



# Semantic Type Safety: Enriching Runtime Type Checking with Knowledge Graph Context

## Introduction: From Type Safety to Semantic Safety

In modern Python projects, **runtime type safety** frameworks help catch errors early by enforcing constraints on data types at execution time <sup>1</sup>. For example, the OSBot-Utils **Type\_Safe** library validates every operation on your data – if you try to append a raw string to a list of safe strings or assign a negative value to an unsigned integer field, it raises a **TypeError** immediately <sup>1</sup>. This goes beyond static type hints (which Python ignores at runtime) by **catching errors at assignment rather than deep in execution** <sup>1</sup>. Such robust type checking (as provided by Type\_Safe) can automatically initialize missing attributes, enforce value ranges, and use domain-specific primitives for common cases like IDs, money, URLs, etc <sup>2</sup>.

This approach has significantly improved code reliability and security. Domain-specific *safe types* wrap raw primitives to prevent common bugs: e.g. `Safe_Str__Username` allows only limited characters and length for usernames, `Safe_UInt__Age` restricts age between 0 and 150 <sup>3</sup>, and `Safe_Str__Email` ensures an email contains an "@" symbol <sup>4</sup>. By banning raw `str`, `int`, and `float` in favor of these constrained types, we eliminate entire categories of errors (SQL injection, overflows, precision loss, etc. <sup>5</sup>). In short, current runtime type safety focuses on **validating the content** of each field – making sure values conform to expected format, range, or pattern.

**However, one key element is still missing: context.** Traditional type checks know nothing about the *meaning* of the data beyond its immediate constraints. They ensure an "age" is a non-negative integer within a plausible range, but they don't know *why* that age is being collected or how it relates to other data. They can validate that a string looks like a URL, but not whether that URL actually points to a reachable or allowed resource in your system. This is where **semantic type safety** comes in – introducing knowledge of *ontology and context* into type validation.

**Semantic type safety** means categorizing and validating data according to its real-world meaning, not just its primitive form. IBM describes semantic typing as a way of categorizing data to define *how to interpret it* – for example, labeling an entity as a "Person" and knowing that subtypes like "Male", "Victim", or "Witness" are all different facets of a person in the real world <sup>6</sup>. In a similar vein, semantic type safety would ensure that data values aren't just correct in isolation, but also make sense in context (e.g. an "Adult" type implies an age  $\geq 18$ , a "CountryCode" type must be one of the known country identifiers, etc.). By leveraging semantic information – possibly from knowledge graphs or ontologies – at runtime, we can enforce **contextual validity** and even require supplementary evidence for certain values. This elevates type checking from purely syntactic or structural compliance into the realm of **meaning and truthfulness** of the data.

## Current Runtime Type Safety Recap

Before diving into semantics, let's briefly recap what the existing runtime type safety (like OSBot-Utils **Type\_Safe**) provides, and where its limits lie:

- **Continuous Runtime Checks:** Every assignment and operation is checked against the expected type. For instance, if you have `store.prices` defined as a `Dict[Safe_Str__ProductId, Safe_Float__Money]`, setting `store.prices["PROD-789"] = "not-a-number"` will immediately throw a type error – even if the code tries to auto-convert, it knows the string is not a valid money value <sup>7</sup>. This ensures type errors are caught at the exact point of violation.
- **Auto-conversion of Primitives:** The system often auto-converts basic types into their safe counterparts when possible. If a field expects a `Safe_Str` and you assign a plain Python string, the framework will attempt to wrap it into `Safe_Str` (applying any defined sanitation or validation). Only if the value truly cannot conform (e.g. wrong format) will it reject it. This makes the developer experience smoother while still maintaining safety.
- **Domain-Specific Primitive Subclasses:** Perhaps the biggest strength is the rich library of safe types tailored to domain semantics. These include:
  - **Safe strings** for various domains: e.g. `Safe_Str__Username` (alphanumeric, max length 32), `Safe_Str__Password` (enforces minimum length) <sup>4</sup>, `Safe_Str__Url` (valid URL format, length-capped) <sup>4</sup>, `Safe_Str__Email` (must contain "@") <sup>4</sup>, `Safe_Str__Html` (for HTML content with some filtering and size limits) <sup>8</sup>, etc. Each of these classes encapsulates rules specific to that kind of data.
  - **Constrained numbers:** e.g. `Safe_UInt__Age` only allows ages 0–150 <sup>3</sup>, `Safe_Float__Money` could enforce two decimal places or use Decimal under the hood to avoid floating-point issues. By using such types, a lot of invalid values (negative ages, unrealistic large ages, etc.) are immediately prevented.
  - **Enumerations and IDs:** Enumerated types for known categories (like a `Enum__Order_Status` that only accepts "pending", "shipped", etc.), or safe identifiers like `Safe_Str__OrderId` that must match a specific pattern. These again ensure that certain fields can only hold logically valid codes.

By banning raw primitives and using these safe types, the system already addresses *syntax-level and format-level correctness*. For example, a field declared as `name: Safe_Str__Username` cannot accidentally receive a full SQL query or a 10,000-character string – it will be sanitized or rejected. A user's age stored in a `Safe_UInt__Age` will never be negative or 10 million. This is a huge improvement in robustness <sup>5</sup>.

**Where it stops short is understanding *contextual meaning*.** The type system as is can tell if a value *looks* like a valid entry, but not if it *truly makes sense* in the bigger picture. Consider a few scenarios that highlight this gap:

- A field `customer_id` might be defined as `Safe_Str__CustomerId` enforcing an alphanumeric pattern for IDs. That ensures the ID string is well-formed, but it **does not guarantee that such a customer actually exists in the database or knowledge base**. The code could accept `customer_id = "CUST-9999"` as it matches the pattern, but perhaps "CUST-9999" is not a real customer in our system – a purely semantic concern.

- You might have `Safe_Str__Html` to store an HTML snippet and it will strip out obviously dangerous content. But it doesn't know if that snippet is being used in the right context. Is the string actually an HTML element of the expected type? For instance, if you expected an `<img>` tag but got a `<script>` tag (which might still pass minimal filtering), the current type won't flag that discrepancy because it doesn't understand the *intended HTML structure or context*.
- If you define `age: Safe_UInt__Age`, you know an age is in [0,150]. But imagine you have a business rule that certain operation requires an **adult** (say  $age \geq 18$ ). The type system by itself won't enforce that higher-level rule – a 15-year-old's age "15" is a valid `Safe_UInt__Age` even though it violates the concept of an adult. The notion of "*adult*" is semantic: it combines the data (age) with a concept (adulthood threshold).

These examples show how purely syntactic type checks can only go so far. They validate the **content format**, but not the **content meaning**. As data flows through more complex systems – involving knowledge graphs, user context, or interdependent fields – we need a way to validate that data in relation to an ontology or schema of the domain. This is the motivation for **semantic type safety**.

## Semantic Type Safety: Adding Ontology and Context to Validation

Semantic type safety proposes an evolution of runtime type checking: incorporate **semantic graphs (knowledge graphs or ontologies)** into the validation process. Instead of (or in addition to) checking a value against a fixed regex or range, the system would check the value *against a model of the data's meaning*. This essentially means performing **runtime semantic checks**: verifying data according to relationships and rules defined in a semantic schema.

How could this work in practice? There are a few key aspects to consider:

- **Ontologies and Knowledge Graphs:** In a semantic web or knowledge graph, entities and concepts are interrelated by facts (triples) and classes. For example, you might have an ontology that defines a class **Person** with a property **age**, and perhaps a subclass **Adult** with the constraint that  $age \geq 18$ . Or an ontology might define the class **Country** and valid country codes, or which HTML elements are allowed inside a `<div>` tag, etc. By leveraging such an ontology, our type system could **know** additional context for a value. A semantic check could query or reference the knowledge graph to validate a value's class membership or its required properties.
- **Semantic Constraints and Shapes:** The semantic web community has languages like OWL (for ontologies) and **SHACL (Shapes Constraint Language)** to impose conditions on data in graphs. For instance, SHACL could declare that *if an entity is of type `Adult`, then that entity's `ex:age` property must be  $\geq 18$* . It can also enforce patterns like "every GeoName URI must match a certain format" <sup>9</sup>. This is analogous to what we want at runtime: if a variable is declared of semantic type `AdultPerson`, then the data must include evidence (an age property) showing the age is above 18, otherwise validation fails. In fact, knowledge graph validation tools already allow rules like requiring age values to be above 18 years <sup>9</sup> – exactly capturing the notion of adulthood in data.
- **Evidence and Triples with Data:** A novel aspect of semantic type safety would be the idea of requiring **evidence** for certain values. In a traditional type check, you provide a value and the system says "okay" or "type error." In a semantic check, you might provide a value *plus some supporting facts* that help establish its validity. For example, instead of just passing an integer

`25` to a field expecting an `AdultAge`, the system might require a proof that this age corresponds to an adult – which could be as simple as a boolean flag or as complex as a reference in a knowledge graph. In practice, this might look like passing a little sub-graph along with the value: e.g. a triple `( :Person123 :age 25 )` and a triple `( :Person123 rdf:type :Adult )`. The runtime validator could then check: *is there a known assertion that Person123 is an Adult and age 25?* If yes, it accepts it; if not, it could either reject or query for more info.

This concept is akin to **contextual validation**. We are no longer just asking “Is this value of the correct primitive type and within allowed range?” but also **“Does this value make sense in context X according to our knowledge?”**. It’s somewhat analogous to how an expert system might require justification: *Don’t just tell me 42, tell me what 42 represents and how it fits our model*. In less formal terms, more evidence might be needed for the system to trust a value.

- **Runtime Knowledge Queries:** Implementing semantic checks might involve real-time lookups or reasoning. Suppose a piece of data claims to be of type `CountryCode` with value `"XZ"`. A semantic validator could consult a knowledge base of country codes (or an ontology like ISO country codes) to see if “XZ” is indeed a valid country code. If not, it flags it as invalid even if it’s a 2-letter string (which superficially might pass a regex check). Another example: if a field expects a `UserAccountActive` type (meaning an ID of an active user), the validator might actually perform a database or knowledge graph query: *Is there an entity with this ID that is active?*. If the answer is no, it doesn’t accept the value, because semantically it doesn’t fulfill the contract of being an active account.

By integrating these elements, **semantic type safety would catch errors or inconsistencies that basic type checks miss**. It moves the goalposts from “well-formed data” to “meaningful data.” We essentially begin to ask the program to *understand* a bit of what the data represents. This aligns with the trend of increasingly using metadata and ontologies in software to improve data quality and interoperability.

## Examples of Semantic Validation in Action

It might help to illustrate with concrete (hypothetical) examples how semantic type safety could operate, compared to standard runtime checks:

- **Adult Verification:** Let’s say we have a system where a user must be an adult to perform a certain action. In current practice, you might enforce this by checking `if user.age < 18: raise Exception`. With semantic type safety, you could declare the input parameter as type `AdultPerson` (a semantic type). When a `User` object or age value is passed in, the runtime will validate semantically: *is this user known to be an adult?* Perhaps the `User` object carries an ontology reference that classifies them as an `Adult` or includes a birthdate that can be evaluated. If that evidence is missing or indicates a minor, the **type check fails**. This is more powerful than a simple age check, because it could also incorporate external knowledge (e.g. checking a government ID service or an internal knowledge graph of users). It ensures that *the concept of adulthood is validated*, not just a number. In SHACL/ontology terms, it’s like having a shape that says `Person.age > 18` for `Adult` and using that at runtime to validate the object <sup>9</sup>.
- **Unit-Aware Numbers:** Consider the classic NASA **Mars Climate Orbiter** mishap, where a thruster impulse was calculated in pound-force seconds by one system but interpreted as Newton-seconds by another, leading to a \$327 million loss <sup>10</sup>. The root issue was a lack of

semantic context (units) on a numeric value – both were just “numbers” to the software. With semantic typing, the impulse could be a type like `ImpulseForce` that *carries a unit*. The runtime would then not only store the numeric value but also expect a unit attribute (e.g. “N·s” vs “lbf·s”). If a mismatch in unit is detected (or if no unit is provided), it would throw a type error. In practice, one could implement this by requiring an object or tuple (value, unit) to satisfy the `ImpulseForce` type. This way, it would be impossible to mix up units silently. The code that calls the function would have to explicitly provide the unit, and the type system could convert or check as needed. This is a form of *semantic evidence* – the unit label – that accompanies the raw number. The benefit is obvious: **no more blindly multiplied numbers that turn out to be in different scales**. Indeed, strongly-typed unit systems (seen in some languages or libraries) have been created to prevent exactly these errors <sup>10</sup>, and they can be seen as a subset of semantic type safety focusing on physical dimensions.

- **Database Entity References:** In many applications, we pass around identifiers for entities (user IDs, product IDs, order IDs). A runtime type like `Safe_Str_OrderId` might ensure the format is correct (say, “ORD-2024-001” matches a pattern <sup>11</sup>), but it doesn’t ensure that such an order actually exists or is valid. With semantic checking, one could integrate an ontology or data lookup such that when you assign `order.id = Safe_Str_OrderId("ORD-2024-001")`, the system could do a quick existence check against the orders database or a cached knowledge graph of order entities. If no order with that ID is found, it could reject the assignment as *semantically invalid*. This is analogous to a foreign key constraint in databases, but happening at the application level via the type system. Going further, the semantic graph might even know if an Order with that ID is in a state that allows certain operations – if the context expects an “OpenOrder” and the order is actually closed, that could be seen as a type violation of a semantic subtype `ClosedOrder` vs `OpenOrder`. While these checks could be done with manual code, encoding them in the type system (or a declarative ontology) means they happen **everywhere consistently** and automatically.
- **HTML and Code Validation:** Suppose you have a field that is supposed to contain an HTML snippet for a user profile bio. With just a safe string type, you might limit its length and strip disallowed tags. A semantic approach might embed an HTML DOM parser or refer to an HTML schema. The type `Safe_Str_Html` could be extended to not just allow/disallow certain characters, but to actually ensure the snippet forms valid HTML (e.g., tags properly closed, only whitelisted tags appear). Even more semantically, if the context of this HTML is known (say it will be inserted into a `<div>` on a page), the type could enforce that the snippet does not contain disallowed sections like `<html><body>` or scripts. Essentially it would “know” where this HTML is headed and what format is acceptable. As an ontology example, one could imagine a simple taxonomy of content types: *PlainText*, *SafeHTML*, *RichText*, etc., with rules for each. The runtime might require certain evidences, like an output from a sanitizer or an HTML validator library, as part of accepting the value as type `SafeHTML`. This reduces the chance of downstream errors or security issues by catching them at assignment.
- **SQL Query Semantics:** A particularly interesting case is when strings carry code or queries (common in applications that generate SQL or DSLs). A `Safe_Str_SQL_Query` type could ensure that the string is not only a valid SQL syntax but also perhaps not containing disallowed statements (e.g. no `DROP TABLE` if it’s only supposed to be a `SELECT`). A semantic check might involve parsing the SQL (turning it into an AST) and then validating that AST against a model of allowed queries. It could also cross-check the query against the database schema: e.g., verify that the tables and columns referenced actually exist and the user has access to them. This crosses into static analysis territory, but doing it at runtime (when queries are constructed) can prevent dangerous or simply broken SQL from ever executing. The “evidence” here could be the

parsed structure or a confirmation from the DB. In effect, the type system acts as a gatekeeper that asks “*Is this query semantically okay to run?*” beyond just “is it a string?”.

These examples demonstrate how combining the **power of a semantic web/graph** with runtime checks could catch a variety of issues: nonexistent references, logically inconsistent states, policy violations, etc. The program not only checks the data's face value but also asks the world (or its internal world model) if the data fits.

## Benefits of Semantic Data Validation

Adopting semantic type safety can yield multiple benefits for software robustness and data quality:

- **Enhanced Data Quality and Consistency:** By encoding real-world rules in the type system, we ensure that data entering the system is not just *well-formed* but also *meaningful*. Errors that would only show up as bugs or corrupt data later (like an invalid ID, or a logically impossible value combination) can be caught at the boundaries. This is similar to how knowledge graphs use ontologies to maintain consistency – for example, an ontology could prevent an individual from being classified as both a Minor and an Adult, or a medical record from having a pregnant male patient. With semantic checks, such contradictions can be automatically flagged. In fact, **ontology-driven validation ensures that data or answers are consistent with expert-approved knowledge**, blocking those that violate the domain rules <sup>12</sup>.
- **Security and Safety Improvements:** Many security issues revolve around data that is syntactically valid but semantically malicious. For instance, an SQL injection attack is basically a string that is valid SQL (so type `str` or even a naive `Safe_Str_SQL` might accept it if it matches expected pattern), but semantically it does something unwanted. A semantic validator could detect that, say, the query is trying to drop a table or always returns true (tautology in a WHERE clause) and reject it. Similarly for HTML/JS injection – a string with `<script>` might technically be valid HTML, but semantically it's dangerous in a context expecting plain text or limited markup. By understanding context, the validation can be more fine-grained. This reduces reliance on just blacklists or regex and moves toward understanding the *intent* of the content relative to its use. Essentially, semantic validation can act as an intelligent whitelist approach: only allowing data that fits known-good patterns defined by the ontology of the application.
- **Discovering Unknown Unknowns:** When we integrate a semantic knowledge graph, the system can leverage relationships that single fields alone wouldn't know. For example, if two fields are semantically related (like a region and a country), a semantic check can ensure coherence (the region is indeed within the specified country). Traditional type checks treat each field in isolation, whereas a semantic model can correlate them. This means the system might catch errors like mismatched combinations (e.g., a shipping address ZIP code that doesn't belong to the stated city) by cross-checking with a geographic knowledge base. It elevates validation to a multi-dimensional level.
- **Self-Documentation and Communication:** Having semantic types also serves as live documentation of data requirements. Instead of burying in comments or docs that “this field must refer to an existing user” or “age must be adult if role is X,” those constraints become part of the type definition. Developers reading the code see `AdultPerson` or `ActiveAccountId` as types and immediately understand the intent, and they cannot easily ignore those rules because the code literally won't run if they break them. This clarity helps especially in large systems or APIs – clients know what is expected beyond just basic type.

- **Bridging to the Semantic Web and Interoperability:** If your application already consumes or produces data linked to an external ontology (for instance, schema.org types, healthcare standards, etc.), semantic type safety could make integration smoother. Your internal types could directly map to ontology classes, and validation ensures any data you produce is compliant with those classes' definitions. Conversely, incoming data can be validated against the reference ontology automatically. This can improve interoperability between systems – since many modern systems are adopting knowledge graphs for data exchange, having your app speak in ontology-backed types means fewer translation errors and mismatches. As a bonus, it might allow more automation, such as generic form generators or UI validators that read the ontology and know what to enforce (units, ranges, categories, etc.).

## Challenges and Considerations

While the idea of semantic type safety is powerful, implementing it is non-trivial. Here are some challenges and points to consider:

- **Performance Overhead:** Runtime semantic checks could be more expensive than simple type checks. A normal type check might be an `isinstance` or a regex match – very fast. A semantic check might involve a database lookup, a SPARQL query to a triplestore, or invoking a reasoning engine. Doing this on every assignment or function call could slow things down. Careful caching or lazy validation might be needed. Perhaps not every value needs to be fully verified via knowledge graph every time – one could cache that "CUST-123" was verified as a valid customer earlier in the request, etc. Alternatively, semantic checks could be reserved for system boundaries (like on API input) rather than every internal assignment, to mitigate cost.
- **Availability of Knowledge:** Semantic validation is only as good as the knowledge base behind it. You need an ontology or graph of your domain that is complete and up-to-date enough to use for validation. In some cases, this is readily available (e.g., a list of all countries is static knowledge; an ontology of medical terms exists in standards like SNOMED). In other cases, you'd have to build and maintain the knowledge graph. For dynamic data (like list of active users), the "knowledge graph" might just be your production database or an API call. Ensuring the validator has access to the latest truth of the system is essential, otherwise it might falsely reject valid data because its cached knowledge is stale. This introduces architectural complexity (synchronizing data to a graph or making on-the-fly queries).
- **Graceful Handling of Uncertainty:** There may be cases where the semantic information is incomplete. For example, you pass a location name "Springfield" to a semantic type `CityName`. If your knowledge graph has multiple Springfields (in different states or countries), is that considered valid or does it need disambiguation? The type checker might need a way to handle uncertain or ambiguous semantic matches – possibly by requesting more evidence (e.g., also provide a state or country to clarify which Springfield). This touches on an interactive aspect: the system could sometimes respond with "I need more info to validate this." In a user-facing scenario, that could trigger a prompt like "Did you mean Springfield, Illinois or Springfield, Massachusetts?". In an automated pipeline, it might log an error or choose a default. Designing how the type system interacts when knowledge is fuzzy is an interesting challenge.
- **Development Effort and Complexity:** Introducing semantic types means developers now must think about ontologies and knowledge graphs to some extent. Defining a new semantic type would likely involve specifying how to validate it (e.g., a SPARQL query or a Python callback that checks a knowledge source). This is more involved than defining a regex for a safe string. It

might require new tooling or frameworks to make it convenient – perhaps an extension of the Type\_Safe library that can register ontology-driven validators. There is also the question of how deeply to integrate this: do we integrate with RDF and OWL directly, or use simpler mappings (like a Python dict of allowed values, or a network call)? Early implementations could hardcode logic (like a `Safe_Str__CountryCode` that checks against a list of codes), but a more general approach might read from an ontology file or endpoint.

- **Error Handling and Security:** If semantic validation fails (meaning the data doesn't make sense), how do we report that? The error messages become trickier: instead of "TypeError: expected int got str", we'd have errors like "SemanticTypeError: Unknown Customer ID – no record found in knowledge base" or "ConstraintError: Provided age does not satisfy Adult ( $\geq 18$ ) requirement". These are very useful messages, but we should be careful about information leakage. For example, if an external user is hitting an API and gets "Unknown Customer ID", that might be okay; but an error like "User exists but is not active" might reveal internal status. As with any validation, we'd sanitize what goes out. Also, hooking into external sources (like a knowledge service) means validation could fail not just due to bad data but due to system issues (e.g., the ontology service is down). We then need fallback behaviors – possibly default to a softer failure or a cached check.
- **Maintaining Ontologies and Rules:** Once you start relying on an ontology for validation, changes to that ontology can affect your application's behavior. This is similar to how changing a schema in a database demands updating application code. Here, if the definition of `Adult` changes or you add a new subclass, it might impact the type logic. Proper versioning and testing around the semantic rules would be necessary. In some domains, ontologies can get quite complex, and modeling every needed rule might be an ongoing project (though a rewarding one in terms of data governance).
- **Who Else Is Doing This?** As we consider implementing semantic type safety, it's worth noting that we are charting somewhat new territory at the intersection of programming languages and semantic web. However, there are related efforts we can draw inspiration from:
  - **Refinement Types and Dependent Types:** In academia and some functional languages, types can include predicates (e.g., a type `AgeOver18` that is an `int` with the condition `x >= 18`). Liquid Haskell and Dafny, for example, allow specifications that certain values must satisfy logical conditions. Those are typically checked at compile-time or verified via theorem provers, but they show the appetite for embedding more semantics in types. Our approach is more runtime and data-driven (using external knowledge), but conceptually it's a sibling idea.
  - **Knowledge Graph Integrity Checks:** The semantic web community uses SHACL and OWL constraints to ensure data integrity in graphs. For instance, a SHACL shape might declare a constraint that a Person's age must be  $\geq 0$  and  $\leq 150$  (which sounds just like our `Safe_UInt_Age`) or that if a Person has a spouse, that spouse is of type Person as well. These are typically applied when data is added to a triple store. By bringing similar logic to our application's core, we're effectively doing a SHACL-like validation on in-memory objects. This could tie in nicely if our app is both reading and writing to a central knowledge graph – the same shapes used for offline validation could potentially be executed at runtime as well <sup>9</sup>.
  - **Data Quality in Industry:** In fields like healthcare and finance, ontologies and rule engines are used to catch data errors. For example, an ontology in healthcare might encode that certain combinations of patient attributes are inconsistent (male with a pregnancy code, as mentioned earlier). These are often checked in data cleaning stages or interactive forms. By making it part of the type system, we could achieve continuous enforcement. The idea of "ontology-driven validation" is gaining traction – even in AI, researchers have used domain ontologies to post-

validate AI-generated answers for consistency <sup>12</sup>. This shows the general principle of *checking facts against a knowledge base* to prevent nonsense or errors.

- **Semantic Data Platforms:** Some modern data integration tools attempt to automatically detect semantic types of columns in datasets (e.g., distinguishing a “City Name” field from just any string, using libraries like Sherlock or machine learning <sup>13</sup>). They then use that to apply appropriate validation or transformation (for instance, recognizing a column as “Date” and parsing it). This is a more heuristic approach compared to our explicit ontology method, but it’s adjacent – it’s about understanding what data *represents* and ensuring it fits that representation. One could imagine a future where an AI suggests semantic types for your variables, and then your type system enforces them.

In summary, while some pieces of semantic type safety exist in various forms, combining them into a coherent runtime framework in a language like Python would be a fresh and exciting development. It aligns with the increasing need for **data integrity, security, and smart automation** in complex systems.

## Conclusion

The concept of **semantic data validation** or **semantic type safety** represents a natural next step in making our software **safer and smarter**. We’ve already seen huge benefits from strict runtime type checking – catching bugs early, preventing invalid states, and documenting assumptions. By infusing that layer with semantic knowledge, we elevate the guarantees we can provide. Instead of simply “an integer in range” or “a string of form X”, we can assert “a valid user of our system”, “a safe HTML snippet for our page”, or “an age consistent with an adult”. These are checks that developers often implement manually (inconsistently and sometimes incorrectly) throughout codebases; with semantic type safety, they become a standardized, transparent part of the infrastructure.

Certainly, realizing this vision will require research and experimentation. We’ll need to explore how to connect fast in-memory operations with heavier semantic queries efficiently, how to represent and pass “evidence” alongside values in a developer-friendly way, and how to scale the approach in distributed systems. We should investigate existing semantic web standards and see how they can interface with our Python type system. Perhaps we can borrow the idea of *shapes* or *constraints* and create a Pythonic API for them.

The good news is that the building blocks are falling into place. With projects like OSBot-Utils Type\_Safe, we already have a strong runtime type framework to build upon. Knowledge graphs and ontologies are more accessible than ever, with many companies adopting them for master data management and AI. There’s also a growing awareness that **data quality** is as important as functionality – and data quality is exactly what semantic validation ensures <sup>14</sup> <sup>15</sup>. By ensuring what the user intends to record is what actually gets recorded (in terms of meaning), we improve not just our code’s correctness but the truthfulness of our systems.

In conclusion, **semantic type safety** can be seen as uniting two historically separate layers: the program’s type system and the world’s knowledge system. It means when we say a variable is of a certain type, we’re also saying something about the real-world entity it represents, and we’re not willing to accept data that doesn’t match that reality. It’s a bold idea, but one that could lead to more resilient software. As we continue to push into areas like semantic knowledge graphs, function flow graphs, and beyond, this approach could become a key feature of next-generation programming frameworks. It’s an exciting area for further research and development – and perhaps the next big leap in how we think about “type safety” in an increasingly data-rich, context-aware world.

## Sources:

- IBM definition of semantic typing [6](#)
  - OSBot-Utils Type\_Safe documentation – runtime type checking and domain-specific primitives [16](#) [5](#)
  - OSBot-Utils Type\_Safe examples – Safe\_UInt\_Age constraint [3](#), Safe string domain examples [4](#) [8](#)
  - Mars Climate Orbiter unit mismatch story (importance of semantic units) [10](#)
  - W3C SHACL (Shapes Constraint Language) example – enforcing age > 18 via shapes [9](#)
  - Ontology-driven validation in cybersecurity (using domain ontologies to ensure consistency and safety) [12](#)
- 

[1](#) [2](#) [3](#) [4](#) [5](#) [7](#) [8](#) [11](#) [16](#) v3.1.1\_for\_llms\_type\_safe\_and\_python-formatting-guide.md  
file:///file\_00000000f98c71f49b47c0ec8e4bd4d1

[6](#) [13](#) Semantic data types. Systematic approach and types registry | by Ivan Begtin | Medium  
<https://medium.com/@ibegin/semantic-data-types-systematic-approach-and-types-registry-a2c2a60a467b>

[9](#) SHACL validation — GraphDB 11.2 documentation  
<https://graphdb.ontotext.com/documentation/11.2/shacl-validation.html>

[10](#) Type safety, units and how (not) to crash the Mars Climate Orbiter | Lune Engineering Blog  
<https://eng.lune.co/posts/2021/09/03/type-safety-units/>

[12](#) CyberRAG in Cybersecurity  
<https://www.emergentmind.com/topics/cyberrag>

[14](#) [15](#) Improving the Quality and Utility of Electronic Health Record Data through Ontologies | MDPI  
<https://www.mdpi.com/2305-6703/3/3/23>