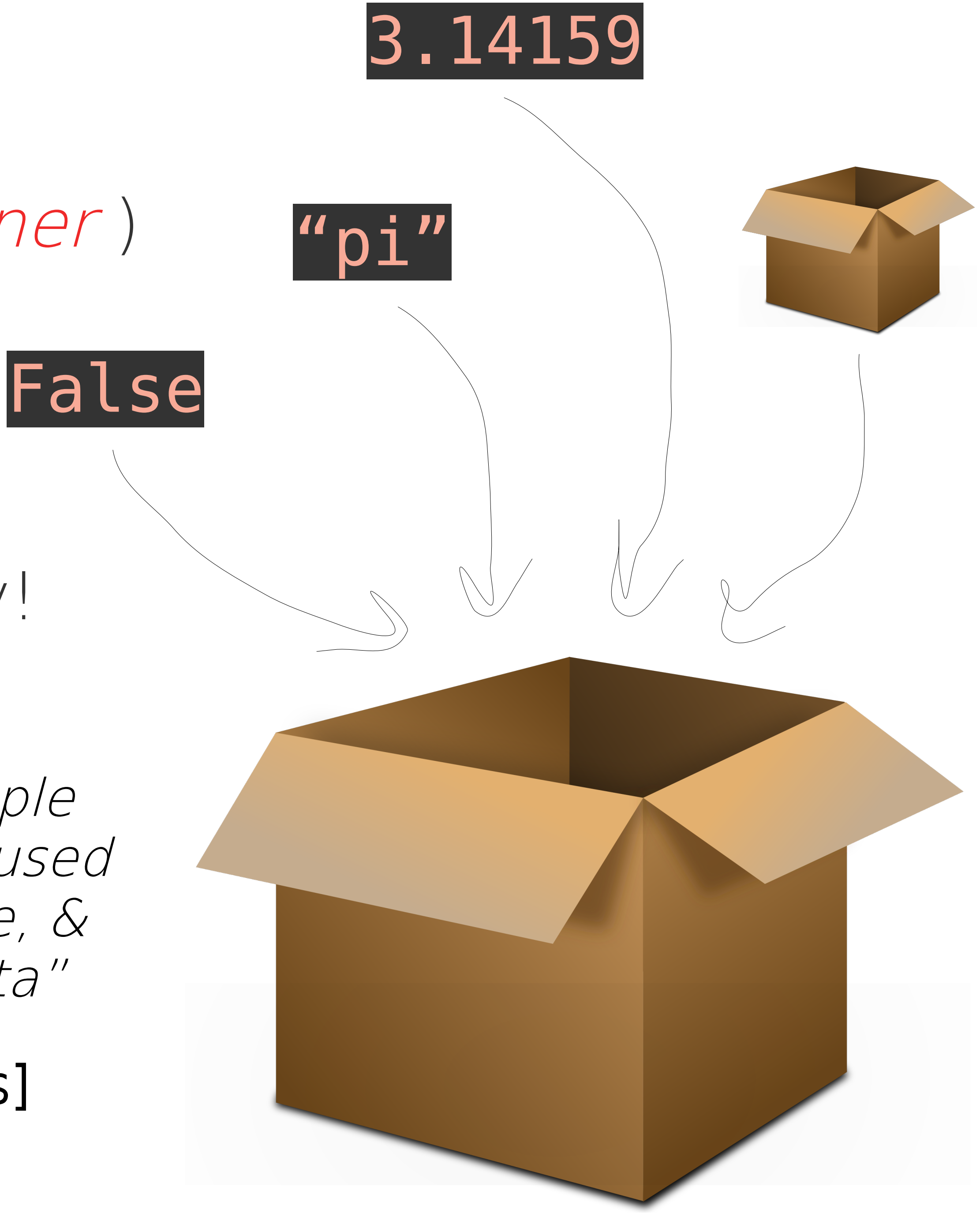
Collection choice by context

Met Office *Python Guild*, August ‘18

Sadie Bartholomew

Terminology

- *collection* (= *container*) data structures
- i.e. non-primitive
- not just the **`collections`** library!



"an object that groups multiple elements into a single unit… used to store, retrieve, manipulate, & communicate aggregate data"

[from Oracle Java Tutorials]

Aims

- review (some) collections available for Python
- highlight characteristics & variety
- summarise considerations for choosing wisely from the *Collections Zoo* for a given context
- **ultimately:** encourage us to think about containers in Python & how we use them

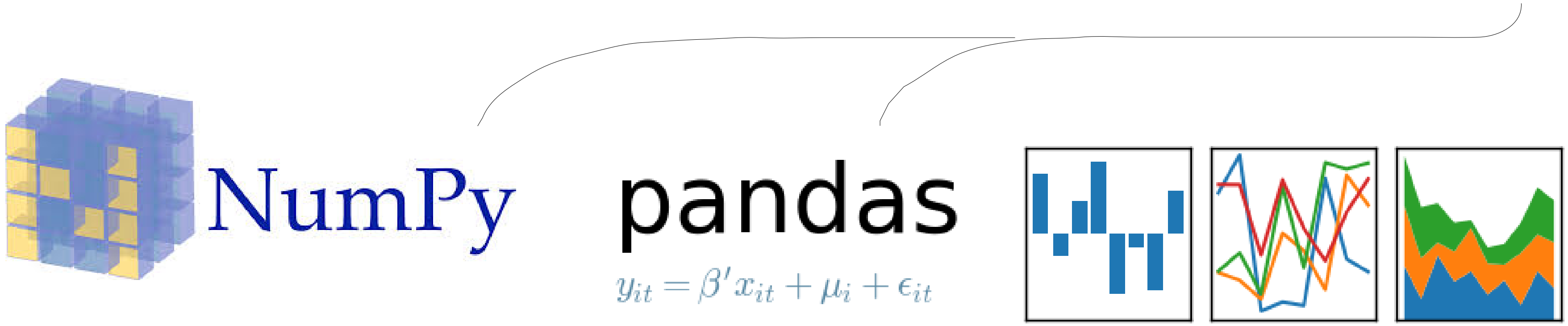
Scope (for this talk)

- *built-in* NB exclude strings as collections of characters
- *standard library* dedicated collections libraries:

  1) `collections`

  2) `array`

  NB exclude compound/serialisation data formats e.g. JSON
- established *external* numeric/data libraries (x2):



(Limited scope) catalogue

Built-in	From <*module*> import ...			
	collections	array	numpy	pandas
list	deque	array	array	
tuple	namedtuple()			
dict	defaultdict OrderedDict ChainMap Counter			Series DataFrame
set				
frozenset				

distinguish

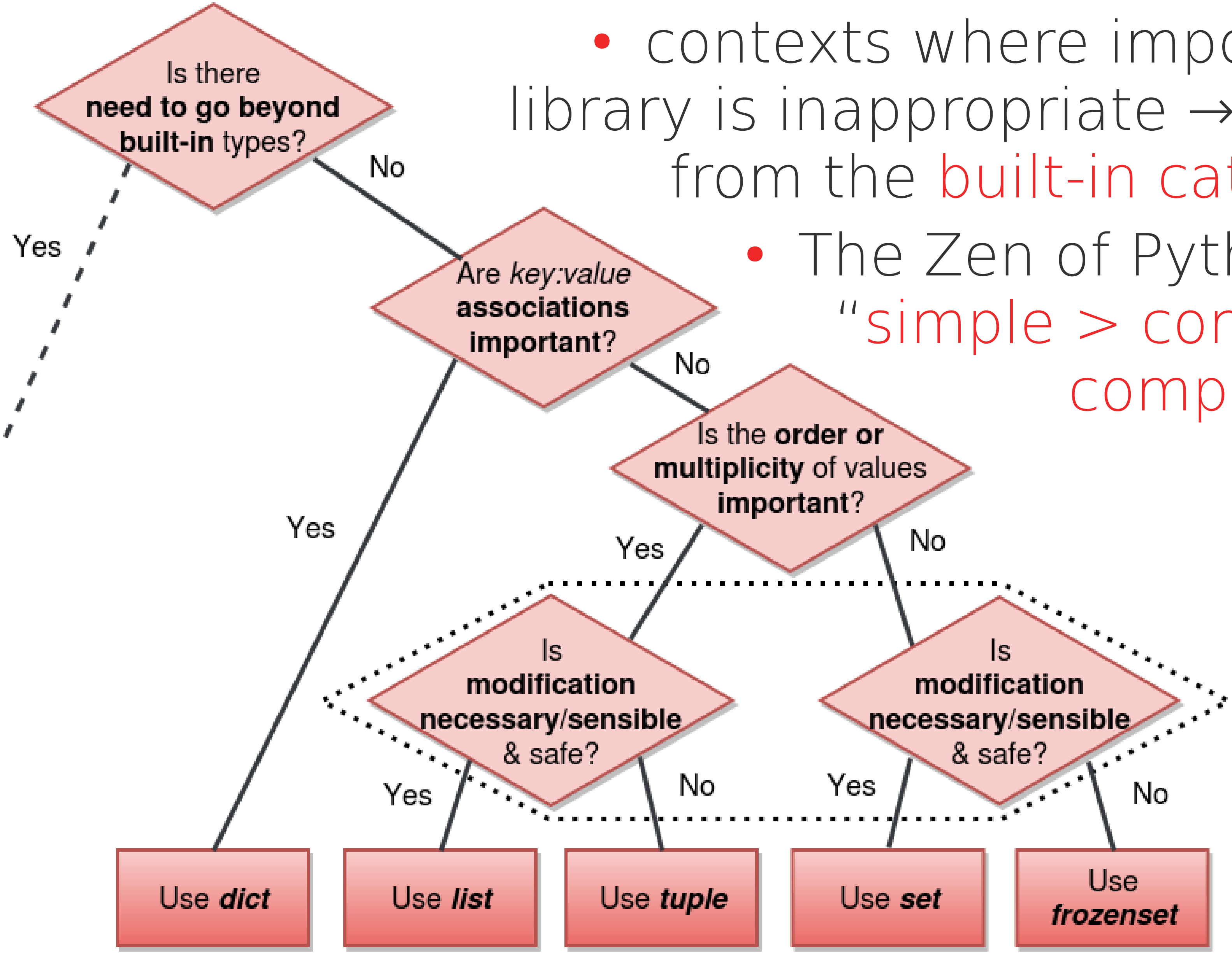
based on & extends

+ **UserDict** & **UserList**
wrappers (not covered)

- non-built-in collections to be outlined throughout

Considerations: Simplicity

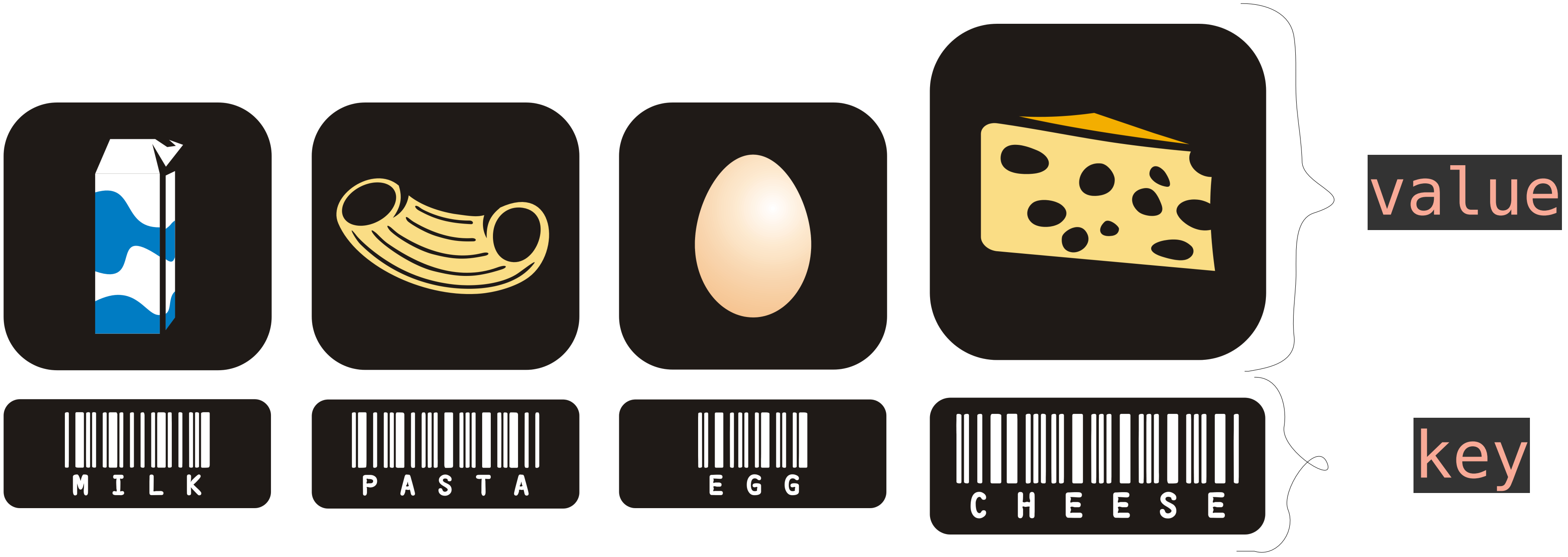
- contexts where importing a library is inappropriate → choice from the built-in catalogue
- The Zen of Python #3: "simple > complex > complicated"



Is there **need to go beyond built-in** types?
No
Yes
Are *key:value* **associations important**?
No
Yes
Is the **order or multiplicity** of values **important**?
Yes
No
Is **modification necessary/sensible** & safe?
Is **modification necessary/sensible** & safe?
Yes
No
Yes
No
Use ***dict***
Use ***list***
Use ***tuple***
Use ***set***
Use ***frozenset***

Considerations: Associations

- to associate, or *not to* associate (your values with keys)? keys essential as distinct mappings
- but… also consider as descriptors/labels:
  - require extra verbosity & memory space (but)
  - minimise lookup time & improve readability.

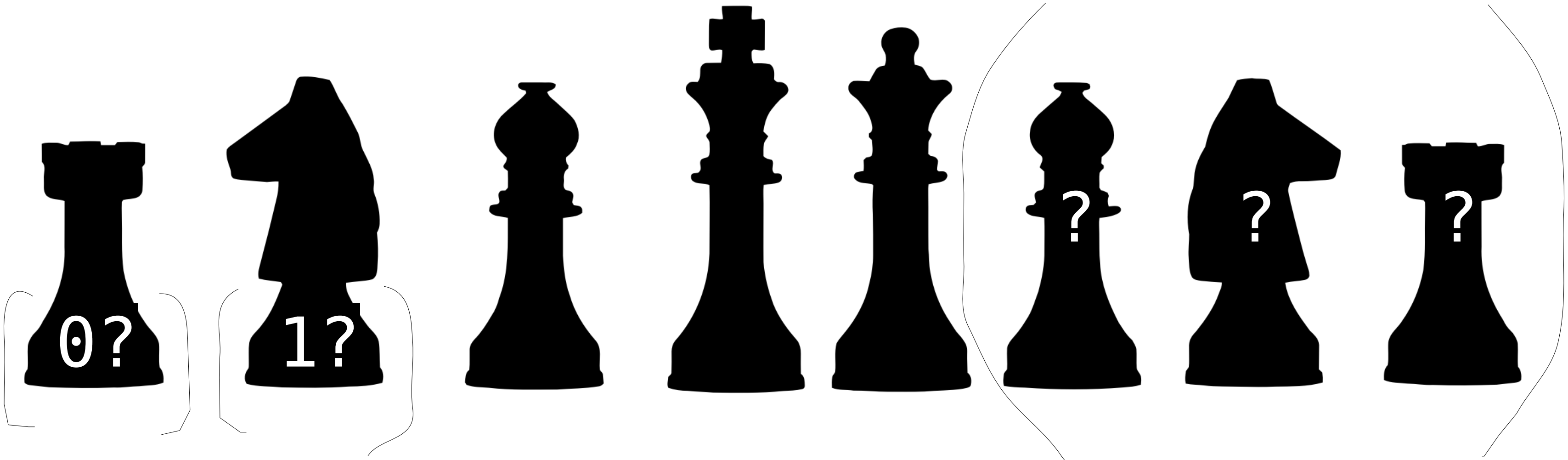


Considerations: Mutability

- to bring about a modification:
  - → *directly alter* the object instance (mutable)
  - → create a *new* object as a *revised copy* (immutable)
- mutable data structures offer:
  - → increased flexibility & avoid expensive copying for alterations (but)
  - → not thread-safe, no identity by state & referencing requires defensive copying

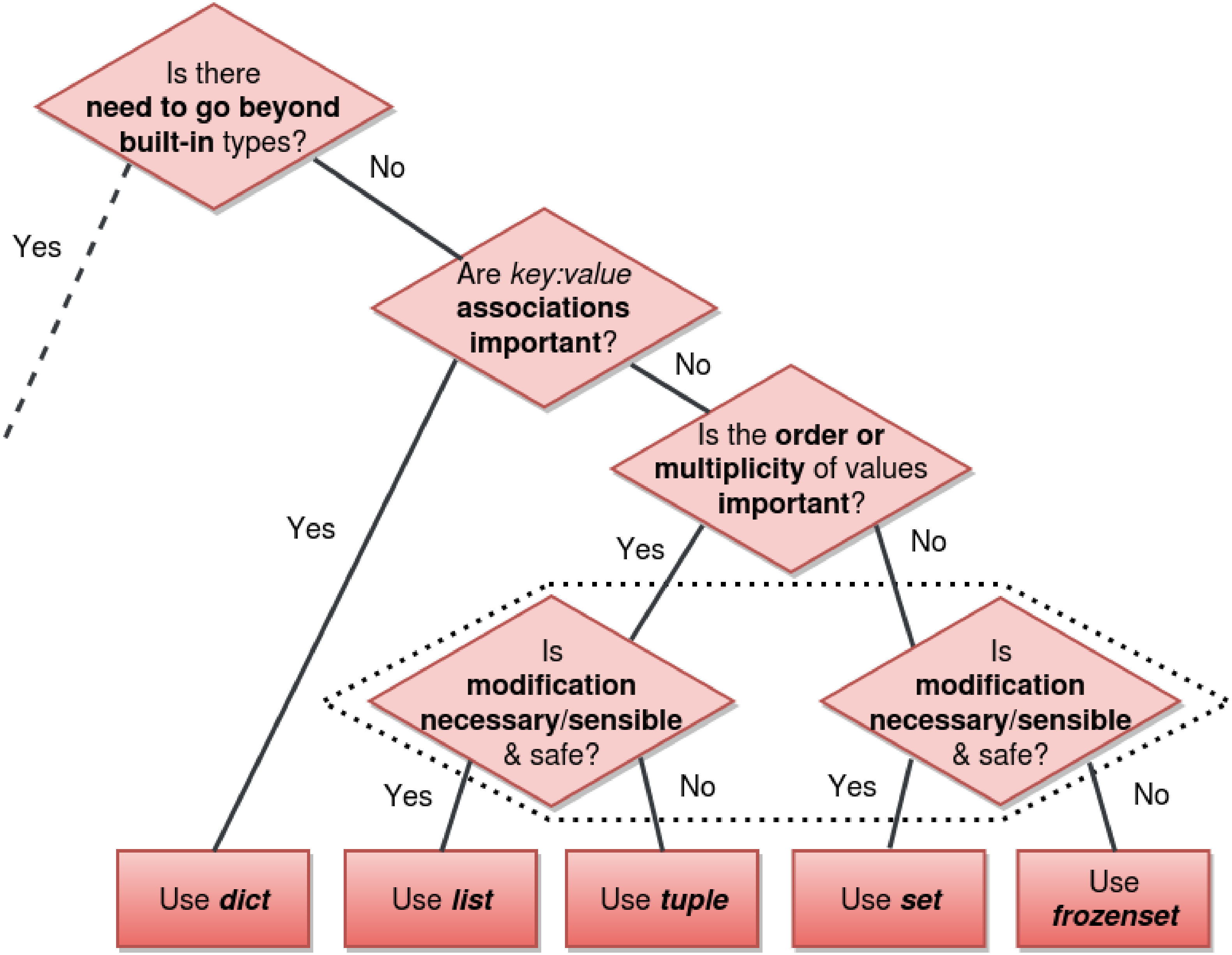
Considerations: Order & Tally

- element order: *position* in the collection
- tally/multiplicity: identical element *count*



- if both irrelevant, consider using **`set`** (or **`dict`**): *unordered* with only *unique* elements (keys)
- `OrderedDict` remembers insertion order
- `Counter` is a **`dict`** storing counts as values

Considerations: *pause & recap*


Is there **need to go beyond built-in** types?
No
Yes
Are *key:value* **associations important**?
No
Yes
Is the **order or multiplicity** of values **important**?
Yes
No
Is **modification necessary/sensible** & safe?
Yes
No
Is **modification necessary/sensible** & safe?
Yes
No
Use ***dict***
Use ***list***
Use ***tuple***
Use ***set***
Use ***frozenset***

Considerations: Readability

- The Zen of Python #7: “Readability counts.”
- difficult to keep track of field identity via index for *long* or *nested* structures or for *like* fields
- named fields & **`dict`** keys make code easier to read & comprehend (self-documenting)

```python
from collections import namedtuple
Box = namedtuple('Box', 'height width length')
box_A = Box(height=10, width=12, length=14)
```

```python
box_A = (10, 12, 14)
```

compare to **`tuple`**

Considerations: Nesting

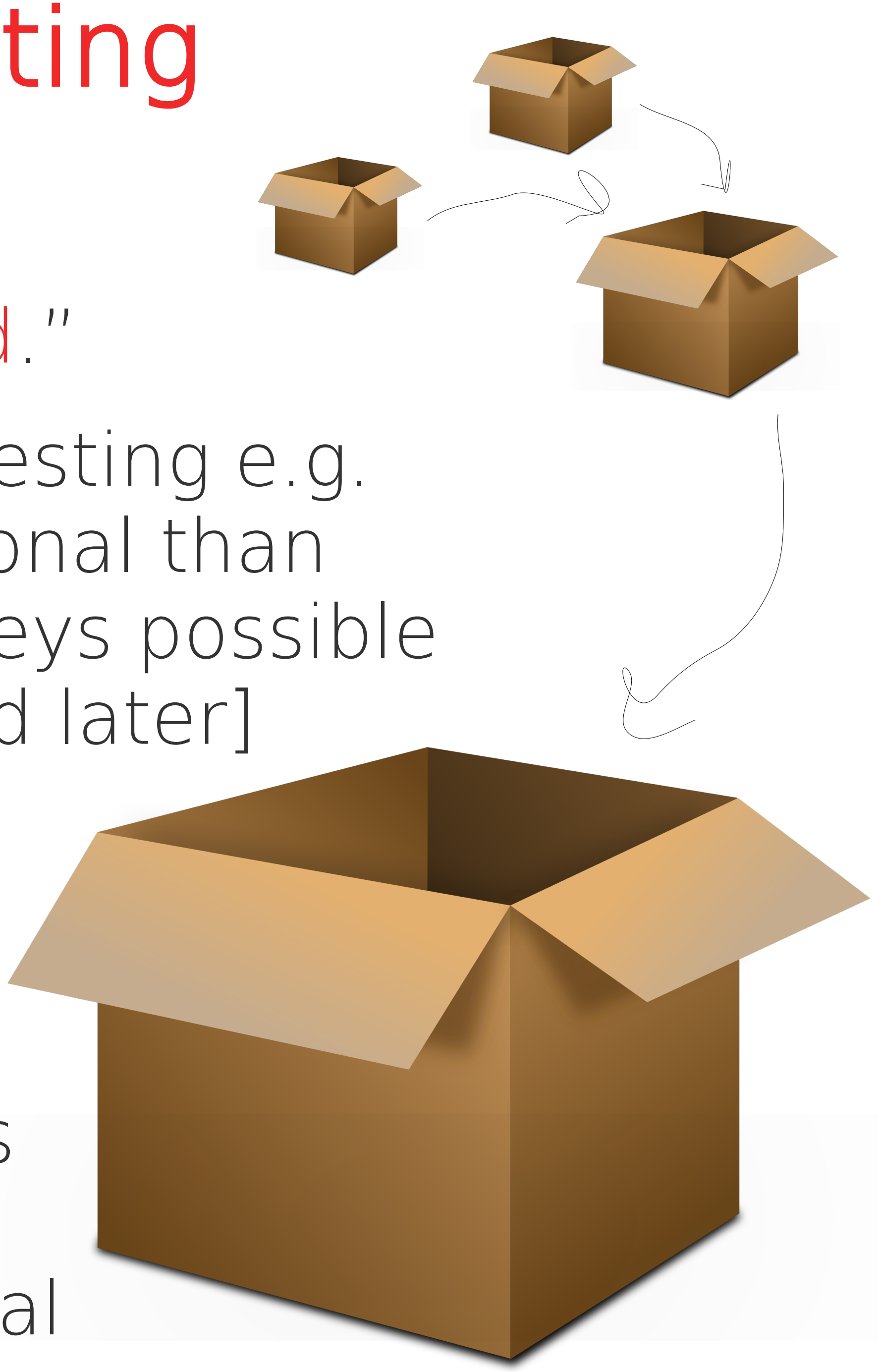
- The Zen of Python #5:

  "Flat is better than nested."

- consider alternatives to nesting e.g. **`numpy.array`** more functional than nested list & layering of keys possible in **`ChainMap`** [both outlined later]

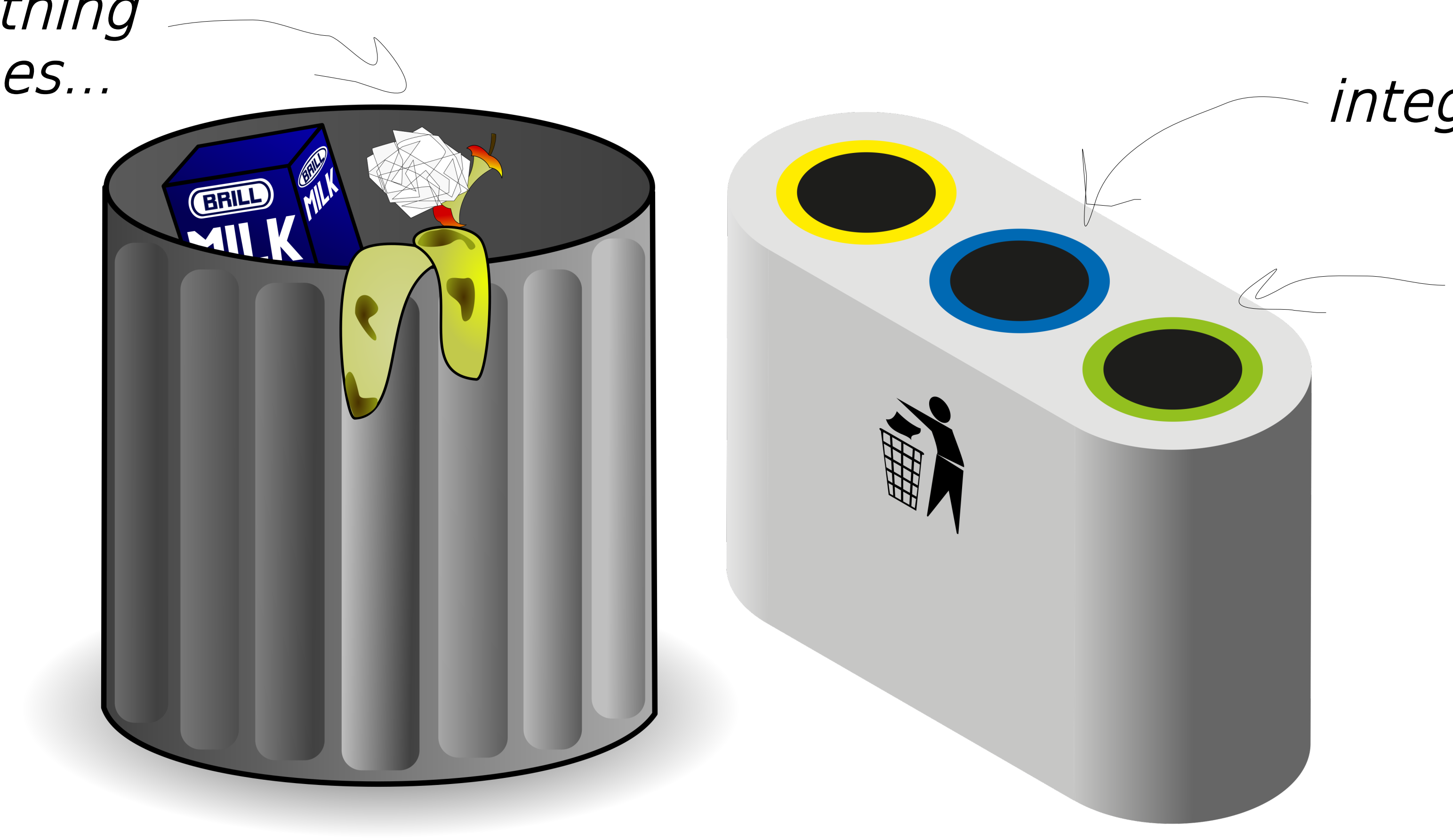
- (shallow) nesting sometimes justified

- for *top-level* choice, note **`set`** elements & **`dict`** keys cannot be mutable, restricting nesting potential



Considerations: Efficiency → Type

- are the data types to be collected uniform?
- `array.array` is *more* efficient by *restricting* element type, e.g. to character, **`int`** or **`float`**



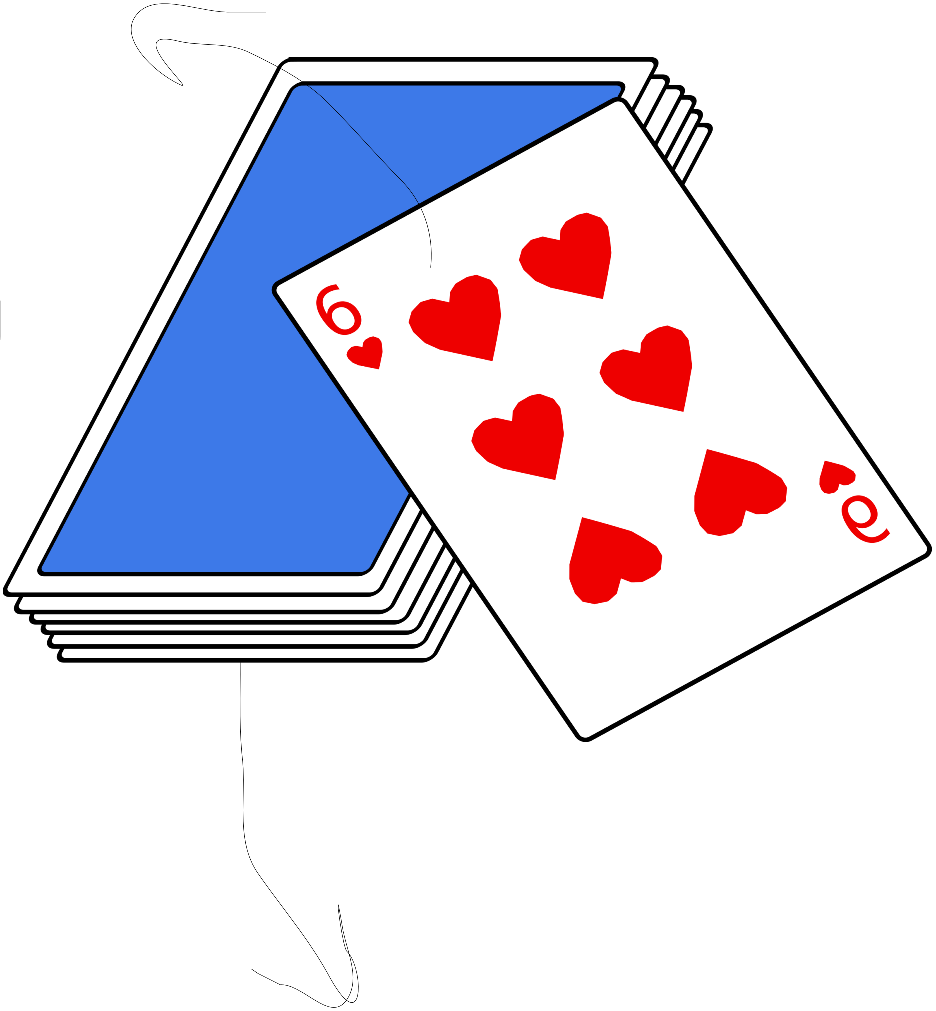
anything goes…

integers only

floats only

Considerations: Efficiency → Access

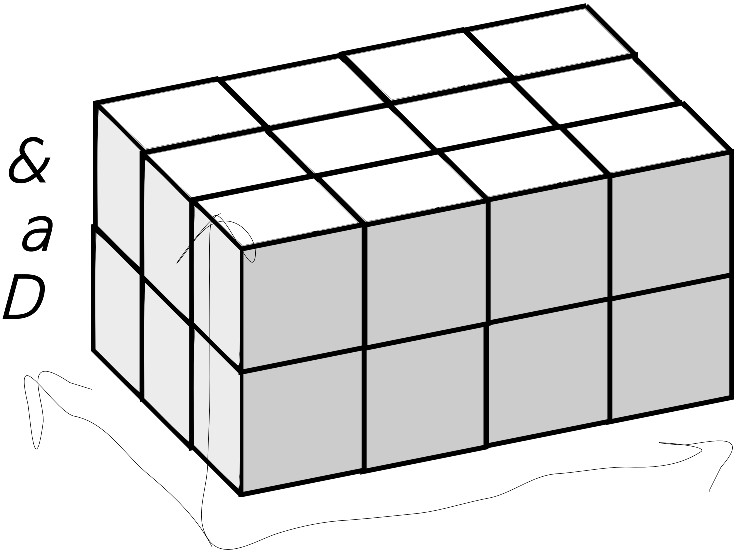
- `list` is fully flexible
- different operations e.g. copying, iterating over, element getting, setting & removing, have different *O(n)*, so consider what is important contextually
- `deque` is *more* efficient by being *less* flexible: can only add, extend & return from either end

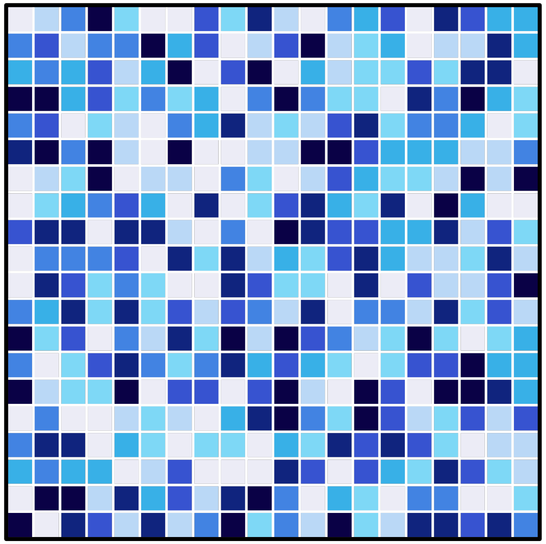


Considerations: Functionality

- external collections possess unique functionality

flexible dimension & shape, e.g. a 2 x 3 x 4 3D array



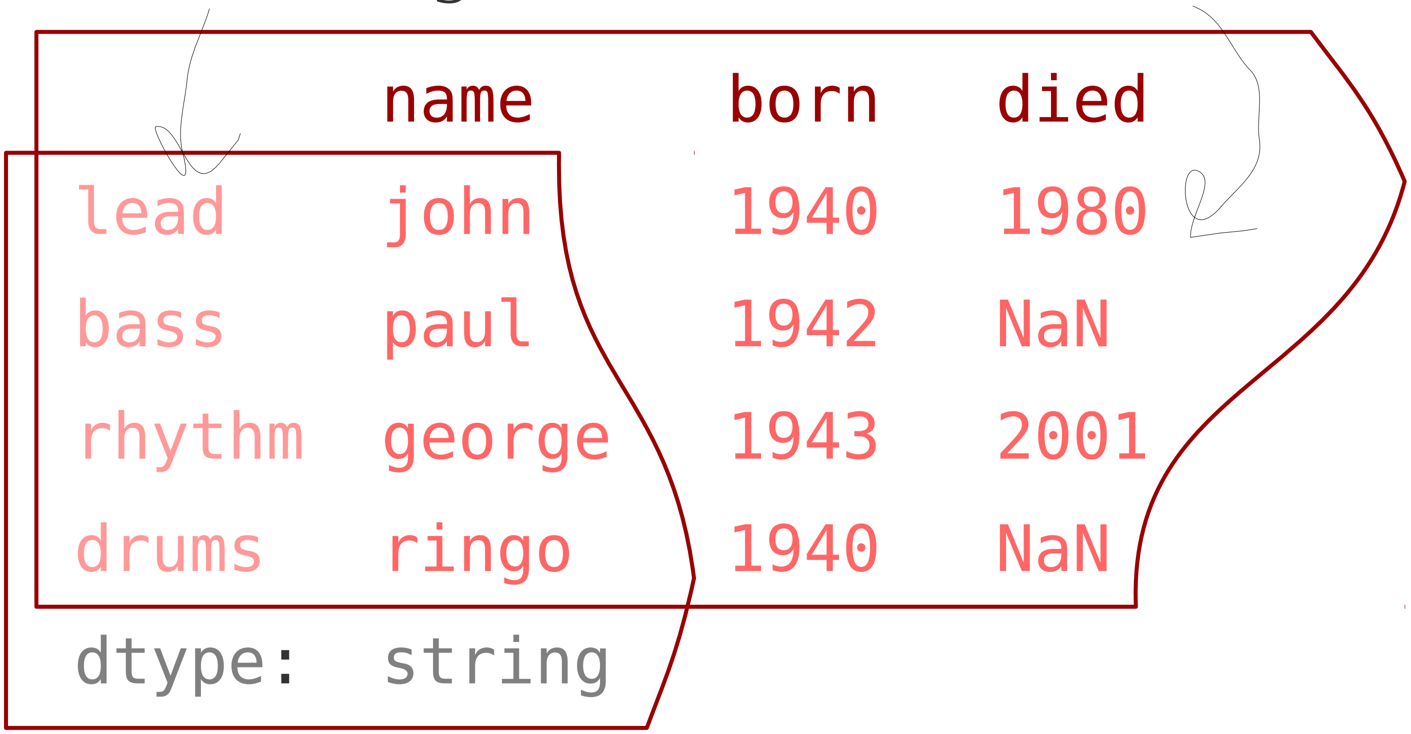


trivial to render adaptably e.g. for a 2D square array

- **`numpy`** library: `numpy.array` is a multi-dimensional array of identical-type data
- efficient high-level (e.g. vector) maths for `array` manipulation, enabling numerical computing
- intuitive for visualisation; **`matplotlib`** utilises

Considerations: Functionality

- pandas library extends NumPy & is based on two data objects that are tabular like statistical tables
- data analysis basis: efficient management of data sets e.g. NaN for missing data
- Series single-column; DataFrame multi-column



	name	born	died
lead	john	1940	1980
bass	paul	1942	NaN
rhythm	george	1943	2001
drums	ringo	1940	NaN

dtype: string

Considerations: Control

- need specific behaviour or functionality?

subclass an existing collection?

```
class myCollection<(otherCollection)>:
    """ My collection that does
    exactly what I want it to.
    """

    # create your own custom collection
```

- search the *full* Python catalogue first: don't reinvent the wheel
- `collections.UserList` & `.UserDict` may help

Loose ends

- final named collections yet to be covered:
  - ➔ `collections.defaultdict`: like `dict` but with a default value assigned on lookup of non-existent keys instead of giving a `KeyError`
  - ➔ `collections.ChainMap`: collects & allows processing of multiple `dict`

Choice by context: mnemonic

- regarding collections, we should all aim to be…

Control (custom class?)

Readability

Associations (keys?)

Functionality

Tally & order

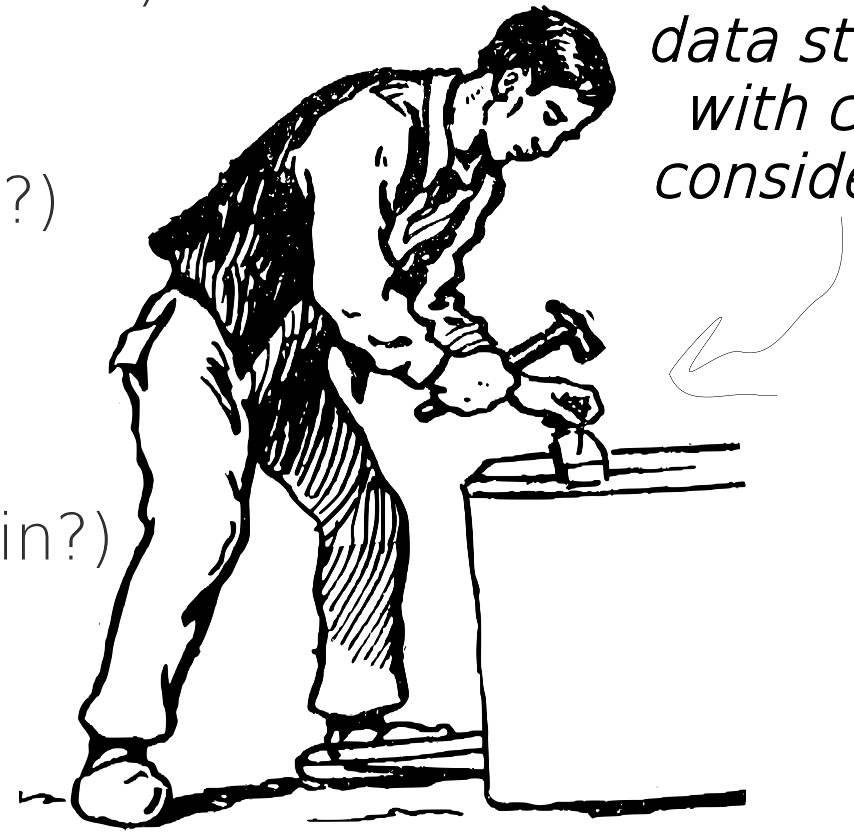
Simplicity (a built-in?)

[**WO**]

Mutability

Efficiency (time & space complexity)

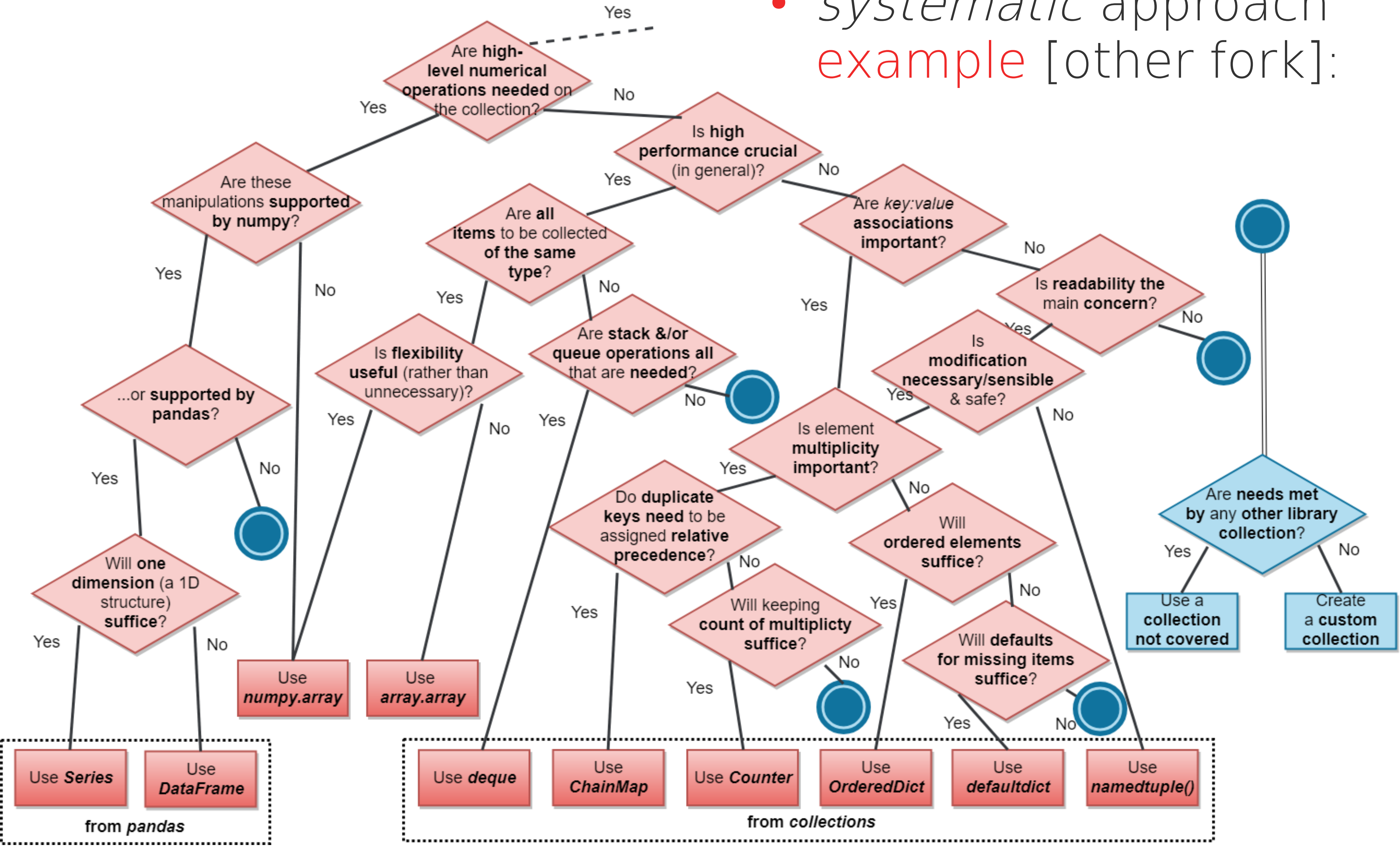
Nesting capability



us forming our data structures with care & consideration

Choice by context: flow chart

- *systematic* approach example [other fork]:



Yes
Are high-level numerical operations needed on the collection?
Yes
No
Are these manipulations supported by numpy?
Is high performance crucial (in general)?
Yes
No
Are all items to be collected of the same type?
Are key:value associations important?
No
Yes
No
Yes
No
Is readability the main concern?
Yes
...or supported by pandas?
Is flexibility useful (rather than unnecessary)?
Are stack &/or queue operations all that are needed?
Is modification necessary/sensible & safe?
Yes
No
No
No
Yes
Yes
No
Yes
Is element multiplicity important?
Yes
No
Yes
Do duplicate keys need to be assigned relative precedence?
Will ordered elements suffice?
Are needs met by any other library collection?
Will one dimension (a 1D structure) suffice?
No
Yes
No
Yes
No
Will keeping count of multiplicity suffice?
Yes
Yes
No
Will defaults for missing items suffice?
Use a collection not covered
Create a custom collection
Use numpy.array
Use array.array
Yes
No
Yes
No
Use Series
Use DataFrame
from pandas
Use deque
Use ChainMap
Use Counter
Use OrderedDict
Use defaultdict
Use namedtuple()
from collections

EOF

- any questions?

- for reference, slides located on GitHub at:
  - *sadielbartholomew/talks/python-collections.pdf*
- clip art sourced from:
  - *openclipart.org*