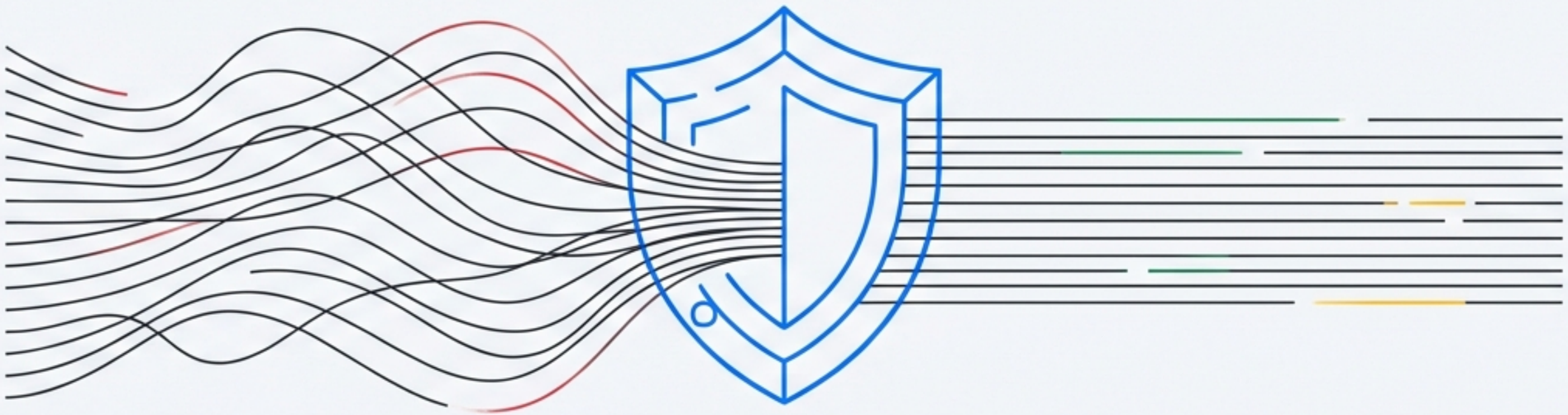
OSBot-Utils: Safe Primitives

Build Safer, Smarter Systems with Type-Safe Foundations.

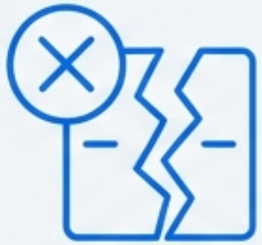


Version v3.28.0 | For the Modern Python Developer

Why Your Data Model is Your First Line of Defence.

1. Runtime Errors

Unexpected `ValueError` and `TypeError` exceptions crashing production systems due to unvalidated data.



2. Security Flaws

Unsanitised input leading to injection vulnerabilities or critical logic errors that bypass security controls.

3. Ambiguous Code

What does `user_id: str` actually mean? A UUID? An email? A database key? This ambiguity leads to bugs.

4. Maintenance Overhead

Validation logic is scattered across services, controllers, and utility functions, making it difficult to maintain and keep consistent.

Enforce Data Integrity at the Point of Creation.

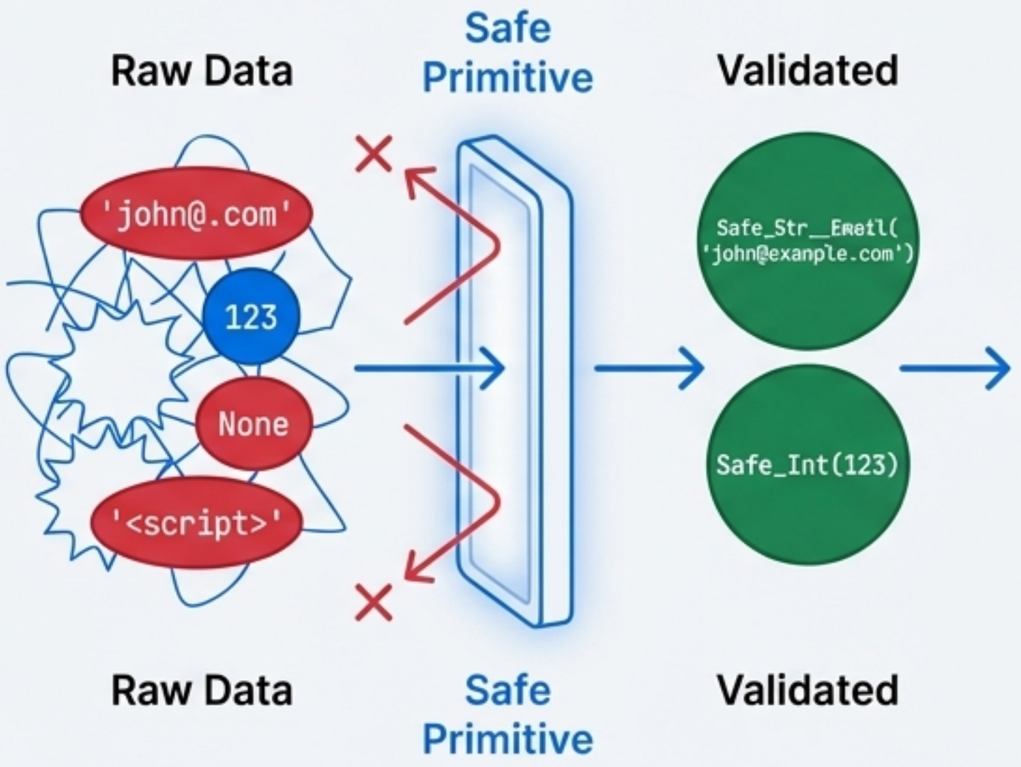
Compile-Time Safety

Catch invalid data at creation time, not during runtime. Your IDE becomes your first validator.

Self-Documenting

Type annotations like `Safe_Str__Email` clearly communicate constraints, making codebases easier to understand and use correctly.

The 'Refinement' Diagram



Zero Performance Overhead

Validation happens once at instantiation. After that, the object behaves like a native Python type with no added cost.

Composable

Designed to be used within `Type_Safe` classes to build completely validated and secure data objects.

The Core Building Blocks: `Int`, `Float`, and `Str`.

`Safe_Int`

For type-safe integers with extensive validation rules.

Feature Highlight

Auto-clamping to a min/max range with `clamp_to_range=True`, preventing `ValueError` on out-of-bounds input.

`Safe_Float`

For type-safe floats with precise decimal handling.

Feature Highlight

Financial-grade accuracy using `use_decimal=True`, which leverages Python's `Decimal` type to avoid floating-point errors.

`Safe_Str`

For robust strings with powerful regex validation and sanitisation.

Feature Highlight

Flexible `regex_mode` allows either strict matching (`MATCH`) or silent sanitisation (`REPLACE`) of invalid characters.

The Everyday Workhorses for Text, Code, and Files.

`Safe_Str__Version`

Enforces strict semantic versioning format (`v{major}.{minor}.{patch}`).

`Safe_Str__Code__Snippet`

Preserves indentation and allows common programming symbols, ideal for storing code fragments.

`Safe_Str__File__Name`

Sanitises filenames to prevent path traversal attacks by removing invalid characters like `/` or `..`.

`Safe_UInt__FileSize`

Represents a file size in bytes and provides convenient conversion methods like `.to_kb()`, `.to_mb()`, and `.to_gb()`.

Hardening Your Application at the Network Edge.

`Safe_Str__Email`

Ensures a string conforms to a valid email structure (e.g., requires an `@` symbol). Max length of 256 characters.

`Safe_Str__Url`

Mandates a string starts with `http://` or `https://` and contains only valid URL characters.

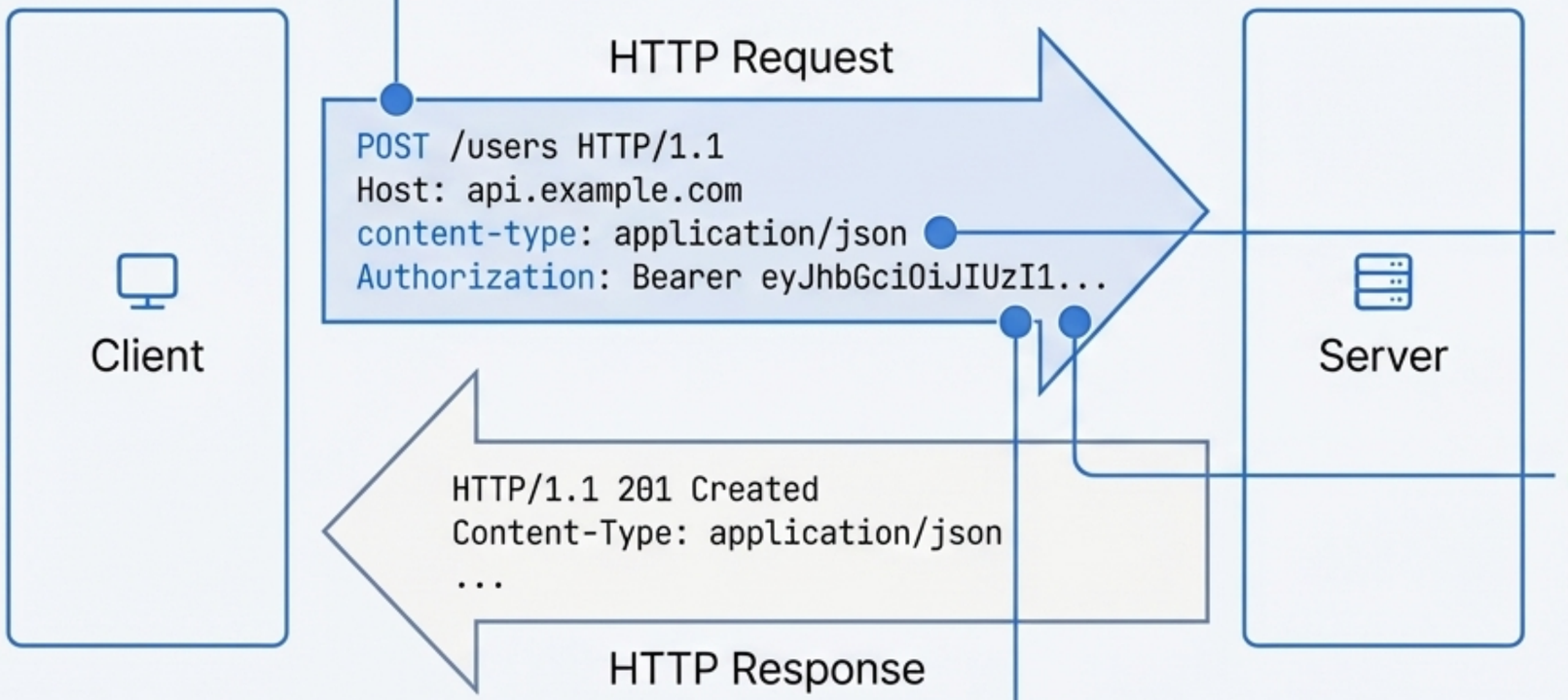
`Safe_Str__IP_Address`

Validates both IPv4 and IPv6 address formats using Python's native `ipaddress` module.

`Safe_UInt__Port`

Constrains an integer value to the valid network port range of 0 to 65535.

Speaking HTTP Fluently and Safely.



1 `Enum__Http__Method`

A type-safe `Enum` for HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, etc.), preventing typos and invalid verbs.

`Safe_Str__Http__Header__Name`

Enforces lowercase format and valid characters, aligning with HTTP/2 and HTTP/3 best practices.

`Safe_Str__Http__Content_Type`

Specifically for validating standard media type strings (e.g., `application/json`).

3 `Safe_Str__Http__Authorization`

A secure string type for authorisation tokens, filtering control characters.

Interact with Version Control with Confidence.

`Safe_Str__Git__Branch`

Validates branch names against official `git-check-ref-format` rules (e.g., cannot start with a dash, no consecutive dots).

`Safe_Str__SHA1`

Ensures a string is a full, 40-character hexadecimal Git commit hash.

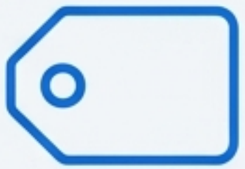
`Safe_Str__GitHub__Repo_Owner`

Validates GitHub username/organisation rules (e.g., max 39 chars, no consecutive hyphens).

`Safe_Str__GitHub__Repo`

A composite type for the full `owner/repo` identifier, validating both parts individually.

For When Uniqueness and Security are Non-Negotiable.



`Guid` / `Random_Guid`

For generating UUIDs. `Guid` is a deterministic UUIDv5 (from input), while `Random_Guid` is a random UUIDv4.

`Safe_Str__SHA1`

For full 40-character hexadecimal Git commit hashes.

`Safe_Str__Slug`

Creates clean, URL-friendly identifiers from arbitrary strings (lowercase, hyphens only).

`Safe_Str__NaCl__Public_Key`

Enforces the exact 64-character hex format (32 bytes) for a NaCl/Curve25519 public key, demonstrating deep domain knowledge.

Precise, Validated Control Over Language Models.

1. `Enum__LLM__Role`

A type-safe `Enum` for message roles: `SYSTEM`, `USER`, `ASSISTANT`, `TOOL`.

2. `Safe_Str__LLM__Model_Id`

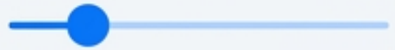
Handles complex model identifiers like `provider/model-name-v1` with specific allowed characters.

3. `Safe_Float__LLM__Temperature`

A float that is automatically clamped to the valid API range of 0.0 to 2.0.

4. `Safe_UInt__LLM__Max_Tokens`

An integer for controlling response length, constrained to the valid range of 1 to 200,000.



From Money to Megabytes, Precision Matters.

`Safe_Float__Money`

Guarantees exact 2-decimal precision using Python's `Decimal` type. Min value is 0.0. No `inf` or `nan` allowed.

`Safe_UInt__Percentage`

A simple, efficient integer constrained to the range 0-100.

`Safe_Float__Engineering`

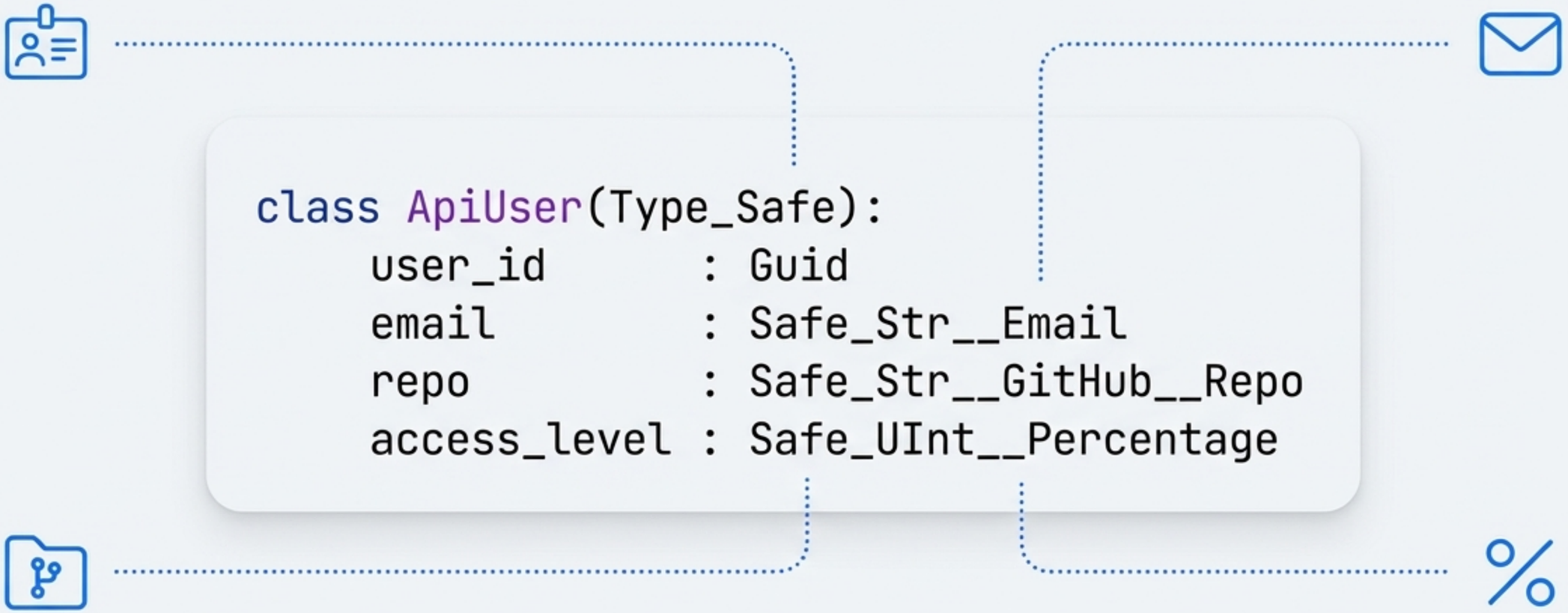
Performance-optimised float with 6-decimal precision for technical calculations where `Decimal` is too slow.

`Safe_Float__Scientific`

High-precision (15 decimals) float that explicitly allows `inf`/`nan` values, as required in scientific domains.

The Power of Composition: Building Fully Validated Objects

Primitives achieve their maximum potential when combined inside `Type_Safe` classes.



The Library is Extensible. Define Your Domain's Rules.

Don't see a primitive you need? It's trivial to create one by inheriting from a base class and defining your constraints.

```
class Safe_Str__Postal_Code(Safe_Str):
    _regex = r'^[A-Z]{1,2}[0-9R][0-9A-Z]?[0-9][A-Z]{2}$'
    _to_lower_case = False
    _max_length = 8
```

Using Primitives Effectively

1. **Be Specific:** Always prefer a domain-specific type like `Safe_Str__Url` over a generic `Safe_Str` with a custom regex.
2. **Compose:** Use primitives within `Type_Safe` classes for maximum benefit and to create fully validated data models.
3. **Handle Errors:** Primitives raise descriptive `ValueError` exceptions on failure. Use `try...except` blocks to handle invalid input gracefully.
4. **Know the Trade-offs:** For performance-critical loops, consider `use_decimal=False` on `Safe_Float` types. Precision has a (small) cost.

Build with Confidence.

OSBot-Utils Safe Primitives provide a comprehensive, performant, and extensible foundation for robust, self-documenting, and secure Python applications.

github.com/owasp-sbot/OSBot-Utils