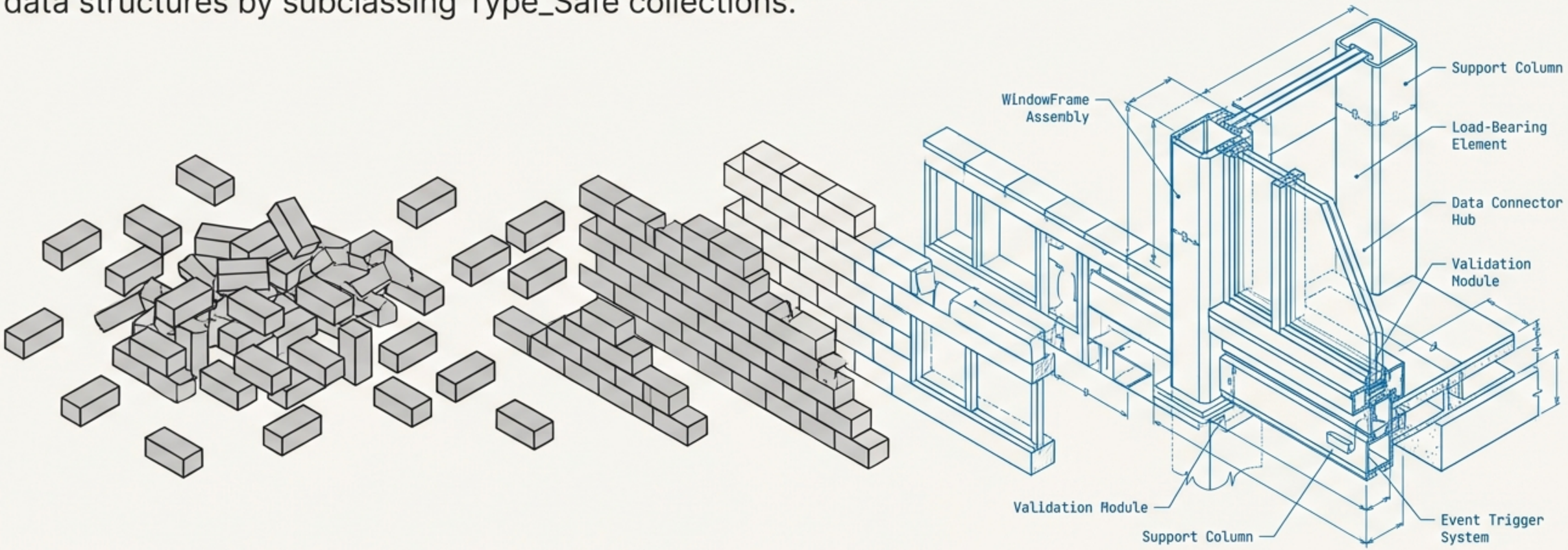
From Generic to Semantic: Mastering Type_Safe Collections

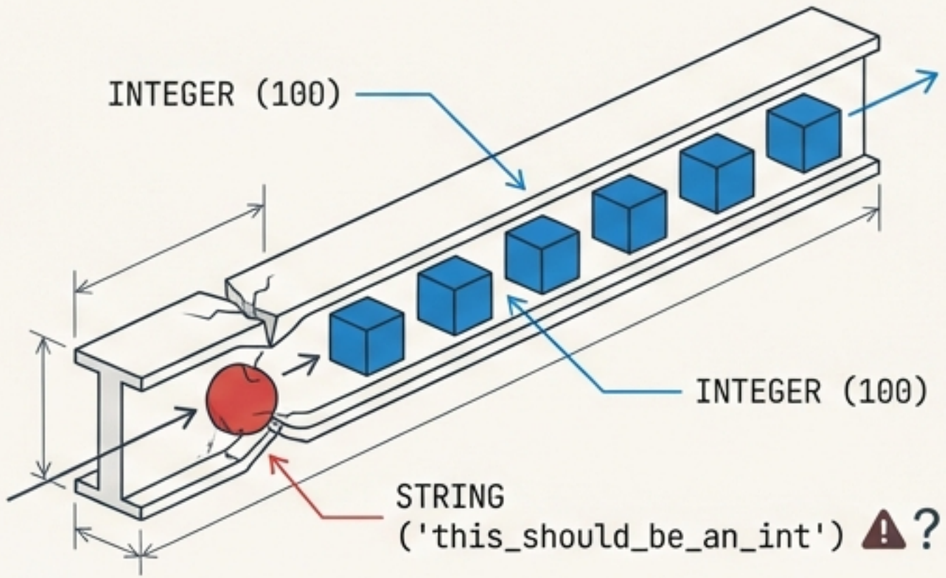
A guide to building robust, reusable, and self-documenting data structures by subclassing Type_Safe collections.



The Hidden Danger in Standard Python Collections

Python's **type hints** (`Dict[str, int]`) are **ignored at runtime.** They are merely suggestions.

```python
# This code runs without error
users: dict[str, int] = {"user_1": 100}
users["user_2"] = "this_should_be_an_int" # No error!
```



This leads to critical issues:

- **Silent Data Corruption:** Wrong types enter your data structures undetected.

- **Delayed Failures:** Errors appear far from where the bad data was introduced.

- **Debugging Nightmares:** "How did a float get into my `Dict[str, User]`?"

The First Line of Defence: `Type_Safe''s Runtime Enforcement

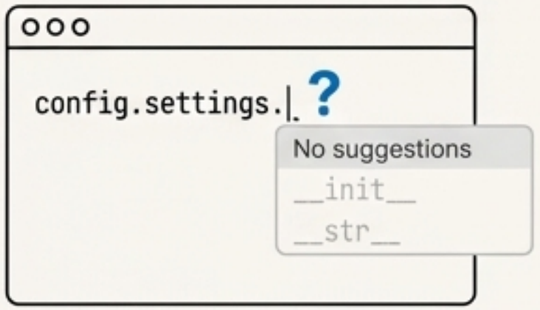
`Type_Safe` collections provide runtime type enforcement for every operation, automatically converting annotated fields.

Capability	Python Built-in	`Type_Safe` Collections
Runtime type checking	❌ None	✔ Every operation
Auto-conversion	❌ None	✔ `"123"` → `Safe_Id("123")`
Validation on insert	❌ None	✔ Invalid data rejected
Works with `Type_Safe` classes	❌ Loses type info	✔ Full integration
JSON serialization	❌ Manual	✔ Automatic with type preservation

```python
class MyConfig(Type_Safe):
    # Auto-converts to a Type_Safe__Dict at runtime
    settings: Dict[str, int]

# This now fails instantly and correctly!
MyConfig(settings={"volume": "11"}) # -> ValidationError
```

The Frustration of Inline Annotations

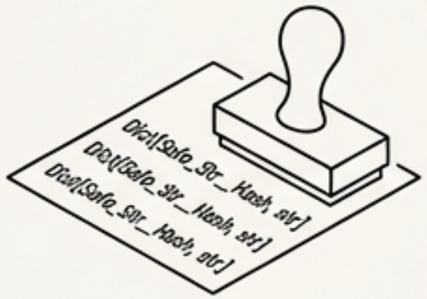
While auto-conversion provides safety, relying solely on inline annotations like `Dict[str, int]` creates practical problems in a growing codebase.



1. IDE Type Mismatch

Your IDE sees the standard `dict` type, not the powerful `Type_Safe__Dict` that exists at runtime, leading to poor autocompletion.

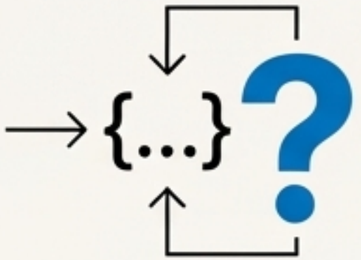
```
# config.settings is a Type_Safe__Dict at runtime,
# but your IDE thinks it's a plain dict.
config.settings.[?] # No Type_Safe methods suggested
```



2. Repetitive & Error-Prone

The same complex type combination must be repeated across your application.

```
def process_hashes(hashes: Dict[Safe_Str__Hash, str]): ...

class Asset(Type_Safe):
    content: Dict[Safe_Str__Hash, str]
```



3. No Semantic Meaning

The type annotation describes the structure (`Dict[Key, Value]`) but not the *purpose* of the data.

```
# What does this dictionary represent?
# A mapping of content hashes to file paths?
# A cache of request hashes to responses?
metadata: Dict[Safe_Str__Hash, str]
```

The Blueprint for Clarity: Subclassing Collections

Instead of repeating generic types, we can create named, reusable, and self-documenting collection types.

Generic Bricks

```
# Repetitive, lacks semantic meaning
# Repetitive, lacks semantic meaning
class Asset(Type_Safe):
    hashes: Dict[Safe_Str__Hash, str]

def process_assets(assets: List[Dict[Safe_Str__Hash, str]]):
    ...
```

Architectural Components

```
# Reusable, clear, self-documenting
class Dict__Content__Hashes(Type_Safe__Dict):
    expected_key_type = Safe_Str__Hash
    expected_value_type = str

class Asset(Type_Safe):
    hashes: Dict__Content__Hashes

def process_assets(assets: List[Dict__Content__Hashes]): ...
```

The Core Pattern: Defining a `Type_Safe__Dict` Subclass

Create a new class inheriting from `Type_Safe__Dict` and define the key and value types as class-level attributes.

```
# A reusable, named type for mapping user IDs to User objects
class Dict__Users__By_Id(Type_Safe__Dict):
    # The expected type for keys in the dictionary.
    expected_key_type = Safe_Id

    # The expected type for values in the dictionary.
    expected_value_type = Schema__User

# --- USAGE ---

# 1. Creating with initial data
users = Dict__Users__By_Id({
    Safe_Id("u-1"): Schema__User(name="Alice"),
    Safe_Id("u-2"): Schema__User(name="Bob")
})

# 2. Using as a type annotation
class Project(Type_Safe):
    members: Dict__Users__By_Id
```

Inherit from `Type_Safe__Dict`

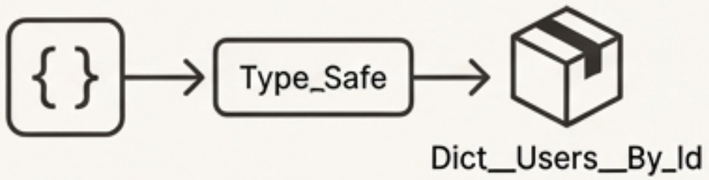
Define the key's type

Define the value's type

Use as a clear type annotation

Seamless Integration, Powerful Results

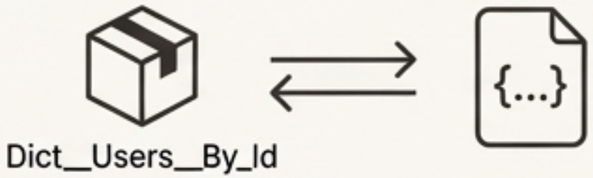
Subclasses are first-class citizens in the `Type_Safe` ecosystem, providing a frictionless development experience.



Auto-Conversion from Plain `dict`s

`Type_Safe` automatically converts a raw `dict` into your subclass type on assignment.

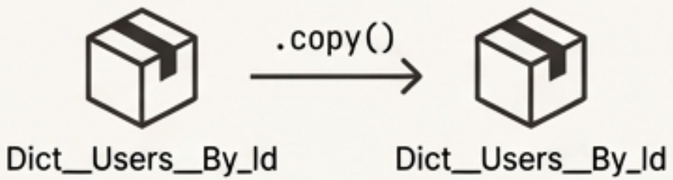
```
class System(Type_Safe):
    users: Dict__Users__By_Id

# This plain dict is automatically converted to Dict__Users__By_Id
system = System(users={"u-1": {"name": "Alice"}})
```



JSON Serialization

Subclasses automatically serialize to plain JSON and deserialize back to the correct subclass type.

```
json_str = system.to_json()
# -> '{"users": {"u-1": {"name": "Alice"}}}'

rehydrated = System.from_json(json_str)
# rehydrated.users is a Dict__Users__By_Id instance
```



Operations Preserve Subclass Type

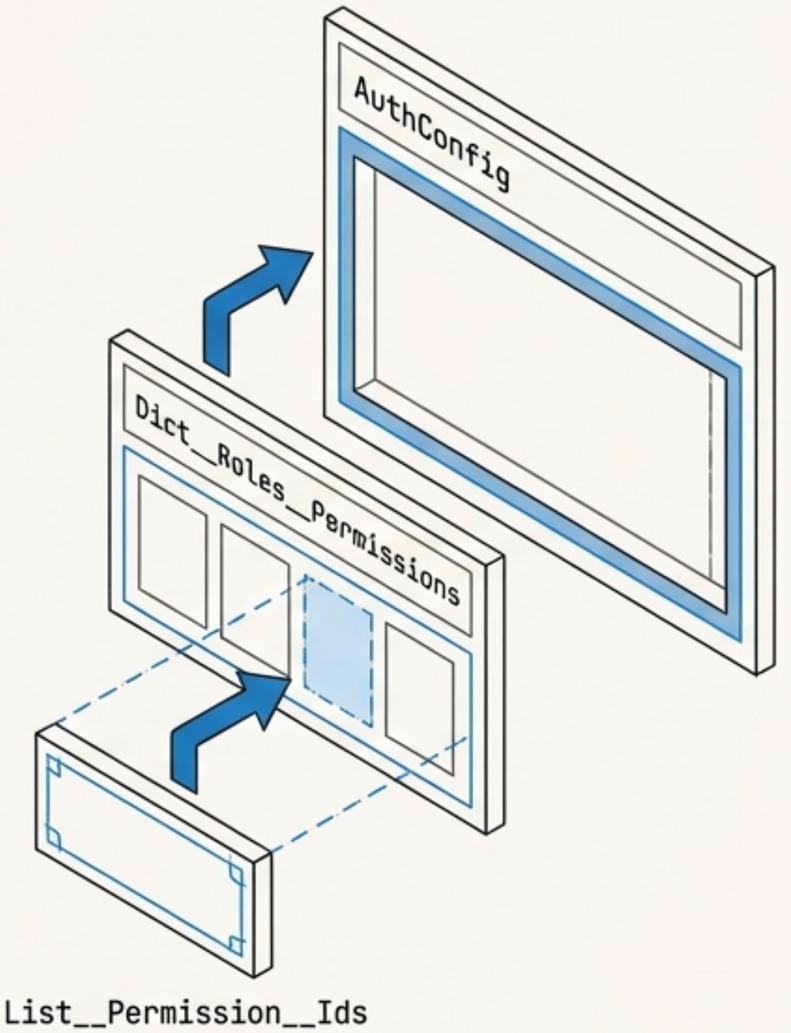
Standard dictionary operations like `.copy()` return an instance of your subclass, not the base `Type_Safe__Dict`.

```
copied_users = system.users.copy()
# type(copied_users) is Dict__Users__By_Id
```

Building with Components: Nested & Composed Collections

Subclasses can be nested to create sophisticated, layered data structures that remain clear and type-safe.

```
# Component 1: A list of permission IDs
class List__Permission__Ids(Type_Safe__List):
    expected_type = Safe_Str__PermissionId

# Component 2: A dictionary mapping roles to permissions
class Dict__Roles__Permissions(Type_Safe__Dict):
    expected_key_type = Safe_Str__RoleName
    expected_value_type = List__Permission__Ids

# Final Structure: The composed configuration object
class AuthConfig(Type_Safe):
    roles: Dict__Roles__Permissions

# --- USAGE ---
# Type safety is enforced at every level of this nested structure
config = AuthConfig(roles={
    "admin": ["perm-read", "perm-write"],
    "viewer": ["perm-read"]
})
```



The Pattern Extends to `List`, `Set`, and `Tuple`

The same subclassing principle provides clarity and reusability for all collection types.

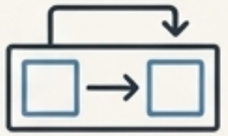
`Type_Safe__List` Subclass

```
class List__Event__Log(Type_Safe__List):
    expected_type = Schema__Event

class Session(Type_Safe):
    events: List__Event__Log
```

`Type_Safe__Set` Subclass

```
class Set__Active__Features(Type_Safe__Set):
    expected_type = Safe_Str__FeatureFlag

class UserProfile(Type_Safe):
    features: Set__Active__Features
```

A Note on `Type_Safe__Tuple`

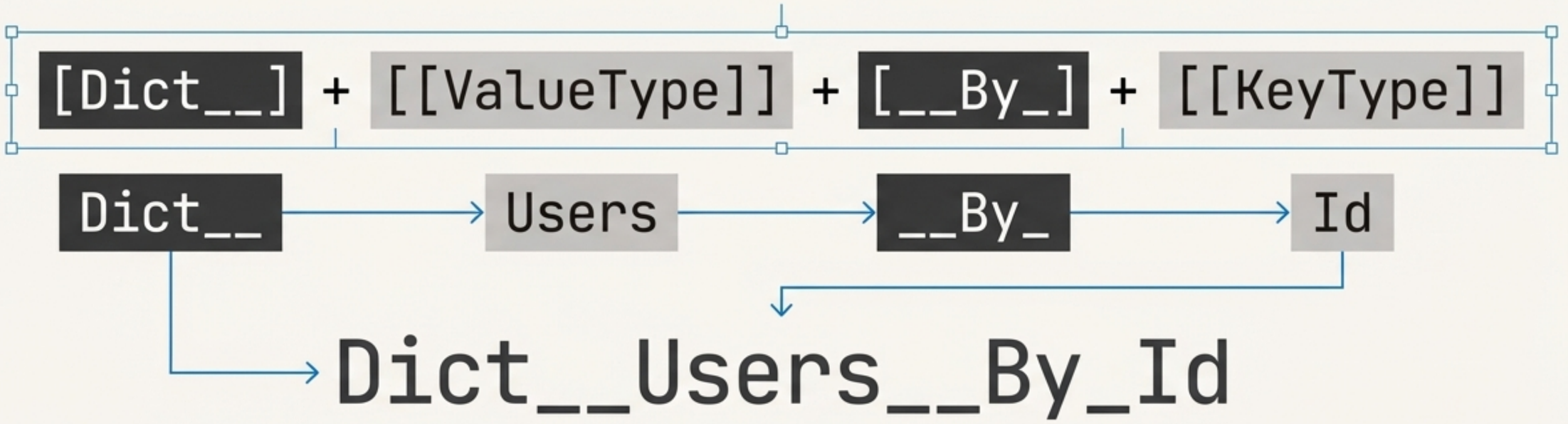


Tuples have fixed positional types. While subclassing is possible, inline annotations are often clearer for simple cases.

```
# For tuples, this is often sufficient
location: Tuple[float, float]
```

Best Practice: Naming for Instant Recognition

Always prefix subclass names with the collection type: `Dict__`, `List__`, `Set__`.



Why this pattern?

Consistency: Mirrors the `Schema__` prefix for `Type_Safe` classes, creating a consistent mental model. (`Schema__User` vs. `Dict__Users__By_Id`).

Immediate Recognition: You know it's a dictionary just by looking at the name, without checking its base class.

Clear Distinction: Separates collection types (`Dict__`) from data structures (`Schema__`).

Project-Wide Searchability: Easily find all dictionary subclasses with a simple search for `class Dict__`.

Best Practice: Creating a Robust Domain Concept

Combine naming conventions with primitive types and clear documentation to define truly robust domain concepts.

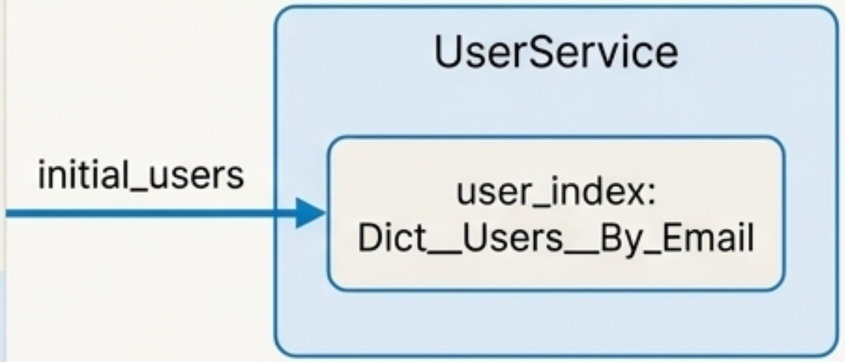
```
# 1. Use for a core domain concept, not just a generic structure.
# 3. Add inline comments explaining constraints (no docstrings).
class Dict__Products__By_Sku(Type_Safe__Dict):
    # A map of unique product SKUs to their corresponding product schemas.
    expected_key_type = Safe_Str__Sku # 2. Use Type_Safe__Primitive keys.
    expected_value_type = Schema__Product

    # This class is now a self-contained, self-documenting, and
    # highly reusable component of your application's domain model.
```

1. **Model Domain Concepts:** Create subclasses when the collection represents a meaningful business concept (e.g., a product index).
2. **Use Primitive Keys:** Combine with `Type_Safe__Primitive` keys like `Safe_Id` or `Safe_Str__Sku` for maximum type safety.
3. **Document with Comments:** Use simple inline comments to explain the purpose and constraints.

In Practice: The Registry/Index Pattern

A `Dict__` subclass is the ideal way to implement a type-safe registry or an in-memory index of domain objects.

```python
# The registry class defines the structure
class Dict__Users__By_Email(Type_Safe__Dict):
    expected_key_type = Safe_Str__Email
    expected_value_type = Schema__User

# The application service uses the registry
class UserService:
    def __init__(self, initial_users: list[Schema__User]):
        # Validate data at the boundary by constructing the subclass
        self.user_index = Dict__Users__By_Email({
            user.email: user for user in initial_users
        })

    def find_user(self, email: Safe_Str__Email) -> Schema__User | None:
        return self.user_index.get(email)
```



Using the subclass constructor (`Dict__Users__By_Email(...)`) is a powerful way to validate entire datasets at application boundaries.

Decision Guide: When to Subclass vs. Use Inline

Choose the right tool for the job. **Subclassing** is for **architecture**; inline annotations are for **local convenience**.

Subclass This... (`Dict__Users__By_Id`)	Use Inline For This... (`Dict[str, bool]`)
✓ When a collection represents a core **domain concept**.	✓ For **one-off** or trivial combinations.
✓ When the type is **reused** in multiple places.	✓ When the structure has no semantic meaning beyond its types.
✓ When you need a **self-documenting** name.	✓ Inside a function's implementation where reuse is not a concern.
✓ When it's part of a public API or shared data structure.	✓ For simple, temporary variables.

Rule of Thumb

If you have to write the same `` `Dict[...]` `` or `` `List[...]` `` annotation more than once, it's time to create a subclass.

Summary & Implementation Checklist

Use this checklist to build clear, robust, and maintainable collections in your Type_Safe applications.

- [ ] **Prefix with Collection Type:** Start names with `Dict__`, `List__`, or `Set__`.
- [ ] **Define Class-Level Types:** Set `expected_key_type` / `expected_value_type` (for `Dict`) or `expected_type` (for `List`/`Set`).
- [ ] **Use Meaningful Domain Names:** e.g., `Dict__Users__By_Email`.
- [ ] **Prefer `Type_Safe__Primitive` Keys:** Use types like `Safe_Id` for dictionary keys.
- [ ] **Trust Auto-Conversion:** Let Type_Safe handle conversion from plain `dict`'s and `list`'s.
- [ ] **Remember Operations Preserve Type:** Methods like `.copy()` will return your subclass instance.
- [ ] **Subclass for Reusability & Semantics:** If it's used more than once or has a specific meaning, give it a name.
- [ ] **Use Inline for Simplicity:** For one-off, local type combinations, inline is fine.

Import Reference

The necessary building blocks for creating collection subclasses.

```
# Base collection types for subclassing
from type_safe import (
    Type_Safe__Dict,
    Type_Safe__List,
    Type_Safe__Set,
    Type_Safe__Tuple
)

# Base class for creating custom data structures
from type_safe import Type_Safe

# Example primitive types for keys and values
from type_safe import (
    Safe_Id,
    Safe_Str__Email,
    Safe_Str__Hash
)
```