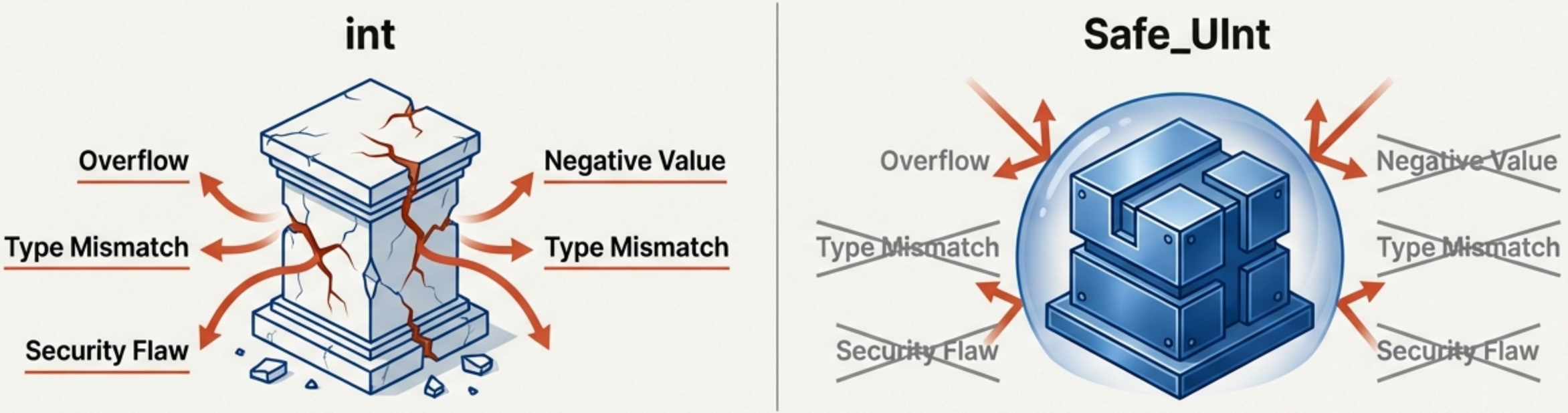
The Primitives You Trust Are Silently Breaking Your Code.



Standard Python types like `str`, `int`, and `float` are too permissive. They allow for a vast range of bugs and security vulnerabilities that type hints alone cannot prevent. `Type_Safe` is a runtime framework that eliminates these risks by enforcing type constraints and providing domain-specific primitives for correctness and security.

Inter Bold

- ☑ Runtime type enforcement on every operation.
- ☑ Domain-specific primitives for IDs, URLs, money, and more.
- ☑ Auto-initialisation of attributes.
- ☑ Type-preserving JSON serialisation.

Our Critical Principle: Ban Raw Primitives.

NEVER use raw `str`, `int`, or `float` in `Type_Safe` classes.

The full range and capabilities of raw primitives are rarely needed and introduce entire categories of errors. By replacing them with specialised types, we eliminate these vulnerabilities by design.

BEFORE (Unsafe)

```
class User:
    user_id: int
    email: str
    balance: float
```

AFTER (Type_Safe)

```
class User(Type_Safe):
    user_id: Safe_Str__Id
    email: Safe_Str__Email
    balance: Safe_Float__Money
```

Why Raw Primitives Are a Liability.

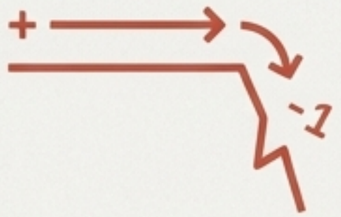
Raw primitives have been the root cause of countless major bugs and security incidents. Their lack of constraints makes your application vulnerable at its very foundation.

`str`

Risks

- SQL Injection
- Cross-Site Scripting (XSS)
- Buffer Overflows
- Command Injection

`int`



Risks

- Integer Overflow Bugs
- Unexpected Negative Values where only positive values are valid.

`float`

Risks

- Financial Calculation Errors
- Catastrophic Precision Loss

The Architectural Foundations of `Type_Safe`.

Type_Safe__Base

Responsibility: Core type checking and conversion logic.

- `is_instance_of_type()`
- `try_convert()`

Type_Safe__Primitive

Responsibility: The foundation for all safe primitive types (`Safe_Str`, `Safe_Int`).

- Extends built-in types
- Overrides operators to maintain safety.

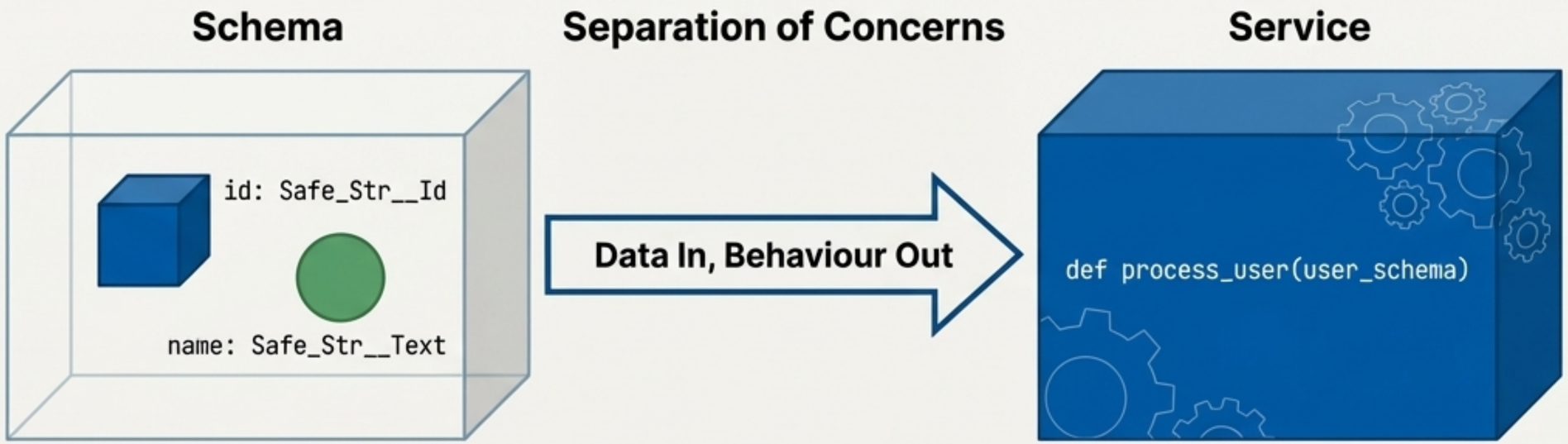
Type_Safe

Responsibility: The main class for all user-defined data schemas.

- Auto-initialisation
- Attribute checking via `__setattr__`
- JSON handling.

Schemas Must Be Pure Data Containers

⚠ CRITICAL: Schema classes that inherit from `Type_Safe` should **ONLY** contain type annotations. **NO methods, NO business logic.**



- **Clarity:** Schemas define the *structure* of data. Behaviour and logic belong in separate service or helper classes.
- **Reliability:** This separation of concerns ensures clean, predictable serialisation and deserialisation.
- **Maintainability:** Upholds the Single Responsibility Principle, making the codebase easier to understand and modify.

The Five Foundational Rules for `Type_Safe` Schemas.

1. **Always Inherit from `Type_Safe`:** This is the entry point to all framework features.

2. **Type Annotate Everything:** Every attribute must have a clear type definition.

3. **Use Immutable Defaults Only:** Assigning mutable defaults like `[]` or `{}` is forbidden to prevent shared state bugs.

4. **Forward References for Current Class Only:** Use string-based forward references (e.g., `'MyClass'`) only for self-referencing types.

5. **Simplify Default Value Assignment:** You don't need constructors for defaults. `Safe_Str` defaults to `''`, and `Safe_Int`/`Safe_Float` default to `0`/`0.0`. Only specify a default if it's different.

Enforcing Runtime Safety on Methods with `@type_safe`.

The `@type_safe` decorator brings the framework's validation capabilities to your method parameters and return values.

Call-Time Validation

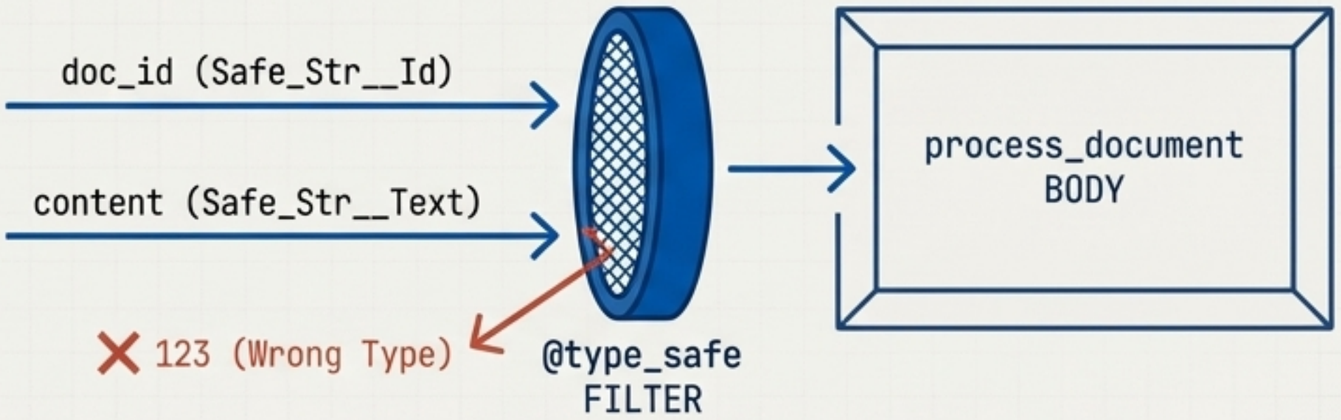
- Checks all parameter types when a method is called.
- Validates elements within collections (e.g., `List[Safe_Str]`).
- Supports `Union` and `Optional` types.

Return-Time Validation

- Ensures the return value matches its type annotation.
- Automatically converts raw Python primitives to their `Safe_*` equivalents.

```python
from type_safe import type_safe, Safe_Str__Id, Safe_Str__Text

@type_safe
def process_document(doc_id: Safe_Str__Id, content: Safe_Str__Text) -> bool:
    # Code is guaranteed to receive the correct types
    ...
    return True
```



For comprehensive details, see v3_63_3__for_llms__type_safe__decorator_guide.md.

Building Robust Structures with Type-Safe Collections.

`Type_Safe` provides its own collection types that validate elements on **every operation** (e.g., `append`, `__setitem__`).

Available collections: `Type_Safe__List` `Type_Safe__Dict` `Type_Safe__Set` `Type_Safe__Tuple`

CRITICAL Rule for Subclassing

- When creating reusable collection types, you **must** prefix the class name with the collection type (e.g., `List__`, `Dict__`).
- These subclass definitions must be **pure type definitions** – NO methods are allowed.

```
# Definition (Pure Type)
class List__User_Ids(Type_Safe__List):
    __type__ = Safe_Str__Id

# Usage
user_ids = List__User_Ids(['user_1', 'user_2']) # Correct
# user_ids.append(123) # This would raise a runtime error
```

The Critical Distinction: Instance IDs vs. Semantic IDs

Choosing the correct ID type is essential for data integrity and predictable serialisation.

Safe_Id (and other Instance IDs)

Behaviour: Auto-generates a unique value (via `Obj_Id()`) if instantiated empty or with `None`.

Use For: Instance identifiers that must be unique.

Examples: `Node_Id`, `Edge_Id`, `Graph_Id`

Warning: Breaks serialisation round-trips, as a new ID is generated on deserialisation.

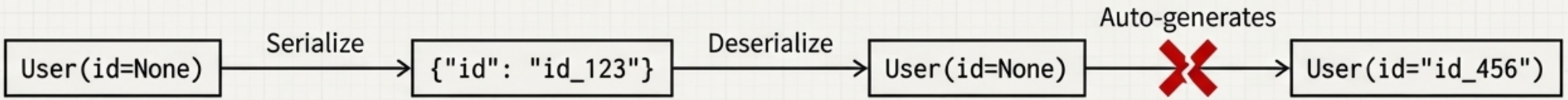
Semantic_Id

Behaviour: **NEVER** auto-generates. An empty value remains empty.

Use For: Human-readable or conceptual identifiers that have inherent meaning.

Examples: `Ontology_Id`, `Node_Type_Id`, `Rule_Set_Id`

Benefit: Ensures stable and predictable serialisation.



Your Toolkit: The Safe Primitives Reference.

`Type_Safe` includes a rich library of domain-specific primitives. Use the most specific type possible for maximum safety.

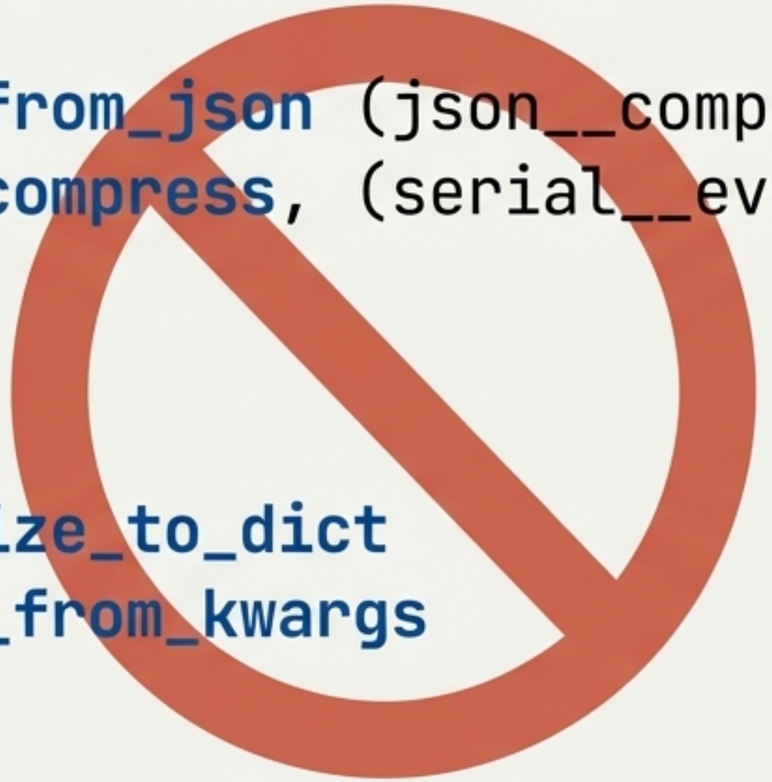
Category	Type	Purpose
S **Core String**	`Safe_Str__Id`	Alphanumeric + underscore + hyphen identifiers
	`Safe_Str__Text`	General purpose text (up to 4KB)
🌐 **Web**	`Safe_Str__Url`	Validates URL format
	`Safe_Str__Email`	Validates email address format
🧠 **LLM**	`Safe_Str__LLM__Prompt`	For LLM prompts (up to 32KB)
	`Safe_Str__LLM__Model_Id`	For model identifiers
123 **Numeric**	`Safe_UInt`	Unsigned integer (min_value=0)
	`Safe_Float__Money`	For currency, uses decimal arithmetic
🫆 **Identity**	`Safe_Id`	Auto-generates if instantiated empty
	`Random_Guid`	Always generates a new UUID v4

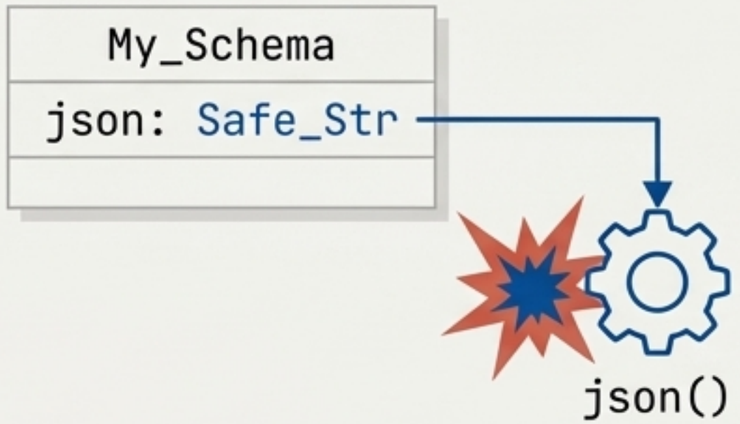
For a complete catalog, see `v3.28.0__osbot-utils-safe-primitives__reference-guide.md`.

Avoid Conflicts: Do Not Use Reserved Attribute Names.

`Type_Safe` classes have built-in methods for core functionality. Using these names for your attributes will shadow the methods and cause unexpected behaviour or errors.

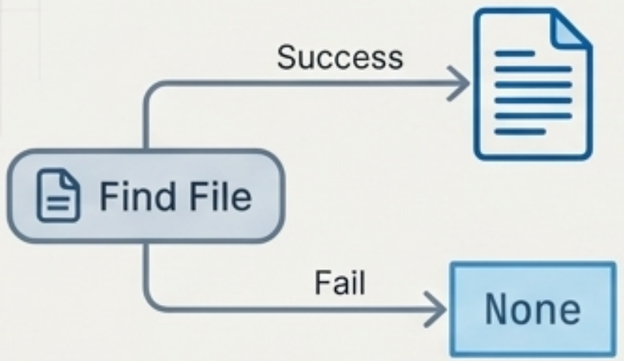
- `json`, `from_json` (`json__compress`)
- `json__compress`, (`serial__event`)
- `obj`
- `reset`
- `print`
- `serialize_to_dict`
- `update_from_kwargs`





The `Type_Safe` Way: Handling `None`, Files, and JSON.

Return `None` for Missing Resources



Functions that look for a resource (like a file) should return `None` on failure, not raise an exception (e.g., `FileNotFoundError`).

A missing resource is a valid state, not an exceptional one. The caller has the context to decide how to handle it.

Use `OSBot-Utils` for I/O

Always use the provided utility functions (`json_load_file`, `json_save_file`, etc.) instead of Python's built-in `open()` or `json` modules.

These utilities are integrated with the framework's philosophy, returning default empty values (`{}`) on error instead of raising exceptions.

Never Use `str()` Casts

Do not explicitly cast variables with `str()`.

Rationale:
The framework's type conversion logic handles this automatically and safely. Explicit casting bypasses these safety checks.

Your `Type_Safe` Development Checklist

- [ ] Inherit from `Type_Safe`.
- [ ] Use `Safe_*` types for all attributes; ban raw primitives.
- [ ] Ensure schemas are PURE DATA (no methods).
- [ ] Ensure collection subclasses are PURE TYPE DEFINITIONS.
- [ ] Add type annotations for ALL attributes.
- [ ] Use immutable defaults only.
- [ ] Prefix collection subclasses (`Dict__`, `List__`, etc.).
- [ ] Use `@type_safe` to validate methods.
- [ ] Return `None` for 'not found'; no `Optional[T]` needed in return types.
- [ ] Do not use attribute names that shadow reserved methods (`json`, `reset`, etc.).
- [ ] Use `OSBot-Utils` for all file and JSON operations.
- [ ] Never use explicit `str()` casts.

A Guide to Deeper Knowledge.

When your work involves these specific areas, request the related documentation to ensure you are following the latest best practices.
Ask your LLM assistant: “Can you provide the [document name] for reference?”

Topic	Document to Request
Core Style	v3.63.4__for_llms__python_formatting_guide.md
Decorator Usage	v3.63.3__for_llms__type_safe__decorator_guide.md
Collections	v3.63.3__for_llms__type_safe__collections__subclassing_guide.md
All Primitives	v3.28.0__osbot-utils-safe-primitives__reference-guide.md
Testing	v3.1.1__for_llms__type_safe__testing_guidance.md
JSON Handling	v3.63.4__json-utility__llm-usage-brief.md

Build with Certainty.

`Type_Safe` is not about adding constraints. It is about removing entire classes of uncertainty from your code.