item: str = “Type Hints in Python”

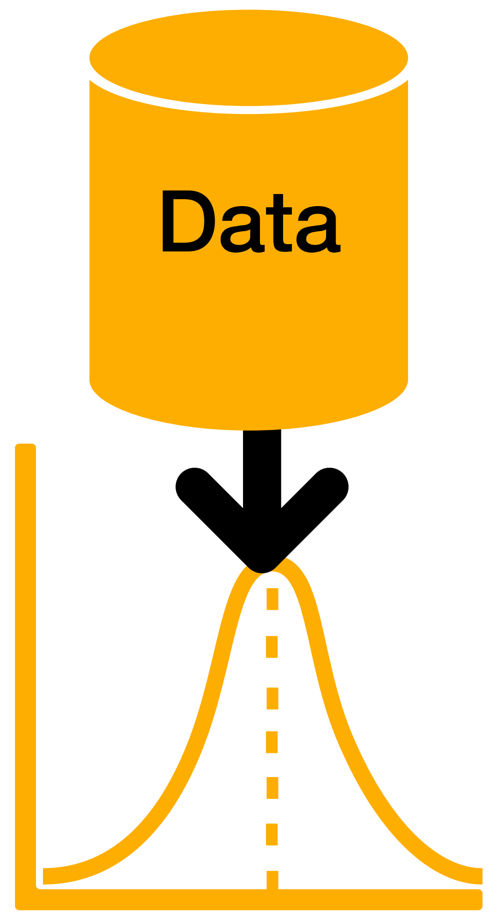
name: str = “Michael Colaresi”

Research is the Transformation of Information

- Input: data, text, images, etc
- Processing: aggregating, transforming, inferring
- Output: Conclusions inferences

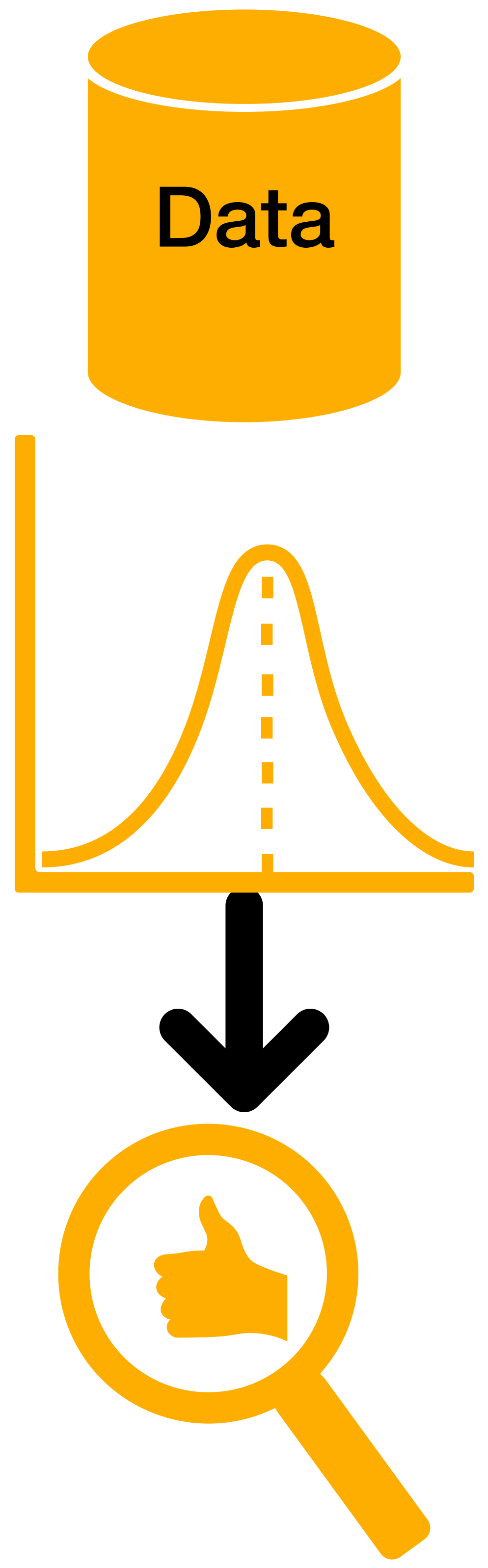
Research is the Transformation of Information

- Input: data, text, images, etc
- Processing: aggregating, transforming, inferring
- Output: Conclusions inferences

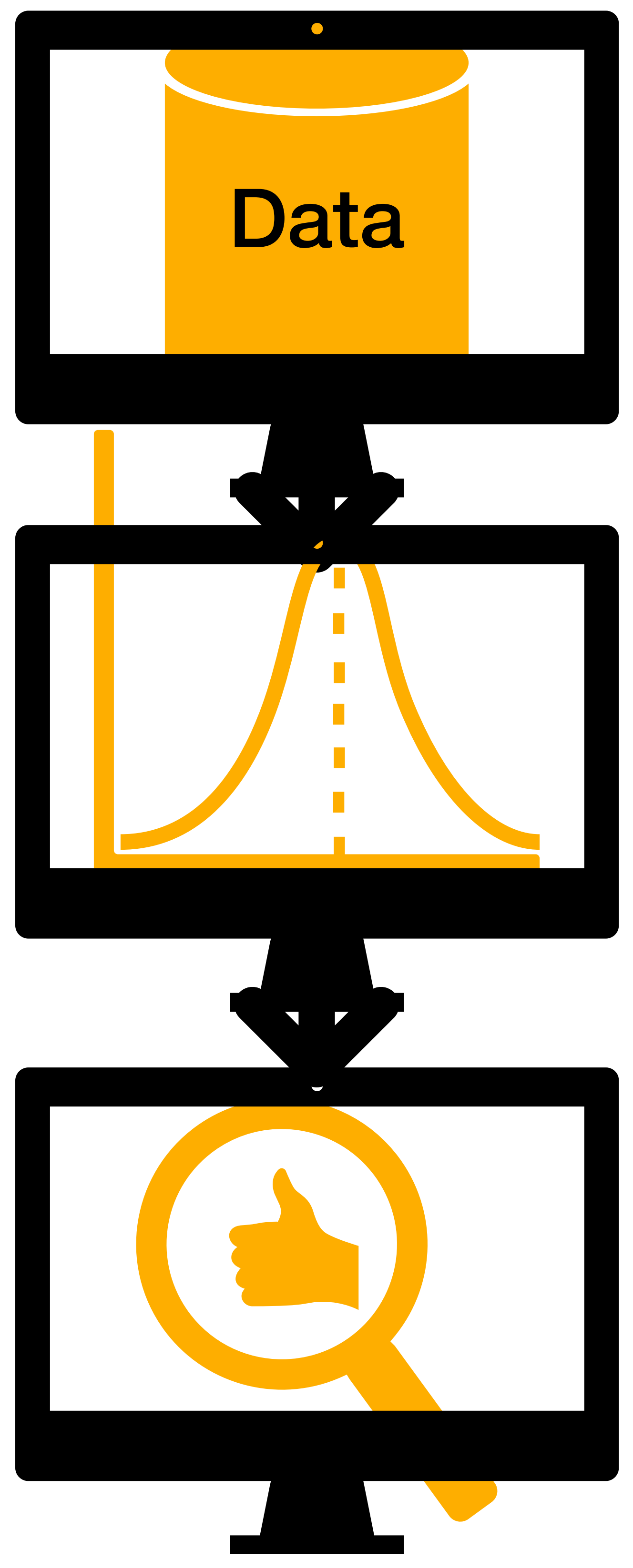


Research is the Transformation of Information

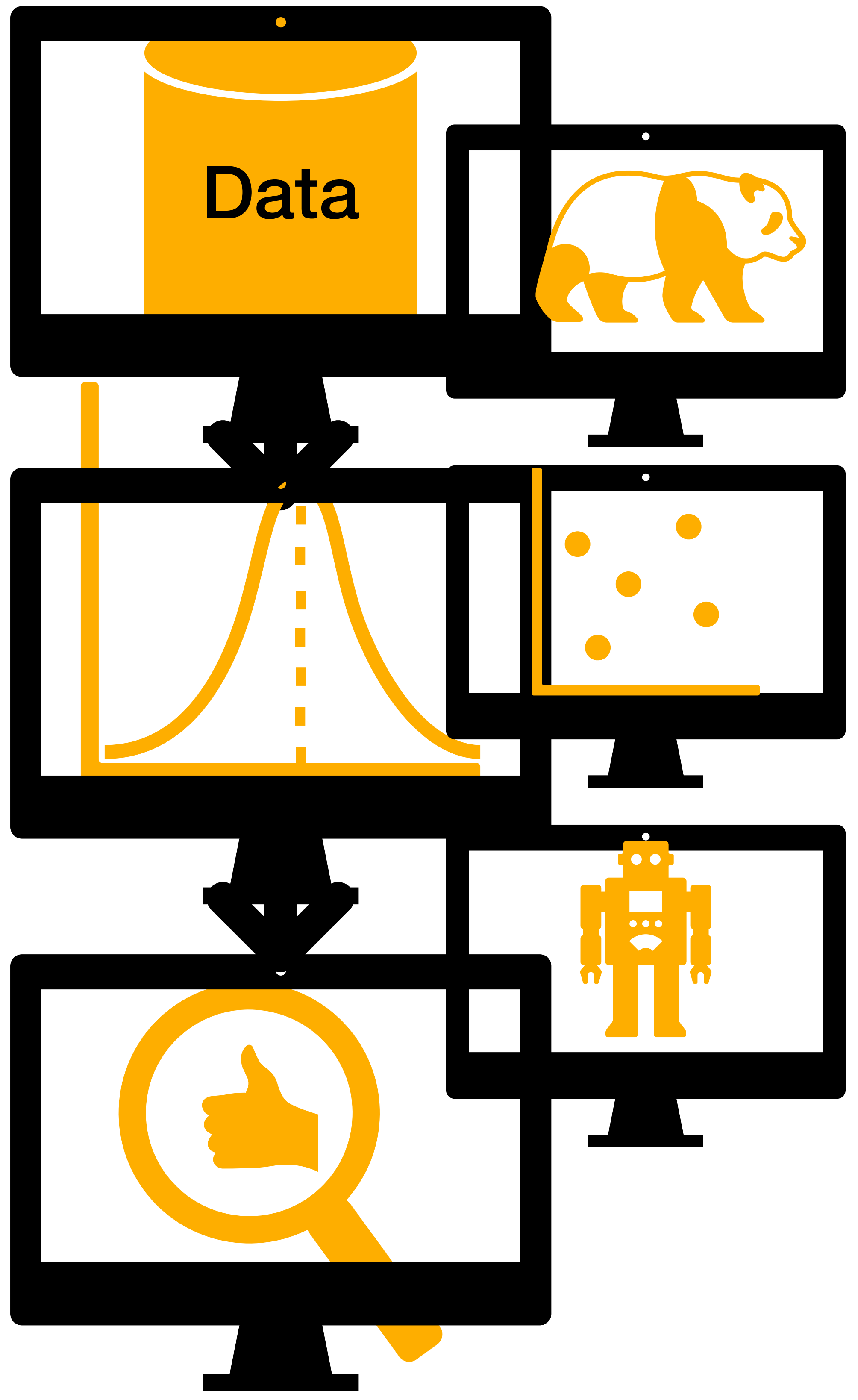
- Input: data, text, images, etc
- Processing: aggregating, transforming, inferring
- Output: Conclusions inferences



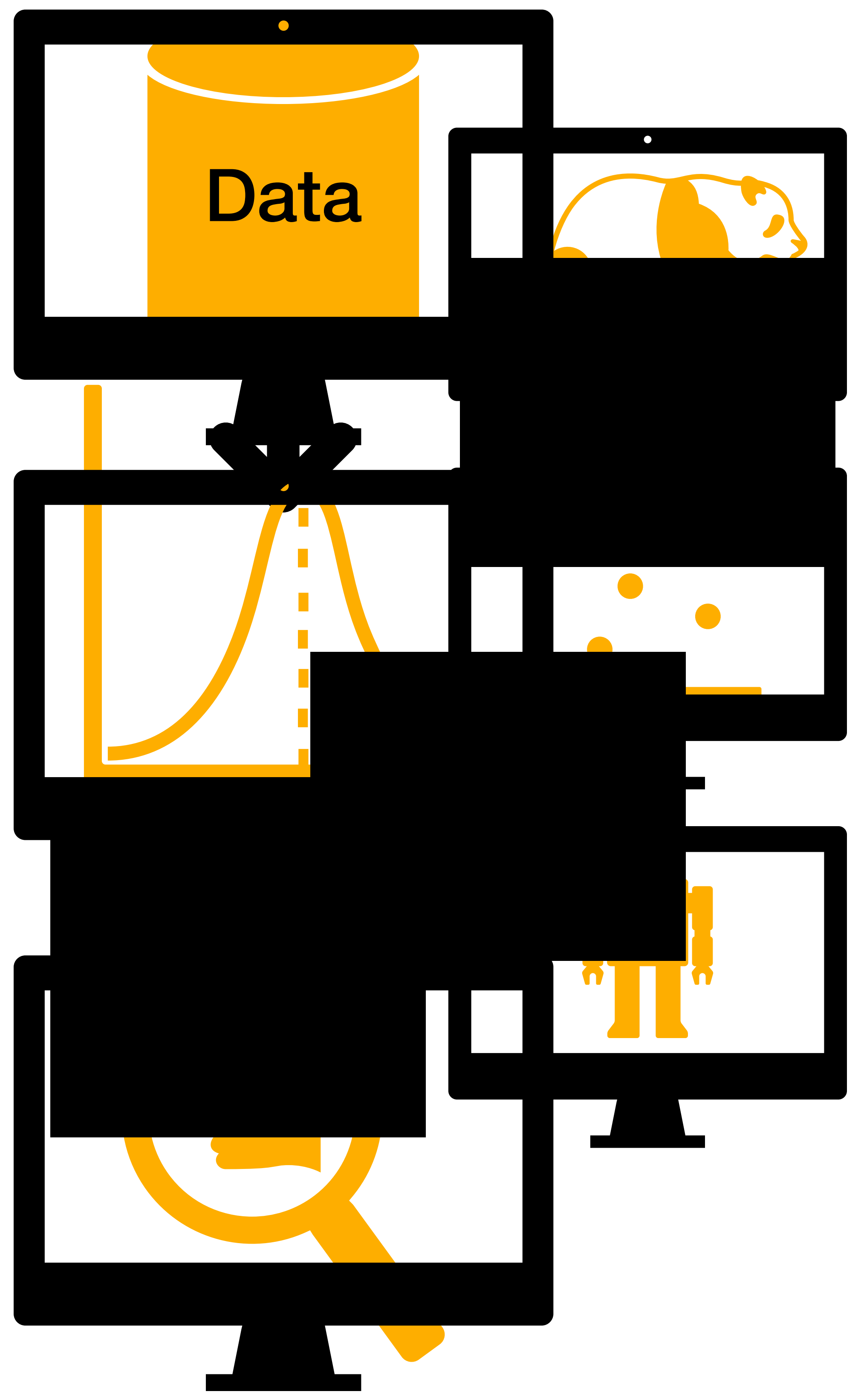
Transformations are (often) within your computer



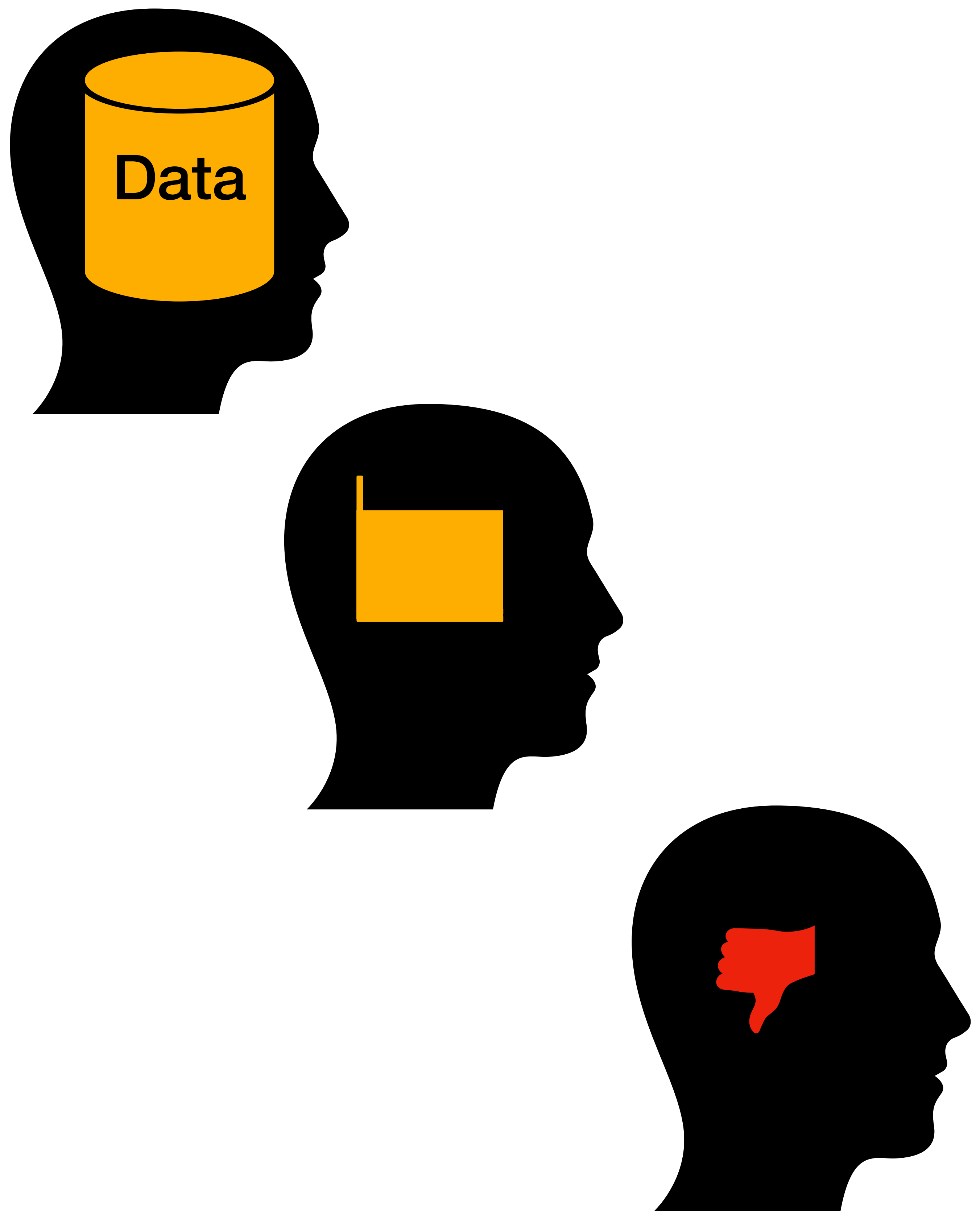
With many steps…



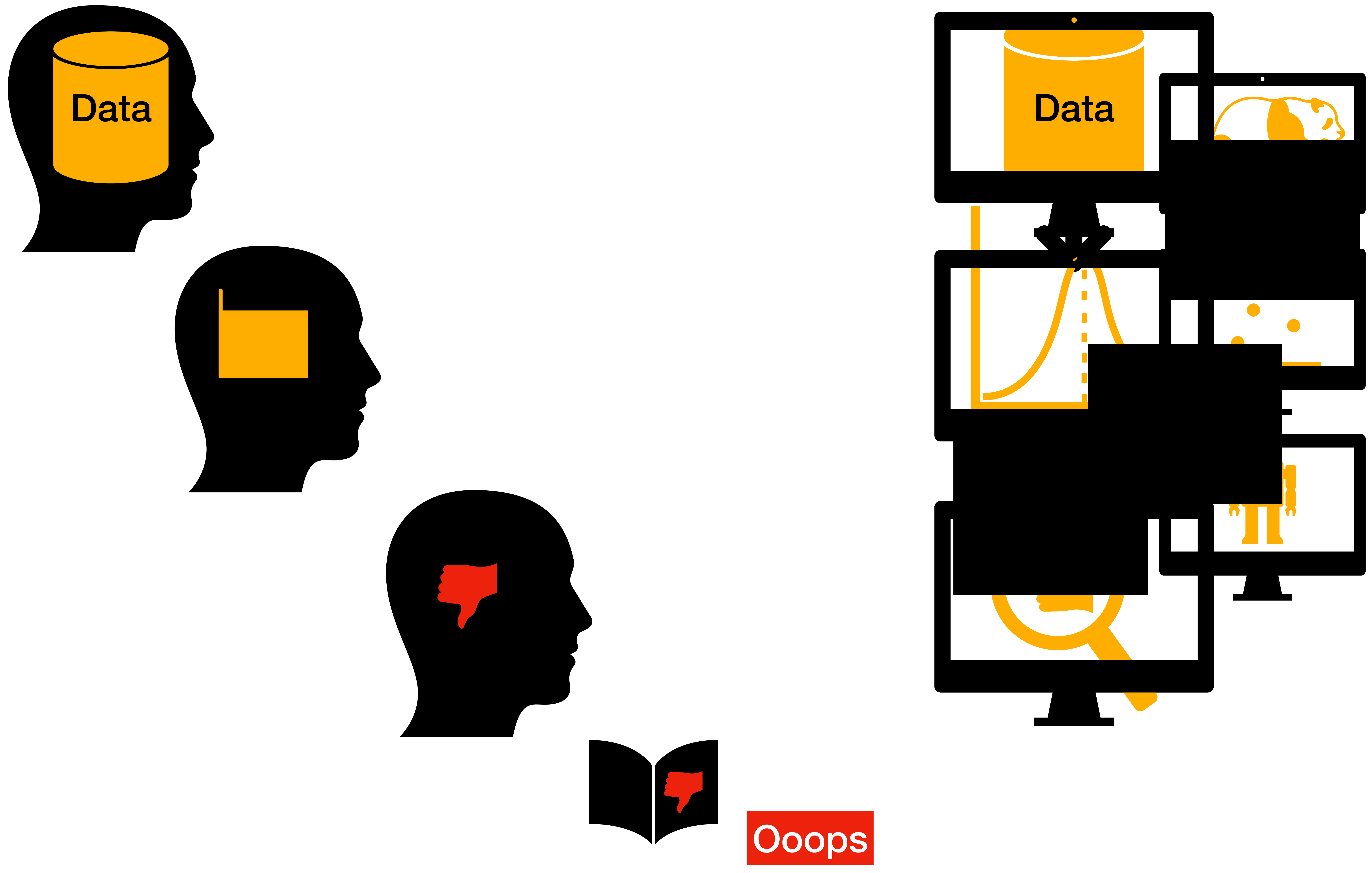
…only some of which we see/partially see



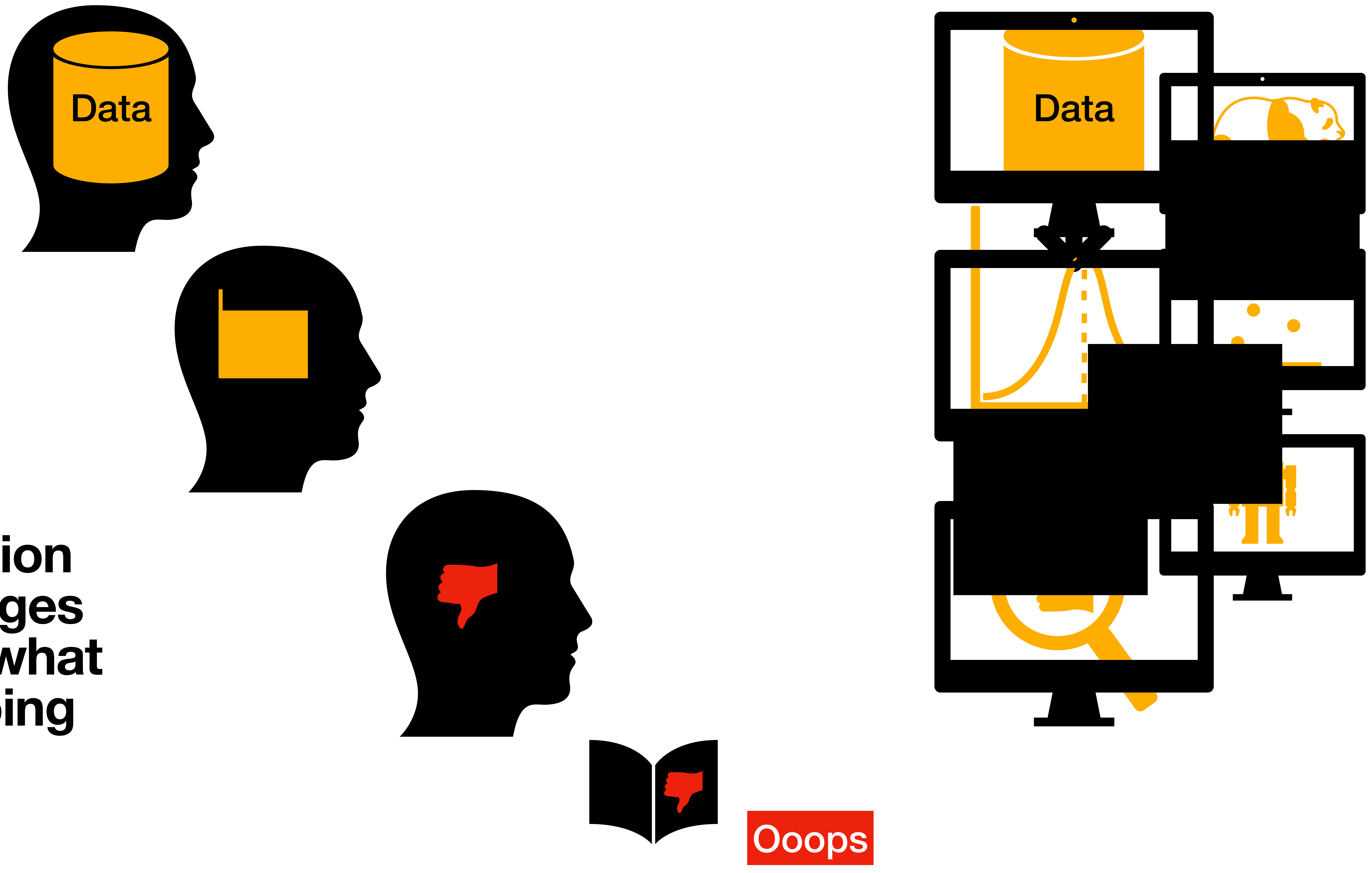
But we understand them in our heads



But we understand them in our heads



But we understand them in our heads



We have an obligation to reduce the changes of misinterpreting what our computer is doing

Common mistakes

- “+” for string (concatenation) or addition for numbers
- A function can return different types dep on conditions (eg float vs. None)
- Checking length of list that you think are floats, but are actually something else
- Arguments are passes from command line as strings;
  - need to be handled correctly (float(argv[2])
  - User needs to be nudged to pass the correct value
    - (python myS.py 1 “oops”
  - Passing arguments to functions and attributes to class constructors can be in wrong order

Common mistakes

IN ALL THESE CASES TYPE HINTING CAN CATCH PROBLEMS THAT LINTING MIGHT NOT

- "+" for string (concatenation) or addition for numbers
- A function can return different types dep on conditions (eg float vs. None)
- Checking length of list that you think are floats, but are actually something else
- Arguments are passes from command line as strings;
  - need to be handled correctly (float(argv[2])
  - User needs to be nudged to pass the correct value
    - (python myS.py 1 "oops"
  - Passing arguments to functions and attributes to class constructors can be in wrong order

Type Hinting Details

Python is and will remain dynamically typed

>A = 2

>A= “my cat”

No problem!

But there are some benefits to communicating types explicitly

PEP 484 (~2015)

Type Hinting Details

Don't get lost in terminology:
Types are just concepts (the idea of an int, vs. float, etc)
Everything is still binary all the way down

Dynamically typed = can change type of an object
Statically typed = an object gets a type when initialized
Duck typed = program tries to figure out what a variable is based on input
(walks like a duck….

Python is and will remain dynamically typed

>A = 2

>A= "my cat"

No problem!

But there are some benefits to communicating types explicitly

PEP 484 (~2015)

Type Hinting Details

The Typing module has objects that are complex types (like Lists, Tuples)

Type hinting is NOT checked by the python compiler!

But by a static checker, like a linter

mypy is the standard

Type Hinting Details

from typing import STUFF

The Typing module has objects that are complex types (like Lists, Tuples)

Type hinting is NOT checked by the python compiler!

But by a static checker, like a linter

mypy is the standard

Type Hinting Details

Give python hints as to the type of variables

objName: Type [assignment]

example: CONSTANT: float = 2.23

Without type hints you would have written CONSTANT = 2.23

def funcName(arg0: Type, arg2: Type, …) -> Type:

example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

example: CONSTANT: float = 2.23

: says we are giving type next

def funcName(arg0: Type, arg2: Type, …) -> Type:

example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

    example: CONSTANT: float = 2.23

Then we give the float type for this object

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

optional; could just set up type, objName: Type

objName: Type [assignment]

    example: CONSTANT: float = 2.23

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

example: CONSTANT: float = 2.23

Functions follow the same pattern with one twist

def funcName(arg0: Type, arg2: Type, …) -> Type:

example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

    example: CONSTANT: float = 2.23

argument/parameters are now the objects that are annotated

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

    example: CONSTANT: float = 2.23

Again use : to say a Type is next!

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

    example: CONSTANT: float = 2.23

Then give type

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

    example: CONSTANT: float = 2.23

Repeat for all arguments/parameters

def funcName(arg0: Type, arg2: Type, …) -> Type:

    example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

example: CONSTANT: float = 2.23

After args, but before the line ending “:”, use -> to denote that the RETURNED type is next

def funcName(arg0: Type, arg2: Type, …) -> Type:

example: def standardize_variable(x: float) -> float:

Type Hinting Details

objName: Type [assignment]

example: CONSTANT: float = 2.23

What type will be returned by function? Can be None

def funcName(arg0: Type, arg2: Type, …) -> Type:

example: def standardize_variable(x: float) -> float:

Type Hinting Details: Sequences, Maps

We will need to: from typing import List

note capital L

List

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

Type is List, so have to “from Typing import List”

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

Can give type of items in the list

listName: List[type, type,…] = [assignment]

If all items in list are same type, just give one value

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

Assignment is again optional

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

listName: List[type, type,…] = [assignment]

    example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

    example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

If there are different types within the list, use Union (see description in a few slides)

dictName: Dict[KeyType, ValType] = [assignment]

    Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

We will need to: from typing import Dict
note capital D

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

Dicts are maps

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

We will need to: from typing import Dict

note capital D

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

keys type are usually str or int

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

We will need to: from typing import Dict
note capital D

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

Value type could be anything

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

Type Hinting Details: Sequences, Maps

We will need to: from typing import Dict
note capital D

listName: List[type, type,…] = [assignment]

example: myList: List[float] = [2.23, 2.1, 2.456, 3.14]

example: myList2: List[Union[float, str, List[float]]] = [45.21, “bl”, [2.2]]

dictName: Dict[KeyType, ValType] = [assignment]

Example: myDict: Dict[str, int] = {‘apples’: 10, ’oranges’: 2}

values, again optional for type hinting

My Type of Type Hints

Remember module.object is how we access an object in a module, we can also from module import obj

- Functions are of type **typing.Callable([Arg0Type, Arg1Type, …], ReturnType)**
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type **typing.Iterable[Type]**
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type typing.Optional[Type]
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type **typing.Tuple[Item0Type, Item1Type, …]**
- Iterables are of type **typing.Iterable[Type]**
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type typing.Optional[Type]
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type **typing.Iterable[Type]**
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type typing.Optional[Type]
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type typing.Optional[Type]
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
    - List_of_ints = List[int]
- Can also have a type typing.Optional[Type]
    - This means that the object could return None, which is its own type
    - test: Optional[float] = None
        - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type **typing.Optional[Type]**
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
    - List_of_ints = List[int]
- Can also have a type **typing.Optional[Type]**
    - This means that the object could return None, which is its own type
    - test: Optional[float] = None
        - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

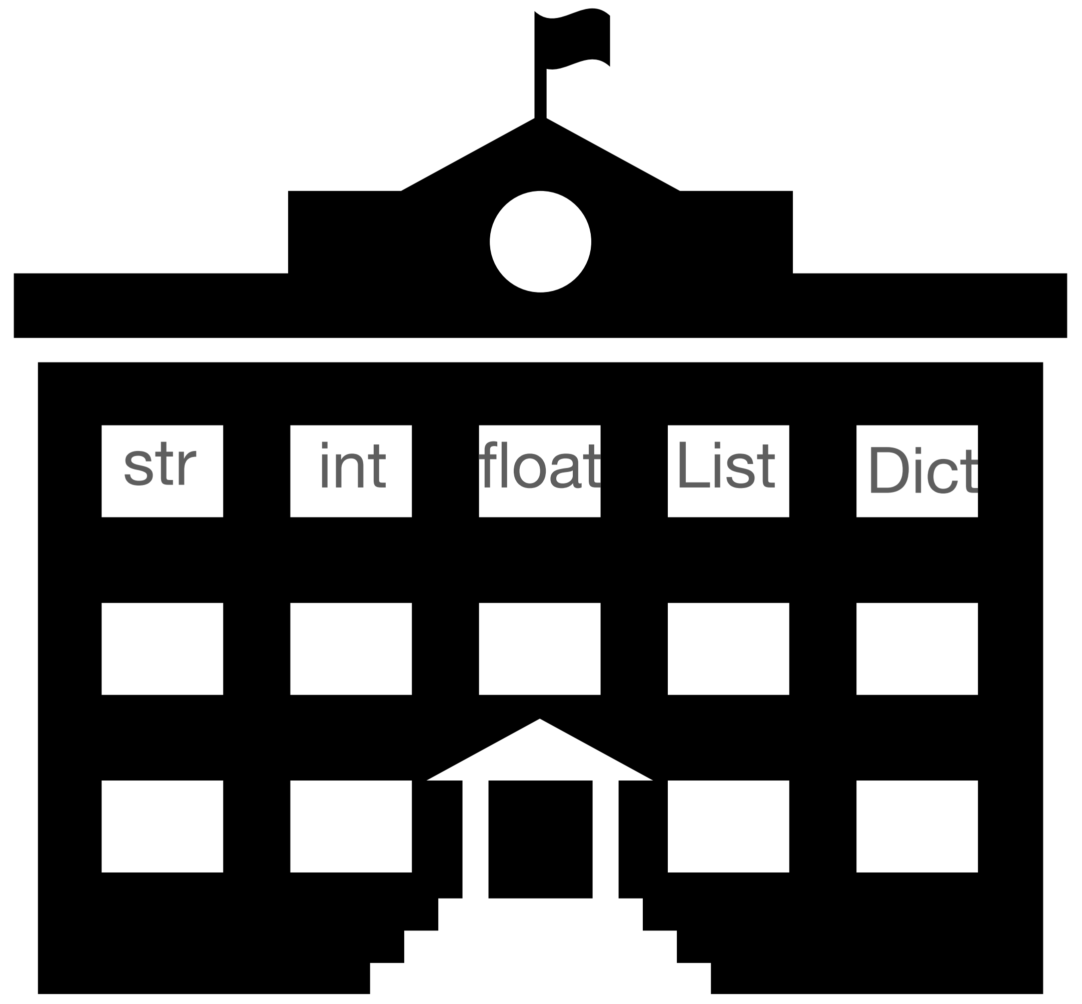
- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
  - List_of_ints = List[int]
- Can also have a type **typing.Optional[Type]**
  - This means that the object could return None, which is its own type
  - test: Optional[float] = None
    - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Functions are of type typing.Callable([Arg0Type, Arg1Type, …], ReturnType)
- Tuples are of type typing.Tuple[Item0Type, Item1Type, …]
- Iterables are of type typing.Iterable[Type]
- Can assign type hints to variables:
    - List_of_ints = List[int]
- Can also have a type **typing.Optional[Type]**
    - This means that the object could return None, which is its own type
    - test: Optional[float] = None
        - Passes mypy, even if None is replaced by a float as the program runs

My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
    - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
    - VAGUE: Any = “what?”
    - VAGUE = 3
    - VAGUE = [1, 2, “tree”]
        - None of these will trip an error in mypy



My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
  - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
  - VAGUE: Any = “what?”
  - VAGUE = 3
  - VAGUE = [1, 2, “tree”]
    - None of these will trip an error in mypy

My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
  - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
  - VAGUE: Any = “what?”
  - VAGUE = 3
  - VAGUE = [1, 2, “tree”]
    - None of these will trip an error in mypy

My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
    - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
    - VAGUE: Any = “what?”
    - VAGUE = 3
    - VAGUE = [1, 2, “tree”]
        - None of these will trip an error in mypy

My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
  - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
  - VAGUE: Any = “what?”
  - VAGUE = 3
  - VAGUE = [1, 2, “tree”]
    - None of these will trip an error in mypy

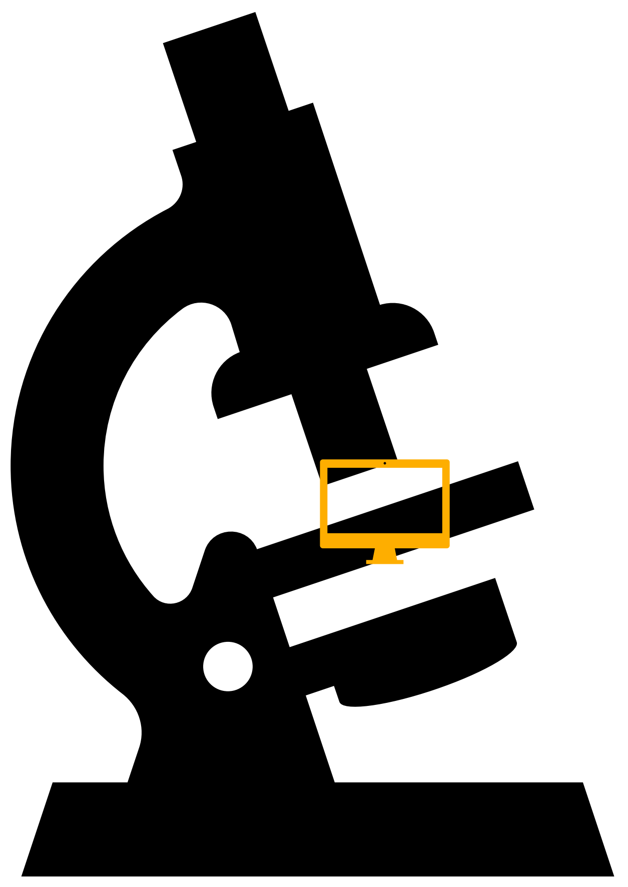
My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
  - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
  - VAGUE: Any = “what?”
  - VAGUE = 3
  - VAGUE = [1, 2, “tree”]
    - None of these will trip an error in mypy

My Type of Type Hints

- Can have more than 1 type possible with **typing.Union**
  - CONST: Union[float, Iterable[float], Callable[[Any], float]]
- If you want to skip typing, typing.Any will not fail in mypy
  - VAGUE: Any = “what?”
  - VAGUE = 3
  - VAGUE = [1, 2, “tree”]
    - None of these will trip an error in mypy

Lets look at a few applied examples



“+” for string (concatenation) or addition for numbers?

No type hints

Will do different things for different types, confusing
Not explicit, prone to mistakes

```
def add_it(arg1, arg2):
    return arg1 + arg2
```

Type hints

Type annotate arguments

```
def add_it(arg1: float, arg2: float) -> float:
    return arg1 + arg2
```

Type annotate arguments Return type

```
def concat_it(arg1: str, arg2: str) -> str:
    return arg1 + arg2
```

Be explicit about what function does to avoid mistakes.
With the name, and type annotation; even without a doctoring
.. still can figure out what is going on

function can return different types dep on conditions (eg float vs. None)

No type hints	Type hints

What is arg1? Are we counting characters? List items?
 	
`def count(arg1):`
`    return len(arg1)`
	Type annotate arguments Return type
	
	`def count_chars(arg1: str) -> int:`
	`    return len(arg1)`
	
	Type annotate arguments Return type
	
	`def count_list_items(arg1: List[str]) -> int:`
	`    return len(arg1)`
	
	Now it is clear, what we are trying to do
	Also, if your program returns None
	Because arg1 is empty,
	mypy will throw an error to help you debug

Arguments are passed from command line as strings

No type hints

Since arguments are passed as string, imagine if “False” passed as the the third arg into the script

```
LESS_POINTS = sys.argv[3]
if not LESS_POINTS:
    print(“we win!”)
```

We would not win!, because the non-empty string “False” is True

Type hints

Type annotate variable

```
LESS_POINTS: bool
```

Type annotate function

```
def assign_bool(arg: str) -> bool:
    assert arg is in [“True”, “False”], “WRONG VALUES”
    if arg==“True”:
        return True
    elif arg==“False”:
        return False

LESS_POINTS = assign_bool(sys.argv[3])
if not LESS_POINTS:
    print(“we win!”)
```

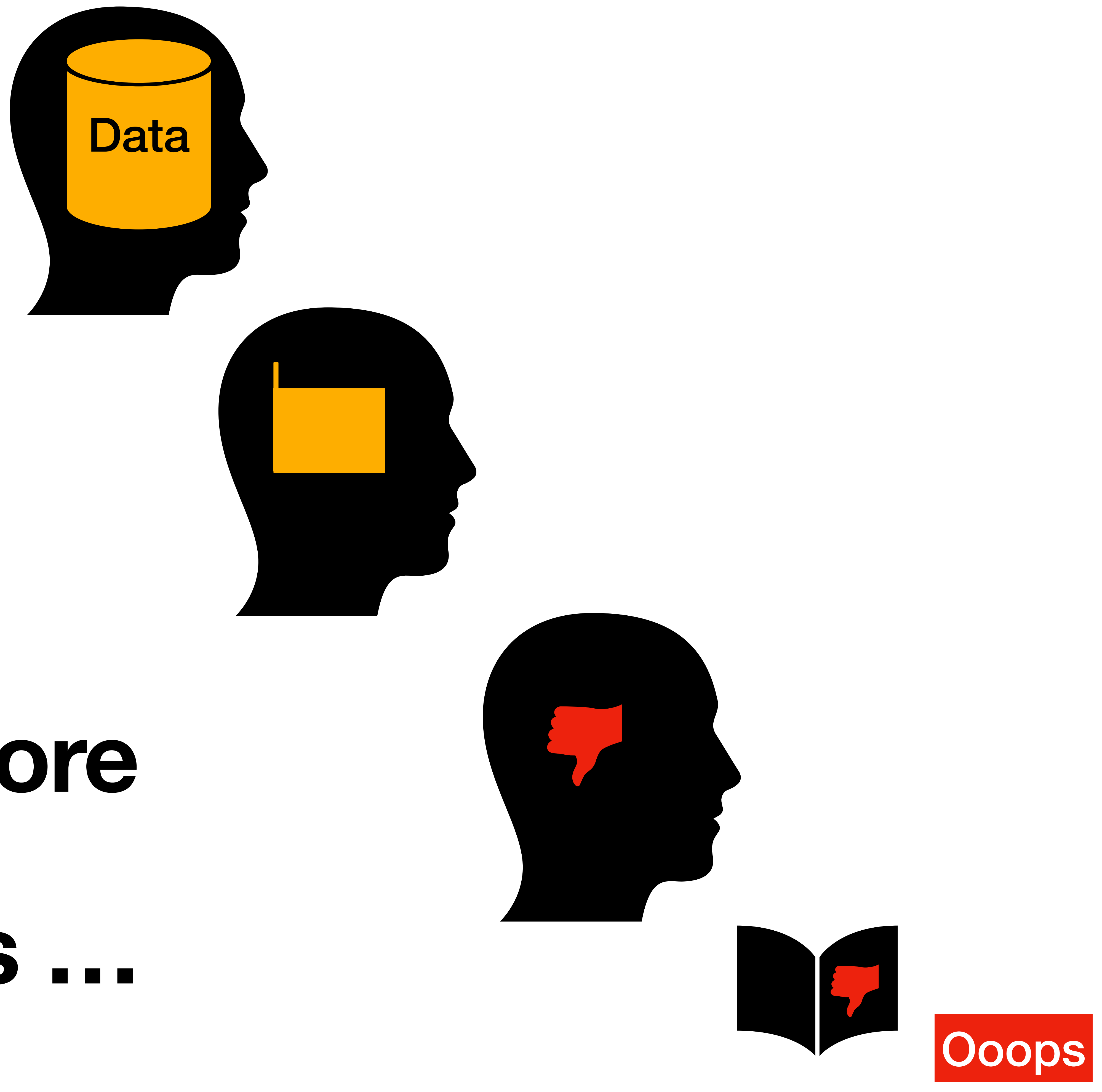
Now, if we do not give temp a bool it will fail in mypy and in our assign_bool function
Passing in argv[3] as “False” will now trigger “we win!”

Arguments are passes from command line as strings

No type hints	Type hints

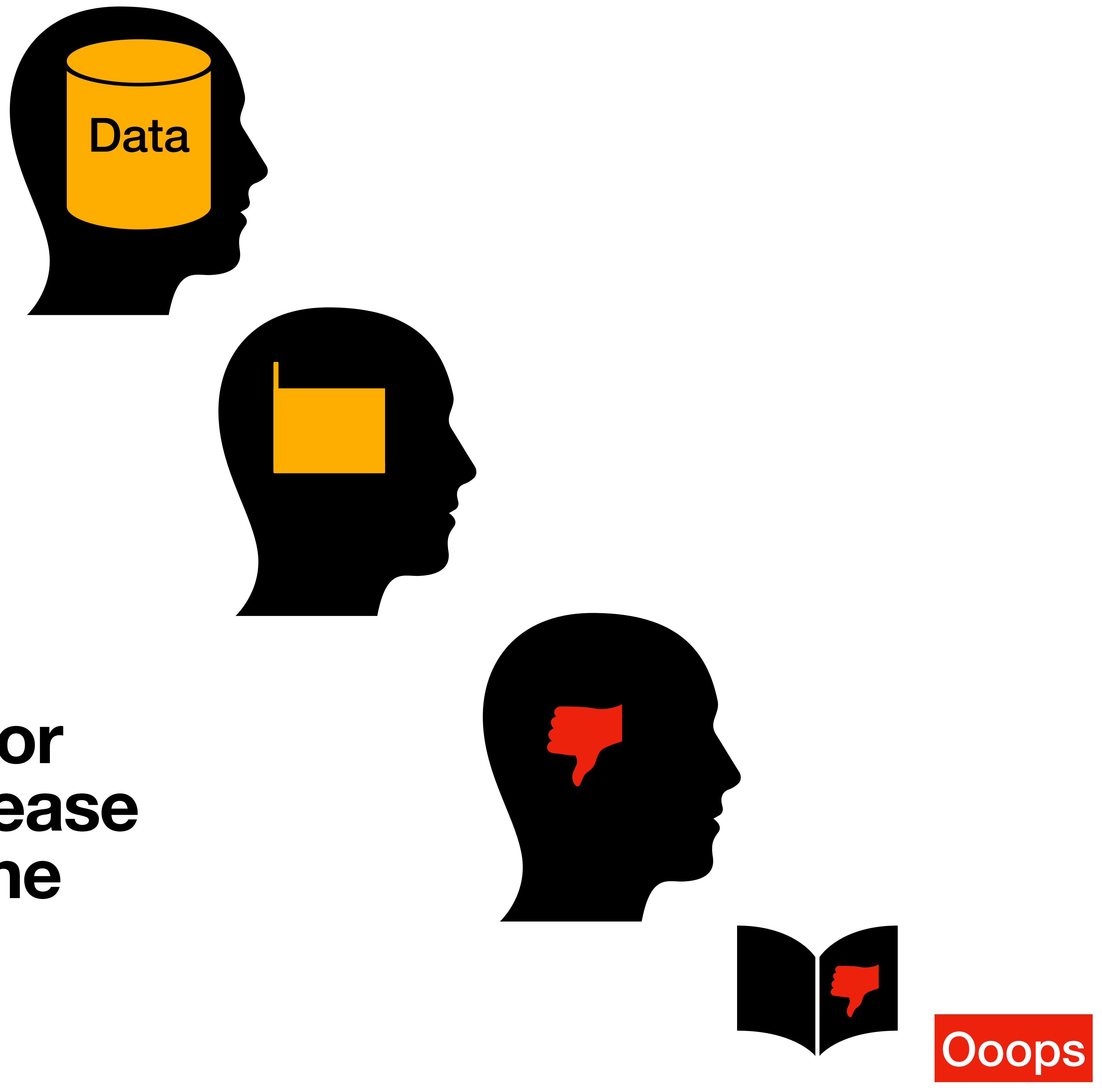
Since arguments are passed as string, imagine if “False” passed as the the third arg into the script
 	
LESS_POINTS = sys.argv[3]
if not LESS_POINTS:
    print(“we win!”)
 	
We would not win!, because the non-empty string “False” is True
	Type annotate variable
	    LESS_POINTS: bool
	Like legos: should fit together!
	Type annotate function
	    def assign_bool(arg: str) -> bool:
	        assert arg is in [“True”, “False”], “WRONG VALUES”
	        if arg==“True”:
	            return True
	        elif arg==“False”:
	            return False
	
	    LESS_POINTS = assign_bool(sys.argv[3])
	    if not LESS_POINTS:
	        print(“we win!”)
	
	Now, if we do not give temp a bool it will fail in mypy and in our assign_bool function
	Passing in argv[3] as “False” will now trigger “we win!”

Pass explicit information to catch mistakes



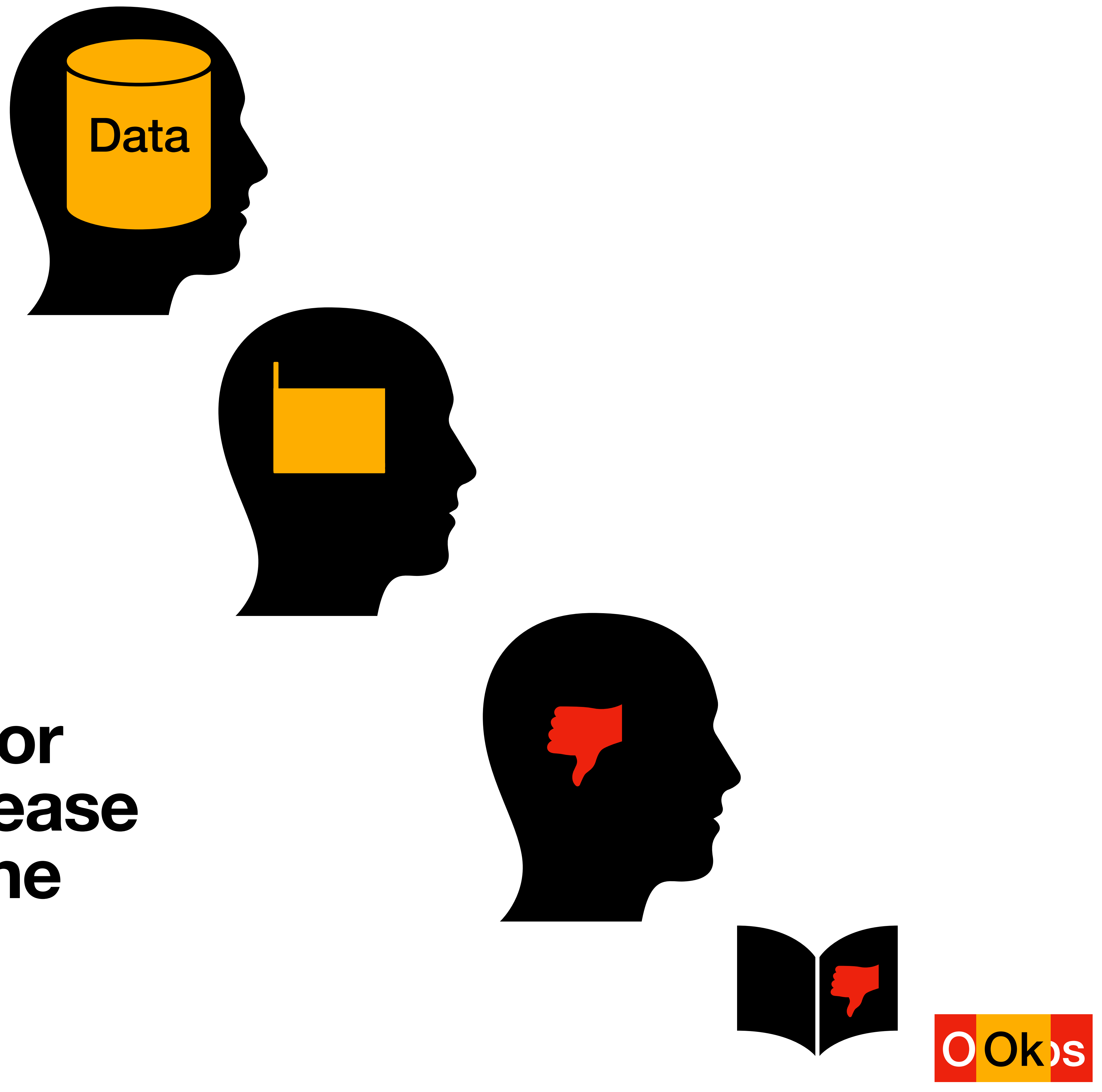
By seeing more of the assumptions …

Pass explicit information to catch mistakes



and laying traps for mistakes we increase the accuracy of the science

Pass explicit information to catch mistakes



and laying traps for mistakes we increase the accuracy of the science