list):
    total = 0.
    for item in list:
        total = total + item
    return total / len(list)

The inability to create functions that return values is one of the limitations of Scratch but Snap! allows this, which makes it useful for a much broader range of computation: thus, for example we can write a function which takes a decimal number and returns the binary equivalent (Figure 2.36):

Figure 2.36

def dec2bin(decimal=0):
    if decimal == 0:
        return '0'
    binary = ''
    while decimal > 0:
        binary = str(decimal % 2) + binary
        decimal = decimal // 2
    return (binary)

Snap! and Python functions for converting decimal numbers to a string of bits as their binary representation. Note that Python has a built-in function for this: `bin()`

Snap! (and Python) also allow functions to be passed as arguments to other, higher order, functions. For example, converting a list of numbers into binary can be done using the higher order map function, which takes a function and applies it to every element of a list, returning a list as a result (Figure 2.37):

Figure 2.37

list(map(dec2bin,[65,67,83]))

Mapping the decimal to binary conversion function above over a list of three numbers
Functions become important in computer science and software engineering later on, partly because it's easier to reason logically and mathematically about what a function does and also because functions, at least in strict, functional programming languages such as Haskell, cannot have side effects, which is important when safety and security are of primary importance.

**Recursion**

Procedures and functions can refer back to themselves.

Fractals are geometrical figures where each part of the figure is a smaller version of the whole. We can draw these in Scratch or Python by defining a procedure which calls itself (Figure 2.38).

```
from turtle import *

def tree(size):
    if size > 1:
        forward(size)
        left(20)
        tree(size*0.5)
        right(35)
        tree(size*0.7)
        left(15)
        forward(-size)
    else:
        return
```

Figure 2.38

**Figure 2.39**

We can define functions recursively, for example to work out the factorial of a number (that is, the product of all the integers up to and including it, represented mathematically as !, that is, $4! = 1 \times 2 \times 3 \times 4$) we could code (Figure 2.40):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Figure 2.40

Functions (or procedures) that call themselves allow the sort of recursive decomposition of a problem that divide and conquer algorithms are built on – you could easily implement binary search this way.

In both of the examples above, note that there’s an exit condition inside the recursive function or procedure, otherwise the code could keep calling itself forever.
Whilst recursion is quite a subtle idea, and not one which all pupils are likely to 'get' immediately, once pupils do understand it and can apply it, it becomes a very powerful way of thinking about problems and systems, and can offer a far more elegant way of expressing computational solutions, both as algorithms and code, than iterating over lists or around loops. Many teachers might choose to leave these ideas to Key Stage 4 or 5.

Class

Another example of modularity is the idea of the class. One way of thinking about a class is as setting up a new data structure to store the state or properties of a particular category of things, together with functions or methods that describe the behaviours of things in that category, including how their states might change. A member of the class is called an object, and this approach to programming is described as 'object oriented'.

We might think of a class of cars, and a particular car as an object in that class. The properties of any particular car might include things like its position, its speed, its direction, its engine capacity, its fuel level, its fuel consumption and so on. We could define methods which operate on all objects in the class, such as accelerate, turn left or put petrol in, which would change some of the properties of any object which they applied to. We're using abstraction here as we define the class of objects, its properties and methods, to implement those that are relevant for our problem, but also to hide within the definition of the class the details of the implementation.

Whilst most teachers are unlikely to want to teach about classes and objects in Key Stage 3, the combination of abstract data structure and associated methods can be useful. For example, we could implement fractions arithmetic by defining a new class in Python and overloading the standard arithmetic operators so that they become methods to operate on objects in this class. The properties of objects in the class are simply the numerator and denominator (in their simplest terms); methods could include creating a new fraction, printing the value of a fraction, adding two fractions together, finding the difference between two fractions and so on.

def hcf(a,b):
    if a == 0:
        return b
    else:
        return hcf (b % a, a)

class fraction:
    def __init__(self,top,bottom):
        gcd=hcf(top,bottom)
        self.numerator=top//gcd
        self.denominator=bottom//gcd

    def __str__(self):
        return str(self.numerator)+"/"+str(self.denominator)

    def __add__(self,other):
        newnum=self.numerator * other.denominator + \ 
        other.numerator * self.denominator
        newden=self.denominator * other.denominator
        return fraction(newnum,newden)

    def __sub__(self, other):
        return(self + fraction(-other.numerator,other.denominator))

This implementation of a fraction class in Python always stores each in its lowest terms, using the helper hcf (highest common factor) function to simplify by dividing by the highest common factor of numerator and denominator. Note also how subtraction is defined using the addition method.

Notice that the person, perhaps another programmer, using our fraction class can type things like print(fraction(2,3)+fraction(1,8)) and get back the correct response without needing to know how this calculation is performed.

Activities

Turtle graphics are a great way to introduce pupils to some of the ideas of procedures. Get pupils to create custom blocks to draw simple shapes (rectangles, squares, trapeziums) and then use these to create a drawing of a house. Get pupils to generalise their procedure for drawing squares so that it draws regular polygons of any size, and explore the repeating patterns which they can use with this as a building block.

Cryptography is a great way to introduce pupils to some of the ideas of functions, creating functions which take plain text and a key and encrypt this, or take cipher text and a key and then decrypt this. Pupils could extend these ideas further by writing a function which takes a message and converts it to Morse code or vice versa. Pupils could write
a function to do frequency analysis on a piece of text, counting how many times each letter of the alphabet occurs: one approach would be to write a function to count how many times a particular character comes up and then use map to apply this to the list of all 26 letters, producing a list as a result.

Pupils could extend the idea of the fractions arithmetic class on to include multiplication, division and comparisons, or even to mixed number arithmetic.

Further resources


Data Structures

Alongside the programming control structures of sequence, selection repetition and modularity, implementing any algorithm or computational abstraction involves deciding how the computer is going to manage the information to be processed – how the data on which the program draws are to be stored and organised.

Not all data structures are available in all programming languages, although often more complex data structures can be built up from simpler ones. In object-oriented languages, classes can be created to implement specific data structures out of more primitive ones, as in the example of fractions earlier where we implement a simple fraction data type using Python’s tuples (essentially, an ordered list of two elements).

Variables

Pupils are introduced to variables at primary school: a variable is a simple data structure. It is a way of storing one piece of information somewhere in the computer’s memory whilst the program is running and getting that information back later. There’s a degree of abstraction involved here – the detail of how the programming language, operating system and hardware manages storing and retrieving data from the memory chips inside the computer isn’t important to us as programmers, just as these
details aren’t important when we’re using the clipboard for copying and pasting text. One way of thinking of variables is as labelled shoeboxes, with the difference that the contents don’t get removed when they are used.

The concept of a variable is one that many pupils struggle with and it’s worth showing them lots of examples to ensure they grasp this. A classic example which pupils are likely to be familiar with, particularly from computer games, is that of score.

You can use variables to store data input by the person using your program and then refer to this data later on.

```python
name = input ('Hello, what is your name?')
print ('Hello, ' + name)
print (name + ' is a very nice name.')
```

**Storing user input in a variable and referring to it in Scratch and Python**

Here (Figure 2.41), name is a variable, in which we store whatever the user types in, and then use it a couple of times in Scratch’s or Python’s response. In the case of Scratch answer is a special temporary variable used to store for the time being whatever the user types in. Notice that variables can store text as well as numbers. Other types of data can be stored in variables too, depending on the particular programming language you are working in.

Variables can also be created by the program, perhaps to store a constant value so that we can refer to it by name (Pi below), or the result of a computation (Circumference in the code below) or random numbers generated by the computer (for example Radius below) (Figure 2.42):

```python
pi = 3.14
radius = randint (1,10)
circumference = 2 * pi * radius
print ('If a circle has a radius of ' + str(radius) + 'cm.
print ('Its circumference would be ' + str(circumference) + 'cm.')
```

**Random circumference calculator in Scratch and Python**

The idea that the contents of the ‘box’ are still there after the variable is used is sometimes a confusing one for those learning to program. Have a look at the following code and decide what will be displayed on the screen (Figure 2.43):

```python
a = 10
b = 20
a = b
print ('a is ' + str(a))
print ('b is ' + str(b))
```

**Variable assignment in Scratch and Python**

You should see ‘a is 20’ followed by ‘b is 20’. Try it! Was it easier to understand in Scratch than Python? (Dehndad and Bornat, 2006; but qv Bornat, 2014).

In Kodu and other game programming, variables are useful for keeping track of rewards, such as a score, and for introducing some sort of limit, such as a time limit or health points that reduce each time you’re hit. Kodu’s event-driven approach allows particular actions to be done when variables reach a predetermined level.
One particularly useful example of variables in programming is as an iterator – this is a way of keeping track of how many times you’ve been round a repeating loop and of doing something different each time you do. To do this in Scratch we initialise a counter to zero or one at the beginning of the loop and then add one to it each time we go round the loop. In Python, we can iterate across values in a list. For example, the following program displays the eight times table (Figure 2.44):

```python
for i in range (1,13):
    print (str(i) + ' x 8 = ' + str(i*8))
```

*Code for the eight times table in Scratch and Python. Note that range (1,13) means the integers from 1 up to but not including 13*

As we have seen, you can also use an iterator like this to work with strings (words and sentences) one letter at a time or through lists of data one item at a time. Take care with the beginning and end, as it’s all too easy to start or end too soon or too late with iterators.

### Further resources

- BBC Bitesize (n.d.) How do computer programs use variables? Available from [www.bbc.co.uk/guides/zw3dwnn](http://www.bbc.co.uk/guides/zw3dwnn)
- Berry, M. (2014) *How to program a Scratch 2.0 times table test*. YouTube. Available from [www.youtube.com/watch?v=YHGyPfGgIx8](http://www.youtube.com/watch?v=YHGyPfGgIx8)

### Lists

A list is an ordered collection of data, each element of which is of the same type (normally), and where we can reference each by its position in the list. Remember that this is simply an abstraction that allows us to store and retrieve data in the computer’s memory, but often it’s a very helpful abstraction when we are dealing with lots of related data, such as marks for pupils in a test, words in a sentence, scores in a game, notes in a tune, locations on a route, etc. Just as we might think of variables as a special sort of shoebox in which a single piece of information can be kept, so we could think of a list as a deck of cards, on each of which a piece of information can be recorded.

In order to work with variables, we need only a few basic operations – creating the variable, retrieving information from the variable and storing new information to it. For lists, things are more complex. Scratch, which has a relatively basic implementation of lists as a data structure, allows the following operations (Figure 2.45):

### Classroom activity ideas

- Get pupils to create a mystery function machine in Scratch or Python, which accepts an input, stores this in a variable and then uses mathematical operators to produce an output shown on screen. Setting the display to full screen in Scratch, or running at the command line in Python, pupils can challenge one another (and you) to work out what the program does by trying different inputs.
- Pupils can use variables in their games programs, in say Scratch or Kodu, using a score to reward the player for achieving particular objectives (such as collecting apples) and imposing a time limit.
These allow data to be added to the end of the list, items from the list to be deleted, items to be inserted at any position in the list, shifting the following items further on, replacing an item with something else. We can retrieve the data stored at any item in the list, find out how long the list is and check whether a list contains a particular value or not.

The equivalent commands in Python are as follows. Note that Python numbers list elements from zero.

```python
list.append('thing')
list.pop(0)
list.insert(0,'thing')
list[0]='thing'
list[0]
len(list)
'thing' in list
```

Both Snap! and Python extend the range of commands that can be used to operate on a list much further: for example Python includes a function to sort a list into order (sorted). Both Snap! and Python allow lists to be made up of other lists, which is one way of providing a multi-dimensional data structure if that's needed.

To illustrate lists, let's take the example of a shopping list:

- We start with an empty list: `[]`.
- We add milk: `['milk']`.
- We add bread: `['milk', 'bread']`.
- We add butter: `['milk', 'bread', 'butter']`.
- We add eggs: `['milk', 'bread', 'butter', 'eggs']`.
- We can sort the list into alphabetical order: `['bread', 'butter', 'eggs', 'milk']`.
- We can check if we need to buy quinoa: no, not on our list, this time.
- We buy some eggs, removing that from our list: `['bread','butter','milk']`.
- We remember that we should buy low fat spread rather than butter: `['bread', 'low fat spread', 'milk']`.
- We count up how many things we need to buy: three.

**Implementing the above operations on a shopping list in Scratch (see Figure 2.46) and Python.** Note that Python allows us to reference elements of the list by their value, whereas Scratch only references by position.

Python provides a useful mechanism for ‘slicing’ a list to extract particular elements or lists of elements. For example, with a list of the first 10 prime numbers `primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]`, we can get the element `primes[0]`, the last element `primes[-1]`, the first three elements `primes[:3]`, the last three elements `primes[-3:]` or the fourth, fifth and sixth primes `primes[3:6]`. Primes[a:b] is from the a+1 the position (remembering we start at zero) up to but not including the b-th position. Negative numbers count from the last item back. It’s worth practicing with this, as it’s simpler than it sounds.
Python also provides ‘list comprehensions’, which allows new lists with particular properties to be created. For example, we can create a list of the first 10 square numbers with the code:

```
squares = [n * n for n in range(1,11)]
```

We can then filter this list to produce a list of just the even square numbers from the first 10:

```
evensquares = [s for s in squares if s % 2 == 0 ]
```

Returning to the deck of cards analogy for lists, we might plan how to implement a random shuffle on a list:

One algorithm would be to swap the last card with one of the cards up to and including it, chosen at random. Then swap the last but one card with one of the ones up to and including it, then the last but two with one of the ones up to and including it, and so on until we get back to the first card in the pack (Fisher and Yates, 1948 [1938]: 26–27; qv Knuth, 1969).

In Scratch we would code (Figure 2.47):

```
from random import randint
for currentplace in range(len(pack)-1,-1,-1):
    swapwith = randint(0,currentplace)
    pack[currentplace],pack[swapwith]=pack[swapwith],pack[currentplace]
```

After shuffling our cards, we might then want to sort them. Combining lists with recursive functions allows us to implement divide and conquer algorithms, such as quicksort, quite elegantly. We can implement quicksort as a function which calls itself on shorter and shorter lists above and below the pivot for the previous list until only an empty list is left, which is trivially already sorted (Figure 2.48).

```
def quicksort(list):
    if len(list)==0:
        return list
    else:
        head = list[0]
        tail = list[1:]
        lower = [x for x in tail if x <  head]
        upper = [x for x in tail if x >= head]
        return quicksort(lower) + [head] + quicksort(upper)
```

Classroom activity ideas

- Pupils could create lists of notes, perhaps with a paired list of durations to play in Scratch or Sonic Pi.
- Pupils could explore implementing the ‘perfect’ riffle shuffle on a deck of cards using list manipulations, splitting the pack in two equal halves and taking cards alternately from top and bottom halves (Diaconis et al., 1983).
- Pupils could implement one or more of the sort algorithms discussed on pages 14 - 17 using list manipulations.
- Can pupils write programs to compute descriptive statistics for lists of numerical values (for example find the mean, median and mode, the minimum or maximum, the total, the standard deviation)?
- Can pupils write a program which would generate valid national lottery results (results are given as ordered lists, noting that no number may occur more than once)?
Further resources


Computing at School (n.d.) Fun with lists by Mark Tranter, looking at how Python’s built in list functions could be implemented. Available from http://community.computingatschool.org.uk/resources/2683 (free registration)


Other data structures

Other data structures are available.

Whilst Scratch does a good job of hiding this from the programmer, variables themselves are more complex than they might appear, as a variable might store data of one of a number of different data types: perhaps a number, but also possibly a Boolean value (true or false) or text, each of which might be represented quite differently inside the computer and for each of which only certain operations would make sense.

Some programming languages are much more demanding (‘strongly typed’) in their treatment of data types, demanding that these be declared explicitly before the variable is ever used. Python is relatively easy going about data types, but even in Python it is sometimes necessary explicitly to change (or ‘cast’) a variable from one type to another. In some of the examples above we use ‘str(x)’ to convert a number x into a string so that we can join it to other strings or ‘int(y)’ to take a string of user input and convert it into a number so we can compare it with other numbers.

Strings of text are quite different from numbers, and can be thought of as simply lists of letters or other characters. Thus, some of the operations we might perform on a list also make sense when working with strings – it makes sense to ask how long a string is, to be able to reference particular characters directly, to be able to replace one character with another (such as converting a string between different cases) or to be able to join two strings together (concatenation).

Scratch provides a few blocks for working with strings (Figure 2.49):

![Figure 2.49](image)

The equivalent commands in Python are:

'hello ' + 'world'

'world'[0]

len('world')

Python provides the same slicing tools for strings as it does for lists, thus if:

```python
cas='Computing At School'
```

then:

- cas[0] is C
- cas[-1] is l
- cas[9] is Computing
- cas[-6:] is School
- cas[10:12] is At

But as with lists, Python’s string handling capabilities extend far beyond this.\(^{(28)}\)

A variable can be thought of as a single number and a list as an ordered, one-dimensional set of

\(^{(28)}\) See https://docs.python.org/3.5/tutorial/introduction.html#strings and https://docs.python.org/3.5/library/stdtypes.html#string-methods. Python’s Natural Language Toolkit provides powerful libraries for working with larger bodies of text: www.nltk.org/
numbers. We could also have two (or higher) dimensional collections of data. These are called **arrays**. Python doesn’t support arrays as standard (although it offers good support through the NumPy library[^29]) nor does Scratch. It is possible in Python and Snap! to construct higher dimensional collections of data using nested lists, but this is unlikely to be particularly appealing or accessible to pupils at Key Stage 3.

Rather than working in Python or Snap! for two-dimensional arrays of data, revisiting Excel may be much more useful and accessible, given the direct and immediate view of all the contents of the array. Excel (and other spreadsheet software such as Google Sheets) can be used for genuinely two-dimensional data, such as heights, temperatures or rainfall for locations on a grid, the presence or absence of a cell in a Life simulation or greyscale values for a monochrome pixel bitmap, allowing calculations to be done with and to these data.

Many pupils might play, or have played Minecraft, in which the world is represented as a 3D array of data about the block at each location. 2D strategy or construction games (such as Sim City or Civilisation) represent the world as a 2D array. Pupils can develop an understanding of arrays, and practice their Python programming skills using a Python API for Minecraft, which is provided as standard for the Raspberry Pi and also available, with a little ingenuity, on other platforms. In the Raspberry Pi version of Minecraft we can create a floating cube of 1000 stone blocks using this code:

```python
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
stone = 1
x, y, z = mc.player.getPos()
mc.setBlocks(x+1, y+1, z+1, x+11, y+11, z+11, stone)
```

Notice the three parameters necessary to specify locations in this 3D virtual world (see Figure 2.50).

![Figure 2.50](anonymised example of teacher’s mark book, with thanks to Firefly Learning)

While a **table** in a spreadsheet could be a 2D array of data, more often though we might think of a table of values in a spreadsheet or a database as providing a structured collection of data about a number of different things – each row of the table becomes a record of an individual case, each column the different fields of those records. For example, a spreadsheet might be used by a teacher to track assessment data for pupils in her class, with each row storing data on an individual pupil, each column recording attainment on particular tasks or tests, as well as additional personal information such as name, roll number and date of birth. The

[^29]: www.numpy.org/
spreadsheet is really then being used as a single table database, and our focus moves somewhat from performing computation to managing and processing this structured information.

Further tables, typically in a **database** now rather than a spreadsheet, could link to these data, perhaps providing further personal details or information about the objectives of each assessment. Relating tables of data in this way means that we only need to store information once, in one place, but can use it in many different ways – typically we would use other software to manage this database (a ‘relational database management system’ [RDBMS], such as SQLite, MySQL or Microsoft Access).

Pupils learn to ‘create, organise, store, manipulate and retrieve digital content’ when in Key Stage 1 and to collect, analyse, evaluate and present data and information in Key Stage 2. The Key Stage 3 curriculum expects pupils to make use of ‘appropriate data structures’ in their programming and to collect and analyse data. Pupils certainly don’t need to create database management programs themselves in order to do this, but can write programs using APIs to work with data stored in a standard database.

Whilst not supporting arrays or databases without additional libraries, Python does include another very useful data structure, the **dictionary**. Like a normal dictionary, this makes it easy to look up one value (the key) and get back a bit of associated information: the value. Unlike normal dictionaries, Python dictionaries aren’t in any particular order – they are unordered collections of key:value pairs, where the value is stored or retrieved using the associated key.

For example, the command:

```python
languages=dict([('Alex', 'Python'), ('Bobbie', 'Snap!'), ('Chris', 'Python'), ('Drew', 'Scratch'), ('Elliott', 'Visual Basic')])
```

creates a new dict, ‘languages’ in which we might store the preferred language of each of five students. Notice that no two students have the same name, but a couple of them like the same language – which is fine.

We can type:

```python
print(languages['Alex'])
```

and get back the response ‘Python’, as expected.

We can change an entry too:

```python
languages['Drew']='Snap!'
```

We can remove an entry from the dict:

```python
del languages['Bobbie']
```

And we can add a new entry very simply:

```python
languages['Frankie']='Kodu'
```

**Graphs** were discussed on page 26 as a particularly useful form of computational abstraction. Graphs can be stored and manipulated programmatically as lists of edges, plus perhaps associated ‘weights’ or as an array showing which nodes are connected to which, with the weight of the edge given in the array. Python has a number of libraries for working with graphs, including NetworkX,\(^{31}\) and a variant of Snap!, Edgy provides tools for dealing with graphs using a block-based language.

As mentioned earlier (page 68), object orientation allows programmers to define their own **classes** of abstract data types, in which different properties of objects can be drawn together, as well as the methods which operate on objects in those classes.

### Classroom activity ideas

- You can set pupils many challenges with string handling – can they remove all the spaces or all the vowels from a sentence? Can they reverse the order of letters in a word? Can they make a list of all the words in a sentence and order these alphabetically? Can they count how many times each letter occurs in a piece of text, and so on.
- Can pupils use a dictionary to convert text into Morse code or vice versa? Can they then play the Morse code?
If you have access to Raspberry Pis (or can install Python scripting for Minecraft on another platform) encourage pupils to experiment building things, or changing things, in Minecraft using Python programming. Can they create charts in Minecraft to visualise data? Can they import a low-res photo into Minecraft?

If you have a school weather station, could pupils use their programming to interface with this, adding readings to an external database or analysing or visualising data from a database of weather records.

Further resources


BBC Bitesize (n.d.) Databases. Available from www.bbc.co.uk/education/topics/zwm6fg8


Can we fix the code?

Back in the days when there were very few computers, which took up a whole room and used electro-mechanical relays rather than transistors, there was a story of one machine that just wouldn’t work as it should – careful investigations revealed that a moth, a literal ‘bug’, had become lodged between the blades of a relay switch, stopping the switch from closing and thus the computer from operating.

Errors in algorithms and code are still called ‘bugs’, and the process of finding and fixing these is called ‘debugging’. Debugging can often take much longer than writing the code in the first place, and whilst fixing a program so that it does work can bring a great buzz, staring at code that still won’t work, apparently no matter what you do, can be the cause of great frustration too: this can be tricky to manage in class. It is worth spending time getting the code right in the first place, through careful planning, logical reasoning and a good command of the language rather than having to spend time fixing things later. Not all bugs get spotted, and those that don’t can have profound consequences. (32)

Bugs fall into two main categories – those in the algorithms, which sometimes are called logic bugs, and are often due to not quite understanding the problem properly, and those in the code.

In text-based languages, many of the bugs in the code are ‘syntax’ errors where the formal rules of the language’s vocabulary and grammar haven’t been adhered to, and so the computer won’t be able to turn the code you’ve written into machine code that its CPU can execute; not all software engineers see these as ‘bugs’, merely as relatively easy to fix syntax errors which a good IDE or text editor should prevent from getting made in the first place. Seemingly cryptic error messages about the syntax error are generated, which is at least a starting point for identifying exactly where indentation or
a colon has been missed out or similar. This can be a good teaching opportunity for emphasising the importance of spelling, punctuation and grammar in all pupils' work.

In graphical languages like Kodu and Scratch it's almost impossible to make syntax errors, so as pupils make the transition from graphical- to text-based languages much of their time might be spent on getting the syntax right.

Much more important than these syntax errors are the logical or semantic errors, where the code runs or compiles perfectly, but it doesn't quite do what is intended. These errors are more likely to be about having the wrong algorithm, of not translating the ideas of the algorithm into code quite correctly or sometimes a misunderstanding of the semantics, the meaning, of the commands of the language or even of how the computer itself operates. Because it's normally clear when a program isn't working properly, particularly if we test programs, procedures and functions carefully, it is often easier to address misconceptions like these in computing than in other subjects where feedback is less immediate.

Sometimes bugs in algorithms or code only become apparent in certain circumstances – a program might function perfectly well most of the time and then crash (suddenly stop working) very occasionally. For example, normally a program could find the mean of a list of numbers by adding them up and dividing by how many there are, but if given an empty list it might attempt to divide by zero, which in some programming languages would cause the code to crash. Creating a good, comprehensive set of test data with known outcomes is important for tracking down these sorts of bugs in a systematic way, but even working out from the input what caused the crash is a great chance for pupils to put logical reasoning to work.

From primary school onwards, pupils should be taught to use logical reasoning to detect and correct errors in algorithms and programs, so it's not really enough for pupils to fix their code without being able to give an explanation for what went wrong and how they fixed this. In programming classes, pupils focussed on the task of writing a program for a particular goal might want help from you or others to fix their programs: tempting as this may be, it's worth you and they remembering that the objective in class is not to get a working program but to learn how to program, and their being able to debug their own code is a big part of that.

One way that you can, and should, help is to provide a reasonably robust, general set of debugging strategies which they can use for any programming or indeed more general strategies which they can use when they encounter problems elsewhere.

The Barefoot Computing team suggest a simple set of four questions, emphasising the importance of logical reasoning:

1. Predict what should happen.
2. Find out exactly what actually happens.
3. Work out where something has gone wrong.
4. Fix it.

One way to help predict what should happen is to get pupils to explain their algorithm and code to someone else (or even an inanimate object such as a rubber duck) – in doing so, it's quite likely that they will spot where there's a problem in the way they are thinking about the problem or in the way they've coded the solution.

In finding out exactly what happens, it can be useful to work through the code, line by line. Seymour Papert described this as 'playing turtle': in a turtle graphics program pupils could act out the role of the turtle, walking and turning as they follow the commands in the language themselves, or following the instructions for themselves with a pencil on paper: Away from the easily visualised world of turtle graphics, pupils could maintain a trace table, keeping a record of the values of variables and lists as they step through their code one line at a time. Some IDEs include debuggers, allowing this process to be automated, although the careful thought involved in doing this by hand might still make it easier to spot, and learn from, what's going wrong.

In working out where something has gone wrong, encourage pupils to look back at their algorithms before they look at their code – before they can get started with fixing bugs, they will need to establish whether it was an issue with their thinking or with the way they've implemented that as code. Another technique is to use something like the 'divide and conquer' algorithm for guessing the hidden number to find a bug – can you work out whether the bug is in the first or second half of the code? In the first or second quarter, and so on. Sometimes this is called wolf-fencing – to find the wolf, build a fence...
and listen whether the howl comes from one side or another, repeat with smaller and smaller areas until you find the wolf.

Debugging is a great opportunity for pupils to learn from their mistakes and to get better at programming. Encourage your pupils to adopt a ‘growth mindset’ and to make the most of the opportunities their bugs present for them to learn more about how to program.

Classroom activity ideas

- Pupils are likely to make many authentic errors in their own code, which they will want to fix. You might find that it’s worth spending some time giving pupils some bugs to find and fix in other programs, both as a way to help develop strategies for debugging and to help with assessment of logical reasoning and programming knowledge. Create some programs with deliberate mistakes in, perhaps using a range of logical or semantic errors and set pupils the challenge of finding and fixing these, for example can pupils find all the errors in the following Python program designed to ask 10 different multiplication test questions:

```python
import Random
a = random.randint(1,12)
b = random.randint(1,12)
for i in range(10):
    question = "What is "+str(a)+" x "+str(b)+"? "
    answer = input(question)
    if answer == str(a*b):
        print("Well done!")
    else:
        print("No.")
```

- Encourage pupils to debug one another’s code. One approach is for pupils to work on their own program for the first part of the lesson and then to take over their partner’s project, completing this and then debugging this for their friend.

- A similar paired activity is for pupils to write code with deliberate mistakes, setting a challenge to their partner to find and then fix the errors in the code.

Further resources


BBC Bitesize (n.d.) *What is debugging?* Available from [www.bbc.co.uk/guides/ztkx6sg](http://www.bbc.co.uk/guides/ztkx6sg); *Writing error-free code*. Available from [www.bbc.co.uk/education/guides/zcjfyrd/revision](http://www.bbc.co.uk/education/guides/zcjfyrd/revision)


---

33 Try online at [https://trinket.io/library/trinkets/2da63b4823](https://trinket.io/library/trinkets/2da63b4823)

Programming

References


