How to teach Computer Programming to Children and Young Adults

A teacher's perspective

This is a set of observations and practical guidelines that I have come up with, based on my experience of teaching computer programming to school children and young adults. Teaching anything to anyone can be a huge challenge – unless of course you are a born teacher. Teaching programming to children and young adults is certainly a specialized activity – one that can be enjoyed and made effective through experience and continuous learning. Mostly, it's about realizing that students actually learn by themselves and the teacher can serve best by becoming a facilitator, mentor, and coach.

Learning objectives:

I personally believe in the Constructivist philosophy of teaching. The underlying goals of constructivism are:

- Active learning (experiential learning, students learn through activity)
- Collaboration (communication and interaction)
- Relevance (to student's universe)
- Developing problem-solving skills

It helps to include interdisciplinary elements while teaching CS concepts. In other words, project work should involve application of CS concepts to various non-CS fields. Many of the objectives above are accomplished when students are able to apply computational ideas to their fields of interest.

In other words, we want students to develop understanding of computational thinking and CS concepts, and not get a panoramic view of as much theory as possible. "Depth of understanding" is higher priority than "breadth of knowledge".

Initial Observations:

- *Programming is about a student teaching the computer, not the computer teaching the student*. As teachers, we must keep this in mind – we must allow students to choose what and how they want to teach their computers.

- *Theory when appropriate*: Generally computer training courses are laden with concepts and acronyms. Students get buried in theory quickly before getting a chance to appreciate the purpose of all of it. Instead of starting with a lot of theory and concepts, you should start with real problem-solving right away and explain concepts as and when necessary. Thus the approach would be to teach theory only when required.

- *Understanding is important, facts are not*: For school-age children and young adults, understanding of underlying principles (why is it so, how does it work, etc.) is more important and interesting. Focus on these ideas and CS principles more than specific language commands and features.

- *The evolution of a solution is very interesting and insightful*. It is tempting (and often necessary due to the lack of time) for teachers to present experiments that validate facts and theories, and also to present readymade solutions to associated problems. All this is interesting. But students benefit immensely if they also get to learn how the experiments were devised or how the solutions were cooked up, because then they would have learnt problem-solving methods and the secret to creativity.

- *When you are learning English, you learn to read first.* When kids learn a language such as English, they first spend considerable time hearing conversations and reading books. Writing usually comes much later. Learning computer programming is thought to be similar to this. Students must get the opportunity to read good programs before starting to write their own. The best way to 'read' a program is to step through it during execution.

Role of the Teacher:

The instructor acts as the lead learner, thus shifting his/her role from being the source of knowledge to being a leader in seeking knowledge. His/her mantra is: "I may not know the answer, but I know that together we can figure it out." He/she should act as the curator of materials, by selecting from an abundance of materials and teaching strategies.

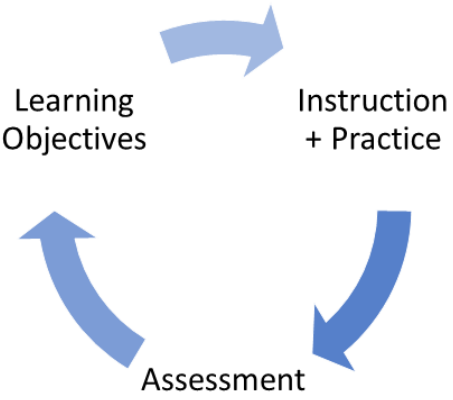
Implications for the instructor:

1. Be open to making mistakes in front of students so that they see it is part of the learning process.
2. Don't give an answer right away, even if you know it. Ask guiding questions that allow students to discover their own solutions.
3. Allow students to dive right into an activity with minimal frontloading. A lead learner recognizes that investigation, mistakes, and iteration are important parts of the learning process.
4. Encourage students to collaborate and seek help.

An effective teacher knows that he cannot teach, but can only inspire his students to learn. Such inspiration can be instilled by demonstrating to the students his own love and passion for CS. Besides making your love for CS evident through your voice, take the trouble to show students examples of programs that you find fun and exciting.

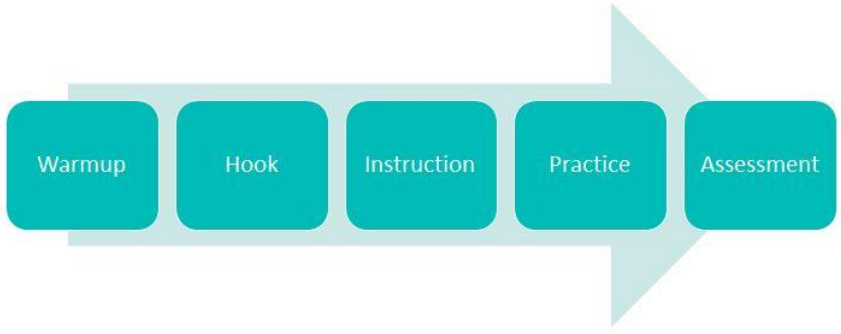
The learning process:

In general, the teaching/learning process looks like this:



This means every lesson must have learning objectives (e.g. "students will create a program that uses looping"). Note that a learning objective must be *assessable*, so, for example, "students will understand looping" is not a good learning objective because it is impossible to assess "understanding". Next, for each learning objective, you will have material to explain the required concepts and practice assignments for students to apply their learning. Finally, you will have the means to assess their learning (which could simply be the students' ability to complete the assignment).

Anatomy of a lesson:



"Warm-up" should:

- Set student mindset, set up the topic for the day
- Review old content or lead into new
- Often a single homework or practice problem or an open-ended question
- Be short, achievable by all

"Hook" is:

An engaging introduction to a new topic. It could be a …

- puzzle
- demo
- show & tell object
- photo
- quote

- mystery
- joke or story
- current event
- promise

"Instruction" can be:

New content or review. It should take 25% of class time and is not just lecture! It can be …

- definitions
- explanations
- demos & examples
- group activities
- research

"Practice" should take 75% of time and it could be:

- worksheets
- group presentations
- labs
- projects
- textbook problems
- Can be combined with instruction

Assessment is of two types:

Formative is about:

- "What are they learning?"
- Multiple times per lesson
- Smaller, fewer points (if any)
- Quiz, labs, discussion, questions

Summative is about:

- "What did they learn?"
- Once or twice per unit
- Larger, more points
- Tests, projects

Pedagogical tools:

Journaling:

Provide students a journal at the beginning of the school year. Prompt students to journal about design decisions, specific challenges, or bugs they encounter. Give students time to revisit previous journal entries and reflect on their growth.

Think-Pair-Share:

Think-Pair-Share is a three part activity where students are presented with a problem or task to work on.

Think: First, students work individually. Working individually gives students the opportunity to collect their thoughts before communicating them with others. They should write down their thoughts in a journal for later sharing.

Pair: Once students have had time to work individually, they then enter the "Pair" stage where they work with a small group. These groups can consist of two or three students. The group discusses the thoughts each member collected during the "Think" stage.

Share: Finally, the groups will share out some of the ideas they discussed to the whole class and the discussion will continue as needed in the whole group setting.

Peer Feedback:

Peer feedback is the practice of students sharing their work with one another in order to prompt discussion, solicit suggestions, and iteratively improve their work. Peer feedback provides students opportunities to learn from each other, both by seeing ways others approach the same problem and by incorporating feedback to improve their own work.

How to use it:

1. Create a structured peer feedback process.
2. Give students time to seek, provide, and incorporate feedback.
3. Provide examples of constructive feedback.
4. Treat this as a skill that students will need to develop.

Pair programming:

Pair programming is a technique in which two programmers work together at one computer. One, the driver, writes code while the other, the navigator, directs the driver on the design and setup of the code. The two programmers switch roles often.

How to use it:

1. Form pairs and give each pair one computer to work on
2. Assign roles: driver and navigator
3. Ensure that students switch roles at regular intervals (every 3 to 5 minutes)
4. Ensure that navigators remain active participants

It can be hard to introduce pair programming after students have worked individually for a while, so teachers should start with pair programming in the first few lessons.

Debugging:

Debugging is the practice of finding and fixing problems, which could be in the design or in the code. Incorporate debugging activities in student exercises.

How to use it:

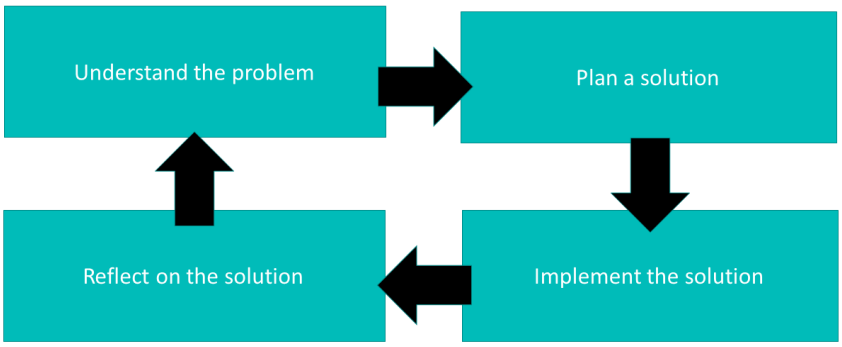
1. Emphasize debugging as a natural and expected component of creating in any CS context.
2. Celebrate discovering (and fixing) new types of bugs to normalize the debugging process.
3. Ask questions about the code and changes made to ensure that students can clearly explain how the code is intended to work.

Some techniques to debug:
- Memory diagrams
- Print statements
- Breaking into pieces

Memory diagram:

It consists of a table-like diagram in which the left column is a list of variables and the right column shows the changing contents of the variables as you trace a piece of code. If you have functions, procedures, or objects, you can stack them up in the diagram as they get called.

Problem-solving steps:



Do you understand the problem?
- Restate
- Draw a diagram
- What information do you need?

Do you have plan?
- Algorithm(s)
- Pseudo-code
- Block diagram
- Sub-problems
- Boundary cases

Solution?
- Code

- Test cases

Reflection?
- What worked and what didn't
- Alternative methods?

Socratic method:

- Answer students' questions by asking more questions.
- Unblock their thinking
- Enable only the next step and come back later to check progress
- Avoid frustrating them with too many questions

Instructional techniques:

- Lecture
- Group discussion
- Group activity
- Walkthrough
- Example
- Question
- Demo
- Debug
- Recap/Repetition

I do, we do, you do!

- Teacher shows first
- He does an activity together with students
- Students try it on their own

Beware of biases:

- "Some students are doing well, therefore I am teaching well."
- "We're so far behind, we'll just stick to slides."
- "I can't think of an activity, let's just lecture."

Using the Whiteboard

- Write clearly
- Take advantage of different locations and colors to signify things
- Set up zones: vocab (definitions), examples, code, state
- Draw pictures and diagrams, not just code
- Don't speak into the whiteboard: have students or TAs write as you talk

Asking questions:

- When you ask a question give 5 to 10 seconds for students to respond.
- Techniques:

    - Call of hands
    - Do a random poll (ask a random student but give warning first, and pause a little)
    - Ask everyone to write the answer
- Upon response, ask follow-up questions, draw others in the discussion
- No wrong answer! Student should not get a feeling that you have given up on him/her. Ask if he/she wants to use a life-line!

Co-teaching models: (when there are multiple teachers)

- One teaches, one supports
- Team teaching: teach together
- Parallel teaching: multiple student groups
- Station teaching: used during labs
- Alternative teaching

Miscellaneous:

- As a general principle, spend about 20 to 30% time talking, and leave rest of the time for actual work (programming).

- Try to match your pace with the pace of learning. Every group of students is different, so it does not make sense to have a fixed "syllabus" for every period/session.

- Try to achieve two goals: (a) Every student should get a sense of learning/achievement through his/her work; (b) Every student should feel challenged. Luckily, in programming this is possible because a "smart" student can take on a harder approach or add more features to his/her program. As a teacher, you can simply have "optional" features in every assignment which should be taken up if possible.

- Update your notes after every session and record what was covered, what issues came up, is there some new/old topic that should be addressed next time, etc.

- Have a programming assignment for every session. This means, at the end of every session, students should be able to finish some "chunk" of work. If the program is complex, tell the students what part of it they should try to finish in that session, and leave the rest for subsequent sessions.

- Know and use student names to develop a connection.

- Emphasize to students the truth (about programming) that there is no right or wrong – there can be different solutions to the same problem. Also, there is no "perfect" program; every program can be improved if there is time and interest. All the software out there has known "bugs".

- It is important to let students know that it is OK to make mistakes. The truth is, even the best programmers can never get their programs working in the first pass. In fact, it is GOOD to make mistakes, because, that is how you make new discoveries, you learn new things.

Recommended Routine for Every Session:

Generally, students are eager to get to their programs and are not in a mood to listen to you. So, you have to carefully orchestrate the available time to get what you want done.

- Begin with a few review questions. This is to reinforce the main ideas and to ensure they have understood the concepts. Refer to "what we learnt" for the project to decide which concepts to review.
- Take questions from students if any.
- If project/program is finished, have students demo if they want to.
- Go to new topics.

Before Starting:

To achieve the objective of learning, it is necessary to make sure you have:

- Students' attention: So rules of shutting monitors off and coming closer to the teacher when teacher is talking, etc.
- Visibility of your demo: Every student must be able to see your demo screen (projector or LCD) clearly.
- Activity: Ensuring that (1) There are no issues with the computers, language (e.g. Scratch) software, working environment, etc., and that (2) Students get help when they are blocked in their work (program bug, conceptual understanding etc.).

For remote classes:

Challenges of teaching remotely:
(1) You don't have the benefit of looking over the shoulders of your students. So if they are stuck or need help, you won't know unless they tell you by email/WhatsApp.
(2) You need an adult supervisor to ensure kids are following your instructions, especially when you are presenting something.

Role of the teaching assistant:

Touch every student:

It is important that every student feels (s)he is getting the attention (s)he deserves. Generally, students are of the following types. The TA must figure out what conversation to have to suit their unique situation. The conversation should focus on: (a) How the student is struggling with failure. (b) What the root cause of the frustration or fear is. (c) How the TA can help.

1. Scared Puppy - This student is completely overwhelmed by their CS class, even though they are doing quite well. They fixate on the number of compiler errors or the number of times they have to change things to get their program working. The errors or confusions they encounter are very typical, but they cannot get past the volume and sometimes can't figure out where to start.
2. Perfectionist - This student is a high achiever and is used to school coming easily to them. Their work in their CS class is, in reality, very strong. But, like most CS students, they often don't get things right the first time and sometimes have errors they don't catch, resulting in lost points. Despite carrying a solid B+ grade, they feel like a failure and are beginning to shut down.
3. Pessimist - This student has never been particularly good at math, and believes that means they are doomed to failure in CS. They're not a bad math student per se, but the subject has never come easily to them. In CS, while they do sometimes struggle with the more math-heavy assignments, they seem to have a knack for finding creative algorithms to solve problems.
4. Hustler - This student is very concerned with their grade, often at the expense of their learning. They will regularly ask "why did I lose these points"◎ or make tenuous arguments attempting to prove that their solution meets the requirements. These discussions occur on nearly every assignment, regardless of the student's original grade.
5. Opportunist - This student wants to do as little work as possible, whether due to laziness or fear of failure. They have figured out that, if they're careful, they can ask for "help"◎ often and get handheld through most of an assignment. They are very good at following instructions or implementing code they've been given, but resist continuing on their own past the last point they were helped to.
6. Show-off - This student has previous programming experience, though it may be limited. Regardless, they see themselves as an expert programmer and insist on finding a "better" way to implement every assignment, sometimes ignoring requirements to do so. As a result, they often achieve much lower grades than expected, either because they did not complete the assignment as written or the advanced technique they tried didn't work.

Grading assignments:

Use the rubric to assess and grade assignments and projects. For example, see the rubric below for the “Pong” project:

Requirement	Value	Project 1	Project 2	Project 3
Players can control paddles with required keys	2 points			

Ball begins play at middle of field at start of game and after each point	3 points			
Ball bounces correctly off upper and lower edges and paddles	4 points			
Ball increases in speed each time it bounces off a paddle	3 points			
A point is scored each time the ball touches the left or right edge	3 points			
Game ends when one player reaches five points	2 points			
Winning player is shown when game ends	1 point			
Players can begin a new game	1 point			
Functional Correctness Total	**__/19**			
Gameplay is smooth, polished, and intuitive	3 points			
Program shows creativity and effort	2 points			
Program is well-documented and exhibits good style	2 points			
Program includes at least three custom blocks, including at least one with arguments	4 points			
Custom blocks, including arguments and reporters, are used where appropriate	2 points			
Technical Correctness Total	**__/21**			
OVERALL TOTAL	**__/40**			

References:

1. Code.org AP-CSP Curriculum:

https://docs.google.com/document/d/1COu_fLJmAB4TAwvz1OC8G6mezYQ4T3Oift1pARKIs90/preview#heading=h.121itm2pggif

2. Memory diagrams:
https://www.youtube.com/watch?list=PL0g5FWk3FEqjmrq4ystAvlRyenEF7lUwa&v=hXz5Cb5m5PM

Written by: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 29 August 2020