Teaching Tech Together

Takeaways from https://teachtogether.tech*

Donny Winston

June 1, 2019

Highs and Lows (5min)

Write brief answers to the following questions.

Find a partner and share with them.

1. What is the best class or workshop you ever took? What made it so good?

2. What was the worst one? What made it so bad?

TLDR; Tips for teaching

- Remember that there is no geek gene
  - Remember that novices are not experts
  - Have learners work with each other: peer instruction; pair programming
  - Use authentic tasks
  - Use a visibly fair mechanism to distribute your attention evenly
- Build things in front of them (and have them build along)
  - Live coding
  - Make mistakes and work through them
  - If you use slides, fill them with graphics rather than text
  - Get learners out of their seats every 45-60 minutes; create a backlog
- Have learners do something interactive every 5-15 minutes
  - Have them make predictions
  - Go back right away to the person who didn't know the answer
  - Use worked examples with labeled subgoals
  - Don't just code

Brown, Neil C. C., and Greg Wilson. "Ten Quick Tips for Teaching Programming." *PLoS Computational Biology* 14, no. 4 (April 2018)

http://third-bit.com/2019/05/04/in-the-classroom.html

Know Thyself (10min)

Write brief answers to the following questions.

Share with your partner.

1. How will you know if you're teaching well?

2. What is one specific thing you believe is true about teaching and learning?

Cognitive development in a problem domain

- *Novices*
  - don’t know what they don’t know
  - lacking usable mental model of the problem domain
- *Competent practitioners*
  - mental model adequate for everyday purposes
  - can do normal tasks with normal effort under normal circumstances
  - some understanding of limits to their knowledge -- knowing what they don’t know
- *Experts*
  - mental models include exceptions and special cases
  - allows allows them to handle situations that are out of the ordinary

Benner, Patricia. *From Novice to Expert: Excellence and Power in Clinical Nursing Practice*. Pearson, 2000.

A different progression (15min)

The novice-competent-expert model of skill development is sometimes called the Dreyfus model. Another commonly-used progression is the so-called four stages of competence:

- Unconscious incompetence: the person doesn't know what they don't know.
- Conscious incompetence: the person realizes that they don't know something.
- Conscious competence: the person has learned how to do something, but can only do it while concentrating and may still need to break things down into steps.
- Unconscious competence: the skill has become second nature and the person can do it reflexively.

Identify one subject where you are at each level. What level do you imagine most of your learners at in the subject you teach most often? What level are you trying to get them to? How do these four stages relate to the novice-competent-expert classification?

A mental model: somewhere to put facts

- simplified representation of the most important parts
  - of some problem domain
  - that is good enough to enable problem solving
  - example: balls-and-springs model of molecules
  - more sophisticated example: balls-with-orbiting-electrons model
  - extra complexity enables people to explain more and solve more problems
  - like software, mental models are never *finished*; they're just used
- presenting a novice with a pile of facts is counter-productive
  - they don't yet have a model to fit those facts
  - can reinforce an incorrect mental model they cobbled together
  - example: specific Unix shell commands don't make sense until novices understand concepts like: *paths*, *tab completion*, *wildcards*, *pipes*, *redirection*, etc.
- novice: competent practitioner :: tutorial: manual
  - *tutorials* help build mental models; *manuals* help fill in gaps in knowledge
  - tutorials frustrate competent practitioners; manuals frustrate novices – *who is your lesson for?*

Your mental models (15min)

- ***think*** – What is one mental model you use to understand your work? Write a few sentences describing it.

- ***pair*** – Give feedback on a partner's.

- ***share*** – Once you have done that, (a few people) share your models with the whole group.

Bio break! (5min)

Address backlog

Are people learning? Formative assessment

"It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so." – Mark Twain

Novices' misconceptions:

- *factual errors* – usually simple to correct
  - Example: believing that Vancouver is the capital of British Columbia (it's Victoria)
- *broken models* – have novices reason through examples where model gives wrong answer
  - Example: believing that motion and acceleration must be in the same direction
- *fundamental beliefs* – learners resist evidence and rationalize contradictions
  - Example: "some kinds of people are just naturally better at programming than others"

***Formative* assessment** is identifying and clearing misconceptions as a lesson is delivered

Symptoms of being a novice (5min)

- Saying that novices don’t have a mental model of a particular domain is not the same as saying that they don’t have a mental model at all.
- Novices tend to reason by analogy and guesswork, borrowing bits and pieces of mental models from other domains that seem superficially similar.
- People who are doing this often say things that are “not even wrong”.

Let’s discuss what some other symptoms of being a novice are.

What does someone do or say that leads you to classify them as a novice in some domain?

Elements of formative assessment exercises

- ***Formative* assessment** is identifying and clearing misconceptions as a lesson is delivered
- Get feedback or don't – there is no "pass" or "fail". Feedback is on
  - how well people are doing , and
  - what they should focus on next
- Quick to administer
- An unambiguous correct answer (so that it can be used with groups)
- Incorrect answers are *plausible distractors* with *diagnostic power*
  - ask yourself: "Where do wrong answers come from?"
  - makes your lessons better because it forces you to think about learners' mental models
  - …so these exercises pay off even if you don't actually use them
- Contrast with ***summative assessment***:
  - tells the learner whether they have mastered the topic
  - tells the teacher whether their lesson was successful
  - a chef tasting food as she cooks? Formative assessment
  - the guests tasting it once it's served? Summative assessment

Example: teaching multi-digit addition

What is 37 + 15?

a) 52
b) 42
c) 412
d) 43

Correct answer is 52, but the other answers provide valuable insights:

- 42? no understanding of what “carrying” means. (e.g. 12 as the answer to 7+5, then overwrite the 1 with the 4 from 3+1.)
- 412? treating each column of numbers as a separate problem. Still wrong, but for a different reason.
- 43? knows to carry the 1, but carrying back into column it came from. Again, different mistake -- requires different clarifying explanation.

Some classic exercise types

- Multiple Choice

In what order do operations occur when the computer evaluates the expression

```
price = addTaxes(cost - discount)?
```

1. subtraction, function call, assignment
2. function call, subtraction, assignment
3. function call, then assignment and subtraction simultaneously
4. none of the above

wrong answers probe for specific misconceptions

- Fill in the blanks

Fill in the blanks so that the code below prints the string 'hat'.

```
text = 'all that it is'
slice = text[____:____]
print(slice)
```

Strategy: ***faded examples*** – a series of examples in which a steadily increasing number of key steps are blanked out.

- Parsons problem

Rearrange and indent these lines to sum the positive values in a list. (You will need to add colons in appropriate places as well.)

```
total = 0
if v > 0
total += v
for v in values
```

Also avoids "blank screen of terror". Concentrates on control flow separately from vocabulary.

Modeling novice mental models (20min)

Create a multiple choice question (MCQ) related to a topic you have taught or may teach and explain the diagnostic power of each its distractors (i.e., what misconception each distractor is meant to identify).

When you are done, trade MCQs with a partner. Is their question ambiguous? Are the misconceptions plausible? Do the distractors actually test for them? Are any likely misconceptions *not* tested for?

Active learning? Active teaching

passive teachers:

- don’t modify what they’re saying or where they’re going

active teachers:

- try out “what if?” scenarios proposed by learners
- modifying an example on the fly based on learners’ interests or puzzled expressions
- use what they learn from and about students as they learn

Passive teaching	Active teaching
train on tracks	four-wheel drive
classical music	jazz
slides	live coding

- idea: use a sequence of formative assessments like a series of chord changes to improvise over rather than having what most people would call a lesson plan
- …but okay if you prefer a fully worked out lesson plan that you can play like an orchestral score.

Notional machines

- A *notional machine* is a general, simplified model of how a particular family of programs executes
  - idealized abstraction of hardware and other aspects of runtime environments
  - enables semantics of programs to be described
  - correctly reflects what programs do when executed
  - a “cartoon version of reality”
- Plausible goal is for learners to have a mental model that includes most or all features of your notional machine.
  - after sufficient instruction
  - …and many more hours of work on their own time

Boulay, “Some Difficulties of Learning to Program.” *Journal of Educational Computing Research* 2, 1 (1986)
Sorva et al., “Notional Machines and Introductory Programming Education.” *ACM Transactions on Computing Education* 13, 2 (2013)

Example notional machine for Python

- Running programs live in memory, divided between a call stack and a dynamically allocated heap.
- Memory for data is always allocated from the heap.
- Every piece of data is stored in a two-part structure. The first part says what type the data is, and the second part is the actual value.
- Booleans, numbers, and character strings are never modified after they are created.
- Lists, sets, and other collections store references to other data rather than storing those values directly. They can be modified after they are created, i.e., a list can be extended or new values can be added to a set.
- When code is loaded into memory, Python converts it to a sequence of instructions that are stored like any other data. This is why it's possible to assign functions to variables and pass them as parameters.
- When code is executed, Python steps through the instructions, doing what each one tells it to in turn.
- Some instructions make Python read data, do calculations, and create new data. Other instructions control what instructions Python executes, which is how loops and conditionals work. Yet another instruction tells Python to call a function.
- When a function is called, Python pushes a new stack frame onto the call stack.
- Each stack frame stores variables' names and references to data. Function parameters are just another kind of variable.
- When a variable is used, Python looks for it in the top stack frame. If it isn't there, it looks in the bottom (global) frame.
- When the function finishes, Python erases its stack frame and jumps backs to the instructions it was executing before the function call. If there isn't a "before, the program has finished.

Your notional machines (20min)

Working in small groups, write up a description of the notional machine you might want learners to use to understand how some (computational) process runs.

- A spreadsheet taking input and updating?
- A web browser rendering a page?
- A program requesting data from an API?
- A Fireworks workflow being queued and run?
- ...pick your own! Need not be computational, but may reflect so-called *computational thinking*

Tips for teaching

- Remember that there is no geek gene
  - Remember that novices are not experts
  - Have learners work with each other: peer instruction; pair programming
  - Use authentic tasks
  - Use a visibly fair mechanism to distribute your attention evenly
- Build things in front of them (and have them build along)
  - Live coding
  - Make mistakes and work through them
  - If you use slides, fill them with graphics rather than text
  - Get learners out of their seats every 45-60 minutes; create a backlog
- Have learners do something interactive every 5-15 minutes
  - Have them make predictions
  - Go back right away to the person who didn't know the answer
  - Use worked examples with labeled subgoals
  - Don't just code

Brown, Neil C. C., and Greg Wilson. "Ten Quick Tips for Teaching Programming." *PLoS Computational Biology* 14, no. 4 (April 2018)

http://third-bit.com/2019/05/04/in-the-classroom.html