

# Typing Python with mypy

## Using types to rule out bugs

**Dominic Orchard**



UNIVERSITY OF  
CAMBRIDGE



Institute of  
Computing for  
Climate Science

**ICCS Summer School - July 10-12th 2024**

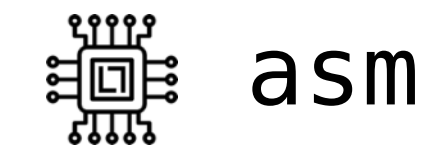
# Warmup!

1 + 1 = 2

"hello" + 1 = "hello1"



= "iello"



= "help"

??!

"hello" \* 2 = "hellohello"



"hello" / 2 = ??? 🤔

Types communicate to us what  
the computer can do

# Learning objectives



- Understand key ideas behind **specification** and **verification**
- Understand some key **concepts** and **terminology** behind types
- Learn about the **mypy tool** for typing in Python
- Develop ability to use types to **avoid bugs** and **write code more effectively**

# Validation

*Did we implement the right equations?*

VS

Focus on here today

# Verification

*Did we implement the equations right?*

# Challenge

*Telling these two apart when results are not as expected*

# Terminology: what does “*verified*” mean?

*Verification wrt. a specification*

i.e. `check(implementation, specification)`

*∴ validation is verification*

*where specification  $\triangleq \approx_{\text{observation}}$*

*The value of a specification is what we make of it;  
it depends on our goals and values*

# How *much* verification?

- Lots of verification techniques out there:
  - Testing
  - Type systems
  - Deductive verification
  - Static analysis
  - Interactive theorem provers
  - Modelling and model checking

# How *much* verification?

“Lightweight Formal Methods” (Jackson, Wing, 1996)

"...except in safety-critical work, the **cost of full verification is prohibitive and early detection of errors is a more realistic goal.**

There can be no point embarking on the construction of a specification until it is known exactly **what the specification is for; which risks it is intended to mitigate**; and in which respects it will inevitably prove inadequate."

*Today we will mitigate against data errors*



# A helpful model: types as sets

- Set defined by its elements (*data*), e.g.,
  - ▶  $\mathbb{N}$  - Natural numbers  $\{1, 2, \dots\}$  or  $\{0, 1, 2, \dots\}$  depending who you ask!
  - ▶  $\mathbb{Z}$  - Integers  $\{\dots, -2, -1, 0, 1, 2, \dots\}$
  - ▶  $\mathbb{R}$  - Real numbers  $\{\dots, 0, 0.1, 0.11, \dots, e, \dots, \pi, \dots\}$
- Sets of pairs of  $A$  and  $B$  written  $A \times B$  (Cartesian product)
  - ▶ e.g.,  $\mathbb{N} \times \mathbb{N} = \{(1,1), (1,2), (2,1), (2,2), \dots\}$
- Functions from  $A$  to  $B$  written  $A \rightarrow B$ 
  - ▶ e.g.  $\text{abs} : \mathbb{Z} \rightarrow \mathbb{N}_0$
  - ▶  $\sqrt{\phantom{x}} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R} \times \mathbb{R}$
  - ▶  $+$  :  $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

Notational convention

*expression : type*

*type signature / specification*



# Static typing

vs.

# Dynamic typing

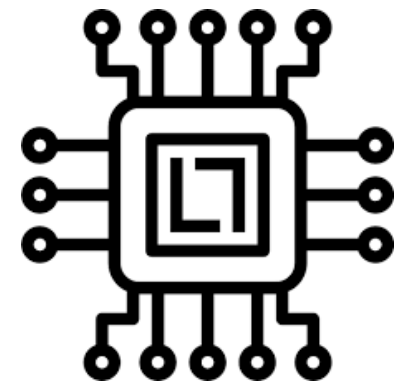


- Compiler first does **type checking**
- **Ill-typed** programs rejected
  - ▶ Intrinsic typing - *Ill-typed programs have no meaning (cannot be run)*
- **Well-typed** programs compiled, using types for optimisation
- Today: **we will use mypy to add static typing to Python**

- No pre-run checks
- Data stored with type information
- Operations check type information
- Errors occur “as it happens”

# Without types?

- E.g., in *assembly languages*



- *One type = bits!*
- **Everything works** / operations may not do what you want
- *Developer has to track meaning themselves*

Types eliminate a class of bugs

*“Well typed programs cannot go wrong”  
(Milner, 1978)*

*(For some definition of wrong!)*



# mypy

## An optional gradual, static type system for Python

- Gradually convert from dynamic to static typing
- Optional  $\implies$  *extrinsic typing* - ill-typed programs can still run (have meaning)
- Maths-like *type signatures*

```
flag : bool = True
```

```
def plus(x : int, y : int) -> int:  
    return x + y
```

# Getting mypy (if you want to 'code along')



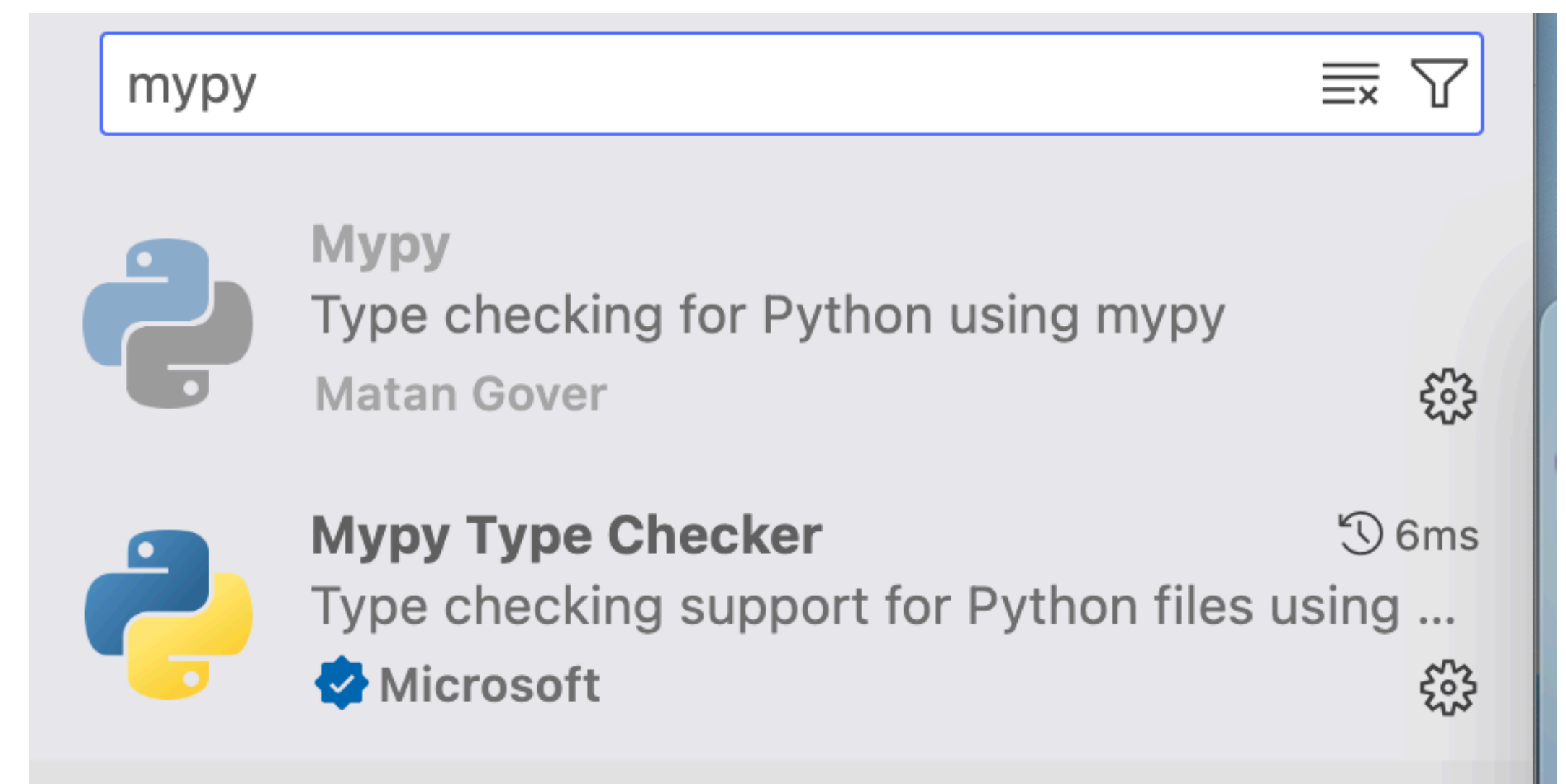
```
python -m pip install mypy
```

Or possibly:

```
python3 -m pip install mypy
```



You may want to use the  
vscode extension





# Mypy/Python primitive types

int

bool

float

str

None

*("no result" type)*

Any

*(fall-back, anything)*

```
def greet(name: str) -> None:  
    print("Hi " + name)
```





# Type constructors

Like *type functions*: create a type from other types

- For some type `t` then `list[t]` captures lists of elements (all) of type `t`

```
def greet_all(names: list[str]) -> None:  
    for name in names:  
        print('Hello ' + name)
```

cf.  $A \times B$  notation on sets

- `tuple[t1, t2, ...]` captures tuples with elements of type `t1`, `t2`, etc.

```
some_data : tuple[int, bool, str] = (42, True, "ICCS")
```





# Type constructors

Like *type functions*: create a type from other types

- `dict[k, v]` captures records/dictionaries of key `k` and value `v` type:

```
x: dict[str, float] = {"field1": 2.0, "field2": 3.0}
```

- `t1 | t2` captures either type `t1` or `t2` type (Python 3.10 `<= Union[t1, t2]`)

```
def myDiv(x : float, y : float) -> (float | None):  
    if y != 0: return x / y  
    else:      return None
```



# Type constructors and classes

Every class name *is* a type constructor

e.g.,

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

```
h : Complex = Complex(3.0, -4.5)
```



# Querying mypy

Ask mypy what it thinks the type is:

```
reveal_type(expression)
```

If you need to run too, hide `reveal_type` from runtime:

```
from typing import TYPE_CHECKING
```

```
if TYPE_CHECKING:  
    reveal_type(d1)
```

# Subtyping

- In theory literature,  $A$  is a subtype of  $B$  written  $A :< B$  (*think subsets*)



- Example: `list[t]` is a “subtype” of `Iterable[t]`

- Can pass arguments of a subtype to a function

$$\frac{x : A \quad f : B \rightarrow C \quad A :< B}{f(x) : C}$$

e.g.

```
def greet_all(names: Iterable[str]) -> None:
    for name in names:
        print('Hello ' + name)
```

```
names = ["Alice", "Brijesh", "Chenxi"]
greet_all(names)    # Ok!
```



# Parametric Polymorphism

(Also known as *generic types*)

- Consider the function

```
def first(xs : list[str]) -> str:  
    return xs[0]
```

- What if we want to use it with list[int] too?

```
def first_int(XS : list[int]) -> int:  
    return xs[0]
```

- Duplication bad for maintenance and understanding



# Parametric Polymorphism

(Also known as *generic types*)

- **Solution:** generalise to any element type T

```
T = TypeVar( 'T' )
```

```
def first(xs : list[type[T]]) -> type[T]:  
    return xs[0]
```

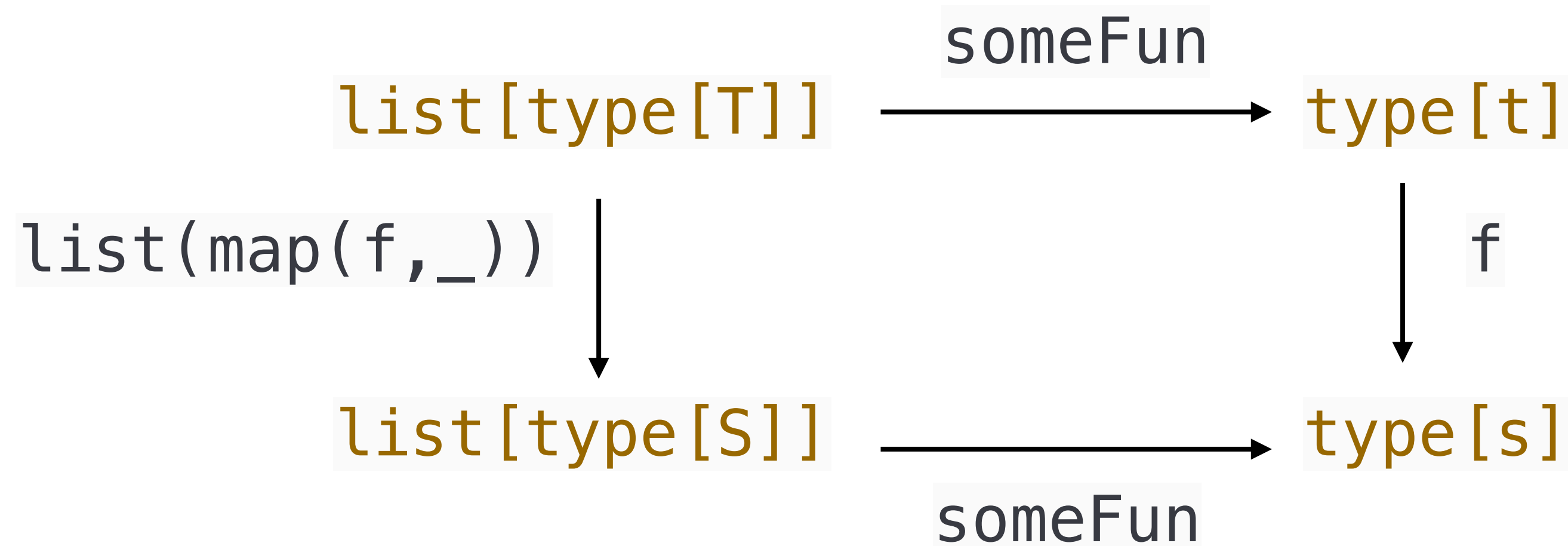
- (Note: requires an import)

```
from typing import TypeVar, Generic
```

# “Free theorems” follow from polymorphic types

- Consider `def someFun(XS : list[type[T]]) -> type[T]`
- “Universality” of `T` tells us we cannot inspect or compute with the `T` elements
- Implies the following (“*naturality*”) property:

`someFun(list(map(f, x))) = f(someFun(x))`



Note the right expression applies `f` once, the left applies it `len(x)` times.

∴ Optimisation!

# Function types

e.g., for typing higher-order functions

For a function with n-inputs (n-ary)  $A_1$  to  $A_n$  and return type  $B$ :

```
Callable[[A1, ..., An], B]
```

cf.  $A \rightarrow B$  notation on sets  
or  $(A_1 \times \dots \times A_n) \rightarrow B$

e.g.,

```
from typing import Callable
S = TypeVar('S')
T = TypeVar('T')
def memo(f : Callable[[S], T], x : S) -> tuple[S,T]:
    return (x, f(x))
```



# Worksheet

<https://shorturl.at/KVFtQ>



# mypy and NumPy

## Types for external libraries

Can use the class names already for numpy, e.g.,

```
import numpy as np  
myArray : np.ndarray = np.ndarray(shape=(2,2), dtype=float)
```

# mypy and NumPy

## Types for external libraries

```
import numpy.typing as npt
```

provides

- `ArrayLike` - objects that can be converted to arrays
- `DTypeLike` - objects that can be converted to dtypes
- `NDArray[T]` - numpy arrays of T values

Needs local config, e.g., via `mypy.ini`

```
[mypy]  
plugins = numpy.typing.mypy_plugin
```

# mypy and NumPy

## Types for external libraries

e.g.

```
import numpy as np
import numpy.typing as npt
```

```
def as_array(a: npt.ArrayLike) -> np.ndarray:
    return np.array(a)
```

```
def scale_array(a: float, arr: npt.NDArray[np.float64]) -> npt.NDArray[np.float64]:
    return a*arr
```

# Escape hatch!

- A type checker  $T$  is complete if, for all (valid) programs  $P$  then  $T(P)$  is true
- Most type checkers are *incomplete*  $\implies$  some valid programs rejected
- mypy has an escape hatch:

```
borked = 0 / "hello" # type: ignore
```

Does not raise a type checking error (though it clearly should)

# Coming into land.... What did we learn?



- Understand key ideas behind specification and verification
- Understand some key concepts and terminology behind types
  - “Sets” model
  - Static vs dynamic
  - Extrinsic vs intrinsic
  - Subtyping
  - Polymorphism

# Coming into land.... What did we learn?



- Learn about the mypy tool for typing in Python
  - mypy gives us extrinsic static typing
- Develop ability to use types to avoid bugs and write code more effectively
  - Go and practice on your own (see worksheet!)
  - Start using in projects



# Thanks- and happy typing!



<https://iccs.cam.ac.uk>



<https://dorchard.github.io>



types.pl/@dorchard



@dorchard



# VScode mypy plugin woes?

## No errors appear

- Check mypy
- Explicitly set path to mypy

```
% which mypy  
/opt/homebrew/bin/mypy
```

- Then edit settings.json, adding, e.g.:

```
"mypy-type-checker.path": ["/opt/homebrew/bin/mypy"]
```