Runtype Documentation

Erez Shinan

Apr 22, 2024

MODULES

CHAPTER

ONE

VALIDATION (ISA & ISSUBCLASS)

This module provides type validation functions for Python with support for the `typing` module.

It uses the same internal mechanism as used by 'dataclass' and 'dispatch' in order to resolve and validate types and values.

You may use them to replace `isinstance()` and `issubclass`. See *benchmarks*.

1.1 Functions

`runtype.validation.`**`isa`**(*obj*, *t*)

Tests if 'obj' is of type 't'

Behaves like Python's isinstance, but supports the `typing` module and constraints.

`runtype.validation.`**`ensure_isa`**(*obj*, *t*, *sampler=None*)

Ensure 'obj' is of type 't'. Otherwise, throws a TypeError

`runtype.validation.`**`assert_isa`**(*obj*, *t*)

Ensure 'obj' is of type 't'. Otherwise, throws a TypeError

Does nothing if Python is run with -O. (like the assert statement)

`runtype.validation.`**`issubclass`**(*t1*, *t2*)

Test if t1 is a subclass of t2

Parameters

- **type** (*t1 - a*) –
- **types** (*t2 - a type or a tuple of*) –

Behaves like Python's issubclass, but supports the `typing` module.

`runtype.validation.`**`is_subtype`**(*t1*, *t2*)

Test if t1 is a subtype of t2

1.2 Element-wise validation

When called on generics such as *List*, *Tuple*, *Set* and *Dict*, runtype will iterate over each element and call *ensure_isa()* recursively.

Example:

```
>>> isa([1,2], List[int])
True

>>> isa([1,"a"], List[int])
False

>>> isa([{1: 2}], List[Dict[int, int]])
True

>>> isa([{1: 2}], List[Dict[int, str]])
False
```

1.3 How does it work?

Runtype maps the given types onto an internal type system, that is capable of expressing the Python type system.

In order to validate a value against a type, we do the following:

1. Convert (cast) the given type into an instance of *PythonType*, that represents the given type within the internal type system. This operation is cached.
2. Call the *PythonType.validate_instance()* method with the given value. Each subclass has its own implementation. For example:
    - In *PythonDataType* (e.g. *Int* or *DateTime*), the method will simply call Python's *isinstance()* on the value.
    - In *SequenceType* (e.g. *List* or *Iter*), after *isinstance()*, this method will call itself recursively for each item.

The internal type system is implemented using the *Type Classes* module.

CHAPTER

TWO

DATACLASS

2.1 Decorator

runtype.dataclass.**dataclass**(*, *check_types: Union[bool, str] = CHECK_TYPES*, *config:* Configuration = *python_config*, *init: bool = True*, *repr: bool = True*, *eq: bool = True*, *order: bool = False*, *unsafe_hash: bool = False*, *frozen: bool = True*, *slots: bool = Ellipsis*) → Callable[[Type[_T]], Type[_T]]

runtype.dataclass.**dataclass**(*_cls: Type[_T]*) → Type[_T]

Runtype's dataclass is a drop-in replacement to Python's built-in dataclass, with added functionality.

Differences from builtin dataclass:

1. Type validation

- Adds run-time type validation (when check_types is nonzero)
- Performs automatic casting (when check_types == 'cast')

2. Ergonomics

- Supports assigning mutable literals (i.e. list, set, and dict). Each instance gets a new copy.
- Adds convenience methods: replace(), aslist(), astuple(), and iterator for dict(this). These methods won't override existing ones. They will be added only if the names aren't used.
- Setting the default as `None` automatically makes the type into `Optional`, if it isn't already.
- Members without a default are allowed after members with a default (but they are required in order to create the instance)

3. Misc

- Frozen by default

All of the above differences are configurable and extendable.

Parameters

- **check_types** (*Union[bool, str]*) – Whether or not to validate the values, according to the given type annotations. Possible values: False, True, or 'cast'
- **config** (Configuration) – Configuration to modify dataclass behavior, mostly regarding type validation.

Example

```
>>> @dataclass
>>> class Point:
...     x: int
...     y: int

>>> p = Point(2, 3)
>>> p
Point(x=2, y=3)
>>> dict(p)          # Maintains order
{'x': 2, 'y': 3}

>>> p.replace(x=30)  # New instance
Point(x=30, y=3)
```

Part of the added ergonomics and functionality were influenced by Pydantic:

- Members that are assigned `None`, automatically become `Optional`. (Unless specified otherwise through `config`)
- Members without a default value, following members with a default value, are now allowed (and will fail if not assigned on init).

2.2 Added methods

The following functions, which are available as at the module level, will also be available as methods of the dataclass instances. These methods won't override existing ones; They will be added only if the names aren't already used.

runtype.dataclass.**replace**(*inst*, ***kwargs*)

Returns a new instance, with the given attibutes and values overwriting the existing ones.

Useful for making copies with small updates.

Examples

```
>>> @dataclass
... class A:
...     a: int
...     b: int
>>> A(1, 2).replace(a=-2)
A(a=-2, b=2)
```

```
>>> some_instance.replace() == copy(some_instance)   # Equivalent operations
True
```

runtype.dataclass.**astuple**(*inst*)

Returns a tuple of the values

runtype.dataclass.**aslist**(*inst*)

Returns a list of the values

runtype.dataclass.**json**(*inst*)

Returns a JSON of values, going recursively into other objects (if possible)

2.3 Configuration

class runtype.dataclass.**Configuration**

Generic configuration template for dataclass. Mainly for type-checking.

To modify dataclass behavior, inherit and extend this class, and pass it to the *dataclass()* function as the `config` parameter. (parameter `check_types` must be nonzero)

Example

```
class IsMember(Configuration):
    @staticmethod
    def ensure_isa(a, b):
        if a not in b:
            raise TypeError(f"{a} is not in {b}")

@dataclass(config=IsMember())
class Form:
    answer1: ("yes", "no")
    score: range(1, 11)

...

>>> Form("no", 3)
Form(answer1='no', score=3)

>>> Form("no", 12)
Traceback (most recent call last):
    ...
TypeError: 12 is not in range(1, 11)
```

on_default(*default*)

Called whenever a dataclass member is assigned a default value.

abstract ensure_isa(*a*, *b*, *sampler=None*)

Ensure that 'a' is an instance of type 'b'. If not, raise a TypeError.

abstract cast(*obj*, *t*)

Attempt to cast 'obj' to type 't'. If such a cast is not possible, raise a TypeError.

The result is expected to pass *self.ensure_isa(res, t)* without an error, however this assertion is not validated, for performance reasons.

class runtype.dataclass.**PythonConfiguration**

Configuration to support Mypy-like and Pydantic-like features

This is the default class given to the `dataclass()` function.

2.4 Casting

When called with the option `check_types="cast"`, values that are provided to instanciate the dataclass will be cast instead of validated.

Runtype will only attempt to cast in situations when no data is lost when converting the value.

The following casts are currently implemented:

- str -> int
- str -> datetime
- int -> float

If a cast fails, Runtype raises a *TypeError*. (same as when validation fails)

More casts will be added in time.

For non-builtin types, Runtype will attempt to call the *cast_from* class-method, if one exists.

Example:

```
@dataclass
class Name:
    first: str
    last: str = None

    @classmethod
    def cast_from(cls, s: str):
        return cls(*s.split())

@dataclass(check_types='cast')
class Person:
    name: Name

p = Person("Albert Einstein")
assert p.name.first == 'Albert'
assert p.name.last == 'Einstein'
```

2.5 Sampling

When called with the option `check_types="sample"`, lists and dictionaries will only have a sample of their items validated, instead of each item.

This approach will validate big lists and dicts much faster, but at the cost of possibly missing anomalies in them.

2.6 Performance

Type verification in classes introduces a small run-time overhead.

When running in production, it's recommended to use the *-O* switch for Python. It will make Runtype skip type verification in dataclasses. (unless *check_types* is specified.)

Alternatively, you can use a shared dataclass decorator, and enable/disable type-checking with a single change.

Example:

```
# common.py
import runtype

from .settings import DEBUG   # Define DEBUG however you want

dataclass = runtype.dataclass(check_types=DEBUG)
```

2.6.1 Compared to Pydantic

Using Pydantic's own benchmark, runtype performs twice faster than Pydantic. (or, Pydantic is twice slower than Runtype)

```
pydantic best=63.839s/iter avg=65.501s/iter stdev=1.763s/iter version=1.9.1
attrs + cattrs best=45.607s/iter avg=45.804s/iter stdev=0.386s/iter version=21.4.0
runtype best=31.500s/iter avg=32.281s/iter stdev=0.753s/iter version=0.2.7
```

See the code here.

CHAPTER

THREE

DISPATCH

Provides a decorator that enables multiple-dispatch for functions.

Features:

- Full specificity resolution
- Mypy support

(Inspired by Julia)

See *benchmarks*.

3.1 Decorator

runtype.**Dispatch**(*typesystem: ~runtype.typesystem.TypeSystem = <runtype.validation.PythonTyping object>*)

Creates a decorator attached to a dispatch group, that when applied to a function, enables multiple-dispatch for it.

Parameters

typesystem (*Typesystem*) – Which type-system to use for dispatch. Default is Python's.

Example

```
>>> from runtype import Dispatch
>>> dp = Dispatch()

>>> @dp
... def add1(i: Optional[int]):
...     return i + 1

>>> @dp
... def add1(s: Optional[str]):
...     return s + "1"

>>> @dp
... def add1(a):  # Any, which is the least-specific
...     return (a, 1)

>>> add1(1)
2
```

(continues on next page)

(continued from previous page)

```
>>> add1("1")
11

>>> add1(1.0)
(1.0, 1)
```

class runtype.dispatch.**MultiDispatch**(*typesystem: TypeSystem*, *test_subtypes: Sequence[int] = ()*)

Creates a dispatch group for multiple dispatch

Parameters

- **typesystem** (*typesystem - instance for interfacing with the*) –
- **test_subtypes** – indices of params that should be matched by subclass instead of isinstance.

3.2 What is multiple-dispatch?

Multiple-dispatch is an advanced technique for structuring code, that complements object-oriented programming.

Unlike in OOP, where the type of the "object" (or: first argument) is always what determines the dispatch, in multiple-dispatch all the arguments decide together, according the idea of specificity: The more specific classes (i.e. subclasses) get picked before the more abstract ones (i.e. superclasses).

That means that when you need to define a logical operation that applies to several types, you can first solve the most abstract case, and then slowly add special handling for more specific types as required. If you ever found yourself writing several "isinstance" in a row, you could probably use multiple-dispatch to write better code!

Multiple-dispatch allows you to:

1. Write type-specific functions using a dispatch model that is much more flexible than object-oriented.
2. Group your functions based on "action" instead of based on type.

You can think of multiple-dispatch as function overloading on steroids.

3.3 Runtype's dispatcher

Runtype's dispatcher is fast, and will never make an arbitrary choice: in ambiguous situations it will always throw an error.

As a side-effect, it also provides type validation to functions. Trying to dispatch with types that don't match, will result in a dispatch-error.

Dispatch chooses the right function based on the idea specificity, which means that *class MyStr(str)* is more specific than *str*, and so on:

MyStr(str) < str < Union[int, str] < object

It uses the *validation* module as the basis for its type matching, which means that it supports the use of *typing* classes such as *List* or *Union* (See "limitations" for more on that).

Some classes cannot be compared, for example *Optional[int]* and *Optional[str]* are ambiguous for the value *None*. See "ambiguity" for more details.

Users who are familiar with Julia's multiple dispatch, will find runtype's dispatch to be very familiar.

Unlike Julia, Runtype asks to instanciate your own dispatch-group, to avoid collisions between different modules and projects that aren't aware of each other.

Ideally, every project will instanciate Dispatch only once, in a module such as *utils.py* or *common.py*.

3.4 Basic Use

Multidispatch groups functions by their name. Functions of different names will never collide with each other.

The order in which you define functions doesn't matter to runtype, but it's recommended to order functions from most specific to least specific.

Example:

```
from runtype import multidispatch as md

@dataclass(frozen=False)
class Point:
    x: int = 0
    y: int = 0

    @md
    def __init__(self, points: list | tuple):
        self.x, self.y = points

    @md
    def __init__(self, points: dict):
        self.x = points['x']
        self.y = points['y']

# Test constructors
p0 = Point()                          # Default constructor
assert p0 == Point(0, 0)              # Default constructor
assert p0 == Point([0, 0])            # User constructor
assert p0 == Point((0, 0))            # User constructor
assert p0 == Point({"x": 0, "y": 0}) # User constructor
```

A different dispatch object is created for each module, so collisions between different modules are impossible.

Users who want to define a dispatch across several modules, or to have more granular control, can use the Dispatch class:

```
from runtype import Dispatch
dp = Dispatch()
```

Then, the group can be used as a decorator for any number of functions, in any module.

Functions will still be grouped by name.

3.5 Specificity

When the user calls a dispatched function group, the dispatcher will always choose the most specific function.

If specificity is ambiguous, it will throw a *DispatchError*. Read more in the "ambiguity" section.

Dispatch always chooses the most specific function, across all arguments:

Example:

```
from typing import Union

@md
def f(a: int, b: int):
    return a + b

@md
def f(a: Union[int, str], b: int):
    return (a, b)

...

>>> f(1, 2)
3
>>> f("a", 2)
('a', 2)
```

Although both functions "match" with *f(1, 2)*, the first definition is unambiguously more specific.

3.6 Ambiguity in Dispatch

Ambiguity can result from two situations:

1. The argument matches two parameters, and neither is a subclass of the other (Example: *None* matches both *Optional[str]* and *Optional[int]*)
2. Specificity isn't consistent in one function - each argument "wins" in a different function.

Example:

```
>>> @md
... def f(a, b: int): pass
>>> @md
... def f(a: int, b): pass
>>> f(1, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
runtype.dispatch.DispatchError: Ambiguous dispatch
```

Dispatch is designed to always throw an error when the right choice isn't obvious.

Another example:

```
@md
def join(seq, sep: str = ''):
```

(continues on next page)

(continued from previous page)

```
    return sep.join(str(s) for s in seq)

@md
def join(seq, sep: list):
    return join(join(sep, str(s)) for s in seq)
...

>>> join([0, 0, 7])                # -> 1st definition
'007'

>>> join([1, 2, 3], ', ')          # -> 1st definition
'1, 2, 3'

>>> join([0, 0, 7], ['(', ')'])    # -> 2nd definition
'(0)(0)(7)'

>>> join([1, 2, 3], 0)             # -> no definition
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
runtype.dispatch.DispatchError: Function 'join' not found for signature (<class 'list'>,
↪<class 'int'>)
```

Dispatch chooses the right function based on the idea specificity, which means that *class MyStr(str)* is more specific than *str*, and so on: *MyStr(str)* < *str* < *Union[int, str]* < *object*.

3.7 MyPy support

multidispatch works with mypy by employing the typing.overload decorator, aiding in granular type resolution.

However, due to the limited design of the *typing.overload* decorator, there are several rules that need to be followed, and limitations that should be considered.

1. For MyPy's benefit, more specific functions should be placed above less specific functions.
2. The last dispatched function of each function group, must be written without type declarations (making it the least specific), and use the multidispatch_final decorator. It is recommended to use this function for error handling and default functionality.

Note: Mypy doesn't support all of the functionality of Runtype's dispatch, such as full specificity resolution. Therefore, some valid dispatch constructs will produce an error in mypy.

Example usage:

```
from runtype import multidispatch as md, multidispatch_final as md_final

@md
def join(seq, sep: str = ''):
    return sep.join(str(s) for s in seq)

@md
def join(seq, sep: list):
    return join(join(sep, str(s)) for s in seq)
```

(continues on next page)

(continued from previous page)

```
@md_final
def join(seq, sep):
    raise NotImplementedError()

# Calling join() with the wrong types -
join(1,2)   # At runtime, raises NotImplementedError

# Mypy generates the following report:
#   error: No overload variant of "join" matches argument types "int", "int"  [call-
↪overload]
#   note: Possible overload variants:
#   note:     def join(seq: Any, sep: str = ...) -> Any
#   note:     def join(seq: Any, sep: list[Any]) -> Any
```

3.8 Performance

Multiple-dispatch caches call-signatures by default, and adds a small runtime overhead after the first call.

See *benchmarks*.

Dispatch is not recommended for use in functions that are called often in time-critical code.

3.9 Limitations

Dispatch currently doesn't support, and will simply ignore:

- keyword arguments (Dispatch relies on the order of the arguments)
- **args*
- ***kwargs*

These may be implemented in future releases.

Dispatch does not support generics or constraints. Avoid using *List[T]*, *Tuple[T]* or *Dict[T1, T2]* in the function signature. (this is due to conflict with caching, and might be implemented in the future)

Union and *Optional* are supported.

CHAPTER

FOUR

TYPE CLASSES

A collection of classes that serve as building blocks for making your own type system.

You can use that type system to:

- reason about your program's logic
- use it with runtype for customized dispatch and validation.

4.1 Types

Note: These types are not specific to the Python type system!

class runtype.base_types.**Type**

Abstract Type class. Every type inherit from it.

class runtype.base_types.**AnyType**

Represents the Any type.

Any may contain any other type, and be contained by any other type.

For any type 't' within the typesystem, t is a subtype of Any (or: t <= Any) But also Any is a subtype of t (or: Any <= t)

class runtype.base_types.**DataType**

Abstract class for a data type.

A data-type is any type that contains non-type information.

Example of possible data-types: int, float, text, list

class runtype.base_types.**SumType**(*types*)

Implements a sum type, i.e. a disjoint union of a set of types.

Similar to Python's *typing.Union*.

class runtype.base_types.**ProductType**(*types*)

Implements a product type, i.e. a record / tuple / struct

class runtype.base_types.**ContainerType**

Base class for containers, such as generics.

class runtype.base_types.**GenericType**(*base:* Type, *item: Union[type,* Type*]*, *variance*)

Implements a generic type. i.e. a container for items of a specific type.

For any two generic types a[i] and b[j], it's true that a[i] <= b[j] iff a <= b and i <= j.

class runtype.base_types.**PhantomType**

Implements a base for phantom types.

A phantom type is a "meta" type that can wrap existing types, but it is transparent (subtype checks may skip over it), and has no effect otherwise.

class runtype.base_types.**PhantomGenericType**(*base*, *item=All*)

Implements a generic phantom type, for carrying metadata within the type signature.

For any phantom type p[i], it's true that p[i] <= p but also p[i] <= i and i <= p[i].

class runtype.base_types.**Validator**

Defines the validator interface.

validate_instance(*obj*, *sampler: Optional[Callable[[Sequence], Sequence]] = None*)

Validates obj, raising a TypeMismatchError if it does not conform.

If sampler is provided, it will be applied to the instance in order to validate only a sample of the object. This approach may validate much faster, but might miss anomalies in the data.

abstract test_instance(*obj*, *sampler=None*)

Tests obj, returning a True/False for whether it conforms or not.

If sampler is provided, it will be applied to the instance in order to validate only a sample of the object.

class runtype.base_types.**Constraint**(*for_type*, *predicates*)

Defines a constraint, which activates during validation.

CHAPTER

FIVE

TYPING SUPPORT

Runtype supports a wide-range of types and typing constructs, however full-support is still work in progress.

For now, some constructs are available for validation, but not for dispatch.

Here is the detailed list:

Types / Constructs	Validation	Dispatch
Primitives (None, bool, float, int, str, etc.)	✓	✓
Date primitives (datetime, date, time, timedelta)	✓	✓
Containers, non-generic (list, tuple, dict)	✓	✓
Callable, non-generic (callable)	✓	✓
abc.Set, abc.MutableMapping, etc.	✓	✓
typing.AbstractSet	✓	✓
typing.Any	✓	✓
typing.Union, Optional	✓	✓
typing.Type (Type[x])	✓	✓
typing.Literal	✓	✓
Generic containers (list[x], tuple[x], dict[x])	✓	TODO
Infinite tuple (tuple[x, ...])	✓	TODO
Generic callable	TODO	TODO
typing.IO	TODO	TODO
TypeVar	TODO	TODO
Protocol	TODO	TODO

CHAPTER

SIX

BENCHMARKS

The following benchmarks were run using *pytest-benchmark* and plotted using *matplotlib*.

The code for running and plotting these benchmarks is included in the repo. See: `docs/generate_benchmarks.sh`

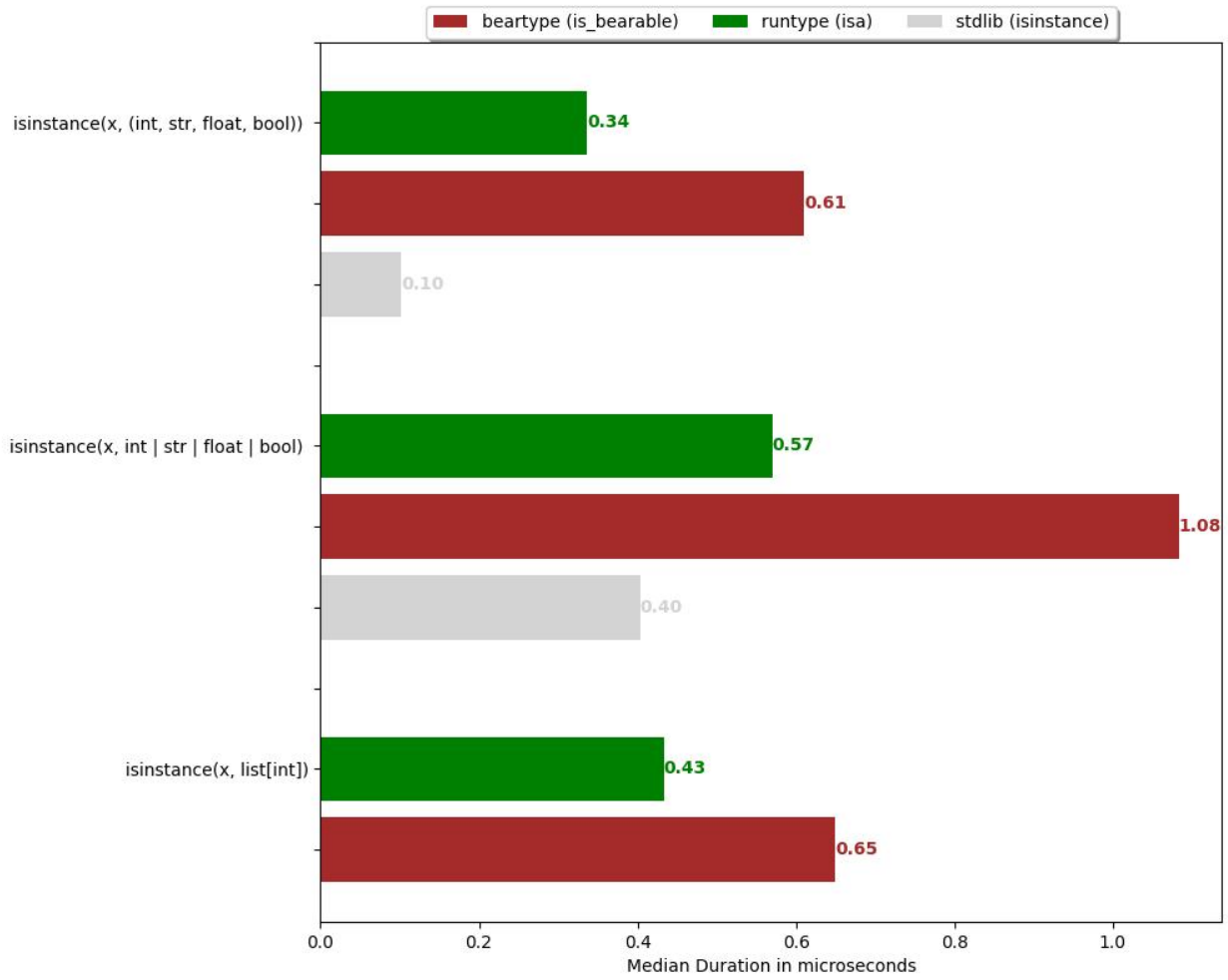
Benchmark contributions for more use-cases or new libraries are welcome!

6.1 Validation (isinstance)

In the image below, we can see runtype is faster than its (only?) competitor, the library beartype.

We can see the native `isinstance()` is faster than runtype's `isa()`. However, it isn't quite a fair comparison, because it doesn't support all the types that `isa()` supports.

Conclusion: Runtype beats the competition!



6.2 Dispatch

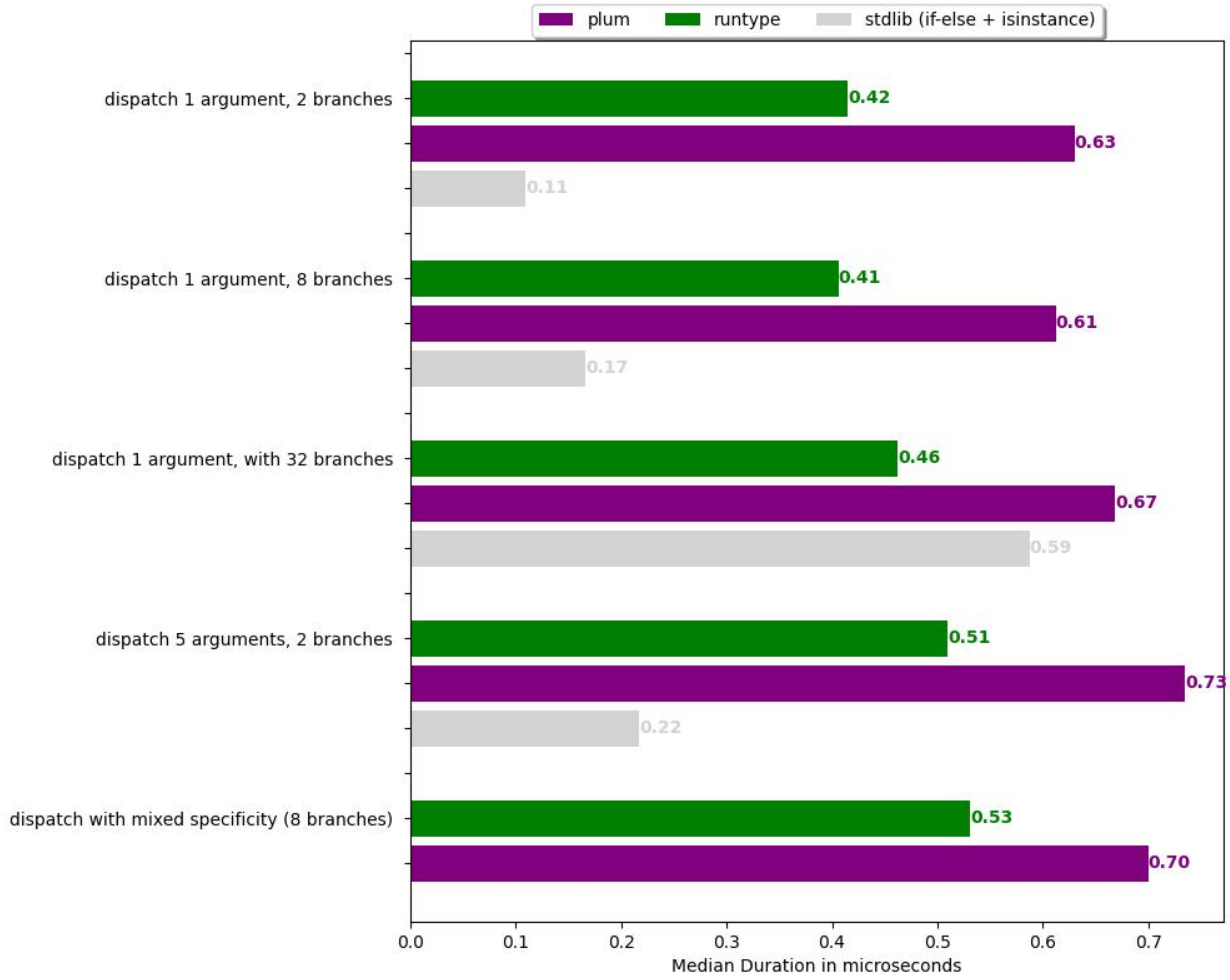
In the images below, we cann see runtyp's multiple dispatch is faster than its (only?) competitor, the library plum.

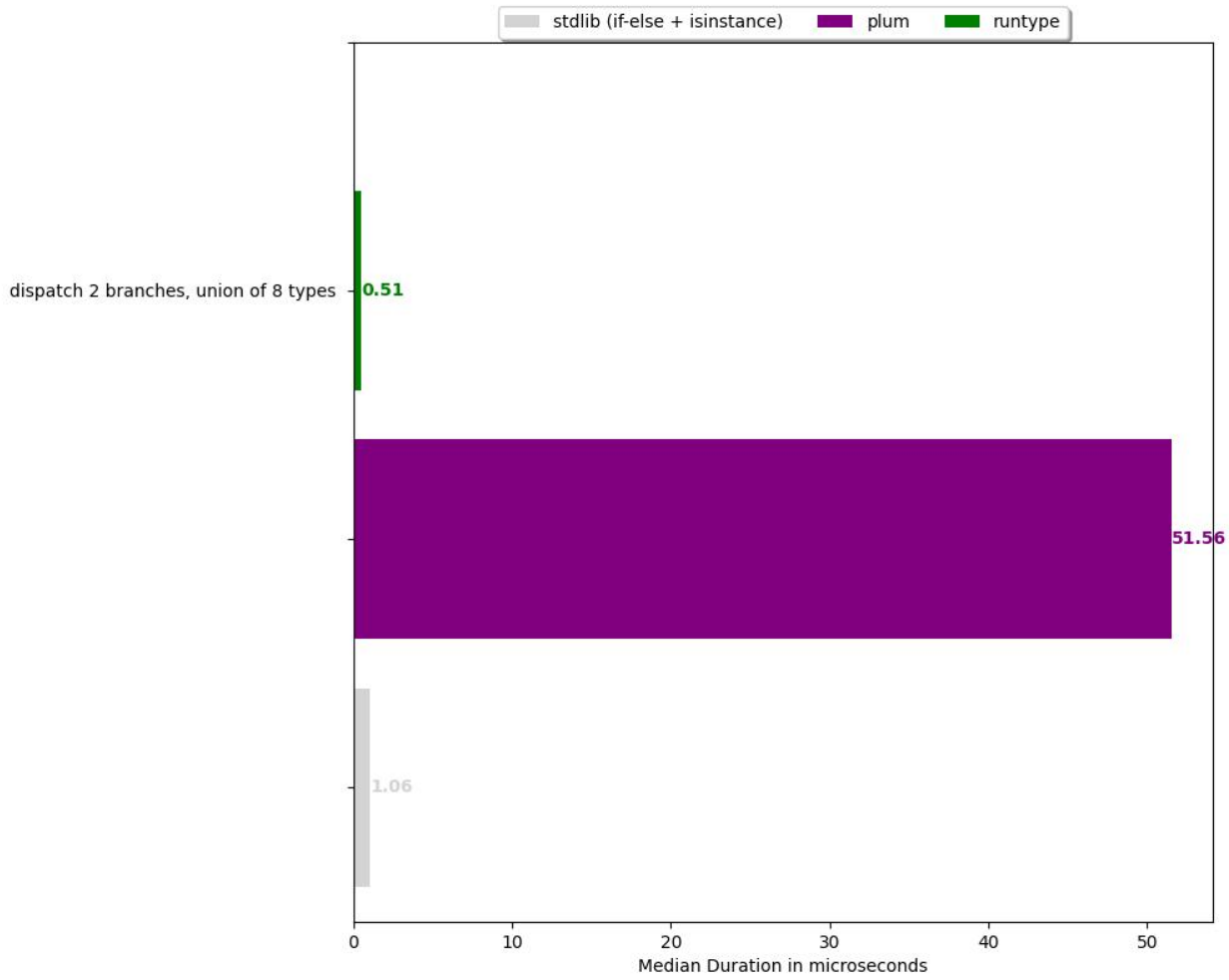
We can see that the naive approach of using if-else is faster for a small amount of branches, but by 32 branches runtype is already significantly faster.

Curiously, for dispatching to unions of types, runtype is twice faster (!) than the naive if-else approach, even for a very low number of branches.

It's worth noting that using while using if-else is sometimes faster, it is order-sensitive, and lacks important abilities that runtype offers, such as specificity-resolution on multiple arguments.

Conclusion: Runtype beats the competition, and sometimes the if-else approach too!


plum
runtype
stdlib (if-else + isinstance)
dispatch 1 argument, 2 branches
0.42
0.63
0.11
dispatch 1 argument, 8 branches
0.41
0.61
0.17
dispatch 1 argument, with 32 branches
0.46
0.67
0.59
dispatch 5 arguments, 2 branches
0.51
0.73
0.22
dispatch with mixed specificity (8 branches)
0.53
0.70
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Median Duration in microseconds



Runtype is a collection of run-time type utilities for Python.

It contains the following user-facing modules:

- *Validation (isa & issubclass)* - Alternatives to 'isinstance' and 'issubclass'
- *Dataclass* - Type-validation in dataclasses
- *Dispatch* - Multiple dispatch
- *Type Classes* - Utilities for creating type systems

CHAPTER

SEVEN

INSTALL

```
pip install runtype
```

No dependencies.

Requires Python 3.6 or up.

7.1 ArchLinux

ArchLinux users can install the package by running:

```
yay -S python-runtype
```

INDEX