A tao of programming

Dan MacLean

2018-09-27

Contents

Chapter 1

What is Programming?

Programming is the task of creating instructions so that a computer can perform a task for you.

At the highest level programming has two steps: design a solution to a problem, then encode the solution so a computer can do the work. The first part is a challenge for creative logic, the second a challenge for language. Programming a computer takes more of our imagination and creativity than it does of our cold, hard logic or mathematics.

This guide is intended to help a beginner reach an understanding of how to design a solution to a problem in such a way that it can later be translated into a programming language. No coding in a programming language is demonstrated in this guide, instead the guide focuses on the building blocks of solutions for programming: sequences, loops, events, conditions, and on the key computational thinking practices: experimentation and iteration, testing, re-use and abstraction. In this guide we use a graphical tool called Scratch to create and apply these concepts in a computer.

The guide aims to teach an understanding of the way that programs are built, before we understand how programs are encoded in a language.

> This is not going to go the way you think.

Be prepared, this is not a typical programming course. It will be a largely self led exploration of concepts and ideas, not a slog through syntax and rote copying. If you've seen a programming guide before, forget it. Here your best tools will be your imagination, a notebook, a friend and your powers of reflection and criticism. This guide will be your research partner in the exploration helping you to reveal the fundamental concepts of programming.

For this course you will need.

1. A computer with an internet connection
2. A notebook
3. A friend (actually this is optional, but will be helpful).

Chapter 2

Reflection

2.1 Reflection and critique as programming tools

The extent to which programming is a slow, difficult and often collaborative effort isn't widely appreciated. Stereotypes of 'hackers' in media abound - and this stereotype suggests that building a program comes from heroic spurts of inspiration and esoteric knowledge. More realistically, building a program is a slow process that involves thinking about a problem, weighing up different potential solutions and expressing them. Reflecting on problems and potential solutions and criticising them (in the strict neutral sense of evaluating and assessing dispassionately) are really important tools for advancing the solution to a problem and its implementation in a program.

The aim of this section is to encourage practice in these skills.

As so much of this guide will involve reflection and criticism, having someone you feel comfortable discussing stuff with will be helpful. It's not absolutely necessary, though. You can do all of this guide by yourself if you'd prefer.

2.2 Scratch - our creativity tool

Scratch is a free graphical computer program for creating media projects. It is available at the MIT Scratch website[1]

With it you can create a wide variety of interactive projects - animations, games etc. Hundreds of thousands of people use Scratch across the world, including primary school children and Harvard computer science undergraduates learning to program. It's designed to be accessible yet complete. It encompasses all the key concepts we'll need to understand programming. Take a look at this introductory video[2].

Let's investigate Scratch!

For you to do

These tasks may seem trivial, hopefully they will seem playful. Have some fun with them.

1. Sign-up for Scratch https://scratch.mit.edu/
2. Browse some starter projects at https://scratch.mit.edu/starter_projects/ online
3. In your notebook, sketch ideas for three different Scratch projects you would like to create.
4. Go to Scratch and make the Scratch cat do something surprising

[1] http://scratch.mit.edu
[2] https://vimeo.com/65583694

huh?

Wait, what?

Yep, item number four does say to go and do something in Scratch.

huh?

But you haven't shown us how to use it? Are we supposed to just go and do it? What are we learning here?

Glad you asked! The object here would be for you to identify the stomach churning desperation that comes from not knowing how to solve something, yet being committed to doing so. And with that burning in your gut you manage not to be paralysed by the darkness of ignorance and still manage to claw your way into the light.

A lot of the time with programming you're not going to know exactly what to do at the outset. This task reaches out to your inquisitiveness and curious spirit. Just give it a go - you can't break anything or go wrong.

Look at this way, if you're sitting there thinking you don't know how to do it, anything you do will be a surprise!

You can do this, I believe in you.

Here's a starter sheet[3] if you would like a *little* hint.

For you to do

When you've built something surprising, reflect on Scratch, perhaps using the points below, jot your responses in your notebook and/or with your friend(s).

1. What did you figure out?
2. What do you want to know more about?

2.2.1 Sharing Scratch Projects with Studios

Scratch Studios are a feature of the Scratch website, basically a sort of gallery to which you can post your creations. Let's use them!

For you to do

1. Find the Scratch Surprise Studio on the website http://scratch.mit.edu/studios/460431. Add your Scratch creation.

Here's a studio sheet[a] for some hints.

[a] worksheets/scratch_studio.pdf

2.3 Criticising projects

As you develop programming skills, being able to criticise and find strengths and weaknesses in your and other projects will help you build stronger, better projects.

Let's think now about criticising each others projects. If you are working with a friend (or group of friends) make sure you can find your projects in the Scratch Surprise Studio. ### An important note on groups of friends

When you choose a group of friends for this work, its going to be best if you can find someone with whom you can exchange ideas frankly. In an academic setting, probably not everyone you could work with will work well as a partner in a critique

[3] worksheets/scratch_surprise.pdf

group. A lot of this will depend on different expectations of roles in people from different social and ethnic backgrounds. Try to work with someone at a similar professional level, it can be hard to tell someone in a more senior role that they're wrong , and what they say can be taken as 'the right answer' - when all contributions are valuable. Be aware too of the influence of a person's background on how they'd express a personal opinion. If you're from a place where it feels more natural to go along with the flow, try to find someone you'd be comfortable to express your ideas with. Experiment with sharing views over written media, rather than face to face.

2.3.1 What is appropriate criticism?

This is a tricky subject, anything about *the project* could be criticised legitamately. But much of people's difficulty with giving and receiving criticism stems from the fact different people take criticism differently. But some rules of thumb can help you come up with constructive comments.

- Keep things about the project, not the person!
- Be convinced that there is a need for the point you wish to make - be sure that you believe it really *will* help
- Find and say a positive before you deliver a negative
- Don't say something because you haven't said something for a while.

2.3.1.1 Delivering (and receiving) criticism well

Some people are totally upset by the slightest perceived negative. These people are awesome to have on a team, they are very valuable as constant re-evaluators: iteration of a project can go much quicker when they're involved. At the other end of the scale some people are impervious to criticism and will take anything you can say and remain upbeat. These people are great at maintaining a positive direction in a team and help stop it from getting mired in details. But such diversity makes it difficult to pitch a point! As you'll have worked out, when delivering a point you need to try to be sensitive about people's views - just because you have a valid point you don't have a right to be rude or insensitive, you still have a responsibility to help maintain the civility of the conversation.

If you know you're the sort of person whom criticism affects strongly, try to accept other people's points with the presumption of good faith from the other side.

If you're the sort of person that criticism just bounces off, try to invest some time in thinking specifically about what you've received, try other people's ideas on for size.

For you to do

1. Fill in the table below for each project by each person in your critique group of friends.

Feedback by	What is something that doesn't work or could be improved?	What is something that is confusing or could be done differently?	What is something that works well or you really like about the project?
??	??	??	??
??	??	??	??

It may be helpful to think about:

- Clarity: Did you understand what the project is supposed to do?
- Features: What features does the project have? Does the project work as expected?
- Appeal: How engaging is the project? Is it interactive, original, sophisticated, funny, or interesting? How did you feel as you interacted with it?

2.4 In this section we...

Round Up

In this section we have endeavoured to practice the reflection skill so necessary for evaluating your designs and creations in software. The iterative development process relies on incremental improvements. Being able to give and receive new ideas (often from yourself) will help massively.

Chapter 3

Sequence

A computer program is a sequence of instructions for a computer to execute. Identifying the proper sequence of events is a vital skill. That's the main aim of this section - breaking things down into sequences. The secondary aim will be to practice iteration, by experimenting with sequences of things we'll initially get things wrong, but by iterating each time we will get closer to a good solution.

3.1 Examinging and building a sequence of instructions

Let's get straight to it and break down a sequence.

For you to do

1. In pairs work out who doesn't mind being bossed and who doesn't mind being bossy[a]
2. Bossed partner: Close your eyes! (or at least look away from the screen)
3. Bossy partner: Watch one of the videos[b]
   - http://vimeo.com/28612347
   - http://vimeo.com/28612585
   - http://vimeo.com/28612800
   - http://vimeo.com/28612970
4. Bossy partner: describe (using spoken words only) how to perform the sequence of moves in the video. Bossed partner: Do only what the bossy partner tells you, are there any points where you need them to be clearer? If they aren't making sense - say so.
5. Write down the steps as you go. Work from the steps.
6. In your design journal:
   - Reflect on what was easy/difficult about being bossy
   - Reflect on what was easy/difficult about being bossed
   - What was difficult about watching?

[a] As far as you possibly can anyway! If you're doing this alone you'll have to do both parts, which may mean developing temporary amnesia. If you can't decide which of you is which, then choose randomly.

[b] If you have more than one pair in the group, cover as many videos as you can.

The process above may have revealed a few points. The most apparent is that in order to be understood it is vital to be explicit about the action. Another is that the receiver of instruction (here the bossed partner) is not (often) able to 'just do what you meant' and get it right. The whole process probably took a few goes at least while you worked it out, so iteration is important. The list of instructions you created was a prototype program for a dance! This form of program, not real code - but a good description of the important parts is called `pseudocode`. The first draft of a real program will often be pseudocode.

3.2 Building a sequence with constraints

Computer languages don't have a command for each possible action. They instead have very restricted sets of key commands that must be chained and used creatively to make larger effects. The commands they do have are often quite limited and do very specific tasks.

This is a deliberate design feature. Flexibility and power comes from these small units, chained together in novel sequences.

In this section we'll examine using small units in different sequences.

For you to do

1. Start a new Scratch project
2. Using *only* the below ten blocks, make something that interests you
   - `go to`
   - `glide`
   - `say`
   - `show`
   - `hide`
   - `set size`
   - `play sound`
   - `wait`
   - `when this sprite clicked`
   - `repeat`
3. Share your creation in the 10 Blocks Studio http://scratch.mit.edu/studios/475480
4. In your critique groups or your design journal, discuss the following:
   - What was difficult about being able to use 10 blocks?
   - What was easy about only being able to use 10 blocks?
   - How did the constraint make you think of things differently?

3.3 Fixing some bugs

Let's examine some broken sequences and try to fix them. In this section the aim will be to explore the central sequence, and practice iterative testing and de-bugging. This is your first real opportunity to try out the major parts of the development cycle.

For you to do

1. Do the five debugging challenges described on this debugging sheet[a]
2. Discuss your testing and debugging practices with a partner. Make note of the similarities and differences in your strategies.

[a] %22worksheets/scratch_debugging.pdf%22

3.4 In this section we...

Round Up

In this section we've looked at the importance of sequence. Breaking a problem down in to small parts is a key step to solving it. Programs are built from a sequence of small units executed in a particular order. Applying and replying combinations of commands, testing them and repeating until the combination works is the main development loop of programming.

Chapter 4

Loops