Chapter 0 - TLDR

An abundance of academic programs do a good job training data scientists to write “academic code”, or code that is good enough to pass a homework assignment. If you’re transitioning to industry, I strongly advise you to upgrade your coding practice. This book is written to fill this gap in data science training, with a particular focus on industrial engineering / operations research (IEOR) graduates.

Chapters 1 through 8 are a mix of allegories, anecdotes, opinions and war stories. There are a few technical terms thrown in, but nothing that can’t be easily googled. These chapters are designed to be something you can read piecemeal on your phone.

Chapters 9, 10 and 11 will require you to be sitting with your computer, ideally with a second monitor. It is in these chapters that I am teaching you how to write professional-grade IEOR code, or at least my version of it. A former colleague of mine, whose brain has been permanently infected by my ideas, suggested that perhaps one out of ten people who read Chapters 1 through 8 will actually go on to complete my programming exercises. That’s fine with me; the Navy Seal course has a much higher rate of attrition. That said, Chapter 10 does provide an easy off-ramp for someone who wants to do no more than get their feet wet with Python.

More to the point, the intended audience for this book is people who are analogous to Navy Seals within the purview of logic and mathematics. Anyone who graduates with an IEOR degree, or with the academic credentials to launch a data science career, has more than enough mental horsepower to write professional Python code. The barrier to doing so is attitude, not aptitude. As I discuss in Chapter 3, the leadership in my field has lowered the bar so much that low-grade code is tolerated, so long as it performs advanced mathematics. This book is my attempt to correct that mistake.

If you think this goal presumptuous, then I could hardly blame you. My objective for this book is to discuss problems that strike me as meriting more attention, and to share the solutions I have used to address those problems. I don't expect the reader to accept my opinions as gospel, or really as anything more than informed opinion.

The singer Leonard Cohen was fond of the Zen saying “Like the pebbles in a bag, the monks polish one another.” Steve Jobs shared a similar analogy about how teammates bump against each other with conflicting ideas, and through their disagreements, come to a deeper understanding. This book is written in that spirit.

Chapter 1 - Introduction

This book serves two purposes. The first is as a Python programming companion to the *Supply Chain Network Design* book (SCND). I am one of the four authors of *SCND*. Of the four, I was the strongest programmer, with the weakest understanding of real-world applications. As we worked on *SCND,* I felt that my area of expertise was being given short shrift. There were no programming examples in our text whatsoever.

One of my co-authors, Mike Watson, and I rectified this deficiency shortly after *SCND* was published by creating a companion text with the somewhat ungainly title *A Deep Dive into Strategic Design Programming.* For brevity, let's call this the OPL book, since it dealt entirely with the OPL programming language. I have to admit, I wrote none of the code for the OPL book. Rather, I just coached Mike as he dealt with the OPL IDE and strung together the mod and dat files. I have an excellent excuse for my dereliction: by the time the OPL book was well underway, I was already in love with Python.

Since we wrote the OPL book, Python has emerged as the lingua franca of data science. When I was abandoning Mike to the tender mercies of OPL, this outcome wasn't obvious. I was spending as much time as possible writing Python code in the hope that gaining such expertise would land me an enjoyable job. But now (in the Spring of 2022), Python skills are essentially a prerequisite for a career as a data scientist. The value in re-writing our OPL book as a Python based-text is obvious, and has been for some time.

The second purpose of this book requires a bit more of a backstory. In the fall of 2018 I was hired by Opex Analytics, an AI startup specializing in supply chain and founded by Mike Watson and Ganesh Ramakrishna. During the interview process, Mike's main concern was my habit of creating unlovely PowerPoint slides. Ganesh was unfazed by this problem (which was easily solved by assigning a colleague the task of beautifying my slides before any public presentation) and instead proposed a more positive vision of how I could help Opex. He wanted me to teach his data scientists how to write industrial code.

This was a role I was eager to fill. The Opex data science team was already standardizing around Python, the language I had become obsessed with five years earlier. In fact, I had created an open source library, ticdat, precisely for the purpose of writing industrial Python code for advanced analytical engines. I thought my background in general, and ticdat in particular, would be a perfect fit.

Over the course of the next 30 months or so, I attempted to fulfill this role more with "show" than with "tell". I worked on a few different projects, always with the same caveat. I wasn't going to commit code to a repository that failed to meet some basic standards for organization, but I was more than happy to do this organizational work myself. Typically, this meant incorporating ticdat. However, my requirements were more nuanced than "We need to use ticdat". In fact, ticdat was simply a means to an end, and my goal was to address structural problems I saw in the typical Opex project. Eventually, I coined the phrase "Tidy, Tested, Safe" to summarize my pedagogy.

Hence the second purpose of this text. This is the book I wish I had written before I was hired by Mike and Ganesh. This is the “Tidy, Tested, Safe” book.

These dual goals inform this book's organization, and specifically the differences from the OPL Book. The initial chapters of this book deal with the three goals of “Tidy, Tested, Safe”. These are software engineering goals that transcend any particular problem (or really, any given programming language). Chapters 2 through 9 discuss why I consider these three goals to be so important, and the danger I see in advanced analytical engines that ignore them. This section contains very little code, but it does reference public ticdat examples that can be found at the ticdat homepage.

A sophisticated programmer could simply study ticdat and then skip on to Chapter 10. If you feel comfortable moving from this through to that, then the programming exercises should come naturally to you. Of course, I hope you read the entire book. I think you will find Chapters 2 through 9 informative and entertaining (not to mention easy to read on your phone). If nothing else, this material might help you interact with colleagues who are themselves reluctant to build industrial code.

Chapter 10, in combination with the tts_scnd_course_files, translates the material in the OPL book into the world of Python and GitHub. Bear in mind that this section is collectively much smaller than the OPL book. There are a few reasons for this.

1. OPL is very much a niche language. As a result, Mike and I felt that we couldn’t assume our readers would necessarily be well served by the OPL training material available to them outside our book. It was pointed out to us a few times that we were repeating OPL instructions that could be found elsewhere. This was by design. Since the OPL community was so small, we felt the responsible thing to do was err on the side of redundancy. With Python, the situation is very different. Python is the third most popular programming language in the world. The internet resources for a student looking to answer a subtle question about the language itself, or about the tools available to support it, is easily 100 times bigger for Python than OPL. Thus, we can omit a lot of low level explanations regarding things like language syntax or debugging, and simply provide the reader with helpful Google search terminology.
2. This book is designed to be useful for a bootcamp course in Python based applied optimization. As a result, it provides programming problems with only partial solutions. The reader is expected to flesh out each example problem herself using a public GitHub repository. The URL of this repository can then be submitted to an instructor in order to receive a bootcamp certification. Details for this process, and options for how to deploy your work as an app, will be maintained on the Tidy, Tested, Safe wiki page here. While I am obviously excited by app-deployment, I want to emphasize that the lionshare of the material I cover in this book is independent of the presence or absence of an app-deployment platform. Your primary goal is to create a well structured, well tested Python package. This package will be built incrementally, according to the principles of Agile development. It will be bullet proofed against the vagaries of real-world data inputs.

These are the minimum requirements of a professional work product, and I think you will find deep satisfaction in learning such craftsmanship. That said, there is also a thrill in seeing your engine brought to life via a web based GUI. For some of us at Opex, there was something remarkable about creating a proper library, geared for re-use by other programmers, and then summoning a business-user friendly application with just one button click. I hope you can share in this excitement.

Chapter 2 - Two Sermons

The purpose of this chapter is to inspire you to get “Tidy, Tested, Safe” (TTS) religion. While all the TTS chapters are less technical than the network design chapters, this one is the most evangelical. That said, even if you are already motivated to learn TTS, I encourage you to read this chapter anyway. You will likely end up sharing code and repositories with teammates, and a standard isn’t a standard unless everyone respects it. You might very well find yourself cast in a preacher role yourself someday, and the sermons here might help you convert others.

Lined-Up Pawns

When I was in junior high I was a member of the chess club. This was in a small town in Vermont that called itself a city. We didn’t compete with anyone other than each other. We played a lot of chess, ate sugar, and talked trash.

Our chess “coach” was a teacher willing to supervise us (in the loosest possible sense) while he graded papers. He didn’t really teach us chess in any direct way, but he would occasionally insult someone’s game while stretching his legs. Our “Tidy, Tested, Safe” analogy begins one afternoon when Coach looked over my position and pronounced it hopeless.

I took umbrage at Coach’s assessment. My opponent and I had traded several matching pieces and nobody was in check. As far as I could tell, we were tied.

“No, no, no, Cacioppi. You’re doomed. Just come talk to me after you lose”.

Thirty minutes or so later I was in Coach's office (a back room where they allowed him to smoke his pipe) and he explained what had happened to me.

“See, your pawns were all lined up. The pawns want to be able to protect each other, diagonally. They can’t do that when they’re in the same file. You have to be careful when trading pieces that you don’t end up with more lined-up pawns than your opponent. Every time you do that, your pawn structure weakens, and your opponent will end up with an advantage in the end game”.

I pondered this advice during my walk home. (5 miles, in the snow, uphill both ways).
Up until now, my chess friends and I had always thought of “winning” as a matter of accumulating a point advantage in terms of captured pieces. A queen was eight points, a rook five, knights and bishops three, and a pawn one. When we traded pieces, it was considered a wash to trade pieces of equal value. But now Coach had challenged this analysis. Perhaps there was an advantage in trading a bishop for a knight, or one rook for another, so long as this trade lined up one of the opponents pawns. More importantly, since I was the only who had braved the pipe smoke, perhaps this insight could lead to dominance over my fellow nerds.

At this point in the story, my older son, Jasper, will start to laugh at me. “Didn’t you guys just study chess strategy?”. A few years ago, Jasper got tired of losing chess games to me. After a

few weeks of intense online tutorials, he quickly developed a better game than his old man. This idea that lined-up pawns could constitute an insider advantage seems ludicrous to him now, as it probably does to any reader who grew up with Google.

But this story is set in Vermont, in the 80s. There was no internet. There were very few computers. Research was done in a small public library with limited hours. Our chess club consisted of an unusual collection of very smart kids (I can think of three current professors off the top of my head), but what we knew of chess was what we taught each other.

Over the next few weeks, Coach's advice allowed me to achieve a "Barry Bonds on steroids" level of domination over my peers. Armed with a new style of play that nobody could quite decipher, I quickly became the best player in the club. All my wins followed the same basic pattern. I would immediately accept any equal-value trade my opponent offered, so long as it lined up one of his pawns. I would create elaborate attacks designed for the sole purpose of creating a same-for-same, lined-up-pawn trade. Not seeing the danger in these trades, my opponents never tried to defend against them. And once my opponent lined up even one of his pawns, the game would move even faster. Bolstered by a stronger pawn structure, I would rapidly trade equal pieces simply for the sake of clearing out the board. I knew something none of the other club members did: in the barren field of the end game, a single lined up pawn could spell doom.

Of course, the rest of the story is entirely predictable. The other nerds figured out what I was doing. My glorious reign lasted but a few weeks, and soon enough, all the chess players were fussing over lined-up pawns. I resumed my previous place in the pecking order. "Smart kid, like the other smart kids".

But a deeper lesson remained. Even a collection of very smart people can overlook a fundamental component of strategy. It's not always enough to be smart. It's not always enough to share ideas with smart peers. Sometimes you need the insight of someone with deeper experience to change what everyone is doing for the better.

Tidy, Tested, Safe was the lined-up pawns of the Opex Analytics data science team. When it began, I was the only one doing it. A rumor spread that my projects went more smoothly, with fewer headaches, and with easier onboarding of teammates. Eventually, it became our de-facto standard, enforced by peer pressure. The Opex data scientists were all high IQ, hard working people. They were well versed in applied mathematics and algorithmic analysis. But I was the first person to explain to them about lined-up pawns.

Seat belts and sober driving

Trigger warning: this whole sermon might offend some people. I'm making an analogy here between something truly dangerous (irresponsible driving) and something merely dangerous to your career (irresponsible programming). If this sort of discussion upsets you, please skip ahead. You've probably gotten the gist of it from this trigger warning.

When I was a kid, drunk driving was normal, and nobody wore seat belts. (And I promise[1] this will be the last of the “when I was a kid” stories). This all seems a little bonkers now. Whenever I talk about my childhood with my own children, it reaffirms their belief that back in the 70s and 80s, everyone was some combination of crazy, evil and stupid. That assessment is of course unfair, but I do think society really has progressed. The good old days were mostly terrible.

The drunk driving culture is the hardest to evoke for a modern reader. Nearly all of my friends had some sort of “when my parents drove drunk” story. My friend Big Pete (I was Little Pete) had an entire stand-up comedy routine about how he’d get scared and make a sudden escape from a Sunday drive. “Hey-there’s-Joey-I’m-going-to-go-play-baseball-bye-Dad!” There went Big Pete, leaping from the car as it slow-rolled through a stop sign.

My story is more of a happy memory. Our car got stuck in a ditch coming home from a get-together with another family who lived out in the boonies. It was a clear night and we had to walk for an hour or so before some passerby gave us all a ride. I recall it as a fun little adventure. We walked around and looked at the stars and then met some nice strangers. As an adult, my mom told me it scared my dad enough to change his ways.

My uncle, tragically, got the memo too late. He gravely injured a family with his drunk driving. He was technically under arrest when I visited him in a hospital during my freshman year in college.

None of these people, my uncle, my dad, Big Pete’s dad, were bad people. They held down jobs, told funny stories, and coached Little League. They drove drunk all those years because of the way people deal with risk. Their predominant experience with drunk driving was that nothing particularly bad happens. They didn’t go out of their way to drive drunk, they just did it when doing otherwise would have been inconvenient. Life was good, and drunk driving was part of normal life.

While drunk driving was widespread, but also atypical, using a seat belt was almost unheard of. Nobody put one on, ever. If you fastened your seat belt, it was a passive -aggressive way to insult the driver. (<Click> - “What, you think I’m too drunk to drive?”). Eventually, the adults did become converts to seat belt wearing, but that was mostly in response to its being mandated by law. However, my friends and I did start wearing seat belts before our parents did. We started wearing seat belts after taking drivers ed.

Before I took drivers ed, I assumed it would be something like tennis lessons. The instructor would demonstrate how to drive, let you drive a little bit, and then correct your behavior.

As it happened, that was only a portion of the drivers ed course. The bulk of the course was cultural deprogramming. The instructor preached to us, day after day, the virtues of driving sober and wearing seat belts. It wasn’t just preaching: there was some sort of machine you’d sit in that would simulate an accident. There were gruesome pictures of car crashes. There were

[1] Don’t trust me, I will break this promise.

tearful videos of drunk driving survivors. While there certainly was some time spent behind the actual wheel of a car, drivers ed was essentially a government run indoctrination campaign, convincing the youth of our community to deviate from the example being set by our parents.

By and large it worked. My peer group would certainly drink (illegally of course) but we'd arrange to all spend the night together, or to use a designated driver. We religiously wore seat belts. We'd even mock our seatbeltless parents when we rode with them, and sometimes shame them into buckling up.

The programming practices of the Opex data scientists bore a lot of resemblance to the car safety habits of the adults of my youth. Both were examples of responsible people behaving irresponsibly.

To be clear, I don't want to insult the Opex alumni with this analogy. My peers and I were raised by a random collection of normal adults, whereas the Opex data science team was a carefully curated selection of exceptional adults. But they were exceptional adults who would occasionally drive metaphorical cars into ditches and then wander around in the dark looking for help. Projects went sideways[2] for reasons that could have been easily prevented. Nobody ended up in the hospital or in jail, but I also think we had some luck on our side. When I worked at LogicTools[3], a large company once threatened to sue us, and only backed down when we proved beyond a shadow of a doubt that our math code, documentation, and tech support were all consistent and correct. Ganesh was wise to ask me to teach industrial programming to his data science team. The typical Opex data science project was not up to the standards I enforced at LogicTools.

It's worth pointing out that so long as Opex functioned purely as a consulting company, perhaps those standards would be more trouble than they were worth. This is a preview of the Rorschach vs Ozymandias debate I'll explain in the next chapter. But if Opex Analytics were to pioneer the practice of "templated apps", then they would need to improve their ability to revive and refashion past projects. They would need to become proficient at upgrading a quick one-and-done into something with true intellectual property value. And to do that, they would need the skill set and the standards I developed at LogicTools.

I developed those standards not because I was smarter or more responsible than the typical Opex data scientist. I developed them because I was thoroughly indoctrinated by years of computer science training and experience. Even among CS people, I was something of a zealot for defensive programming. My first serious programming course was taught by Chip Elliot, who was a guest instructor with a distinguished industry background. Our capstone project required diligent assert statements for every subroutine, and I happily complied. But even without Chip's guidance, there was peer pressure from shared homework assignments, from subcontractor

[2] My definition of "sideways", in the context of programming, is somewhat broad. The typical client feedback was undeniably positive.
[3] LogicTool was a small supply chain analytics company. Mike, Ganesh, and nearly all of the Opex leadership team worked there at one time or another.

jobs, and from nerdy trash talking. I didn't naturally start writing Tidy, Tested, Safe code. Tidy, Tested, Safe was programmed into me.

In truth, I was always more afraid than I was responsible. I was scared of the car going into the ditch, or worse, with my programming projects. I was wise to be scared, and I was fortunate there were leaders who scared me so early in my adult life. In the spirit of Andy Grove's "Only the paranoid survive", I believe this: "only the frightened write solid code".

Chapter 3 - Tidy Tested Safe Overview

By now, you probably agree that optimization projects can easily go awry. It's not enough to simply hire smart, well-educated people. Between industry and academia, there is clearly a training gap that fails to prepare students for industrial projects. Bear in mind, I am not so egotistical as to think I can simply fix this problem with a short book[4]. But I do think I can help. My goal here is not so much to revolutionize the teaching of optimization, but rather to give optimization practitioners a perspective that appears to be missing.

The Status Quo

Before I get to solutions, let's discuss the source of the problem. Optimization isn't typically taught as part of computer science, and so its pedagogy is fully divorced from the principles of software engineering. Industrial engineering/operations research (IEOR) students are not required to take a course like the one Chip Elliot led, nor to be taught by someone with extensive professional experience. Homework assignments involve implementing clearly defined algorithms, and are run on pristine data sets. The idea that users might do inconvenient things like change the project requirements on the fly, or feed malformed data into your program, is rarely even discussed.

This is all compounded by the same problems that afflict academic computer science. The technology moves quickly, and thus the curriculum needs to hit a moving target. The nature of school is to provide short assignments, often done individually. This tends to leave students ill prepared for the challenges of repurposing code, adding new features to a project that has long since gone stale, or onboarding new teammates.

I don't want to point fingers exclusively at academia. Professional optimization libraries are provided by three big players - Gurobi, CPLEX, and Xpress (which is part of FICO). Historically, I would say all three have been the equivalent of the automotive companies before they added public safety outreach programs. Their educational efforts are largely documentation for their tools, and their demonstration code is toy examples (sometimes glorified toy examples) that ignore the real-world challenges of industrial code.

Of the three, my favorite is, by far, Gurobi. I'm flattered they use my endorsement as part of their promotional material. They were literally the pioneers (pyoneers?) in promoting an algebraic and intuitive Python API in lieu of an engine-specific modeling language like OPL or Mosel. I was an early adopter of Python because of their example, and for that good advice I will always be grateful.

That said, I don't think I am unreasonable in asking Gurobi, as the industry leader, to lead. They have more influence over the world of optimization than any other company. I think they are wise to encourage optimization developers to enter the modern world and craft their solutions

[4] I'm lying. I actually do think this.

with Python. However, I think they need to take more care to avoid the implication that “If you got it to run, then you must have done a good job”.

Python doesn’t hold your hand and nudge you into sound practices to the same extent that the legacy optimization-based modeling languages did (a point I will itemize in the very next section). When moving past the modeling languages, you shouldn’t ignore the lessons learned from the modeling languages. Gurobi (but of course CPLEX and Xpress as well) don’t address this with their Python examples.

That said, Gurobi has listened to my complaints and is now taking on more responsibility in this area. They recently included `ticdat` in their example section, and there is a decent chance that you’re reading this book as a result of their endorsement.

We Are All Consenting Adults

Now that I’ve pointed fingers at an academic community that has provided me with nearly all of my employment history, and a specific company that’s the clear leader in my industry, I think it's time to also spread blame to my favorite programming language: Python itself. Python has a dual quality that brings with it certain dangers. It is a flexible language that allows for easy onboarding of new programmers. Very little is explicitly prohibited, and learning by experimentation is encouraged. In lieu of rigid rules, Python relies on a set of conventions and principles to guide the development of serious projects. This is summarized in the phrase “We are all consenting adults”.

Historically, optimization training has been done with dedicated modeling languages like OPL or AMPL. In some sense, the limitations of these modeling languages have been a strength. For example, there is no need for a “separating model from data” example in AMPL, as nearly every example demonstrates this principle. Separating model from data is explicit in the language. It is demonstrated every time an AMPL solution is implemented as a .mod/.dat pair.

We can go down the `ticdat` template this way. Should the solution be based around an exposed solve function? The .mod file itself is effectively just such a subroutine. Should foreign key references be validated? That’s accomplished using AMPL set definitions. Data types checks? Row predicate checks? These can both be enforced in the param definitions.

All this structure that was once built into the language is now gone when we transition to Python. As a general purpose programming language with a rich ecosystem, Python brings tremendous advantages. But it also carries its own set of responsibilities, which will likely not be observed by new practitioners unless they are taught.

As a bit of a sidebar, Git (the open source foundation of GitHub) has similar characteristics to Python in terms of relying on convention. Please don’t kid yourself into thinking that simply because you made a commit you’re done learning about Git. In these pages, we will discuss standard GitHub protocols that your team ought to observe. Serious projects follow conventions

that are specific to GitHub, conventions that are specific to Python, and conventions that interweave the two. The book will introduce you to all three.

A Modest Proposal

As a society, we seem to have arrived at the following consensus when it comes to cars.

Wear your seat belt. Don't drive drunk.

Do these two sentences comprehensively address all the problems of automobile safety? Of course not. Driving is a thousand little things, any one of which can result in disaster. That said, nobody really disagrees with these two simple rules. Not so long ago, they represented a challenge to the status quo. Now they are so deeply ingrained in our culture as to be considered unworthy of discussion.

In a similar manner, I'm going to make a modest proposal in an attempt to achieve a broad consensus regarding optimization projects.

Write normal Python. Automate the tests. Protect against dirty data.

My personal practice is to use `ticdat` to help achieve these goals. At Opex, I never found a project that followed this advice without relying on `ticdat`. I am certainly sympathetic to the notion that every library has its own aesthetics, and there might be something about the `ticdat` style that some people find unappealing. What I find unacceptable is the idea that one can use personal taste as an excuse to reject sound architecture. The `ticdat` examples exist to provide concise example code to illustrate my three foundational rules. I would happily link to other demonstration code fulfilling the same function with a different library, but no such library seems to exist. It's my opinion that, as of this date, the only practical way to follow these guidelines is to either import `ticdat` directly, or to create your own equivalent library.

One of the goals of this book is to start a conversation. If you don't like the `ticdat` template, please propose an alternative.

The Danger of Deontology

The next three chapters will dive into each of these rules in more detail. But before we do that, let me clarify my broader philosophy of programming. I am a utilitarian, not a deontologist.

If "deontology" is a new word for you, then you're in good company. I'm still not entirely sure how to pronounce it. But I do think I understand the concept. A believer in deontology will follow rules because he believes the rules themselves define ethical behavior. A utilitarian uses rules simply as useful advice to achieve the best possible outcome. If you are familiar with the *Watchmen* universe, Rorschach is a deontologist, and Ozymandias is a utilitarian.

Of course, in *Watchmen* both Rorschach and Ozymandias are anti-heros at best. They both demonstrate profound indifference (in different forms) to human suffering, and in doing so, lose their essential humanity. But in the realm of programming, utilitarianism poses no such risk. Lengthy projects can and should be abandoned when they cease to be of value. Code has no feelings, and programmers should avoid a sentimental attachment to their work.

To this end, I disavow any moral high ground by promoting Tidy, Tested, Safe. What I call "irresponsible programming" is meant to imply an indifference to (or more commonly an ignorance of) tangibly unpleasant outcomes. It has been said that a programmer should force himself to write good tests the way Superman forces himself to work a 9-5 job. I disagree completely. A programmer should write good tests because, for any project past proof-of-concept, those tests will likely save both his time and his reputation. A project with solid unit testing runs smooth, and smooth is quick.

If you can bear with one more war story[5], I'd like to illustrate this utilitarian-deontological distinction with a human resources experience I had at LogicTools. One of my colleagues, Aemon, was fairly new to the team. He had less than two years of workplace experience prior to joining us. At the time of this story, Aemon had been only working on one thing: the map.

Being a mapping programmer at LogicTools brought its own set of distinct challenges. The map was somewhat separated from the rest of the code base. The bulk of the code for one of our applications involved uploading data and then generating solution reports via the solve process. The map would then be invoked to visualize the reports. Line by line, the mapping programmer actually wrote a relatively small amount of code. The challenge of the job was understanding the mapping technology and using it to create the correct visualizations. The job required someone to be more of an explorer than a builder. The mapping technology was very difficult to master, and very powerful in the hands of a master.

At one point, a different colleague, Roger, invited me to lunch to discuss a burgeoning problem. Over burritos, Roger explained his concern. Aemon's maps seemed to be too buggy. Although about the same age, Roger had been with us since before he graduated college, and was well indoctrinated in my "bugs waste time, go slow to go fast" mantra. I was also the senior programmer, and no shrinking violet. Shouldn't I do something to straighten Aemon out?

In this case, I declined. I thought Aemon's style was perfectly suited to his task. While he sometimes plugged tab A into slot B, he added functionality at a rapid pace and had greatly expanded our map demo. Moreover, his bugs, while perhaps somewhat embarrassing, never really caused any harm. They were easy to spot and easy to describe. Something on the map would look goofy, and a quick review of the underlying report table would clarify the problem. So long as the reports themselves were correct, then the bugs in the map would be merely cosmetic. Users never got angry, and could always avail themselves of other aspects of Aemon's expansive range of mapping options until the bug was fixed in the next release. Nobody went to the hospital, or to jail, and nobody even spent more than a few minutes

[5] I'm lying. This is not my last war story.

wandering around in the dark. Precisely because the rest of the team worked slow and steady, the smart move was to allow Aemon to work fast and loose.

This is just one example of “Ozymandias, not Rorschach” that I’ll provide. (But it is the longest). As I work through Tidy, Tested and Safe in detail in each of the next three chapters, I will be sure to give Ozymandias his due. For each rule, I’ll provide some guidance as to when it should be bent, or even broken.

Chapter 4 - Tidy

*This chapter assumes you know what unit tests are. Don't worry, you can skip ahead to the **Tested** chapter if the meaning is not clear from context.*

Of the three components of Tidy, Tested and Safe, the first ought to require the least motivation. Tidy largely amounts to "do it the normal way", or, "follow the best practices defined by the broader community". Of course, I did promise to be a utilitarian and not a deontologist. In that spirit, here are some practical advantages to being tidy.

- Any standard, even an imperfect one, will help your organization. When projects follow a consistent high-level pattern, team members can onboard more easily, and context switching between projects requires less cognitive overhead.
- The more "invented" your standard is, the harder it will be to achieve buy-in. The easiest form of documentation is done by the hive mind, and can thus be easily Googled by anyone. What's more, high quality employees will want to learn skills they can carry throughout their career. Building skills in Python (the third largest programming language) and GitHub (the largest source code control system) is clearly in line with both of these goals.
- Python and Git are both, quite literally, works of genius. You're probably not a genius. You probably shouldn't set yourself the goal of inventing something better than Python or Git. By inventing your own best practices, that's really what you're doing. Fully enjoying the benefits of Python and GitHub requires learning their conventions, understanding why these conventions were developed, and then following them as appropriate. By which I mean, nearly always. The broader standard should be your default, and exceptional behavior should follow only from exceptional circumstances.
- Troubleshooting is greatly enhanced by tidy code development. This point is less obvious than the prior three. If you don't have experience supporting a community of users, then you might not realize how critical it is to identify the version of the code they are running with 100% accuracy. Using Python and GitHub in a disciplined way makes this straightforward.

tts_diet

I created the `tts_diet` repository as a public demonstration of the organizational structure I advocated for Opex. It demonstrates GitHub and Python being used in concert, in a manner following commonly held best practices. This organizational structure is not my own original invention, but rather a close copy of references like this. I'm missing the requirements.txt file and a docs section, but only for the sake of expediency. A solid unit testing section is often a passable substitute for requirements.txt, and I know how to develop requirements.txt if need be. At any rate, you can see how simple it is to make a functional `setup.py` file. A well organized public interface, with well written docstrings, typically renders the docs directory superfluous.

In Python, you create a public interface for a project by organizing the code into a package, and then publishing the public components in the `__init__.py` file. Don't be distracted by the absence of internal subroutines and classes in `tts_diet`. This is just a demonstration repo using a small canonical example. A real project would surely see the need to add internal subroutines (and possibly classes). Those can be placed in `tts_diet.py` or in new .py files inside the `tts_diet` directory.

Broadly speaking, the structure demonstrated by the `tts_diet` repo is the "normal Python" way to organize large blocks of code into a coherent logical unit. It is the Python answer to the question "How do I manage something bigger than a little get-something-working trial balloon?" When reviewing this text with Austin Bren, an Opex colleague who had previously never written a single `__init__.py` file, he couldn't help but blurt out "Gotta be honest with you - it's a great feeling to look at code like this. Everything is well organized and trustworthy." That was my feeling as well when I first started to build complex optimization engines using Python best practices. Sharing that feeling was my primary motivation for writing this book.

Note that `tts_diet` demonstrates a one branch protocol with one tagged release. Both of these points will be discussed in detail in the next section. You can also jump ahead to the `tts_netflow_a` and `tts_netflow_b` repositories to see a richer demonstration of branch and release management.

Don't get hung up on the particular code idioms in the `solve` function of the `tts_diet.py` file. The purpose of TTS is to train you on the high level structural requirements of production code. My experience is that IEOR students take very naturally to Python syntax, and do just fine once they get down to the nuts and bolts of building equations. Really, this is true of Python programming in general. If you're the sort of person who excels at mathematics, then Python syntax will probably come naturally to you.

You might also wonder why the `pandas` package isn't used. This is just a matter of taste. I find the dict-of-dicts table format demonstrated by `tts_diet` to be more convenient for Mixed Integer Programming (MIP) code, but I see no value in arguing this point. If your inclination is to work directly from `DataFrame` objects, then just study the `tts_netflow_b` repository.

In my opinion, one doesn't need to be a Python expert to develop a project that follows the `tts_diet` template. For example, the `class` keyword is never used, and thus a developer is not required to study object-oriented programming. The programmer does need to understand how to define a function, and needs to have some awareness that functions are first-class citizens (i.e. that a function can itself be passed as an argument to another function). You need to understand the purpose of the `__init__.py` file. Since you likely want the ability to invoke your package from the command line, you need to implement a `__main__.py` file. You will need some knowledge of the import mechanism, and how it relates to package development. In short, you need to be a Python intermediate, but not necessarily a Python expert.

What about Python beginners who want to develop advanced analytical engines? My advice is to become an intermediate as fast as possible. I don't believe it's possible to write industrial code with only a beginners level knowledge of the actual programming language. This strikes me as axiomatic, and likely to be true for every programming language. If you want to go to the Olympics in synchronized swimming, then you first need to become a strong swimmer.

GitHub Branch Management

There are a variety of GitHub branch management strategies, but they all have one thing in common - branches are "born to die". A fairly sophisticated branching strategy is visualized here. You can see that there are only two perpetual branches - develop and main. (Standard conventions have recently replaced "master" with "main"). Other branches have a limited lifespan, and are killed when they are merged into develop (release and hotfix branches are killed when they are simultaneously merged into develop and main). Develop will occasionally merge into main, but it is never deleted. A nearly identical two perpetual branch protocol (differing largely in semantics) is described here.

Are two perpetual branches needed? No. A simpler one-perpetual branch strategy is documented here. I have worked with teams that followed both two-perpetual-branches and one-perpetual-branch protocols. There are strengths and weaknesses to both approaches. I would say the two branch protocol makes more sense as the team gets bigger, and as the run time of the complete unit tests get slower. For the one-branch strategy, it is important that the unit tests always pass on the perpetual branch. For two branches, this burden is only strictly observed for the main branch. It is a deep cultural failure of the team to make a release that doesn't pass the unit tests, and it is assumed that the unit tests will be of high quality.

It is also important that all releases be tagged, and that no two tags share the same `__version__` for the package. It doesn't really matter when or how the `__version__` is incremented, so long as the `__version__` of a given tag is consistent with the name of the tag itself. Of course, we're not barbarians. We will never deploy code to a client that isn't tagged for release.

Unfortunately, within the optimization space, it is not uncommon to encounter deployment tools (aka app building platforms) that encourage (or even require) an app developer to edit the app after it's been deployed. Since such editing is not reflected in the tagged version of the source code, such functionality will in fact do active harm to your ability to troubleshoot client experiences.

Remember, your code "lives" in GitHub. If building an app means "upload the GitHub code, and then perform twelve additional steps", then your deployment process is only thinly captured by your GitHub repo and it will be easy to foul up the redeployment of your app. As I said in Chapter 3, the status quo of IEOR app building is not yet well aligned with software engineering best practices. I believe such an alignment will eventually occur, and one purpose of this book is to hasten that happy day.

A Tale of Two Teams - Tidy and Untidy

It is a characteristic of untidiness that people simply don't like to throw anything away. This tendency needs to be resisted, both with the code itself and with the GitHub branches. Commenting out code so it's never forgotten is sheer foolishness. Remembering old code is exactly what GitHub is for. Failing to delete a branch after it has been merged is similarly foolish. What is the point of your merge if not to allow you to delete the branch? This clutter does not help anyone who knows how to use GitHub, and sorting through it all will in fact waste valuable developer time.

The pandas GitHub repo right now boasts 26,250 commits. There are currently 15 branches and 131 release tags. Ten of those branches are quite stale, and are clearly identified with a prior release. They are most likely being kept alive in case a hotfix is needed, even for client code using a very old version of pandas. This demonstrates a high functioning team that you should emulate.

On the other extreme, a project I am (very loosely) supervising has 414 commits, 57 branches, and zero release tags. This is an example of exceptional people behaving irresponsibly. (This includes me, irresponsibly taking the easy way out and complaining here, instead of engaging the rest of the team directly). The project isn't particularly big, yet it already boasts a huge number of branches. My teammates learned enough GitHub to lay claim to being able to use GitHub, but they didn't learn enough to actually use GitHub wisely. To follow the example of the pandas team, and resist the temptation of my wayward team, does not require a significant upfront investment of time or unusually brilliant minds. It is largely a matter of awareness, attention and concern.

Tidy Python

The choice between colleagues who are untidy with GitHub but sophisticated with Python, or the reverse, is to my mind no choice at all. A weird repository protocol is a lot less damaging to a project's success than primitive coding practices.

When I say primitive here, I want to be very clear. This is very much an example of defeating smart kids with lined-up-pawns. The people receiving advanced IEOR degrees are clearly whip smart, but their coding practice is often akin to nerds whose knowledge of chess comes exclusively from playing against each other. As a result, there are only a handful of dysfunctional behaviors, and a brief description of each of them ought to suffice.

The most egregious is the avoidance of subroutines. A subroutine is the fundamental logical building block of programming. Whenever code reuse needs to be achieved, the only two real choices are to define a subroutine, or to use one of the looping structures. When faced with a problem like "How do I get the code to run on different data sources?", the answer is always the same: use subroutines.

One characteristic of subroutine deprived code is an anti-pattern I call “leftmost code”. That is to say, it’s a .py file (or loosely connected sequence of files) for which most of the logical code was never tab indented. To import such a file would be to actually read data or try to solve for a solution. As I said earlier, example files like this are leftmost code, but that is because Gurobi is trying to teach you how to use `gurobipy`, and not how to write professional code.

Professional code largely defines classes and subroutines on import. To some extent, static data structures might also be defined on import, but import is never the right place to do things like read user data. If command line functionality is required, then it is done through the `__main__.py` file of your package.

This brings us to packages themselves. Python became famous initially as a glue language, and of course people do find value in short scripts that are directly launched from the command line. That said, industrial Python programming is characterized by package based development. As much of your production code as possible should be organized into packages. (It’s sometimes convenient to package the unit testing code, but it's generally fine to share that section as “files-thrown-together”). Inside an organization, proprietary packages can be easily distributed as versioned .whl files, although there are other techniques. Developing a package requires so little overhead that I wouldn’t wait until the project becomes too large for its code to be maintained in a single file. I would simply set up a `tts_diet` like repository structure from the very beginning.

Still Reluctant to Package?

Unfortunately, you might have already been brainwashed that package development is for other people. Sure, pros like the Gurobi engineers are required to put their code in a package, but there isn’t much need for such engineering when building analytics for a small audience, is there?

This thinking, while sadly common, is entirely misguided. At the end of the day, you write code for other people to run. Thus, your code needs to execute on someone else’s computer. So how is that going to happen? There are only two choices. You can copy your code onto another computer, or you can install it there.

The entire history of software is telling you that installing it is better. But if you want me to make an argument, the most obvious one is that copying begets copying, and unmanaged copies lead to madness. If you send Fred `diet.py`, it is inevitable that he’ll copy it himself to whatever file location is most convenient for him. Fred will likely forget where he put all his copies. The process of then upgrading Fred to a new version of `diet.py` will be an exercise in futility.

On the other hand, if you send Fred a .whl file that installs the `tts_diet` package, then all Fred can do is `pip install` it. The .whl file itself contains the version information, which is also printed out by `pip install`, so Fred will always know which version of the code he installed.

But of course, if he ever forgets, you can always ask him to execute the following statement from the command line to find out.

```
python -c "import tts_diet; print(tts_diet.__version__)"
```

Moreover, you've made Fred's life easier by no longer giving him the obligation to find the `diet.py` file whenever he wants to run your engine. Fred can just execute the following statement from any directory in order to run the currently installed version.

```
python -m tts_diet -i input_data -o solution_data
```

If Fred rejects version n+1 and wants to go back to version n, that's easily accomplished by pip as well. This all assumes that Fred is a business user who just wants to run your engine from the command line. The benefits of a package are even more pronounced if Fred is a software engineer who wishes to integrate your engine into a live system.

The bottom line is this: copying is fine if there is only ever one version of your engine, but "just one version" engines only happen with homework assignments.

Is it Ever Smart To Be Untidy?

Banging out a short script for demonstration purposes is fine. The ticdat examples themselves largely consist of single files that are meant to illustrate a pattern. For industrial purposes, you are best off implementing that pattern as an actual package (i.e. mimicking `tts_diet`).

Your bias should always be using Jupyter notebooks for demonstration purposes. Don't fall into the trap of creating a .py file when a .ipynb file would have been better. This critique applies to the Gurobi and CPLEX examples. Their single file examples don't guide one towards a well-structured package, and in a few cases, they demonstrate the "files thrown together in a directory" anti-pattern. You should use their code to learn how to use their work product, which is, quite properly, deployed as a versioned package, just as yours will be. As a rule, when you want to teach someone how to use a package, or to illustrate a flaw in their code, you should create a notebook.

A Python mantra is "readability counts", but that doesn't mean that all code needs to achieve the same level of readability. The importance of readability grows with the size of your project, and thus it's natural to refactor the codebase as it grows. A common workflow is for some short and ugly proof-of-concept code to demonstrate how to use some new technology, which is then used for instructional purposes when this change is adapted into the more hygienic production code. Even when working on the production code itself, I often prefer to create "ugly but it works" on a feature branch, and then clean things up prior to merging into a perpetual branch. I think it's counterproductive to put as much effort into the readability of the testing code as you do the production code. The primary purpose of readable code is to reduce bugs, and bugs in the testing code tend to be easier to discover and less consequential if they remain hidden.

We should also discuss the relationship between readability and performance. Bear in mind, most programs will obey some form of the 80-20 rule. That is to say, your code likely has a performance bottleneck by which at least 80% of the run time results from at most 20% of your code. Thus the only noticeable run time improvements will come from performance tuning a small fraction of your code. It is for this reason that you should largely ignore run times when first getting your code to work. After you have achieved a reasonable milestone of functionality, back stopped with a solid test suite, you can then worry about performance.

For the sort of optimization problems we discuss here, that means first discovering whether the performance bottleneck is inside the core MIP subroutine, or in the Python code that creates the MIP and processes the solution. For the former case, you are dealing with the sort of mathematical subtleties that are well addressed by IEOR training. For the latter, you should use the `cProfile` tool (or something equivalent) to isolate the performance bottleneck within your Python code. In some situations, you might be able to vastly speed up run time with a highly targeted rewrite of just a few troublesome lines. In other cases, a bottleneck can be addressed by using a Cython based library like `pandas` or `numpy`. Regardless, `cProfile` will identify where prioritizing readability over performance is the obvious choice (likely nearly all the code) and where trading off readability for performance might be worthwhile (likely a small portion of the code).

Chapter 5 - Tested

Let me begin by clarifying what may have been a false impression. Everyone agrees we need to test code before releasing it. I didn't encounter a single Opexer who answered the question "How do you test this?" with "I don't." I am not implying that the IEOR academic community, or the three big companies selling MIP software, endorse the distribution of untested code.

In fact, the first well tested code I ever saw was during a very brief internship at an optimization company (LINDO Systems). In the midst of a whirlwind tour my guide typed in some short command and text started flying by the screen. "What's this now?" "This is the test suite. This is how we make sure a new version still works." Bang, bang, bang, bang, one after another, the LINDO core engine was solving a model, with the newly computed result cross referenced against the expected result.

Shortly after, I broke a promise and abandoned LINDO, in much the same way I abandoned Mike Watson. In this case, I was running away from FORTRAN and towards my new loves, C++ and the Standard Template Library. The STL was quite new at the time, and it wasn't easy to find an employer who trusted it. I solved that problem with LogicTools, where my employer didn't yet know enough to forbid it. By the time she raised the issue, the STL constituted half our codebase, and thus I was allowed to continue to use it[6].

But prior to all this, my introduction to the LogicTools network design engine went something like this. "How do you make sure a new version still works?" "I manually run it against three or four data sets, and then visually inspect the results". This was the answer I was given by the typical Opexer twenty-five years later. My response was the same in both instances. "There's no way I'm working like that."

So my first IEOR work product was organizing an automated testing suite for someone else's fancy math engine. There was no obvious sign this was the sort of work I was hired to do. I neither requested nor received permission. But I believed to my bones it was the most important thing I could possibly do.

The result, to quote John Prine "weren't much to look at." But it did run. It was fully automated. It contained at least ten data sets. The ones I added were small and completely artificial, and they served the purpose of exercising subroutines I found confusing. In lieu of a code coverage report, I would temporarily edit the production code with deliberate bugs to ensure the test suite was comprehensive (a technique I still use, as a compliment to the `coverage` tool).

By the time I was done, I had accomplished two important goals - I had created automated tests for the engine, and I fully understood the engine code itself. I then proceeded to refactor the engine code. That is to say, I made it more readable and easier to meaningfully change, while using the automated tests as protection against accidental breakage.

---

[6] This isn't the worst thing I've done. I don't even consider it a misdeed.

It wasn't until months later that I actually did something that a non-programmer might recognize as "work". A client had created a data set that was unacceptably slow. Everyone (all four of us) jumped in trying to solve the problem. I came up with a fix within a day of receiving the problematic data. I identified a bottleneck and implemented an improvement that involved presorting one of the tables. For this client, the code now ran roughly 100 times faster. My colleagues were amazed I had solved the problem so quickly. I didn't feel the need to explain that I had actually been working on it for months. The unsexy work of automated testing, followed by refactoring, was the foundation of my success.

A Testing Hierarchy

We all agree that testing is important, but we don't all test the same way. Here is the hierarchy with which I judge the various testing techniques I've seen.

- Manual testing and inspection. The LogicTools way when I was hired, and the method used by some (perhaps most) Opexers when I was hired. This appears to be the level of testing an IEOR professor expects for an academic project.
- Automated testing and inspection, but without using `unittest` or `pytest`. To generalize this past Python, automated testing that ignores the language specific best practice for automated testing. This was my first work product for LogicTools, and it was the practice of Opexers who were doing something more sophisticated than manual inspection.
- Arms length language specific best practice. This is what I demonstrate in `tts_diet` and it's been my testing strategy since I started using Python. I create a `unittest` based script, or series of scripts, that test my production code comprehensively. If I write more than one test script, I write a supplemental script that exercises all those scripts in sequence. I don't use nose, but that's just a matter of taste, as my tests are easily deployed on nose. I will use Jenkins, but I don't set up the Jenkins server myself, and the engineer who does will typically need to make a few minor changes to my testing code. Moreover, I use some slightly idiosyncratic subroutines to facilitate running only one test from a suite, and to force an unhandled exception to drop into the debugger. This is all to say, I don't hold up my testing practice as elite in the broader world of Python, but they are far superior to the Opex standard as I found it. (I would say something similar about my .whl files and packaging techniques).
- True language specific best practice. You can google "continuous integration" if you want to study the extent to which successful tech companies typically invest in quality automated testing. The goal of this book is to teach you to write code that would at least be recognizable as automated testing by a software engineer working for such a company.

test_tts_diet

The `test_tts_diet` directory should be largely self-explanatory. I encourage you to study it. Note that we use a `data` directory to store the input data sets to our engine. We use `ticdat`

subroutines to retrieve a complete data set from this directory. Similar subroutines were used to create the files stored in the `data` directory. Here, we demonstrate storing testing data in json format. Here is a run-down of the different file formats.

- Excel. Always a bad choice for GitHub. Excel files are there to make end users happy. Please use only pure text formats for GitHub serialization of testing data. As I said, there are `ticdat` subroutines that make format conversion dead easy.
- CSV. Not quite so bad a choice. Indeed, it is text. On the other hand, Wes McKinney wisely said "Whenever someone says 'I'll send you the data as a csv file' I always hear 'I'll destroy some of the data and then send it to you'". I would treat csv similarly to Excel and use it as needed to satisfy business users.
- JSON. Basically the gold standard. Nobody should ever complain, ever, about json files, not ever. Data (in particular, data types) are not destroyed, and it is a very widely understood and accepted format.
- SQLite files. This is a text file format that is supported by `ticdat.TicDatFactory`. What I like about it is that it tends to make forward conversions of testing data easy. Forward data conversions of input data sets will be discussed later in the book.

When not to unittest

For production code? Never. Setting up a testing suite is part of my muscle memory. It takes me no time at all. I view production code that lacks unit tests the exact same way I'd view an unbuckled occupant of a moving car. It's always wrong, and sometimes tragically so. When reviewing a project, the first thing I look for is unit testing. The second thing I look for is matching commits between production and testing. A solid unit testing practice should feel like double-entry bookkeeping. Every feature should always be addressed twice: the production code edit that adds the feature and the testing code edit that tests the feature.

Admittedly, the most stringent unit testing best practices have always struck me as more Rorschach than Ozymandius. There are teams that follow the dictum "Every production subroutine merits a dedicated unit testing subroutine, and the latter should always be written before the former". I don't do that and wouldn't require a colleague to do that. I'd join a team that enforced these rules, but only if it were an established company and not a start-up.

As I said earlier, I tend to look for two things in unit testing code.

1. What does the coverage report look like? A large proportion (usually more than 90%, and ideally more than 95%) of the production code should be exercised. The code that isn't exercised will typically be for truly exceptional cases, such as the `tts_diet` behavior if `gurobipy` is not installed.
2. What happens if I play gremlin and start introducing subtle bugs into the production code? Do they crash the unit tests or not? Ideally, it should be the former, no matter how tricky my gremlin. In other words, my unit tests should be powerful enough to protect against a gremlin programmer doing a bad job of refactoring production code.

Let me share the advice I was given when I asked a more experienced programmer how much I should test. The answer: “More than you expect”. Testing code represents real work that adds client value. Your projects are far more likely to struggle from too little testing than too much.

Chapter 6 - Safe

There is always push back to change, and promoting Tidy, Tested, Safe within Opex was no different. Those conversations tended to fall into three categories, depending on which aspect of Tidy, Tested, Safe was being discussed.

Objections to Tidy:

- "*This is just an argument over taste. It's like arguing over your favorite color, or chocolate cake versus apple pie*[7]". I would take this line of argument seriously, and point out the scale of the broader Python community, and also the deep programming knowledge of the people who developed the Python standards.
- "*If you can't convince me that what I'm doing is wrong, then it means I have just invented the new standard"* I never took this response seriously. My reaction here was to simply avoid using any of this programmer's work in a direct way, and to advise others to do the same. Someone who thinks they are re-inventing Python is useful only for proof-of-concept code. I.e. for code that demonstrates how to do a very narrow and specific task.

Objections to Tested:

- "*"I'm too busy to learn how to write unit tests"* You'll waste more time by releasing easily preventable bugs.
- *"My client will get mad if I take the time to write unit tests".* Your client wants well tested code, whether they say so or not.

Objections to Safe:

- "*Ummm… Okay you've got a point there. I guess my code should anticipate mistakes in data entry."* In my experience, Safe was the bullet point that sold itself. Everyone realized the users just couldn't be trusted to upload perfect data, and everyone agreed that prescreening for a broad category of data mistakes would surely save all parties a lot of time and grief.
- *"What about this type of data error? Can your ticdat methodology check for this?"* This was the best part of working for Opex. My prescreening skill set expanded greatly in response to a wide range of data integrity problems presented by Opex apps. In some cases, `ticdat` itself advanced (see here and here). In other cases, the team used `ticdat` in a more sophisticated way, by creating a dedicated schema solely for reporting advanced integrity problems above and beyond those that could be diagnosed directly with a `ticdat` subroutine.
- *"The user has 1.5 million rows of input data, and only five data entry mistakes. Does it make sense to stop the entire solve process to report those five problems?"* This is a great question. I've seen many variations on this theme and the answer is always the same. It depends on what the user really wants. So long as the user is fully aware that certain types of mistakes will be automatically corrected, then you can incorporate a data

[7] Are you serious? Apple pie.

cleaner into the solve process. In this case, the data cleaning should not be done quietly. The user should be given some type of feedback summarizing the data fixes that were made during presolve. However, if the user is unable to define an iron-clad suite of data cleaning rules, then even a single data integrity problem is sufficient to block the solver. Whatever you do, don't evangelize the value of data cleaning. The user will let you know which integrity rules can be automated away, and any opinion you have will likely be mere guesswork.

Safety and the Network Design Wars

Our main product at LogicTools was called LogicNet Plus (LNP). It was used primarily for supply chain network design studies. LNP enjoyed a period of dominance for several years, but it was eventually displaced (and even absorbed) by LLamasoft, a company that eventually grew to be much larger than LogicTools. There was a transition period that I like to call the Network Design Wars. During the beginning, LNP did quite well, but by the end Supply Chain Guru (the LLamasoft tool) won every engagement. During this time, Supply Chain Guru (SCG) evolved radically in the area of safety.

In the early days of SCG, the tool was very loose in terms of what sort of data it allowed a user to upload. Data integrity problems simply passed through to their back-end database. This would sometimes lead to easily avoidable, unhappy outcomes for their clients. "It would just barf" was how it was put to me by a LLamasoft veteran.

LNP never had this sort of problem. From the very beginning, I was adamant - LNP should never store corrupt data of any sort. Duplicate rows, foreign key failures, nulls in columns where they had no meaning… none of this was ever allowed. This strictness was consistent with my computer science training in general, and with database best practices in particular. At the time this seemed an obvious rule: "Block the storage of bad data".

Obeying this rule, however, created a fair number of headaches for users and developers both. The data requirements of a tool like LNP or SCG are nuanced, so dirty data was completely natural. (I believe this to be true for the optimization space in general, not just network design). At LogicTools, our GUI developers had to put significant work into customizing the data entry experience in order to provide good error messages with respect to dirty data. When a single cell data edit was rejected, the user needed to be told why. Even trickier was bulk uploading of data. When 50 out of 200,000 rows were rejected on import, the user needed to know which rows, and for what reason. This took quite a bit of work, but it did save the user from the "just barfed" solve experience.

To be fair, not quite all our users were pleased. At one point, our founder flamed out in a demo because the GUI had led him into a safety cul-de-sac. In front of potential clients, he was unable to successfully make a data change, because the GUI was (in this case awkwardly) protecting the integrity of stored data. Even though I wasn't a GUI developer at the time, the safety requirements were my design, and I tried to shield the rest of the team from the blowback.

While it's true that the GUI could have been more intuitive in this particular instance, I felt the strategy was sound. Moreover, the onus should always be on the salesman to fully prep his demo[8].

At some point, SCG addressed this exact problem, but with a technique completely different from ours. The new version of SCG continued to accept dirty data, but it wouldn't quietly solve on such an input set. Instead, it would alert the user to data integrity failures as part of the solve process. To my mind, the game was over then and there. This was clearly the better approach, for users and developers both. Users preferred the experience of collecting their input data all in one place, and then discovering the integrity problems in aggregate. The LNP experience of being blocked at each and every dirty data upload attempt made it far harder for them to stage a comprehensive data set, and forced them to upload data in a specific sequence.

For developers, the workflow was both simplified and more well organized. The GUI developer was now responsible for uploading data regardless of quality, and for publishing the integrity failure report. The data scientist (the sensible choice for determining the solve requirements) was responsible for validating the input data prior to solve.

Typically, there is a natural trade off between developer effort and ease-of-use. To make life easier for the users, the developers need to work harder. In this case, there was in fact a solution that's easier for everyone. I completely missed it when I laid down the foundation for LogicTools software, but now you can learn from my mistake.

Many years later, I was part of LLamasoft, and I swapped war stories with a few people on the SCG team. Their side of it was that LLamasoft founder and CEO, Don Hicks, was personally responsible for this innovation. The engineers themselves were inclined to follow the LogicTools approach (which was, after all, more in line with CS orthodoxy). To my mind, Don's insight here is on par with Steve Jobs' determination that a touchscreen-based device should replace the BlackBerry. When I listen to someone say that a sophisticated analytical app should block users from uploading dirty data, I hear Steve Ballmer saying the iPhone won't appeal to business users because it lacks a keyboard.

The Hicks' Hierarchy of Analytical Apps

I've seen many analytical apps. Like the testing hierarchy of the last chapter, I'm going to rank them into four categories, from worst to best, based on how they deal with presolve data integrity. Since the top category is based on SCG, I think "Hicks' Hierarchy" is the proper crediting. If you ever meet Don Hicks, feel free to give him a fist bump..

- "Let it barf" This is the early version of SCG that I described. It's also a fair assessment of the Opex apps, other than those that took advantage of `ticdat`. In this case, the user is allowed to upload and/or manually enter dirty data, and the solve will give little to no indication that such problems exist. At Opex, there were sometimes carefully scripted

[8] How do I remember this stuff? Sicialian Alzheimers. You forget everything except for the grudges.

bulk upload processes that were aware of data integrity requirements, but the user was still allowed to damage the data when editing a table manually through the data grid. I will never forget the crestfallen look on a data scientist's face when I broke his data integrity with a single one cell edit, and then showed how his solve quietly ran on the resulting nonsensical input data set[9].

- "This shouldn't even be happening". In this case, the app doesn't allow for the storage of dirty data, but it also doesn't give the user particularly helpful messages when they try to enter such data. This sort of app is somewhat easy to put together, because back-end SQL databases typically allow for triggers to guard against bad data updates. That said, I still think the validation report technique is easier to implement, in addition to providing a far better user experience.
- "Powder and paint". These apps prevent the storage of dirty data, and give the user rich error messages when such an attempt is made. This describes the LogicTools apps, including LNP. The code required to diagnose data integrity failures was particularly daunting for bulk data uploaders.
- "Gold Standard" This describes SCG, as well as the Opex apps that used `ticdat`. The design decisions I made when building `ticdat` were strongly influenced by the goal of facilitating gold standard app-building for intermediate-level Python programmers.

Safe Analytical Packages

Up until now, this chapter has described safety in the context of analytical apps. A reasonable person might ask "What about a data scientist who isn't building an app at all, but is simply developing Python code to be shared with another engineer?"

In this case, the exact same concept applies. In fact, I think the safety requirement is even more obvious. Let's walk through the argument.

Without loss of generality, we can assume the analytical code will be distributed as a package. (The other choice is proof-of-concept code, which implies neither safety nor hygiene, and is typically best accomplished as a notebook). This package exposes subroutines (either directly or as member functions) via the `__init__.py` file. To be professional code, any public subroutine needs to be bombproofed with respect to preconditions.

That means you can't have a square root subroutine that goes into an endless loop, or otherwise behaves in some erratic way, when passed a negative number. One option is to raise an exception (for example, the `math.sqrt` function). Another option is to emit a warning and return a flagging value (in this case `numpy.sqrt` - one could write a whole book on the philosophy of `pandas` and `numpy`, and how it differs from Python proper). As you might have guessed, a third option is to return *i*, the imaginary number that is the square root of negative one (as does `cmath.sqrt`). But under no circumstances does industrial code "just barf" when a bad argument is passed to the deliberately published public function of a package.

[9] My therapist tells me that normal people don't treasure these sort of memories.

In the case of the `tts_diet` template, the strategy is as follows.

- The package is based around a public `solve` function.
- The input data set of this function is passed as the `dat` argument.
- The docstring of `solve` informs your fellow engineer that the preconditions of the `dat` argument are defined by `input_schema`, which is itself a public attribute of the package.
- If those preconditions are violated, then an `assert` exception is raised by the `solve` function.

The logic behind using `assert` here is that the client programmer has full access to the same `input_schema` member functions that define the preconditions, so it's likely that the client code will call those functions prior to calling `solve`. Since the data integrity functions sometimes require a noticeable amount of time, a production system (i.e. a system using the `-O` optimization flag) will not call them twice as long as the `solve` function uses `assert`.

As I said, `tts_diet` is just one example of how to enforce safety for an analytical package. The larger point is to make life as easy as possible for the programmer importing your code. This requires validating the preconditions of your public functions.

Safety and the solution object

I've been asked more than once why the `ticdat` template defines integrity checks for the input data and not the solution data. I really like this question, because it demonstrates an interest in defensive programming. Projects rarely fail because the programmer took too many precautions.

The answer here is that the input data and the solution data have different validation requirements because they are sourced differently. The input data is provided by an end user interacting with a GUI, or by a client programmer who has imported your package into her code. Client or colleague, this is someone you need to protect. You aren't flattering this person by assuming they will provide pristine data, but rather, you are putting them in peril. You should recognize that the data integrity requirements of your `solve` function are difficult to understand, much less satisfy, and thus input data validation is the only professional form of cooperation.

The solution data, on the other hand, is created by the `solve` function. If the solution data fails integrity checks, it simply means the `solve` function is buggy. Moreover, this function can be buggy in 1,000 ways that don't manifest as integrity failures. Thus, my practice is to develop double check routines for the solution as part of unit testing. Solutions that are generated during a unit test run are programmatically inspected, and a unit test failure is raised if they flunk some form of sanity checking. For expediency sake, these sanity checks are not as polished as the `ticdat` integrity checks. Of course there is no need for these sanity checks to be professionalized, since their purpose is to simply raise a red flag for the developer of the `solve`

function (me) that something is amiss during unit test execution. I will discuss these sanity checks a bit more in Chapter 11.

Chapter 7 - Is Tidy, Test, Safe the Same Thing as Agile?

The development of Tidy, Tested, Safe was strongly influenced by the following assertion.

The traditional roles of consultant and programmer are blurring together into the new role of data scientist. Agile is the most appropriate software methodology for managing this trend.

At the risk of starting a religious war[10], Agile is the most important software development methodology being used today. If it isn't the most popular, then it ought to be. Certainly most of the data scientists I have worked with are at least aware of Agile, and will give some indication that they prefer to work that way.

However, when push comes to shove, even very talented people will often do a much better job of participating in the daily scrum meeting than they do of building quality automated tests or targeting a reasonable MVP. I assert that Tidy, Tested, Safe is a prerequisite for Agile. If you are unable to perform the Tidy, Tested, Safe tasks, then your team will likely be performing “Agile theater”, where everyone uses Agile terminology but nobody delivers Agile customer value.

Let's consider the four core values proclaimed by the Agile Manifesto and frame them within the context of Tidy, Tested, Safe.

Agile: Individuals and interactions over processes and tools

TTS: Team hierarchies have flattened as languages and tools have democratized.

The field of IEOR is sadly plagued by legacy technology. While they made sense at the time of creation, IEOR's math modeling languages are now technology silos that impair the communication between data scientists and engineers. Similarly, the app-building tools serving the IEOR community often fail to play nicely with widely popular code collaboration tools like GitHub and BitBucket. While I see many advantages in aligning IEOR with Python and Git, the most significant is removing the Tower-of-Babel communication barrier between teammates who have different backgrounds and areas of expertise. When there is a consensus on basic technology, then the technology details fade into the background as individuals come together to form deeply collaborative and dynamic teams.

Moreover, let me emphasize that the production code requirement of Tidy, Tested, Safe is not a barrier to code sharing between people with broadly divergent backgrounds. One of my favorite work roles is that of the TTS engineer assisting someone with deep expertise in a particular domain. For example, my knowledge of `sklearn` (and traditional ML) comes largely from collaborating with a data scientist who, at the beginning of our project, fit the classic stereotype of the Python beginner and ML expert. At launch, his job was to lead me on a detailed tour of

[10] I’m ok with starting a religious war.

`sklearn` so I could focus on production code. At my request, his code sharing began with notebooks sent as email attachments. By the end, he was opening GitHub pull requests into the perpetual develop branch that would then await my review and merge. In the meantime, I had become far more skilled with `sklearn` in particular and ML in general. This is what comes to my mind when I hear “Individuals and interactions over processes and tools.”

Agile: Working software over comprehensive documentation

TTS : Readable code that implements trustworthy software is more important than documentation.

Specifically questions like "What tables and fields does it require? What are the data integrity rules when populating those tables and fields?" should be easily answered in two ways. First, by reading a clearly defined subsection of the code dedicated to the input data. Second, by experimenting with sample input data sets and seeing what sort of integrity error messages are generated. Supporting both of these techniques is far more important than formal documentation.

Bear in mind, trustworthy software takes on additional importance in the area of optimization. You should expect your engine to produce counterintuitive results. After all, if intuitive results were all that was required, then the problem could be solved more easily with simple heuristics.

It is for this reason that Tested and Safe take on extra importance when developing optimization engines with Agile methodology. Software that behaves badly with data-entry errors, or misfires with the actual modeling equations, becomes “not working” in the worst possible way. Such code degrades client trust. Unfortunately, the “as designed" counterintuitive behaviors that are endemic to optimization will also drain your client’s good will. In combination, this can lead to a dysfunctional relationship that cripples your ability to adapt to changing requirements (the next Agile core value).

Agile: Responding to change over following a plan

And also

Agile: Customer collaboration over contract negotiation

TTS: Once you can agree on the input data, then implement the mathematics in the most granular way possible.

Some features truly do require other features. But to the extent that logic allows, you should iterate as tightly as possible over the cycle of "add a feature, test a feature, seek client feedback". Bear in mind, it’s critical that we don't introduce new bugs every time we respond to new information about client requirements. It is for this reason that Agile requires us to update our automated testing code with every such iteration.

As I explained in the last section, it is normal for optimization engines to confound even smart, knowledgeable users. Thus, you should expect requirements to change based on actual client interactions. There is nothing dysfunctional about laboring over a fussy set of equations, only to discard them based on feedback. The important thing is to manage this dynamic with care.

For example, I once made the worst possible mistake with a client. After proposing what appeared to be a modest modeling change, he asked “this is easy, you’ll get it done fast, right?” My response was “It’s likely to be harder than it looks. It often takes some time to adapt the unit testing code to new features”. I should have just stopped after eight words, as the client’s next request was that the troublesome unit testing code be abandoned.

You are the best judge of who to trust with the details of how sausages are made. It is my experience that the end users who understand the value of unit tests the least will be exactly those clients who impose the most changes, and thus need unit testing the most. It is normal to encounter people for whom optimization is “magic”. As clients, these people tend to struggle both with specifying clear requirements and understanding the challenges of meeting those requirements. For the purpose of client updates, the unit testing overhead can remain simply part of the work you do between meetings. Agile does not require the programmer to provide a time-motion study with each new release, nor to open the black box of her craft to an unappreciative audience. If your manager doesn’t understand the importance of testing, then you need to look for a new job. But if the client doesn’t understand it, then you only need a deliberate communication strategy.

Chapter 8 - The Golden Triangle

I have often characterized myself as someone who likes computer programming, but dislikes computers. That's a bit of an exaggeration. I like computers when all the tools I need are installed and working properly. In my ideal world, magic elves would handle my system administration needs.

In recent years, my life has been rapidly improved by the rise of both Python and GitHub. These technologies share a lot in common, as we've already discussed. They both are enormously popular, with large and growing communities. They both have low barriers to entry for novice developers. They both rely more on well understood convention than strictly enforced rules to guide large, complex projects. In this chapter, I discuss another point of overlap. They both have very easy install and support requirements for use with a Windows or Mac laptop.

I group the applications I am enthusiastic about into three general categories: PyCharm, Anaconda, and GitHub. Regardless of what applications you use, I do feel compelled to point out that the data science community is very well served by tools that are free (or at least dirt cheap), widely used, and easy to maintain and install.

I would be very leery of a vendor or an employer trying to impose a "pure cloud" workflow on data scientists. While you will need some sort of license for a commercial MIP engine, the development and debugging experience from a modern laptop is so good that the proper use of the cloud strikes me as purely supplemental. One can rent a sizable machine as needed through the Gurobi Cloud (or something similar) when a large memory footprint is needed. An app-deployment tool (such as the one you can find here) can be used to create a GUI for your end users. Google Colab is a great, cloud based, demonstration code solution. Databricks is a great place to explore huge data sets using a Jupyter-style notebook. All these technologies are fantastic, but you shouldn't let them distract you from your core mission: the development of properly tested Python packages.

My goal with this book is to convince you that production code belongs in well-tested Python packages, and that the development of well-tested Python packages is the natural next step after mastering Python syntax. I'm not a partisan for using one tool or another. What seems clear, however, is that data scientists have a tough enough job as it is and should not be denied the ability to choose their own toolkit.

For example, it's axiomatic that software works best when it is actually installed (as opposed to copied around). One of the wonderful things about Python is that package development is fairly easy, and thus a broad community of people can now build installable software. But, when I say "fairly easy" I mean "fairly easy with the widely used, free tools I describe below". I can't guarantee that you will have a smooth package development experience (or even a Tidy Tested Safe experience) when building in a pure cloud development environment. I do feel confident making such a statement if you install the Golden Triangle, and spend some time exploring the functionality they provide.

PyCharm

PyCharm is a Python editor and IDE that is distributed via a freemium pricing strategy. Other tools like Spyder, PyDev and VS Code provide similar functionality. I will discuss PyCharm here since it is the one I am most familiar with. I suspect that the competing tools are similar enough that the best advice is to simply pick one of them. For the purposes of this text, just assume I am using PyCharm as a generic term, like Kleenex.

I group the PyCharm feature sets into two broad categories: editing and debugging.

Python is best written with a feature rich text editor. Line numbers should be readily apparent. It should be easy to do bulk indent and un-indent, and also to do bulk comment and bulk uncomment. There should be an automated way to rename functions and variables. It should be easy to navigate directly to a function definition. The latter two features require an editor capable of inspecting your Python code, which PyCharm clearly does very well.

In addition, PyCharm provides an integrated debugger. It is easy to place breakpoints in the code. It is easy to step line by line through the code, with a choice between stepping into subroutine calls or stepping over them. One can navigate up and down the call stack, and evaluate an arbitrary Python expression within the context of whichever call stack is current.

To be clear, I perform all the debugger tasks with the `ipdb` command line based debugger, which is actually part of the Anaconda technology stack I describe below. I include the debugger in the PyCharm section because most of the Opex data scientists preferred the PyCharm debugger to `ipdb`. It's not really worth arguing over which debugger is better, as the larger point is that you will need a debugger. The main thing to avoid is the folly of debugging your code with print statements.

People really tend to fall in love with one debugger or another, and I think such a bond is entirely understandable. The debugger is the tool that's saving your hide, over and over again. With practice, its nuances become a core part of your skillset. The nice thing about the PyCharm (or `ipdb`, or Spyder) debugger is that since these tools are free, you shouldn't have to worry about a future employer demanding you switch.

Anaconda

Your computer probably comes pre-installed with Python. You shouldn't use this at all. You should install Anaconda, and allow the Anaconda Python to be your default.

Anaconda serves a few different roles for a data scientist.

- It creates a broader set of "standard libraries". For example, packages like `pandas` and `numpy` are part of Anaconda Python.

- Anaconda will allow Jupyter notebooks to run locally. As I said, notebooks are the preferred tool for demonstrating code behavior.
- Anaconda functions as a “baby Docker” for running Python. I describe this in a bit more detail below.

If you’re not already familiar with Docker, then you should at least read its wikipedia page. Containerization and OS virtualization are truly game-changing technologies that data scientists need to understand, if only at a high level. That said, I don’t think it's reasonable to require data scientists to master Docker. Anaconda's environment management can function as a quasi-container to assist in reproducing code behavior. Anaconda is far easier to learn than Docker, and you will likely need to install Anaconda under any event. What’s more, Anaconda even includes functionality to generate Docker images. I posit that the answer to “Should a data scientist learn Docker?” is “Only after you understand Anaconda (and the rest of Tidy, Tested, Safe)”.

Anaconda has wonderful free online documentation, so I’m not going to walk through the exact steps here. I do maintain a wiki page that discusses some of these issues in more detail.

GitHub

The rise of GitHub has been truly remarkable. For many of us, gone are the days when you needed to learn a new source code control tool with each new job, or new teammates. It appears that in addition to his other project, which you probably have heard about, Linus Torvalds has solved the code-sharing Tower of Babel.

You can assume that I am using GitHub a bit loosely, again like Kleenex. There are other competing Git hosting sites. This book will acquaint you with GitHub in particular, but the skills you learn will carry over to other Git services, such as Bitbucket and GitLab.

My GitHub workflow centers around three specific tools - the git command line tool, the GitHub desktop application, and the GitHub website itself. Of course, you only need a browser to access the GitHub website. The git command line tool is likely already installed on your computer. GitHub Desktop can be downloaded here.

I jump back and forth between these to use whichever tool is most convenient for the job at hand. It generally breaks down like this.

- I typically `git clone` from the command line, just because it's so convenient. I also use the `git` command line tool to create a new branch.
- After I’ve edited the local copy of my code, I use the GitHub Desktop app to compare the local code to the current status of the branch. If you’re savvy, you can do this via PyCharm as well. **I find this incredibly helpful**. Once you get deep into the weeds of a problem, it is natural to make temporary, experimental changes, and reviewing your edits before committing them prevents these edits from actually becoming permanent.
- My commits are performed from the GitHub Desktop app.

- I use pull requests created from the GitHub website to merge into the trunk branch. Of course, I delete the sub-branch after the merge is complete, again from GitHub itself.
- I use the GitHub website to create releases. GitHub makes it easy to compare the current state of the main branch to any prior release by using `/compare` syntax in the URL. This helps me organize my release notes before actually making the release, and is also a useful code review process.

I would also recommend making use of the GitHub wiki, the GitHub issue tracking system, and GitHubs ability to render notebooks directly from the website. The `tts_netflow_a` and `tts_netflow_b` demonstration repositories both have a `notebooks` directory that serves as a form of supplemental documentation.

Chapter 9 - Tidy Tested Safe in Action

By now, you might be ready to start writing Tidy, Tested, Safe code. A combination of sermons and war stories have provided motivation, and perhaps even inspiration. We have examined each part of Tidy, Tested and Safe in detail. The TTS guidelines have been framed within the concept of Agile, the most important software development methodology being used today[11]. And finally, you've been introduced to the Holy Trinity of free, or dirt cheap, tools that you need to craft the code itself. It's time to roll up your sleeves and make some sausage.

I see three ways to start applying this knowledge.

1. Simply start building TTS repos as part of your job right now. If your manager balks at the idea of working in a manner consistent with standard Python and GitHub best practices, then you should consider finding a new employer that promotes meaningful professional development.
2. Work through the network design examples that constitute the rest of this book.
3. Develop a homework assignment according to the TTS template.

The primary goal of TTS development is to align IEOR within at least an arms length of how proper software engineers develop code. If you can meet these objectives with Python, but without `ticdat`, then I'd love to see it. If you take the more straightforward path of closely following the template illustrated by the `tts_` demonstration repositories, then you will probably also enjoy deploying your project as a web-based app. Doing so will familiarize you more deeply with duck-typing, and you'll experience the sheer joy of seeing your code come to life for a broader audience. As of this writing (spring 2022) I can name two `ticdat` compliant app deployment tools that are commercially available. Going forward, I will maintain this page with current bootcamp course information.

Regardless of which path you choose, you should first study the `tts_netflow_a` and `tts_netflow_b` demonstration repositories. These repos are more sophisticated than `tts_diet`, particularly in terms of illustrating meaningful change from one release to the next. They demonstrate my take on how Agile methodologies actually work in practice, within the context of optimization. I have created fictional bugs that needed to be fixed and fictional client feedback that was addressed.

If you are following path 3, or path 1, then I encourage you to similarly work in a deliberate, granular manner. Feel free to create milestones for your academic assignment, to meet these milestones with tagged releases, and to forward convert testing data as appropriate. For actual paid work, your milestones will likely be driven by client expectations, but you are probably better off following Agile methodology and supplying frequent, stable, product releases. For the network design problems constituting the remainder of this book, I have arranged things so that your work will be organized into predetermined feature-based milestones.

[11] Are these "fighting words"? I'm ok with it.

Backwards Compatibility

Before we dig into the demo repos, we need to discuss the concept of backwards compatibility.

The package you are developing can be used in two ways.

1. It can be imported by some other body of Python code, which will then use its public interface. I like to use the term “client code” to refer to code that uses the package that I’ve developed. This term applies even if the downstream code is written by colleagues, and not paying clients.
2. It can be run from the command line. This is optional for packages in general, but mandatory for the TTS template. Lets use “client data” to refer to the input data sets that can be solved this way.

The term “backwards compatibility” refers to whether or not a package will continue to function with pre-existing client code or client data when a new version of the package is deployed. Any instance where a prior version of the package worked, but the newer version fails is considered a break in backwards compatibility.

Is breaking backwards compatibility bad? No. To maintain strict backwards compatibility at all times would be to step into a deontological trap worthy of Rorschach. You might even find yourself unable to fix bugs, out of concern that client code actually only works with the original, albeit incorrect, behavior of your subroutine.

However, backwards compatibility is something you should take seriously. The most important question to answer here is “How big is my client base?” The broader the audience using your code, the more care you need to take with backwards compatibility. With a very narrow audience, you enjoy a nearly unilateral range of freedom to change the public behavior of your package. (By definition, changing the internal implementation, while maintaining the same public functionality, does not break backwards compatibility at all. As discussed earlier, this is called refactoring). On the other extreme, `pandas` and the Python language itself take backwards compatibility very seriously. The switch from Python 2 to Python 3 required a massive publicity effort and a multi-year phase out. Similarly, you will see `pandas` emit deprecation warnings well in advance of the actual backwards compatibility break.

Here are some other considerations to bear in mind with respect to backwards compatibility

- Feature or bug? Behavior that is truly buggy can simply be fixed without much fanfare beyond a reference in the release notes. However, sometimes reasonable people disagree on what does or does not constitute a bug. A bug-fix that might be considered a feature change needs to be more carefully considered, and might merit indirect support via a work-around.
- Is there a work-around? If there is an easy way for client code to recreate old behavior (perhaps by overriding the default value of a newly introduced function argument) then the break is far less problematic. Sometimes, the smart move is to consciously enable

historical behavior via a work around, while introducing the bug fix as the new default behavior.

- Insidious or obvious? It is always better for any backwards compatibility break to be made as obvious as possible. The release notes aren't supposed to be purely promotional material. In addition, a break that consistently throws an exception is better than one that quietly changes subroutine behavior.
- Is it really a backwards compatibility break, or was I reaching into the black box? Packages in Python have a very clear and obvious public interface:. that which is defined in the `__init__.py` file. However, under the rubric of "we're all consenting adults", Python doesn't force you to use only the public interface of the package. If you want to import a function or class from a package that isn't part of its public interface, nobody can stop you. If you choose to do so, you're taking a risk that this functionality might change willy-nilly in future releases of the package you're importing. This is because the package developer is only obligated to consider backwards compatibility with respect to the public interface. Indeed, one could think of the public interface and backwards compatibility as interwoven concepts. In Python, the public interface of the package is that portion of the package that the package developer promises to manage with respect to backwards compatibility.

Even if you are only writing bespoke packages for individual clients, it's important to understand backwards compatibility. You might be tasked with evaluating different vendors, and those who don't take backwards compatibility seriously are far less trustworthy than those who do.

The tts_netflow_a Repository

The `tts_netflow_a` repository demonstrates the development of a network flow solution. Release 0.0.1 solves a simple min cost network flow problem. There is no need to model commodities, and the input table consists of only two tables.

The 0.0.2 release enhances the package by introducing commodities. That is, it now solves the more sophisticated multi-commodity network flow problem. This enhancement might be in response to a client request, or it might be that the 0.0.1 functionality was always intended to bootstrap a more sophisticated solution.

Let's examine the commit history that advanced `tts_netflow_a` from 0.0.1 to 0.0.2. To begin, we make a new branch, `multi_commodity`. Our first commit does nothing but upgrade the input schema. At this point, our unit tests are now broken - in no small part because the schema change results in a backwards compatibility break. The first step is thus to forward convert the testing data to the new schema. The notebook that accomplished this task is archived here. Only now (after upgrading the schema, and forward converting the testing data) do we perform the sexy and celebrated work of advancing the `solve` function with more advanced mathematics. At this point, the unit tests are passing, but there are no unit tests that involve actual multi-commodity data. (This is because the unit tests at this stage consist of forward converted 0.0.1 data sets). Thus, the very next commit exercises the new functionality by

incorporating a multi-commodity data set into our unit tests. This new data set includes arc capacity constraints that meaningfully affect the solution.

This completes the actual code enhancements needed for 0.0.2, but we still have GitHub tasks to perform. Before making the 0.0.2 release, we merge the `multi_commodity` branch back into the perpetual branch, delete `multi_commodity`, and bump the version number.

This might seem like a lot of overhead. Do you need to make this many commits for such a small feature upgrade? Of course not. I used fine grain commits to demonstrate the sequence of steps I would have followed. In real life, had everything gone smoothly, this could have been just one big commit, but two or three would have been more likely. The primary rule I follow is to avoid breaking the unit tests on a perpetual branch. This rule is much more strictly observed when following a one-perpetual-branch protocol, like I am here. If it's helpful to temporarily break the unit tests, then just do it on a sub-branch that will be later merged and destroyed.

The other rule is to align the release tags with the `__version__` value. The exact timing of when to bump the version isn't particularly important, so long as no two release tags share the same version. So long as you take care to only deploy from a tagged release, you will then be able to reliably identify the exact SHA being used by any given client.

The 0.0.3 release addresses a fictional bug report, as documented here. Consider a (source, destination, product) triplet from the cost table that has no matching (source, destination) record in the arcs table. The 0.0.2 release handled this by quietly ignoring the cost record. Issue #2 documents the user's complaint about this behavior. An easy fix is to simply flag such a triplet as violating pre-solve data integrity. This is accomplished by using the compound foreign key functionality in `ticdat`.

The tts_netflow_b Repository

The `tts_netflow_b` repository tackles the same problem, but uses a different approach to the MVP criteria. Before, our 0.0.1 release solved a simpler but related problem to that which was ultimately addressed. Here, the 0.0.1 release has a dummy `solve` that is incapable of creating a solution. The input schema, however, is fully defined, to include the cost-to-arcs compound foreign key. The goal with this release is to solicit feedback based on sample data (generated from testing data here) and a package capable of fully validating input.

I like this IEOR Agile strategy a lot, but it does require a user who can take a low-math version of the code seriously. That is to say, `tts_netflow_b` demonstrates the workflow I prefer, but it only works with an end user who will create real-world input data for the sole purpose of validating data integrity, as that is all the 0.0.1 release is capable of doing.

In this case, we do have such a user (albeit a fictional one). In response to the 0.0.1 release, our user decided that having a single inflow table, that modeled supply with a positive quantity and demand with a negative quantity, was too abstract. Instead, we split inflow into two tables,

one for supply and one for demand. The Quantity field for both tables is required to be non-zero, and we also include an integrity check that the tables don't overlap.

For the 0.0.2 release, we don't fully remove the original inflow-table based schema, however. Instead, we retain it as an alternate input schema that is more friendly to a math modeler. This additional overhead is clearly for instructional purposes, as a modeler would surely have no real trouble building the math equations directly from the supply-demand schema. We provide two input schemas here to illustrate a problem that I believe to be common: the data tables most helpful to a math modeler might not make a lot of sense to a business-oriented user. In this example, the two schemas deviate only slightly, but you should be prepared for situations where the difference between business-friendly and math-friendly tables is far more significant.

Here we demonstrate a pattern to address this problem: the data pipeline. The complete solution (which we won't get to until release 0.0.3) will work like this. The business user will be presented with an input schema that conforms to his peccadilloes. A data-munging function will be implemented that converts the business user's data set into a schema more palatable to a mathematical modeler. The advanced mathematics will be implemented via a function that creates a solution from this math-oriented schema. Finally, the `solve` function will daisy-chain these two functions together to complete an end-to-end solution.

The data pipeline is well suited to real-world optimization problems. At Opex, I saw this situation addressed with apps that essentially had two sets of input tables. Such an app would include a data action that would populate the modeler-friendly tables from the business user tables, with the `solve` reading exclusively from the former. LLamasoft's Data Guru product can be used to build data pipelines between operational data and SCG. Alteryx can similarly be used to build pipelines that feed analytical apps. Of course, I prefer to build pipelines with Python, `pandas`, and `ticdat`, which is what I demonstrate here. One advantage of this approach is that these technologies are all open-source, and thus you can carry such skills with you throughout your career.

The 0.0.2 release doesn't get us all the way to a complete pipeline. As before, our `solve` function doesn't yet do anything, but we now present a more business user friendly input schema. As with `tts_netflow_a` , the commit history from 0.0.1 to 0.0.2 includes the forward conversion of the testing data (demonstrated here). The unit tests are also advanced to validate that the supply-demand non-overlapping integrity rule works correctly.

In this fictional world, our user is happy with the 0.0.2 input schema, and thus can finally add the sexy math[12]. The 0.0.3 release thus implements a proper `solve` function. Since we are demonstrating `PanDatFactory` here, and not `TicDatFactory`, the code that translates tables into equations is slightly different, but the underlying mathematics is exactly the same.

[12] I admit I beat this horse to death a few chapters ago, but I'll never grow tired of assaulting this equine corpse. There are a lot of workmanlike software engineering steps you ought to do in support of the sexy math, and you're better off doing most of them first.

Our unit tests are advanced to create actual solutions and validate that they are populated with the expected results.

There is also one additional twist to the 0.0.3 release, which I include as part of the pipelining pattern. The command line interface for the `tts_netflow_b` reads from the business-friendly input schema. However, we also support a user who wants to use the command line to solve from the math modeling schema. This is done via the auxiliary script, `tts_netflow_b_modeling_schema.py`. As this code is a free-standing script, and not a proper package, one might wonder why I consider it production code at all.

The answer is simple: the file is tiny. The real work is all done by redirecting to the `tts_netflow_b` package. A Rorschach developer would complain about such a file, and insist the `tts_netflow_b.__main__.py` file be edited to present a more complex set of command line arguments instead. While there is nothing wrong with this approach, my personal Ozymandias would reject it as the only approach. Stand-alone .py files can be used responsibly, but only if they remain small. Unless the explicit purpose is for demonstration, then code that can't be easily scanned by a human being belongs in a package. In this case, our .py file has only a few lines, and thus it is safe to distribute.

In Praise of Problem Reduction

Before we dig into the network design exercises, let's drive home one final lesson from the field of computer science. What do this notebook, this notebook, and this subroutine all have in common? I'll accept "data munging" as an answer, but obviously I'm looking for something deeper. They are really all examples of problem reduction.

I'm assuming any reader of this text already understands NP-complete, and has designed problem reduction algorithms as some form of coursework. My point here is that problem reduction is a deep pattern that is woven throughout all of computer science. The easiest form of customization is when you use another solve engine to do your heavy lifting. This is problem reduction. This term I invented, "forward conversion of testing data", is another real-world example of problem reduction. If you enjoy theoretical computer science, then the ubiquity of problem reduction should give you hope for a non-academic career.

I'm emphasizing this point because my experience is that forward conversion of testing data, like all testing work, seems sadly unloved by practitioners with a theoretical background. This is foolish, from the perspective of Ozymandius and Rorschach both.

Leaving aesthetics aside, it's just pragmatic to capture the low-lying testing fruit associated with forward conversions. Debugging and troubleshooting are all about isolation. After you've established that the prior tests still work, the job of validating the new functionality becomes vastly simpler.

Of course, sometimes a user might ask for a feature to be altered in such a way that only some of the prior testing sets will reproduce. This is hardly a reason to throw the baby out with the bathwater. Your understanding of your own engine will only become deeper when you carefully prune out the tests that need to be removed, while retaining those that should still pass given proper forward conversion of testing data.

That being said, I freely admit that I am as uncompromising as Rorschach when it comes to my testing data. I always carry forward every single test that can be carried forward, and I never think about the cost-benefit analysis of this strategy. Of course I sometimes make mistakes, and sometimes those mistakes are with the forward conversion logic, and not with the newer version of the `solve` function. I don't have hard data here, but my sense is the former is far more common. This means I sometimes go on long troubleshooting journeys, only to discover the "bug" was never in the production code in the first place, and would never have been visible to the end user.

This might sound strange, but such an experience never frustrates me. Every time this happens, I feel as satisfied as Rorschach dispensing righteous justice. The purpose of learning P and NP was not merely to give you contextual background for practical applications. It was to rewire your brain so as to change the way you'd engage practical applications. Of course you want to see problem reduction brought to life, so of course you will carry forward the testing sets by writing the code that reduces `solve` version n to `solve` version n+1. This is the best part of the job, because the logic of problem reduction means you are guaranteed to succeed, if only you don't quit. Deontological thinking has its dangers, but it can also be a boon to your reputation. I recommend you don't quit on this. Forward convert your testing data until the prior tests all pass, or until you learn which tests have to be pruned, and why.

Chapter 10 - Strategic Network Design Exercises

The exercises in these last two chapters constitute the Tidy Tested Safe course. Perhaps you skipped straight here from Chapter 1? If so, kudos. I admire such bravery. Even so, you might get a little disoriented, so please refer to the Tidy, Tested, Safe recap page. This would also be a good time to study the tts_netflow_a and tts_netflow_b demonstration repos.

This course requires you to build a package named `tts_scnd`, with a repo of the same name. If you complete the entire course, your repo will have six releases, one live branch, and as many commits as needed. You will also make development branches, merge them with pull requests, and delete them after merge. I deliberately developed the two netflow demonstration repos using these exact steps, so you have a clear demonstration of what the `tts_scnd` repo will look like.

What Do I Need to Get Started?

Create a GitHub account, install Anaconda and install PyCharm (or equivalent). I also recommend GitHub Desktop. If these terms are unfamiliar to you, please read Chapter 8 now. If you think you can work through this course purely with a web browser, please read Chapter 8 now. If you think you can build professional Python engines purely with a web browser, please read Chapter 8 now. I agree that to avoid looking foolish, I will eventually have to rewrite this book to encourage pure web development of professional Python code. My guess is no sooner than 2025, but I'll be happy if I'm proved to be wrong.

After that, `pip install ticdat`. For best results, you'll want version 0.2.22 or later. Be advised some of the `PanDatFactory` json logic was upgraded from 0.2.21 in order to support this course.

You'll need the ability to solve MIPs locally. I recommend `gurobipy`, which can also be installed using `pip`. The free Gurobi licensing currently supports 2,000 by 2,000 MIP models. While you can learn a great deal with an engine of this size, it isn't really large enough for you to study performance bottlenecks. For this reason, some of my sample data sets require a full Gurobi license. My understanding is that Gurobi is generous with temporary, fully featured licenses for academic or evaluation purposes. You won't need to `import gurobipy` until the third release, so feel free to get the first two milestones under your belt before contacting them for a license.

The tidy_tested_safe public repository has all the sample files you need in the `tts_scnd_course_files` directory. The easiest way to access these files is to either clone the repo itself, or download it as a .zip file from the Code download button.

App building is a happy side effect of following the `ticdat` template. Deploying your engine as an app is recommended, as it demonstrates duck-typing, and it's also good fun. That said, it is

not required for this course, nor is it a prerequisite for solid Python programming. I will maintain information on this page for anyone interested in creating apps.

Proof-of-Concept work

Not to brag[13], but my muscle memory is such that even my proof-of-concept code is fairly tidy. That is to say, I always start a project by creating a repo consistent with the `tts_` template and building a package backed by unit tests. For me, proof-of-concept is just a package with limited functionality, as demonstrated by the **Release 0.0.1** section below.

That said, one of the strengths of Python is the ease with which you can get results with quick-and-dirty code. A Jupyter notebook, or a left-most-code .py file, can be a fine way to start a project, or to contribute useful code samples that someone else can weave into a tidy repo. With that in mind, I've put together some Jupyter based warm up exercises in this directory. Since Jupyter notebooks include LaTeX, these notebooks also include the system-of-equations descriptions for **Release 0.0.3** and **Release 0.0.4**

If you are intimidated by writing Tidy, Tested, Safe code, or if you think such precautions are not needed[14], then I hope you at least complete the notebook exercises. You do not need `ticdat` to do so, as the sample data is clean and I provide the code to load the input data into convenient Python data structures. The rest of this book (and the bulk of the course) will be waiting for you if you someday change your mind.

Release 0.0.1

In this section, you will begin to write TTS code. To get started, you should make a GitHub repo named `tts_scnd`. A public repo is better, so that someone can review it. This repo will support the development of a package that's also named `tts_scnd`. (Please **don't** ever push a package with this name to PyPi, or else we'll all be confusing each other. It's a weird combination of characters, so I doubt it will happen unless somebody reading this book decides to be a smart alec.)

The files to help you build the 0.0.1 release are here. I've built the core .py file for the `tts_scnd` package for you. Use `tts_scnd.py` if you want to use `TicDatFactory`, and `tts_scnd_pd.py` if you prefer `PanDatFactory`. If you're confused as to which to use, then read this page. If you're curious, I put about ten times as much work into supporting `TicDatFactory` than `PanDatFactory`, and I almost always use the former for my engines, but I'm still flattered by people who use `PanDatFactory` exclusively.

What problem are we solving with the 0.0.1 release? Well, as you can see from the `solve` function, we're not actually solving a math problem yet. We're getting things organized prior to

[13] Ha!
[14] Listen to me now and hear me later. They are needed.

solving a math problem[15]. The math problem we're going to solve is a very simple strategic network design problem. This problem is described in Chapter 3 of the *SCND* book, and again in Chapter 2 of the OPL book. The LaTeX equations for this model are also provided in this notebook. Here, I will just discuss the input schema.

- The cities table defines both our demand points and our potential facilities. (We will model distinct demand and facility locations in the 0.0.6 release). Each city has a non-negative, non-infinite demand value associated with it.
- The distances table defines the distances between cities. The distance itself must be non-negative and non-infinite. The distances table must refer to a valid city both for source and destination.
- We have a single parameter for this model - "Number of Centroids". This refers to the number of facilities to open. By default (i.e. if the "Number of Centroids" record is missing from the parameters table), the solver (once we actually have a solver) will be directed to select four centroids.

The tasks required to complete the 0.0.1 release will largely be cutting and pasting. You're going to make a repo that looks like the `tts_` demonstration repos, with either `tts_scnd.py` or `tts_scnd_pd.py` as the core execution file of the `tts_scnd` package. The `test_tts_scnd.py` file you will write will only execute a single unit test. This test will create a `dat` object from the `MIP_for_9_City_Example.json` file, and call `tts_scnd.solve(dat)` on this object. Testing success here is just the absence of unhandled exceptions. You will need to adjust your Python environment to make the `tts_scnd` package you are developing universally importable in order to run your `test_tts_scnd.py` file.

You will also create a Jjupyter notebook to demonstrate your ability to create an Excel file, or directory of csv files, from the `MIP_for_9_City_Example.json` file. You can see example notebooks demonstrating this in both the `tts_netflow` repos. Such functionality is important because JSON format, while the best format for GitHub, is not universally recognized by end users. If you are deploying your `tts_scnd` repo as an app, for example, that app will likely need Excel or CSV format in order to upload testing data.

The full itemization of what your repo needs for the 0.0.1 release is documented here.

Release 0.0.2

The 0.0.2 responds to a fictional user questioning the proper population of the distances table. While this table has "Source" and "Destination" fields, it's reasonable to ask if there really is an implied direction to a given data row. That is to say, if a user provides a distance from Charlotte to Atlanta, does she also need to provide a record for Atlanta to Charlotte?

[15] I'm still hitting this dead horse. It's therapeutic for me, but I don't expect the horse to get up.

There isn't one correct answer to this problem. There is a correct strategy, however. "Do whatever the end user wants, while protecting the end user from data-entry mistakes." Agile development methodology excels at working through issues just like this. You can't expect the user to know every little detail at the inception of the project. Providing a partially functional release early (even if that functionality is only validating data integrity) will help bring these issues to light as soon as possible. For a deeper discussion of this point, see Chapter 7. This is the *"responding to change over following a plan"* core value of Agile development.

In this case, our end user wanted the best of both worlds. Specifically, if Charlotte to Atlanta is provided, then Atlanta to Charlotte isn't needed. However, if both Charlotte to Atlanta and Atlanta to Charlotte are present, then they should specify the same distance. Imagine the case where the end user is populating the distances table by blending a variety of data sources. She doesn't want the solve process to be blocked by the presence of only one of those two records. However, if both records are present, then it means the distances were provided by two different sources, and she wants to be alerted if those sources aren't in agreement.

This is just one example of what an end user might want. The end user might have just as easily requested that the `solve` function (which we will build in the 0.0.3 release) recognize that the distance from Charlotte to Atlanta might actually be different from the distance from Atlanta to Charlotte. That seems a little silly for strategic network design but I suppose it's possible. It wouldn't be strange at all for a city level routing application to have such a rule, in response to things like one way streets and the vagaries of freeway access.

At any rate, the 0.0.2 release addresses this point by adding advanced row predicates to the input schema. I've implemented these row predicates for you in the .py file provided in this directory. If you want to read more about `ticdat` advanced row predicates, look here (in addition to the relevant docstrings).

I want this to be a "build great Python code" course, and not a "learn `ticdat` tricks" course, so I'm doing all the `ticdat` work for you. Nevertheless, building sophisticated unit tests certainly is part of great Python, so you're going to have to add a unit test to validate that the advanced row predicate actually does check for bi-directional safety of the distances table. You can see here and here for examples of unit tests that validate a data integrity check. To be clear, writing such unit tests for every possible form of data integrity failure would surely be overkill, but it strikes me as reasonable for particularly tricky integrity checks like this one. Moreover, I want anyone completing this course to be familiar with the `try/except` idiom, which is clearly needed for moderately sophisticated Python.

More details on the 0.0.2 requirements are here.

Release 0.0.3

This is the release where you will actually build and solve a MIP. If you are using `gurobipy`, then you will need a local license file from here on out, as the `LargeScale3Location.json` data set (provided here) is too large to actually solve with the free Gurobi license.

In my personal experience, writing out a system of equations isn't particularly helpful for real-world situations. It's certainly needed when introducing Mixed Integer Programming to new people, and it's helpful when two experts discuss a subtle piece of modeling, but I have never seen a comprehensive system of equations add value to a large, well-written code base. Even so, if you do wish to incorporate a system of equations into your work product, my advice is to do so via the LaTeX functionality provided by Jupyter notebooks, as is demonstrated here and here. A notebook can easily be converted to a PDF file, as one only needs to follow the GitHub link to see nicely rendered equations.

The 0.0.3 solver should open a subset of the cities as centroids, and then assign each city to exactly one of the opened cities. A city is allowed to be assigned to itself, but only if it is one of the opened cities. The "Number of Centroids" parameter determines the number of cities to open. (See here for a demonstration of how to use optionally provided input parameters with the `create_full_parameters_dict` function.) A city-to-city assignment is allowed only if there is a corresponding record in the distances table. As discussed above, the distances table is bi-directionally safe, and thus an assignment between cities A and B is allowed if either (A, B) or (B, A) is present in the distances table. The objective function is to minimize the demand weighted total distance of the city-to-city assignments. That is to say, if city B has demand 5, and is 100 miles from city A, then assigning B to A will contribute 500 to the objective function.

For the 0.0.3 release, you need to extend the `solve` function to create this MIP, solve it, and translate the results into the tabular reports defined by the solution schema. (I defined the solution schema in the 0.0.1 .py files). The 0.0.3 `solve` function should be able to find feasible solutions for both the `MIP_for_9_City_Example.json` data set and the `LargeScale3Location.json` data set. The expected outcomes of those solves are described here. Note that I require you to solve the `MIP_for_9_City_Example.json` data set three times, with slight variations made to the `dat` object prior to solving. (See here and here for examples of editing the `dat` object and resolving).

The `solve` function I wrote for this release involved creating two sets of binary variables.

- An assignment variable was created for each city-city pair, provided the pair (in either order) was present in the distances table.
- An opening variable was created for each city.

I then created a constraint for every assignment variable, to ensure that if the assignment variable was turned on, then the "assigned to" city of the assignment variable was also turned on; that is, you can't assign B to A unless A is opened.

I also created a constraint for each city ensuring that exactly one assignment variable specifying the assignment of this city was turned on. (I.e. each city is a demand point that must be assigned to exactly one city).

Finally, there is a constraint to require that the correct number of cities be opened (whatever is specified by the “Number of Centroids” parameter). The objective is to minimize the total weighted assignment distance.

Release 0.0.4

The 0.0.3 release allowed for only a single objective: minimizing the total weighted assignment distance. In real life, you are better off creating Key Performance Indicators (KPIs) for your model, and then allowing the user to select whichever KPI she wants as the objective. In addition, the user should also be given the option to restrict a KPI. This allows for something roughly akin to lexicographical optimization in order to manage multiple objectives. True lexicographical optimization might also be warranted, but for less sophisticated users, “you can minimize X while bounding Y to keep it from getting out of hand” is far more intuitive. Finally, whatever KPI’s you define should be reported in your solution.

For the 0.0.4 release we define two KPIs

- Average Service Distance - this is the average assignment distance, weighted by demand. In other words, the objective value of a 0.0.3 solve, divided by the total demand.
- Percent High Service Demand - this is the percentage of demand that is met with high service. Demand is met with high service when it is assigned to a city within a predetermined threshold distance. All other demand is met with low service.

A solution generated for 0.0.4 should report on both of those KPIs, regardless of which was selected by the user to be the optimization objective.

In the directory here, I provide partial `tts_scnd.py` and `tts_scnd_pd.py` files. I only define the new input schema, which is changed only by adding five new input parameters.

- High Service Distance: This is the threshold distance required to determine the Percent High Service KPI. Any assignment that is within this distance is high service. Since the default here is zero, then, by default, only self-assigned demand will be high service. For a typical run, the user will supply this parameter with an appropriate value.
- Maximum Average Service Demand: This is an upper bound on the Average Service Distance KPI. Since it is infinity by default, then the default behavior is to allow the Average Service Demand to be as large as is needed. A user would typically supply this parameter only if the objective was to maximize the Percent High Service Demand KPI.
- Minimum Percent High Service Demand: This is a lower bound on the Percent High Service Demand KPI. Since it is zero by default, then the default behavior is to allow the Percent High Service Demand to be as small as is needed. A user would typically supply this parameter only if the objective was to minimize the Average Service Distance.

- Maximum Individual Service Distance: This allows for the user to restrict the largest possible distance in the solution assignments. In principle, this constraint isn't needed. For example, instead of providing a value of 700 here, the user could just remove every record from the distances table with a distance larger than 700. This parameter is provided as a convenient alternative. The logic behind a constraint like this is that the MIP engine is not concerned at all with the aesthetics of a solution, and thus might assign a very small demand point to a very distant city because the impact of doing so is negligible on the Average Service Distance. If you do a lot of work with MIPs you might similarly discover constraints that are applied only to "clean" a solution and not to meaningfully improve a solution KPI.
- Objective: This can be set to one of two strings - "Minimize Average Service Distance" or "Maximize Percent High Service Demand". The typical user would select one KPI to be the objective, while optionally restricting the other.

The Maximum Individual Service Distance constraint should be the easiest to implement. All you need to do here is refrain from generating a boolean assignment variable if the assignment distance is too large.

The easiest way to implement the two KPIs is to create dedicated variables that measure their value. These are two continuous variables, unlike the binary variables that we are used elsewhere. When you create the Average Service Distance variable, you set its upper bound from the Maximum Average Service Distance parameter. Similarly, the Percent High Service Distance variable gets its lower bound from the Minimum Percent High Service Demand parameter. Each KPI variable needs a dedicated constraint to make sure that the MIP sets the KPI to the correct value based on which assignment variables are turned on. Finally, the objective function will simply be whichever of the KPIs the user has selected based on the Objective parameter value. (Don't forget that one of the KPIs will be minimized as the objective, while the other will be maximized).

Some people make the modeling mistake of thinking that the MIP objective function must itself be a long complicated expression. I have found that the simpler the objective function, the easier troubleshooting becomes. I typically use an objective value that involves, at most, a small number of KPI variables. If the user is allowed to select from different objectives, then the simplest solution is to use the appropriate single-variable objective function, as I demonstrate here.

The specific requirements of the 0.0.4 unit tests are outlined here. Note that there is no need to forward convert the testing data, as the new parameters all have sensible default values. You will need to rescale the validation numbers you look for in the unit tests you're carrying over from the 0.0.3 release, as the Average Service Distance objective is not the exact same as the 0.0.3 objective. That is, if the solution objective value for a 0.0.3 solution was X, then the Average Service Distance KPI for the 0.0.4 solution should be X divided by the total demand.

Release 0.0.5

The 0.0.5 release adds a new constraint to the total amount of demand that can be assigned to an opened city. We add the field Max Assignment Capacity field to the cities table. This is a non-negative numeric field that is allowed to be infinite (to model a city that has no restriction on the demand that can be assigned to it). To be fully clear, if cities *a*, *b*, and *c* are assigned to city *d*, then the sum total demand of *a*, *b* and *c* can't exceed the Max Assignment Capacity of *d*.

To implement the 0.0.5 release, you will have to forward convert both the `MIP_for_9_City_Example.json` data set and the `LargeScale3Location.json` data set. As of 0.0.4, these two files are consistent with an input schema that has two fields (Name and Demand) for the cities table. As part of the 0.0.5 release, you will have to adjust these files to be consistent with a cities table that has Name, Demand and Max Assignment Capacity. Since the 0.0.4 release didn't apply a maximum assigned demand constraint to an opened city, this conversion process should populate the Max Assignment Capacity with `float(inf)` (which in JSON is rendered as `Infinity`) for every record of the cities table. Both the `tts_netflow_a` and `tts_netflow_b` repos include notebooks that demonstrate the forward conversion of testing data.

Of course, the 0.0.5 release actually has to extend the `solve` function to implement this new constraint. There are a few different ways to do this. Of the two that I tried, one will have a marginally tighter linear relaxation, and the other is marginally easier to implement. If I was grading this as a course, I would accept either one.

This is probably a good time to point out that `gurobipy` will allow you to use infinity as the RHS of a less-than-equals constraint, but it won't allow you to use infinity as a variable coefficient. There is sound reasoning for this logic: infinity is, by definition, the RHS that allows any less-than-equals constraint to be satisfied. Thus, by allowing infinity as the RHS to a less-than-equals constraint, Gurobi allows the programmer to use `float(inf)` in a manner consistent with how it's used in broader Python. (As far as I know, the other APIs do this as well). However, if infinity were allowed as a coefficient, then it becomes impossible to even evaluate whether or not a constraint is satisfied by a given solution vector. This is because the result of multiplying infinity by zero is, by definition, not a number.

At any rate, if you implement the city-specific Max Assignment Capacity constraint correctly then the new unit test that I define here will pass.

Chapter 11 - A Baby Network Design Model

Chapter 10 guided you through the first five releases of the network design tool we are developing. This chapter is dedicated to the sixth and final release of the course. To be honest, if you completed the first five releases successfully, then I'd be happy to have you as a colleague. You can organize code into a repo that itself deploys a package. Each iteration of the package is tagged as a proper release. You've developed well-organized regression tests for each release, and maintained those tests through functional enhancements. Your solve function is bulletproof against the input data imperfections you're bound to encounter in the real world.

I think the right thing for me to do at this point is thank you for the attention and effort you've put in so far. If you want to check out here, you can do so with my blessing. Anyone getting this far understands both the concept and the practice of Tidy, Tested, Safe.

If you'd prefer to do one more round, then the next one is a big step towards a real-world network design tool. The number of input tables is going to jump up from three to ten. Instead of generic cities, the input data will involve plants, warehouses and customers. We will also add a products table, and total cost will be added to our solution KPIs. In short, you will be building a baby LogicNet Plus, but with Python.

At the risk of making the gods laugh, I'm predicting that in the not too distant future the strategic network design space will no longer be dominated by large pre-built applications like LNP or Supply Chain Guru. Instead, the community of recent IEOR graduates will discover that building a basic network design tool isn't very hard. Tools will be spun up from scratch, or customized from stock, based on the idiosyncratic needs of each client. The idea that "the tool can't do exactly that, but here is a work-around to perform something equivalent" will be banished by the pipeline pattern and the Python programming language.

I think this outcome will happen with or without me. It is an inevitable outcome of teaching both optimization and a real programming language to such a large and talented community. My hope is to make `ticdat` a part of this future. I've spent a long time kicking around the strategic network design space, and I want to leave it better than I found it. If somebody reads this book, takes this course, and then uses their 0.0.6 release to actually perform a network design study for a paying client, then nobody will be more pleased than me[16]. However, be very careful with any Gurobi academic license you have been issued. My understanding is such licenses are strictly for educational purposes. Please contact Gurobi directly if you have even the slightest doubt about whether you are using their license appropriately.

The New Input Schema

[16] I've taken the legal precaution of creating a non-profit as a legal shield, just in case some people are displeased.

I have provided you with new `tts_scnd.py` and `tts_scnd_pd.py` files which define the new input and solution schemas. You will have to create the new `solve` function for the 0.0.6 release.

There are ten input tables, including the parameters table. Walking through them will give you a good sense of the mathematical model you'll need to build. In my experience, a clear and well-defined description of the input and solution tables is typically more helpful than a LaTeX system of equations.

- The plants table has no data fields. Its purpose is simply to list plants. Plants are sites that produce products.
- The warehouses table has two data fields
    - The Max Assignment Capacity field serves a similar role as before. Each customer, product pair (i.e. each demand point) will need to be assigned to a warehouse. The total volume of assigned demand cannot exceed the Max Assignment Capacity of the warehouse.
    - The Fixed Cost is the cost associated with opening the warehouse. Demand points can only be assigned to open warehouses.
- The customers table has no data fields Its purpose is to simply list customers. Customers are sites that consume products.
- The products table has a single data field.
    - Warehouse Volume is the per SKU volume of the product when it's assigned to a warehouse as part of a demand point assignment. The total warehouse capacity consumed will be the sum of Demand times Warehouse Volume for every demand point assignment.
- The supply table determines the production capacity of every plant, product pair. If there is no supply record for a plant, product pair, then the plant is unable to produce that product.
- The plant-to-warehouse costs table determines the cost of shipping one unit of product between a plant and a warehouse. If a given plant, warehouse, product triplet is missing from this table, then such a shipment isn't possible.
- The warehouse-to-customer costs table is similar, except for warehouse-to-customer shipping.
- The warehouse to customer distances table lists the distances between warehouse and customer.
    - You'll notice I am defining a compound foreign key between these two tables, so that there is a distance associated with every warehouse-to-customer shipping record.

The New Solve Function

The 0.0.6 MIP broadly resembles a blend of the 0.0.5 MIP and the multi-commodity network flow MIP from the `tts_netflow` examples. I will summarize the variables and constraints I used to solve the 0.0.6 problem.

- Assignment variables. Similar to 0.0.5, each warehouse, customer, product triplet that has a corresponding entry in both the demand table and the warehouse-to-customer costs table generates a binary assignment variable. Of course, we fail to generate an assignment variable whose distance is larger than that specified by the Maximum Individual Service Distance parameter.
- Warehouse opening variables. Each warehouse generates a binary variable that determines if the warehouse is to be opened or not.
- Production variables. Each record in the supply table generates a continuous variable that specifies the amount of product that plant is to produce. The upper bound of this variable is drawn from the Supply field.
- Plant-to-warehouse shipping variables. For each record in the plant-to-warehouse table, we generate a continuous variable.
- KPI variables.
    - Average Service Distance is defined in the exact same way as 0.0.5. Note that this is computed only with respect to the warehouse-to-customer assignments.
    - The same statement can be made about Percent High Service Demand.
    - The Total Cost is defined as the aggregate sum of the costs of opening warehouses, shipping goods from plants to warehouses, and shipping goods from warehouses to customers.
    - There are thus three KPI constraints, one for each of the KPI variables.
- Similar to before, there is a dedicated constraint for each assignment variable to ensure that an assignment is only made to a warehouse that has been opened.
- There is a constraint for each demand point that ensures that it is assigned to exactly one warehouse.
- Each warehouse has a constraint that ensures that the total volume of assigned demand doesn't exceed the Max Assignment Capacity.
- There are two categories of conservation-of-flow constraints. For each category, care is taken not to generate the constraint for all pairs, but only for pairs where there is an entry on at least one side of the constraint.
    - Conservation of flow for plant-product pairs ensures that the total outflow of plant-to-warehouse shipping matches the production.
    - Conservation of flow for warehouse-product pairs ensures the total inflow of plant-to-warehouse shipping matches the total assigned demand.
- There is a constraint to ensure that the number of warehouses opened is equal to the "Number of Warehouses" parameter.

Forward Converting the 0.0.5 Testing Data

Before we dig into the forward conversion code, let me just remind you why I like forward conversions of testing data so much. This task brings together two of my favorite things.

- Data munging
- Problem reduction

The first is something Python is particularly good at. The second is something that algorithmically inclined programmers are good at. If you're reading this book, you're an algorithmically inclined Python programmer. You can do this.

- The 0.0.5 "Number of Centroids" parameter should populate the 0.0.6 "Number of Warehouses" parameter.
    - Don't forget to copy over the other 0.0.5 parameters as-is.
- The 0.0.6 data set you are creating will need a single product. For simplicity, give this product the name "P". Product P will need to have a Warehouse Volume of 1.
- For each record in the 0.0.5 cities table
    - Create a 0.0.6 plant with the same name.
        - Create a 0.0.6 supply record for the plant you just created and product "P". The Supply for this record needs to be infinity.
    - Create a 0.0.6 warehouse with the same name and Max Assignment Capacity and a Fixed Cost of zero.
    - Create a 0.0.6 customer with the same name.
        - Create a demand record for the customer you just created and product "P". The Demand for this record needs to be the same as the Demand for the 0.0.5 city that sourced it.
- For each record of the 0.0.5 distances table.
    - Create a record in the 0.0.6 warehouse-to-customer distances table with the same source, destination and distance.
    - Create a record in the 0.0.6 warehouse-to-customer costs table with the same source and distance, with product set to "P", and with a cost of zero.
    - Bear in mind that the 0.0.5 solver reads the distances table bi-diretionally, but there is no equivalent functionality for the 0.0.6 solver. (This is because the 0.0.6 solver explicitly separates the locations into plants, warehouses and customers, so the idea of reversing the source and destination when looking up shipment information makes no sense.) To accommodate this change in functionality from 0.0.5 to 0.0.6, you might have to repeat the two row creation steps above, but this time, reverse the source and destination. Particularly with `PanDatFactory`, you need to be careful not to perform this additional step if doing so creates a duplicate row.
- For each pair of 0.0.5 cities, create a corresponding plant-to-warehouse costs record with product of "P" and cost of zero.
- As always, I recommend running the 0.0.6 integrity checks on the `dat` object you just created prior to saving it to memory.

Note that you might not discover a goof in your forward conversion logic until you run the 0.0.6 unit tests and discover that a test that ought to be retained is failing to pass. What I sometimes do in this case is bounce back and forth between the development branch I'm using to build release N and the code pulled from release N-1 until I discover my mistake. At this point, it often makes sense to just start again with a new development branch forked from the N-1 release, which can easily be populated with a corrected version of the logic from the old development

branch. My point: making new branches is easy, copying code is easy, and it's no big deal to declare a development branch hopelessly knackered and kill it after the useful code changes have been salvaged.

The Unit Test Sanity Checks

Chapter 6 has a complete answer to the question "Why don't you define integrity checks to the solution data?". The very short answer is that it makes more sense to perform bespoke sanity checks on the solution data as part of the unit testing code instead.

With this in mind, my 0.0.6 unit tests double checked the feasibility of every solution created by `solve`. The sanity checks I performed on the `solve` return object were as follows.

- Each demand point in the input data should be assigned to exactly one warehouse by the warehouse-to-customer report in the solution data.
- There should be no flow imbalances for any plant-product pair. The units produced should balance with the outbound flow.
- There should be no flow imbalances for any warehouse-product pair. The inbound flow should balance with the outbound flow.
- Any warehouse-product pair with outbound flow should be opened.
- The total cost reported as a KPI in the parameters table aligns with the sum of the Fixed Cost from the warehouses opened report, plus the sum of the Shipment Cost field from the plant-to-warehouses shipments report, plus the sum of the Shipment Cost field from the warehouse-to-customers report.

The full set of requirements for the 0.0.6 release, and the tests you should perform with the new `100_big_cities.json` data set, are detailed here.

About the Author

Peter Cacioppi has been masquerading as an optimization expert since 1996. His trick is to apply the principles of computer science to computational problems in the Industrial Engineering and Operations Research space. Feel free to connect with Pete at peter.cacioppi@ttspython.org.