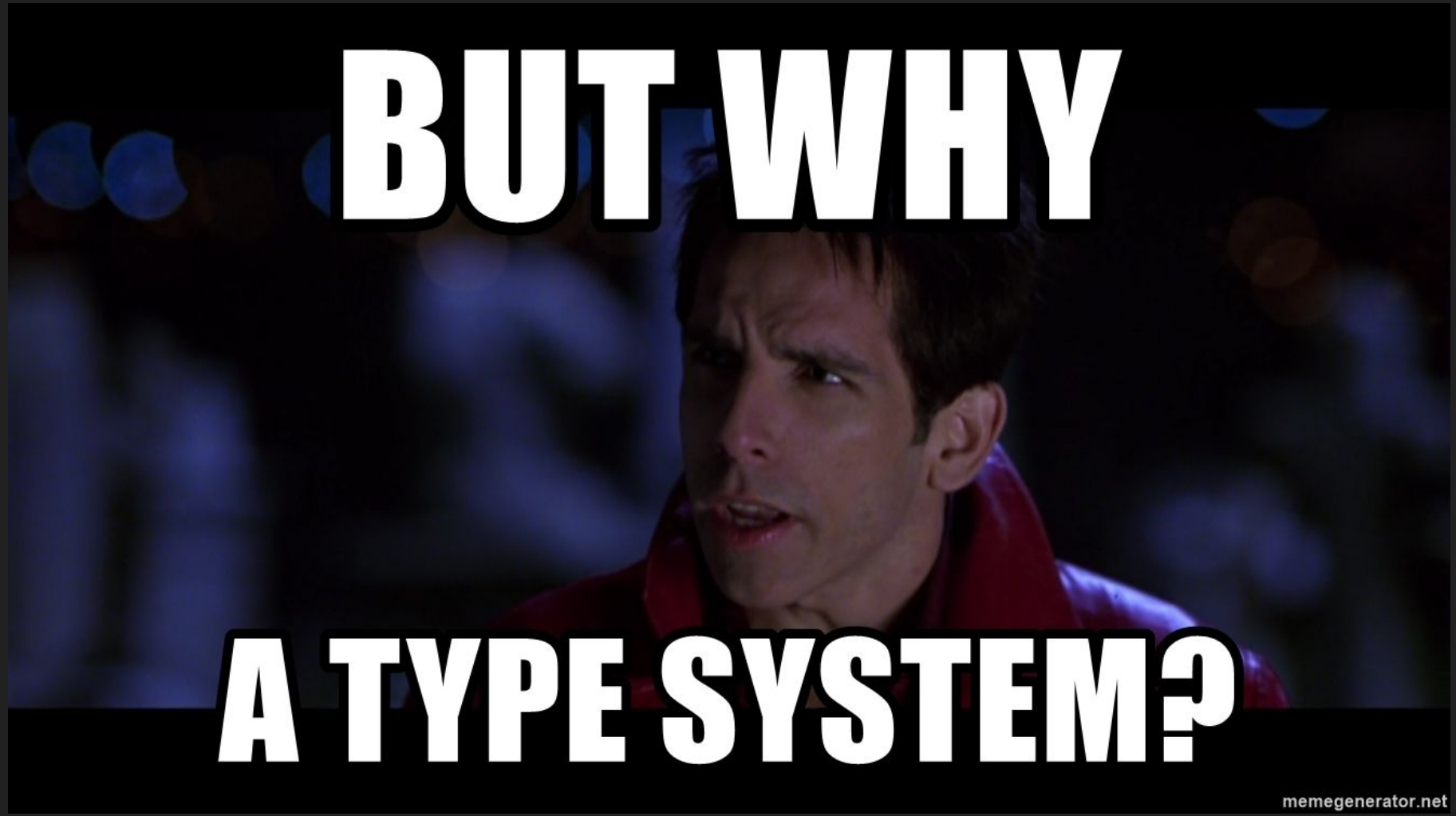
Type Systems

What we gonna talk about ?

- What it’s like to be a type checker
- What is Any
- Casting?

- Also we will be mainly focused on type hinting and static type checking
- Luckily Python and Typescript (TS) both function quite similarly in terms of type checking


BUT WHY
A TYPE SYSTEM?
memegenerator.net

Python or Typescript?

- This talk aims to be mostly language-independent but there will be deviations
- Also we will be mainly focused on type hinting and static type checking
- Luckily Python and Typescript (TS) both function quite similarly in terms of type checking

101 101 101 101 101 101 101 101 101

- **WHAT**: A type defines a set of possible values and a set of operations (for an object).

- **WHY**: Programming languages use type systems because they help both the programmer and computer better reason about programs.

- **HOW (static checking)**: We label variables with annotations that tell the Python/Typescript (TS) type checkers what their types are

At Runtime

Python is strongly typed:

3 + “five” —> TypeError

JavaScript is weakly typed:

[] + {} → “[object Object]”

AKA in Python if you do a random operation between two random types it is more likely to raise a TypeError; in Javascript it is more likely to do something quite odd

At Runtime

Both Python and JavaScript are dynamically typed:

```
x = 5
x = “five”
```

AKA variables can be reassigned to different types at runtime.

NOTE that this is not allowed statically by the default mypy/typescript settings

How does a type checker see code?

As a broad overgeneralization:
mypy and the TypeScript compiler look at your code and create a graph which has variables as its nodes and functions/operations as its edges

It then checks this graph for any inconsistencies based on type **annotations** and **inferences** eg:

- Calling functions using incorrect parameter types
- Assigning an incorrect type to a variable

What is a type annotation?

talk_length: int = “Type Systems”

const talkName: string = 30

A type annotation is an opportunity to tell the type checker and your fellow devs what you expect/require a variable/parameter to be.

What is a type inference?

In general mypy/Typescript (TS) will infer the types of variables that are not annotated. These inferences can be faulty/imprecise

What is a type inference?

EG if you want to create a list of strings and integers:

```
let count = 3
let mixed_list = [“hi”]
mixed_list.push(count )
```

TS infers that count is a number

TS infers that mixed_list is a str[]

TS will not add an int to a str[]

What is a type inference?

Solution:

```
let count = 3

let mixed_list: (string | number)[] = [“hi”]

mixed_list.push(3)
```

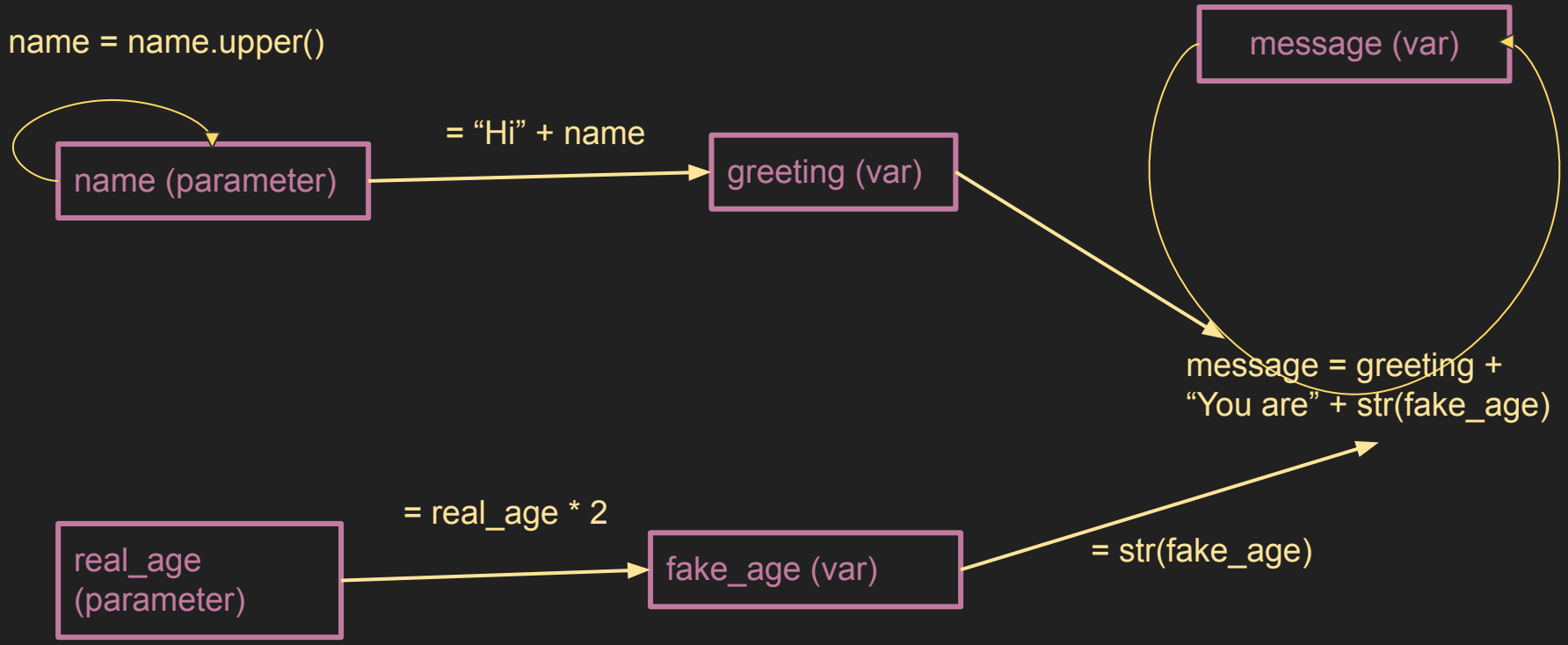
By explicitly labelling the list when it is first created we don’t run into the insufficiency of inference

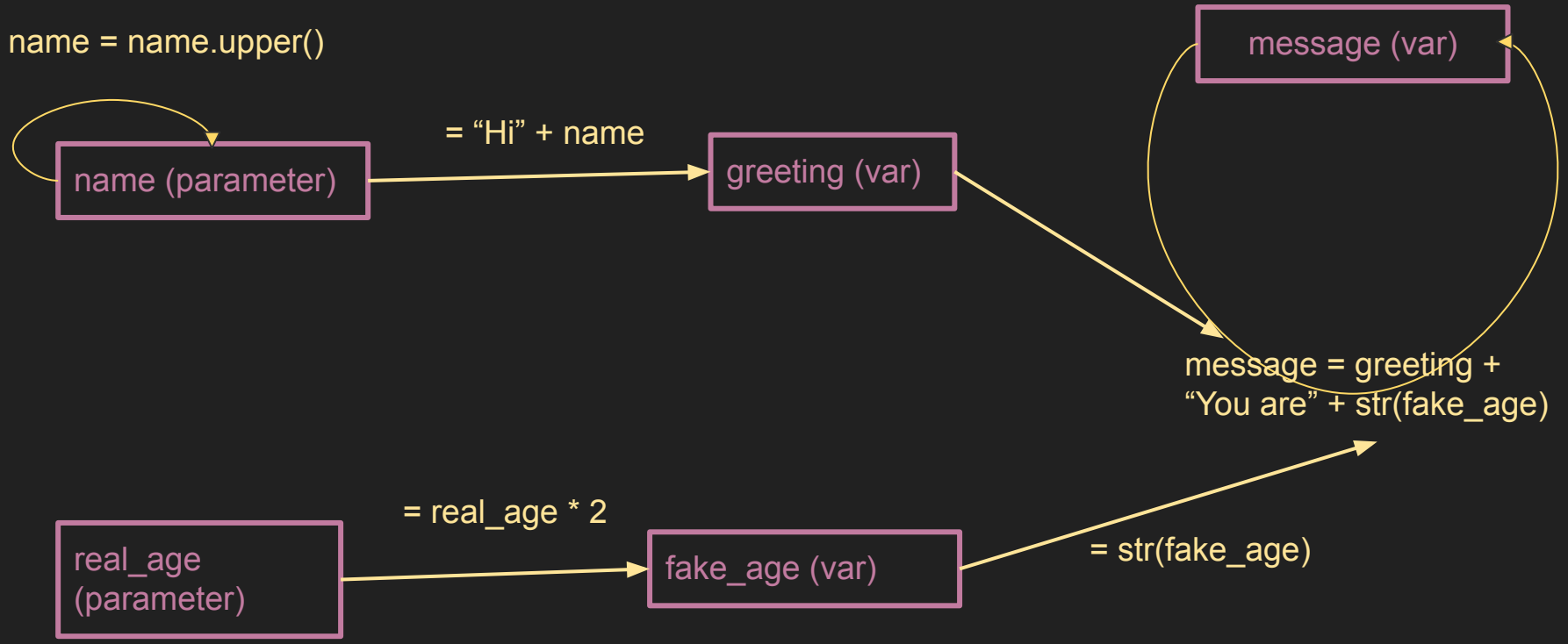
How does a type checker see code?

```
def prank_bday_message(name, real_age):
    message = ''
    name = name.upper()
    greeting = 'Hi, ' + name + ' it is your birthday.'
    fake_age = real_age * 2
    message = greeting + 'You are ' + str(fake_age) + ' today.'
    return message



prank_bday_message("Luke", 15)
>> 'Hi, LUKE it is your birthday.You are 30 today.'
```


name = name.upper()
name (parameter)
= “Hi” + name
greeting (var)
message (var)
message = greeting +
“You are” + str(fake_age)
real_age
(parameter)
= real_age * 2
fake_age (var)
= str(fake_age)



We want to guarantee that message is a string at the end

What can go wrong?

```
def prank_bday_message(name, real_age):
    message = ''
    name = name.upper()
    greeting = 'Hi, ' + name + ' it is your birthday.'
    fake_age = real_age * 2
    message = greeting + 'You are ' + str(fake_age) + ' today.'
    return message
```

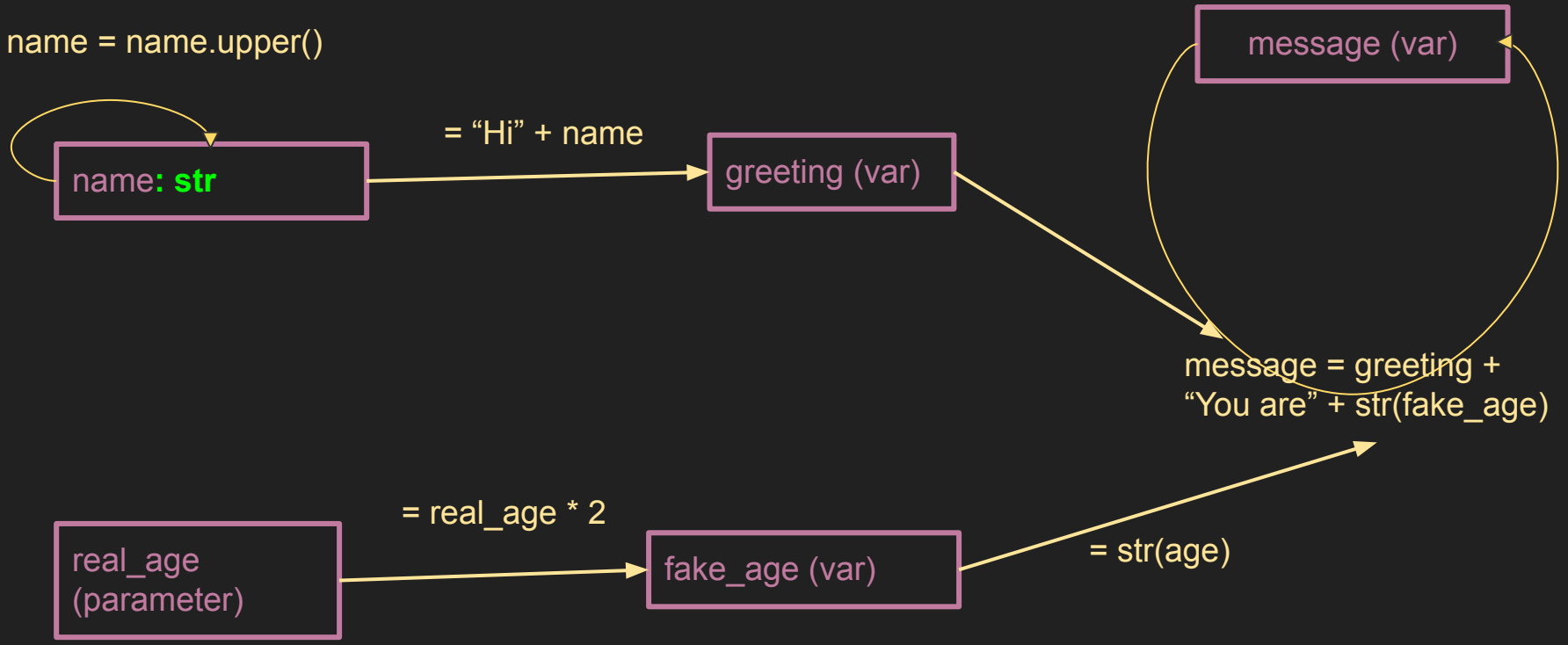
```
prank_bday_message("Luke", "ten")
>> 'Hi, LUKE it is your birthday.You are tententen today.'
```

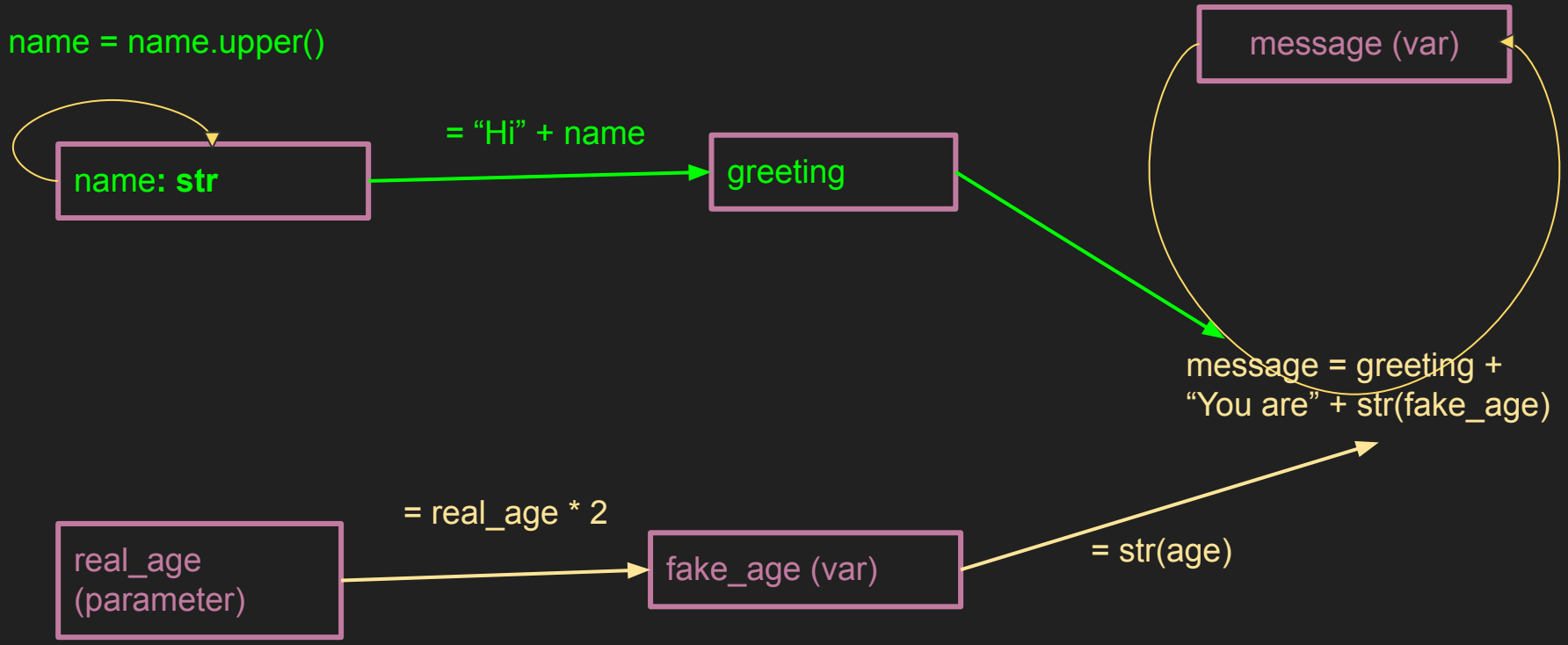
```
prank_bday_message(11, "ten")
>> AttributeError: 'int' object has no attribute 'upper'
```

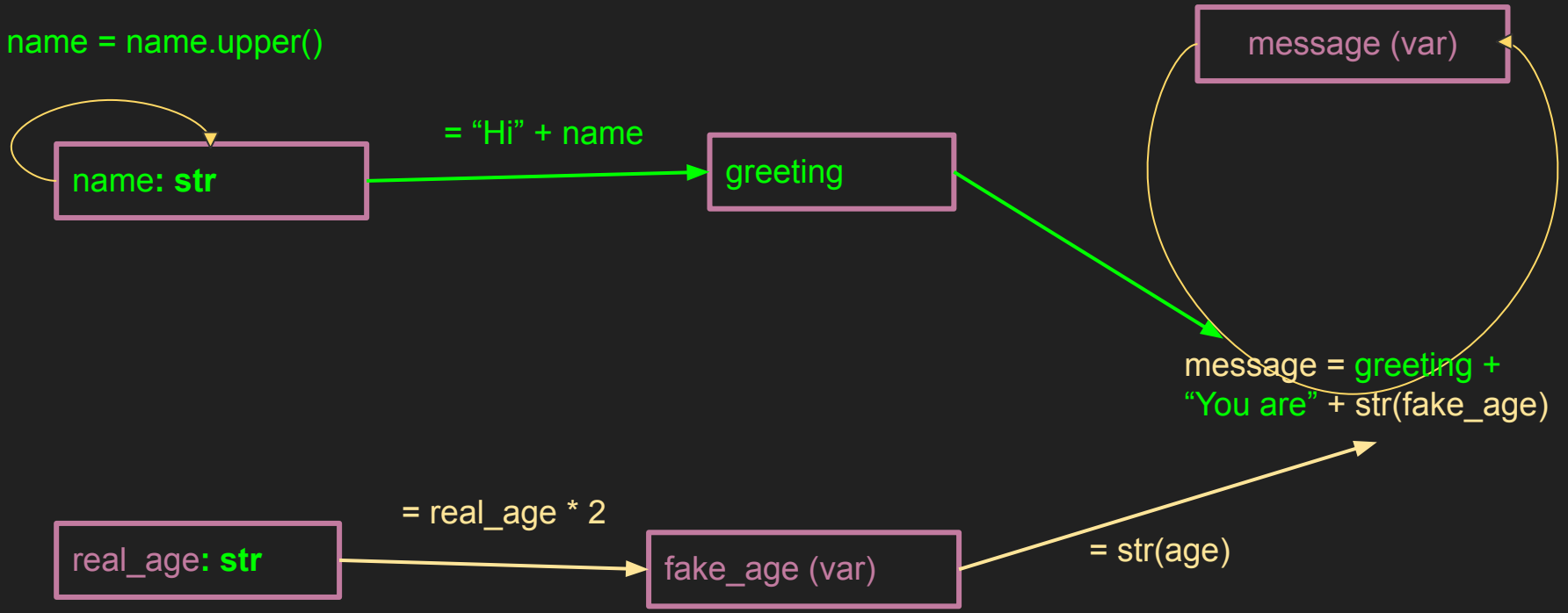
What do we need to check for type consistency?

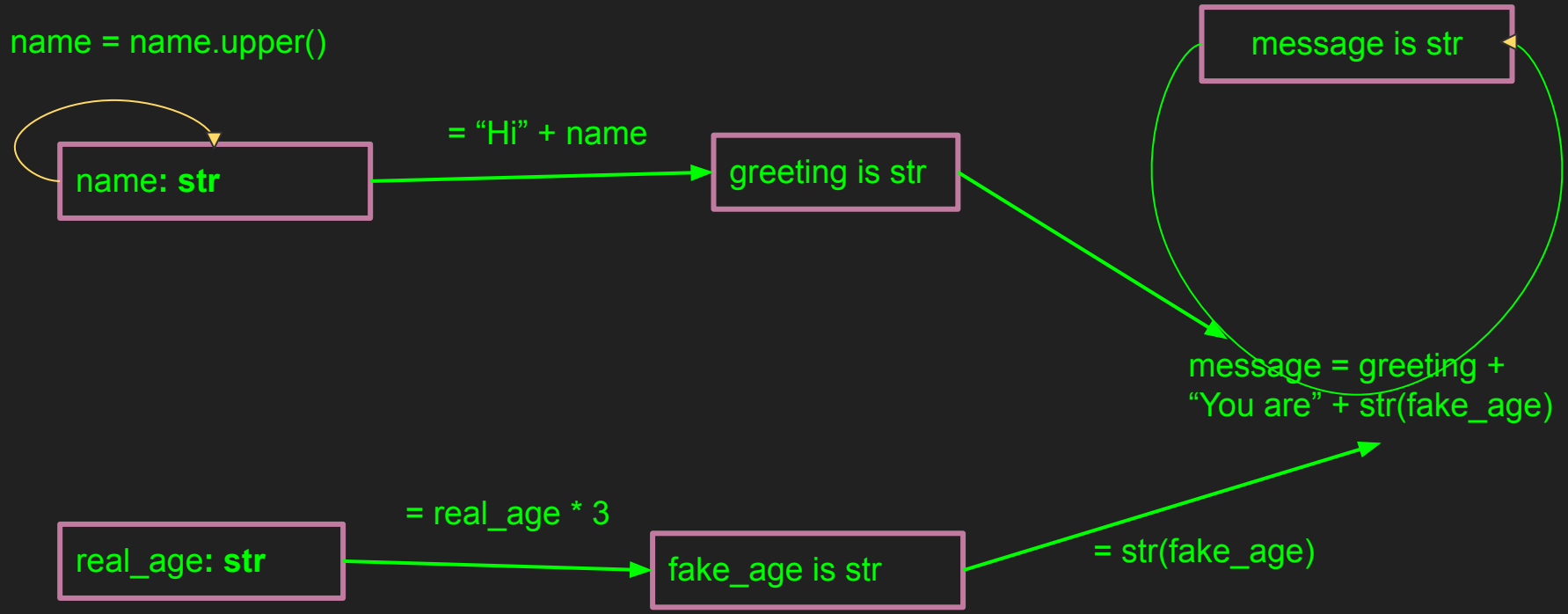
- name has the method .upper()
- name.upper() can do + with “Hi”
- greeting can do + with “You are”
- real_age can do * with 2
- we can call str() on fake_age
- greeting + “You are” + str(fake_age) is a str

ugh.

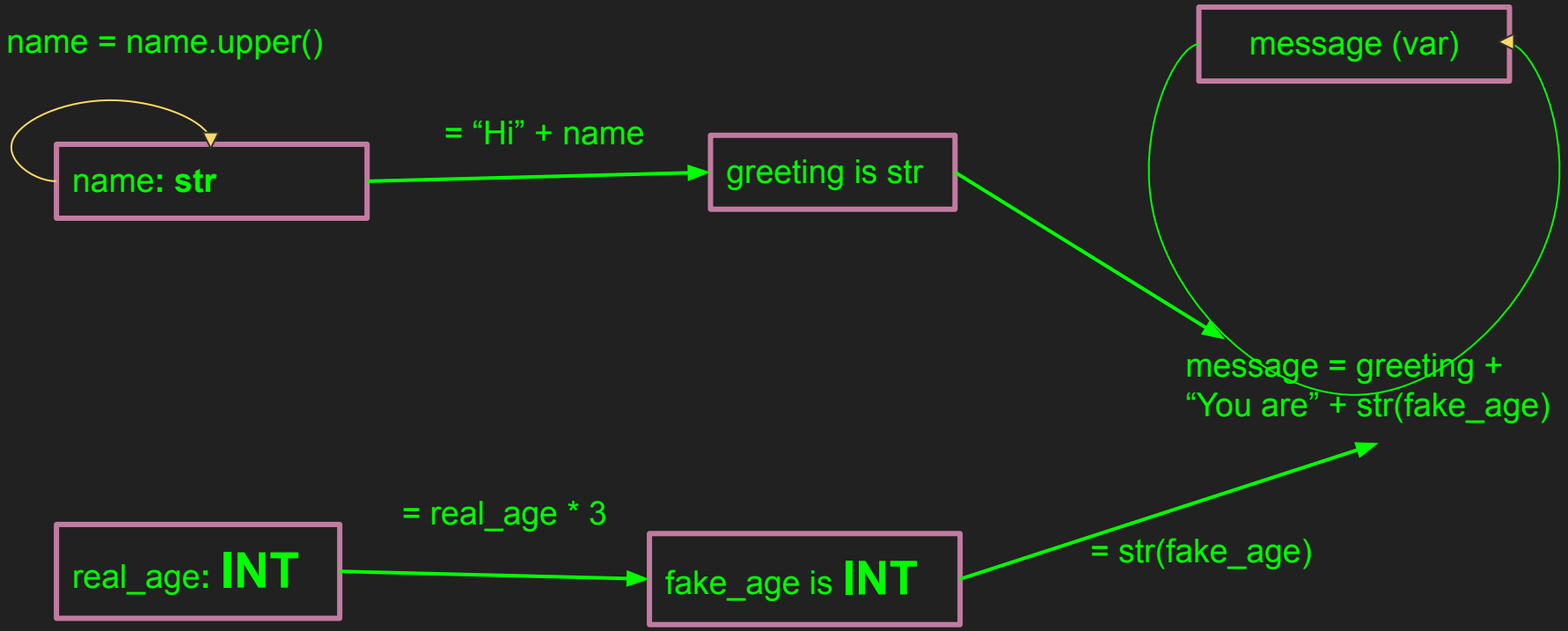

name = name.upper()
name: str
= “Hi” + name
greeting (var)
message (var)
message = greeting +
“You are” + str(fake_age)
real_age
(parameter)
= real_age * 2
fake_age (var)
= str(age)


name = name.upper()
name: **str**
= “Hi” + name
greeting
message (var)
message = greeting +
“You are” + str(fake_age)
real_age
(parameter)
= real_age * 2
fake_age (var)
= str(age)


name = name.upper()
name: str
= “Hi” + name
greeting
message (var)
message = greeting +
“You are” + str(fake_age)
real_age: str
= real_age * 2
fake_age (var)
= str(age)




```
prank_bday_message("Luke", "ten")
>> 'Hi, LUKE it is your birthday.You are tententen today.'
```

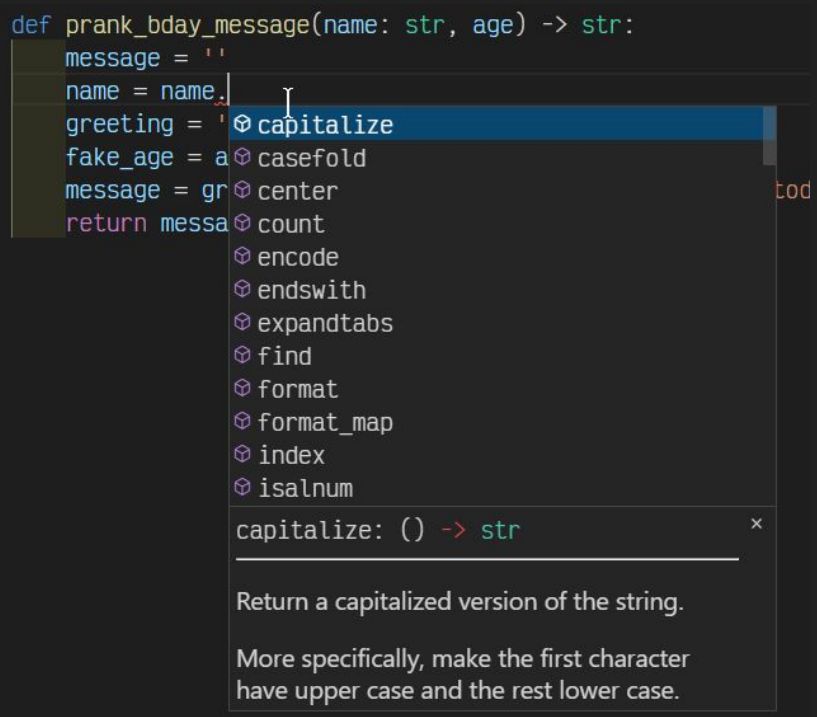


Types can only protect us when they are correct!

The Type checker is really smart! It did all this:

- name has the method .upper()
- name.upper() can do + with “Hi”
- greeting can do + with “You are”
- real_age can do * with 2
- we can call str() on fake_age
- greeting + “You are” + str(fake_age) is a str

The Type checker is smart! It even did this:



Our IDEs use type information to help us out!

Because name is annotated to be str, VSCode is able to suggest all the possible string functions that could be called.

Without the annotation this is not possible!

The Type checker is dumb! It only did this:

- Our friendly neighbourhood programmer claims that name is a string
- The python class `str` has the method .upper()
- str.upper() claims it returns a string
- str.__add__() claims it takes two strings and returns a string
- etc…

The Type checker is really dumb

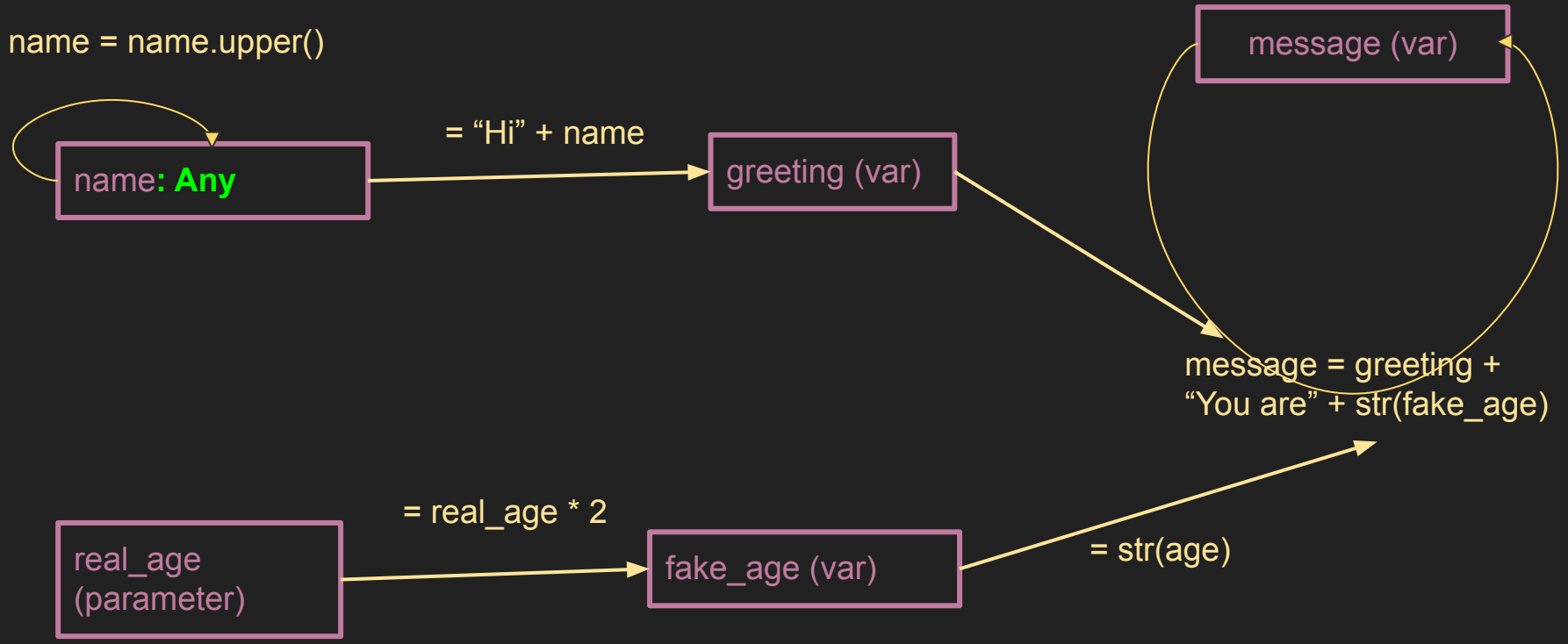
It reads your code line by line and follows some simple type-checking rules

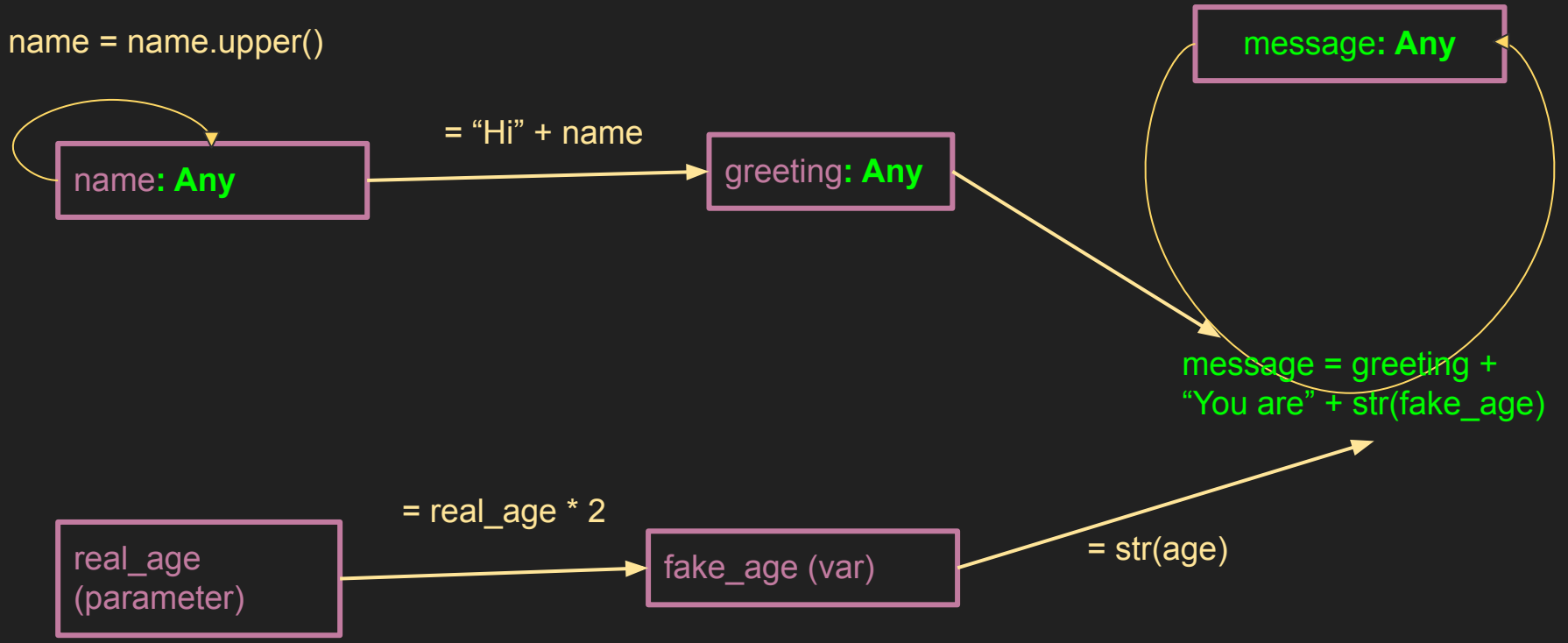
When you get a type error there is no big mystery. One of the type-checking rules has found an inconsistency

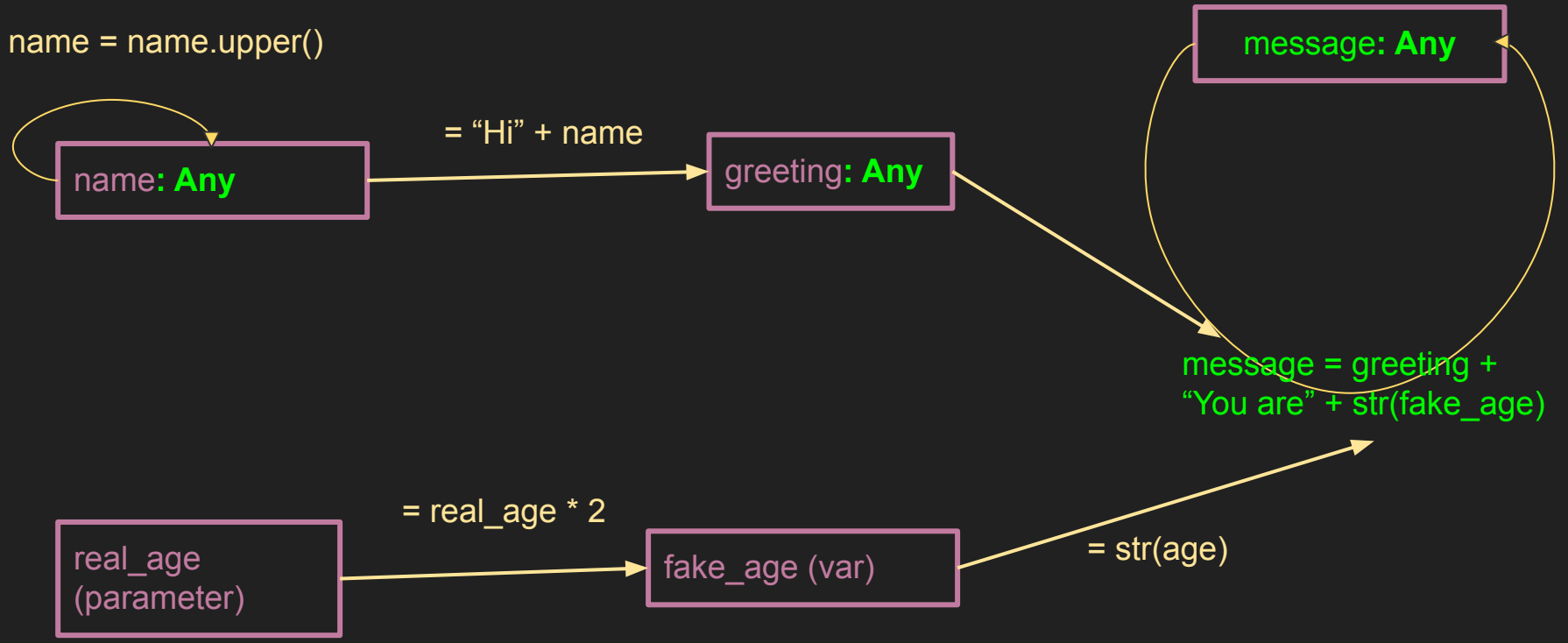
Any aka: *Dumb and Dumber*

Both Python and Typescript offer the user the ability to assign an any type

When you use `any` you are saying that your function/variable **really can** take the form of any possible type. That means that the type checker cannot find any discrepancies!


name = name.upper()
name: Any
= “Hi” + name
greeting (var)
message (var)
message = greeting +
“You are” + str(fake_age)
real_age
(parameter)
= real_age * 2
fake_age (var)
= str(age)


name = name.upper()
name: Any
= “Hi” + name
greeting: Any
message: Any
message = greeting +
“You are” + str(fake_age)
real_age
(parameter)
= real_age * 2
fake_age (var)
= str(age)



Congrats! You just passed the type check for ***message: str*** …

Don’t use Any unless you absolutely have to

```
const foo = (x: string): number=> {...}

const bar= (y: number): boolean => {...}
```

If the type hints are correct then whenever we call

bar(foo(“hello”))

we are guaranteed by the type checker that we will get true or false in return

Don’t use Any unless you absolutely have to

```
const foo = (x: string): number=> {

        let z: any = oh no

        return z
}

const bar= (y: number): boolean => {...}
```

The Type checker will not complain when we happily call *bar(foo(“hello”))*
But we could get ***any*** thing in response

Union types: when one type doesn't rule them all

Often you have variables which can be one of several types.

Eg:

- Optional[str] is either a string or None
- number | number[] is either a number or an array of numbers

Generic types:

Generic types are to types as functions are to variables *ish*

Eg:

- T = TypeVar('T')
  Optional[T] is either type T or None

- type T = …
  T[] is an array of type T

Casting: aka lying to mypy/TS

In a perfect world, you would type everything 100% correctly. Sadly we do not live in a perfect world yet.

There are situations where this would require you to write a lot of extra code to get the right type checking and it isn’t worth it.

Some libraries that we use are not properly typed and we need to avoid leaking Anys from their code into our code!

What is a cast?

A cast is basically a kind of type annotation where you assert that a given value is a specific type. The type-checker will still check for consistency!

```
def divide(x: int, y: int) -> Optional[int]:
    return x // y if y else None
```

It ***might*** be ok to cast the result in contexts where y is never 0 (or is always 0)
Note that you can only cast to int or None

What is a cast?

A cast is basically a kind of type annotation where you assert that a given value is a specific type. The type-checker will still check for consistency!

```
T = TypeVar('T')
def divide(x: T, y: int) -> Optional[T]:
    return x // y if y else None
```

It ***might*** be ok to cast the result in contexts where y is never 0 (or is always 0)
Note that you can only cast to T or None

What is a cast?

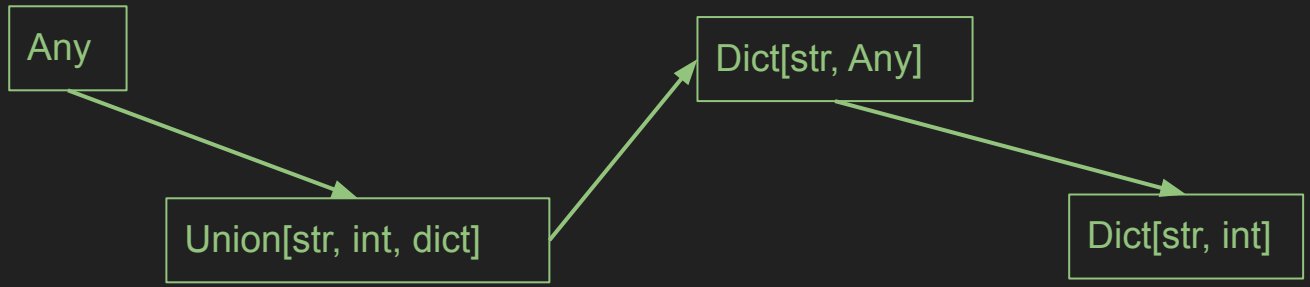
A cast is basically a kind of type annotation where you assert that a given value is a specific type. The type-checker will still check for consistency!

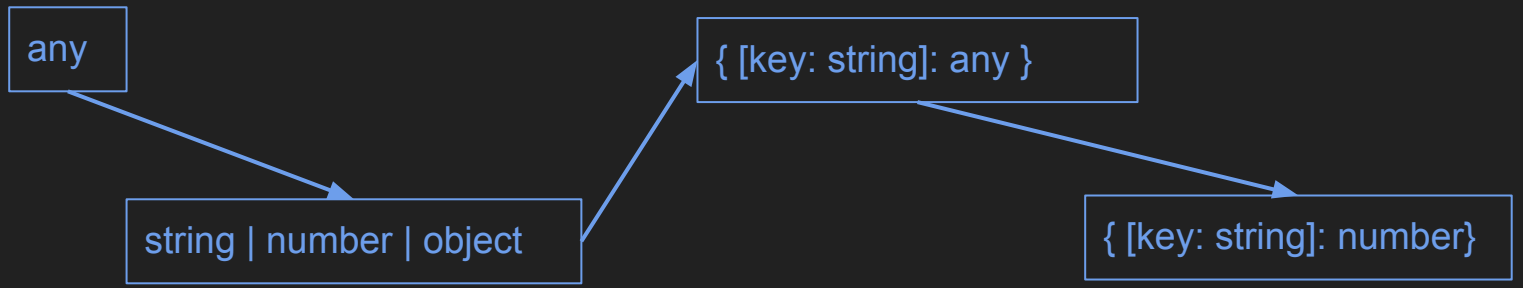
```
T = TypeVar('T')
def divide(x: T, y: int) -> Optional[T]:
    return x // y if y else None
```

It ***might*** be ok to cast the result in contexts where y is never 0 (or is always 0)
Note that you can only cast to T or None

What is a cast?

You have to get more specific with each cast

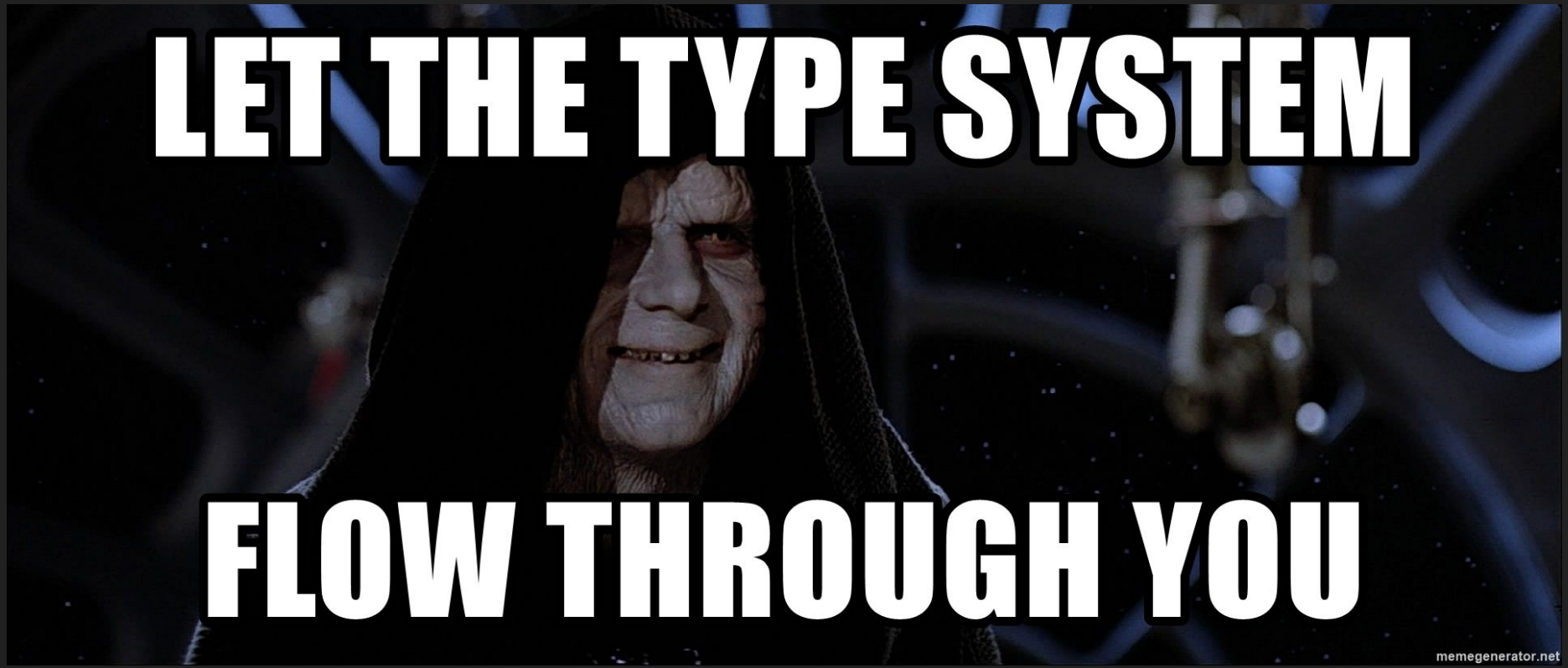




Don’t cast if you can avoid it

It’s not as bad as Any but our type-checking does still lose information.

Often rewriting code to avoid a cast can improve the code layout since it forces you to think in a type-safe way


LET THE TYPE SYSTEM
FLOW THROUGH YOU
memegenerator.net