O'REILLY®

Robust Python

Write Clean and Maintainable Code

Early Release

RAW & UNEDITED

Patrick Viafore

Robust Python

Write Clean and Maintainable Code

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Patrick Viafore

Robust Python

by Patrick Viafore

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Amanda Quinn

Development Editor: Sarah Grey

Production Editor:

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator:

February 2022: First Edition

Revision History for the Early Release

- 2020-11-06: First Release
- 2020-12-03: Second Release

- 2021-01-19: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098100667 for release details.

978-1-098-10066-7

[LSI]

Chapter 1. Introduction to Robust Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

This book is all about making your Python better. To help you manage your codebase, no matter how large it is. To provide a toolbox of tips, tricks and strategies to build maintainable code. This book will guide you towards less bugs and happier developers. You'll be taking a hard look at how you write code, and learn the implications of your decisions. When discussing how code is written, I am reminded of these wise words from C.A.R. Hoare:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*[1]

This book is about developing systems the first way. It will be more difficult, yes, but have no fear. I will be your guide on your journey to leveling up your Python game such that, as C.A.R. Hoare says above, *there are obviously no deficiencies* in your code. Ultimately, this is a book all about writing *robust* Python.

In this chapter I'm going to cover what *robustness* means and why you should

care about it. I'll go through how your communication method implies certain benefits and drawbacks, and how best to represent your intentions. The Zen of Python states that *there should be one-- and preferably only one — obvious way to do it*. You'll learn how to evaluate if your code is the obvious way or not, and what you can do to fix it. First, I need to talk basics. What is *robustness* in the first place?

Robustness

What Does "Robust" Mean?

Every book needs at least one dictionary definition, so I'll get this out of the way nice and early in the book. Merriam-Webster offers many definitions for *robustness* [2]:

1. having or exhibiting strength or vigorous health
2. having or showing vigor, strength, or firmness
3. strongly formed or constructed
4. capable of performing without failure under a wide range of conditions

These are fantastic descriptions of what we are aiming for. We want a *healthy* system, one that stays bug-free for years. We want our software to *exhibit strength*; it should be obvious that this code will stand the test of time. We want a *strongly constructed* system, one that is built upon solid foundations. Crucially, we want a system that is *capable of performing without failure*; we don't want the system to be fragile or brittle as conditions change.

It is common to think of a software like a skyscraper, some grand structure that stands as a bulwark against all change and a paragon of immortality. The truth is, unfortuately messier. Software systems constantly evolve. Bugs are fixed, user interfaces get tweaked, features are added, removed, and then re-added. Frameworks shift, components go out of date, security bugs arise. Software changes. It is more akin to handling sprawl with city planning than it is building a static building. With ever changing codebases, how can you make your code robust? How can you build a strong foundation that is resilient to bugs?

The truth is, you have to accept the change. Your code will be split apart, stitched together and reworked. New use cases will alter huge swaths of code. And that's okay. Embrace it. Understand that it's not enough that your code can easily be changed; it might be best for it to be deleted and rewritten as it goes out of date. That doesn't diminish its value; it will still have a long life in primetime. Your job is to make it easy to rewrite parts of the system. Once you start to accept the ephemeral nature of your code, you start to realize that it's not enough to write bug-free code for the present; you need to enable the codebase's future owners to be able to change your code with confidence. That is what this book is about.

You are going to learn to build strong systems. This strength doesn't come from rigidity, like a bar of iron. It instead comes from flexibility. Your code needs to be strong like a tall willow tree, swaying in the wind, flexing, but not breaking. Your software will need to handle situations you would never dream of. Your codebase needs to be able to adapt to new circumstances, and it won't always be you maintaining it. Those future maintainers need to know they are working in a healthy codebase. Your codebase needs to communicate it's strength. You must write Python code in a way that reduces failure, even as future maintainers tear it apart and reconstruct it.

Writing robust code means deliberately thinking about the future. You want future maintainers to look at your code and understand your intentions easily, not curse your name during late night debugging sessions. You must convey your thoughts, your reasoning, and cautions. Future Developers need to bend your code into new shapes, and do it without worrying that each change knocks over a teetering house of cards.

Put simply, you don't want your systems to fail, especially when the unexpected happens. Testing and quality assurance are huge parts of this, but neither of those bake quality completely in. They are more suited to illuminating gaps in expectations and offering a safety net. Instead, you must make your software stand the test of time. In orrder to do that, you must write *clean and maintainable* code.

Clean code expresses its intent clearly and concisely, in that order. When you look at a line of code and say to yourself "ah, that makes complete sense", that's an indicator of clean code. The more you have to step through a debugger, the more you have to look at a lot of other code to figure out what's happening, the

more you have to look at a lot of other code to figure out what's happening, the more you have to stop and stare at the code, the less clean it will be. Clean code does not favor clever tricks if it makes the code unreadable to other developers. Just like C.A.R. Hoare said earlier, you do not want to make your code so obtuse that it will be difficult to visually inspect it to understand it.

Maintainable code is code that, well, can be easily maintained. Maintenance starts immediately after the first commit and continues until there is not a single developer looking at the project anymore. Developers will be fixing bugs, adding features, reading code, extracting code for use in other libraries, etc. Maintainable code makes these tasks frictionless. Software lives for years, if not decades, so you need to focus on maintainability *today*.

You don't want to be the reason systems fail, whether you are actively working on them or not. You need to be proactive in making your system stand the test of time. You need a testing strategy to be your safety net, but you also need to be able to avoid falling in the first place. So with all that in mind, I offer my definition of robustness in terms of software:

Robust software is resilient and error-free, in spite of constant change

Why Does Robustness Matter?

A lot of energy goes into making software do what it is supposed to. Development milestones are not easily predicted. It doesn't help that you build something brand-new just about every time. Human factors such as UX, accessibility and documentation only increase the complexity. Now add in testing to ensure that you've covered a slice of known and unknown behaviors, and you are looking at lengthy cycles.

The purpose of software is to provide value. It is in stakeholder's interests to deliver that full value as early as possible. Given the uncertainty around some development schedules, there is often extra pressure to meet expectations. We've all been on the wrong end of an unrealistic schedule or deadline. Unfortunately, many of the tools to make software incredibly robust only add onto our development cycle.

This does not mean that robust code is unimportant or "not worth it". It's true that there is an inherent tension between immediate delivery of value and making code robust. If your software is "good enough", why add even more

making code robust. If your software is "good enough", why add even more complexity? To answer that, consider how often that piece of software will be iterated upon. Delivering software value is typically not a static exercise; it's rare that a system provides value and is never modified again. Software is ever-evolving by its very nature. The codebase needs to be prepared to deliver value frequently, and for long periods of time. This is where robust software engineering practices come into play. If you can't painlessly deliver features quickly and without compromising quality), you need to re-evaluate techniques to make your code more maintainable.

If you deliver your system late, or broken, there are real-time costs that are incurred. Think through your codebase. Ask yourself what happens if your code breaks a year from now because someone wasn't able to understand your code. How much value do you lose? Your value might be measured in money, time, or even lives. Ask yourself what happens if the value isn't delivered on time? What are the repercussions? If the answers to these questions are scary, good news, the work you're doing is valuable. But it also underscores why it's so important to eliminate future errors.

You need to consider future developers. Multiple developers work on the same codebase simlutaneously. Many software projects will outlast most of those developers. You need to find a way to communicate to the present and future developers, without having the benefit of being there in person to explain. Future developers will be building off of *your* decisions. Every false trail, every rabbit hole, and every yak-shaving[3] adventure will slow them down, which impedes value. You need empathy for those who come after you. You need to step into your shoes. This book is your gateway to thinking about your collaborators and maintainers. You need to write code that lasts. The first step to making code that lasts is being able to communicate through your code. You need to make sure future developers understand your intent.

What's Your Intent?

Why should you strive to write clean code? Why should you care so much about robustness? The heart of these answers lies in communication. You're not delivering static systems; software evolves and grows over time. Maintainers change over time. Your goal, when writing code, is to deliver value, but it's also to write your code in such a way that other developers can deliver value just as

to write your code in such a way that other developers can deliver value just as quickly. In order to do that, you need to be able to communicate reasoning and intent without ever meeting your future maintainers.

Let's take a look at a code block found in a hypothetical legacy system. I want you to estimate how long it takes for you to understand waht this code is doing. It's okay if you're not familiar with all the concepts here, or if you feel like this code is convoluted (it intentionally is!).

```
# Take a meal recipe and change the number of servings
# by adjusting each ingredient
# A recipe's first element is the number of servings, and the
remainder
# of elements is (name, amount, unit), such as ("flour", 1.5, "cup")
def adjust_recipe(recipe, servings):
    new_recipe = [servings]
    old_servings = recipe[0]
    factor = servings / old_servings
    recipe.pop(0)
    while recipe:
        ingredient, amount, unit = recipe.pop(0)
        # please only use numbers that will be easily measurable
        new_recipe.append((ingredient, amount * factor, unit))
    return new_recipe
```

This function takes a recipe, and adjusts every ingredient to handle a new number of servings. however, this code breeds many questions.

- What is the `pop` for?
- What does `recipe[0]` signify? Why is that the old servings?
- Why do I need a comment for numbers that will be easily measurable?

This is a bit of questionable python, for sure. I won't blame you if you feel the need to rewrite it. It looks much nicer if it were something like this:

```
def adjust_recipe(recipe, servings):
    old_servings = recipe.pop(0)
    factor = servings / old_servings
    return ({"servings": servings} |
            {ingredient: (amount*factor, unit)}
            for ingredient, amount, unit in recipe
```

Those who favor clean code probably prefer the second version (I certainly do).

No raw loops. Variables do not mutate. I'm returning a dictionary instead of a list of tuples. All these changes can be seen as positive, depending on the circumstances. But I may have just introduced three subtle bugs.

1. In the first example, I was clearing the original recipe out. Even if it's just one area of calling code that is relying on this behavior, I broke assumptions.
2. By returning a dictionary, I have removed the ability to have duplicate ingredients in a list. This might have an effect on recipes that have multiple parts (such as a main dish and a sauce) that both use the same ingredient.
3. If any of the ingredients are named "servings" you've just introduced a collision with naming.

Whether these are bugs or not depends on two inter-related things: the author's intent and calling code. The author intended to solve a problem, but I am unsure of why they wrote the code the way they did. Why are they popping elements? Why is servings a tuple inside the list? Why is a list used? Presumably, the author knew why, and communicated it locally to their peers. Their peers wrote calling code based on those assumptions, but as time wore on, that intent became lost. Without communication to the future, I am left with two options of maintaining this code:

1. Look at all calling code and confirm that this behavior is not relied upon before implementing. Good luck if this is a public API for a library with external callers. I spend a lot of time doing this, which frustrates me.
2. Make the change and wait to see what the fallout is (customer complaints, broken tests, etc.). If I'm lucky, nothing bad will happen. If I'm not, I spend a lot of time, which frustrates me.

Neither option feels good in a maintenance setting (especially if I have to modify this code). I don't want to waste time; I want to deal with my current task quickly and move on to the next one. It gets worse if I consider how to call this code. Think about how you interact with previously unseen code. You might see other examples of calling code, copy them to fit your use case, and never realize that you needed to pass a specific string called servings as your first element of

your list.

These are the sort of decisions that will make you scratch your head. We've all seen them in larger codebases. They aren't written maliciously, but organically over time with the best intentions. Functions start simple, but as use cases grow and multiple developers contribute, that code tends to morph and obscure original intent. This is a sure sign that maintainability is suffering. You need to express intent in your code upfront.

So what if the original author made use of better naming patterns and better type usage? What would that code look like?

```
# Take a meal recipe and change the number of servings
# recipe should be a Recipe class
def adjust_recipe(recipe, servings):
    new_ingredients = list(recipe.ingredients)
    recipe.clear_ingredients()

    for ingredient in new_ingredients:
            ingredient.adjust_propoprtion(Fraction(servings,
recipe.servings))
    return Recipe(servings, new_ingredients)
```

This looks much better, and expresses original intent clearly. The original developer encoded their ideas directly into the code. From this snippet, you know the following is true:

- I am using a `Recipe` class. This allows me to abstract away certain operations. Presumably, inside the class itself, there is an invariant that allows for duplicate ingredients. (I'll talk more about classes and invariants in Chapter 5.) This provides a common vocabulary that makes the function's behavior more explicit.
- Servings are now an explicit part of a recipe class, rather than needing to be the first element of the list, which was handled as a special case. This greatly simplifies calling code, and prevents inadvertent collisions.
- It is very apparent that I want to clear out ingredients on the old recipe. No ambiguous reason for why I needed to do a `.pop(0)`.
- Ingredients are a separate class, and handle fractions rather than an

explicit float. It's clearer for all involved that I am dealing with fractional units, and can easily do things such as `limit_denominator()`, which can be called when people want to restrict measuring units (instead of relying on a comment)

I've replaced fields with types, such as a recipe type and an ingredient type. I've also defined operations (`clear_ingredients`, `adjust_proportion`) to communicate my intent. By making these changes, I've made the code's behavior crystal clear to future readers. They no longer have to come talk to me to understand the code. Instead, they comprehend what I'm doing without ever talking to me. This is *asynchronous communication* at its finest.

Asynchronous Communication

It's weird talking about asynchronous communication in a Python book without mentioning `async` and `await`. But I'm afraid I have to talk about asynchronous communication in a much more complex place: *the real world*.

Asynchronous communication means that producing information and consuming that information are independent of each other. There is a time gap between the production and consumption. It might be a few hours, as is the case of collaborators in different time zones. Or it might be years, as future maintainers try to do a deep dive into the inner workings of code. You can't predict when somebody will need to understand your logic. You might not even be working on that codebase (or for that company) by the time they consume the information you produced.

Contrast that with *synchronous communication*. Synchronous communication is when people talk face-to-face (in-person or otherwise) and share knowledge. This form of direct communication is one of the best ways to express your thoughts, but unfortunately, it doesn't scale, and you won't always be around to answer questions.

In order to evaluate how appropriate each method of communication is when trying to understand intentions, I'll look at two axes: proximity and cost.

Proximity is how close in time the communicators need to be in order for that communication to be fruitful. Some methods of communication excel with real-time transfer of information. Other methods of communication excel at

communicating years later.

Cost is the measure of effort to communicate. You must weigh the time and money expended to communicate with the value provided. Your future consumers then have to weigh the cost of consuming the information with the value they are trying to deliver. Writing code and not providing any other communication channels is your baseline; you have to do this to produce value. To evaluate additional communication channel's cost, here is what I factor in:

- Discoverability: How easy was it to find this information outside of a normal workflow? How ephemeral is the knowledge? Is it easy to search for information?
- Maintenance Cost: How accurate is the information? How often does it need to be updated? What goes wrong if this information is out of date?
- Production Cost: How much time and money went into producing the communication?

In Figure 1-1, I plotted some common communication methods' cost and proximity required.

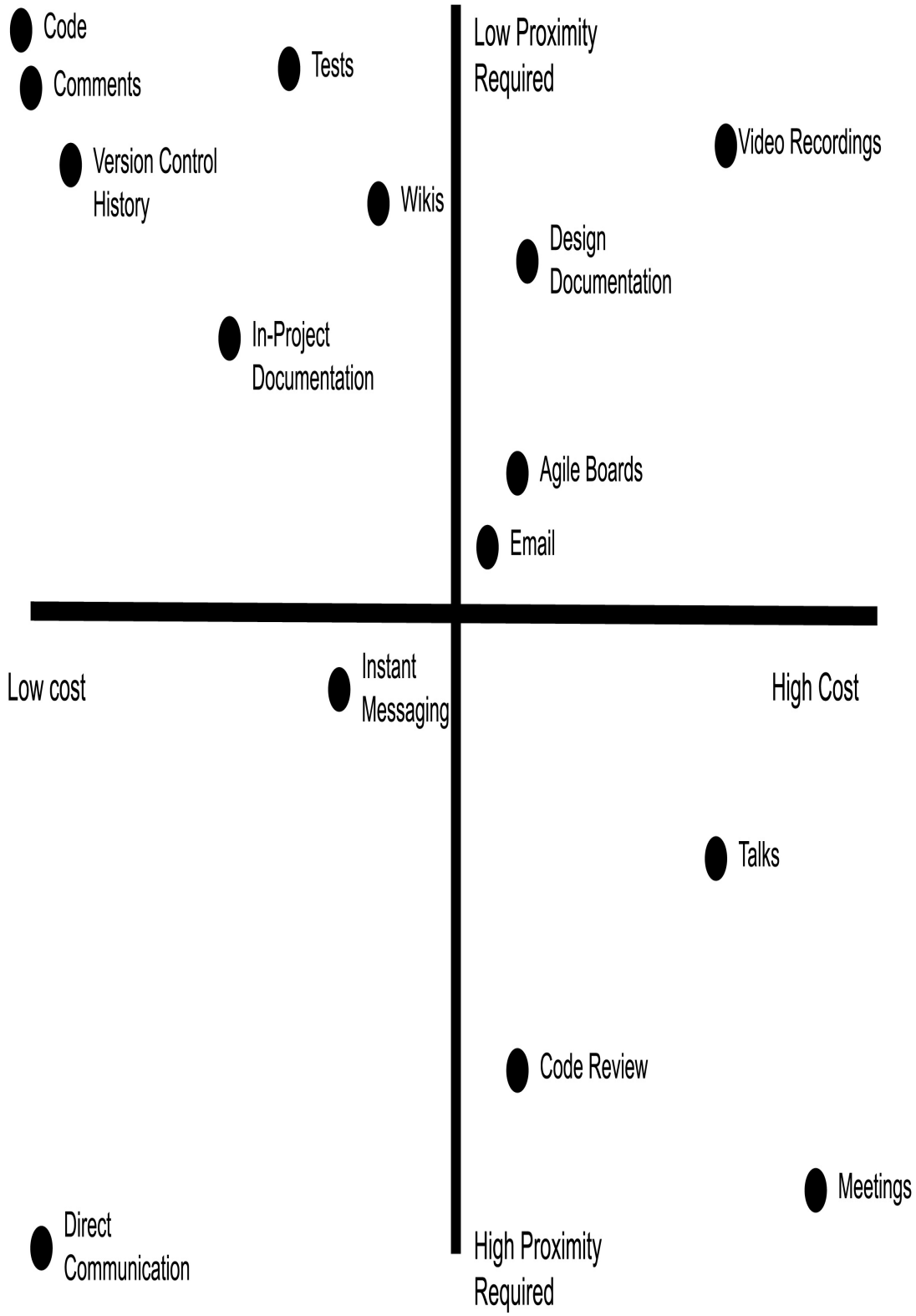
Code
Comments
Tests
Version Control History
Wikis
In-Project Documentation
Low Proximity Required
Video Recordings
Design Documentation
Agile Boards
Email
Low cost
Instant Messaging
High Cost
Talks
Code Review
Meetings
Direct Communication
High Proximity Required

Figure 1-1. Plotting Cost and Proximity of Communcation Methods

There are 4 quadrants that make up the cost/proximity graph.

Low Cost, High Proximity Required

These are cheap to produce and consume, but are not scalable across time. Direct communication and instant messaging are great examples of these methods. Treat these as snapshots of information in time; they are only valuable when the user is actively listening. Don't rely on these methods to communicate to the future.

High-cost, High Proximity Required

These are costly events, and often only happen once (such as meetings or conferences). There should be a lot of value delivered through these events at the time of communication, because they do not provide much value to the future. How many times have you been to a meeting that felt like a waste of time? This is direct loss of value you are feeling. Talks require a multiplicative cost for each attendee (time spent, hosting space, logistics, etc.). Code reviews are rarely looked at once they are done.

High Cost, Low Proximity Required

These are costly, but that cost can be paid back over time in value delivered, due to the low proximity needed. Emails and agile boards contain a wealth of information, but are not discoverable by others. These are great for bigger concepts that don't need frequent updates. It becomes a nightmare to try and sift through all the noise just to find the nugget of information you are looking for. Video recordings and design documentation are great for understanding snapshots in time, but are costly to keep updated. Don't rely on these communication methods to understand day-to-day decisions.

Low Cost, Low Proximity Required

These are cheap to create, and are easily consumable. Code comments, version control history and project READMES all fall into this category, since they are adjacent to the source code we write. Users can view this communication years after it was produced. Anything that is in a developer's workflow will become easily discoverable. These communication methods

are a natural fit for the first place someone will look after the source code. However, your code is one of your best documentation tools, as it is the living record and single source of truth for your system.

DISCUSSION TOPIC

This plot was created based on generalized use cases - think about the communication paths you and your organization uses. Where would you plot them on the graph? How easy is it to consume accurate information? How costly is it to produce information? Your answers to these questions may result in a slightly different graph, but the single source of truth will be in the executable software you deliver.

Low cost, low proximity communication methods are the best tool for communicating to the future. You should strive to minimize the cost of production and of consumption of communication. You have to write software to deliver value anyway, so the lowest cost option is making your code your primary communication tool. Your codebase becomes the best possible option for expressing your decisions, opinions, and workarounds clearly.

However, for this assertion to hold true, the code has to be cheap to consume as well. Your intent has to come across clearly in your code. Your goal is to minimize the time needed for a reader of your code to understand it. Ideally, a reader does not need to read your implementation, but just your function signature. Through the use of good types, comments and variable names, it should be crystal clear what your code does.

SELF-DOCUMENTING CODE

The wrong response to this plot is “Self-documenting code is all I need!” Every communication path provides value that code alone won’t be able to. Version control will give you a history of changes. Design Documents discuss sweeping ideals that are not local to any one code file. Meetings (when done right) can be an important event for synchronizing plan execution. You should absolutely strive for code to be self-documenting, but realize that just handles *what* the code is doing. Don’t disparage any other communication path.

The other quadrants of communication are still valuable. Design

documentation absolutely has a place for big picture decisions. Talks are an incredibly effective way of sharing your ideas across large audiences. Meetings (when done effectively) are essential to act as a sync point between interconnected teams. Do not discount these methods, but understand that each communication method is tailored for specific use cases. This book focuses on what you can do in your code, but do not rely on just code to communicate your intent.

Examples of Intent In Python

Now that I've talked through what intent is and how it matters, let's look at examples through a Python lens. How can you make sure that you are correctly expressing your intentions? Consider some common mistakes you come across when reading Python code?

Collections

When you pick a collection, you are communicating specific information. You must pick the right collection for the task at hand. Otherwise, maintainers will infer the wrong intention from your code.

Consider this code that takes a list of cookbooks and provides a count of how many times an author shows up:

```
def create_author_count(cookbooks: List[Cookbook]):
    counter = {}
    for cookbook in cookbooks:
        if cookbook.author not in counter:
            counter[cookbook.author] = 0
        counter[cookbook.author] += 1
    return counter
```

What does my use of collections tell you? Why am I not passing a dictionary or a set? Why am I not returning a list? Based on my current usage of collections, here's what you can assert:

- I pass in a list of cookbooks. There can be duplicate cookbooks in this list (I might be counting a shelf of cookbooks in a store with multiple

copies).

- I am returning a dictionary. Users can look up a specific author, or iterate over the entire dictionary. I do not have to worry about duplicate authors in the returned collection.

What if I wanted no duplicates in the list? A list communicates the wrong intention. Instead, I should have chosen a set to communicate that this code absolutely will not handle duplicates.

Choosing a collection tells readers about your specific intentions. Here's a list of common collection types, and the intentions they convey:

List

This is a collection to be iterated over. It is *mutable*: able to be changed at any time. Very rarely do you expect to be retrieving specific elements from the middle of the list (using a static list index). There may be duplicate elements. The cookbooks on a shelf might be stored in a list.

String

An immutable collection of characters. The name of a cookbook would be a string.

Generators

A collection to be iterated over, and never indexed into. Each element access is performed lazily, so it may take time and/or resources through each loop iteration. They are great for computationally expensive or infinite collections. An online database of recipes might be returned as a generator; you don't want to fetch all the recipes in the world when the user is only going to look at the first ten results of a search.

Tuple

Tuples are immutable collections. You do not expect it to change, so it is more likely to extract specific elements from the middle of the tuple (either through indices or unpacking). It is very rarely iterated over. The information about a specific cookbook might be represented as a tuple, such as `(cookbook_name, author, pagecount)`

Set

An iterable collection that contains no duplicates. You cannot rely on ordering of elements. The ingredients in a cookbook might be stored as a set.

Dictionary

A mapping from keys to values. Keys are unique across the dictionary. Dictionaries are typically iterated over, or indexed into using dynamic keys. A cookbook's index is a great example of a key to value mapping (from topic to page number.)

Do not use the wrong collection for your purposes. Too many times have I come across a list that should not have had duplicates or a dictionary that wasn't actually being used to map keys to values. Everytime there is a disconnect between what you intend and what is in code, you create a maintenance burden. Maintainers must pause, work out what you really meant, and then work around their faulty assumptions.

DYNAMIC VS. STATIC INDEXING

Depending on the collection type you are using, you may or may not want to use a *static index*. A static index is when you always have the same index into the collection, regardless of collection, such as `my_list[4]` or `my_dict["Python"]`. In general, lists, and dictionaries will not often need a use case for this. You have no guarantee that the collection has the element you are looking for at that index, due to their dynamic nature. If you are looking for specific fields in these types of collections, this is a good sign that you need a user-defined type (explored in later sections). It is safe to use a static index into tuples, since they are fixed size. Sets and generators are never indexed into.

Exceptions to this rule include:

- Getting the first or last element of a sequence (`my_list[0]` or `my_list[-1]`)
- Using a dictionary as an intermediate data type such as reading

JSON or YAML

- Operations on a sequence that specifically deal with fixed chunks (such as always splitting after the third element, or checking for a specific character in a fixed-format string)
- Performance reasons for a specific collection type

In contrast, *dynamic indexing* is whenever you index into a collection with a variable that is not known until runtime. This is the most appropriate choice for lists, and dictionaries. You'll see this when iterating over collections or searching for a specific element with a `index()` function..

These are basic collections, but there are more ways to express intent. Here are some special collection types that are even more expressive in communicating to the future:

`frozenset`

a set that is immutable.

`OrderedDict`

a dictionary that preserves order of elements based on insertion time

`defaultdict`

A dictionary that provides a default value if the key is missing. For example, I could rewrite my earlier example as follows:

```
from collections import defaultdict
def create_author_count(cookbooks: List[Cookbook]):
    counter = defaultdict(lambda: 0)
    for cookbook in cookbooks:
        counter[cookbook.author] += 1
    return counter
```

```
This introduces a new behavior for end users - if they query the
dictionary for a value that doesn't exist, they will receive a 0. This
might be beneficial in some use cases, but if it not, you just return
`dict(counter)` instead.
```

`Counter`

a special type of dictionary used for counting how many times an element appears. This greatly simplifies our above code to the following:

```
from collections import Counter
def create_author_count(cookbooks: List[Cookbook]):
    return Counter(book.author for book in cookbooks)
```

> **NOTE**
>
> Built-in dictionaries are also ordered from CPython 3.6 and Python 3.7 onwards.

Take a minute to reflect on that last example. Notice how using a Counter gives us much more concise code without sacrificing readability. If your readers are familiar with Counter, the meaning of this function (and how the implementation works) is immediately apparent. This is a great example of communicating intent to the future through better selection of collection types. I'll continue to explore collections further in Chapter 5.

There are plenty more types to explore, such as array, bytes, ranges and more. Whenever you come across a new collection type, built-in or otherwise, ask yourself how it differs from other collections and what it conveys to future readers.

Iteration

Iteration is another example where the abstraction you choose dictates the intent you convey.

How many times have you seen code like this?

```
text = "This is some generic text"
index = 0
while index < len(text):
    print(text[index])
    index += 1
```

This simple code prints each character on a separate line. This is perfectly fine

for a first pass at Python for this problem, but the solution quickly evolves into the more Pythonic:

```
for character in text:
    print(character)
```

or even more simply:

```
print("\n".join(text))
```

Take a moment and reflect on why these last two options are preferable. In the `join()` case, it is because I am using a named abstraction of a loop (which again, communicates intent clearly). But even the for loop is clearer than the while loop. It's because the for loop is the more appropriate choice for my use case. Just like collection types, the looping construct you select explicitly communicates different concepts. Here's a list of looping constructs and what they convey:

For-loops

> For loops are used for iterating over each element in a collection or range and performing an action/side effect

While-loops

> While loops are used for iterating until a certain condition occurs

Comprehensions

> Comprehensions are used for transforming one collection into another (normally does not have side effects, especially if the comprehension is lazy)

Recursion

> Recursion is used when the sub-structure of a collection is identical to the structure of a collection (for example, each child of a tree is also a tree).

You want each line of your codebase to deliver value. Furthermore, you want each line to clearly communicate what that value is to future developers. This drives a need to minimize any amount of boilerplate, scaffolding, and

superfluous code. In the example above, I am iterating over each element and performing a side-effect (printing an element), which makes the for loop an ideal looping construct. I am not wasting code. In contrast, the while loop requires us to explicitly track looping until a certain condition occurs. In other words, I need to track a specific condition, and mutate a variable every iteration. This distracts from the value the loop provides, and provides unwanted cognitive burden.

Law of Least Surprise

Distractions from intent are bad, but there's a class of communication that is even worse: when code actively surprises your future collaborators. You want to adhere to the *Law of Least Surprise*. When someone reads through the codebase, they should almost never be surprised at behavior or implementation (and when they are surprised, there should be a great comment near the code to explain why it is that way). This is why communicating intent is paramount. Clear and clean code lowers chances for miscommunication.

> **NOTE**
>
> The *Law Of Least Surprise*, also known as the *Law of Least Astonishment* states that a program should always respond to the user in the way that astonishes them the least[4]. Surprising behavior leads to confusion. Confusion leads to misplaced assumptions. Misplaced assumptions lead to bugs. And that is how you get unreliable software.

Bear in mind, you can write completely correct code and still surprise someone in the future. There was one nasty bug I was chasing early in my career that crashed due to corrupted memory. Putting the code under a debugger or putting too many print statements in affected timing such that the bug would not manifest (a true "heisenbug[5]"). There were literally thousands of lines of code that related to this bug.

So I had to do a manual bisect, splitting the code in half, see which half actually had the crash by removing the other half, and then do it all over again in that code half. After two weeks of tearing my hair out, I finally decided to inspect an innocuous sounding function called getEvent. It turns out that this function was actually *setting* an event with invalid data. Needless to say, I was very surprised.

The function was completely correct in what it was doing, but because I missed the intent of the code, I overlooked the bug for at least three days. Surprising your readers will cost their time.

A lot of this surprise ends up coming from complexity. There are two types of complexity: *necessary complexity* and *accidental complexity*. Necessary complexity is the complexity inherent in your domain. Deep learning models are necessarily complex - they are not something you browse through the inner workings of and understand in a few minutes. Optimizing Object-Relational Mapping is necessarily complex - there is a large variety of possible user inputs that have to be accounted for. You won't be able to remove necessary complexity, so your best bet is to make sure it doesn't sprawl across your codebase.

In contrast, accidental complexity is the complexity that produces superfluous, wasteful or confusing statements in code. It's what happens when a system evolves over time and developers are jamming features in without re-evaluating old code to see if their original assertions hold true. I once worked on a project where adding a single command line option (and associated means of programmatically setting it) touched no fewer than 10 files. Why would adding one simple value ever need to require changes all over the codebase?

You know you have accidental complexity if you've ever experienced the following:

- Things that sound simple (adding users, changing a UI control, etc.) are non-trivial to implement
- Difficulty onboarding new developers into understanding your codebase. New developers on a project are your best indicators of how maintainable your code is right now - no need to wait years.
- Estimates for adding functionality are always high, and you slip the schedule anyway.

Remove accidental complexity and isolate your necessary complexity wherever possible. Those will be the stumbling blocks for your future collaborators. These sources of complexity compound miscommunication, as they obscure and diffuse intent throughout the codebase.

DISCUSSION TOPIC

What accidental complexities do you have in your codebase? How challenging would it be to understand simple concepts if you were dropped into the codebase with no communication to other developers? What can you do to simplify those complexities (especially if they are in often-changing code)?

Throughout the rest of the book, I will look at different techniques in Python to make our systems more robust. This book is broken up into 4 parts

Part 1

I'll start with types in Python. Types are fundamental to the language, but are not often delved into. The types you choose matter, as these convey a very specific intent. I'll talk about type annotations and what specific annotations communicate to the developer. I'l also go over typecheckers and how those help catch bugs early.

Part 2

After talking about how to think about Python's types, I'll focus on how to create your own types. I'll talk about enumerations, dataclasses and classes in depth. I'll explore how making certain design choices in designing a type can increase or decrease the robustness of your code.

Part 3

Now that you've learned how to communicate your intentions, I'll focus on how to enable users to change your code effortlessly. You will learn how to take that strong foundation, and let others build with confidence. I'll cover extensibility, dependencies, and architectural patterns that allow you to modify your system with minimal impact.

Part 4

Lastly, I'll explore about how to build a safety net, so that you can gently catch your future collaborators when they do fall. Their confidence will increase, knowing that they have a strong, robust system that they can fearlessly adapt to their use case. I'll cover a variety of static analysis and testing tools that will help you catch rogue behavior.

Wrap-up

Robust code matters. Clean code matters. Your code needs to be maintainable for the entire lifetime of the codebase, and in order to do that, you need to put active foresight into what you are communicating and how. You need to clearly embody your knowledge as close to the code as possible. It will feel like a burden to continuously look forward, but with practice it becomes natural, and you start reaping the gains as you work in your own codebase.

Every abstraction, every line, and every choice in a codebase communicates something, whether intentional or not. I encourage you to think about each line of code you are writing and ask yourself "What will a future developer learn from this?". You owe it to future maintainers to be able to deliver value at the same speed that you can today. Otherwise, your codebase will get bloated, schedules will slip, and complexity will grow. It is your job as a developer to mitigate that risk.

Look for potential hotspots, such as incorrect abstractions (such as collections or iteration) or accidental complexity. These are prime areas where communication can break down over time. If these are areas that change often, these are a priority to address now.

In the next chapter, you're going to take what you learned from this chapter, and apply it to a fundamental Python concept: types. The types you choose express your intent to future developers, and picking the correct type is just as important as picking the correct abstraction.

---

1 1980 Turing Award Lecture "The Emperor's Old Clothes"

2 *https://www.merriam-webster.com/dictionary/robust*

3 Yak-Shaving describes the situation where you frequently have to solve unrelated problems before you can even begin to tackle the original problem to solve. You can learn about the origins of the term at *https://seths.blog/2005/03/dont_shave_that/*

4 Geoffry James, The Tao of Programming

5 A bug that displays different behavior when being observed. SIGSOFT '83: Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging

Chapter 2. Introduction to Python Types

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

Welcome to Part 1, where I will focus on *types* in Python. Types model behavior of your program. Beginner programmers understand that there are different types in Python, such as *float* or *string*. But what is a type? How does mastering types make your codebase stronger? Types are a fundamental underpinning of any programming language, but, unfortunately, most introductory texts gloss over just how types benefit your codebase (or if misused, those same types increase complexity).

Tell me if you've seen this before:

```
>>>type(3.14)
<class 'float'>

>>>type("This is another boring example")
<class 'str'>

>>> type(["Even", "more", "boring", "examples"])
<class 'list'>
```

This could be pulled from almost any beginner's guide to Python. You learn about ints, strings, floats, bools, and all sorts of things the language offers. And

about ints, strings, floats, bools, and all sorts of things the language offers. And then, boom, you move on, because let's face it, this Python is not flashy. You want to dive into the cool stuff, like functions and loops and dictionaries, and I don't blame you. But it's a shame that many tutorials never revisit types and give them their proper due. As users dig deeper, they may discover type annotations (which I cover in the next chapter) or start writing classes, but often miss out on the fundamental discussion about when to use types appropriately.

That's where I'll start. To write maintainable Python, you must be aware of the nature of types and be deliberate about using them. I'll start by talking about what a type actually is and why that matters. I'll then move on to how the Python language's decisions about its type system affects the robustness of your codebase.

What's In a Type?

I want you to pause and answer a question: Without mentioning numbers, strings, text, or booleans, how would you explain what a type is?

It's not a simple answer for everyone. It's even harder to explain what the benefits are, especially in a language like Python where you do not have to explicitly declare types of variables.

I consider a type to have a very simple definition: a communication method. Types convey information. They provide a representation that users and computers can reason about. I break the representation down into two different facets:

Mechanical Representation

: Types communicate behaviors and constraints to the Python language itself

Semantic Representation

: Types communicate behaviors and constraints to other developers

Let's go learn a little more about each representation:

Mechanical Representation

At its core, computers are all about binary code. Your processor doesn't speak Python, all it sees are the presence or absence of electrical current on circuits going through it. Same goes for what's in your computer memory.

Suppose your memory looked like the following

0011001010001001000101001001000100100010000001010100101
01010101000000111111100100101001111101001001010010001
0010100`**010100000100000101010100**``1010010010010001010100010100 10
010101010010010010010000111101010110101101001010111`

Looks like a bunch of gibberish. Let's zoom in on the part that I've bolded:

01010000 01000001 01010100

There is no way to tell exactly what this number means by itself. Depending on computer architecture it is plausible that this could represent the number 5259604 or 5521744. It could also be the string "PAT". Without any sort of context, you can't know for certain. This is why computers need types. Type information gives Python what it needs to know to make sense of all the ones and zeroes. Let's see it in action:

```
from ctypes import string_at
from sys import getsizeof
from binascii import hexlify

a = 0b01010000_01000001_01010100
print(a)
>>> 5259604

# prints out the memory of the variable
print(hexlify(string_at(id(a), getsizeof(a))))
>>> b'0100000000000000607c05499555000001000000000000000054415000'

text = "PAT"
print(hexlify(string_at(id(text), getsizeof(text))))
>>>b'0100000000000000a00f06499555000003000000000000000375c9f1f02acdbe4e
5379218b77f00000000000000000000000050415400
```

NOTE

I am running CPython 3.9.0 on a little-endian machine, so if you see different results, don't worry, there are subtle things that can change your answers. (This code is not guaranteed to run on other Python implementations such as Jython or PyPy).

These hex-strings display the actual memory of a Python object. You'll find pointers to the next and previous object in a linked list (for garbage collection purposes), a reference count, a type, and the actual data itself. You can see the bytes at the end of each returned value to see the number or string (look for the bytes 0x544150 or 0x504154). The important part of this is that there is a type encoded into that memory. When Python looks at a variable, it knows exactly what type everything is at runtime (such as when you use the `type()` function.)

It's easy to think that this is the only reason for types - the computer needs to know how to interpret various blobs of memory. It is important to be aware of how Python uses types, as it has some implications for writing robust code, but even more important is the second representation: semantic representation.

Semantic Representation

While the first definition of types is great for lower-level programming, it's the second definition that applies to every developer. Types, in addition to having a mechanical representation, also manifest a semantic representation. A semantic representation is a communication tool; the types you choose communicate information across time and space to a future developer.

Types tell a user what behaviors they can expect about that entity. These behaviors are the operations that you associate with that type (plus any pre-conditions or post-conditions). They are the boundaries, constraints, and freedoms that a user interacts with whenever they use that type. Types used correctly have low barriers to understanding; they become natural to use. Conversely, types used poorly are a hindrance.

Consider the lowly int. Take a minute to think about what behaviors an integer has in Python. Here's a quick (non-comprehensive) list I came up with:

- Constructible from integers, floats, or strings
- Mathematical operations such as addition, subtraction, division, multiplication, exponentiation and negation
- Relational comparison such as <, >, ==, and !=

- Bitwise operations (manipulating individual bits of a number) such as &, |, ^, ~, and shifting
- Convertible to a string using str or repr functions
- Able to be rounded through ceil, floor, round methods (even though these return the integer itself, these are supported methods).

An `int` has many behaviors. You can view the full list if you if you type `help(int)` into your REPL.

Now consider a datetime:

```
>>>import datetime
>>>datetime.datetime.now()
datetime.datetime(2020, 9, 8, 22, 19, 28, 838667)
```

A datetime is not that different from an int. Typically it's represented as a number of seconds or milliseconds from some epoch of time (such as January 1st, 1970). But think about the behaviors a datetime has:

- Constructible from a **string, or a set of integers representing day/month/year/etc**
- Mathematical Operations such as addition and subtraction of **Time Deltas**
- Relational comparison
- **No bitwise operations available**
- Convertible to a string using str or repr functions
- **Is not** able to be rounded through ceil, floor, round methods

Datetimes support addition and subtraction, but not of other datetimes. We only add time deltas (such as adding a day or subtracting a year). Multiplying and dividing really don't make sense for a datetime. Similarly, rounding dates is not a supported operation in the standard library. However, datetimes do offer comparison and string formatting operations with similar semantics to an integer. So even though datetime is at heart an integer, it contains a constrained subset of operations.

> **NOTE**
>
> *Semantics* refers to the meaning of an operation. While `str(int)` and `str(datetime.datetime.now())` will return different formatted strings, the meaning is the same: I am creating a string from a value.

Datetimes also support their own behaviors, to further distinguish them from integers. These include

- Changing values based on time zones
- Being able to control the format of strings
- Finding what weekday it is

Again, if you'd like a full list of behaviors, type `import datetime; help(datetime.datetime)` into your REPL.

Datetimes are more specific than an integer. They convey a more specific use case than just a plain old number. When you choose to use a more specific type, you are telling future contributors that there are operations that are possible and constraints to be aware of that aren't present in the less specific type.

Let's dive into hhow this ties into robust code. Say you inherit a codebase that handles the opening and closing of an completely automated kitchen. You need to add in functionality to extend a kitchen's hours on holidays.

```
def close_kitchen_if_past_cutoff_time(point_in_time):
    if point_in_time >= closing_time():
        close_kitchen()
        log_time_closed(point_in_time)
```

You know you need to be operating on `point_in_time`, but how do you get started? What type are you even dealing with? Is it a string, integer, datetime, or some custom class? What operations are you allowed to perform on `point_in_time`? You didn't write this code, and you have no history with it. The same problems exist if you want to call the code as well. You have no idea what is legal to pass into this function.

If you make a wrong assumption one way or the other, and that code makes it to production, you've made the code less robust. Maybe that code doesn't lie on a codepath that is executed often. Maybe some other bug is hiding this code from being run. Maybe there aren't a whole lot of tests around this piece of code, and it becomes a runtime error later on. No matter what, there is a bug lurking in the code, and you've decreased maintainability.

Responsible developers do their best to not have bugs hit production. They will search for tests, documentation (with a grain of salt, of course — documentation can go out of date quickly), or calling code. They will look at `closing_time()` and `log_time_closed()` to see what types they expect or provide, and plan accordingly. This is a correct path in this case, but I still consider it a suboptimal path. While an error won't reach production, they are still expending time in looking through the code, which prevents value from being delivered as quickly. With such a small example, you would be forgiven for thinking that this isn't that big a problem if it happens once. But beware of "death by a thousand cuts": any one slice isn't too detrimental on its own, but thousands piled up and strewn across a codebase will leave you limping along, trying to deliver code.

The root cause is that the semantic representation was not clear for the parameter. So as you write code, do what you can to express your intent through types. You can do it as a comment where needed, but I recommend using type annotations (supported in Python 3.5+) to explain parts of your code.

```
def close_kitchen_if_past_cutoff_time(point_in_time:
datetime.datetime):
    if point_in_time >= closing_time():
        close_kitchen()
        log_time_closed(point_in_time)
```

All I need to do is put in a `: <type>` after my variables in the function signature. Most of my code examples in this book will annotate the types to make it clear what type I'm expecting.

Now, as developers come across this code, they will know what's expected of `point_in_time`. They don't have to look through other methods, tests or documentation to know how to manipulate the variable. They have a crystal-clear clue on what to do, and they can get right to work performing the

modifications they need to do. You are conveying semantic representation to future developers, without ever directly talking to them.

Furthermore, as developers use a type more and more, they become familiar with it. They won't need to look up documentation or `help()` to use that type when they come across it. You begin to create a vocabulary of well-known types across your codebase. This lessens the burden of maintenance. When a developer is modifying existing code, they want to focus on the changes they need to make, without getting bogged down.

Semantic representation of a type is extremely important, and the rest of Part 1 of this book will be dedicated to covering how you can use types to your advantage. Before I move on though, I need to talk about some fundamental choices Python has made as a language, and how they impact codebase robustness.

> **DISCUSSION TOPIC**
>
> Think about types used in your codebase. Pick a few and ask yourself what their semantic representations are. Enumerate their constraints, use cases and behaviors. Could you be using these types in more places? Are there places where you are misusing types?

Typing Systems

As discussed earlier in the chapter, a type system aims to give a user some way to model the behaviors and constraints in the language. Programming languages set expectations about how their specific type system works, both during code construction and runtime.

Strong vs. Weak

Typing systems are classified on a spectrum from weak to strong.

Languages towards the stronger side of the spectrum tend to restrict the use of operations to the types that support them. In other words, if you break the semantic representation of the type, you are told (sometimes quite loudly) through a compiler error or a runtime error. Languages such as Haskell, TypeScript, Rust are all considered strongly typed. Proponents advocate strongly

typed languages because errors are more apparent when building or running code.

In contrast, languages towards the weaker side of the spectrum will not restrict the use of operations to the types that support them. Types are often coerced into a different type to make sense of an operation. Languages such as JavaScript, Perl, and older versions of C are weakly typed. Proponents advocate the speed in which it takes to quickly iterate on code, without fighting language along the way.

Python falls towards the stronger side of the spectrum. There are very few implicit conversions that happen between types. It is noticeable when you perform illegal operations:

```
>>>[] + {}
TypeError: can only concatenate list (not "dict") to list

>>> {} + []
TypeError: unsupported operand type(s) for +: 'dict' and list
```

Contrast that with a weakly typed language, such as JavaScript:

```
>>> [] + {}
"[object Object]"

>>> {} + []
0
```

In terms of robustness, a strongly typed language such as Python certainly helps us out. While errors still will show up at runtime instead of at development time, they still will show up in a very obvious `TypeError` exception. This reduces the time taken to debug issues significantly, again allowing you to deliver incremental value quicker.

ARE WEAKLY-TYPED LANGUAGES INHERENTLY NOT ROBUST?

Codebases in weakly-typed languages can absolutely be robust; by no means am I dumping on those languages. Consider the sheer amount of production-grade JavaScript that the world runs on. However, a weakly typed language

requires extra care to be robust. Developers come to rely very heavily on linters, tests, and other tools to improve maintainability. I'll talk more about this in Part 4 of this book, Building a Safety Net.

Dynamic vs. Static

There is another typing spectrum I need to discuss: static vs dynamic typing. This is fundamentally a difference in handling mechanical representation of types.

Languages that offer static typing embed their typing information in variables during build time. Developers may explicitly add type information to variables or some tool such as a compiler infers types for the developer. Variables do not change their type at runtime (hence the name static.) Proponents of static typing tout the ability to write safe code out of the gate and to benefit from a strong safety net.

Dynamic typing, on the other hand, embeds type information with the value or variable itself. Variables can change types at runtime quite easily, because there is no type information tied to that variable. Proponents of dynamic typing advocate the flexibility and speed that it takes to develop; there's nowhere near as much fighting with compilers.

Python is a dynamically typed language. As you saw during the discussion about mechanical representation, you saw that there was type information embedded inside the values of a variable. Python has no qualms about changing the type of a variable at runtime:

```
>>>a = 5
>>>a = "string"
>>>a
"string"

>>>a = tuple()
()
```

Unfortunately, the ability to change types at runtime is a hindrance to robust code in many cases. You cannot make strong assumptions about a variable throughout its lifetime. As assumptions are broken, it's easy to write unstable assumptions on top of them, leading to a ticking logic bomb in your code.

assumptions on top of them, leading to a ticking logic bomb in your code.

ARE DYNAMICALLY-TYPED LANGUAGES INHERENTLY NOT ROBUST?

Just like weakly typed languages, it is still absolutely possible to write robust code in a dynamically typed language. You just have to work a little harder for it. You will have to make more deliberate decisions to make your codebase more maintainable. On the flip side, being statically typed doesn't guarantee robustness either; one can do the bare minimum with types and see little benefit.

To make things worse, the type annotations I showed earlier have no effect on this behavior at runtime:

```
>>>a: int = 5
>>>a = "string"
>>>a
"string"
```

No errors, no warnings, no anything. But hope is not lost, and you have plenty of strategies to make code more robust. (Otherwise, this would be quite the short book). We will discuss one last thing as a contributor to robust code, and then start diving into the meat of improving our codebase

Duck Typing

I feel like it is some unwritten law that whenever someone mentions duck typing, someone must reply with:

```
_If it walks like a duck and it quacks like a duck, then it must be a
duck._
```

My problem with this saying is that I find it completely unhelpful for explaining what duck typing actually is. It's catchy, concise, and crucially, only comprehensible to those who already understand duck typing. When I was younger, I just nodded politely, afraid that I was missing something profound by this simple phrase. It wasn't until later on that I truly understood the power of duck typing

duck typing.

Duck typing is the ability to use objects and entities in a programming language as long as it adheres to some interface. It is a wonderful thing in Python, and most people use it without even knowing it. Let's look at a simple example to illustrate what I'm talking about.

```
def print_items(items):
    for item in items:
        print(item)

print_items([1,2,3])
print_items({4, 5, 6})
print_items({"A": 1, "B": 2, "C": 3})
```

In all three invocations of `print_items`, we loop through the collection and print each item. Think about how this works. `print_items` has absolutely no knowledge of what type it will receive. It just receives a type at run-time and operates upon it. It's not introspecting each argument and deciding to do different things based on the type. The truth is much simpler. Instead, all print_items is doing is checking that whatever is passed in can be iterated upon (by calling an `__iter__` method). If the attribute `__iter__` exists, it's called and the returned iterator is looped over.

We can verify this with a simple code example:

```
>>>for x in 5:
>>>    print(x)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Duck typing is what makes this possible. As long as a type supports the variables and methods expected by a function (based on what's actually used), you can use that type in that function freely.

Here's another example:

```
>>>def double_value(value):
>>>    return value + value

>>>double_value(5)
```

```
10

>>>double_value("abc")
"abcabc"
```

It doesn't matter that we're passing an integer in one place or a string in another; both support the + operator, so either will work just fine. Any object that supports the + operator can be passed in. We can even do it with a list:

```
>>>double_value([1, 2, 3])
[1, 2, 3, 1, 2, 3]
```

So how does this play into robustness? It turns out that duck typing is a double-edged sword. It can increase robustness because it increases composability (we'll learn more about composability in XREF HERE). Building up a library of solid abstractions able to handle a multitude of types lessens the need for complex special cases. However, if duck typing is overused, you start to break down assumptions that a developer can rely upon. When updating code, it's not simple enough to just make the changes; you must look at all calling code and make sure that the types passed into your function satisfy your new changes as well.

With all this in mind, it might be best to reword the idiom at the beginning of this chapter as such:

If it walks like a duck, and quacks like a duck, and you are looking for things that walk and quack like ducks, then you can treat it as if it were a duck

Doesn't quite roll off the tongue as well, does it?

DISCUSSION TOPIC

Do you use duck typing in your codebase? Are there places where you can pass in types that don't match what the code is looking for, but things still work? Do you think these increase or decrease robustness for your use cases?

Wrap-up

Types are a pillar of clean, maintainable code and serve as a communication tool

to other developers. If you take care with types, you communicate a great deal, creating less burden for future maintainers. The rest of Part 1 will show you how to use types to enhance a codebase's robustness.

Remember, Python is dynamically and strongly typed. The strongly typed nature will be a boon for us; Python will notify us about errors when we use incompatible types. But its dynamically typed nature is something we will have to overcome in order to write better types. These language choices shape how Python code is written, and we'll be referring back often to them throughout the book.

In the next chapter, we're going to talk about type annotations, which is how we can be explicit about the type we use. Type annotations serve a crucial role: our primary commmunication method of behvaiors to future developers. They help overcome the limitations of a dynamically-typed language and allow you to enforce intentions throughout a codebase.

Chapter 3. Type Annotations

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

Python is a dynamically-typed language; types can be changed at runtime.. This is an obstacle when trying to write robust code. Since types are embedded in the value itself, developers have a very tough time knowing what type they are working with. Sure, that name looks likes a string today, but what happens if someone makes it `bytes`? Assumptions about types are built on shaky grounds with dynamically typed languages. Hope is not lost, though. In Python 3.5, a brand-new feature was introduced: type annotations.

Type annotations brings your ability to write robust code to a whole new level. Guido van Rossum, creator of Python, says it best

> *I've learned a painful lesson that for small programs dynamic typing is great. For large programs you have to have a more disciplined approach and it helps if the language actually gives you that discipline, rather than telling you "Well, you can do whatever you want"*[1]

Type annotations are the more disciplined approach, the extra bit of care you need to wrangle larger codebases. In this chapter, you'll learn how to use type annotations, why they are so important, and how to utilize a tool called a typechekcer to enforce your intentions throughout your codebase.

Type Annotations

In Chapter 2, you got your first glance at a type annotation:

```
def close_kitchen_if_past_close(point_in_time: datetime.datetime): ❶
    if point_in_time >= closing_time():
        close_kitchen()
        log_time_closed(point_in_time)
```

❶ The type annotation here is `: datetime.datetime`

Type annotations are an additional syntax notifying the user of an expected type of your variables. These annotations serve as *type hints*; they provide hints to the reader, but they are not actually used by the Python language. In fact, you are completely free to ignore the hints:

```
# CustomDateTime offers all the same functionality with
# datetime.datetime. I'm using it here for it's better
# logging facilities
close_kitchen_if_past_close(CustomDateTime("now")) # no error
```

> **WARNING**
>
> It should be a rare case where you go against a type hint. The author very clearly intended a specific use case. If you go against that use case, and the code changes, you don't have any protections that you can work with the changed method.

Python will not throw any error at runtime in this scenario. As a matter of fact, it won't use the type annotations at all during runtime. There is no checking or cost for using these when Python executes. These type annotations still serve a crucial purpose: informing your readers of the expected type. Maintainers of code will know what types they are allowed to use when changing your implementation. Calling code will also benefit, as developers will know exactly what type to pass in. By implementing type annotations, you reduce friction.

Put yourself in your future maintainer's shoes. Wouldn't it be nice to come across code that is intuitive to use? You wouldn't have to dig through function after function to determine usage. You wouldn't assume a wrong type and then need to deal with fallout of exceptions and wrong behavior.

Consider a piece of code that takes in employee's availability and a restaurant's opening time, and then schedules available workers for that day. You want to use this piece of code, and you see the following:

```
def schedule_restaurant_open(open_time, workers_needed):
```

Let's ignore the implementation for a minute, because I want to focus on first impressions. What do you think can get passed into this? Stop, close your eyes, and ask yourself what are reasonable types that can be passed in before reading on. Is `open_time` a datetime, the number of seconds since epoch, or maybe a string containing an hour? Is `workers_needed` a list of names, a list of `Worker` objects, or something else? If you guess wrong, or aren't sure, you need to go look at either the implementation or calling code, which I've established takes time and is frustrating.

Let me provide an implementation and you can see how close you were.

```
import datetime
import random

def schedule_restaurant_open(open_time: datetime.datetime,
  workers_needed: int):
    workers = find_workers_available_for_time(open_time)
    # use random.sample to pick X available workers
    # where X is the number of workers needed.
    for worker in random.sample(workers, workers_needed):
        worker.schedule(open_time)
```

You probably guessed that `open_time` is a datetime, but did you consider that `workers_needed` could have been an int? As soon as you see the type annotations, you get a much better picture of what's happening. This reduces cognitive overhead and reduces friction for maintainers.

WARNING

This is certainly a step in the right direction, but don't stop here. If you see code like this, consider renaming the variable to `number_of_workers_needed` to reflect just what the integer means. In the next chapter, I'll also explore type aliases, which provides an alternate way of expressing yourself..

So far, all the examples I've shown have focused on parameters, but you're also allowed to annotate *return types*.

Consider the `schedule_restaurant_open` function. In the middle of that snippet, I called `find_workers_available_for_time`. This returns to a variable named `workers`. Suppose you want to change the code to pick workers who have gone the longest without working, rather than random sampling? Do you have any indication what type `workers` is?

If you were to just look at the function signature, you would see the following:

```
def find_workers_available_for_time(open_time: datetime.datetime):
```

Nothing in here helps us do your job quicker. You could guess and the tests would tell us, right? Maybe it's a list of names? Instead of letting the tests fail, maybe you should go look through the implementation.

```
def find_workers_available_for_time(open_time: datetime.datetime):
    workers = worker_database.get_all_workers()
    available_workers = [worker for worker in workers
                         if is_available(worker)]
    if available_workers:
        return available_workers

    # fall back to workers who listed they are available in
    # in an emergency
    emergency_workers = [worker for worker in get_emergency_workers()
                         if is_available(worker)]


    if emergency_workers:
        return emergency_workers

    # Schedule the owner to open, they will find someone else
    return [OWNER]
```

Oh no, there's nothing in here that tells you what type you should be expecting. There are three different return statements throughout this code, and you hope that they all return the same type (surely every if statement is tested through unit tests to make sure they are consistent, right? Right?) You need to dig deeper. You need to look at `worker_database`. You need to look at `is_available` and `get_emergency_workers`. You need to look at the

`OWNER` variable. Every one of these needs to be consistent, or else you'll need to handle special cases in your original code.

And what if these functions also don't tell you exactly what you need? What if you have to go deeper through multiple function calls? Every layer you have to go through is another layer of abstraction you need to keep in your brain. Every piece of information contributes to cognitive overload. The more cognitive overload you are burdened with, the more likely a mistake will happen.

All of this is avoided by annotating a return type. Return types are annotated by putting `-> <type>` at the end of the function declaration. If you came across this function signature:

```
def find_workers_available_for_time(open_time: datetime.datetime) ->
list[str]:
```

You now know that you should indeed treat workers as a list of strings. No digging through databases, function calls or modules needed.

Finally, you can annotate variables when needed.

```
workers: list[str] = find_workers_available_for_time(open_time)
numbers: list[int] = []
ratio: float = get_ratio(5,3)
```

Most of the time, I won't bother annotating variables, unless there is something specific I want to convey in my code (such as a type that is different from expected). I don't want to get too into the realm of putting type annotations on `literally` everything - the lack of verbosity is what drew many developers to Python in the first place. The types can clutter your code, especially when it is blindingly obvious what the type is.

```
number: int = 0
text: str = "useless"
values: list[float] = [1.2, 3.4, 6.0]
worker: Worker = Worker()
```

None of these type annotations provide useful value than what is already provided by Python itself. Readers of this code know that `"useless"` is a string. Remember, type annotations are used for type hinting; you are providing

notes for the future to improve communication. You don't need to state the obvious everywhere.

TYPE ANNOTATIONS BEFORE PYTHON 3.5

If you have the misfortune of not being able to use a later version of Python, hope is not lost. There is an alternative syntax for type annotations, even for Python 2.7.

To write the annotations, you need to do so in a comment:

```
ratio = get_ratio(5,3) # type: float
def get_workers(open): # type: (datetime.datetime) -> List[str]
```

This is easier to miss, as the types are not visually close to the variable itself. If you have the ability to upgrade to Python 3.5, consider doing so and using the newer method of type annotations.

Benefits

As with every decision you make, you need to weigh the costs and benefits. Thinking about types up front helps your deliberate design process, but are there other benefits type annotations provide? I'll show you how type annotations really pull their weight through tooling.

Autocomplete

I've mainly talked about communication to other developers, but your Python environment benefits from type annotations as well. Since Python is dynamically-typed, it is difficult to know what operations are available. With type annotations, many Python-aware code editors will autocomplete your variable's operations.

In Figure 3-1, you'll see a screenshot that illustrates VSCode detecting a datetime and offering to autocomplete my variables.

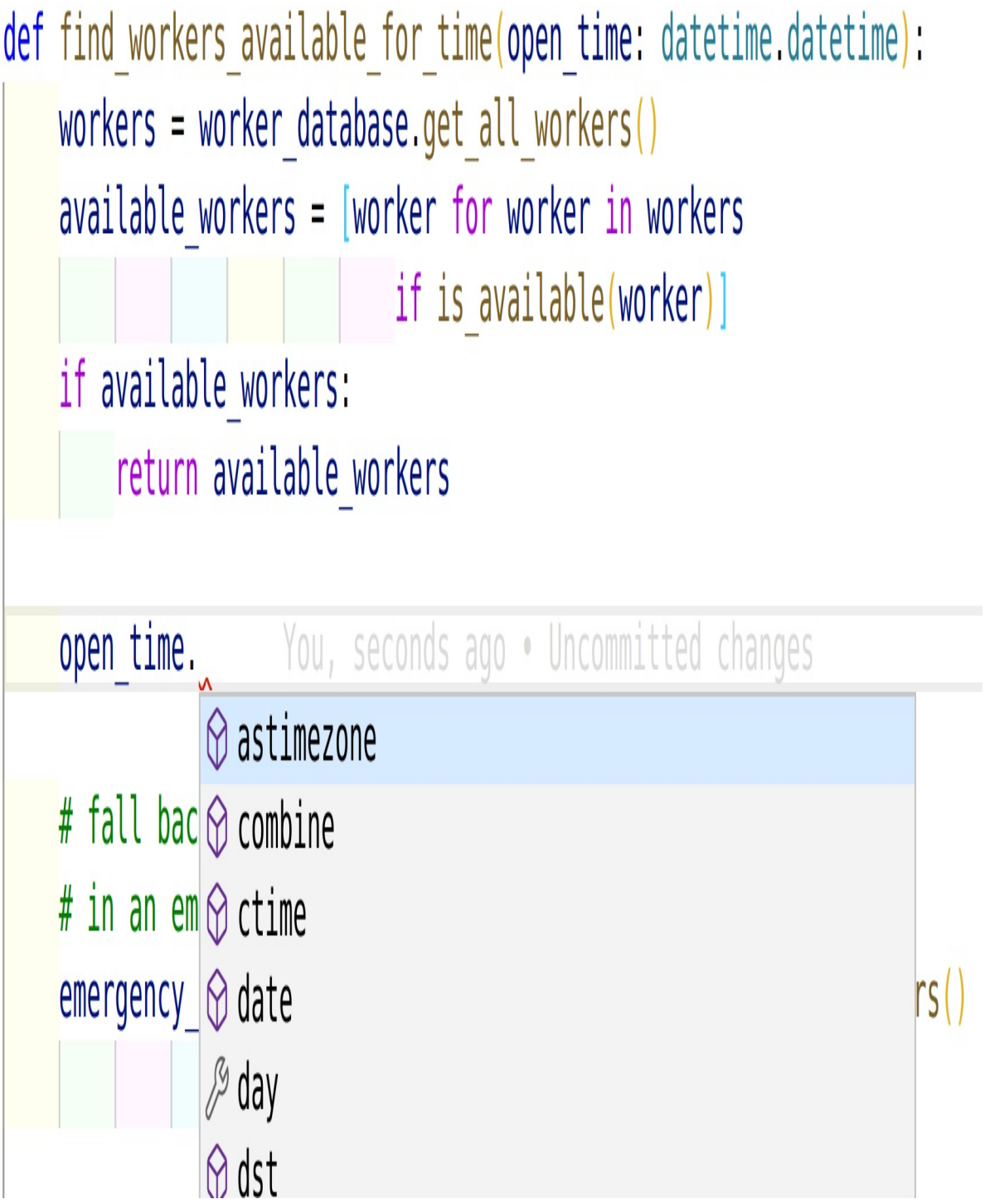

Figure 3-1. An IDE showing autocompletion

Typecheckers

Throughout this book, I've been talking about how types communicate intent, but have been leaving out one key detail: No programmer has to honor these type annotations if they don't want to. If your code contradicts a type annotation

type annotations if they don't want to. If your code contradicts a type annotation, it is probably an error, and you're still relying on humans to catch bugs. I want to do better. I want a computer to find these sorts of bugs for me.

I showed this snippet when talking about dynamic typing back in Chapter 2:

```
a: int = 5
a = "string"
a
>>> "string"
```

Herein lies the challenge: How do type annotations make your codebase robust, when you can't trust that develoeprs will follow their guidance? In order to be robust, you want your code to stand the test of time. To do that, you need some sort of tool that can check all your type annotations and flags if anything is amiss. That tool is called a typechecker.

Typecheckers are what allow the type annotations to transcend from communication method to a safety net. It is a form of static analysis. *Static analysis tools* are tools that run on your source code, and don't impact your runtime at all. You'll learn more about static analysis tools in XREF HERE, but for now, I will just explain typecheckers.

First, I need to install one. I'll use mypy, a very popular typechecker.

```
pip install mypy
```

Now I'll create a file named `invalid_type.py` with incorrect behavior:

```
a: int = 5
a = "string"
```

If I run mypy on the command line against that file, I will get an error:

```
mypy invalid_type.py

chapter3/invalid_type.py:2: error: Incompatible types in assignment
(expression has type "str", variable has type "int")
Found 1 error in 1 file (checked 1 source file)
```

And just like that, my type annotations become a first line of defense against errors. Anytime you make a mistake and go against the author's intent, a type checker will find out and alert you. In fact, in most development environments

checker will find out and alert you. In fact, in most development environments, it's possible to get this analysis in real-time, notifying you of errors as you type. (Without reading your mind, this is about as early as a tool can catch errors, which is pretty great.)

Exercise: Spot the Bug

Here are some more examples of mypy catching errors in my code. I want you to look for the error in each code snippet and time how long it takes you to find the bug or give up, and then check the output listed below the snippet to see if you got it right.

```
def read_file_and_reverse_it(filename: str) -> str:
    with open(filename) as f:
        # Convert bytes back into str
            return f.read().encode("utf-8")[::-1]


mypy chapter3/invalid_example1.py
chapter3/invalid_example1.py:3: error: Incompatible return value type
(got "bytes", expected "str")
Found 1 error in 1 file (checked 1 source file)
```

Whoops, I'm returning bytes, not a string. I made a call to encode instead of decode, and got my return type all mixed up.I can't even tell you how many times I made this mistake moving Python 2.7 code to Python 3. Thank goodness for typecheckers.

Here's another example:

```
from typing import List
# takes a list and adds the doubled values
# to the end
# [1,2,3] => [1,2,3,2,4,6]
def add_doubled_values(my_list: List[int]):
    my_list.update([x*2 for x in my_list])

add_doubled_values([1,2,3])


mypy chapter3/invalid_example2.py
chapter3/invalid_example2.py:6: error: "List[int]" has no attribute
"update"
Found 1 error in 1 file (checked 1 source file)
```

Another innocent mistake I made by calling `update` on a list instead of `extend`. These sort of mistakes can happen quite easily when moving between collection types (in this case from a `set`, which does offer an `update` method, to a `list`, which doesn't)

One more example to wrap it up:

```
# The restaurant is named differently in different
# in different parts of the world
def get_restaurant_name(city: str) -> str:
    if city in ITALY_CITIES:
            return  "Trattoria Viafore"
    if city in GERMANY_CITIES:
            return "Pat's Kantine"
    if city in US_CITIES:
            return "Pat's Place"
    return None


if get_restaurant_name('Boston'):
    print("Location Found")
```

```
chapter3/invalid_example3.py:14: error: Incompatible return value type
(got "None", expected "str")
Found 1 error in 1 file (checked 1 source file)
```

This one is subtle. I'm returning None when a string value is expected. If all the code is just checking conditionally for the restaurant name to make decisions, like I do above, tests will pass, and nothing will be amiss. This is true even for the negative case, because None is absolutely fine to check for in if statements (it is false-y). This is an example of Python's dynamic typing coming back to bite us.

However, a few months from now, some developer will start trying to use this return value as a string, and as soon as a new city needs to be added, the code starts trying to operate on None values, which causes exceptions to be raised. This is not very robust; there is a latent code bug just waiting to happen. But with typecheckers, you can stop worrying about this, and catch these mistakes early.

WARNING

With typechecker available, do you even need tests? You certainly do. Typecheckers catch a specific class of errors - those of incompatible types. There are plenty of other classes of errors that you still need to test for. Treat typecheckers as just one tool in your arsenal of finding bugs.

In all of these examples, typecheckers found a bug just waiting to happen. It doesn't matter if the bug would have been caught by tests, or by code review, or by customers; typecheckers catch it earlier, which saves time and money. Typecheckers start giving us the benefit of a statically-typed language, while still allowing the Python runtime to remain dynamically-typed. This truly is the best of both worlds.

At the beginning of the chapter, you'll find a quote from Guido van Rossum. Guido van Rossum is the creator of the Python programming language. While working at Dropbox he found that large codebases struggled without having a safety net. He became a huge proponent for driving type hinting into the language. If you want your code to communicate intent and catch errors, start adopting type annotations and typechecking today.

DISCUSSION TOPIC

Has your codebase had an error slip through that could have been caught by typecheckers? How much do those errors cost you? How many times has it been a code review or an integration test that caught the bug? How about bugs that made it to production?

When To Use

Now, before you go adding types to everything, I need to talk about the cost. Adding types is simple, but can be overdone. As users try to test and play around with code, they may start fighting the typechecker because they feel bogged down when writing all the type annotations. There is an adoption cost for users who are just getting started with type hinting. I also mentioned that I don't type annotate everything. I won't annotate all my variables, especially if the type is obvious. I also won't typically type annotate parameters for every small private method in a class.

When should you use typecheckers?

- Functions that you expect other modules or users to call (e.g. public API, library entry points, etc.)
- Code that you want to highlight where a type is complicated (e.g. a dictionary of strings mapped to lists of objects) or unintuitive.
- Areas where mypy complains that you need a type (typically when assigning to an empty collection - it's easier to go along with the tool than against it.)

Typecheckers will infer types for any value that it can, so even if you don't fill in all types, you still reap the benefits.

Wrap-up

There was consternation in the Python community when type hinting was introduced. Developers were afraid that Python was becoming a statically typed language like Java or C++. Developers felt that adding types everywhere would slow them down and destroy the benefits of the dynamically typed language they fell in love with.

However, type hints are just that: hints. They are completely optional. I don't recommend them for small scripts, or any piece of code that isn't going to live a very long time. But if your code needs to be maintainable for the long term, type hints are invaluable. They serve as a communication method, make your environment smarter, and detect errors when combined with typecheckers. They protect the original author's intent. When annotating types, you decrease the burden a reader has in understanding your code. You reduce the need to read the implementation of a function to know what its doing. Code is complicated, and you should be minimizing how much code a developer needs to read. By using well thought out types, you reduce surprise and increase reading comprehension.

The typechecker is also a confidence builder. Remember, in order for your code to be robust it has to be easy to change, rewrite and delete if needed. The typechecker can allow developers to do that with less trepidation. If something was relying on a type or field that got changed or deleted, the typechecker will flag the offending code as incompatible. Automated tooling makes you and your future collaborator's jobs simpler; less bugs will make it to production and

future collaborator's jobs simpler, less bugs will make it to production and features will get delivered quicker.

In the next chapter, you're going to go beyond basic type annotations and learn how to build a vocabulary of all new types. These types will help you constrain behavior in your codebase, limiting the ways things can go wrong. I've only scratched the surface of how useful type annotations can be.

---

1 A Language Creators' Conversation, PuPPy Annual Benefit 2019, *https://www.youtube.com/watch?v=csL8DLXGNlU*

Chapter 4. Constraining Types

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

Many developers learn the basic type annotations and call it a day. But I'm far from done. There is a wealth of advanced type annotations that are invaluable. These advanced type annotations allow you to constrain types, further restricting what they can represent. Your goal is to make illegal states unrepresentable. Developers should physically not be able to create types that are contradictory or otherwise invalid in your system. You can't have errors in your code if it's impossible to create the error in the first place. You can use type annotations to achieve this very goal, saving time and money. In this chapter I'll teach you six different techniques:

Optional

: Use to replace null references in your codebase

Union

: Use to present a selection of types

Literal

: Use to restrict developers to very specific values

Annotated

Annotated

Use to provide additional description of your types

NewType

Use to restrict a type to a specific context

Final

Use to prevent variables from being rebound to a new value.

Let's start with handling null references with `Optional` types.

Optional Type

Null references is often referred to as the "billion dollar mistake", coined by C.A.R. Hoare:

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.* [1]

While null references started in Algol, they would pervade countless other languages. C and C++ are often derided for null pointer dereference (which produces a segmentation fault or other program-halting crash). Java was well-known for having to catch `NullPointerException` throughout your code. It's not a stretch to say that these sorts of bugs have a price tag measured in the billions - think of the developer time, customer loss, and system failures due to accidental null pointers or references.

So, why does this matter in Python? C.A.R Hoare's quote is about object oriented compiled languages back in the 60s; Python must be better by now, right?. I regret to inform you that this billion-mistake is in Python as well. It appears to us under a different name: `None`. I will show you a way to avoid the

costly `None` mistake, but first, let's talk about why `None` is so bad.

> **NOTE**
>
> It is especially illuminating that C.A.R. Hoare admitted that null references were born out of convenience. It goes to show you how taking the quicker path can lead to all sorts of pain later in your development lifecycle. Think how your short-term decisions today will adversely affect your maintenance tomorrow.

Let's consider some code that runs an automated hot dog stand. I want my system to take a bun, put the hotdog in the bun, and then squirt ketchup and mustard through automtaed dispensers, as described in Figure 4-1. What could go wrong?

Figure 4-1. Workflow for the automated hot dog stand

```
def create_hot_dog():
    bun = dispense_bun()
    hotdog = dispense_hotdog()
    hotdog.place_in_bun(bun)
    ketchup = dispense_ketchup()
    mustard = dispense_mustard()
    hotdog.add_condiments(ketchup, mustard)
    return hotdog
```

Pretty straightforward, no? Unfortunately, there's no way to really tell. It's easy to think through the happy path, or the control flow of the program when everything goes right, but when talking about robust code, you need to consider error conditions. If this were an automated stand with no manual intervention, what errors can you think of?

Here's my non-comprehensive list of errors I can think of:

- Out of ingredients (buns, hotdog, or ketchup/mustard)
- Order cancelled mid-process.
- Condiments get jammed

- Power gets interrupted
- Customer doesn’t want ketchup or mustard, and tries to move the bun mid-process
- Rival vendor switches the ketchup out with catsup. Chaos ensues.

Now, your system is state-of-the-art and will detect all of these conditions, but it does so by returning None when any one ingredient fails. What does this mean for this code? You start seeing errors like the following:

```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'place_hotdog'

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'add_condiments'
```

It would be catastrophic if these errors bubbled up to your customers; you pride yourself on a clean UI and don’t want ugly tracebacks defiling your interface. To address this, you start to code *defensively*, or coding in such a way that you try to foresee every possible error case and account for it. Defensive programming is a good thing, but it leads to code like this:

```
def create_hot_dog():
    bun = dispense_bun()
    if bun is None:
        print_error_code("Bun unavailable. Check for bun")
        return None
    hotdog = dispense_hotdog()
    if hotdog is None:
        print_error_code("Hotdog unavailable. Check for hotdog")
        return None
    hotdog.place_in_bun(bun)


    ketchup = dispense_ketchup()
    mustard = dispense_mustard()
    if ketchup is None or mustard is None:
        print_error_code("Check for invalid catsup")
        return None
    hotdog.add_condiments(ketchup, mustard)
    return HotDog
```

This feels, well, tedious. Because any value can be `None` in Python, it seems like you need to engage in defensive programming and do an `is None` check before every dereference. This is overkill; most developers will trace through the call stack and ensure that no `None` values are returned to the caller. That leaves calls to external systems and maybe a scant few calls in your codebase that you always have to wrap with `None` checking. This is error-prone; you cannot expect every developer to ever touch your codebase to know instinctively where to check for `None`. Furthermore, the original assumptions you've made when writing (e.g. this function will never return `None`) can be broken in the future, and now your code has a bug. And herein lies your problem: counting on manual intervention to catch error cases is unreliable.

EXCEPTIONS

A valiant attempt at solving the billion-dollar problem is exceptions. Anytime something goes wrong in your system, throw an exception! When an exception is thrown, that function stops executing and the exception gets passed up the call chain, until either a) some code catches it in an appropriate except block, or b) nobody catches it and it terminates the program. This will not help your robustness problems. You still rely on manual intervention to catch errors (by someone writing an appropriate `except` block). If that manual intervention isn't applied, the program crashes and the user will have a bad time.

This should not come as a surprise; dereferencing `None` values throws an exception, so it's the exact same behavior. In order to be able to detect exceptions through static analysis, you typically need support in the language for checked exceptions: exceptions that are part of your type signature that tell your static analysis tools what exceptions to expect. Python does not support any sort of checked exception at the time of this writing and I am doubtful it ever will, due to the verbosity and viral nature of checked exceptions.

The reason this is so tricky (and so costly) is that `None` is treated as a special case. It exists outside the normal type hierarchy. Every variable can be assigned to `None`. In order to combat this, you need to find a way of representing `None`

inside your type hierarchy. You need `Optional` types.

Optional types offer you two choices: either you have a value or you don't. In other words, it is optional to set the variable to a value.

```
from typing import Optional
maybe_a_string: Optional[str] = "abcdef" # This has a value
maybe_a_string: Optional[str] = None     # This is the absence of a
value
```

This code indicates that the variable `maybe_a_string` may optionally contain a string. That code typechecks just fine, whether there is a string value bound to `maybe_a_string` or a `None` value.

At first glance, it's not apparent what this buys you. You still need to use `None` to represent the absence of a value. I have good news for you, though. There are three benefits I associate with `Optional` types.

First, you communicate your intent more clearly. If a developer sees an `Optional` type in a type signature, they view that as a big red flag that they should expect `None` as a possibility.

```
def dispense_bun() -> Optional[Bun]:
# ...
```

If you notice a function returning an `Optional` value, take heed and check for `None` Values.

Secondly, you are able to further distinguish the absence of value from an empty value. Consider the innocuous list. What happens if you make a function call and receive an empty list? Was it just that no results were provided back to you? Or was it that an error occurred and you need to take explicit action? If you are receiving a raw list, you don't know without trawling through source code. However, if you use an `Optional`, you are conveying one of three possibilities:

- A list with elements - valid data to be operated on
- A list with no elements - no error occurred, but no data was available (provided that no data is not an error condition)

- `None` - An error occurred that you need to handle

Finally, typecheckers can detect `Optional` types and make sure that you aren't letting None values slip through.

Consider:

```
def dispense_bun() -> Bun:
    return Bun('Wheat')
```

Let's add some error cases to this code:

```
def dispense_bun() -> Bun:
    if not are_buns_available():
        return None
    return Bun('Wheat')
```

When run with a typechecker, you get the following error:

```
code_examples/chapter4/invalid/dispense_bun.py:12: error: Incompatible
return value type (got "None", expected "Bun")
```

Excellent! The typechecker will not allow you to return a `None` value by default. By changing the return type from `Bun` to `Optional[Bun]`, the code will typecheck successfully. This will give developers hints that they should not return `None` without encoding information in the return type. You can catch a common mistake and make my code more robust. But what about the calling code?

It turns out that the calling code benefits from this as well. Consider:

```
def create_hot_dog():
    bun = dispense_bun()
    hotdog = dispense_hotdog()
    hotdog.place_in_bun(bun)
    ketchup = dispense_ketchup()
    mustard = dispense_mustard()
    hotdog.add_condiments(ketchup, mustard)
    return hotdog
```

If dispense_bun returns an Optional this code will not typecheck. It will complain with the following error:

```
code_examples/chapter4/invalid/hotdog_invalid.py:27: error: Item
"None" of "Optional[Bun]" has no attribute "place_hotdog"
```

> **WARNING**
>
> Depending on your typechecker, you may specifically need to enable an option to catch these sort of errors. Always look through your typechecker's documentation to learn what options are available. If there is an error you absolutely want to catch, you should test that your typechecker does indeed catch the error (I highly recommend testing out Optionals specifically. For the version of `mypy` I am running, I have to use `--strict-optional` as a command line flag to catch this error.

If you are interested in silencing the typechecker, you need to check for `None` explicitly and handle the `None` value, or assert that the value cannot be None. The following code typechecks successfully.

```
def create_hot_dog():
    bun = dispense_bun()
    if bun is None:
        print("Bun could not be dispensed")
        return
    hotdog = dispense_hotdog()
    hotdog.place_in_bun(bun)
    ketchup = dispense_ketchup()
    mustard = dispense_mustard()
    hotdog.add_condiments(ketchup, mustard)
    return hotdog
```

`None` values truly are a billion dollar mistake. If they slip through, programs can crash, users are frustrated, and money is lost. Use `Optional` types to tell other developers to beware of None, and benefit from the automated checking of your tools.

> **DISCUSSION TOPIC**
>
> How often do developers deal with `None` in your codebase? How confident are you that every possible `None` value is handled correctly? Look through bugs and failing tests to see how many times you've been bitten by incorrect `None` handling. Discuss how `Optional` types will help your codebase.

Union Types

`Optional` types are great, but what if you want to communicate error messages with specific information attached?

```
def dispense_hotdog() -> HotDog:
    if not are_ingredients_available():
        throw RuntimeError("Not all ingredients available")
    if order_interrupted():
        throw RuntimeError("Order interrupted")
    return create_hot_dog()
```

If I convert this code to use `Optional`, I lose information: the error messages are no longer returned. In these situations. Instead of an `Optional`, I can instead use a `Union`. `Unions` are versatile. If `Optional` lets you choose between a type and None, Union allows you to choose between any two types.

In the example above, I choose to return a HotDog or a string instead of throwing an exception.

```
from typing import Union
def dispense_hotdog() -> Union[HotDog, str]:
    if not are_ingredients_available():
        return "Not all ingredients available"
    if order_interrupted():
        return "Order interrupted"
    return create_hot_dog()
```

> **NOTE**
>
> `Optional` is just a specialized version of a `Union`. `Optional[int]` is the same exact thing as `Union[int, None]`.

`Unions` can be used for more than error handling as well. If you can return more than one type, you can indicate that with a Union as well. Suppose you want your hot dog stand to get into the lucrative pretzel business too. Instead of trying to deal with weird class inheritance (we'll cover more about inheritance in Part 2) that don't belong between hot dogs and pretzels, you simply can return a Union of the two (plus a string for catching errors).

```
from typing import Union
def dispense_snack(user_input: str) -> Union[HotDog, Pretzel, str]:
    if user_input == "Hotdog":
        return dispense_hotdog()
    elif user_input == "Pretzel":
        return dispense_pretzel()
    return "ERROR: Invalid User Input"
```

You will find `Unions` very useful in a variety of situations:

- Handling disparate types returned based on user input (as above)
- Handling error return types a la Optionals, but with more information, such as a string or error code.
- Handling different user input (such as if a user is able to supply a list or a string.)
- Returning different types, say for backwards compatibility (returning an old version of an object or a new version of an object depending on requested operation)
- And any other case where you may legitimately have more than one value represented.

Using a `Union` offers much of the same benefit as an `Optional`. First, you reap the same communication advantages. A developer encountering a `Union` knows that they must be able to handle more than one type in their calling code. Furthermore, a typechecker is just as aware of `Union` as it is `Optional`.

Suppose you had code that called the `dispense_snack` function but were only expecting a `Hotdog` or a `string` to be returned:

```
from typing import Optional, Union
def place_order() -> Optional[HotDog]:
    order = get_order()
    result = dispense_snack(order.name)
    if isinstance(result, str):
        print("An error occurred" + result)
        return None
    # Return our HotDog
    return result
```

As soon as dispense_snack starts returning `Pretzels`, this code fails to typecheck.

```
code_examples/chapter4/invalid/union_hotdog.py:22: error: Incompatible
return value type (got "Union[HotDog, Pretzel]", expected
"Optional[HotDog]")
```

The fact that your typechecker errors out in this case is fantastic. If any function you depend on changes to return a new type, their return signature must be updated to `Union` a new type, which forces you to update your code to handle the new type. This means that your code will be flagged when your depenendencies change in a way that contradicts your assumptions. With the decisions you make today, you can catch errors in the future. This is the mark of robust code; you are making it increasingly harder for developers to make mistakes, which reduces their error rates, which reduces the number of bugs users will experience.

There is one more fundamental benefit of using a `Union`, but to explain it, I need to teach you a smidge of *type theory*, which is a branch of mathematics around type systems.

Product and Sum Types

`Unions` are beneficial because they help constrain *representable state space*. Representable state space is the set of all possible combinations an object can take.

Take this dataclass:

```
from dataclasses import dataclass
from typing import Set
# If you aren't familiar with dataclasses, you'll learn more in
chapter 10
# but for now, treat this as four fields grouped together and what
types they are
@dataclass
class Snack:
    name: str
    condiments: Set[str]
    error_code: int
    disposed_of: bool
```

```
Snack("Hotdog", {"Mustard", "Ketchup"}, 5, False)
```

I have a name, the condiments that can go on top, an error code in case something goes wrong, and if something does go wrong, a boolean to track if I have disposed of the item correctly or not. How many different combinations of values can be put into this dictionary? Potentially infinite right? The `name` alone could be anything from valid values (“hotdog”, “pretzel”) to invalid values (“samosa”, “kimchi”, “poutine”) to absurd (“12345”, “”, “(ﾉ°□°)ﾉ ⁀ ┻━┻”). Condiments has a similar problem. As it stands, there is no way to compute the possible options.

For the sake of simplicity, I will artificially constrain this type:

- The name can be one of three values: hotdog, pretzel or veggie burger
- The condiments can be empty, mustard, ketchup or both
- There are 6 error codes (0-6) (0 indicates success)
- disposed_of is only true or false

Now how many different values can be represented in this combination of fields? The answer is 144, which is a grossly large number. I achieve this by the following:

3 possible types for name * 4 possible types for condiments * 6 error codes * 2 boolean values for if the entry has been disposed of = 3*4*6*2 = 144. If you were to accept that any of these values could be `None`, the total balloons to 280. While you should always think about `None` while coding (see earlier in this chapter about `Optional`), for this thought exercise, I’m going to ignore `None values`.

This sort of operation is known as a *product type*; the number of representable states is determined by the product of possible values. The problem is, not all of these states are valid. The variable `disposed_of` should only be set to `True` if an error code is set to non-zero. Developers will make this assumption, and trust that the illegal state never shows up. However, one innocent mistake can bring your whole system crashing to a halt. Consider the following code:

```
def serve(snack):
    # if something went wrong, return early
    if snack.disposed_of:
        return
    # ...
```

In this case, a developer is checking `disposed_of` without checking for the non-zero error code first. This is a logic bomb waiting to happen. This code will work completely fine as long as `disposed_of` is true *and* the error code is non-zero. If a valid snack ever sets the `disposed_of` flag to `True` erroneously, this code will start producing invalid results. This can be hard to find, as there's no reason for a developer who is creating the snack to check this code. As it stands, you have no way of catching this sort of error other than manually inspecting every use case, which is intractable for large code bases. By allowing an illegal state to be representable, you open the door to fragile code.

To rememdy this, I need to make this illegal state unrepresentable. To do that, I'll rework my example and use a `Union`:

```
from dataclasses import dataclass
from typing import Union, Set
@dataclass
class Error:
    error_code: int
    disposed_of: bool

@dataclass
class Snack:
    name: str
    condiments: Set[str]

snack: Union[Snack, Error] = Snack("Hotdog", {"Mustard", "Ketchup"})

snack = Error(5, True)
```

In this case, `snack` can be either a `Snack` (which is just a name and condiments) or an `Error` (which is just a number and a boolean). With the use of a `Union`, how many representable states are there now?

For `Snack`, there are 3 names and 4 possible list values, which is a total of 12 representable states. For `ErrorCode`, I can remove the 0 error code (since that was only for success) which gives me 5 values for the error code and 2 values

for the boolean for a total of 10 representable states. Since the `Union` is an either/or construct, I can either have 12 representable states in one case or 10 in the other, for a total of 22. This is an example of a *sum type*, since I'm adding the number of representable states together rather than multiplying.

22 total representable states. Compare that with the 144 states when all the fields were lumped in a single entity. I've reduced my representable state space by almost 85%. I've made it impossible to mix and match fields that are incompatible with each other. It becomes much harder to make a mistake, and there are far fewer combinations to test. Anytime you use a sum type, such as a Union, you are dramatically decreasing the number of possible representable states.

Literal Types

When calculating the number of representable states, I made some assumptions in the last section. I limited the number of values that were possible, but that's a bit of a cheat, isn't it? As I said before, there are almost an infinite number of values possible. Fortunately, there is a way to limit the values through Python: `Literals`. `Literal` types allow you to restrict the variable to a very specific set of values.

I'll change my earlier `Snack` class to employ `Literal` values:

```
from typing import Literal, Set
@dataclass
class Error:
    error_code: Literal[1,2,3,4,5]
    disposed_of: bool

@dataclass
class Snack:
    name: Literal["Pretzel", "Hotdog"]
    condiments: Set[Literal["Mustard", "Ketchup"]]
```

Now, if I try to instantiate these dataclasses with wrong values:

```
Error(0, False)
Snack("Not Valid", set())
Snack("Pretzel", {"Mustard", "Relish"})
```

I receive the following typechecker errors:

```
code_examples/chapter4/invalid/literals.py:14: error: Argument 1 to
"Error" has incompatible type "Literal[0]"; expected
"Union[Literal[1], Literal[2], Literal[3], Literal[4], Literal[5]]"

code_examples/chapter4/invalid/literals.py:15: error: Argument 1 to
"Snack" has incompatible type "Literal['Not Valid']"; expected
"Union[Literal['Pretzel'], Literal['Hotdog']]"

code_examples/chapter4/invalid/literals.py:16: error: Argument 2 to
<set> has incompatible type "Literal['Relish']"; expected
"Union[Literal['Mustard'], Literal['Ketchup']]"
```

`Literals` were introduced in Python 3.8, and they are an invaluable way of restricting possible values of a variable.

Annotated Types

What if I wanted to get even deeper and specify more complex constraints? It would be tedious to write hundreds of literals, and some constraints aren't able to be modelled by `Literal` types. There's no way with a literal to constrain a string to a certain size or to match a specific regex. This is where `Annotated` comes in. With `Annotated`, you can specify arbitrary metadata alongside your type annotation.

```
x: Annotated[int, ValueRange(3,5)]
y: Annotated[str, MatchesRegex('[0-9]{4}')]
```

Unfortunately, the above code will not run, as ValueRange and MatchesRegex are not built-in types; they are arbitrary expressions. You will need to write your own metadata as part of an `Annotated` variable. Secondly, there are no tools that will typecheck this for you. The best you can do until such a tool exists is write dummy annotations or use strings to describe your constraints. At this point, `Annotated` is best served as a communication method.

NewType

While waiting for tooling to support Annotated, there is another way to represent more complicated constraints: `NewType`. `NewType` allows you to, well, create a new type.

Suppose I want to separate my hot-dog stand code to handle two separate cases : a hotdog in its unprepared form, and a hotdog that is ready to be served (a prepared hotdog). However, some functions should only be operating on the hot dog in one case or the other.

For example:

- An unprepared hot dog needs to be put into a bun and can have condiments dispensed on top of it.
- A prepared hot dog needs to be put on a plate, given napkins, and served to a customer.

For example, our plating function might look something like this:

```
class HotDog:
    # ... snip hot dog class implementation ...

def place_on_plate(hotdog: HotDog):
    # note, this should only accept prepared hot dogs.
    # ...
```

However, nothing in the language prevents us from passing in an unprepared hot dog. If a developer makes a mistake and passes an unprepared hot dog to this function, customers will be quite surprised to just see their order with no bun or condiments come out of the machine.

Rather than relying on developers to catch these errors whenever they happen, you need a way for your typechecker to catch this. To do that, you can use `NewType`

```
from typing import NewType

class HotDog:
    ''' Used to represent an unprepared hot dog'''
    # ... snip hot dog class implementation ...

PreparedHotDog = NewType(HotDog)
```

```
def place_on_plate(hotdog: PreparedHotDog):
    # ...
```

A `NewType` takes an existing type and creates a brand new type that has all the same fields and methods as the existing type. In this case, I am creating a type `PreparedHotDog` that is distinct from `HotDog`; they are not interchangeable. What's beautiful about this is that this type restricts implicit type conversions. You cannot use a `HotDog` anywhere you are expecting a `PreparedHotDog` (you can use a `PreparedHotDog` in place of `HotDog`, though). In the above example, I am restricting `place_on_plate` to only take `PreparedHotDog` values as an argument. This prevents developers from invalidating assumptions. If a developer were to pass a `HotDog` to this method, the typechecker will yell at them:

```
code_examples/chapter4/invalid/newtype.py:10: error: Argument 1 to
"place_on_plate" has incompatible type "HotDog"; expected
"PreparedHotDog"
```

It is important to stress the one-way nature of this type conversion. As a developer, you can control when your old type becomes your new type.

For example, I'll modify a function from earlier in the chapter:

```
def create_hot_dog() -> PreparedHotDog:
    bun = dispense_bun()
    if bun is None:
        print("Bun could not be dispensed")
        return
    hotdog = dispense_hotdog()
    hotdog.place_in_bun(bun)
    ketchup = dispense_ketchup()
    mustard = dispense_mustard()
    hotdog.add_condiments(ketchup, mustard)
    return PreparedHotDog(hotdog)
```

Notice how I'm explicitly returning a `PreparedHotDog` instead of a normal hotdog. This acts as a "blessed" function; it is the only sanctioned way that I want developers to create a `PreparedHotDog`. Any user trying to use a method that takes a `PreparedHotDog` needs to create a hot dog using

`create_hot_dog` first.

It is important to notify users that the only way to create your new type is through a set of "blessed" functions. You don't want users creating your new type in any circumstance other than a predetermined method, as that defeats the purpose.

```
def make_snack():
    place_on_place(PreparedHotDog(HotDog))
```

Unfortunately, Python has no great way of telling users this, other than a comment.

```
from typing import NewType
# NOTE: Only create PreparedHotDog using create_hot_dog method.
PreparedHotDog = NewType(HotDog)
```

Still, `NewType` is applicable to many real-world scenarios. For example, these are all scenarios that I've run into that a `NewType` would solve.

- Separating a `str` from a `SanitizedString`, to catch bugs like SQL injection vulnerabilities. By making `SanitizedString` a `NewType`, I made sure that only properly sanitized strings were operated upon, eliminating the chance of SQL injection.
- Tracking a `User` object and `LoggedInUser` separately. By restricting `Users` with `NewType` from `LoggedInUser`, I wrote functions that only applicable to users that were logged in.
- Tracking an integer that should represent a valid User ID. By restricting the User ID to a `NewType`, I could make sure that some functions were only operating on IDs that were valid, without having to check if statements.

In XREF HERE, you'll see how you can use classes and invariants to do something very similar, with a much stronger guarantee of avoiding illegal states. However, NewType is still a useful pattern to be aware of, and is much more lightweight than a full-blown class.

TYPE ALIASES

`NewType` is not the same as a type alias. A type alias just provides another name for a type and is completely interchangeable with the old type.

For example

`IdOrName = Union[str, int]`

If a function expects `IDOrName`, it can take either an `IDOrName`, or a `Union[str,int]` and it will typecheck just fine, where a `NewType` will only work if a `IDOrName` is passed in.

I have found type aliases to be very helpful when I start nesting complex types,such as `Union[Dict[int, User], List[Dict[str, User]]`. It's much easier to give it a conceptual name, such as `IDOrNameLookup`, to simplify types.

Final Types

Finally (pun intended), you may want to restrict a type from changing it's value. That's where `Final` comes in. `Final`, introduced in Python 3.8, indicates to a typechecker that a variable cannot be bound to another value. For instance, I want to start franchising out my hot dog stand, but I don't want the name to be changed by accident.

```
VENDOR_NAME: Final = "Viafore's Auto-Dog"
```

If a developer accidentally changed the name later on, they would see an error.

```
def display_vendor_information():
    vendor_info = "Auto-Dog v1.0"
    # whoops, copy-paste error, this code should be vendor_info +=
VENDOR_NAME
    VENDOR_NAME += VENDOR_NAME
    print(vendor_info)

code_examples/chapter4/invalid/final.py:3: error: Cannot assign to
final name "VENDOR_NAME"
Found 1 error in 1 file (checked 1 source file)
```

In general, `Final` is best used when the variable's scope spans a large amount of code, such as a module. It is difficult for developers to keep track of all the uses of a variable in such large scopes; letting the typechecker catch immutability guarantees is a boon in these cases.

> **WARNING**
>
> `Final` will not error out when mutating an object through a function. It only prevents the variable from being rebound (set to a new value)

Wrap-up

You've learned about many different ways to constrain your types in this chapter. All of them serve a specific purpose, from handling `None` with `Optional` to restricting to specific values with `Literal` to preventing a varible from being rebound with `Final`. By using these techniques, you'll be able to encode assumptions and restrictions directly into your codebase, preventing future readers from needing to guess about your logic. Typecheckers will use these advanced type annotations to provide you with stricter guarantees about your code, which will give maintainers confidence when working in your codebase. With this confidence, they will make less mistakes, and your codebase will become more robust because of it.

In the next chapter, you'll move on from type annotating single values, and learn how to properly annotate collection types. Collection types pervade most of Python; you must take care to express your intentions for them as well. You need to be well versed in all the ways you can represent a collection, including in cases where you must create your own.

---

1 Null References: The Billion Dollar Mistake (QCon London 2009)

Chapter 5. Collection Types

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

You can't go very far in Python without encountering *collection types*. Collection types store a grouping of data, such as a list of users, or a lookup between restaurant or address. Whereas other types (ints, floats, bools) may focus on a single value, collections may store any arbitrary amount of data. In Python, you will encounter common collection types such as dictionaries, lists, and sets (oh, my!). Even a string is a type of collection; it contains a sequence of characters. However, collections can be difficult to reason about when reading about new code. Different collection types have different behaviors.

Back in Chapter 1, I went over some of the differences between the collections, where I talked about mutability, iterability, and indexing requirements. However, picking the right collection is just the first step. You must understand the implications your collection implies, and ensure that users can reason about it. You also need to recognize when the standard collection types aren't cutting it, and you need to roll your own. But the first step, is knowing how to communicate your collection choices to the future. For that, I'll turn to my old friend: type annotations.

Annotating Collections

I've covered type annotations for non-collection types, and now you need to know how to annotate collection types. Fortunately, these annotations don't differ too much from the annotations you've already learned.

To illustrate this, suppose I'm building a digital cookbook app. I want to organize all my cookbooks digitally so I can search them by cuisine, ingredient or author. One of the questions I might have about a cookbook collection is how many books from each author I have:

```
def count_authors(cookbooks: list) -> dict:
    counter = defaultdict(lambda: 0)
    for book in cookbooks:
        counter[book.author] += 1
    return counter
```

This function has been annotated; it takes in a list of cookbooks and will return a dictionary. Unfortunately, while this tells me what collections to expect, it doesn't tell me how to use the collections at all. There is nothing telling me what the elements inside the collection are. For instance, how do I know what type the cookbook is? If you were reviewing this code, how do you know that the use of `book.author` is legitimate? Even if you do the digging to make sure `book.author` is right, this code is not future-proof. If the underlying type changes, such as removing the `author` field, this code will break. I need a way to catch this with my typechecker.

I'll do this by encoding more information with my types by using bracket syntax to indicate information about the types *inside* the collection.

```
AuthorToCountMapping = dict[str, int]
def count_authors(cookbooks: list[Cookbook]) -> AuthorToCountMapping:
    counter = defaultdict(lambda: 0)
    for book in cookbooks:
        counter[book.author] += 1
    return counter
```

WARNING

In Python 3.8 and earlier, built-in collection types such as `list`, `dict` and `set` did not allow this bracket syntax, such as `list[Cookbook]` or `dict[str,int]`. Instead, you needed to use type annotations from the typing module:

```
from typing import Dict,List
AuthorToCountMapping = Dict[str, int]
def count_authors(cookbooks: List[Cookbook]) ->
AuthorToCountMapping:
    # ...
```

I can indicate the exact types expected in the collection. The cookbooks list contains `Cookbook` objects and the return value of the function is returning a dictionary mapping strings (keys) to ints (values). Note that I'm using a type alias to give more meaning to my return value. Mapping from a string to an int does not tell the user the context of the type. Instead, I create a type alias named `AuthorToCountMapping` to make it clear how this dictionary relates to the problem domain.

You need to think through what types are contained in collection in order to be effective in type hinting it. In order to do that, you need to think about homogeneous and heterogeneous collections.

Homogeneous vs. Heterogeneous Collections

Homogeneous collections are collections where every value in the collection has the same type. In contrast, *heterogeneous collections* have values that may have different types within them. From a usability standpoint, your lists, sets and dictionaries should nearly always be homogenous. Users need a way to reason about your collections, and they can't if they don't have the guarantee that every value is the same type. If you make a list, set or dictionary a heterogeneous collection, you are indicating to the user that they need to take care to handle special cases. Suppose I want to resurrect an example from Chapter 1 for adjusting recipes for my cookbook app.

```
# Take a meal recipe and change the number of servings
# by adjusting each ingredient
# A recipe's first element is the number of servings, and the
remainder
# of elements is (name, amount, unit), such as ("flour", 1.5, "cup")
def adjust_recipe(recipe, servings):
    new_recipe = [servings]
    old_servings = recipe[0]
```

```
    factor = servings / old_servings
    recipe.pop(0)
    while recipe:
            ingredient, amount, unit = recipe.pop(0)
            # please only use numbers that will be easily measurable
            new_recipe.append((ingredient, amount * factor, unit))
    return new_recipe
```

At the time, I mentioned how parts of this code were ugly, and one of the things made this tough to work with was the fact that the first element of the recipe list was a special case: an integer representing the servings. This contrasts from the rest of the list elements which are tuples representing actual ingredients, such as `("flour", 1.5, "cup")`. This highlights the troubles of a heterogeneous collection. For every use of your collection, the user needs to remember to handle the special case. This is predicated on the assumption that the developer even knew about the special case in the first place. There's no way in the type system to represent that a specific element needs to be handled differently. Therefore, a typechecker will not catch when a developer forgets. This leads to brittle code down the road.

When talking about homogeneity, it's important to talk about what a *single type* means. When I mention single type, I'm not necessarily referring to a concrete type in Python; rather, I'm referring to a set of behaviors that define that type. A single type indicates that a consumer must operate on every value of that type in the exact same way. For the cookbook list, the single type is a `Cookbook`. For the dictionary example, the key's single type is a string and the value's single type is an integer. For heterogeneous collections, this will not always be the case. What do you do if you must have different types in your collection and there is no relation between them?

Consider what my ugly code from Chapter 1 communicates:

```
# Take a meal recipe and change the number of servings
# by adjusting each ingredient
# A recipe's first element is the number of servings, and the
remainder
# of elements is (name, amount, unit), such as ("flour", 1.5, "cup")
def adjust_recipe(recipe, servings):
    # ...
```

There is a lot of information in the comment, but comments have no guarantee

of being correct. They also won't protect developers if they accidentally break assumptions. This code does not communicate intention adequately to future collaborators. Those future collaborators won't be able to reason about your code. The last thing you want to burden them with is having to go through the codebase, looking for invocations and implementations to work out how to use your collection. Ultimately, you need a way to reconcile the first element (an integer) with the remainder of the elements, which are tuples? To answer this, I'll use a `Union` (and some type aliases to make the code more readable).

```
Ingredient = tuple[str, int, str] # (name, quantity, units)
Recipe = list[Union[int, Ingredient]] # the list can be servings or
ingredients
def adjust_recipe(recipe: Recipe, servings):
    # ...
```

This takes a heterogeneous collection (items could be a integer or an ingredient) and allows developers to reason about the collection as if it were homogeneous. The developer needs to treat every single value as the same: it is either an integer or an Ingredient before operating on it. While needing more code to handle the typechecks, you can rest easier knowing that your typechecker will catch users not checking for special cases. Bear in mind, this is not perfect by any means; it'd be better if there was no special case in the first place and that servings was passed to the function another way. But for the cases where you absolutely must handle special cases, represent them as a type so that the typechecker benefits you.

This can go too far, though. The more special cases of types you handle, the more code a developer has to write every time they use that type, and the more unwieldy the codebase becomes.

At the far end of the spectrum lies the `Any` type. `Any` can be used to indicate that all types are valid in this context. This sounds appealing to get around special cases, but it also means that the consumers of your collection have no clue what to do with the values in the collection, defeating the purpose of type annotations in the first place.

WARNING

Developers working in a statically typed language don't need to put in as much care to ensure

> Developers working in a statically typed language don't need to put in as much care to ensure collections are homogeneous; the static type system does that for them already. The challenge in Python is due to Python's dynamically typed nature. It is much easier for a developer to create a heterogeneous collection without any warnings from the language itself.

Heterogeneous collection types still have a lot of uses; don't assume that you should use homogeneity for every collection type because it is easier to reason about. Tuples, for example, are often heterogeneous.

Suppose that I represent a `Cookbook` as a tuple.

```
Cookbook = tuple[str, int] # name, page count
```

I am describing specific fields for this tuple: name and page count. This is a prime example of an heterogeneous collection:

- Each field (name and page count) will always be in the same order
- All names are strings; all page counts are integers.
- Iterating over the tuple is rare, since I won't treat both types the same
- Name and page count are fundamentally different types, and should not be treated as equivalent.

When accessing a tuple, you will typically index to the specific field you want:

```
food_lab: Cookbook = ("The Food Lab", 958)
odd_bits: Cookbook ("Odd Bits", 248)

print(food_lab[0])
>>> "The Food Lab"

print(odd_bits[1])
>>> 248
```

However, in many codebases, tuples like these soon become burdensome. Developers tire of writing `cookbook[0]` whenever they want a name. A better thing to do would be to find some way to name these fields. A first choice might be a dictionary.

```
food_lab = {
```

```
    "name": "The Food Lab",
    "page_count": 958
}
```

Now, they can refer to fields as `food_lab['name']` and `food_lab['page_count']`. The problem is, dictionaries are typically meant to be a homogeneous mapping from key to a value. However, when dictionaries are used to represent data that is heterogeneous, you run into similar problems as above when writing a valid type annotation. I cannot write a meaningful type to represent this data. If I wanted to try to use a type system to represent this dictionary, I end up with the following:

```
def print_cookbook(cookbook: dict[str, Union[str,int]])
    # ...
```

This approach has the following problems:

- Large dictionaries may have many different types of values. Writing a `Union` is quite cumbersome.
- It is tedious for a user to handle every case for every dictionary access (since I indicate that the dictionary is homogeneous, I convey to developers that they need to treat every value as the same type, meaning typechecks for every value access. I know that the `name` is always a `string` and the `page_count` is always an `int`, but a consumer of this type would not know that.
- Developers do not have any indication what keys are available in the dictionary. They must search all the code from dictionary creation time to the current access to see what fields have been added.
- As the dictionary grows, developers have a tendency to use `Any` as the type of the value. Using Any defeats the purpose of the typechecker in this case.

NOTE

`Any` can be used for valid type annotations; it merely indicates that you are making zero assumptions what the type is. For instance, if you wanted to copy a list, the type signature

would be `def copy(coll: list[Any]) -> list[Any]`. Of course, you could also do `def copy(coll: list) -> list` as well, and it means the same thing.

These problems all stem from heterogeneous data in homogeneous data collections. You either pass the burden onto the caller, or abandon type annotations completely. In some cases, you want the caller to explicitly check each type on each value access, but in other cases, this is overcomplicated and tedious. So, how can you explain your reasoning with heterogeneous types, especially in cases where keeping data in a dictionary is natural, such as API interactions or user-configurable data. For these cases, you should use a *TypedDict*.

TypedDict

`TypedDict`, introduced in Python 3.8, is for the scenarios where you absolutely must store heterogeneous data in a dictionary. These scenarios are typically ar e when you can't avoid heterogeneous data. JSON APIs, YAML, TOML, XML and CSVs all have easy-to-use Python modules that convert these data formats into a dictionary and are naturally hetereogeneous. Which means the data that gets returned has all the same problems as listed in the previous section. Your typechecker won't help out much and users won't know what keys and values are available.

TIP

If you have full control of the dictionary, meaning you create it in code you own and handle it in code you own, you should consider using a dataclass or a class instead.

For example, suppose I want to augment my digital cookbook app to provide nutritional information for the recipes listed. I decide to use the Spoonacular API[1] and write some code to get nutritional information:

```
nutrition_information = get_nutrition_from_spoonacular(recipe_name)
# print grams of fat in recipe
print(nutrition_information["fat"]["value"])
```

If you were reviewing the code, how would you know that this code is right? If you wanted to also print out the calories, how do you access the data? What guarantees do you have about the fields inside of this dictionary? To answer these questions, you have two options:

- Look up the API documentation (if any) and confirm that the right fields are being used. In this option, you hope that the documentation is actually complete and correct.
- Run the code and print out the returned dictionary. In this option, you hope that test responses are pretty identical to production responses.

The problem is that you are requiring every reader, reviewer and maintainer to do one of these two steps in order to understand the code. If they don't, you will not get good code review feedback and developers will run the risk of using the response incorrectly. This leads to incorrect assumptions and brittle code. `TypedDict` allows you to encode what you've learned about that API directly into your type system.

```
from typing import TypedDict
class Range(TypedDict):
    min: float
    max: float

class NutritionInformation(TypedDict):
    value: int
    unit: str
    confidenceRange95Percent: Range
    standardDeviation: float

class RecipeNutritionInformation(TypedDict):
    recipes_used: int
    calories: NutritionInformation
    fat: NutritionInformation
    protein: NutritionInformation
    carbs: NutritionInformation

nutrition_information:RecipeNutritionInformation =
get_nutrition_from_spoonacular(recipe_name)
```

Now it is incredibly apparent exactly what data types you can rely upon. If the API ever changes, a developer can update all the `TypedDict` classes and let

the typechecker catch any incongruities. Your typechecker now completely understands your dictionary, and readers of your code can reason about responses without having to do any external searching. Even better, these `TypedDict` collections can be as arbitrarily complex as you need them to be. You'll see that I nested `TypedDict` instances for reusability purposes, but you can also embed your own custom types, `Unions` and `Optionals` to reflect the possibilities that an API can return. And while I've mostly been talking about API, remember that these benefits apply to any heterogeneous dictionary, such as when reading JSON or YAML.

> **NOTE**
>
> `TypedDict` is only for the type-checker's benefit. There is no run-time validation at all; the run-time type is just a dictionary.

So far, I've been teaching you how to deal with built-in collection types: lists/sets/dictionaries for homogeneous collections and tuples/TypedDict for heterogenous collections. What if these types don't do *everything* that you want? What if you want to create new collections for your using? To do that, you'll need a new set of tools.

Creating New Collections

When writing a new collection, you should ask yourself: Are you trying to write a new collection that isn't representable by another collection, or are you trying to modify an existing collection to provide some new behavior? Depending on the answer, you may need to employ different techniques to achieve your goal.

If you write a collection type that isn't representable by another collection type, you are bound to come across *generics* at some point.

Generics

A generic type indicates that you don't care what type you are using. However, it helps restrict users from mixing types where inappropriate.

Consider the innocuous reverse list function:

Consider the innocuous reverse list function:

```
def reverse(coll: list) -> list:
    return coll[::-1]
```

To achieve this, I use a generic, which is done with a `TypeVar` in Python.

```
from typing import TypeVar
T = TypeVar('T')
def reverse(coll: list[T]) -> list[T]:
    return coll[::-1]
```

This says that for a type “T”, reverse takes in a list of elements of type “T”, and returns a list of elements of type “T”. I can’t mix types: a list of integers will never be able to become a list of strings if those lists aren’t using the same `TypeVar`.

I can use this sort of pattern to define entire classes. Suppose I want to integrate a cookbook recommender service into the cookbook collection app. I want to be able to recommend cookbooks or recipes based on your ratings. To do this, I want to store each of these information into a *graph*. A graph is a data structure that contains a series of entities known as nodes, and tracks relationships between those nodes, known as edges. However, I don’t want to write separate code for a cookbook graph and a recipe graph. So I define a graph class that can be used for generic types. In my example, I’ll use `T` for my node type and `W` for my edges.

```
from collections import defualtdict
from typing import Generic, TypeVar

T = TypeVar("T")
W = TypeVar("W")

# directed graph
class Graph(Generic[T, W]):
    def __init__(self):
        self.edges: dict[T, list[W]] = defaultdict(list)

    def add_relation(self, node: T, to: W):
        self.edges[node].append(to)

    def get_relations(self, node: T) -> list[W]:
        return self.edges[node]
```

With this code, I can define all sorts of graphs and still have them typecheck successfully.

```
cookbooks: Graph[Cookbook, Cookbook] = Graph()
recipes: Graph[Recipe, Recipe] = Graph()

cookbook_recipes: Graph[Cookbook, Recipe] = Graph()

recipes.add_relation(Recipe('Pasta Bolognese'),
                     Recipe('Pasta with Sausage and Basil'))

cookbook_recipes.add_relation(Cookbook('The Food Lab'),
                              Recipe('Pasta Bolognese'))
```

While this code does not typecheck:

```
cookbooks.add_relation(Recipe('Cheeseburger'), Recipe('Hamburger'))
```

```
code_examples/chapter5/invalid/graph.py:25: error: Argument 1 to
"add_relation" of "Graph" has incompatible type "Recipe"; expected
"Cookbook"
```

Using generics can help you write collections that use types consistently throughout their lifetime. This reduces the amount of duplication in your codebase, which minimizes the chances of bugs and reduces cognitive burden.

OTHER USES FOR GENERICS

While often used for collections, you can technically use generics for any type. For example, suppose you want to simplify your API error handling. You've already forced your code to return a `Union` of the response type and an error type like so:

```
def get_nutrition_info(recipe: str) -> Union[NutritionInfo,
APIErrorResponse]:
    # ...

def get_ingredients(recipe: str) -> Union[list[Ingredient],
APIErrorResponse]:
    #...

def get_restaurants_serving(recipe: str) ->
```

```
    Union[list[Restaurant], APIErrorResponse]:
        # ...
```

But this is unneccessarily duplicated code. You have to specficy a `Union[X, APIErrorResponse` each time, where only `X` changes. What if you wanted to change the error response class, or force users to handle different types of errors separately? Generics can help with deduplicating these types:

```
T = TypeVar("T")
APIResponse = Union[T, APIErrorResponse]

def get_nutrition_info(recipe: str) -> APIResponse[NutritionInfo]:
    # ...

def get_ingredients(recipe: str) -> APIResponse[list[Ingredient]]:
    #...

def get_restaurants_serving(recipe: str) ->
APIResponse[list[Restaurant]]:
    # ...
```

Now you have a single place to control all of your API error handling. If you were to change it, you can rely on your typechecker to catch all the places needing change.

Modifying Existing Types

Generics are nice for creating your own collection types, but what if you just want to tweak some behavior of an existing collection, such as a list or dictionary. Having to completely rewrite all the semantics of a collection would be tedious and error-prone. Thankfully, methods exist to make this a snap. Let's go back to our cookbook app. I've written code earlier that grabs nutrtition information, but now I want to store all that nutrition information in a dictionary. However, I hit a problem: the same inredient has very different names depending on where you're from. Take a dark leafy green, common in salads. While a U.S. chef might call it "arugula", a European might call it "rocket". This doesn't even begin to cover the names in languages other than English.d To combat this, I want to create a dictionary-like object that automatically handles these aliases:

```
nutrition = NutritionalInformation()
nutrition["arugula"] = get_nutrition_information("arugula")
print(nutrition["rocket"]) # arugula == rocket
```

So how can I write `NutritionalInformation` to act like a dict?

A lot of developer's first instinct is to sub-class dictionaries. No worries if you aren't awesome at subclassing, I'll be going much more in depth in a later chapter. For now, just treat sub-classing as a way of saying "I want my subclass to behave exactly like the parent class".

```
class NutritionalInformation(dict): ❶
    def __getitem__(self, key): ❷
        try:
            return super().__getitem__(key) ❸
        except KeyError:
            pass
        for alias in get_aliases(key):
            try: ❹
                return super().__getitem__(alias)
            except KeyError:
                pass
        raise KeyError(f"Could not find {key} or any of its aliases")
❺
```

❶ The `(dict)` syntax indicates that we are subclassing from dictionaries

❷ `__getitem__` is what gets called when you use brackets to check a key in a dictionary. (`nutrition["rocket"]`) calls `__getitem__(nutrition, "rocket")`

❸ If a key is found, use the parent dictionaries key check.

❹ For every alias, check if it is in the dictionary

❺ Throw a KeyError if no key is found, either with what's passed in or any of its aliases.

We are overriding the `__getitem__` function, and this works!

If I try to access `nutrition["rocket"]` in that snippet above, I get the same nutritional information as `nutrition["arugula"]`. Huzzah! So you deploy it in production and call it a day.

But (and there's always a but), as time goes on, a developer comes to you and

complains that sometimes, the dictionary doesn't work. You spend some time debugging, and it always works for you. You look for race conditions, threading, API tomfoolery or any other nondeterminism, and come up with absolutely zero potential bugs. Finally, you get some time where you can sit with the other developer and see what they are doing.

And sitting at their terminal is the following line:

```
# arugula == rocket
nutrition = NutritionalInformation()
nutrition["arugula"] = get_nutrition_information("arugula")
print(nutrition.get("rocket", "No Ingredient Found"))
```

The get function on a dictionary tries to get the key, and if not found, will return the second argument (in this case "No Ingredient Found". And whenever this line is executed, you see just that: "No Ingredient Found". And herein lies the problem: When subclassing from a dictionary and overriding methods, you have no guarantee that those methods are called from every other method in the dictionary. Built-in collection types are built with performance in mind; many of methods use inlined code to go fast. This means that overriding one method, such as `__getitem__`, will not be used in most dictionary methods. This certainly violates the Law of Least Surprise, which we talked about in Chapter 1.

> **NOTE**
>
> It is okay to subclass from the built-in collection if you are only adding methods, but because future modifications may make this same mistake, I still prefer to use one of the other methods of building custom collections.

So overriding dict is out. Instead I'l use types from the collections module. For this case, there is a handy type called `UserDict`. `UserDict` fits the exact use case that I need: I can subclass from `UserDict`, override key methods, and get the behavior I expect.

```
from collections import UserDict
class NutritionalInformation(UserDict):
    def __getitem__(self, key):
        try:
```

```
            return self.data[key]
        except KeyError:
            pass
        for alias in get_aliases(key):
            try:
                return self.data[alias]
            except KeyError:
                pass
        raise KeyError(f"Could not find {key} or any of its aliases")
```

This fits your use case exactly. You subclass from `UserDict` instead of `dict`, and then use `self.data` to access the underlying dictionary.

You go run your teammate's code again:

```
# arugula == rocket
print(nutrition.get("rocket", "No Ingredient Found"))
```

And you get the nutrition information for arugula.

`UserDict` isn't the only collection type that you can override in this case. There also is a `UserString` and a `UserList` in the collections model. Anytime you want to tweak a dictionary, string or list, these are the collections you want to use.

> **WARNING**
>
> Inheriting from these classes does incur a performance cost. Built-in collections make some assumptions in order to achieve performance. With `UserDict`, `UserString`, and `UserList`, methods can't be inlined, since you might override them. If you need to use these constructs in performance-critical code, make sure you benchmark and measure your code to find potential problems.

You'll notice that I talked about dictionaries, lists and strings above, but left out one big built-in: sets. There exists no `UserSet` in the `collections` module. I'll have to select a different abstraction from the `collections` module. More specifically, I need abstract base classes (or `collections.abc`.)

As Easy as ABC

Abstract Base Classes in the `collections.abc` module provide another grouping of classes that you can override to create your own collections. These Abstarct Base Classes (or ABCs) is a class that is intended to be subclassed, and require the subclass to implement very specific functions. For the `collections.abc`, these ABCs are all centered on custom collections. In order to create a custom collection, you must override specific functions, depending on the type you want to emulate. You can find a full list of required functions to implement at the `collections.abc's` module documentation. [2] Once you implement these required functions though, the ABC fills in other functions automatically.

> **NOTE**
>
> In contrast to the `User*` classes, there is no built in storage, such as self.data, inside these classes. You must provide your own storage

Let's look at a `collections.abc.Set`, since there is no `UserSet` elsewhere in collections. I want to create a custom set that automatically handles aliases of ingredients (such as rocket and arugula). In order to create this custom set, I need to implement three methods:

- `__contains__`: This is for membership checks: `"arugula" in ingredients`.
- `__iter__`: This is for iterating: `for ingredient in ingredients`
- `__len__`: This is for checking the length: `len(ingredients)`

Once these three methods are defined, methods like relational operations, equality operations and set operations (union,intersection,difference,disjoint) will just work. That's the beauty of `collections.abc`. Once you define a select few methods, the rest come for free. Here it is in action:

```
import collections
class AliasedIngredients(collections.abc.Set):
    def __init__(self, ingredients: set[str]):
        self.ingredients = ingredients
```

```
    def __contains__(self, value: str):
        return value in self.ingredients or any(alias in
self.ingredients for alias in get_aliases(value))

    def __iter__(self):
        return iter(self.ingredients)

    def __len__(self):
        return len(self.ingredients)

ingredients = AliasedIngredients({'arugula', 'eggplant', 'pepper'})
for ingredient in ingredients:
    print(ingredient)
>>> 'arugula'
'eggplant'
'pepper'

print(len(ingredients))
>>> 3

print('arugula' in ingredients)
>>> True

print('rocket' in ingredients)
>>> True

list(ingredients | AliasedIngredients({'garlic'}))
>>>['pepper', 'arugula', 'eggplant', 'garlic']
```

That's not the only cool thing about `collections.abc`, though. Using `collections.abc` in type annotations can help you write more generic code. Take this code from all the way back in Chapter 2:

```
def print_items(items):
    for item in items:
            print(item)

print_items([1,2,3])
print_items({4, 5, 6})
print_items({"A": 1, "B": 2, "C": 3})
```

I talked about how duck typing can be both a boon and a curse for robst code. It's great that I can write a single function that can take so many different types, but communicating intent through type annotations becomes challenging. Fortunately, I can use the collections.abc classes to provide type hints:

```
def print_items(items: collections.abc.Iterable):
    for item in items:
            print(item)
```

In this case, I am indicating that items are simply iterable through the `Iterable` ABC. As long as the parameter supports an `__iter__` method (and most collections do), this code will typecheck.

As of Python 3.9, there are 25 different ABCs for you to use. Check them all out in the Python documentation[3].

Wrap-up

You can't go far without running into collections in Python. Lists, dictionaries, and sets are commonplace, and it's imperative that you provide hints to the future about what collection types you're working with. Consider if your collections are homogeneous or heterogeneous, and what that tells future readers. For the cases where you do use heterogeneous collections, provide enough information for other developers to reason about them, such as a TypedDict. Once you learn the techniques to allow other developers to reason about your collections, your codebase becomes so much more understandable.

Always think through your options when creating new collections:

- If you are just extending a type, such as adding new methods, you can subclass directly from collections such as list or dictionary. However, beware the rough edges, as there is some surprising Python behavior if a user ever overrides a built-in method.
- If you are looking to change out a small part of a list, dictionary or string, use `collections.UserList`, `collections.UserDict`, or `collections.UserString`, respectively. Remember to reference `self.data` to access the storage of the respective type.
- If you need to write a more complicated class with the interface of another collection type, use `collections.abc`. You will need to

provide your own storage for the data inside the class and implement all required methods, but once you do, you can customize that collection to your heart's content.

DISCUSSION TOPIC

Look through your use of collections and generics in your codebase, and assess how much information is conveyed to future developers. How many custom collection types are in your codebase? What can a new developer tell about the collection types by just looking at type signatures and names? Are there collections you could be defining more generically? What about other types using generics?

Now, type annotations don't reach their full potential without the aid of a typechecker. In the next chapter, I'm going to focus on the typechecker itself.You'll learn how to effectivly configure a typechecker, generate reports, and evaluate different checkers. The more you know about a tool, the more effectively you can wield it. This is especially true for your typechecker.

---

1 *https://rapidapi.com/spoonacular/api/recipe-food-nutrition?endpoint=59b3d500e4b0b0cacf7c5aaa*

2 *https://docs.python.org/3/library/collections.abc.html#module-collections.abc*

3 *https://docs.python.org/3/library/collections.abc.html#module-collections.abc*

Chapter 6. Customizing Your Typechecker

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

Typecheckers are one of your best resources for building robust codebases. Jukka Lehtosalo, the lead developer of mypy, offers a beautifully concise definition of typecheckers: *In essence, [a typechecker] provides verified documentation*.[1] Type annotations provide documentation about your codebase, allowing other developers the ability to reason about your intentions. And typecheckers use those annotations to verify that the documentation matches the behavior.

As such, a typechecker is invaluable. The wise sage Confucius once said *The mechanic, who wishes to do his work well, must first sharpen his tools.* [2] This chapter is all about sharpening your typechecker usage. Great coding techniques can get you far, but its your surrounding tooling that takes you to the next level. Don't stop with just learning your editor, compiler, or operating system. Learn your typechecker too. I will show you some of the more useful options to get the most out of your tools.

Configuring Your Typechecker

I will focus on one of the most popular typecheckers out there: mypy. Mypy offers quite a few configuration options to control the typechecker's *strictness*, or amount of errors reported. The stricter you make your typechecker, the more type annotations you need to write, providing better documentation and creating fewer bugs. However, make the typechecker too strict, and you will find the minimum bar for developing code too high, incurring high costs to make changes. Mypy configuration options are what controls these strictness levels. I'll go through the different options available to you, and you can decide where that bar lies for you and your codebase.

First, you need to install mypy (if you haven't already). The easiest way is through pip on the commandline:

```
pip install mypy
```

Once you have mypy installed, you you can control configuration through one of three ways:

Command Line

When instantiating mypy from a terminal, you can pass various options to configure behavior. This is great for exploring new checks in your codebase

Inline Configuration

You can specify configuration values at the top of a file to indicate any options you may want to set. For example:

```
# mypy: disallow-any-generics`
```

at the top of your file will tell mypy to explicitly look for Any values in a generics and fail if it finds any.

Configuration File

You can set up a configuration file to use the same options every time mypy runs. This is extremely useful when needing to share the same options across a team. This file is typically stored in version control alongside the code.

Configuring Mypy

When running mypy, mypy looks in your current directory for a configuration file named `mypy.`ini`. This file will define which options you have set up for the project. Some options will be global, applied to every file, and other options will be per-module. A sample `mypy.ini` file might look as follows:

```
# Global options:

[mypy]
python_version = 3.9
warn_return_any = True

# Per-module options:

[mypy-mycode.foo.*]
disallow_untyped_defs = True

[mypy-mycode.bar]
warn_return_any = False

[mypy-somelibrary]
ignore_missing_imports = True
```

> **TIP**
>
> You can use the `--config-file` command line option to specify config files in different places. Also, mypy will look for configuration files in specific home directories if it can't find a local config file, in case you want the same settings across multiple projects. For more information, check out the mypy documentation.

I'll cover some of the important options that you need to be aware of; they will help you control the behavior you want out of your typechecker. As a note, I won't cover too much more about the configuration file. Most options that I'll talk about work in both a configuration file and on the command line, and for the sake of simplicity, I'll show you how to run the commands on mypy invocations.

Catching Dynamic Behavior

As mentioned before, Python's dynamically typed nature will make maintenance hard on long-living code bases. When dynamically typed, every variable is essentially an `Any` type. `Any` indicates that literally any type would work for the

annotated code. There is not much value to be had in most of these cases; the typechecker won't flag any problems, and you aren't communicating anything special to future developers.

Mypy comes with a set of flags that you can turn on to flag instances of the `Any` type.

For instance, you can turn on the `--disallow-any-expr` option to flag any expression that has an `Any` type. The following code will fail with that option turned on:

```
from typing import Any
x: Any = 1
y = x + 1

test.py:4: error: Expression has type "Any"
Found 1 error in 1 file (checked 1 source file)
```

Another option I like for disallowing `Any` in type declarations such as in collections is `--disallow-any-generics`. This catches the use of `Any` for anything using a generic, such as collection types. The following code fails to typecheck with this option turned on.

```
x: list = [1,2,3,4]
```

You would need to use `list[int]` explicitly to get this code to work.

Check out all the ways to disable the use of `Any` in the mypy dynamic typing documentation.

Be careful with disabling `Any` too broadly, though. There is a valid use case of `Any` that you don't want to flag erroneously. `Any` should be reserved for when you absolutely don't care what type something is, and that it is up to the caller to verify the type. A prime example is a heterogeneous key-value store (perhaps a general-purpose cache).

Requiring Types

An expression is *untyped* if there is no type annotation. In these cases, mypy treats the result of that expression as an `Any` type if it can't otherwise infer the

type. However, the above checks for disallowing `Any` will not catch where a function is left untyped. There is a separate set of flags for checking for untyped functions.

This code will not error out in a typechecker unless the `--disallow-untyped-defs` option is set.

```
def plus_four(x):
    return x + 4
```

With that option set, you receive the following error:

```
test.py:4: error: Function is missing a type annotation
```

If this is too severe for you, you might want to check out `--disallow-incomplete-defs`, which only flags functions if they are partly annotated, or `--disallow-untyped-calls`, which only flags calls from annotated functions to unannotated functions. You'll find all the different options concerning untyped functions in the mypy documentation.

Handling None/Optional

In Chapter 4, you learned how easy it was to make the "billion-dollar mistake" when using `None` values. If you turn on no other options, make sure that you have `--strict-optional` turned on in your typechecker to catch these costly errors. You absolutely want to be checking that your use of `None` is not hiding any latent bugs.

When using `--strict-optional`, you must explicitly perform `is None` checks, otherwise your code will fail typechecking.

If `--strict-optional` is set (the default is different depending on the mypy version, so be sure to double check), this code should fail:

```
from typing import Optional
x: Optional[int] = None
print(x+5)
```

```
test.py:3: error: Unsupported operand types for + ("None" and "int")
test.py:3: note: Left operand is of type "Optional[int]"
```

It's worth noting that mypy also treats `None` values as `Optionals` implicitly. I recommend turning this off, so that you are being more explicit in your code. For example:

```
def foo(x: int = None) -> None:
    print(x)
```

The parameter `x` is implicitly converted to an `Optional[int]`, since `None` is a valid value for it. If you were to do any integer operations upon `x`, the typechecker would flag it. However, it's better to be more explicit and express that a value can be `None` (to disambiguate for future readers).

You can set `--no-implicit-optional` in order to get an error, forcing you to specify `Optional`. If you were to typecheck the above code with this option set, you would see:

```
test.py:2: error: Incompatible default for argument "x"
          (default has type "None", argument has type "int")
```

Mypy Reporting

If a typechecker fails in the forest and nobody is around to see it, does it print an error message? How do you know that mypy is actually checking your files, and that it will actually catch errors? Thankfully, mypy comes with some built-in reporting techniques.

First, you can get a HTMl report about how many lines of code mypy was able to check by passing in `--html-report` to mypy. This produces a HTML file that will provide a table that looks similar to Figure 6-1:

mypy.sharedparse	0.00% imprecise	114 LOC
mypy.sitepkgs	3.13% imprecise	32 LOC
mypy.solve	3.90% imprecise	77 LOC
mypy.split_namespace	38.24% imprecise	34 LOC
mypy.state	0.00% imprecise	18 LOC

Figure 6-1. HTML report from running mypy on the mypy source code

TIP

If you want a plaintext file, you can use `--linecount-report` instead.

Mypy also allows you to track explicit `Any` expressions to understand how you are doing on a line-by-line basis. When using the `--any-exprs-report` command line option, mypy will create a text file enumerating per-module statistics for how many times you use `Any`. This is very useful for seeing how explicit your type annotations are across a codebase. Here are the first few lines from running the `--any-exprs-report` option on the mypy codebase itself.

```
                    Name   Anys    Exprs   Coverage
-------------------------------------------------
          mypy.__main__      0       29    100.00%
               mypy.api      0       57    100.00%
        mypy.applytype      0      169    100.00%
           mypy.argmap      0      394    100.00%
           mypy.binder      0      817    100.00%
       mypy.bogus_type      0       10    100.00%
            mypy.build     97     6257     98.45%
          mypy.checker     10    12914     99.92%
        mypy.checkexpr     18    10646     99.83%
      mypy.checkmember      6     2274     99.74%
   mypy.checkstrformat     53     2271     97.67%
    mypy.config_parser     16      737     97.83%
```

If you'd like more machine-readable formats, you can use the `--junit-xml` option to create an XML file in the JUnit format. Most Continuous Integration systems can parse this format, making it ideal for automated report generation as part of your build system. To learn about all the different reporting options, check out the mypy report-generation documentation.

Speeding up Mypy

One of the common complaints about mypy is the speed in which it takes to typecheck large codebases. By default, mypy *incrementally* checks files. That is, it uses a cache (typically a .mypy_cache folder, but this is also configurable) to only check what has changed since last typecheck. This does speed up typechecking, but as your codebase gets larger, your typechecker will take longer to run, no matter what. This is detrimental for fast feedback during

development cycles. The longer a tool takes to provide useful feedback to developers, the less often developers will run the tool, thus defeating the purpose. It is in everyone's interest for typecheckers to run as fast as possible, so that developers are getting type errors at near real-time.

In order to speed up mypy even more, you may want to consider a *remote cache*. A remote cache provides a way of caching your mypy typechecks somewhere accessible to your entire team. This way, you can cache results based on specific commit IDs in your version control, and share typechecker information. Building this system is outside the scope of this book, but the remote cache documentation in mypy will provide a solid start.

You also want to consider mypy in daemon mode. Daemon mode is when mypy runs as a standalone process, and keeps previous mypy state in memory rather than on a file system (or across a network link). You can start a mypy daemon by running `dmypy run -- mypy-flags <mypy-files>`. Once the daemon is running, you can run the exact same command to check the files again.

For instance, I ran mypy on the mypy source code itself. My initial run took 23 seconds. Subsequent typechecks on my system took between 16 and 18 seconds. This is *technically* faster, but I would not consider it fast. When I use the mypy daemon, though, my subsequent runs ended up being under half a second. With times like that, I can run my typechecker much more often to get feedback faster. Check out more about dmypy in the mypy daemon mode documentation.

Alternative Typecheckers

Mypy is very configurable, and it's wealth of options will let you decide on the exact behavior you are looking for, but it won't meet all of your needs all the time. It isn't the only typechecker out there. I'd like to introduce two other type-checkers to you: pyre (written by Facebook) and pyright (written by Microsoft).

Pyre

You can install Pyre with through pip:

```
pip install pyre-check
```

Pyre runs very similarly to mypy's daemon mode. A separate process will run, from which you can ask for typechecking results. To typecheck your code, you need to set up pyre (by running `pyre init`) in your project directory, and then run `pyre` to start the daemon. From here, the information you receive is pretty similar to mypy. However, there are two features that set Pyre apart from other typecheckers: Codebase querying and the Python Static Analyzer framework (Pysa).

Codebase Querying

Once the pyre daemon is running, there are a lot of cool queries you can make to inspect your codebase. I'll use the mypy codebase as an example codebase for all of the following queries.

For instance, I can learn about the attributes of any class in my codebase:

```
pyre query "attributes(mypy.errors.CompileError)" ❶

{
   "response": {
       "attributes": [
           {
               "name": "__init__", ❷
               "annotation":
"BoundMethod[typing.Callable(mypy.errors.CompileError.__init__)
[[Named(self, mypy.errors.CompileError), Named(messa
ges, typing.List[str]), Named(use_stdout, bool, default),
Named(module_with_blocker, typing.Optional[str], default)], None],
mypy.errors.CompileE
rror]",
               "kind": "regular",
               "final": false
           },
           {
               "name": "messages", ❸
               "annotation": "typing.List[str]",
               "kind": "regular",
               "final": false
           },
           {
               "name": "module_with_blocker", ❹
               "annotation": "typing.Optional[str]",
               "kind": "regular",
               "final": false
           },
```

```
            {
                "name": "use_stdout", ❺
                "annotation": "bool",
                "kind": "regular",
                "final": false
            }
        ]
    }
}
```

❶ Pyre query for attributes

❷ Describing the constructor

❸ A list of strings for messages

❹ An Optional String describing a module with blocker

❺ A flag indicating printing to a screen

Look at all this information I can find out about the attributes in a class. I can see their type annotations to understand how the tool sees these attributes. This is incredibly handy in exploring classes as well.

Another cool query is the callees of any function.

```
pyre query "callees(mypy.errors.remove_path_prefix)"

{
   "response": {
       "callees": [
           {
               "kind": "function", ❶
               "target": "len"
           },
           {
               "kind": "method", ❷
               "is_optional_class_attribute": false,
               "direct_target": "str.__getitem__",
               "class_name": "str",
               "dispatch": "dynamic"
           },
           {
               "kind": "method", ❸
               "is_optional_class_attribute": false,
               "direct_target": "str.startswith",
               "class_name": "str",
               "dispatch": "dynamic"
```

```
            },
            {
                "kind": "method", ❹
                "is_optional_class_attribute": false,
                "direct_target": "slice.__init__",
                "class_name": "slice",
                "dispatch": "static"
            }
        ]
    }
}
```

❶ Calls the length function

❷ Calls the string.*getitem* function (such as `str[0]`)

❸ Calls the `startswith` function on a string

❹ Initializes a list slice (such as `str[3:8]`)

The typechecker needs to store all this information to do its job. It's a huge bonus that you can query the information as well. I could write a whole extra book on what you can do with this information, but for now, check out the pyre query documentation[3]. You will learn about different queries you can execute, such as observing class hierarchies, call graphs, and more. These queries allow you to learn more about your codebase or to build new tools to better understand your codebase (and catch other types of errors that a typechecker can't - such as temporal dependencies, which I'll cover in Part 3).

Python Static Analyzer (Pysa)

Pysa (pronounced like the Leaning Tower of Pisa), is a static code analyzer built into Pyre. Pysa specializes in a type of security static analysis known as *taint analysis*. Taint analysis is the tracking of potentially tainted data, such as user supplied input. The tainted data is tracked for the entire lifecycle of the data; pyre makes sure that any tainted data cannot propagate to a system in an insecure fashion.

Let me walk through the process to catch a simple security flaw (modified from the Pyre documentation.) Consider the case where a user creates a new recipe in a filesystem.

```
import os

def create_recipe():
   recipe = input("Enter in recipe")
   create_recipe_on_disk(recipe)

def create_recipe_on_disk(recipe):
   command = "touch ~/food_data/{}.json".format(recipe)
   return os.system(command)
```

This looks pretty innocuous. A user can enter in `carrots` to create the file `~/food_data/carrots.json`.. But what if a user enters in `carrots; ls ~;`? If this were entered, it would print out the entire home directory (the command becomes `touch ~/food_data/carrots; ls ~;.json`) Based on input, a malicious user could enter in arbitrary commands on your server (this is known as remote code execution, or RCE), which is a huge security risk.

Pysa provides tools to check this. I can specify that anything coming from `input()` is potentially tainted data (known as a taint source), and anything passed to `os.system` should not be tainted (known as a taint sink). With this information, I need to build a *taint model*, which is a set of rules for detecting potential security holes. First, I must specify a `taint.config` file:

```
{
  sources: [
    {
      name: "UserControlled", ❶
      comment: "use to annotate user input"
    }
  ],

  sinks: [
    {
      name: "RemoteCodeExecution", ❷
      comment: "use to annotate execution of code"
    }
  ],

  features: [],

  rules: [
    {
      name: "Possible shell injection", ❸
```

```
      code: 5001,
      sources: [ "UserControlled" ],
      sinks: [ "RemoteCodeExecution" ],
      message_format: "Data from [{$sources}] source(s) may reach
[{$sinks}] sink(s)"
    }
  ]
}
```

❶ Specify an annotation for user controlled input

❷ Specify an annotation for Remote Code Execution flaws

❸ Specify a rule that makes any tainted data from UserControlled sources an error if it ends up in a RemoteCodeExecution sink.

From there, I must specify a taint model to annotate these sources as tainted.

```
# stubs/taint/general.pysa

# model for raw_input
def input(__prompt = ...) -> TaintSource[UserControlled]: ...

# model for os.system
def os.system(command: TaintSink[RemoteCodeExecution]): ...
```

These stubs tell pysa through type annotations about where your taint sources and sinks are in your system.

Finally, you need to tell pyre to detect tainted information by modifying the .pyre_configruation to add in your directory:

```
"source_directories": ["."],
"taint_models_path": ["stubs/taint"]
```

Now, when I run `pyre analyze` on that code, Pysa flags anerror.

```
[
    {
        "line": 9,
        "column": 26,
        "stop_line": 9,
        "stop_column": 32,
        "path": "insecure.py",
        "code": 5001,
```

```
        "name": "Possible shell injection",
        "description":
            "Possible shell injection [5001]: Data from
[UserControlled] source(s) may reach " +
            "[RemoteCodeExecution] sink(s)",
        "long_description":
            "Possible shell injection [5001]: Data from
[UserControlled] source(s) may reach " +
            "[RemoteCodeExecution] sink(s)",
        "concise_description":
            "Possible shell injection [5001]: Data from
[UserControlled] source(s) may reach " + "
            "[RemoteCodeExecution] sink(s)",
        "inference": null,
        "define": "insecure.create_recipe"
    }
]
```

In order to fix this, I either need to make this data flow impossible, or run tainted data through a *sanitizer* function. Sanitize functions take untrusted data, and inspect/modify them so that they can be trusted. Pysa allows you decorate functions with `@sanitize` to specify your sanitizers.[4]

This was admittedly a simple example, but Pysa allows you to annotate your codebase to catch more complicated problems (such as SQL injection and cookie mismanagement). To learn everything that Pysa can do (as well as built-in common security flaws), check out the complete documentation.

Pyright

Pyright is a typechecker designed by Microsoft. I have found it to be the most configurable of typecheckers I've come across. If you would like more control than your current typechecker, explore the Pyre configuration documentation for all that you can do. However, Pyright has an additional awesome feature : VS Code Integration.

VS Code (also built by Microsoft) is an immensely popular code editor for developers. Microsoft leveraged the ownership of both tools to create a VS Code extension called Pylance. You can install Pylance from your VS Code Extensions browser. Pylance is built upon Pyright, and uses type annotations to provide a better code editing experience. Before, I mentioned that autocomplete was a benefit of type annotations in IDEs, but Pylance takes it to the next level.

Pylance offers the following features:

- Automatic insertion of imports, based on your types
- Tooltips with full type annotations based on signatures
- Codebase browsing such as finding references or browsing a call graph
- Real-time diagnostic checking

It's this last feature that sells Pylance/Pyright for me. Pylance has a setting that allows you to constantly run diagnostics in your whole workspace. This means that every time you edit a file, pyright will run across your entire workspace (and it runs fast, too) to look for additional areas that you broke. You don't need to manually run any commands; it happens automatically. As someone who likes to refactor often, this tool is invaluable for finding breakages early. Remember, you want to find your errors as close to real-time as possible.

I've pulled up the mypy source codebase again and have Pylance enabled and in workplace diagnostics mode. I want to change one type on line 19 from a `Sequence` to a `Tuple` and see how the rest of the workspace responds in Figure 6-2

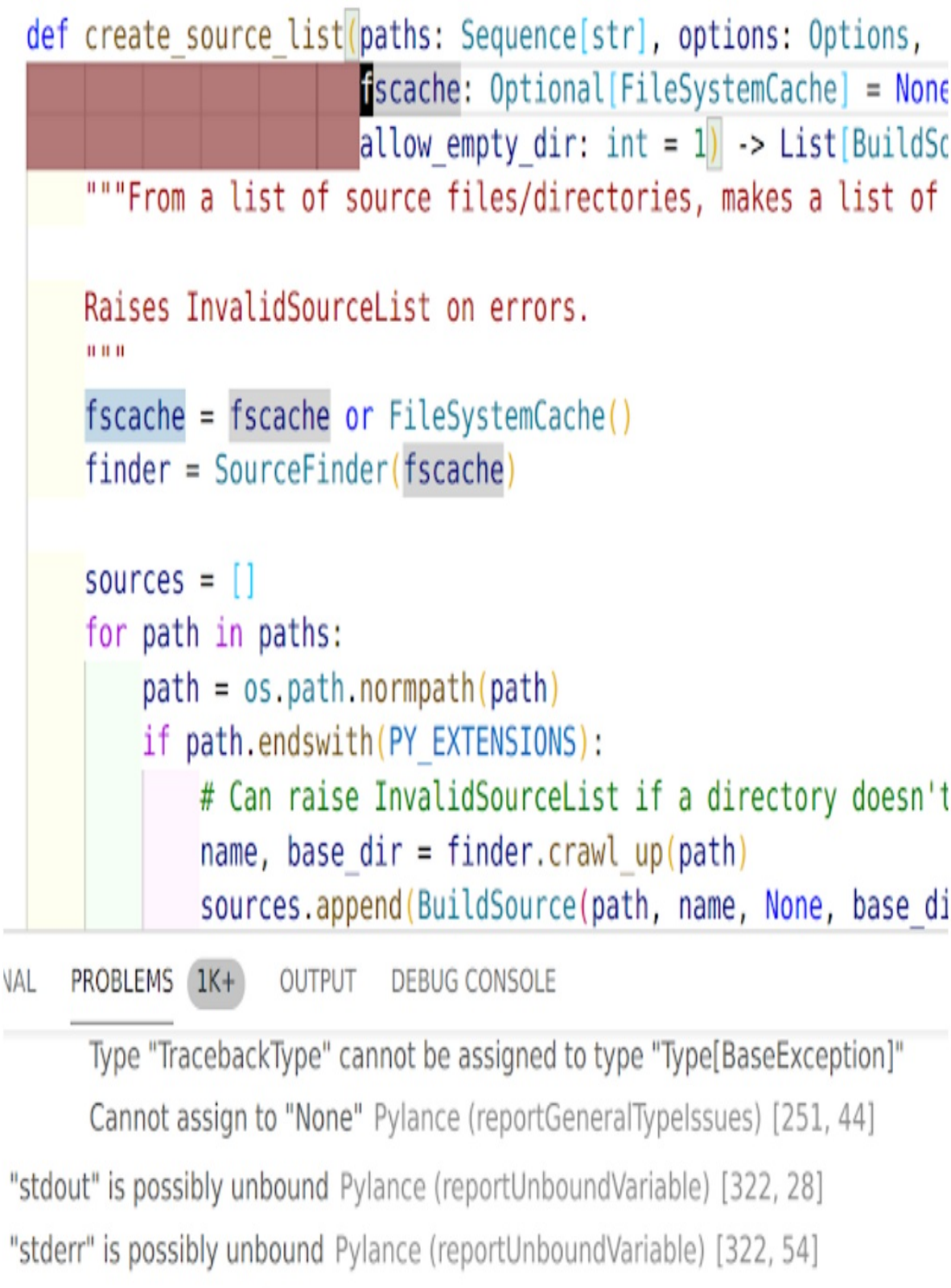

Figure 6-2. Problems in VS Code before editing

Notice at the bottom where my “Problems” are listed. The current view is

showing issues in another file that imports and uses the current function I’m editing. Once I change the paths argument from Sequence to a Tuple, see how the “Problems” change in Figure 6-3.

```
def create_source_list(paths: Tuple[str], options: Options,
                       fscache: Optional[FileSystemCache] = None,
                       allow_empty_dir: int = 1) -> List[BuildSou
    """From a list of source files/directories, makes a list of B

    Raises InvalidSourceList on errors.
    """
    fscache = fscache or FileSystemCache()
    finder = SourceFinder(fscache)

    sources = []
    for path in paths:
        path = os.path.normpath(path)
        if path.endswith(PY_EXTENSIONS):
            # Can raise InvalidSourceList if a directory doesn't
            name, base_dir = finder.crawl_up(path)
```

NAL PROBLEMS 1K+ OUTPUT DEBUG CONSOLE

Type "TracebackType" cannot be assigned to type "Type[BaseException]"

Cannot assign to "None" Pylance (reportGeneralTypeIssues) [251, 44]

"stdout" is possibly unbound Pylance (reportUnboundVariable) [322, 28]

"stderr" is possibly unbound Pylance (reportUnboundVariable) [322, 54]

^ Argument of type "Sequence[str]" cannot be assigned to parameter "paths" of typ

"Sequence[str]" is incompatible with "Tuple[str]" Pylance (reportGeneralTypeIssu

^ Argument of type "List[str]" cannot be assigned to parameter "paths" of type "Tup

"List[str]" is incompatible with "Tuple[str]" Pylance (reportGeneralTypeIssues) [3

Figure 6-3. Problems in VS Code after editing

Within half a second of saving my file, new errors have shown up in my "Problems" pane, telling me that I've just broken assumptions in calling code. I don't have to wait to run a typechecker manually, or wait for a CI process to yell at me; my errors show up right in my editor. If that doesn't lead me to finding errors earlier, I don't know what will.

Wrap-up

There's a wealth of options at your disposal, and you need to be comfortable with advanced configuration to get the most out of your tool. You can control severity options, reporting, or even use different typecheckers to reap benefits. As you evaluate tools and options, ask yourself how strict you want your typecheckers to be. As you increase the scope of errors that can be caught, you will increase the amount of time and effort needed to make your codebase compliant. However, the more informative you can make your code, the more robust it will be in its lifetime.

In the next chapter, I will talk about how to assess the trade-offs between benefits and costs. You'll learn how to identify important areas to typecheck and use strategies to mitigate your pain.

---

1 Our journey to type checking 4 million lines of Python, *https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python*

2 The Analects, Confucius, c.a. 500 BCE

3 *https://pyre-check.org/docs/querying-pyre*

4 You can learn more about sanitizers at *https://pyre-check.org/docs/pysa-basics#sanitizers*

Chapter 7. Adopting Typechecking Practically

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *pat@kudzera.com*.

Quite a few developers dream of the days where they can work in a completely *greenfield* project. A green-field project is one that is brand new, where you have a blank slate with your code's architecture, design, and modularity. However, most projects soon become *brown-field*, or legacy code. These projects have been around the block a bit; much of the architecture and design has been solidified. Making big sweeping changes will impact real users. However, the word brown-field is often seen as derogatory, especially when it feels like you are slogging through a big ball of mud.

However, not all brownfield projects are a punishment to work in. Michael Feathers, author of *Working Effectively With Legacy Code* has this to say:

> *In a well-maintained system, it might take a while to figure out how to make a change, but once you do, the change is usually easy and you feel much more comfortable with the system. In a legacy system, it can take a long time to figure out what to do, and the change is difficult also.*[1]

Brownfield projects can absolutely be in a well-maintainted state. It's true that htey are complex at first, but once you get past the initial hurdle of understanding, it becomes easy to change. Unmaintainable code is where the real

understanding, it becomes easy to change. Unmaintainable code is where the real problem lies. This is why robustness is paramount. Writing robust code eases the transition from green-field to brown-field through maintainability.

Most of the type annotations strategies that I've shown in the first part of this book are easier to adopt when a project is new. To adopt these practices in a mature project is more challenging. It is not impossible, but the cost may be higher. This is the heart of engineering: making smart decisions about trade-offs.

Trade-offs

Every decision you make involves a trade-off. Lots of developers focus on the classic time vs. space trade-off in algorithms. But there are plenty of other trade-offs, often involving intangible qualities. I've already covered the benefits of a typechecker quite extensively throughout this first part of the book: * A typechecker increases communication and reduce the chances of bugs. * A typechecker provide a safety net for making changes and increase the robustness of your codebase. * A typechecker allows you to deliver functionality faster,

But what are the costs? Adopting type annotations is not free, and they only get worse the larger your codebase is. These costs include:

- First you need buy-in. Depending on culture, it might take some time convincing an organization to adopt typechecking
- Once you have buy-in there is an initial cost of adoption. Developers don't start type annotating their code overnight, and it takes time before they grok it. They need to learn it and experiment before they are on board.
- It takes time and effort to adopt tooling. You need some centralized checking of some fashion, and developers need to familiarize themselves with running the tooling as part of their workflows.
- It will take time to write type annotations in your codebase.
- As type annotations are checked, developers will have to get used to the slowdown in fighting the typechecker. There is additional cognitive overload in thinking about types.

Developer time is expensive, and it is easy to focus on what else those developers could be doing. Adopting type annotations is not free. Worse, with a large enough codebase, these costs can easily dwarf the initial benefit you get from typechecking. The problem is fundamentally a chicken-and-egg conundrum. You won't see benefits for annotating types until you have written enough types in your codebase. However, it is tough to get buy-in for writing types when the benefit isn't there early on. You can model your value as such:

Value = (Total Benefits) - (Total Costs)

Your benefits and costs will follow a curve; they are not linear functions. I've outlined the basic shapes of the curves in Figure 7-1.

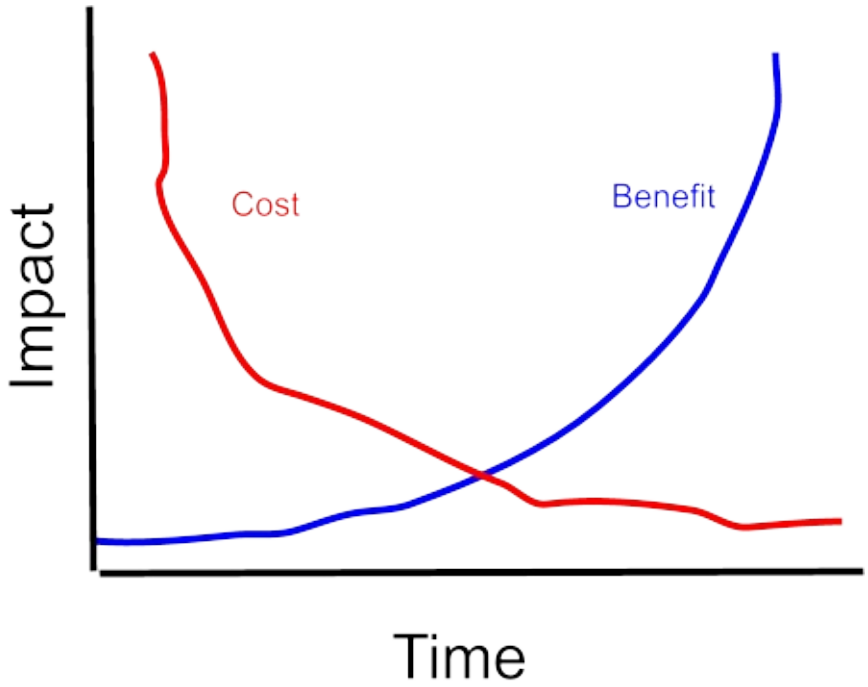

Figure 7-1. Cost and Benefit Curves Over Time

I've purposely left off the range, because the scale will change depending on the size of your codebase, but the shapes remain the same. Your costs will start out high, but get easier as adoption increases. Your benefits will start off low, but as you annotate your codebase, you will see more value. You won't see a return on investment until these two curves meet. To maximize value, you need to reach that intersection as early as possible.

Breaking Even Earlier

To maximize the benefits of type annotations, you need to either get value earlier, or decrease your costs earlier. The intersection of these two curves is a break-even point; this is where the amount of effort that you're expending is paid back by the value you are receiving. You want to reach this point as fast as sustainably possible to make your type annotations a positive impact. Here are some strategies to do that.

Finding Your Pain Points

One of the best ways to produce value is to reduce the pain you are currently experiencing. Ask yourself, where do you currently lose time in your process? Where do you lose money? Take a look at your test failures and customer bugs. These error cases incur real costs; you should be doing root cause analysis. If you find that a common root cause that can be fixed by type annotations, you have a solid case for type annotation adoption. Here are specific bug classes you need to keep an eye out for:

- Any error surrounding `None`
- Invalid attribute access, such as trying to access variables of functions on the wrong type
- Errors surrounding type conversions such as ints vs strings, bytes vs strings, or lists vs tuples.

Also, talk to the people who have to work in the codebase itself. Root out the areas that are a constant source of confusion. If developers have trouble with certain parts of the codebase today, it's very likely that future developers will struggle too. Don't forget to talk to those who are invested in your codebase, but maybe don't directly work in it, such as your tech support, product management and QA. They often have a unique perspective on painful areas of the codebase that might not be apparent when looking through the code. Try to put these costs into concrete terms, such as time or money. This will be invaluable in evaluating where type annotations will be of benefit.

Target Code Strategically

You may want to focus on trying to receive value earlier. Type annotations do

not appear overnight in a large codebase. Instead, you will need to identify specific and strategic areas of code to target for type annotations. The beauty of type annotaitons is that they are completely optional. By typechecking just these areas, you very quickly see benefits without a huge upfront investment. Here are some strategies that you might employ to selectively type annotate your code.

Type Annotate New Code Only

Consider leaving your current, un-annotated code the way it is and annotate code based on these two rules:

1. Annotate any new code that you write
2. Annotate any old code that you change

Throughout time, you'll build out your type annotations in all code except code that hasn't been changed in a long time. Code that hasn't been changing is relatively stable, and is probably not read too often. Type annotating it is not likely to gain you much benefit.

Type Annotate From The Bottom Up

Your codebase may depend on common areas of code. These are your core libraries and utilities that serve as a foundation upon which everything else is built upon. Type annotating these parts of your codebase makes your benefit less about depth and more about breadth. Because so many other pieces sit atop this foundation, they will all reap the benefits of typechecking. New code will quite often depend on these utilities as well, so your new code will have an extra layer of protection.

Type Annotate Your Moneymakers

In some codebases, there is a clear separation between the core business logic and all the rest of the code that supports your business logic. Your *business logic* is the area of your system that is most responsible for delivering value. It might be the core reservation system for a travel agency, an ordering system in a restaurant, or a recommendation system for media services. All of the rest the code (such as logging, messaging, database drivers and user interface) exists to support your business logic. By type annotating your business logic, you are protecting a core part of your codebase. This code is often long-lived, making it

an easy win for long-lasting value.

Type Annotate The Churners

Some parts of your codebase change. Some parts of your codebases change way more often than the others. Everytime a piece of code changes, you run the risk of an incorrect assumption introducing a bug. The whole point of robust code is to lessen the chance of introducing errors, so what better place to protect than the code that changes the most often? Look for your code that has many different commits in version control, or analyze which files have the most lines of code changed over a time period. Also take a look at which files have the most committers; this is a great indication that this is an area where you can shore up type annotations for communication purposes.

> **NOTE**
>
> **Discussion Topic**: Which of these strategies would benefit your codebase the most? Why does that strategy work best for you? What the cost would be to implement that strategy.

Lean On Your Tooling

There are things that computers do well, and there are things that humans do well. This section is about the former. When trying to adopt type annotations, there are some fantastic things that automated tooling can assist with. First, let's talk about the most common typechecker out there: mypy.

I've covered the configuration of mypy quite extensively in Chapter 6, but there's a few more options I'd like to delve into that will help you adopt typechecking. One of the biggest problems you will run into is the sheer number of errors that mypy will report the first time you run it upon a larger codebase. The biggest mistake you can make in this situation is to keep the hundreds (or thousands) of errors turned on and hope that developers whittle away at the errors over time.

These errors will not get fixed in any quick fashion. If these errors are always turned on, you will not see the benefits of a typechecker, because it will be near impossible to detect new errors. Any new issue will simply be lost in the noise of the multitude of other issues.

With mypy, you can tell the typechecker to ignore certain classes of errors or modules through configuration. Here's a sample mypy file, which globally warns if Any types are returned, and sets config options on a per module basis:

```
# Global options:

[mypy]
python_version = 3.9
warn_return_any = True

# Per-module options:

[mypy-mycode.foo.*]
disallow_untyped_defs = True

[mypy-mycode.bar]
warn_return_any = False

[mypy-somelibrary]
ignore_missing_imports = True
```

Using this format, you can pick and choose which errors your type checker tracks. You can mask all of your existing errors, while focusing on fixing new errors. Be as specific as possible in defining what errors get ignored; you don't want to mask new errors that show up in unrelated parts of the code.

To be even more specific, mypy will ignore any line commented with `# type: ignore`.

```
# typechecks just fine
a: int = "not an int" # type: ignore
```

WARNING

`# type: ignore` should not be an excuse to be lazy! When writing new code, don't ignore type errors and fix them as you go.

Your first goal for adopting type annotations is to get a completely clean run of your typechecker. If there are errors, you either need to fix them with annotations (recommended), or accept that not all errors can be fixed soon, and ignore them.

Over time, you want to make sure the number of ignored sections of code decrease. You can track the number of lines containing `# type : ignore` or the number of configuration file sections that you are using; no matter what, you want to strive to ignore as few sections as you can (within reasonable sense, of course - there is a law of diminishing returns).

I also recommend turning the `warn_unused_ignores` flag on in your configuration, which will warn when an ignore directive is no longer required.

Now, none of this helps you get any closer to actually annotating your codebase, it just gives you a starting point. To help annotate your codebase with tooling, you will need something that can automatically insert annotations.

MonkeyType

MonkeyType is a tool that will automatically annotate your Python code. This is a great way to typecheck a large amount of code without a lot of effort.

First install monkey type with pip:

```
pip install monkeytype
```

Suppose your codebase controls an automatic chef with robotic arms and was capable of cooking perfect food every time. You want to program the chef with my family's favorite recipe: Pasta With Italian Sausage:

```
# Pasta with Sasuage Automated Maker  ❶
italian_sausage = Ingredient('Italian Sausage', 4, 'links')
olive_oil = Ingredient('Olive Oil', 1, 'tablespoon')
plum_tomato = Ingredient('Plum Tomato', 6, '')
garlic = Ingredient('Garlic', 4, 'cloves')
black_pepper = Ingredient('Black Pepper', 2, 'teaspoons')
basil = Ingredient('Basil Leaves', 1, 'cup')
pasta = Ingredient('Rigatoni', 1, 'pound')
salt = Ingredient('Salt', 1, 'tablespoon')
water = Ingredient('Water', 6, 'quarts')
cheese = Ingredient('Pecorino Romano', 2, "ounces")
pasta_with_sausage = Recipe(6, [italian_sausage,
                                olive_oil,
                                plum_tomato,
                                garlic,
                                black_pepper,
                                pasta,
```

```
                                salt,
                                water,
                                cheese,
                                basil])

def make_pasta_with_sausage(servings): ❷
    sauté_pan = Receptacle('Sauté Pan')
    pasta_pot = Receptacle('Stock Pot')
    adjusted_recipe = adjust_recipe(pasta_with_sausage, servings)

    print("Prepping ingredients") ❸
    garlic_and_tomatoes =
recipe_maker.dice(adjusted_recipe.get_ingredient('Plum Tomato'),

adjusted_recipe.get_ingredient('Garlic'))
    grated_cheese =
recipe_maker.grate(adjusted_recipe.get_ingredient('Pecorino Romano'))
    sliced_basil =
recipe_maker.chiffonade(adjusted_recipe.get_ingredient('Basil
Leaves'))

    print("Cooking Pasta") ❹
    pasta_pot.add(adjusted_recipe.get_ingredient('Water'))
    pasta_pot.add(adjusted_recipe.get_ingredient('Salt'))
    recipe_maker.put_receptacle_on_stovetop(pasta_pot, 10)
    pasta_pot.add(adjusted_recipe.get_ingredient('Rigatoni'))
    recipe_maker.set_stir_mode(pasta_pot, ('every minute'))

    print("Cooking Sausage")
    sauté_pan.add(adjusted_recipe.get_ingredient('Olive Oil'))
    medium = recipe_maker.HeatLevel.MEDIUM
    recipe_maker.put_receptacle_on_stovetop(sauté_pan, medium)
    sauté_pan.add(adjusted_recipe.get_ingredient('Italian Sausage'))
    recipe_maker.brown_on_all_sides('Italian Sausage')
    cooked_sausage = sauté_pan.remove_ingredients(to_ignore=['Olive
Oil'])

    sliced_sausage = recipe_maker.slice(cooked_sausage,
thickness_in_inches=.25)

    print("Making Sauce")
    sauté_pan.add(garlic_and_tomatoes)
    recipe_maker.set_stir_mode(sauté_pan, ('every minute'))
    while recipe_maker.is_not_cooked('Rigatoni'):
        time.sleep(30)
    cooked_pasta = pasta_pot.remove_ingredients(to_ignore=['Water',
'Salt'])

    sauté_pan.add(sliced_sausage)
    while recipe_maker.is_not_cooked('Italian Sausage'):
```

```
            time.sleep(30)

        print("Mixing ingredients together")
        sauté_pan.add(sliced_basil)
        sauté_pan.add(cooked_pasta)
        recipe_maker.set_stir_mode(sauté_pan, "once")

        print("Serving") ❺
        dishes = recipe_maker.divide(sauté_pan, servings)

        recipe_maker.garnish(dishes, grated_cheese)
        return dishes
```

❶ Definition of all ingredients

❷ Function to make Pasta With Sausage

❸ Prepping instructions

❹ Cooking instructions

❺ Serving instructions

I've left out a lot of the helper functions to save space, but this gives you an idea of what I'm trying to achieve. You can see the full example in the GitHub repo that goes along with this book.

Throughout the entire example, I have zero type annotations. I don't want to write all the type annotations by hand, so I'll use monkeytype. To help, I can generate *stub files* to create type annotations. Stub files are files that just contain function signatures.

In order to generate the stub files, you have to run your code. This is an important detail; monkey type will only annotate code that you run first. You can run specific scripts like so:

```
monkeytype run code_examples/chapter7/main.py
```

This will generate a SQLite database that stores all the function calls made throughout the execution of that program. You should try to run as many parts of your system as you can in order to populate this database Unit tests, Integration tests, and test programs all contribute to populating the database.

> **TIP**
>
> Because monkeytype works by instrumenting your code using sys.setprofile, other instrumentation such as code coverage and profiling will not work at the same time. Any tool that uses instrumentation will need to be run separately.

Once you have run through as many paths of your code as you want, you can generate the stub files:

```
monkeytype stub code_examples.chapter7.pasta_with_sausage.
```

This will output the stub file for this specific module:

```
def adjust_recipe(
    recipe: Recipe,
    servings: int
) -> Recipe: ...


class Receptacle:
    def __init__(self, name: str) -> None: ...
    def add(self, ingredient: Ingredient) -> None: ...


class Recipe:
    def clear_ingredients(self) -> None: ...
    def get_ingredient(self, name: str) -> Ingredient: ...
```

It won't annotate everything, but it will certainly give you more than enough of a head start in your codebase. Once you are comfortable with the suggestions, you can apply them with `monkeytype apply <module-name>`. Once these annotations have been generated, search through the codebase for any use of `Union`. A `Union` tells you that more than one type has been passed to that function as part of the execution of your code. This is a *code smell*, or something that smells a little funny, even if it's not totally wrong (yet). In this case, the use of a `Union` may indicate unmaintainable code; you code is receiving different types and might not be equipped to handle them. If wrong types are passed in to a function, that's a sure sign that assumptions have been invalidated.

To illustrate, the stubs for my `recipe_maker`contains a `Union` in one of my function signatures:

```
def put_receptacle_on_stovetop(
    receptacle: Receptacle,
    heat_level: Union[HeatLevel, int]
) -> None: ...
```

The parameter heat_level has taken a HeatLevel in some cases, and an integer in other cases. Looking back at my recipe, I see the following lines of code:

```
recipe_maker.put_receptacle_on_stovetop(pasta_pot, 10)
# ...
medium = recipe_maker.HeatLevel.MEDIUM
recipe_maker.put_receptacle_on_stovetop(sauté_pan, medium)
```

Whether this is an error or not depends on the implementation of the function. In my case, I want to be consistent, so I would change the integer usage to `Enum` usage. For your codebase, you will need to determine what is acceptable and what is not.

Pytype

One of the problems with Monkeytype is that it only annotates code it sees at runtime. If there are branches of your code that are costly or unable to be run, monkeytype will not help you that much. Fortunately, a tool exists to fill in this gap:, pytype, written by Google. Pytype is completely done through static analysis, which means it does not need to run your code to figure out type annotations.

To run pytpye, install it with pip:

```
pytype code_examples/chapter7
```

This will generate a set of .pyi files in a `.pytype` folder. These are very similar to the stub files that mypy created. They contain annotated function signatures and variables that you can then copy into your source files.

Pytype offers other intriguing benefits as well. Pytype is not just a type annotator; it is a full linter and typechecker. It has a different typechecking philosophy than other typecheckers such as mypy, pyright and pyre.

Pytype will use inference to do its typechecking, which means it will typecheck your code, even in the absence of type annotations. This is a great way to get the

benefit of a typechecker without having to write types throughout your codebase.

Pytype is a also little more lenient on types changing in the middle of their lifetime. This is a boon for those who fully embrace Python's dynamically typed nature. As long as code will work at runtime, pytype is happy. For instance:

+

```
# Run in Python 3.6
from typing import List
def get_pasta_dish_ingredients(ingredients: List[Ingredient]
                               ) -> List[str]:
names = ingredients
# make sure there is water to boil the pasta in
if does_not_contain_water(ingredients)
    names.append("water")
return [str(i) for i in names]
```

+ In this case, names will start off as a list of `Ingredients`. If there is no water in the ingredients, I add the string "water" to the list. At this point, the list is heterogeneous; it contains both ingredients and strings. If you were to annotate names as a `List[Ingredient]`, mypy would error out in this case. I would typically throw a red flag here as well; heterogeneous collections are harder to reason about in the absence of good type annotations. However, the next line renders both mypy and my objections moot. Everything is getting converted to a string when returned, which fulfills the annotation of the expected return type. Pytype is intelligent enough to detect this, and consider this code to have no issues.

Pytype's leniency and approach to typechecking make it very forgiving for adopting into existing codebases. You don't need any type annotations in order to see the value. This means you get all the benefits of typechecker with very minimal work. High value, but low cost? Yes, please.

However, pytype is a double-edged sword in this case. Make sure you don't use pytype as a crutch; you should still be writing type annotations. It becomes incredibly easy with pytype to think that you don't need type annotations at all. However, you should still write them for two reasons:

1. Type annotations provide a documentation benefit, which helps your code's readability

2. Pytype will be able to make even more intelligent decisions if type annotations are present.

Wrap-up

Type annotations are incredibly useful, but there is no denying their cost. The larger the codebase, the higher the cost will be for practically adopting type annotations. Every codebase is different; you need to evaluate the value and cost of type annotations for your specific scenario. If type annotations are too costly to adopt, consider three strategies to get past that hurdle:

Find Pain Points

> If you can eliminate entire classes of pain points through type annotations, such as errors, broken tests or unclear code, you will save time and money. You target the areas that hurt the most, and by lessening that pain, you are making it easier for developers to deliver value over time (which is a sure sign of maintainable code).

Target Code Strategically

> If you can eliminate entire classes of pain points through type annotations, such as errors, broken tests or unclear code, you will save time and money. You target the areas that hurt the most, and by lessening that pain, you are making it easier for developers to deliver value over time (which is a sure sign of maintainable code)

Lean On Your Tooling

> Use mypy to help you selectively ignore files (and make sure that you are ignoring fewer lines of code over time). Use type annotators such as monkeytype and pytype to quickly generate types throughout your code. Don’t discount pytype as a typechecker either, as it’s intelligence can find bugs lurking in your code with minimal setup.

This wraps up Part 1 of this book. It has focused exclusively on type annotations and typechecking. Feel free to mix and match strategies and tools. You don’t need to type annotate absolutely everything, as type annotations can constrain

expressiveness if too strictly applied. But you should strive to clarify code and make it harder for bugs to crop up. You will find the balance over time, but you need to start thinking about types in Python and how you can express them to other developers. Remember, the goal is a maintainable codebase. People need to understand your as much of your intentions as they can from the code alone.

In Part 2, I'm going to focus on creating your own types. You've seen a little of this with building your own collection types, but you can go so much further. You'll learn about enumerations, dataclasses and classes, and learn why you should pick one over the other. You'll learn how to craft an API, subclass types, and model your data. You'll continue to build a vocabularly that improves readability in your codebase.

---

1 Working Effectively With Legacy Code, Michael Feathers. 2004

About the Author

Patrick Viafore has been working in the software industry for 13+ years, working on mission critical software systems, including in lightning detection, telecommunications and operating systems. His work in static typed languages has influenced his approach to dynamic languages and how we can make them safer and more robust. He also is an organizer of the HSV.py meetup, where he can observe common Python obstacles developers face, helping both beginners and experts alike. His goal is to make computer science/software engineering topics more approachable to the developer community.

Patrick currently works at Canonical, developing pipelines/tools to deploy Ubuntu images to public cloud providers. He also does software consulting and contracting through his business Kudzera, LLC.

1. 1. Introduction to Robust Python
    a. Robustness
        i. What Does “Robust” Mean?
        ii. Why Does Robustness Matter?
    b. What’s Your Intent?
        i. Asynchronous Communication
    c. Examples of Intent In Python
        i. Collections
        ii. Iteration
        iii. Law of Least Surprise
    d. Wrap-up
2. 2. Introduction to Python Types
    a. What’s In a Type?
        i. Mechanical Representation
        ii. Semantic Representation
    b. Typing Systems
        i. Strong vs. Weak
        ii. Dynamic vs. Static
        iii. Duck Typing
    c. Wrap-up
3. 3. Type Annotations
    a. Type Annotations

    b. Benefits
        i. Autocomplete
        ii. Typecheckers
        iii. Exercise: Spot the Bug
    c. When To Use
    d. Wrap-up
4. 4. Constraining Types
    a. Optional Type
    b. Union Types
        i. Product and Sum Types
    c. Literal Types
    d. Annotated Types
    e. NewType
    f. Final Types
    g. Wrap-up
5. 5. Collection Types
    a. Annotating Collections
    b. Homogeneous vs. Heterogeneous Collections
    c. TypedDict
    d. Creating New Collections
        i. Generics
        ii. Modifying Existing Types
        iii. As Easy as ABC

    e. Wrap-up
6. 6. Customizing Your Typechecker
    a. Configuring Your Typechecker
        i. Configuring Mypy
        ii. Mypy Reporting
        iii. Speeding up Mypy
    b. Alternative Typecheckers
        i. Pyre
        ii. Pyright
    c. Wrap-up
7. 7. Adopting Typechecking Practically
    a. Trade-offs
    b. Breaking Even Earlier
        i. Finding Your Pain Points
        ii. Target Code Strategically
        iii. Lean On Your Tooling
    c. Wrap-up