CS — Python — Instructor Notes

Spring 2017

**COURSE PLANNING**

**Computer Science Python Course Planning:**

* Obtain computer science notes/archives from the previous time the course was taught.
* Update the notes/archives each term the course is taught.
* Notify the school of any updates to where these notes/archives are kept.

**Learning Objectives to Consider:**

1. College Board AP Computer Science A (I think of this as "classical computer science".)

<https://apstudent.collegeboard.org/apcourse/ap-computer-science-a>

1. College Board AP Computer Science Principles (I think of this as "digital citizenship"---not classical CS.)

<https://apstudent.collegeboard.org/apcourse/ap-computer-science-principles>

1. Washington State Computer Science K-12 Learning Standards

<http://www.k12.wa.us/ComputerScience/LearningStandards.aspx>

1. Instructor-specific learning objectives

**Note:** The process and resources for the teaching of AP courses can be found at <http://apcentral.collegeboard.com>.

* Steps for the school: <http://apcentral.collegeboard.com/apc/public/start_grow_ap/start_ap/232581.html>
* Educational resources: <http://apcentral.collegeboard.com/apc/public/professional_development/index.html>
* This includes summer courses, which might not be required, but are a good idea.
* This also include forums allowing discussion with other AP instructors.

**COURSE STRUCTURE**

In Spring 2017, the sophomore Python class was divided into 4 "units", of roughly the same length:

1. The basics of the Python programming language
   * Types, values, variables, expressions, statements, lists, dictionaries & Counters, control flow (if/elif/else, while, for, continue, break), list & dictionary comprehensions.
2. Software engineering (also, brainstorming possible Course Projects). Material includes:
   * Docstrings, map and filter, classes (w/ both instance and class-level data/functions)
   * Caching via the use of a class-level dict (even though in practice this would be done via decorators)
   * A quick tour of logging, testing, and debugging, but not in-depth enough to be tested
3. Graphics (also, writing a Project Plan for a Course Project).
   * Turtle graphics (on Quiz #3), as well as tkinter and matplotlib (not tested)
4. Writing of the Course Project.
   * They could work on a project individually, or in teams. For teams, I encouraged an explicit division of responsibilities, though it's easier to have the distribution of work be fluid. Even if there is a significant amount of work besides writing code (e.g., storyboarding), everyone should be writing some code. The instructor should keep an eye on the balance of work among teams, especially teams of more than two.
   * Each team had a Project Plan, and Project Updates (due on Thursdays) were tracked against the plan.
   * This is as much about successfully managing a software project as it is about writing the code. That includes thinking of risks ahead of time and tracking them. This goes toward teaching them that they are responsible for the overall success of their project. Lack of awareness or understanding should not be viewed as an excuse, but rather as a weakness that needs to be remedied.
   * Presentations to an audience were done on the last day. There was no final exam.

**COURSE CONTENT**

**Reminder:** This is a high school course. Pedagogy trumps conventional software design & implementation conventions.

**Course Material Choices:**

Teaching a five-month high school introduction to Python requires many choices of what to teach and what not to.

|  |  |
| --- | --- |
| Taught in Spring, 2017 | Omitted from Spring, 2017 |
| Unit #1 |  |
| string's format function | old-style formatting (%), string.Template, Literal (from 3.6) |
| assert(False) | raising exceptions |
| function args (mainly positional) | Keyword args, and unpacking function args with \*args and \*kwargs |
|  | Scoped resources in "with" statements (PEP 343) |
|  | Async |
| Loops | Iterators & Generators; Monte Carlo; iterative approximations |
| Reading files with open(path).readlines()  used for examples, but not on any test. | File & networking IO |
| Unit #2 |  |
| Docstrings (covered, but not tested) | Docstring tags |
| Lambdas | Not including closures |
| map & filter (exposure to HoF) | reduce (payoff not worth the extra topic) |
| classes | class inheritance, metaclasses, coding standards, design principles |
| keyword arguments, default arg values |  |
| caching using a class-level dict | caching using decorators (didn't mention decorators) |
| Logging, testing, debugging (all covered briefly, but not on any test) | CS: Trees, graphs, sorting, searching, complexity, etc. |
| Unit #3 |  |
| Turtle graphics |  |
| tkinter, matplotlib, Jupyter (all covered briefly, but not on any test) | Writing multi-threaded code. |

**Overall Content Notes:**

1. Use a fixed-width font (Lucida Console) for code in slides and homework, so that spacing can be seen more easily.
   1. For example, students might not notice the space after "if" in

if \_\_name\_\_ == '\_\_main\_\_':

1. **Run local Python only.**  There are online Python interpreters available, but I would discourage their use, at least until after the students have become comfortable running Python locally. Some online interpreters don’t set \_\_name\_\_ properly. Also, handling multiple files and creating local windows in online interpreters requires effort that could better be spent learning Python itself, and standard Python tools.
2. The terms "instance-level variable" or "class-level variable" promote clarity. Though the term "class variable" might be common among software developers for static variables, this could lead to confusion among students, given that instance variables are also defined within a class.
3. Adding more detail to a topic doesn't always result in more understanding. For example, detailing the entire set of print conversion specifiers available would be overkill. Only a few are needed.
4. Just as a Language Arts teacher might provide both stylistic suggestions as well as corrections to a paper, feedback on software can range from minor coding conventions (e.g., spacing around operators) to grievous blunders that should be corrected (e.g., things that make the code prone to error, or things that are likely to confuse or annoy the user). Although what constitutes a grievous blunder could be described as a matter of opinion, such opinions are usually based on years of experience and a grasp of the expectations of the user, which high school students likely have not yet acquired. This should be made clear to the students near the beginning of the course, and suggested changes should be accompanied with a reason, so that they don't see these changes as arbitrary. (They might also not care about subject-verb agreement in Language Arts papers, but hopefully they soon come around.)

**Unit #1 Notes:**

1. Early homework tasks:
   1. Include a task to send you email with an introduction.
   2. Download PDF of course text (I used [ThinkPython, 2nd ed.](http://greenteapress.com/wp/think-python-2e/), by Allen B. Downey, pub by GreenTeaPress.com)
   3. Install Python (Anaconda?) and verify that it's the right version of Python with python --v.
   4. Install an IDE (PyCharm?)
      * **Note:** MacBooks come with Python 2 pre-installed. **Once Anaconda's Python 3 is installed**, Terminal will use that version. To configure PyCharm on a MacBook to use it, do the following.
        1. In a terminal window, type "which python" to see the path used. Then run python --version to confirm that the instance of Python at that path uses version 3 (e.g., v 3.5.2).
        2. Use File 🡪 Settings 🡪 Project: untitled 🡪 Project Interpreter. In the right-hand pane, click on the Configuration icon in the upper-right corner, then select Add Local.
        3. In the Select Python Interpreter window that appears, enter the path seen in (a).
   5. Run a list of valid IPython commands to passively become familiar with the syntax
   6. Run a list of invalid IPython commands to learn to associate error messages with their cause.
      * I had them send me a description of some error messages and explain what each one meant.
      * This gave me a sense of how articulate and detailed their observations and conclusions were.
      * Be careful about explicitly requiring them to describe the error messages "in their own words". If they then copy the descriptions of others, you've got to decide how to handle the infraction.
2. Some basic computer-related tasks the students might have difficulty with:
3. Creating a text file, and changing the suffix of a text file (e.g., from .txt to .py)
4. Saving a file to disk, and saving in a given file format. (The OS X ".pages" file format is not good for sharing.)
5. Save the output of a program and sending it as an attachment by email
6. Locating on disk a file that's being edited (e.g., in PyCharm).
7. Unzipping a zip file.
8. Control flow (if/elif/else, while, for, break, continue) takes a long time to master, and requires many examples.
   1. Creating auxiliary variables (e.g., to track the number of items in a list) takes lots of practice.
   2. Walking them through a taxonomy of auxiliary variables used to track state might or might not be helpful.
      * A Boolean variable, used to track whether or not some condition has been met (e.g., contains).
      * An integer, used to track how many times a condition is met (e.g., word\_count).
      * An integer, used to track the ordinality of the items (e.g., print out file lines w/ line numbers).
      * A running total (e.g., sum, product, int2str, str2int).
   3. Walking them through the "aha" moment when they find the right auxiliary variables might be more helpful.
9. It takes effort to get an accurate sense of how familiar students are with the material. Some challenges:

* It's not easy for them to reflect on or describe their state of understanding, or to formulate questions.
* Answering individual student questions doesn't give the instructor a map of the larger terrain of difficulty.
* There is a lot of collaboration on the homework, which can obscure individual difficulties.

**Unit #2 Notes:**

1. At the end of Unit #1 (introduction to basic Python language features), the students have seen the basic language features of Python, buy they are not yet ready to work on a Course Project. For example, working out the definition for a short (say, 5-10 line) function could still take them almost an hour to accomplish.
2. The first quiz will expose knowledge gaps that the students have kept quiet about.
3. Through the Unit #2 HW problems, the students were exposed to longer programs, including simple "games".
4. The choice of programs used in HW assignments was influenced by their ideas for Course Projects. For example, there was significant interest in adventure-type games, so one problem involved a skeletal adventure program.
5. As we get into classes that model real-world entities (e.g., BankAccount), some students wonder, even after they're introduced to OO modeling and walked through the definition of the class, how the computer "understands" what the data and functions in the class "mean". For example, how does the computer knows what "balance", "deposit", and "withdraw" mean? We hope to replace this anthropomorphic world view with a deeper understanding of how software works, and an awareness that the computer's "understanding" comes entirely from the accuracy with which classes are designed to model real-world entities. This reminds me of an old quote from Charles Babbage:

"On two occasions I have been asked, — 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' […] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

**Unit #3 Notes:**

1. turtle graphics allows a visual representation of some concepts seen earlier:

* iteration ↔ figures with repetitive structure (e.g., polygons, chess boards), and recursion ↔ fractals.

1. Exposing the students briefly to tkinter, matplotlib, and Jupyter gives them tools that they might find decide to use in their Course Projects. tkinter offers three Layout Managers available: pack, grid, and place. If they use tkinter for a project, it might be best if they only used on of the three layout managers. I recommend grid.
2. If you include a data visualization package, consider pandas instead of matplotlib.

**Unit #4 Notes:**

1. The decision to include Course Projects requires that the students learn enough Python to be able to conceive and write the software necessary to complete the project. A few alternatives are available:

* No Course Project: This alleviates concern over covering enough material for a Course Project to succeed.
* Have a single Course Project that all students work on in parallel (e.g., Hunt the Wumpus)
  + Each student could come up with their own variations. This streamlines the instruction required.
* Students/teams work on their own self-directed projects: More independence & more individualized instruction.
  + This is the path we followed.
* One extreme option taken in some introductory courses is for the entire course to revolve around a project.
  + For example, see the story of RMU Monopoly: <http://www.iiisci.org/journal/CV$/sci/pdfs/ZE817AN.pdf>

1. Carrying out Course Projects is partially about learning to manage a project from start to finish. Give the students some guidelines, and a template Project Plan to fill out, but also give them some leeway in how to proceed. Enough leeway that they might stumble, and perhaps learn useful lessons from such minor missteps.
2. Some things the students have learned that might be particularly difficult for them to apply to Course Projects:

* Choosing meaningful variable names.
* Recognizing repetitive code as such, and replacing it with functions that encapsulate the right abstractions.
* (and searching online to find answers to questions)

1. If there is a presentation of their Course Projects to an audience, a rehearsal presentation is needed, and it should be clear beforehand what is expected (e.g., slides, description of the components of the software, length of presentation, etc.). Giving them free reign on how to present their projects would not be as effective.
2. I considered having a Coding Conventions document that would be jointly owned by all students in the course, just as some companies maintain one. They could add rules to it when they found a bug that could have been avoided by following some rule. I decided this would be too much of a distraction from learning the course material itself.