The 10X Python Team

How Engineering Leaders Can Boost Python Development Velocity, Expand Team Capabilities, And Propagate Skills Team-Wide For The Long Term

Aaron Maxwell
https://powerfulpython.com/teams/

Overview: The Three Pillars

Our thesis is that leaders of Python development teams can dramatically increase speed, quality, and effectiveness of software development through a systematic process - as outlined in this book.

This process is based on directing your team's focus to the first principles of software engineering. When your team starts to operate in terms of these first principles, many positive compounding effects occur:

- **Code Quality:** They start writing cleaner, more maintainable and agile code
- **Robustness:** Avoiding more bugs, and more adroitly handling the rest
- **Velocity:** Increased speed at which developers complete tasks and deliver features
- **Efficiency:** How well developers use tools, resources, and time
- **Judgment:** Making better decisions, especially in novel situations in which the "right" answer is not known
- **Problem-Solving:** Improved ability to effectively tackle and resolve technical challenges
- **Innovation:** Finding novel game-changing solutions that open new opportunities
- **Propagation:** Spreading the knowledge and skills of top performers team-wide, even to new team members as staff changes over time
- **Morale:** Lifting a positive team dynamic that enhances every other factor
- **Retention:** "A" players want to stay for the long term
- **Recruitment:** New "A" players are itching to get in
- **Presence:** This strong performance is clearly evident outside the team, including to higher-level management

We call this a "10X Team". This is not the same as a 10X developer. An individual may indeed be a "rock star". But their contribution is mild compared to a team operating at a high level together.

It is important to understand this is not just about development velocity. It also affects *capability.*

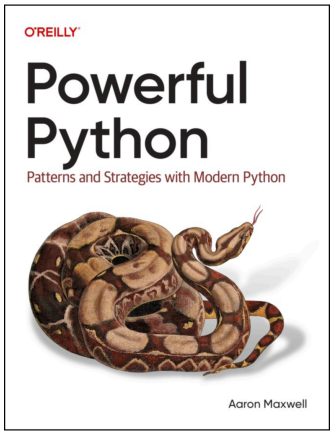
In other words, when a team becomes a “10X Team”, they accomplish things they simply could not before. This qualitative difference in capability lifts morale and magnifies the team’s actual and perceived performance within the company’s larger mission. It even boosts retention and recruitment of exceptional talent - because A players want to work with other A players.

Perhaps the best part is that this can be made to *persist* for the team as a whole, outlasting changes in staff over time.

The difference between elite engineers and "normal" coders lies in the distinctions they make, the mental models they leverage, and their ability to perceive what others cannot. This can be trained in individuals, and that is where it starts. But the greater value comes from continually propagating these characteristics for the long term… to the point it becomes ingrained in the culture and “DNA” of the team itself. The result is that this higher level of effectiveness becomes not just a quality of the current team members, but an enduring trait of the engineering organization as a whole.

There is a process by which technical teams can consistently achieve this, as detailed in this document. The high-level strategy is independent of programming language. But many technical details depend on the specific language and its ecosystem, so we will focus on Python.

My name is Aaron Maxwell. I am a software engineer and author. I have worked in several Silicon Valley engineering teams, including two Unicorns (startups reaching $1 billion valuation), SnapLogic and Lyft.



After working full-time as an engineer for a decade, I partnered with O’Reilly Media to create an extensive advanced Python curriculum. This was designed for working technology professionals… not for people new to coding, or new to Python. Over the course of 2 years, I taught this curriculum in repeated live sessions to over 10,000 professionals worldwide - in almost every engineering domain, country and culture. I also wrote a book, published with O’Reilly.

While I am proud of the Powerful Python book, it is written from the perspective of the individual contributor who wants to learn how to write better Python code faster. The book you are reading now has a different focus: how you, as an engineering leader, can navigating all the conflicting requirements and pressures to boost the performance of your team as a whole, in a lasting and effective way.

What does it take to “10X” your team, in the sense we are talking about? There are three broad pillars:

- The *Engineering Foundation* is the most technical. It deals with the first principles of effective software engineering and development. While the core concepts apply to every domain of technology, many of the details depend on the programming language and ecosystem. So we focus on Python in particular.
- The next pillar is *Thought Leadership*. Meaning the non-technical “soft” skill stack of professional communication, that enables each member of your team, at every level, to lead each other to their highest level of collective excellence. This is the pillar that allows the “10X” to automatically propagate to future team members.
- The final pillar is *AI Acceleration*. This means using AI code generation tools to amplify the effectiveness of your team, and of each team member, for creating quality production software. People have different beliefs on how important this will be. My view is that nondeterministic LLM-based generative AI will not be able to replace engineers in any meaningful sense. But it *can* improve the productivity, speed, and effectiveness of those with a solid engineering foundation.

These enhance each other synergistically.

Core Distinctions

The Team Learning Process

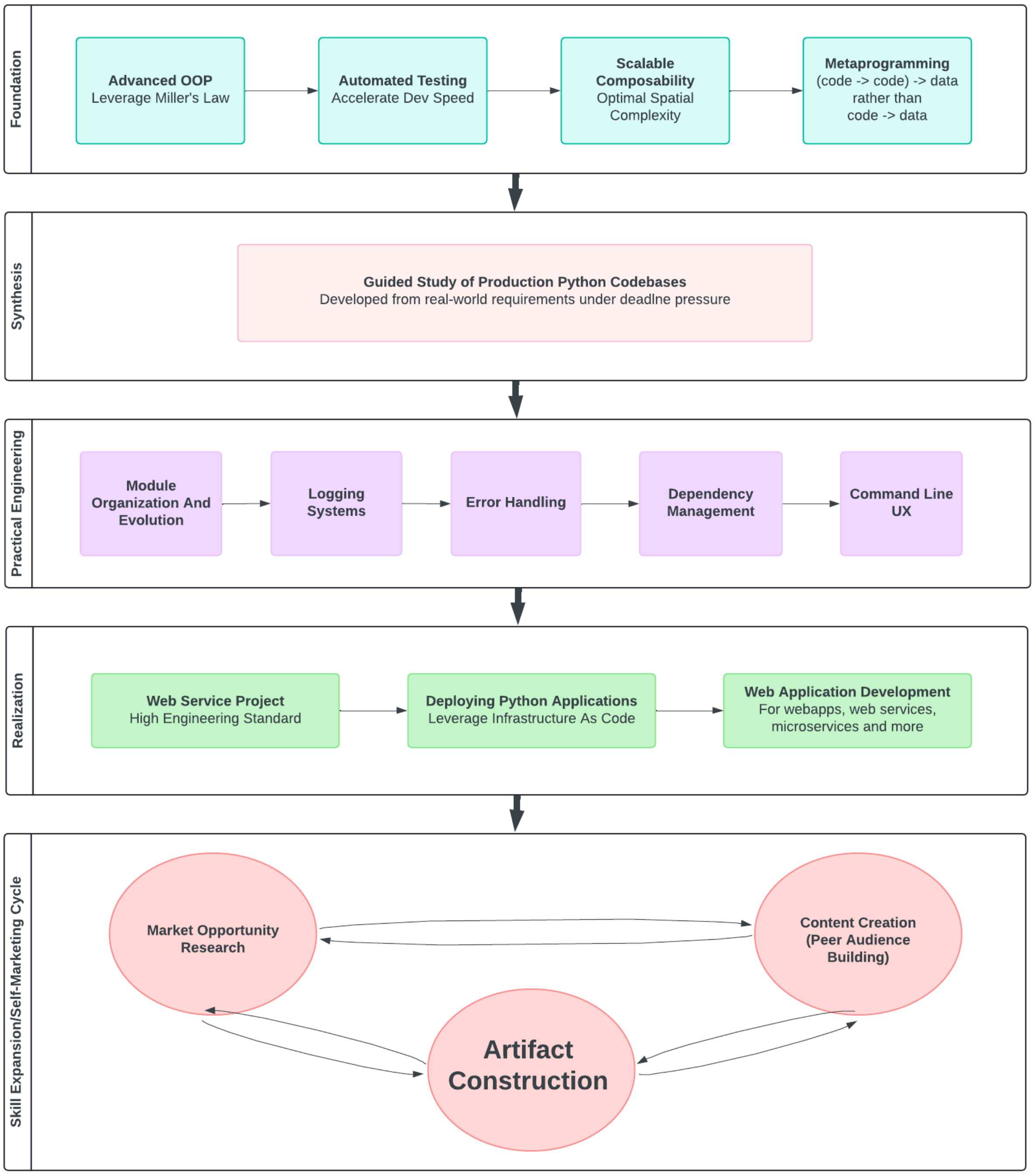
The transformation of your team requires *efficiency of learning*. This is not optional; members of your team must ramp up their knowledge and capabilities substantially, and do so with a few hours per week of focused attention. After all, you need them to be working most of the time; the engineering pipeline cannot stop for weeks or months while your people skill up.

With that in mind, here is the high level process:

- Focus on first principles. This produces the greatest boost to useful abilities from the lowest investment of deep-learning time.
- Choose learning topics in “pareto-optimal” order (explained below).
- As this foundation is established, actively apply what is learned in ongoing development work.
- Do all this in a group of ambitious peers, for accountability and better learning. This is why it is valuable for all team members to be training concurrently, when possible.

- Institute practices by which knowledge, skills and capabilities are continually transferred and reinforced between team members, including new staff as they come in.

This diagram illustrates the important building blocks, and how they relate to each other. The rest of this book explains what each element represents, and how they relate.


Foundation
Advanced OOP
Leverage Miller's Law
Automated Testing
Accelerate Dev Speed
Scalable Composability
Optimal Spatial Complexity
Metaprogramming
(code -> code) -> data rather than code -> data
Synthesis
Guided Study of Production Python Codebases
Developed from real-world requirements under deadlne pressure
Practical Engineering
Module Organization And Evolution
Logging Systems
Error Handling
Dependency Management
Command Line UX
Realization
Web Service Project
High Engineering Standard
Deploying Python Applications
Leverage Infrastructure As Code
Web Application Development
For webapps, web services, microservices and more
Skill Expansion/Self-Marketing Cycle
Market Opportunity Research
Content Creation (Peer Audience Building)
Artifact Construction

First Principles

Every domain of human activity has its own *first principles*. These are the foundational concepts, distinctions, ideas, and mental models upon which that domain is based, and which generate all value, results and new innovations. Distinguish between:

- Language-independent first principles of software development
- First principles of the *programming language* being used

The Strategy Pattern is a first principle of software development. A first principle of Python is the function object abstraction, which you can use to easily implement Strategy - as demonstrated by key functions for Python's sorted() built-in:

```
>>> numbers = [7, -2, 3, 12, -5]

>>> # Sort the numbers.
>>> sorted(numbers)
[-5, -2, 3, 7, 12]

>>> # Sort them by absolute value.
>>> sorted(numbers, key=abs)
[-2, 3, -5, 7, 12]
```

Other examples of first principles of software engineering include:

- Design patterns
- Mental models like complexity analysis (big-O notation)
- Best practice guidelines such as the SOLID principles

There are other first principles of software engineering which do not have standard names and which are not well known, except among the most exceptional performers and thought leaders.

When you learn how to think in terms of first principles, *and* gain knowledge of the first principles of a particular domain, you are suddenly capable of great achievements in that domain. You get creative insights quickly and repeatedly, routinely solve seemingly intractable problems, and regularly produce surprising new inventions.

Imagine what effect this will have on the members of your team, and the synergistic impact of how they work together.

Let me make a bold assertion: *All innovation comes from combining first principles.* Specifically, from combining first principles in novel ways; from discovering new first principles; or both.

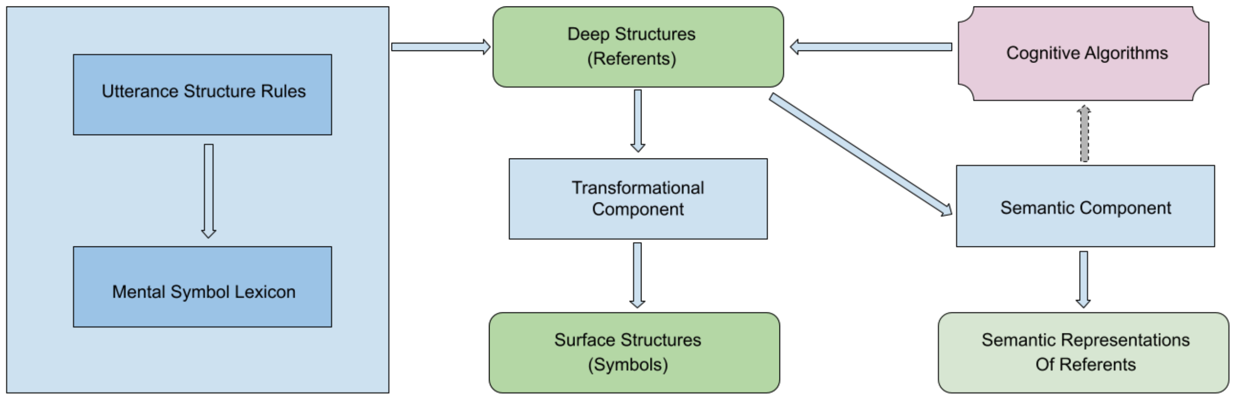
There may be some exceptions to this claim. But if you come up with some examples of what you consider to be genuine innovation, you will probably find this assertion generally fits.

For software development in particular, mastery of first principles produces a quantum leap difference in development velocity, code quality, and successfully fulfilling business requirements.

When you learn how to reason and operate from first principles, and apply that to a domain, you become a creative genius in that domain. Your abilities are recognized as rare and valuable. Your level of mastery allows you to contribute more than anyone else on the team. This applies on the individual-contributor level, and even more powerfully on the team level.

Cognitive Psychology

Cognitive Psychology is the area of psychology concerned with internal mental processes of the human mind. This includes perception; attention; internal representations of information forms, including representations of logical abstractions; and manipulation of those representations to produce conclusions and make decisions - a process we call 'thought'.


Utterance Structure Rules
Mental Symbol Lexicon
Deep Structures (Referents)
Transformational Component
Surface Structures (Symbols)
Cognitive Algorithms
Semantic Component
Semantic Representations Of Referents

Cognitive psychology has important clinical applications. But our focus here is improving performance of information technology professionals. This touches almost every aspect of the craft of software construction, from the acquisition and application of “hard” technical skills, to the “soft” skills of inter-personal communication.

Facets include:

- Internal conscious representations of data structures and other logical forms, mainly relying on the visual modality (i.e. what you see in your mind’s eye)
- Specific ‘algorithms’ for manipulation of visualized symbols, to improve rapid complex logical reasoning capability
- Internal representations of abstractions, and techniques for operating on them to establish new insights and connections

- The cognitive operation of ‘chunking’
- Improving cognitive efficiency, efficacy and correctness
- Designing new and empowering mental models which grant you differential advantage
- Understanding the distinction between symbols (e.g. words) and referents (what the words refer to)

Leveraging these gives you a key advantage as a technology professional. In the same way you can take a bloated and barely-functioning algorithm, and refactor it to be faster, clearer, more robust and more efficient, you can learn to audit and even design your cognitive processes to make you exponentially better at solving engineering problems and writing code.

The results can be profound. Simply put, **learning cognitive psychology lets technology professionals perform better and *innovate faster*.**

The tools of cognitive psychology also allow you to communicate in more efficient and transformative ways. This includes normal conversations, but also in your writing; your public speaking; and any other form of one-to-one or one-to-many communication. People get massive insights consuming your content that they simply do not get from similar content from others. They may not know *why*, but you will.

This is because your effectiveness as a thought leader depends not just on your technical expertise, but on your ability to understand how human minds work. When you can empathically communicate both individually and through mass channels; pace their current understanding and context; then lead them to more empowered and effective viewpoints, techniques, and mental models, people are genuinely transformed by your communication. They recognize that, value it highly, and naturally hold you in higher esteem as a result.

This is why an early focus on cognitive psychology is so powerful. You first leverage it to develop greater technical skills faster, and elevate your full understanding of your technology domain. And then, as you start to engage your wider professional audience through writing and speaking, you find your prowess in communication standing out even among other thought leaders.

Pareto-Optimal Skill Acquisition

Because I have trained thousands and thousands of technology professionals, I have thought deeply about the process of learning, and how to accelerate it. Especially when the people learning are smart and dedicated professionals (i.e. your team), and the topic is difficult (e.g. advanced Python software engineering). That is quite different from a beginner learning to write helloworld.py.

An important framework for this is what I call Pareto-Optimal Learning Theory, or POLT for short. This is loosely based on a concept from economics, called “pareto optimality” - hence the name.

The details are almost mathematical, but the core idea is simple:

> *When a high performer learns a complex skill, the process will go faster and easier when they learn steps in a certain optimal order. Go out of that order, and they can certainly still master that complex skill. But it may take much longer, and become harder to fully master.*

When I say “faster”, I do not mean like 10% faster. I mean *exponentially* faster. This is especially important for busy teams who are trying to get work done, and are not able to take ample time off for long and drawn-out training.

Technical Foundation

The Foundation Is Advanced OOP

Most people know that object-oriented programming helps organize code. Fewer understand it organizes how you *think* about code, too.

This has roots in cognitive psychology. Like all humans, your reasoning is denominated in the abstractions you create. This means you can improve the results of your reasoning by using better abstractions.

This is related to the well-known “7 plus or minus 2" rule, created by Harvard psychologist George A. Miller. Dr. Miller's publication on this concept has become one of the most highly cited papers in the history of psychology. Briefly, human minds optimally reason in terms of a small number of chunks (yes, “chunk" is the formal term used in cognitive psychology). That number is close to 7 in most people, hence the name of the rule.

The obvious way this applies to programming: if you have a small set of classes which map well to the business logic of what your program is attempting to do, it becomes exponentially easier to reason about your codebase and accomplish what is needed.

That is because classes represent abstractions we can reason about. When you choose the right abstractions, and design classes which represent those abstractions well, your thinking is denominated in the most empowering mental model for solving the problem at hand. You find you can write high quality software faster, and with a greater chance of successfully solving the problem you want to solve. Your code seems more elegant and clear; it omits anything not necessary, uncluttered with cruft.

A good example is the DataFrame class from Pandas. It is an abstraction, and a Python class, which has changed the world. But it is made up; once upon a time, there was no such thing as a dataframe. Someone created it, empowering countless coders to perform remarkable feats of data processing with far less effort than before. Similar comments apply to any other widely used class. *This means you can create your own abstractions (classes) that unlock massive productivity for you, your team, and possibly the world.*

As a simpler example, imagine a class called DateInterval, which represents an interval or range of days. It has a start date and an end date, and allows you to succinctly express operations on that interval. You set it up like this:

```
>>> from datetime import date
>>> interval = DateInterval(date(2050, 1, 1), date(2059, 12, 31))
```

You can check whether a particular day is in the interval or not:

```
>>> some_day = date(2050, 5, 3)
>>> another_day = date(2060, 1, 1)
>>> some_day in interval
True
>>> another_day in interval
False
```

You can ask how many days are in the interval:

```
>>> len(interval)
3652
```

You can even use it in a for-loop:

```
>>> for day in interval:
...     process_day(day)
```

The DateInterval class is the Python-code manifestation of an idea, which you are now thinking in terms of as you write code using instances of this class. Here is the source code of DateInterval:

```
from datetime import (
    date,
    MINYEAR,
    MAXYEAR,
    timedelta,
    )
```

```python
class DateInterval:
    BEGINNING_OF_TIME = date(MINYEAR, 1, 1)
    END_OF_TIME = date(MAXYEAR, 12, 31)

    def __init__(self, start=None, end=None):
        if start is None:
            start = self.BEGINNING_OF_TIME
        if end is None:
            end = self.END_OF_TIME
        if start > end:
            raise ValueError(
                f"Start {start} must not be after end {end}")
        self.start = start
        self.end = end

    @classmethod
    def all(cls):
        return cls(cls.BEGINNING_OF_TIME, cls.END_OF_TIME)

    def __contains__(self, when):
        return self.start <= when <= self.end

    def __iter__(self):
        for offset in range(len(self)):
            yield self.start + timedelta(days=offset)

    def __len__(self):
        return 1 + (self.end - self.start).days
```

Whether you currently understand this Python code is not important. What matters is you understand that you are creating a new abstraction, called a date interval, that empowers you to more efficiently reason about your codebase. And of course, it also makes it easier to write good code, and to write it faster.

Classes are also important for more classically understood benefits like code reuse, encapsulation, data hiding, etc. But even these factors relate to how we think about our code.

Automated Testing

If OOP is the foundation, writing tests is the supercharger. When you apply the patterns and best practices of writing unit tests, integration tests, and other test forms, you find it tremendously boosts your capability to create sophisticated, powerful software systems.

Automated testing does not help much when writing small scripts. But there is little value in that anyway. Your greatest contributions come from creating complex software systems that solve hard problems people care about. This is where automated testing becomes a superpower.

In particular, automated testing will tremendously speed up development. Again, not for small programs; but that is not why you are here. Automated tests enable you to successfully create advanced software systems you simply could not before.

The full power of automated testing depends on OOP. You can create simple tests without classes. But the most valuable testing patterns require full use of Python's object system. This is why OOP comes first in pareto-optimal learning.

One reason automated testing is so valuable: it effectively reduces the complexity of the program you are implementing. This makes it easier to implement mid-sized programs, but the *real* value is to enable you to create extremely complex software that would otherwise be out of reach. It raises the ceiling of complexity you can successfully handle, and it does so by a *lot*.

Imagine you are designing a web application framework. You want view objects which represent the HTTP response, and a way to map incoming URLs to those views. You create specialized View classes for different response types, and create a class called WebApp to coordinate this configuration:

```
app = WebApp()
app.add_route("/api", JSONView({"answer": 42}))
app.add_route("/about", HTMLView('about.html'))
```

This object has a get() method to retrieve the HTTP response for a URL:

```
>>> app.get('/api')
‘{"answer": 42}’
>>> app.get('/about')
‘<html><body>About Page</body></html>’
```

There is enough complexity here that if you just start coding it, odds are high bugs will be missed by manual testing. Even worse, at this threshold of complexity, every new feature you add risks breaking something in an unexpected way. The only way to maintain full correctness is to exhaustively test after each change, which if done manually becomes extremely labor-intensive.

The solution is to start by writing a suite of automated tests, exercising the key functionality. Like this:

```
import unittest
from webapp import (
    WebApp,
    HTMLView,
    JSONView,
    )

class TestWebapp(unittest.TestCase):
    def test_route(self):
        app = WebApp()
        json_view = JSONView({"alpha": 42, "beta": 10})
        html_view = HTMLView('about.html')
        app.add_route('/api', json_view)
        app.add_route('/about', html_view)

        # The JSON object as a string
        expected = '{"alpha": 42, "beta": 10}'
        actual = app.get('/api')
        self.assertEqual(expected, actual)

        # The contents of about.html
        expected = '<html><body>About Page</body></html>\n'
        actual = app.get('/about')
        self.assertEqual(expected, actual)
```

Then it becomes straightforward to implement. You simply write code to make your tests pass. And when that is done, you feel full confidence that your program is working correctly. Here is one way to implement it (and make the above tests pass):

```
import abc
import json

class WebApp:
    def __init__(self):
        self._routes = {}
    def add_route(self, url, view):
        self._routes[url] = view
    def get(self, url):
        view = self._routes[url]
        return view.render()
    def urls(self):
        return sorted(self._routes.keys())

class View(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def render(self):
        pass

class HTMLView(View):
    def __init__(self, html_file):
        with open(html_file) as filename:
            self.content = filename.read()

    def render(self):
        return self.content

class JSONView(View):
    def __init__(self, obj):
        self.obj = obj

    def render(self):
        return json.dumps(self.obj)
```

The details of this are less important than understanding the pattern, which is:

- Fully specify the desired behavior of the complex system, by writing automated tests
- Rely on those tests to provide thorough feedback as you implement functionality
- Run these tests frequently during development, as it costs you no effort to do so, and immediately exposes bugs and holes in functionality
- Benefit from the test suite as you refactor the codebase, add significant new features, and otherwise make disruptive changes which threaten to break your program in unexpected ways
- In the end, you have created a rich and powerful program, with low stress, high confidence in its continued correctness, and little effort spent on tedious manual testing

This is why we say automated testing is a superpower. It simply puts you in a different tier.

Data Scalability

When processing large amounts of data, some programs are responsive, efficient, and rock-solid reliable. Others hog the machine's resources, randomly hang without warning, or even crash when you push too much data into them.

The difference comes down to *space complexity*. This is the area of algorithm analysis focused on effective usage of memory by programs. When designing an algorithm to process a large quantity of data, how can you make it use as little memory as possible, with a reasonable upper-bound that is independent of input size?

This matters because of how operating systems manage running programs. When a program creates a large data structure, it must request a block of virtual memory from the OS. But the job of the OS is to allow all programs to run, and prevent any one program from starving the others of enough resources to run. So the OS does not always give a program the full amount of memory it asks for. If a program asks for too much, the OS will instead "page to disk" - which means giving the program a block of pseudo-memory, written to and read from the persistent storage layer (the hard disk, SSD, etc.) instead of actual memory.

This shoves your performance off a cliff. The sluggish I/O speeds of disk compared to memory reduce raw performance tremendously, typically between 75% and 90%. It can literally make your program 10 times slower. If a human is interacting with your program directly, they will experience it to 'hang' and appear stuck. In extreme cases, to ensure other programs have the resources they need, the operating system will kill your process completely.

Those skilled at writing big-data programs learn to code in memory-efficient ways, fitting in some reasonable memory footprint regardless of total data input size. This is an essential skill for top performers. Big data is here to stay, for everyone; software processing more data than fits into memory is increasingly the norm, not the exception.

How do you accomplish this in Python? The key is a feature called *generators*. Many Python users have heard of this; you have probably seen the 'yield' keyword. But most do not know its depth, including:

- The coroutine model of generator functions, and its implications for algorithm design
- Generator design patterns, like fanning record streams in and out
- The Scalable Composability strategy for implementing robust, flexible internal data processing pipelines

This relates to first principles: of memory-efficient algorithms, which are independent of language; and of Python's memory model, along with its generator feature. The language-level first principles form a bridge to the higher-level, language-independent first principles in Python programs. You need both.

Consider this Python function:

```
# Read lines from a text file, choosing
# those which contain a certain substring.
# Returns a list of strings (the matching lines).

def matching_lines_from_file(path, pattern):
    matching_lines = []
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                matching_lines.append(line.rstrip('\n'))
    return matching_lines
```

This returns a list of strings. It creates a new data structure - the list - whose size scales up in direct proportion to the size of the input. If the text file is small, or few lines in the file match the pattern, this list will be small. But if the text file is large, with many matching lines, the memory allocated for the list may become substantial. It can easily cross the threshold where the process must page to disk, plummeting performance of the entire program.

Here is another approach:

```
# Generator function
def matching_lines_from_file(path, pattern):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

This is called a *generator function*, signaled by the ‘yield’ in its final line. Without going into details here, what this accomplishes is to transform the algorithm completely. Rather than a list of all matching lines, it is now producing matching lines one at a time. Typically you are using the result in a “for” loop or something similar, where you only process one at a time anyway. This means your memory footprint is now the size of the longest line in the file, rather than the total size of all matching lines. This holds true no matter how large of an input file you feed it. Suddenly your algorithm is exponentially more memory efficient, with none of the problems plaguing the first version.

The best approach is to proactively infuse all your Python code with memory-efficient Python generator functions, as a standard practice when developing. This naturally makes your programs perform their best, regardless of hidden memory bottlenecks you missed, and regardless of unexpectedly large input sizes.

Metaprogramming

Most Python code operates on data. You can also write Python code that operates *on other Python code*. This is called "metaprogramming", and is one of highest-leverage things you can do when creating software.

Coders who use metaprogramming rapidly create tools and components that other developers could not create in a lifetime. It like the difference between riding a tricycle, and flying a jet. You not only "travel" faster; you go places you simply could not reach at all with the lessor vehicle.

You can find examples of metaprogramming in the source code of the most successful Python libraries: Django, Pandas, Flask, Twisted, Pytest, and more. It is no mistake that extremely powerful tools rely on metaprogramming.

As an example, imagine this function which makes an HTTP request to an API endpoint:

```
import requests
def get_items():
    return requests.get(API_URL + '/items')
```

Suppose the remote API has an intermittent bug, so that 1% of requests fail with an HTTP 500 error (which indicates an uncaught exception or similar error on the remote server). You realize that if you make the same request again, it has a 99% chance of succeeding. So you modify your get_items() function to retry up to 3 times:

```
def get_items():
    MAX_TRIES = 3
    for _ in range(MAX_TRIES):
        resp = requests.get(API_URL + '/items')
        if resp.status_code != 500:
            break
    return resp
```

However, your application does not have just this one function. It has many functions and methods which make requests to different endpoints of this API, all of which have the same problem. How can you capture the retry pattern in an easily reusable way?

The best way to do this in Python is with a metaprogramming tool called a *decorator*. The code for it looks like this:

```
def retry(func):
    def wrapper(*args, **kwargs):
        MAX_TRIES = 3
        for _ in range(MAX_TRIES):
            resp = func(*args, **kwargs)
            if resp.status_code != 500:
                break
        return resp
    return wrapper
```

Then you can trivially apply it to the first function, and other functions like it, by simply typing @retry on the line before:

```
@retry
def get_items():
    return requests.get(API_URL + "/items/")

@retry
def get_item_detail(id):
    return requests.get(API_URL + f"/item-detail/{id}/")
```

Even better, the retry logic is fully encapsulated in the retry() function. You can raise the number of maximum retries, implement exponential backoff, or make any other changes you want in one place. All decorated functions and methods immediately reflect your updates.

Crucially, metaprogramming often allows you to implement solutions to complex and difficult problems in a way that people can easily reuse. This retry() function - and the @retry decorator it implements - is somewhat advanced code. But your @retry tool is instantly usable by everyone on your team. Your presence on the team has amplified everyone's productivity and systematically improved the reliability of the entire application, in a way that everyone recognizes right away.

Thought Leadership

Cultivating internal thought leadership may be the most important thing you can do for the long-term success of your team. It boosts the knowledge and mastery of everyone on your team, helps spread the qualities of your best contributors to everyone, and creates a culture of engineering excellence which outlasts the tenure of any current team member.

That last part is may be the most important. As a leader, you invest a lot of time, energy and budget into building a great team right now. But as the months and years pass, it is inevitable that its staff will change. People will leave for other opportunities, be let go, or get promoted out of your team; new folks will come in, for all sorts of reason. Since we know this will happen, it makes sense to plan for it, and ask if what we can do to ensure we keep great people as long as we can, and attract even better people in the future.

That is where “thought leadership” comes in. By this, I mean a very specific skill stack. One which is entirely *non*-technical, and focused on the “soft” skills of professional communication,

Content Creation

Writing is the foundational skill of thought leadership. Everything else builds on it.

Training your team members in simple writing skills is easy to do. It pays great dividends for them in their professional (and even personal) development. It is even more rewarding for you as team leader. It immediately starts improving quality of work team-wide, and rapidly elevates the knowledge and wisdom of the team as a whole.

This operates at several levels. The most basic is that when we put our thoughts into writing, that naturally creates more clarity and understanding in our minds. We realize new distinctions and insights. Many have observed this, and you probably understand this already based on your own experience.

At the team level, writing helps propagates knowledge and culture. When top performers write, it can be consumed by more junior team members to elevate them faster and more completely. This is especially true as staff changes over time. When people leave the team, the knowledge in their head leave with them; everything they wrote stays. And when new people join the team, they can access years of wisdom by reading.

As engineering teams increasingly use generative AI tools, writing becomes important in a new way. Those who know how to write well, with completeness, clarity and precision, are able to prompt LLM-based tools in more productive and powerful ways. This is certainly a separate ability from writing code; many excellent coders struggle to use AI tools well, after all. The disparity may in large part be due to natural-language writing skills. By learning to write well for their peers, your teammates may well become able to get more out of AI tooling as well.

Another way AI relates: as your team is writing more, the collective volume of their output can become an input corpus that other AI tools can ingest. One limitation of current agentic coding tools is they are only looking at the codebase itself, and are ignorant of much important context that is only in the head of your team. The more they write, the more that critical context becomes stored in a form that can at least in principle be accessed and leveraged by current and future tools.

Implementation Path

If you want help doing everything described in this document for your own team, schedule a call with us here.