SYLLABUS

Why would you want to program in Python?

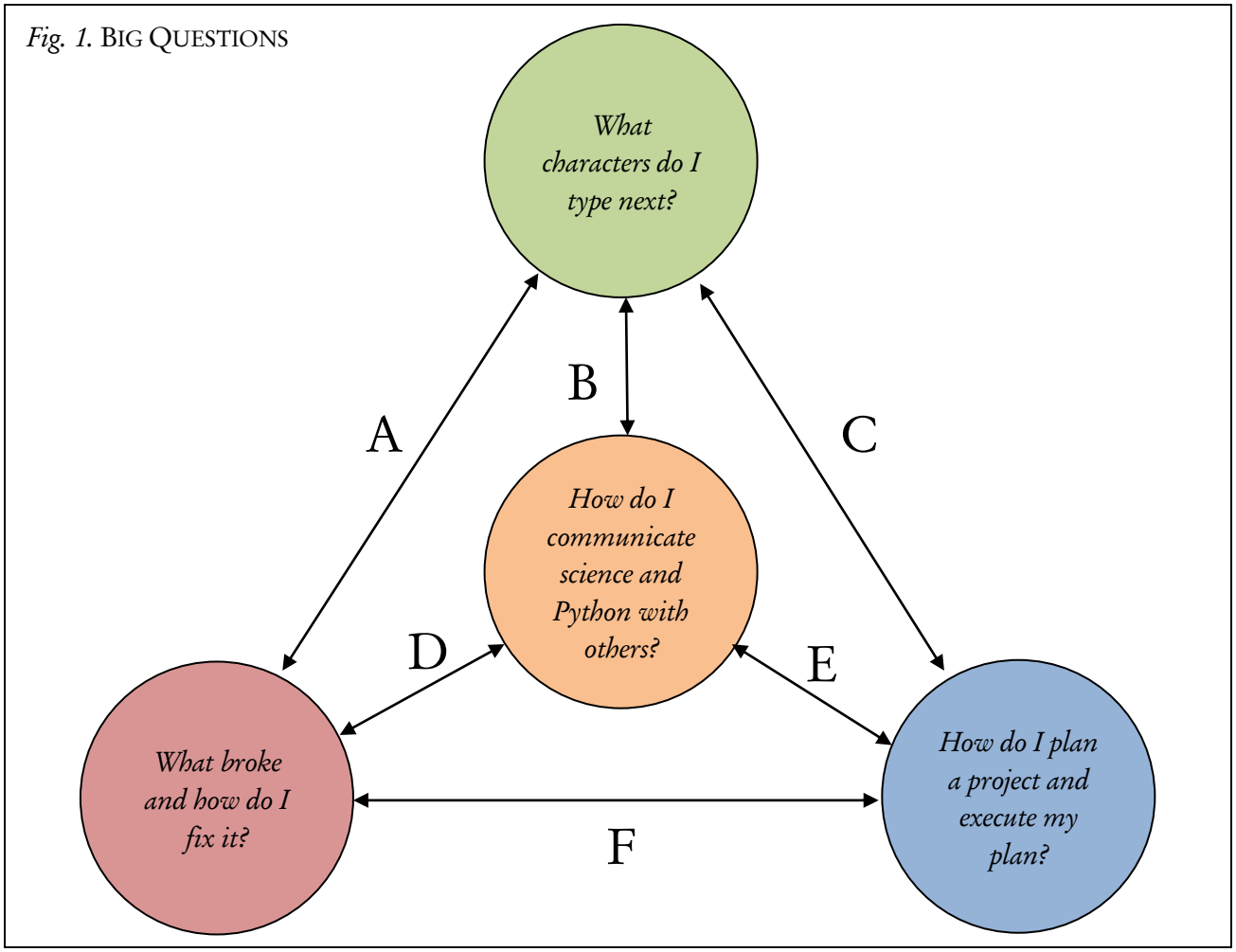
If you're considering going into a scientific field in the modern era, writing code will likely be an essential job skill. For these applications, Python is the most legible and most versatile language around. It's easy to start learning because it's transparent, and it's easy to keep learning because bountiful libraries and documentation are available online. This means that in Python you can get your code working very quickly and get to the science right away. Furthermore, you can use available libraries to optimize your code so that it can efficiently process large simulations and data sets. If this is what you're looking for, then Python is the language for you.

What will you get out of this course?

You will use Python to complete a project that is interesting and relevant to you.

Everything we do is to prepare you to succeed on that project, and to help you apply the lessons from that project to future work. Towards that goal, we will focus on four questions (see Fig. 1):

Fig. 1. BIG QUESTIONS



We came up with these questions by examining our own process for approaching Python projects. These are questions that we ask and answer constantly while working on a project, to the extent working on the project *constitutes* answering these questions. Consequently, our approach to helping you develop proficiency with Python is to help you answer these questions for yourself.

About the Research

In college, the person in front of a classroom is supposed to be an expert at what they do. We won't claim to be expert Python programmers, but we do have expertise that you don't (or else why would you be here?). Our expertise is what allows us to help you achieve the same expertise, but at the same time it is a liability: once we have mastered a skill, we have a tendency to forget all the component knowledge and sub-skills that go into performing that skill (*HLW* 98). We operate from a place of *unconscious competence*, also called the *expert blind spot*.

The framework of questions we've constructed for this class is our attempt to work backwards to fill in that blind spot, to become conscious once more of the skills that contribute to our competence (in line with *HLW* 112, as well as Pace & Middendorf). The result is informed by three pieces of research about mastery:

- **Decomposing component skills:** When you have an expert blind spot, it's easy to accidentally not teach something important. In order to tell where you're struggling so that you can practice that skill, we need to be aware of all the ways you could struggle—i.e. all the skills you need to have to succeed (*HLW* 100). Above, we've begun the process of categorizing those skills so that we can target them during instruction.
- **Applying component skills:** The research shows that students "often do not successfully apply relevant skills or knowledge in novel contexts" (*HLW* 108), i.e. *transfer* their knowledge. Having a skill doesn't mean you know when to use it. In order to facilitate transfer (Hansen 33), we've organized those skills into *big ideas*—in this case, associated with questions that you'll find yourself asking while coding—so that when those questions come up you'll have access to the answers you've learned. We introduce the big ideas below.
- **Integrating component skills:** Skills that you can perform individually are often difficult to combine (*HLW* 103). While programming, you'll be jumping back and forth between the questions above, so you'll be answering multiple questions at once. We've put our questions into a framework and labeled the connections (A–F, see below) as a first step towards helping you integrate the skills into a coherent whole. Each of the six connections describes a skill that solves problems aligned with two big ideas at once.

Hansen recommends organizing a course around only two or three big ideas (36). We've opted to use four because (1) otherwise you would be missing a piece of the puzzle; (2) organizing the concepts in this way is ideally suited to the task for which we wish to prepare you (*HLW* 48), so recalling all four big ideas is unlikely to be an issue; and (3) we have selected enduring understandings to connect big ideas, not underlie them, so they are manageably few.

What are the big ideas of this course?

The questions above have no simple answers. The answers are large and complicated, which is why we'll spend 10 weeks helping you start to unpack them. So that we can constantly let you know where we are in the process of answering each question, we've associated each question with a concept that scientific programmers use to think about the process of programming. These are our big ideas:

What characters do I type next?

I. Syntax & Functionality

Syntax is the set of rules that the computer uses to read your code. Functionality is what the computer can do with it. You'll learn all of the syntax and functionality Python has to offer. You'll also learn how to use a few common libraries used for scientific computing, which provide their own functionality. This big idea is about the easiest and fastest ways to get Python to do what you want it to. This is the big idea we'll talk about most.

How do I plan a project and execute my plan?

II. Data Flow & Workflow

A program takes input data and produces output. That's data flow. A programmer takes a desired task and produces a program that accomplishes it. That's workflow. This big idea is about building the perspective and infrastructure to support a programming project.

What broke and how do I fix it?

III. Debugging

Python is pretty good at telling you what went wrong. But you need to know how to interpret what it's telling you, and figuring out how to fix it can be a lot more complicated. This big idea is about how to know when your code doesn't work, and what to do about it.

How do I communicate science and Python with others?

IV. Plotting & Documentation

For scientists, the ultimate goal in programming is to learn and communicate what they've learned. That's what plotting is for. For software engineers, the final step in writing a program is often to make it accessible to others who wish to use it. That's what documentation is for. We'll learn how to make plots and how to read and write documentation. This big idea is about profiting from and giving back to the existing scientific computing community.

What enduring understandings will I gain from this course?

Some of the ideas we'll discuss provide powerful ways of combining skills from different big ideas. They may seem obvious at first, but they have counterintuitive repercussions that deserve special emphasis. While Big Ideas are about ideas and declarative knowledge, these enduring understandings are about habits and procedural knowledge. We've separated them because you'll need a different kind of practice to make them effective and automatic. For ease of recall, we've integrated them into Fig. 1.

SYNTAX & FUNCTIONALITY

A: PYTHON IS A PHYSICAL SYSTEM. EXPERIMENT!

DEBUGGING

Scientific programmers can theorize and experiment about their code. When your code misbehaves, change something and see if it works! If not, figure out why. You'll learn to investigate your code by experimenting with it.

SYNTAX & FUNCTIONALITY

B: LET ME GOOGLE THAT FOR YOU.

PLOTTING & DOCUMENTATION

Google is renowned for its ability to produce results relevant to programmers. When you need to learn or re-learn how to use a piece of functionality, Google it! You'll learn when to turn to Google to find the answer to your question.

SYNTAX & FUNCTIONALITY

C: COMPUTING TIME IS CHEAP—USE IT.

DATA FLOW & WORKFLOW

For everyday tasks, the slowest step is almost never the computer—it's the programmer. Python gives you the resources to write things quickly and get them done, so use them! You'll learn how to prioritize coding time over runtime.

PLOTTING & DOCUMENTATION

D: READ THE ERROR OUTPUT. READ IT.

DEBUGGING

When a remote-control car stops working, it doesn't tell you why. Did the remote batteries die, or the car batteries? Python, on the other hand, is downright loquacious about errors. You'll learn to read and interpret the error output to fix your code.

PLOTTING & DOCUMENTATION

E: DON'T REINVENT THE WHEEL.

DATA FLOW & WORKFLOW

In many languages, you have to write everything from scratch. In Python, that's almost never true: if it's general enough, someone else has probably done it. You'll learn to recognize the point when you can use the wheels others have already built.

DEBUGGING

F: WRITE AND TEST, WRITE AND TEST,...

DATA FLOW & WORKFLOW

You know that coding project at the end of the quarter? You're not going to write it correctly all at once, and if you try to it's going to be a pain to debug. Split it up! You'll learn to write code in chunks and test them along the way.

About the Research

The term *enduring understanding* comes from a framework for course design called *backwards design*. Enduring understandings are counterintuitive ideas that discipline experts see as established fact (Hansen 36). When these ideas fall in an expert's blind spot, their counterintuitive nature makes them particularly difficult for students to grasp; hence, they deserve special emphasis.

In selecting our enduring understandings, we have been guided by Hansen, but we have departed from it in important respects. In particular, whereas Hansen sees enduring understandings as subordinate to big ideas, we have selected enduring understandings that bridge big ideas. We have made this choice in light a few considerations:

- as noted above, we've restricted ourselves to only a few of them, so the scope of the course is still manageable;
- in our experience teaching this class previously, most of the content of this class is easily integrated into existing knowledge structures (syntax, functionality), but there are some difficult-to-integrate *habits* that we believe students will find valuable (the enduring understandings); and
- these enduring understandings still meet the criteria outlined on p. 37, in particular that they "summarize important strategic principles" for this skill area (Hansen 37).

What are the learning outcomes of this course?

We've picked one learning outcome for each big idea, and one for each enduring understanding. By the end of the course, you should show evidence that you can do these things:

Table 1. LEARNING OUTCOMES

Label	Learning Outcome
I.	Apply syntax and functionality of Python and common Python scientific computing modules, including control statements, loops, functions, data structures, programming methodologies, and regular expressions, towards solving real-world programming tasks.
II.	Organize project files and decompose functionality in a manner that transparently shows workflow and data flow. Use Git for version control. Navigate Unix and a text editor.
III.	Apply the Python debugging toolkit, including print debugging, PDB and UnitTest, to identify, explain, and solve problems with self-written code.
IV.	Learn/relearn features of Python and its modules by reading documentation. Produce transparent documentation for code and plots for data.
A.	Use experimentation to explain bugs in code and learn/relearn functionality and syntax.
B.	Find answers to syntax and functionality questions using online search engines.
C.	Identify when processing power is not an issue. In those situations, write code quickly instead of writing fast code.
D.	Respond to Python-raised exceptions by reading the error report, including the description and the traceback. Interpret it to inform debugging.
E.	Predict when desired functionality is likely already implemented and available online.
F.	Write code modularly and test functionality incrementally.

About the Research

These learning outcomes don't describe what you should "know" or "understand" by the end of the course—they describe what you *can do after* completing the course (Hansen 40). That's because your knowledge won't be useful to you unless you can use it, and because your knowledge won't be useful to you if it doesn't persist beyond the end of the course. Phrasing them in terms of what you can do also makes them measurable, so that we can target these outcomes with our assessments (see below).

We have aimed to make these learning outcomes appropriately challenging for a 2-unit course. If we've succeeded, it means you'll be motivated to reach these goals without being overextended. In particular, we are not trying to cause a paradigm shift in your worldview—this is a small, practical course. Consequently, in choosing action verbs we have stuck mostly to the "application" facet of understanding (Hansen 41).

The rest of Hansen's facets would be great if this were a course that focused on ideas and how we make them valuable to you, but they have little to say about ways to understand a technical skill. (Maybe he's criticizing courses that thoughtlessly teach skills without regard to their value. We recognize that criticism and its relevance to the social problems in the Silicon Valley, but nevertheless, cautiously, we proceed.) We've had to introduce a couple of our own:

- **Response:** fluid performance of a task requires making sound choices and judgments automatically. Response is explicit in learning outcome D, and it is implicit to some extent in the habitual nature of learning outcomes A–F.
- **Learning/relearning:** scientific computing develops rapidly, so programmers have a lot to gain from staying on top of new developments by teaching themselves new tools. Programmers often use a piece of functionality once and then need to recall how it works five years later, so they need to be able to re-teach themselves tidbits of easily forgotten knowledge. Outcome IV is all about being a part of the scientific computing learning/relearning community.

You might worry that this is too many learning objectives for a 2-unit course. Fear not; most of our time will be spent learning content related to objective I, and much of the rest on objectives II–IV. Objectives A–F are small, contained habits that we'll help you cultivate, and as such will need little in-class time to understand. Most of the learning on these objectives will take place when we merely remind you of their existence.

How will you be assessed?

The summative assessment of this course will be the final project you will complete throughout the quarter on a topic of interest and relevance to you. It will assess your progress towards all 10 learning objectives of the course. We will track your progress towards the learning objectives by evaluating your in-class your coding assignments (see below).

This class is graded on a Satisfactory/No Pass basis. In order to guarantee a passing grade, you need to complete the final project satisfactorily (we'll provide a rubric), on time, in its entirety; be present for every lesson; and submit substantial working code at the end of every lab. If you do not complete one or more of these requirements, let's talk, but we can't guarantee you'll pass.

The Final Project

Your task will be to plan, implement, debug, and document a piece of software of a topic of interest and relevance to you. This project will be the exclusive homework for the class, and will require a time commitment of between 10 and 19 hours over the course of the quarter. (Along with the 38 hours of class time, this completes the 57-hour commitment expected of a 2-unit class over 9.5 weeks.)

Most of the time you'll spend on the project will be spent coding. But since this is your first experience using Python for a project, we'll ask you to spend some portion of your time *thinking about coding* using the tools from this class. This will help you apply those tools to the project and to future projects, and it also provides an avenue through which we can assess your learning. Here's a preview of all the parts of the project:

I. SYNTAX AND FUNCTIONALITY: We'll be looking through the code you submit for evidence that you can select elements from the categories of syntax and functionality we'll discuss and apply them appropriately towards your programming task.

II. DATA FLOW & WORKFLOW and F. WRITE AND TEST: We'll inspect the way you have decomposed your code to look for transparency of data flow. We'll also ask for the repository in which you store your final project so that we can inspect the way you have stored your data, tracked your progress on the project, and tested functionality incrementally. There will be a short presentation on the last day of class during which we will assess your ability to navigate Unix and operate a text editor.

III. DEBUGGING: We can guarantee that you'll encounter bugs during the project. We'll ask you to approach them conscientiously, using the tools we'll discuss in class, and then to write briefly about how you used those tools to identify and solve one bug.

IV. PLOTTING & DOCUMENTATION: We will ask you to research and implement functionality from one module we do not discuss in class that helps you with your project, and then briefly explain how that piece of functionality works. This will require reading and interpreting other programmers' documentation. We will further ask you to document sparsely your own code, as well as one portion in depth. Finally, we will ask you to produce a plot showing the results that your project produced, and to explain it during the presentation.

A–E. ENDURING HABITS: Each of these learning objectives describes a habitual response to a certain situation: (A) unexplained program behavior, (A and B) missing knowledge, (C) implementation options, (D) error reports, and (E) wanting complicated, general functionality. We will ask you to approach these coding situations conscientiously, with an eye for the habits that we've presented as solutions, and then to describe briefly one instance in which you used the habit to solve your coding problem.

About the Research

This is a big project. It will be challenging, but it will be doable—the class is tailored to prepare you for it. The project is intended to put you off-balance so that you'll need the class

to help you succeed. This approach to course design Gabe learned from Senior Lecturer Denise Pope in the Graduate School of Education, who pointed out that students faced by a moderately challenging project are aware of what they still need to be able to do to succeed, and thus are incentivized to learn lasting lessons from classes and assignments. In the terminology of *HLW* Ch. 3, assigning you a personalizable project to encompass all of the course objectives gives *value* to your everyday learning tasks. (If you find that coding an independent project is not valuable to you, this is probably not the right class for you.) The learning tasks, meanwhile, provide a *supportive environment* and boost your *efficacy* at completing the final project (if we're doing our jobs right). When students see value in a task, feel efficacious at it, and feel supported by the environment that assigns it, research shows that they are *motivated* (*HLW* 82)—that's how we're hoping you'll feel about learning how to program in Python.

Part of the value in this task is that it's *authentic*—it's pretty much what you'll want to be doing with your learning from the class anyway. Among the six characteristics of authentic assessments that Hansen lists on p. 87 is that they are "designed to be part of the learning act itself": by asking you to think about coding while coding, we're not only assessing your knowledge; we're incentivizing you to gain reflective experience applying the tools you're learning during the class.

How will you help us prepare to complete the final project?

One way that we'll help you complete your final project is by setting due dates for parts of the project. This encourages you to work on the project steadily throughout the quarter, rather than completing the whole thing the weekend before it's due. Here's a preview of what will be due when:

- **Week 2 (Apr 7):** Read this syllabus and take a crack at a tentative project idea.
- **Week 3 (Apr 14):** Set up the workflow for your project.
- **Week 4 (Apr 21):** Take another crack. Write up the topic, scope, and data flow of your project.
- **Week 5 (Apr 28):** Get a piece of code working and document it. Habit summary 1.
- **Week 6 (May 5):** Write a piece of object-oriented or functional code. Habit summary 2.
- **Week 7 (May 12):** Research and implement an online library. Habit summary 3.
- **Week 8 (May 19):** Submit a plot of some preliminary results. Habit summary 4.
- **Week 9 (May 25):** Submit working rough draft. Habit summary 5.
- **Week 10 (June 1):** Submit final draft and present. Habit summary 6.

The other way we'll help you is by using our class sessions explicitly to help you learn and practice the skills you'll need to succeed on the final project. Each class session is divided roughly in half:

- **Interactive lecture:** Conrad and Young will spend the first 50 minutes bringing you from where you are now to where you'll need to be to complete the lab. Lectures will begin with feedback to the class based on recent assignments. Much of the rest of the time Conrad or Young will be at the board demonstrating how new syntax and functionality work, modeling its use on the projector. They'll intersperse the lecture with brief practice questions and discussions to gauge your understanding and give you targeted practice. They'll also model techniques for workflow, debugging, and documentation, and help you organize and prioritize the concepts we'll be discussing.

About the Research

In Gabe's class, he and a classmate settled on a good criterion for what class time should be spent on: it should be at least as useful to students as if they were spending that time learning on their own, through reading, doing tutorials, or whatever else students might think of. McKeachie provides some tips about what lectures are good for that help us meet this criterion: updating, unifying, and personalizing resources; orienting students to these resources and to the structure of the class; prioritizing key concepts; and modeling an expert's approach to the discipline (59). You'll notice above that we'll be doing a lot of modeling, orienting, and prioritizing, but the rest will be there too.

- **Individual and group lab work:** You'll spend the last hour working on programming tasks we've prepared that give you targeted practice with the skills you'll be learning. We've set learning goals for each lesson (see Table 2), your progress towards which we'll assess by evaluating the lab assignments you submit. While the mid-lecture exercises will provide targeted practice of isolated skills, the labs will give students opportunities to practice integrating these skills with existing skills and applying them to realistic problems. We'll provide if-you-have-time problems for the lab so that learners at different levels of proficiency can challenge themselves appropriately.

  Labs will be submitted individually, but the lab format will take advantage of the students as a team. You'll be able to refer to your classmates for help, and in addition we'll use individual students' epiphanies as learning opportunities for everyone. Occasionally, we'll have an activity explicitly designed for pairs or groups; in Week 6 you'll attempt to learn how to use bits of other students' code using their documentation, and in Week 7 you'll attempt to "break" each other's code—to find cases in which it throws exceptions or returns unexpected results.

About the Research

The labs serve as a critical part of the *feedback cycle*: you engage in practice, informed by the goals we have set; we observe how well you performed at each of those goals; we tell you specific things to maintain and improve so that you can get closer to those goals; and then you practice once more, informed by that feedback (*HLW* 126). Our job is to make the labs challenging enough that you learn but easy enough that you can do them. The sweet spot is called the *Zone of Proximal Development*: a task that "the student cannot perform successfully on his or her own but could perform successfully with some help from another person or group" (132). We achieve this by setting up challenging tasks accompanied by *scaffolding* to reduce the cognitive load demanded of you by extraneous tasks.

Works Cited

McKeachie, Wilbert J. and Marilla D. Svinicki. *McKeachie's Teaching Tips*. 14th ed.
Ambrose, Susan A., *et al*. *How Learning Works.* San Francisco, CA: Jossey-Bass, 2010.
Hansen, Edmund J. *Idea-Based Learning*. Sterling, VA: Stylus, 2011.

Table 2. CLASS OUTLINE

WEEK 1 Mar 31, Apr 2	if/else, I/O, loops, funcs; print debugging; Strings, lists, advanced structs, types; A	PROJECT MILESTONE DUE DATES
WEEK 2 Apr 7, 9	Unix, text editors, Git, workflow; file I/O; how to read errors; D	Read syllabus; first crack
WEEK 3 Apr 14, 16	data flow; NumPy arrays; basic plots; NumPy and SciPy funcs; how to read docs; B	Set up workflow
WEEK 4 Apr 21, 23	basic object-oriented programming; debugging with PDB; exceptions; how to write docs; F	Second crack; pick topic, scope, data flow
WEEK 5 May 28, 30	advanced object-oriented programming; functional programming; E	Piece of working, documented code; Habit summary 1
WEEK 6 May 5, 7	how to find libraries; project work time; regular expressions; C	Piece of OOP or functional code; Habit summary 2
WEEK 7 May 12, 24	testing; UnitTest; how to make beautiful plots; project work time	Research, implement library; Habit summary 3
WEEK 8 May 19, 21	advanced topics; guest lectures; project work time	Plot of preliminary results; Habit summary 4
WEEK 9 May 26, 28	advanced topics; guest lectures; project work time	Working rough draft; Habit summary 5
LEARNING OBJECTIVES	I. SYNTAX & FUNCTIONALITY; III. DEBUGGING; II. DATA FLOW & WORKFLOW; IV. PLOTTING & DOCUMENTATION	FINAL PROJECTS DUE Jun 1; PRESENTATIONS