A PARENT & CHILD CURRICULUM

From Zero to Developer

A one-year, step-by-step coding plan for a 12-year-old who has never written a line of code.

Starting point	Complete beginner
Pace	3–4 hours a week (four 45-minute sessions)
Length	52 weeks, in four stages
Cost	$0 — every tool in this plan is free
Equipment	Any computer with a web browser

Built for you and your son. Print it, stick it on the wall, tick the boxes.

01. Before You Start

Five minutes of reading that will save you both a lot of frustration.

Your son is twelve. That is a wonderful age to start: old enough to hold an idea in his head for an hour, young enough that he still thinks making a computer do something silly is worth an afternoon. The single biggest predictor of whether he is still coding in three years is not talent and not the language he starts with. It is whether the first six months felt like play.

So this plan is built backwards from that. Every week ends with something he can show you on a screen. Nothing is learned "because you'll need it later." He learns loops because his character needs to walk in a circle; he learns variables because his game needs a score. The theory arrives as the answer to a problem he already has.

What "becoming a software developer" actually looks like

Parents often imagine one long ladder. It is really four different skills that grow alongside each other, and this plan grows all four at once rather than finishing one before starting the next.

The first is **problem breakdown** — taking "make a game" and turning it into eleven small things that can each be done in ten minutes. This is the real skill, and it transfers to homework, music, and everything else. The second is **syntax**, the actual typing of a language. This is the easy part, and the part everyone mistakes for the whole job. The third is **debugging**, which is the willingness to sit with something broken and narrow down why, without concluding that you are stupid. The fourth is **finishing** — getting a project from ninety percent to done, which almost nobody teaches and which is what separates people who code from people who used to code.

> **THE ONE THING THAT MATTERS MOST**
>
> He must type the code by hand. Every character. Copying and pasting a working program teaches nothing, because the learning happens in the typo — in noticing that `Print` is not `print`, and finding out why the computer cared. Resist the urge to speed this up. The slowness is the lesson.

The 45-minute session recipe

Four sessions a week, about 45 minutes each. Use the same shape every time. A predictable shape means he spends his energy on the code instead of on wondering what happens next.

TIME	PART	WHAT HAPPENS
5 min	Warm-up	He opens last session's project and re-runs it. He explains to you, out loud, what one line does. Talking about code is how it moves from fingers to head.
15 min	Learn	The new idea for the week, plus the small exercises. He types, you sit nearby, you do not touch the keyboard.
20 min	Build	He adds to the week's project. This is where it gets messy and good. Let it run over if he is absorbed.
5 min	Show	He demos it to somebody — you, a sibling, a grandparent on a video call. Then he writes one line in his log: what he built, and what broke.

IF YOU ONLY HAVE TWO SESSIONS THIS WEEK

Do two. A plan you actually follow at half speed beats a perfect plan you abandon in March. Weeks are units of content, not deadlines. If a week takes ten days, the week took ten days. Falling "behind" is a concept that does not apply here — there is nobody to be behind.

Six rules for you, the teacher

You do not need to know how to code to use this plan. You need to be interested and to hold the line on these six things.

Never take the keyboard. When he is stuck and you can see the missing bracket, it is agonising. Point at the screen instead: "Something on line 4 is upsetting it. Read it out loud to me." If you fix it yourself, you have taught him that stuck means fetch an adult.

Ask questions you know the answer to. "What do you think that line does?" "What did you expect to happen?" "What actually happened?" Those three questions solve maybe seventy percent of beginner bugs without you knowing anything about programming.

Let him make ugly things. His first game will have a purple background and a character called Bob who screams. Do not tidy it. Ownership is the fuel.

Treat errors as normal, not as failure. Red error text is not a scolding, it is the computer being specific about what confused it. When one appears, say "great, it's telling us something" — and mean it. Professional developers see errors hundreds of times a day.

Praise the process, not the brain. "You kept going for twenty minutes on that bug" beats "you're so smart." Children praised for being smart avoid hard things, because hard things threaten the label.

Protect the finish. He will want to abandon week 9's project for a shinier idea in week 10. Sometimes allow it. But make sure that roughly every third project gets properly finished, named, and shown to somebody. Finishing is a muscle and it only grows by finishing.

02. The Year at a Glance

Four stages. Each one ends with a project he chooses, plans, and builds himself.

WEEKS	STAGE	TOOL	WHAT HE CAN DO AT THE END
1–10	**Thinking Like a Computer**	Scratch (browser)	Build a playable game with a score, levels, enemies and sound — and explain loops, conditions and variables in his own words.
11–26	**Real Code**	Python	Write real text programs: games, quizzes, calculators, a text adventure that saves your progress to a file.
27–38	**The Web**	HTML, CSS, JavaScript	Build and publish a real website at a real address that his friends can visit, with pages that respond to clicks.
39–52	**Becoming a Developer**	Git, terminal, chosen path	Use the tools professionals use, plan a project from scratch, and ship a capstone he designed himself.

WHY SCRATCH FIRST, WHEN HE WANTS "REAL" CODING

Twelve-year-olds often find Scratch babyish and want to skip to Python. Here is the honest argument to give him, in his own terms: in Scratch, an idea takes four minutes to test. In Python, the same idea takes forty, because half the time is spent on spelling. Ten weeks in Scratch means he tests roughly a hundred ideas about how programs work. That is the fastest available route to being good at Python, not a detour around it. Also worth saying plainly: he is allowed to find it easy. Easy means move faster, not skip.

If he genuinely hates it after three weeks, compress Scratch to weeks 1–5 (do weeks 1, 2, 4, 5 and jump to the milestone project) and start Python early. A motivated child moving fast beats a resentful child moving correctly.

What he needs, and what it costs

Nothing in this plan costs money. He needs a computer with a web browser — Windows, Mac, Chromebook or a Linux machine all work, and stages 1, 3 and most of stage 2 run entirely in the browser, so even a school Chromebook is fine. Headphones help. A cheap notebook and pen for sketching game ideas helps more than you would think.

If he has a full Windows, Mac or Linux computer, around week 14 he will install Thonny, a free Python editor designed for beginners. If he only has a Chromebook or tablet, he stays in the

browser for the whole year using the online editors listed on the resources page, and loses nothing important.

ACCOUNTS, AGES AND SAFETY — READ THIS BEFORE WEEK 1

Scratch allows accounts for under-13s but asks for a parent's email address, and you will get the confirmation mail. Turn on the setting that requires your approval for comments. The Scratch community is heavily moderated but it is still a comment section on the internet.

GitHub requires users to be at least 13. He turns 13 during or after this plan; until then, in week 39, create the account in *your* name and let him use it while you are in the room. This is the normal arrangement and costs nothing.

Anything with AI code assistants — hold these off until at least week 27, and preferably later. A tool that writes the answer for him removes exactly the struggle that builds the skill. There is a right time for those tools and it is after he can already do the thing.

STAGE ONE · WEEKS 1–10

Thinking Like a Computer

Tool: Scratch at scratch.mit.edu · Goal: every core idea in programming, with zero typing

Everything a professional developer does every day — sequence, repetition, decisions, storing values, sending messages between parts of a program — exists in Scratch as coloured blocks he drags together. He will learn all of it here, fast, and then meet it again in Python already knowing what it means. The only thing Scratch does not teach is typing syntax, which is the part that takes a fortnight to pick up.

WEEK 1 · SEQUENCE

Make something move

IDEA	A program is a list of instructions, done in order, exactly as written. The computer is not clever and does not guess what you meant.
LEARN	The Scratch screen: stage, sprite list, block palette, script area. Green flag to run, red to stop. Motion, Looks and Sound blocks.
EXERCISES	Make the cat move 10 steps. Make it say "hello" for 2 seconds. Make it change colour. Now make it do all three, in an order *you* choose — then change the order and watch what happens.
PROJECT	**Name Badge.** A sprite walks in from the left, stops in the middle, says his name, plays a sound, and grows. Around ten blocks.
ASK HIM	"What happens if we swap those two blocks?" Have him predict *before* running it. Predicting is the whole skill in miniature.

WEEK 2 · LOOPS

Making the computer repeat itself

IDEA Computers are stupid but tireless. Anything you would do twice, tell it to do a hundred times instead. `repeat` runs a set number of times; `forever` never stops.

LEARN Control blocks: `repeat`, `forever`, `wait`. The Pen extension (add it from the bottom-left button).

EXERCISES Draw a square with pen down, repeat 4, move 100, turn 90. Then a triangle (repeat 3, turn 120). Then work out the rule connecting the number of sides to the turn angle — he will discover 360 ÷ sides himself, which is a genuinely lovely moment.

PROJECT **Spirograph.** A loop inside a loop: repeat 36 times { draw a square, turn 10 degrees }. It produces something beautiful in eight blocks and it never stops being satisfying.

WEEK 3 · EVENTS & MESSAGES

Making things happen at the right moment

IDEA Programs do not only run top to bottom. They wait for things: a key press, a click, a message from another part of the program. This is how every app you have ever used works.

LEARN Event blocks: green flag, key pressed, sprite clicked. Broadcast and "when I receive" — one sprite sending a message to another.

EXERCISES Arrow keys move a sprite around. Space bar makes it jump. Clicking it plays a sound.

PROJECT **The Conversation.** Two sprites have a five-line back-and-forth conversation, taking turns properly using broadcasts — not by guessing at `wait` timings. Make it funny. Funny is a legitimate design goal.

ASK HIM "Why doesn't using wait blocks work as well as broadcasts?" This is his first taste of why programs need to coordinate rather than hope.

WEEK 4 · CONDITIONS

Teaching the program to decide

IDEA `if` means: check something, and only do this when it is true. The computer checks it every single time round the loop.

LEARN `if / then`, `if / then / else`. Sensing blocks: touching colour, touching sprite, key pressed. Boolean (true/false) blocks are the pointy hexagonal ones.

EXERCISES If touching red, say "ouch". If touching the edge, bounce. If touching the green goal, say "you win".

PROJECT **Maze Game, part 1.** Draw a maze background in the paint editor. Arrow keys move the player. Touching a wall sends him back to the start. Reaching the goal shows a win message.

WEEK 5 · VARIABLES

Remembering things

IDEA A variable is a labelled box the program can put a value into and read back later. Score, lives, time, level — all boxes.

LEARN Making a variable. `set` versus `change by` — the single most common beginner confusion, worth ten minutes on its own. Showing and hiding variables on stage.

EXERCISES A counter that goes up when you press space. A countdown from 10 that says "liftoff" at zero.

PROJECT **Maze Game, part 2.** Add a score that rises when he collects coins, a lives counter that drops on hitting a wall, game over at zero lives, and a timer. It is now an actual game.

ASK HIM "Where do you have to put 'set score to 0' so it works properly?" Getting this wrong — and finding out why the score never resets — is a better lesson than getting it right first time.

WEEK 6 · COORDINATES & MOTION

Where things are, in numbers

IDEA The stage is a grid: x is across (−240 to 240), y is up and down (−180 to 180). Every position is two numbers, and movement is arithmetic on those numbers.

LEARN `go to x y`, `change x by`, `glide`, `if on edge bounce`, direction and turning.

EXERCISES Move a sprite to each corner in turn. Make one sprite follow another with `point towards`. Make a ball bounce off all four walls.

PROJECT **Pong.** A paddle on the left controlled by mouse or keys, a ball that bounces, a score, and a lose condition when the ball gets past. The classic for a reason — every concept so far, in one small game.

WEEK 7 · CLONES

Many copies of one thing

IDEA You do not build fifty enemies by hand. You build one, and tell the program to make copies of it that each run the same instructions independently.

LEARN `create clone of myself`, `when I start as a clone`, `delete this clone`. Why deleting clones matters (leave them and the project slows to a crawl — his first performance bug).

EXERCISES Rain: clones fall from random x positions at the top and delete at the bottom. Stars that twinkle at random spots.

PROJECT **Catcher.** Objects fall from the sky at increasing speed; a basket at the bottom catches them. Good things add points, bad things cost a life. Difficulty rises as the score rises.

WEEK 8 · LISTS

Storing many values at once

IDEA A list is a numbered row of boxes under one name. Where a variable holds one thing, a list holds a hundred — questions, high scores, names, inventory.

LEARN Making a list, `add`, `item n of`, `length of`, `delete`. Using a loop counter to walk through a list one item at a time.

EXERCISES A list of five jokes, one picked at random. A list of names read out one by one.

PROJECT **Quiz Show.** Questions in one list, answers in another, matched by position. Ask each in turn, check the answer, keep score, show a final grade with a message that changes depending on how well he did.

WEEK 9 · DEBUGGING & TIDYING

The week nobody teaches, and everybody needs

IDEA Working code and *good* code are different things. Good code is code you can still understand next month.

LEARN A method for finding bugs: describe what should happen, describe what does happen, find the first place the two disagree. Slow the program down with `wait` blocks to see it working. Rename sprites and variables so the names say what they are.

EXERCISES Deliberately break one of his old projects in three ways, hand it to a sibling or parent to fix, then swap. Breaking things on purpose is how you learn what breakage looks like.

PROJECT Take the week 5 maze game and improve it without adding a single new feature: better names, no duplicated block stacks, a proper start screen. Then add one feature he has wanted all along.

WEEK 10 · MILESTONE PROJECT 1

His own game, start to finish

THE BRIEF Any game he likes. It must use, somewhere: a loop, an `if`, at least two variables, a broadcast, and clones. Beyond that it is entirely his.

SESSION 1 No computer. Paper only. He draws the screen, lists what the player can do, lists what can go wrong, and writes the rules. Then he cuts the idea in half, because it is too big — it always is, for everyone, forever.

SESSIONS 2–3

Build the smallest playable version first: one thing moving, one way to lose.
Only then add.

SESSION 4 Playtest with a real person who has not seen it. Watch where they get confused without helping them. Fix those things. Add instructions. Give it a title screen and a proper name.

THEN Share it on the Scratch website (with your account settings applied) and send the link to a grandparent. The moment somebody outside the house plays his game is the moment this stops being homework.

STAGE 1 CHECKPOINT — HE IS READY TO MOVE ON WHEN HE CAN

- Explain, in his own words and without prompting, what a loop, an `if`, and a variable are.
- Take an idea like "the enemy should get faster over time" and work out the blocks himself.
- Find a bug in his own project without you in the room.
- Say what a program does *before* running it, and be right most of the time.

If two of these are shaky, spend an extra fortnight making more games. There is no prize for reaching Python in week 11.

STAGE TWO · WEEKS 11–26

Real Code

Tool: Python · Goal: write, read, fix and finish real programs in a real language

Python is the right first text language: the code reads almost like English, there are no semicolons or curly braces to forget, and it is genuinely the language used by professionals for AI, science, web servers and automation. Nothing he learns here is training wheels.

For weeks 11 to 13 he works in a browser editor so there is nothing to install and no excuse not to start. Around week 14, if he has a Windows, Mac or Linux computer, he installs **Thonny** — a free editor built specifically for beginners, with the clearest error messages of any Python tool. On a Chromebook he simply stays in the browser all year.

THE FIRST WEEK WILL FEEL LIKE A STEP BACKWARDS

After ten weeks of dragging colourful blocks that cannot be spelled wrong, he now has a blank white screen and can break everything with a missing quote mark. Expect one frustrating session. Tell him in advance that it is coming and that it lasts about two weeks. Then it clicks, and he never wants to go back.

WEEK 11 · FIRST CONTACT

Making the computer talk

IDEA `print()` puts text on the screen. Text in quotes is a *string*. The computer is fussy about spelling and always will be.

LEARN `print()`, quote marks, comments with `#`, and the fact that `Print` with a capital P is an error. Read one error message together, slowly, all the way through.

```
# My first program
print("Hello! My name is Alex and I just wrote my first program.")
print("Ask me anything.")
print("...")
print("Anything.")
```

EXERCISES Print a joke with the punchline on a second line. Print a picture made of keyboard characters. Deliberately remove a quote mark and read what the computer says about it.

PROJECT **Joke Bot.** Ten lines that tell a short story or joke, with comments explaining each part.

WEEK 12 · VARIABLES & INPUT

Programs that listen

IDEA A variable in Python is the same labelled box as in Scratch, written `name = "Alex"`. `input()` asks the user a question and hands back what they typed — always as text, even when it looks like a number.

LEARN `=` means "put this in the box", not "equals". Types: string, integer, float. `int()` to convert. f-strings for mixing text and values.

```
name = input("What is your name? ")
age = int(input("How old are you? "))

print(f"Nice to meet you, {name}!")
print(f"Next year you will be {age + 1}.")
```

EXERCISES Ask for a favourite colour and animal, then print a silly sentence using both. Try removing the `int()` and adding 1 to the age — the error that appears is the whole lesson about types.

PROJECT **The Interviewer.** Six questions, then a paragraph-long profile of the person built from their answers.

WEEK 13 · MATHS

Numbers that do useful work

IDEA Programs calculate. Two operators are new and worth knowing: `//` divides and throws away the remainder, `%` gives you only the remainder.

LEARN `+ - * /`, then `//`, `%`, `**`, and `round()`. Why `%` is secretly everywhere: even/odd, wrapping around a clock, every third item.

```
slices = int(input("How many pizza slices? "))
people = int(input("How many people? "))

each = slices // people
left_over = slices % people

print(f"Everyone gets {each} slices.")
print(f"There are {left_over} slices left over.")
```

EXERCISES Convert Celsius to Fahrenheit. Work out how many days until his birthday. Check whether a number is even using `%`.

PROJECT **Pocket Money Planner.** Takes weekly allowance and the price of something he wants, and reports how many weeks of saving it needs — and what the date will be.

WEEK 14 · DECISIONS

if, elif, else

IDEA Same idea as Scratch, new shape. In Python, the colon and the indentation are what say "this bit belongs inside the if". Indentation is not decoration — it is the grammar.

LEARN `if / elif / else`, comparisons `== != < > <= >=`, and `and / or / not`. The classic trap: `=` assigns, `==` compares.

ALSO THIS WEEK

Install Thonny (free, from thonny.org) if he is on a full computer. Show him the Run button and how errors appear at the bottom.

```
guess = int(input("Guess my number: "))
secret = 7

if guess < secret:
    print("Too low!")
elif guess > secret:
    print("Too high!")
else:
    print("You got it!")
```

EXERCISES A grade calculator (90+ is an A, and so on). A program that says whether he can watch a film based on age and whether homework is done — that one needs `and`.

PROJECT **Personality Quiz.** Five questions, scores tracked, and one of four endings depending on the total.

WEEK 15 · WHILE LOOPS & RANDOMNESS

Keep going until

IDEA A `while` loop repeats for as long as something stays true. If it never stops being true, the program hangs forever — and he should absolutely make that happen once, on purpose, and learn to press stop.

LEARN `while`, `break`, counters, and `import random` with `random.randint()` and `random.choice()`.

```
import random

secret = random.randint(1, 100)
tries = 0

while True:
    guess = int(input("Guess (1-100): "))
    tries = tries + 1

    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
    else:
        print(f"Got it in {tries} tries!")
        break
```

PROJECT **Guess My Number, both ways.** First he guesses the computer's number. Then — much more interesting — the computer guesses *his*, and he tells it higher or lower. Ask him to make the computer as fast as possible. He will invent binary search on his own, which is a real computer science algorithm and he will be twelve.

WEEK 16 · FOR LOOPS

Repeat a known number of times

IDEA Use `while` when you do not know how many times; use `for` when you do.

LEARN `for i in range(10)`, and how `range` counts from 0 up to but not including the end. Nested loops.

EXERCISES Print a times table. Count down from 10 to "liftoff". Draw a triangle of stars with a nested loop.

PROJECT **Times Table Trainer.** Ten random questions, immediate feedback, a score at the end, and a list of the ones he got wrong so he can practise them again.

WEEK 17 · LISTS

Many things, one name

IDEA Same as Scratch lists, but counting starts at zero. This trips up every single beginner and he is no exception. Zero-indexing is worth its own ten minutes.

LEARN Square brackets, `append()`, `remove()`, `len()`, indexing, slicing, and the `in` keyword.

PROJECT **Shopping List Manager.** A menu loop that lets you add an item, remove one, view the list, or quit. This is the first program that feels like software rather than an exercise.

WEEK 18 · LOOPS OVER LISTS

Doing something to everything

IDEA `for item in my_list` walks through every element without you managing a counter. This is the most-used line in all of Python.

LEARN Looping over lists, building a new list inside a loop, and `sum()`, `max()`, `min()`, `sorted()`.

PROJECT **Grade Tracker.** Enter any number of test scores, then report the average, highest, lowest, and how many were above average.

WEEK 19 · FUNCTIONS

Teaching Python new words

IDEA A function is a chunk of code with a name. Write it once, use it anywhere. This is how programs stay manageable once they pass about fifty lines.

LEARN `def`, parameters, `return`, and the difference between a function that prints and one that returns — subtle, important, worth doing slowly.

```
def area_of_rectangle(width, height):
    return width * height

print(area_of_rectangle(4, 5))   # 20
print(area_of_rectangle(10, 3))  # 30
```

PROJECT Go back to the week 16 and week 18 programs and rewrite them using functions. Nothing new appears on screen — the program behaves identically — and that is exactly the point. He is learning that code has an inside as well as an outside.

WEEK 20 · STRINGS IN DEPTH

Taking text apart

IDEA Text is a sequence of characters, which means you can loop over it, slice it, and rebuild it.

LEARN `upper()`, `lower()`, `strip()`, `replace()`, `split()`, `join()`, slicing with `[1:4]`, and `.index()`.

```
def encode(message, shift):
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    coded = ""

    for letter in message.lower():
        if letter in alphabet:
            old_spot = alphabet.index(letter)
            new_spot = (old_spot + shift) % 26
            coded = coded + alphabet[new_spot]
        else:
            coded = coded + letter

    return coded

print(encode("meet me at noon", 3))   # phhw ph dw qrrq
print(encode("phhw ph dw qrrq", -3))  # meet me at noon
```

PROJECT **Secret Code Machine.** Encode and decode messages with a Caesar cipher. Then the challenge: write a cracker that tries all 26 shifts and prints them, so you can break a message without the key. Send coded notes to each other for a week.

WEEK 21 · DICTIONARIES

Looking things up by name

IDEA A list finds things by position. A dictionary finds them by *name* — which is almost always what you actually want.

LEARN Curly braces, key–value pairs, looking up, adding, checking with `in`, and looping over `.items()`.

```
capitals = {
    "France": "Paris",
    "Japan": "Tokyo",
    "Egypt": "Cairo",
}

country = input("Name a country: ")

if country in capitals:
    print(f"The capital is {capitals[country]}.")
else:
    print("I don't know that one yet!")
```

PROJECT **Anything Translator.** English to emoji, English to Pirate, or a made-up family language. Dictionary lookup on every word of a sentence, leaving unknown words alone.

WEEK 22 · FILES

Remembering after the program closes

IDEA Everything so far vanished when the program ended. Files are how programs remember — and this is the week his projects start to feel real.

LEARN `open()` with `with`, reading, writing, appending, and `try / except FileNotFoundError` for the first run when the file does not exist yet.

```
def save_score(score):
    with open("highscore.txt", "w") as f:
        f.write(str(score))

def load_score():
    try:
        with open("highscore.txt") as f:
            return int(f.read())
    except FileNotFoundError:
        return 0
```

PROJECT Add a saved high score to the week 15 guessing game, then build a **Journal** program that appends dated entries to a file and can read them all back.

WEEK 23 · HANDLING MISTAKES

Programs that survive the user

IDEA Users type nonsense. Not because they are stupid, but because they are human. A good program expects it. Have him hand a program to a younger sibling with instructions to break it — they will manage in under a minute.

LEARN `try / except`, catching `ValueError` and `ZeroDivisionError`, and validation loops that re-ask until the input is sensible.

PROJECT **The Unbreakable Calculator.** It must not crash on letters, on empty input, or on dividing by zero. Ten minutes of a sibling attacking it is the acceptance test.

WEEK 24 · MODULES & TURTLE

Standing on other people's work

IDEA Nobody writes everything themselves. `import` borrows code other people already wrote and tested.

LEARN `random`, `time`, `math`, `datetime`, and `turtle` for drawing. What a library is and why professionals use hundreds of them.

PROJECT **Turtle Art.** Spirals, snowflakes, and a recursive tree. This closes a loop from week 2 — the same spirograph idea, now in real code, with real functions, and infinitely more control.

WEEK 25 · DESIGNING A BIGGER PROGRAM

How to build something that will not fit in your head

IDEA Big programs are not written top to bottom. They are sketched, split into functions, and built one working piece at a time.

LEARN Planning on paper. Choosing data structures — why rooms belong in a dictionary and an inventory belongs in a list. Writing a function skeleton with comments first, then filling it in.

DO Plan the week 26 adventure game completely, on paper, before writing any code. Map the rooms, list the items, decide how to win.

WEEK 26 · MILESTONE PROJECT 2

The Text Adventure

THE BRIEF An explorable world he invents, played entirely by typing commands. Every Python idea from the stage, in one program of maybe 150 lines — genuinely substantial work for a beginner.

MUST HAVE At least six rooms in a dictionary. Movement between them. Items you can pick up, held in a list. At least one puzzle that needs an item. A way to win and a way to lose. Saving and loading progress to a file. Input that does not crash on nonsense.

```
rooms = {
    "hall": {
        "description": "A dusty hall. Doors lead north and east.",
        "north": "kitchen",
        "east": "library",
        "item": "rusty key",
    },
}

def move(current_room, direction):
    room = rooms[current_room]
    if direction in room:
        return room[direction]
    print("You can't go that way.")
    return current_room
```

TAKE FOUR SESSIONS

Rooms and movement first. Then items. Then the puzzle. Then saving. Then hand it to a family member and watch them play without helping.

STAGE 2 CHECKPOINT — HE IS READY TO MOVE ON WHEN HE CAN

- Write a 30-line program from a plain-English description without looking anything up.
- Read an error message, find the line it names, and usually fix it alone.
- Choose sensibly between a list and a dictionary, and say why.
- Break a problem into functions before starting to type.

STAGE THREE · WEEKS 27–38

The Web

Tools: HTML, CSS, JavaScript · Goal: build something at a real address that his friends can visit

This stage exists for one reason above all others: at the end of it he can send a link to a friend. Everything he has built so far lived on one computer. A website lives everywhere, and for a twelve-year-old that difference is enormous.

Three languages, three jobs. HTML is the *structure* — the skeleton that says this is a heading and that is a picture. CSS is the *appearance* — colours, spacing, layout. JavaScript is the *behaviour* — what happens when you click. Keep that three-way split clear in his head from day one and the rest is much easier.

WEEK 27 · HTML

The skeleton of every web page

IDEA A web page is a text file with tags marking what each part is. That is the entire secret, and it is a bit of a let-down in a wonderful way.

LEARN The page skeleton, then `<h1>` to `<h3>`, `<p>`, `<ul>`, `<img>`, `<a>`. Opening and closing tags. Saving as `.html` and opening it in a browser.

DO THIS FIRST

Open any website he likes and press F12, or right-click and choose Inspect. Show him it is all just tags underneath. Let him edit the text of a headline on a site he knows. The look on his face is worth the whole week.

PROJECT **All About Me.** A page with a heading, a photo, three paragraphs and a list of favourite things.

WEEK 28 · MORE HTML

Links, images and structure

LEARN Linking between his own pages, image sizing, `<div>` and `<section>` for grouping, tables, and simple forms. Why `alt` text on images matters for people using screen readers — a good first conversation about building for everybody.

PROJECT Turn last week's page into a three-page site — Home, Projects, Contact — with a navigation bar that works on every page.

WEEK 29 · CSS

Making it look like something

LEARN Selectors by tag, class and id. Colours (including hex codes), fonts, and the box model: margin, border, padding, content. The box model is the one CSS concept that repays being drawn on paper.

PROJECT Style the site. He will make it garish. Let him. Then ask him to make a second version copying the colours of a website he admires, and talk about the difference.

WEEK 30 · LAYOUT

Putting things where you want them

LEARN Flexbox — `display: flex`, `justify-content`, `gap` — which handles most real layouts. Then a media query so the site does not look broken on a phone. Test on an actual phone; it is far more motivating than a resized window.

PROJECT A card grid of his projects that rearranges itself into a single column on a phone.

WEEK 31 · PUBLISHING

Putting it on the actual internet

LEARN Free hosting. Neocities is the simplest and has no age restriction beyond a parent's email; GitHub Pages is the more professional route but needs the account arrangement described on page 4.

PROJECT Publish the site. Send the link to family. Then — importantly — fix the three things that look wrong on somebody else's screen. Shipping and then fixing is the actual rhythm of the job.

WEEK 32 · JAVASCRIPT

A second language, learned in a week

IDEA He already knows how to program. This is the same set of ideas wearing different clothes — and noticing that is the real lesson of the week. Languages are dialects, not different worlds.

LEARN `let` and `const`, `console.log()`, the browser console, functions with curly braces, `if` and `for`. Sit down and write out the Python version and the JavaScript version of the same five-line program side by side.

PROJECT Rewrite the week 14 personality quiz in JavaScript, running in the console.

WEEK 33 · THE DOM

JavaScript that changes the page

IDEA The page is an object JavaScript can reach into and change while it is running. This is what makes a web page an app rather than a poster.

LEARN `document.getElementById()`, changing `textContent` and `style`, and `addEventListener("click", ...)`.

PROJECT A button that counts clicks, a button that changes the background colour, and a dark-mode toggle for his own site.

WEEKS 34–35 · A REAL WEB APP

Putting the three languages together

LEARN Reading values out of form inputs, arrays of objects to hold data, showing and hiding elements, and building HTML from JavaScript in a loop.

PROJECT **Quiz App.** Questions stored in an array, one shown at a time, buttons for answers, instant right/wrong feedback, a score, and a results screen with a restart button. This is a genuine small application and he should be proud of it.

WEEKS 36–37 · CREATIVE CODING WITH P5.JS

Drawing and animation in the browser

LEARN The p5.js web editor. `setup()` and `draw()`, the canvas, shapes, colours, `mouseX` and `mouseY`, and animating by changing a variable every frame. The `draw()` loop is exactly Scratch's `forever` block — say so out loud.

PROJECT An interactive artwork that responds to the mouse, then a small game — dodge the falling blocks, or a paddle-and-ball. Compare it to the Scratch version from week 6 and talk about what is easier and what is harder now.

WEEK 38 · MILESTONE PROJECT 3

His portfolio site

THE BRIEF A real website, published at a real address, that shows off everything he has made this year. Scratch games embedded, screenshots of Python programs, the quiz app playable in the page, the p5.js sketch running live.

MUST HAVE A home page that says who he is. A project page for each milestone with a screenshot and a paragraph on what was hard about it. Working navigation. Decent on a phone. Published and shareable.

WHY THIS MATTERS

Writing "what was hard about this" is the part that lasts. It is the first time he explains his own work to a stranger, which is half of a developer's actual job — and in five years, a version of this page is what gets him an internship.

STAGE FOUR · WEEKS 39–52

Becoming a Developer

Tools: Git, the terminal, and a path he chooses · Goal: work the way professionals work

He can code now. This stage is about everything around the code — the tools, habits and judgement that separate someone who writes programs from someone who builds software. It is also where he starts steering.

WEEK 39 · GIT & GITHUB

A time machine for code

IDEA Git saves snapshots of a project so you can always go back. Every professional team on earth uses it. It is also the cure for the "I broke everything and can't undo it" catastrophe.

LEARN `git init`, `git add`, `git commit -m`, `git log`, `git push`. Writing a commit message that says what changed and why.

DO Put every project from the year into repositories. Write a README for each — what it does, how to run it, what he learned.

REMEMBER GitHub requires account holders to be 13 or over. Until his birthday, the account is yours and he uses it with you present.

WEEK 40 · THE COMMAND LINE

Talking to the computer directly

LEARN `cd`, `ls` or `dir`, `mkdir`, running a Python file with `python file.py`, and installing a package with `pip install`. Enough to stop being frightened of a black window, which is the entire goal.

WEEK 41 · GETTING UNSTUCK

The most valuable week in this document

IDEA Professional developers are stuck most of the day. The skill is not avoiding it — it is having a reliable procedure for getting out.

LEARN Read the error properly: which file, which line, what type. Search well: paste the error text, not a description of your feelings about it. Read official documentation. Rubber-duck debugging — explain the code out loud, line by line, to an object; the bug frequently reveals itself mid-sentence. And how to ask a good question: what I tried, what I expected, what happened, here is the code.

EXERCISE Hand him three programs you have deliberately broken in different ways and time how long each takes. Do it again in week 50 and compare.

WEEK 42 · PLANNING LIKE A PROFESSIONAL

From "I want to build X" to a list of small jobs

LEARN User stories in plain English: "as a player, I want to see my high score." Sketching screens on paper. Splitting a project into tasks of an hour or less. Deciding what version one does *not* include — the single most useful professional habit there is.

WEEKS 43–45 · CHOOSE A PATH

He picks the direction now

Three sessions of guided exploration down whichever road interests him. There is no wrong choice and he can do another one later.

GAMES Pygame in Python: sprites, the game loop, collision detection, sound. Build a complete small arcade game.

WEB APPS Deeper JavaScript, saving data in the browser, fetching live data from a free public API — a weather app or a jokes app that pulls real information from the internet.

DATA & AI Python with pandas and matplotlib: load a real spreadsheet — football results, weather, his own game scores — and produce charts that answer a question he cares about.

ROBOTICS If there is any budget at all, a micro:bit or Raspberry Pi Pico. Code that moves something in the physical world hits differently for a lot of children.

WEEKS 46–49 · THE CAPSTONE

Something he designed, built and shipped himself

THE BRIEF Anything, as long as it is genuinely his idea and genuinely useful or fun to somebody who is not him. Four weeks. You are now a project manager, not a teacher.

WEEK 46 Pitch it to you in five minutes. Write the user stories. Sketch the screens. Cut the scope in half. Then cut it again.

WEEKS 47–48

Build. Commit to Git at the end of every session. Keep a running list of bugs rather than chasing each one as it appears.

WEEK 49 Test it on three real people. Fix what confuses them, not what annoys him. Write the README. Publish it.

WEEK 50 · A FIRST TASTE OF COMPUTER SCIENCE

Why some code is fast and some is slow

LEARN Linear search versus binary search — connect it back to the guessing game in week 15, where he invented binary search without knowing its name. Time a program that searches a list of a million items both ways and watch the difference. That is the whole intuition behind algorithms, and he now owns it.

WEEK 51 · SHOW YOUR WORK

Demo day

DO A real presentation to family, or a video: here is the problem, here is what I built, here is a demo, here is what broke and how I fixed it, here is what I would do next. Ten minutes. Rehearsed. Explaining technical work to people who are not technical is a career skill and it starts now.

WEEK 52 · LOOK BACK, LOOK FORWARD

One year of work in one place

DO Open the week 1 Scratch project and the capstone side by side. Let that land — twelve months ago he could not make a cat walk across a screen. Then he writes his own plan for year two: what to learn, what to build, and one thing he wants to be able to do by his next birthday.

03. After the Year: The Road to Software Developer

What comes next, roughly, from age 13 to the first paid job.

The one-year plan is the hard part, because it takes him from nothing to competent. What follows is less structured on purpose: from here his own interests should drive it, and the best thing you can do is fund the curiosity and stay out of the way.

AGE	FOCUS	WHAT THAT MEANS IN PRACTICE
13–14	Depth in one language, plus real projects	Go deep on Python or JavaScript rather than collecting new languages. Learn how programs are organised into objects and classes. Build three or four projects that other people actually use. Enter a competition or a game jam — deadlines and an audience do more for skill than any course.
14–15	Fundamentals and collaboration	Data structures and algorithms properly: lists, stacks, queues, trees, hash tables, sorting, and Big-O notation. How databases store things. Contributing to an open-source project, which means working in somebody else's code and taking review from strangers — a genuine step up.
15–16	Building complete systems	A full application with a front end, a back end, a database and user accounts, deployed somewhere real. Learn testing, and why professionals write code that checks their code. If school offers computer science, take it — but by now he will be ahead of it, and that is fine.
16–18	Specialising and going public	Pick a direction: web, mobile, games, data science, machine learning, embedded systems, security. Build a portfolio and a GitHub history that shows a year of steady commits. Internships, freelance work for a local business or charity, hackathons. Many developers get their first paid work at 16 or 17 this way.
18+	Formal study, or straight into work	A computer science degree is a strong route and teaches the theory that is hard to pick up alone. It is not the only route — plenty of excellent developers arrive through self-teaching, apprenticeships and bootcamps, and the industry generally judges on what you can build. The right choice depends on the country, the cost, and the child.

WHAT ACTUALLY GETS PEOPLE HIRED, IN ORDER

Things they have built that you can look at and use. Then: the ability to explain their thinking out loud while solving a problem. Then: being someone a team wants in the room. Then, some distance behind: credentials. This ordering has held steady for years, and it is good news for a twelve-year-old, because he can start accumulating the first one this week.

Two honest warnings

He will want to quit somewhere around month four. Almost everyone does. The novelty has worn off, the projects are harder, and the gap between what he can build and what he imagines is at its widest. The fix is not discipline — it is a project he badly wants to exist. Let him abandon the curriculum for two weeks to build something silly he cares about. Motivation is a resource to be managed, not a character trait to be judged.

Do not let it become school. If coding starts to feel like an obligation with a parent checking output, it will end when the obligation can be dropped. Keep it closer to a hobby you happen to be enthusiastic about together. Ask to see things. Be impressed. Ask him to teach you something and let him actually teach you — being the expert in the room, at twelve, is powerful.

04. When Things Go Wrong

The five situations you will definitely hit, and what to do about each.

"I'm bored, this is too easy"

Take him seriously — it may well be true. Skip the exercises entirely and go straight to the projects, adding a constraint that makes it interesting: build it without using any `if` statements, or make it work for two players, or finish it in twenty minutes. If a whole stage is too easy, compress it. Boredom in a capable child is a signalling problem, not a motivation problem.

"I don't get it and I'm stupid"

Stop the session. Do not push through — pushing through is how the association forms. Go back two weeks to something he found easy and let him build something with it, so the last thing he feels is competence. Come back to the hard concept in a few days from a different angle, ideally a different explanation from a video rather than a repeat of yours. Concepts arrive suddenly after a delay, not gradually under pressure. Say out loud that being stuck is what the job *is*, and that you have watched him get unstuck before.

He only wants to play games, not make them

Meet it head on. Have him modify a game he already plays — Minecraft with Python mods, Roblox with Lua scripting. It is real coding, and the connection to something he loves does the motivational work for you. Ask him what one thing about his favourite game annoys him, and build the fix.

He wants to use AI to write the code

Reasonable, and eventually correct — professionals use these tools daily. But not yet, and here is the argument to give him: an AI that writes his code for him is a spotter who lifts the weight. Nothing happens. From week 27 onward, one narrow permitted use: he may ask an AI to *explain* an error message or a concept, never to write or fix code he has not first tried himself. After this year, that restriction can loosen, because by then he will be able to tell when the machine is wrong — and that is the actual skill.

Weeks have gone by and nothing has happened

Do not restart from week 1 and do not have a conversation about commitment. Sit down together and rebuild one small thing he already made, and get one win on the screen inside thirty minutes. Then carry on from where you were. Gaps are normal; the only fatal move is treating the gap as evidence that it is not working.

05. Everything He Needs, All Free

TOOL	WHERE	WHAT IT IS FOR
Scratch	scratch.mit.edu	Stage 1. Runs in the browser, free, no download. An offline version exists for use without internet.
Thonny	thonny.org	The friendliest Python editor for beginners. Free, Windows/Mac/Linux. Install around week 14.
Online Python editors	trinket.io/python online-python.com	Python in the browser for Chromebooks, tablets, or before installing anything.
p5.js Web Editor	editor.p5js.org	Weeks 36–37. Creative coding and animation, entirely in the browser.
Neocities	neocities.org	Free, simple website hosting. The easiest way to get his site onto a real address.
GitHub & Pages	github.com	Version control and free hosting. Age 13+ — parent-held account until then.
freeCodeCamp	freecodecamp.org	Free interactive courses. Excellent as extra practice for stage 3.
Code.org	code.org	Free structured courses. Good backup or supplement for stage 1.
MDN Web Docs	developer.mozilla.org	The reference for HTML, CSS and JavaScript. Teach him to look things up here rather than in a random blog.

Books worth buying, if you buy any

Two are consistently good for this age: *Python for Kids* by Jason Briggs, which covers roughly the ground of stage 2 with a lighter touch, and *Automate the Boring Stuff with Python* by Al Sweigart, which is free to read online and becomes useful around week 20 when he starts wanting his programs to do real work.

The weekly log

Print this page a few times, or use a plain notebook. Two minutes at the end of each week. The "what broke" column is the valuable one — after six months it becomes a record of

problems he can now solve without thinking, which is the clearest evidence of progress there is.

WEEK	WHAT I BUILT	WHAT BROKE	HOW IT FELT (1–5)

ONE LAST THING, FOR YOU

The reason this plan asks you to sit nearby for 45 minutes, four times a week, is not that he needs supervision. It is that a child who codes alone in a bedroom usually stops, and a child whose parent is visibly interested in what they made usually does not. You do not have to understand the code. You have to look at the screen and ask what it does.

06. The Fridge Card

Cut this out. Ask the three questions every Friday, for a year.

These three questions do more for a young coder than any curriculum, including this one. The first makes him name his work. The second makes him think of code as a way of solving something rather than a thing you type. The third — *can you show me* — is the one that changes his behaviour all week, because a child who knows a demo is coming builds toward a demo.

Ask them in that order, on the same day each week, and mean the third one. Sit down and actually watch. Two spares are printed below in case one gets lost.

EVERY FRIDAY

1. **What did you build?**
2. **What problem did you solve?**
3. **Can you show me?**

Then praise the **process**, not the brain. “You sat with that bug for twenty minutes and you got it” — not “you’re so smart.”
And never take the keyboard.

EVERY FRIDAY

1. **What did you build?**
2. **What problem did you solve?**
3. **Can you show me?**

Then praise the **process**, not the brain. “You sat with that bug for twenty minutes and you got it” — not “you’re so smart.”
And never take the keyboard.

WHERE THESE CAME FROM

These three questions are the best part of the one-page coaching guide you already had, and they work with any curriculum, including none at all. If everything else in this document falls apart, keep these.