syllabus for

problem solving and python programming

comp 115

Instructor:	Mark LeBlanc (**mleblanc**@wheatoncollege.edu)	**Open Hours** to meet in my office: by appointment
Office:	DC-1322	*or* **TW 10-11** *or* **TW 2 – 3:30**
Phone:	286-3970 (on campus: x3970)	
Class: MW 12:30 – 1:50, DC 1349 (near computer museum)		**Lab**: Thu**R 3:30**-5:20 pm, (same room) **DC 1349**

Online Text:

Interactive Python: How to Think Like a Computer Scientist
https://runestone.academy/runestone/books/published/thinkcspy/index.html
by Brad Miller and David Ranum

Preamble: Programming. There seems to be no end to the urgent need for humans to understand how to communicate with machines, to design workflows, to write what is known as "scripts, software, code, programs, apps". Mobile phones, AI models, embedded medical devices, scientific experimentation, and most computational devices require software. This course is about learning how to solve computational problems, establish workflows and algorithms for solutions, and program. And learn how to do it well. We use the programming language **Python** because it offers a full range of rigor and elegance, not to mention it remains the "swiss-army knife" for coding, including a leading use in AI. Glad you are on board; like riding a bike, once you learn to apply computational thinking and to program, you'll never forget. Let's cut code!

Content: Problem-solving techniques and algorithm development with emphasis on best practices, software reuse, numerical methods, and testing for correctness. Topics include abstraction, design and decomposition, control flow, the elementary data structures of lists (arrays) and dictionaries, and a peek at how these features form the building blocks of user-defined classes of objects. Our emphasis this semester will be on best practices for safe, efficient, scientific programming and how to increase coverage when testing for program correctness. Knowing when code is good *vs.* when it is not is vital when solving problems in our data-centric, AI-embedded world.

No previous programming experience is assumed. Out-of-class assignments and in-class labs emphasize the physical limitations of problem-solving machines, as well as techniques to write programs that are both safe and correct, with accompanying unit tests to verify correctness.

Your Grade:

Things to do	Grading Percents	Deadline
5 Programs	40%	Due to Canvas by 4am after deadline
(1) Eye-Tracking Data *v*1.0	5%	Tue, Feb 3
(2) Eye-Tracking Data *v*2.0	5%	Tue, Feb 24
(3) Detecting Huntington's Disease in DNA	10%	Tue, Mar 24
(4) Matching DNA from Doorknobs	10%	Tues, Apr 14
(5) Detecting Grade Reading Level of Texts	10%	Tue, Apr 28
Labs (approximately 12)	15%	Weekly
Quizzes (four)	10%	Prep for Exams (tbd)
Exams (two) [will be taken during lab time]	10% 10%	Exam 1 Thurs, Feb. 19 Exam 2, Thur, Apr. 9
Final Exam	15%	Tue. May 5, 2026 9am – 12pm

Your aggregate (overall, final) score will receive a "plus/minus" letter grade according to the following key:

A = 100 to 93	C = < 77 to 73
A- = < 93 to 90	C- = < 73 to 70
B+ = < 90 to 87	D+ = < 70 to 67
B = < 87 to 83	D = < 67 to 60
B- = < 83 to 80	F = < 60 to 0
C+ = < 80 to 77	

Computational Thinking

There is much misconception about computer science these days. For many, computing means using an iPhone. Yeah, that's cool … but that is not computing. Computer Science is the study of computation – what can be computed and how to compute it. The discipline encompasses "computational thinking" (Wing, 2006)[1], a universal metaphor of reasoning that defines how creative and imaginative humans use computation to facilitate communication, model complex systems, and visualize content.

Given the tangible, ubiquitous, embedded, and rapidly evolving nature of computing in our lives, the discipline of computer science faces the challenge of how to attract students to study within a discipline where some perceive they are already "experts". Clearly, relative to the previous generations that defined technology and computing by the electronic technologies at hand, the thumb-wielding, wireless generation of new students appears undaunted as they master and demand new hardware. But hardware is not computer science; a power-user is not a computational thinker. This course is about becoming a computational thinker.

Computational thinking is:
- a move away from "literacy" and toward "fluency," a broader concept including contemporary skills and intellectual capacities
- using abstraction and decomposition when attacking a large complex task
- choosing an appropriate representation for a problem
- modeling relevant aspects of a problem to make it tractable
- using invariants to describe a system's behavior succinctly
- developing heuristics to posit if and when an approximate solution is good enough
- using randomization to our advantage
- planning and scheduling in the presence of uncertainty
- search, search, and more search
- about ideas, not artifacts – it's not just the hardware and software that will be physically present everywhere, it will be the computational concepts we use to approach and solve problems, manage our lives, and interact with other people.

"Our civilization runs on software."
Bjarne Stroustrup, lead designer of the programming language C++

Learning Outcomes

This programming-focused introductory course develops essential skills in students early on by providing a *lingua franca* in which other computer science concepts can be described. While we recognize that

(programming ≠ computer science)

writing software is a core part of the *craft* in our discipline and this craft requires much practice. Our introductory course sequence uses the **imperative-first programming paradigm**, although objects are also introduced, typically at the end of this first course and throughout the following second course (COMP 118 Object-oriented Programming).

The primary learning objectives in COMP 115 come from the **knowledge areas** of Software Design Fundamentals (**SDF**) and Programming Languages (**PL**) from the CS2023 Curricula Guidelines.

[1] Wing, J. (2006). A Vision for the 21st Century: Computational Thinking, *CACM* vol. 49, no. 3, pp. 33-35.

SDF/Algorithms and Design

Topics

- The concept and properties of algorithms
- Informal comparison of algorithm efficiency (e.g., operation counts)
- The role of algorithms in the problem-solving process
- Problem-solving strategies
    - Iterative mathematical functions
    - Iterative traversal of introductory data structures (e.g., arrays/lists)
- Fundamental design concepts and principles
    - Abstraction
    - Program decomposition

Learning Outcomes:

1. Discuss the importance of algorithms in the problem-solving process.
2. Discuss how a problem may be solved by multiple algorithms, each with different properties.
3. Create algorithms for solving simple problems.
4. Use a programming language to implement, test, and debug algorithms for solving simple problems.
5. Apply the techniques of decomposition to break a program into smaller pieces.

SDF/Fundamental Programming Concepts

Topics:

- Basic syntax and semantics of a higher-level language
- Variables and primitive data types (e.g., integer and real numbers, characters, Booleans, strings)
- Expressions and assignments
- Simple Input/Output (I/O), including file I/O
- Conditional and iterative control structures
- Functions and parameter passing

Learning Outcomes:

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs variables, expressions, assignments, I/O, control constructs, functions, and parameter passing.
2. Identify and describe uses of primitive data types.
3. Write programs that use primitive data types.
4. Modify and expand short programs that use standard conditional and iterative control structures and functions.
5. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing.
6. Write a program that uses file I/O to provide persistence across multiple executions.
7. Choose appropriate conditional and iteration constructs for a given programming task.

SDF/Fundamental Data Structures

Topics:

- Arrays
- Strings and string processing
- Strategies for choosing the appropriate data structure

Learning Outcomes:

1. Discuss the appropriate use of built-in data structures.
2. Describe common applications for each of the following data structures: matrix, array/list
3. Choose the appropriate data structure for modeling a given problem.

SDF/Development Methods

Topics:

- Program comprehension
- Program correctness
    - o Types of errors (syntax, logic, run-time)
    - o The concept of a specification
    - o Unit testing fundamentals and test-case generation
- Modern programming environments
    - o Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

Learning Outcomes:

1. Trace the execution of a variety of code segments and write summaries of their computations.
2. Explain why the creation of correct program components is important in the production of high-quality software.
3. Identify common coding errors that lead to insecure programs (e.g., buffer overflows) and apply strategies for avoiding such errors.
4. Apply a variety of strategies to the testing and debugging of simple programs.
5. Construct, execute and debug programs using a modern IDE and visual debuggers.
6. Construct and debug programs using the standard libraries available with a chosen programming language.
7. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software.

DETAILS

Late Submissions:

Due is due. Always turn in whatever you have **on time**. Something turned in on time even if unfinished is much better than not having it accepted because it is late. **Late is not an option**. (*Good, glad we can all agree with this*). Note that the Canvas page associated with this course will have Assignment submissions (uploads) that are time-triggered; once you are late, submissions will be blocked. Before the deadline, you may resubmit assignments and I will only review your last submission.

All programming assignments are due by 4am the (early) morning following the deadline. For example, your first programming assignment is due on Tuesday, Feb. 3, so you can submit until 4 am the following Wednesday morning.

Assessments & Grades:

This course has been designed in compliance with Wheaton College's credit hour policy.

Hardcopy printouts of your Python code will be due at the beginning of the next class following the assigned date. Thus, if a program is due on a Tuesday, you must bring your professionally formatted and printed/stapled hardcopy to class the next day. Of course, you must have previously uploaded your code to Canvas by the Tuesday deadline. Your **stapled** hardcopy submissions will be collected at the beginning of the class prior to beginning that day's new topic. Your Prof absolutely needs this hardcopy, it is not optional. Yes, the **stapled** hardcopy submission is worth points on the grade key.

Class attendance and participation is required. In class, you will often work with others to solve problems and practice the terminology (in small groups and/or at the whiteboard). **You should consider class meetings like a "briefing" at a real job, thus you must be involved, participate, question, and engage with the material during class time.**

Communications: You may communicate with myself or the TA primarily via email or in-person visits.

Honor Code Revisited:

It goes without saying that all submitted work will be the student's own, in keeping with the Wheaton Honor Code. For in-class labs, you may get "help" from fellow classmates, but remember that all completed work must be your

own. Use discretion; don't ask others for "the" answer or for lines of code. However, I do encourage you to discuss the problem in general, such as the type of statements or functions one might use. For programming assignments, your answers and software must be your own from beginning to end. Here is an analogy. Almost no one would ever "use/steal" a line or two from another person's poem. Consider it the same with your programs. Don't "borrow/use" lines or sections of code from another person. Your program is (like) your poem; everyone's program should be unique. Be wise. If someone is asking you for too much help, be honest and remind them that your program is just that, *your* program.

Course AI Policy
This course will follow Wheaton's "Unrestricted Use of AI with Acknowledgment" policy [AI Use in the Classroom: Guidelines for syllabus statements and framework for the development of a college policy].

In this course, I encourage you to use all the tools at your disposal. However, as with any other resource you use to aid your work... you must acknowledge (i.e., cite) the AI tools that you use in the development of your work. We will lean on and leverage bots to help with various tasks and workflows, but we will also agree to include citations and acknowledgments in our own works.

What About ChatBots?
We treat AI-based assistance, such as chatGPT or Gemini (or Claude or Copilot or …), the same way we treat collaboration with other people: you are welcome to talk about your ideas and work with other people, both inside and outside the class, as well as with AI-based assistants. However, on Programming Assignments, your submitted code must be your own.

You should never include in your assignment any lines of Python that were not written directly by you. We know chatBots can write introductory Python, but copy/paste is not the competency we are seeking. Including lines of code you did not write in your assignment will be treated as an academic misconduct case.

If you are unsure where the line is between collaborating with AI and copying from AI, we recommend the following heuristics:

- Never hit "Copy" within your conversation with an AI assistant (and then "Paste" into your code). You can copy your own work into your conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.
- Do not have your assignment and the AI agent itself open on your device at the same time. Similar to above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI assistants that are directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.

Portions of the grading for some programming assignments will include an in-person debrief meeting. During open hours or at an agreed upon time, you will lead a "walkthrough" of your solution, including your oral summary of your solution's workflow as well as answer questions about your solution.

And to be fair, from my faculty point of view, I will include a clear statement in any class materials or assessments that involve AI use.

Work outside of class: It is expected that you spend at least 2-3 hours on reading and practice problems for every day we meet. Reading the text and working through the online text's exercises are a good way to prepare for Exams. Also, it is expected that you also spend at least 6 hours per week on your current programming assignment. WARNING: Programmers typically underestimate the time it takes to complete a software project; 6 hours per week on your programming assignment may be one of those "under-estimations."

In-class Labs: The labs are a critical part of the course. In almost all cases, the current lab will be preparing you for the current programming assignment. That is, if you complete and understand the labs, you should be well on your way to a solution for the programming assignment. In order to best grasp the material in some labs, I strongly suggest that you completely redo any labs that you find difficult. (Read that last sentence again, unless of course you've already reread it once).

Quizzes and Exams: There will be no make-ups, nor will the lowest grades be dropped. If you have a conflict with a quiz or exam date, please talk to me. While practicing, try not to peek at the textbook or an earlier solution. NOTE: it's easy to "read" code; it's much more difficult to "write" code from scratch on a blank sheet of paper; you just must practice! **Quizzes and exams will ask you to write code from scratch; practice ownership of the concepts!**

Our online text, our hands-on time "in class," and the lectures will continually provide homework exercises for you to complete. Of course, if you do not work on them consistently, it will be very difficult to work through them all later on. Most of these exercises are small in nature, at least compared to your programming assignments. Let me attempt to motivate you with an analogy: if you had to compete in a diving competition (and you had never dived from a diving board before in your life), you would certainly *practice* many, many times *before* you allowed yourself to be watched. So it is with learning a spoken language *and* a programming language: it will take *much* practice. And note that I do not mean practice so that you "almost get it", rather I mean a level of practice where you feel that you *own* most, if not all of the techniques … at a level where you can apply them correctly.

In computer science, if you are almost correct you are a liability.
Fred Kollett (1941-1997), Math/CS, Wheaton College

Academic Honesty Statement:

Established by students for the purpose of self-governance in 1921, The Honor Code is a commitment to the ideals of academic excellence and individual responsibility:

As members of the Wheaton community, we commit ourselves to act honestly, responsibly, and above all, with honor and integrity in all areas of campus life. We are accountable for all that we say and write. We are responsible for the academic integrity of our work. We pledge that we will not misrepresent our work nor give or receive unauthorized aid. We commit ourselves to behave in a manner that demonstrates concern for the personal dignity, rights and freedoms of all members of the community. We are respectful of college property and the property of others. We will not tolerate a lack of respect for these values.

In short, you signed the Honor Code; thus, you've declared that you will not cheat. I believe you.

Accessibility Statement

Wheaton College is committed to providing equitable access and supportive services for all students to fully access and thrive in the academic, residential and social aspects of student life at Wheaton College. Affirmatively, Wheaton provides appropriate accommodations for eligible students with documented disabilities to afford equal access to educational programs and services. Individuals with disabilities and other access concerns requiring accommodations or information on accessibility should reach out to Accessibility Services at the Filene Center, either via email at accessibility@wheatoncollege.edu or via phone at (508) 286-3794.