


**SciPy 2024***July 8 - July 14, 2024*

Proceedings of the 23<sup>rd</sup>  
Python in Science Conference  
ISSN: 2575-9752

# Improving Code Quality with Array and DataFrame Type Hints

**Christopher Ariza**<sup>1</sup>  <sup>1</sup>Research Affiliates

---

## Abstract

This article demonstrates practical approaches to fully type-hinting generic NumPy arrays and StaticFrame DataFrames, and shows how the same annotations can improve code quality with both static analysis and runtime validation.

---

**Keywords** typing, static analysis, runtime validation, variadic generics

As tools for Python type annotations (or hints) have evolved, more complex data structures can be typed, improving maintainability and static analysis. Arrays and DataFrames, as complex containers, have only recently supported complete type annotations in Python. NumPy [1] 1.22 introduced generic specification of arrays and dtypes. Building on NumPy's foundation, StaticFrame [2] 2.0 introduced complete type specification of DataFrames, employing NumPy primitives and variadic generics. This article demonstrates practical approaches to fully type-hinting arrays and DataFrames, and shows how the same annotations can improve code quality with both static analysis and runtime validation.

## 1. TYPE HINTS IMPROVE CODE QUALITY

Type hints [3] improve code quality in a number of ways. Instead of using variable names or comments to communicate types, Python-object-based type annotations provide maintainable and expressive tools for type specification. These type annotations can be tested with type checkers such as `mypy` [4] or `pyright` [5], quickly discovering potential bugs without executing code.

The same annotations can be used for runtime validation. While reliance on duck-typing over runtime validation is common in Python, runtime validation is more often needed with complex data structures such as arrays and DataFrames. For example, an interface expecting a DataFrame argument, if given a Series, might not need explicit validation as usage of the wrong type will likely raise. However, an interface expecting a 2D array of floats, if given an array of Booleans, might benefit from validation as usage of the wrong type may not raise.

Many important typing utilities are only available with the most-recent versions of Python. Fortunately, the `typing-extensions` [6] package back-ports standard library utilities for older versions of Python. A related challenge is that type checkers can take time to implement full support for new features: many of the examples shown here require `mypy` 1.9.0, released just a few months ago.

## 2. ELEMENTAL TYPE ANNOTATIONS

Without type annotations, a Python function signature gives no indication of the expected types. For example, the function below might take and return any types:

**Published** Jul 10, 2024

**Correspondence to**  
Christopher Ariza  
[ariza@flexatone.com](mailto:ariza@flexatone.com)

**Open Access** 

Copyright © 2024 Ariza. This is an open-access article distributed under the terms of the [Creative Commons Attribution 4.0 International](https://creativecommons.org/licenses/by/4.0/) license, which enables reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator.

```
def process0(v, q): ... # no type information
```

By adding type annotations, the signature informs readers of the expected types. With modern Python, user-defined and built-in classes can be used to specify types, with additional resources (such as `Any`, `Iterator`, `cast()`, and `Annotated`) found in the standard library typing module. For example, the interface below improves the one above by making expected types explicit:

```
def process0(v: int, q: bool) -> list[float]: ...
```

When used with a type checker like `mypy`, code that violates the specifications of the type annotations will raise an error during static analysis (shown as comments, below). For example, providing an integer when a Boolean is required is an error:

```
x = process0(v=5, q=20)
# tp.py: error: Argument "q" to "process0"
# has incompatible type "int"; expected "bool" [arg-type]
```

Static analysis can only validate statically defined types. The full range of runtime inputs and outputs is often more diverse, suggesting some form of runtime validation. The best of both worlds is possible by reusing type annotations for runtime validation. While there are libraries that do this (e.g., `typeguard` and `beartype`), `StaticFrame` offers `CallGuard`, a tool specialized for comprehensive array and DataFrame type-annotation validation.

A Python decorator is ideal for leveraging annotations for runtime validation. `CallGuard` offers two decorators: `@CallGuard.check`, which raises an informative `Exception` on error, or `@CallGuard.warn`, which issues a warning.

Further extending the `process0` function above with `@CallGuard.check`, the same type annotations can be used to raise an `Exception` (shown again as comments) when runtime objects violate the requirements of the type annotations:

```
import static_frame as sf

@sf.CallGuard.check
def process0(v: int, q: bool) -> list[float]:
    return [x * (0.5 if q else 0.25) for x in range(v)]

z = process0(v=5, q=20)
# static_frame.core.type_clinic.ClinicError:
# In args of (v: int, q: bool) -> list[float]
# └─ Expected bool, provided int invalid
```

While type annotations must be valid Python, they are irrelevant at runtime and can be wrong: it is possible to have correctly verified types that do not reflect runtime reality. As shown above, reusing type annotations for runtime checks ensures annotations are valid.

### 3. ARRAY TYPE ANNOTATIONS

Python classes that permit component type specification are “generic”. Component types are specified with positional “type variables”. A list of integers, for example, is annotated with `list[int]`; a dictionary of floats keyed by tuples of integers and strings is annotated `dict[tuple[int, str], float]`.

With NumPy 1.20, `ndarray` and `dtype` become generic. The generic `ndarray` requires two arguments, a shape and a `dtype`. As the usage of the first argument is still under development, `Any` is commonly used. The second argument, `dtype`, is itself a generic that requires a type variable for a NumPy type such as `np.int64`. NumPy also offers more general generic types such as `np.integer[Any]`.

For example, an array of Booleans is annotated `np.ndarray[Any, np.dtype[np.bool_]]`; an array of any type of integer is annotated `np.ndarray[Any, np.dtype[np.integer[Any]]]`.

As generic annotations with component type specifications can become verbose, it is practical to store them as type aliases (here prefixed with “T”). The following function specifies such aliases and then uses them in a function.

```
from typing import Any
import numpy as np

TNDArrayInt8 = np.ndarray[Any, np.dtype[np.int8]]
TNDArrayBool = np.ndarray[Any, np.dtype[np.bool_]]
TNDArrayFloat64 = np.ndarray[Any, np.dtype[np.float64]]

def process1(
    v: TNDArrayInt8,
    q: TNDArrayBool,
) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s
```

As before, when used with `mypy`, code that violates the type annotations will raise an error during static analysis. For example, providing an integer when a Boolean is required is an error:

```
v1: TNDArrayInt8 = np.arange(20, dtype=np.int8)
x = process1(v1, v1)
# tp.py: error: Argument 2 to "process1" has incompatible type
# "ndarray[Any, dtype[float64]]"; expected "ndarray[Any, dtype[bool_]]" [arg-type]
```

The interface requires 8-bit signed integers (`np.int8`); attempting to use a different sized integer is also an error:

```
TNDArrayInt64 = np.ndarray[Any, np.dtype[np.int64]]
v2: TNDArrayInt64 = np.arange(20, dtype=np.int64)
q: TNDArrayBool = np.arange(20) % 3 == 0
x = process1(v2, q)
# tp.py: error: Argument 1 to "process1" has incompatible type
# "ndarray[Any, dtype[signedinteger[_64Bit]]]"; expected "ndarray[Any,
dtype[signedinteger[_8Bit]]" [arg-type]
```

While some interfaces might benefit from such narrow numeric type specifications, broader specification is possible with NumPy’s generic types such as `np.integer[Any]`, `np.signedinteger[Any]`, `np.float[Any]`, etc. For example, we can define a new function that accepts any size signed integer. Static analysis now passes with both `TNDArrayInt8` and `TNDArrayInt64` arrays.

```
TNDArrayIntAny = np.ndarray[Any, np.dtype[np.signedinteger[Any]]]
def process2(
    v: TNDArrayIntAny, # a more flexible interface
    q: TNDArrayBool,
) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process2(v1, q) # no mypy error
x = process2(v2, q) # no mypy error
```

Just as shown above with elements, generically specified NumPy arrays can be validated at runtime if decorated with `CallGuard.check`:

```
@sf.CallGuard.check
def process3(v: TNDArrayIntAny, q: TNDArrayBool) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process3(v1, q) # no error, same as mypy
x = process3(v2, q) # no error, same as mypy
v3: TNDArrayFloat64 = np.arange(20, dtype=np.float64) * 0.5
x = process3(v3, q) # error
# static_frame.core.type_clinic.ClinicError:
# In args of (v: ndarray[Any, dtype[signedinteger[Any]]],
#   ndarray[Any, dtype[bool_]] -> ndarray[Any, dtype[float64]]
#   └─ ndarray[Any, dtype[signedinteger[Any]]]
#       └─ dtype[signedinteger[Any]]
#           └─ Expected signedinteger, provided float64 invalid
```

StaticFrame provides utilities to extend runtime validation beyond type checking. Using the typing module's Annotated class [7], we can extend the type specification with one or more StaticFrame Require objects. For example, to validate that an array has a 1D shape of (24,), we can replace `TNDArrayIntAny` with `Annotated[TNDArrayIntAny, sf.Require.Shape(24)]`. To validate that a float array has no NaNs, we can replace `TNDArrayFloat64` with `Annotated[TNDArrayFloat64, sf.Require.Apply(lambda a: ~a.insna().any())]`.

Implementing a new function, we can require that all input and output arrays have the shape (24,). Calling this function with the previously created arrays raises an error:

```
from typing import Annotated

@sfc.CallGuard.check
def process4(
    v: Annotated[TNDArrayIntAny, sf.Require.Shape(24)],
    q: Annotated[TNDArrayBool, sf.Require.Shape(24)],
) -> Annotated[TNDArrayFloat64, sf.Require.Shape(24)]:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process4(v1, q) # types pass, but Require.Shape fails
# static_frame.core.type_clinic.ClinicError:
# In args of (v: Annotated[ndarray[Any, dtype[int8]], Shape((24,))], q: Annotated[ndarray[Any,
#   dtype[bool_]], Shape((24,))]) -> Annotated[ndarray[Any, dtype[float64]], Shape((24,))]
#   └─ Annotated[ndarray[Any, dtype[int8]], Shape((24,))]
#       └─ Shape((24,))
#           └─ Expected shape ((24,)), provided shape (20,)
```

## 4. DATAFRAME TYPE ANNOTATIONS

Just like a dictionary, a DataFrame is a complex data structure composed of many component types: the index labels, column labels, and the column values are all distinct types.

A challenge of generically specifying a DataFrame is that a DataFrame has a variable number of columns, where each column might be a different type. The Python `TypeVarTuple` variadic generic specifier [8], first released in Python 3.11, permits defining a variable number of column type variables.

With `StaticFrame` 2.0, `Frame`, `Series`, `Index` and related containers become generic. Support for variable column type definitions is provided by `TypeVarTuple`, back-ported with the implementation in `typing-extensions` for compatibility down to Python 3.9.

A generic `Frame` requires two or more type variables: the type of the index, the type of the columns, and zero or more specifications of columnar value types specified with NumPy types. A generic `Series` requires two type variables: the type of the index and a NumPy type for the values. The `Index` is itself generic, also requiring a NumPy type as a type variable.

With generic specification, a `Series` of floats, indexed by dates, can be annotated with `sf.Series[sf.IndexDate, np.float64]`. A `Frame` with dates as index labels, strings as column labels, and column values of integers and floats can be annotated with `sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.float64]`.

Given a complex `Frame`, deriving the annotation might be difficult. `StaticFrame` offers the `via_type_clinic` interface to provide a complete generic specification for any component at runtime:

```
>>> v4 = sf.Frame.from_fields([range(5), np.arange(3, 8) * 0.5],
                             columns=('a', 'b'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))
>>> v4
<Frame>
<Index>      a      b      <<U1>
<IndexDate>
2021-12-30    0      1.5
2021-12-31    1      2.0
2022-01-01    2      2.5
2022-01-02    3      3.0
2022-01-03    4      3.5
<datetime64[D]> <int64> <float64>

# get a string representation of the annotation
>>> v4.via_type_clinic
Frame[IndexDate, Index[str_], int64, float64]
```

As shown with arrays, storing annotations as type aliases permits reuse and more concise function signatures. Below, a new function is defined with generic `Frame` and `Series` arguments fully annotated. A cast is required as not all operations can statically resolve their return type.

```
TFrameDateInts = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.int64]
TSeriesYMBool = sf.Series[sf.IndexYearMonth, np.bool_]
TSeriesDFloat = sf.Series[sf.IndexDate, np.float64]

def process5(v: TFrameDateInts, q: TSeriesYMBool) -> TSeriesDFloat:
    t = v.index.iter_label().apply(lambda l: q[l.astype('datetime64[M]')]) # type: ignore
    s = np.where(t, 0.5, 0.25)
    return cast(TSeriesDFloat, (v.via_T * s).mean(axis=1))
```

These more complex annotated interfaces can also be validated with `mypy`. Below, a `Frame` without the expected column value types is passed, causing `mypy` to error (shown as comments, below).

```

TFrameDateIntFloat = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.float64]
v5: TFrameDateIntFloat = sf.Frame.from_fields([range(5), np.arange(3, 8) * 0.5],
columns=('a', 'b'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))

q: TSeriesYMBBool = sf.Series([True, False],
index=sf.IndexYearMonth.from_date_range('2021-12', '2022-01'))

x = process5(v5, q)
# tp.py: error: Argument 1 to "process5" has incompatible type
# "Frame[IndexDate, Index[str_], signedinteger[_64Bit], floating[_64Bit]]"; expected
# "Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]]" [arg-type]

```

To use the same type hints for runtime validation, the `sf.CallGuard.check` decorator can be applied. Below, a `Frame` of three integer columns is provided where a `Frame` of two columns is expected.

```

# a Frame of three columns of integers
TFrameDateIntIntInt = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.int64, np.int64]
v6: TFrameDateIntIntInt = sf.Frame.from_fields([range(5), range(3, 8), range(1, 6)],
columns=('a', 'b', 'c'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))

x = process5(v6, q)
# static_frame.core.type_clinic.ClinicError:
# In args of (v: Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]],
# q: Series[IndexYearMonth, bool_] -> Series[IndexDate, float64]
# └─ Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]]
#   └─ Expected Frame has 2 dtype, provided Frame has 3 dtype

```

It might not be practical to annotate every column of every `Frame`: it is common for interfaces to work with `Frame` of variable column sizes. `TypeVarTuple` supports this through the usage of unpack operator `*tuple[]` expressions (introduced in Python 3.11, back-ported with the `Unpack` annotation). For example, the function above could be defined to take any number of integer columns with that annotation `Frame[IndexDate, Index[np.str_], *tuple[np.int64, ...]]`, where `*tuple[np.int64, ...]` means zero or more integer columns.

The same implementation can be annotated with a far more general specification of columnar types. Below, the column values are annotated with `np.number[Any]` (permitting any type of numeric NumPy type) and a `*tuple[]` expression (permitting any number of columns): `*tuple[np.number[Any], ...]`. Now neither `mypy` nor `CallGuard` errors with either previously created `Frame`.

```

TFrameDateNums = sf.Frame[sf.IndexDate, sf.Index[np.str_], *tuple[np.number[Any], ...]]

@sf.CallGuard.check
def process6(v: TFrameDateNums, q: TSeriesYMBBool) -> TSeriesDFloat:
    t = v.index.iter_label().apply(lambda l: q[l.astype('datetime64[M]')]) # type: ignore
    s = np.where(t, 0.5, 0.25)
    return tp.cast(TSeriesDFloat, (v.via_T * s).mean(axis=1))

x = process6(v5, q) # a Frame with integer, float columns passes
x = process6(v6, q) # a Frame with three integer columns passes

```

As with NumPy arrays, `Frame` annotations can wrap `Require` specifications in `Annotated` generics, permitting the definition of additional run-time validations.

## 5. TYPE ANNOTATIONS WITH OTHER LIBRARIES

While `StaticFrame` might be the first `DataFrame` library to offer complete generic specification and a unified solution for both static type analysis and run-time type validation, other array and `DataFrame` libraries offer related utilities.

Neither the `Tensor` class in PyTorch (2.4.0), nor the `Tensor` class in TensorFlow (2.17.0) support generic type or shape specification. While both libraries offer a `TensorSpec` object that can be used to perform run-time type and shape validation, static type checking with tools like `mypy` is not supported.

As of Pandas 2.2.2, neither the `Pandas Series` nor `DataFrame` support generic type specifications. A number of third-party packages have offered partial solutions. The `pandas-stubs` library, for example, provides type annotations for the Pandas API, but does not make the `Series` or `DataFrame` classes generic. The `pandera` library [9] permits defining `DataFrameSchema` classes that can be used for run-time validation of Pandas `DataFrames`. For static-analysis with `mypy`, `pandera` offers alternative `DataFrame` and `Series` subclasses that permit generic specification with the same `DataFrameSchema` classes. This approach does not permit the expressive opportunities of using generic NumPy types or the unpack operator for supplying variadic generic expressions.

## 6. CONCLUSION

Python type annotations can make static analysis of types a valuable check of code quality, discovering errors before code is even executed. Up until recently, an interface might take an array or a `DataFrame`, but no specification of the types contained in those containers was possible. Now, complete specification of component types is possible in NumPy and `StaticFrame`, permitting more powerful static analysis of types.

Providing correct type annotations is an investment. Reusing those annotations for runtime checks provides the best of both worlds. `StaticFrame`'s `CallGuard` runtime type checker is specialized to correctly evaluate fully specified generic NumPy types, as well as all generic `StaticFrame` containers.

## REFERENCES

- [1] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: <https://doi.org/10.1038/s41586-020-2649-2>.
- [2] `StaticFrame` Development Team, “`StaticFrame`.” [Online]. Available: <https://github.com/static-frame/static-frame>
- [3] G. van Rossum, J. Lehtosalo, and Ł. Langa, “PEP 484 - Type Hints,” 2014. [Online]. Available: <https://www.python.org/dev/peps/pep-0484/>
- [4] `mypy` Development Team, “`mypy`.” [Online]. Available: <https://mypy.readthedocs.io/en/stable/>
- [5] `pyright` Development Team, “`pyright`.” [Online]. Available: <https://github.com/microsoft/pyright/>
- [6] `typing_extensions` Development Team, “`typing_extensions`.” [Online]. Available: <https://typing-extensions.readthedocs.io/>
- [7] R. Gonzalez, P. House, I. Levkivskyi, L. Roach, and G. van Rossum, “PEP 593 - Flexible Function and Variable Annotations,” 2019. [Online]. Available: <https://www.python.org/dev/peps/pep-0593/>
- [8] M. Mendoza, M. Rahtz, P. K. Srinivasan, and V. Siles, “PEP 646 - Variadic Generics,” 2020. [Online]. Available: <https://www.python.org/dev/peps/pep-0646/>
- [9] N. Bantilan, “`pandera`: Statistical Data Validation of Pandas Dataframes,” in *Proceedings of the 19th Python in Science Conference*, M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, Eds., 2020, pp. 116–124. doi: <https://doi.org/10.25080/Majora-342d178e-010>.