Course Information

I. Overview

Computer Science E-7 is a broad introduction to some of the most important concepts in the field of computer science. To paraphrase Alan Biermann, this course is for people who sometimes wonder

> *" … what is meant by a Python module, a file server, ethernet, TCP/IP, or* ***P****retty* ***G****ood* ***P****rivacy … You have been told you need a compiler, but* *why* *do you need it? What does it actually do? Watch out for certain types of problems— they are 'NP-complete,' and you may not be able to compute the answer. You want to have a neat graphics picture jump onto your Web page when you push the button labeled 'surprise.' Can you make it happen? Your laptop runs at 2.5 gigahertz: that is two and one-half billion* *what* *per second? (And, by the way, what exactly* *is* *a computer?)"*[1]

This course covers a superset of the content in Computer Science E-10a, albeit in a different programming language. The lecture videos are being recorded from Dr. Leitner's spring semester Harvard College course, Computer Science 1: *Great Ideas in Computer Science with Python*. This course will expose students to

❶ The principles and practices of *functional* and *object-oriented programming* (OOP) using a methodology that places a high value on programs that not only generate the "right answers," but that are also easy to read, maintain, and modify. These elements are important in any programming, regardless of the specific language used.

[1] Excerpted from page xiii (the Preface) in *Great Ideas in Computer Science with Python*, by Alan W. Biermann and Dietolf Ramm. Published by the MIT Press. ISBN: 9780262024976

Much of the programming will be done using *Python*, through which we will investigate fundamental *data structures* and their *algorithms*.

Preliminary programming exercises will utilize HTML and the *Scratch* environment, free software that runs on both Mac OS and Windows OS platforms.

❷ The mathematical, statistical, and computational methods that will enable you to think critically about data as it is employed in fields of inquiry across the Faculty of Arts and Sciences, thereby meeting the *Quantitative Reasoning with Data Requirement* (QRD). You will be able to analyze real data, draw inferences and make predictions to answer questions – as well as understand the limitations of these methods. Results can be summarized and communicated through visualization.

❸ The landscape of computer science as it exists today, with some reference to its past and future, as this will enable us to touch on a variety of really fascinating topics and intellectual paradigms, i.e., some of the "Great Ideas in Computer Science," such as [2]

- **Simulation:** Suppose you wish to observe something, but you cannot because it's too expensive or too slow or just impossible to observe. You might be able to get your wish if you can successfully *simulate* that "thing" (e.g., a physical process or experiment). This means you must discover a model for that thing, program it, run the program and do your observations. If your model is good, you will be able to see what would have happened with the original thing, had you been able to observe it.

- **Ethics:** "Technology is not neutral," says Professor Nehran Sahami, a faculty member at Stanford who formerly worked at Google as a senior research scientist. "The choices that get made in building technology then have social ramifications." Consider the algorithms that get implemented in autonomous vehicles and weapon systems, or the fact that social media sites such as Twitter can be responsible for the spreading of "fake news."

- **Computer Architecture:** When you purchased your last laptop computer, what did you actually get? In fact, you received hardware that executes machine language instructions in the *fetch-execute cycle* at a very high speed, some memory, a hard drive, and some input-output devices (like a keyboard and display). You will learn what machine instructions are, how the fetch-execute cycle works, and how memory is used to enable the machine to do its job. You will learn the operation and mechanisms of the bare-bones computer and we'll demystify a lot

[2] A few of these summaries are paraphrased from Biermann and Ramm's descriptions.

of computer jargon that you might not otherwise understand.

- **Security and Privacy**: While computers and networking are critically important in our lives, they also can become vehicles of mischief. What if our personal secrets, our medical information, and/or our financial records are stolen and sold online?[3] What if we become dependent on machines and suddenly they cease to work because of an electronic attack? We will explain the kinds of attacks that might be brought against individuals or organizations. And we'll describe some of the defenses that one can use, including various encryption methods.

- **Computer Communications**: You will learn what a Local Area Network (LAN) is, and how *packets* get shipped around. You will learn also about Wide Area Networks, especially the Internet, what they are and how they work. You will learn some networking terminology (UDP, TCP/IP) and you will learn a bit about addressing schemes, network servers, and what the "Internet of Things" is all about.

- **Program Execution Time**: We will discuss a limitation of computer science, describing a major hitch that prevents scientists from solving some important problems. It turns out that it could take a billion years of computation to solve certain problems, unless quantum computing comes to the rescue . Even if conventional machines get much faster, these problems will continue to be out of reach. We will introduce the ideas of *tractability*, referring to problems that usually can be solved in practical situations, and *intractability*, referring to problems that tend to require too much execution time to solve. We'll provide examples of both kinds of problems so that you may gain some intuition for these phenomena.

- **Noncomputability:** There is a class of very strange problems that mathematicians have proven can *never* be solved by any computer within known computational paradigms. This mystical and elusive class of programs seems to place an impenetrable blockage to progress in certain aspects of computer science.

❹ One or more of these additional topics, as time allows, such as:

- **Parallel Computation:** If a problem requires too much time to solve with one computer, perhaps we could spread it across many machines, thousands or millions of them, and then solve the very time-consuming calculation. We investigate this idea with both positive and

[3] See, for example: http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/

negative results. Though many computations can be speeded up tremendously by putting them on parallel machines, one cannot always do this easily and sometimes the speedup is not enough.

- **Computer Graphics**: How do computers render and manipulate photographs, drawings and movies — some of which may involve pictures of things that do not exist (or perhaps could never exist)? Why is human face recognition a difficult computational problem for computers (not for humans), and what sort of 3D modeling techniques are being developed for face modeling and animation?

- **Artificial Intelligence:** Theories of machine intelligence have evolved over several decades; we can examine the idea of representing knowledge and what it means to use that knowledge to "understand," as well as the possibility of automating the learning process. We can also study problem-solving and how knowledge contributes to the ability to solve problems. In addition, we can examine how to use this theory to build machines that seem to be intelligent. Some examples are a speech-recognition system, a game-playing program that improves performance from experience, and a so-called expert system.

- **Low-level Machine Architecture:** Out of very simple logic gates, such as the AND, OR and NOT, we can easily create circuits that compute primitive and complex functions. It's also interesting to understand how such circuits are embodied by transistors and very large scale integration (VLSI) chips.

- **Language Translation**: But how can we type Python programs or programs in other languages into our computer and expect them to work, given that the basic computer hardware can only understand very primitive instructions? The solution is to build a *translator* that will transform the language that we prefer into a form that the machine can process. You will learn what people mean when they say they "compiled" a program, and you will do some compiling yourself.

⌛ Although we believe the content of CSCI E-7 is fairly straightforward, this is one of those courses where the homework can be time-consuming. It is not unusual for students to spend as many 12-15 (or more) hours per week doing the problem sets. If you have other major time commitments (e.g., a part-time job, other courses, a family, friends, hobbies, etc.), then you may wish to reconsider whether or not to take this course. You have been warned!

II. Staff

Faculty:	Dr. Henry Leitner 51 Brattle St., rm. W-719 (617) 495-9096 email: leitner@harvard.edu

Dr. Leitner will generally be available for consultation[4] on Wednesday mornings. Because of the uncertainties due to the on-going Covid pandemic, it's probably best to arrange an appointment via email.

Teaching Assistants:	Mr. Benjamin Basseri *(head TA):* basseri@cs50.harvard.edu
	Mr. Nabib Ahmed: nahmed@college.harvard.edu
	Mr. Thomas Ballatore: ballatore@cs50.harvard.edu
	Ms. Victoria Gong, victoriagong@college.harvard.edu
	Ms. Apekshya Panda, apekshya@mit.edu
	Ms. Lara Zeng: lzeng@college.harvard.edu
	others, if enrollment warrants

The TAs are responsible for grading problem sets and for helping students, in general, with the material covered in this course. The head TA will assist with a number of administrative matters, such as maintaining the course website.

Each student is expected to attend a semi-mandatory 60-90 minute section meeting every week; these section meetings are run entirely by the teaching assistants, and will probably be held online only via Zoom due to the pandemic. The precise day and times will be announced later on. **Section meetings will begin the week of January 30.**

Up-to-date office hours for the various teaching assistants will be posted in a Google calendar on our course website.

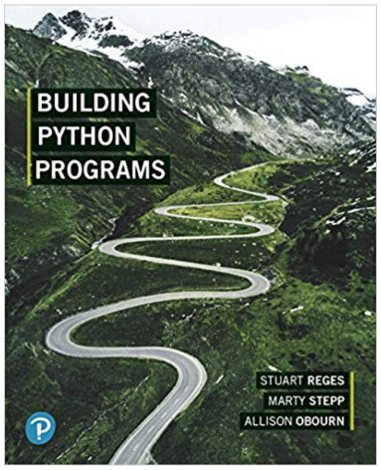
[4] … and occasional "consolation"

III. What to Read

You should acquire the habit of consulting our course website often (perhaps once every couple of days). The URL is:

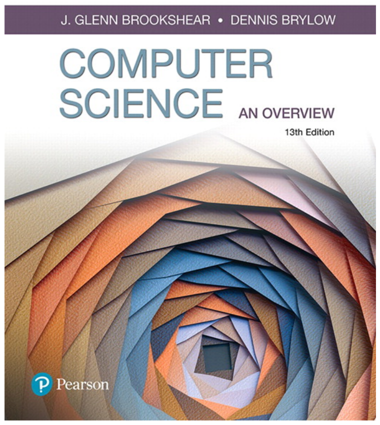
https://canvas.harvard.edu/courses/96344

Optional reading materials for this course are for sale at the Harvard COOP bookstore and available through Amazon or other online sellers:



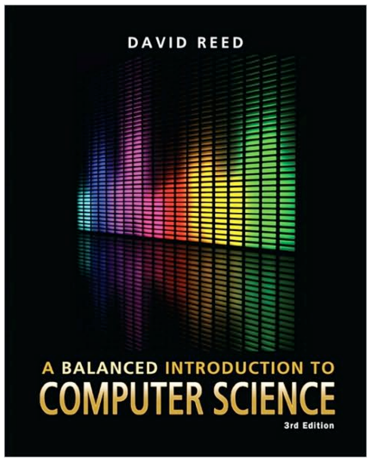
***Building Python Programs (A Back to Basics Approach) 1st edition*, published by Addison-Wesley, 2020. ISBN #978-0135205983.** Although this book is not required for purchase, it is highly recommended.

A less costly text is *Think Python (2nd edition)*, by Allen B. Downey. Published by O'Reilly Press, 2016. ISBN #978-1491939369


J. GLENN BROOKSHEAR • DENNIS BRYLOW
COMPUTER SCIENCE AN OVERVIEW
13th Edition
Pearson

Although we do <u>not</u> suggest that you acquire any of the following books at this time, some of these texts shown below may be of interest to those of you who wish to explore specific topics in more depth than we will have time for in class.

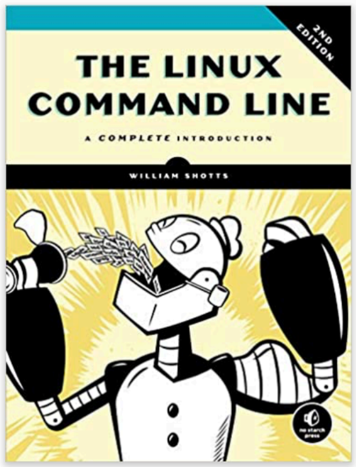
Computer Science, An Overview (13th edition), by J. Glenn Brookshear and Dennis Brylow. Published by Pearson, 2019. The ISBN # is 978-0134875460. This book offers a clear and concise survey of computer science, covering a wealth of topics and presenting the scope of the discipline as well as the terminology in the field.



A Balanced Introduction to Computer Science (3rd edition), by David Reed. Published by Prentice-Hall, 2010. The ISBN # is 9780132166751.

LEARN TO PROGRAM WITH SCRATCH
A VISUAL INTRODUCTION TO PROGRAMMING WITH GAMES, ART, SCIENCE, AND MATH
MAJED MARJI
COVERS SCRATCH 2

Learn to Program with Scratch: A Visual Introduction to Programming with Games, Art, Science, and Math, by Majed Marji. Published by No Starch Press, San Francisco, 2014. The ISBN # is 9781593275433.



The Linux Command Line: A Complete Introduction (2nd edition), by William Shotts. Published by No Starch Press, San Francisco, 2019. The ISBN # is 978-1593279523. Since you will be writing your Python programs in a Linux environment, you may find it helpful to learn a bit more about this important operating system. A free PDF version is available at https://linuxcommand.org/tlcl.php

IV. The Problem Sets, the Term Project, and Grading

The majority of the assignments will involve problem solving using a Mac OS or Windows-based personal computer. Some of the homework exercises will be of the short "paper-and-pencil" variety. You will be required to do "electronic submission," a process we will describe in class.

Please do not attempt to finish up an assignment during lecture or during section! *Problem sets will be due, in general, prior to 5:00 PM on Fridays.*

Each student is allotted a budget of 5 no-penalty "late days" for use throughout the term. Additional late days may be "purchased" for a 6-point penalty each. Extra *no-penalty* late days may ordinarily be obtained only by submitting a doctor's note to Dr. Leitner. Exceptions to these rules may be sought for unusual circumstances by petition. Do not fritter away the no-penalty late days early in the semester; you may find yourself wishing you had saved a few toward the end of the semester. **If electronic submission of your work is more than 10 minutes late arriving, then it will be considered a full day late.** At section meetings and during regular office hours, your assigned TA will return your graded homework to you. As I usually plead, please, *please*

Don't fall behind on your problem sets!

Just as you cannot expect to learn how to drive a car by reading about it or by watching other people do it, the same holds true for working with computer hardware and software.

Get started on the problem sets early — this is one course you simply cannot "cram" for at the last minute, so don't even try! We cannot stress this strongly enough. Remember that some of you may find the problem sets to be time-consuming on occasion, so please take your other commitments into consideration.

Final grades for students enrolled in this course will be based on

➢ the homework	(60%),
➢ one open-book exam	(20%),
➢ a final term project	(20%),
➢ ... and your teaching assistants' appraisal of individual achievement	(*priceless*).

Undergraduate-credit students will have two distinct alternatives for the term project, described below. Graduate-credit students must complete the second option (a Python programming project).

⇒ An original paper that is presented in the form of a **website**, covering in depth, a "Great Idea" in computer science. The subject matter does not have to be one of the topics we have explicitly touched on in class this year. A colleague of mine at Stanford named Eric Roberts has been teaching a course named "The Intellectual Excitement of Computer Science" for several years; on our course website I will post links to some of the better projects completed by both Harvard students as well as students in Prof. Roberts' course over the past few years. If you are interested in this option, your project need not have animations or flashy graphics, but we DO expect the website to be well-organized and to present your "paper" in a form that's easy to navigate.

⇒ The alternative option is to write an original program in *Python* that will run on the **ide.cs50.io** platform using ***Python***.[5] We will post some specific

[5] You may also use an integrated development environment (IDE) for Python on your personal computer, such as *Spyder* or *PyCharm*.

examples of Python-based projects on our website later on.

A document describing the term project in more detail (along with specific suggestions) will be handed out later in the semester.

V. A Commentary On Programming Ethics[6]

```
A computer program written to satisfy a course
requirement is, like a paper, expected to be the
original work of the student submitting it.  Copying
a program from another student or from any other
source is a form of academic dishonesty, as is
deriving a program substantially from the work of
another.
```

Persons who do not know how to program a computer are understandably puzzled about how the concept of plagiarism could possibly be applicable in a computer course. Is a program not an exact object, like a number, and must not any two correct solutions to a programming problem be identical? The truth is quite different. Superficial appearances aside, computer programs more closely resemble essays than numbers. The copyright laws recognize this; so do standards of behavior at Harvard.

Two programmers may adopt radically different approaches to solving the same problem, as different as the ideas of two students asked to write critically on the same painting. Very small programs do not admit this much variability in overall design, but anything over a page long certainly does, unless the design itself was specified as part of the exercise. But even when two programs are based on the same overall design, the variation in possible form of expression is vast.

Programming courses attempt to teach graceful and forceful forms of expressions for computational ideas, but every programmer has idiosyncrasies of style and vocabulary. The likelihood of two programmers independently creating identical programs of more than two or three lines in length is no larger than the likelihood of two writers independently writing identical paragraphs. And programs that are identical except for their choice of names (for example, one has "x" everywhere the other has "y") are as improbable as two short stories that are identical except for the names of the characters. Paraphrase is as possible, and as dishonest, with programs as with papers; two programs can be copies of each other even though no single line of one is identical to any line of the other. There should be little confusion about what is legitimate and what is not in the production of a computer program; the rule is simple,

[6] The principal author of this section is Prof. Harry R. Lewis, Harvard College Professor of Computer Science at Harvard.

simpler than in expository writing, since programming generally does not involve library research and use of sources: **Do not submit as your own work a program based on the work of another! Violations of this rule is *plagiarism*; it is dishonorable behavior, and the penalty for it is requirement to withdraw from Harvard College.**

Two obvious "exceptions" to this rule may be noted in passing. Courses sometimes supply the main idea or even some of the text of a program that is to be completed as an exercise; naturally, students are expected to use this assistance. And there is merit in "copying from oneself" in a course that develops cumulative programming skills. Here programs differ from papers; no author would want to write two different pieces with several paragraphs in common, but with computer programs, this is not unusual. A skill taught in programming courses is how NOT to reinvent the wheel; when a small phrase or short sentence has proven useful and reliable in one program, a programmer should feel free to reuse it if the same thing needs to be said in another program. Such clauses play the role of aphorisms; they make a point but they are not the main point of the piece being written.

Of course, neither of these examples obscures the basic point that a program submitted as original work should not have been derived from the work of another unless the course has specifically permitted this.

How much help on a programming exercise may you obtain before you are stealing, rather than being assisted? Teaching assistants and user assistants know the limitations of what is fair and legitimate; their goal is to help you understand how to solve your own problem, not to solve it for you. If you seek help from other students you are treading on much thinner ice. When a student answers a simple factual question which could have been answered out of a manual, no violation of principle is involved; it is not dishonest to ask another student the value of *PI* or the statement of the *Pythagorean Theorem*. But the more your request is for part or all of the solutions to the programming exercise itself, rather than for general factual information, the less acceptable it is. In the extreme case one student asks for and receives the actual text of a program which both were to have created independently; in this case both are guilty of academic dishonesty.

In the Harvard College *Handbook for Students* is a section related to collaboration:

> It is expected that all homework assignments, projects, lab reports, papers, theses, and examinations and any other work submitted for academic credit will be the student's own. Students should always take great care to distinguish their own ideas and knowledge from information derived from sources.

In some courses students are expected to work in teams on the implementation of very large programs. Just because you see two students huddled over the same terminal and discussing programs in great detail, do not assume that this is standard and acceptable behavior in your course! If you have any doubt about what type of collaboration is permissible, do not make assumptions: ask the instructor. A general argument that you were only doing what you saw others doing is not a legitimate defense.

VI. Diversity, Inclusion and Other Matters

I would like to create a learning environment in our class that supports a diversity of thoughts, perspectives and experiences, and honors your identities (including race, gender, class, sexuality, socioeconomic status, religion, ability, etc.). I (like many people) am still in the process of learning about diverse perspectives and identities. If something was said in class (by anyone) that made you feel uncomfortable, please talk to me about it. If you feel like your performance in the class is being impacted by your experiences outside of class, please don't hesitate to come and talk with me. As a participant in course discussions, you should also strive to honor the diversity of your classmates.[7]

If you have a health condition that affects your learning or classroom experience, please let me know as soon as possible. Students who would like to request accommodations for disabilities should contact the Accessibility Services office at Accessibility@dcemail.harvard.edu or 617-998-9640 See this website for more information: https://www.extension.harvard.edu/resources-policies/accessibility-services

As a Harvard Extension student you have certain responsibilities. Please familiarize yourself with the school policies at this website: https://www.extension.harvard.edu/resources-policies/student-conduct

[7] This statement is excerpted from one by Dr. Monica Linden at Brown University.

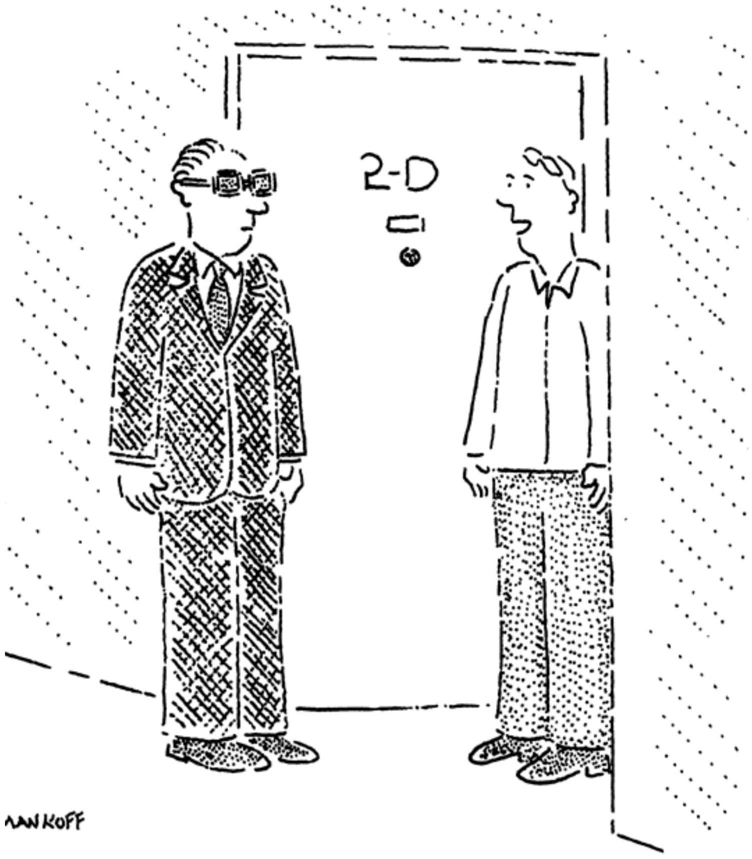
VII. Syllabus

The following calendar and outline should give you an idea of how CSCI E-7 will progress. Because of the pandemic, the actual dates of these lectures may change if Harvard decides to delay the start of the spring semester.

Date	*Topics to be Covered and Optional Reading*
Tues., Jan. 25	Course overview and a brief look at some of the "great ideas in computer science," including the notion of algorithm and its relationship to programming.
Thrs., Jan. 27	Introduction to programming in *Scratch*, a visual, object-oriented programming language.
Tues., Feb. 1	*Scratch* programming, conclusion. The *ide.cs50.io* Linux environment.
Thrs., Feb. 3	A quick overview of Unix/Linux, HTML5 and the World Wide Web. A little bit of *Javascript.*
Tues., Feb. 8	Elementary programming in Python (variables, constants, assignment, console output). Comparison with *Scratch* constructs.
Thrs., Feb. 10	Elementary Python, part 2: arithmetic operators, strings, parameterless methods, nested loops using **for**. The *range* function.
Tues., Feb. 15	Elementary Python, part 3: operator precedence. Programming with "style." Keyboard input using the *input* function. Type conversion functions.
Thrs., Feb. 17	Printing interesting patterns with nested loops. Testing programs and the flow of control.
Tues., Feb. 22	The *Boolean* data type and boolean operators and variables; formulating complex conditions using the logical operators. Conditional evaluation using if-else. Algorithm for "nested squares."
Thrs., Feb. 24	Guest lecture on ethical issues in computing.
Tues., Mar. 1	Integer overflow and floating-point imprecision. Other things that

	can go wrong. Computing Fibonacci numbers. Parameter-passing mechanism. Returning values from methods. Importing modules.
Thrs., Mar. 3	Simulation and Monte-Carlo methods through the use of pseudo-random numbers. A somewhat whimsical look at a particular aspect of computing history. Indefinite repetition using the *while* loop.
Tues., Mar. 8	Using a *boolean* variable. ASCII and UNICODE conventions for character encoding.
Thrs., Mar. 10	Fun with *String* objects. Introduction to Python *lists*.
Saturday, March 12 through Sunday, March 20 is *Spring break!* No courses meet	
Tues., Mar. 22	More on *Lists*. Simple sorting algorithms. Complexity, and Big-*O* notation. Discussion of efficiency using various sorting methods (*bubblesort*, *selection sort*, *quicksort*) and search (*linear* vs *binary*).
Thrs., Mar. 24	Running time of programs, asymptotic notation and growth rates. Multidimensional lists. A program that "learns" through experience.
Tues., Mar. 29	Introduction to file I/O and command-line arguments. Manipulation of CSV files.
Thrs., Mar. 31	Utilizing API keys to download data from the web. Simple plotting using *matplotlib* module. Introduction to *Dictionaries*
Tues., Apr. 5	Turtle graphics and DrawingPanel objects. Other Python collections: *Tuples* and *Sets*
Thrs., Apr. 7	*Recursive computation.* Towers of Hanoi puzzle as an example of intractable computation.
Tues., Apr. 12	Introduction to object-oriented programming (OOP). Creating *classes* -- writing constructors, accessors, mutators and special methods. Implementing a r*ational number* class.
Thrs., Apr. 14	**OPEN-BOOK EXAM TODAY! Covers Python programming up through April 7 lecture.**

Tues., Apr. 19	Illustrating a theorem in geometry using objects. Inheritance in OOP.
Thrs., April 21	Computer architecture: The P88 and typical von-Neumann architecture. CPU, memory, bus, peripherals, storage, etc. The basic machine cycle (fetch, decode, execute, store) and P88 instruction set (including binary representation). Viewing programs as data and data as programs.
Tues., April 26	Term project peer-review presentations for "extra credit." *This is the last regular class meeting.*
Thrs., April 28	No class! Reading period begins today and lasts up through May 4.
Sat., May 7	**Term projects are due today prior to 5 pm**.



"*I think you're looking for 3-D.*"