

Python Fundamentals

Collections

Robert Smallshire
🐦 @robsmallshire
rob@sixty-north.com



Presenter

Austin Bingham
🐦 @austin_bingham
austin@sixty-north.com



pluralsight
hardcore developer training



tuple ★

str

range

list

dict

set

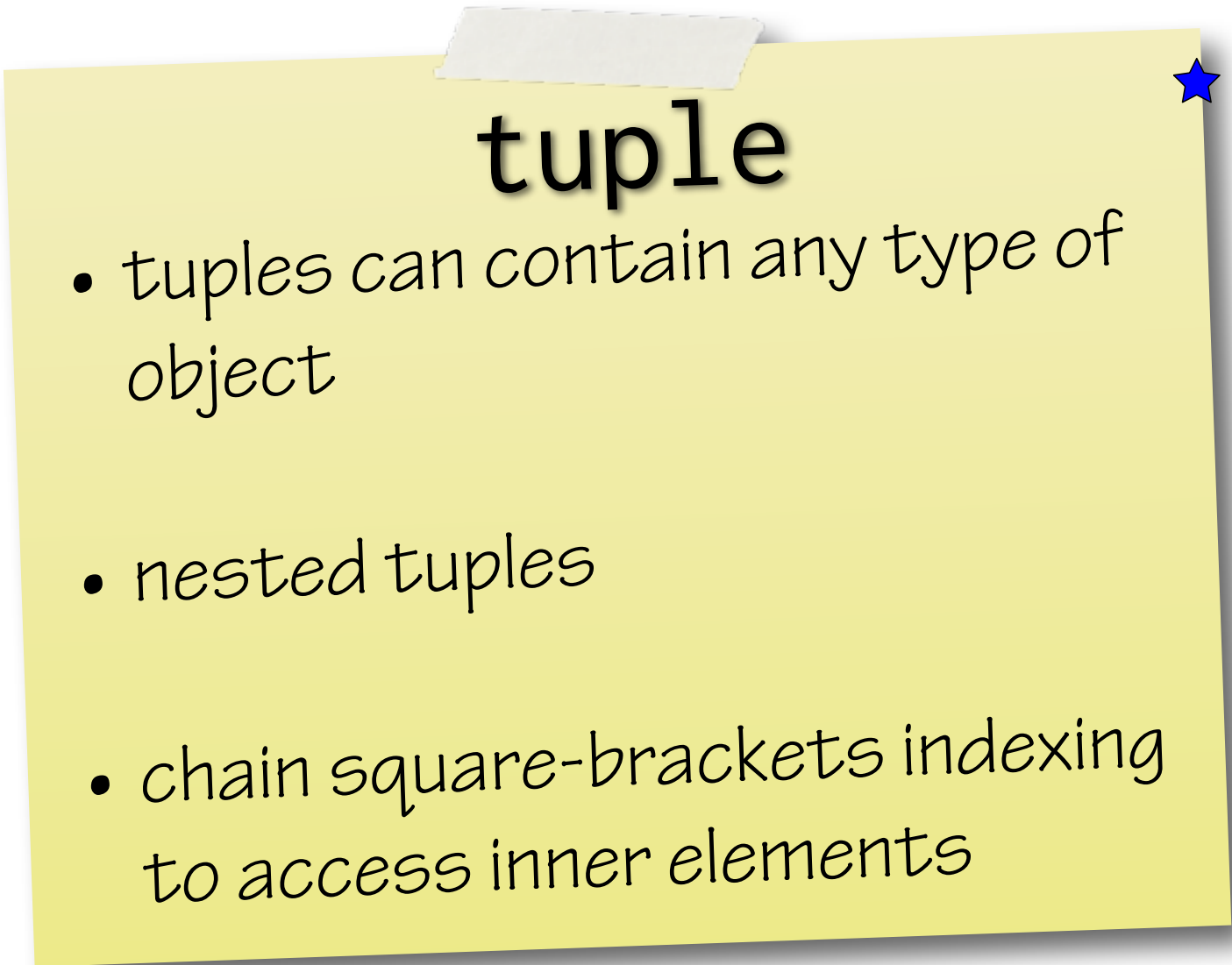


tuple

heterogeneous immutable sequence

★ tuple

- Delimited by parentheses
- Items separated by commas
- Element access with square brackets and zero-based index `t[index]`
- `len(t)` for number of elements
- Iteration with `for` loop
- Concatenation with `+` operator
- Repetition with `*` operator



tuple


- tuples can contain any type of object
- nested tuples
- chain square-brackets indexing to access inner elements

tuple

- Can't use one object in parentheses as a single element tuple
- For a single element tuple include a **trailing comma**
- The empty tuple is simply empty parentheses

tuple★

- Delimiting parentheses are optional for one or more elements
- Tuples are useful for multiple return values
- Tuple unpacking allows us to destructure directly into named references



tuple

- The **in** and **not in** operators can be used with tuples – and other collection types – for membership testing





★
str

homogeneous immutable sequence of
Unicode codepoints (characters)

str

- `len(s)` gives number of codepoints (characters)



str

- The **+** operator can be used for string concatenation.
- Strings are immutable, so the **+=** operator re-binds the reference to a new object.
- Use sparingly – concatenation with **+** or **+=** can cause performance degradation.



str

- Call the `join()` method on the **separator** string
- Use `split()` to divide a string into a list
- Without an argument, `split()` divides on whitespace
- `join()`-ing on an **empty separator** is an important and fast way of concatenating a collection of strings

Moment of Zen

The way may not
be obvious at first

To concatenate
Invoke join on empty text
Something for nothing



str

- The **partition()** method divides a string into three around a separator: *prefix, separator, suffix*
- Tuple unpacking is useful to destructure the result
- Use underscore as a dummy name for the separator
- Underscore convention understood by many tools



str

- Use **format()** to insert values into strings
- *Replacement fields* delimited by { and }
- Integer field names matched with positional arguments
- Field names can be omitted if used in sequence
- Named fields are matched with keyword arguments

str

- Access values through keys or indexes with square brackets in the replacement field.
- Access attributes using dot in the replacement field.
- The replacement field mini-language provides many value and alignment formatting options.



range

arithmetic progression of integers

range

- *stop* value is one-past-the-end
- ranges are “half-open” – *start* is included but *stop* is not
- *stop* value of a range used as *start* value of consecutive range
- optional third *step* value

| Constructor | Arguments | Result |
|-------------------------------|-------------------|--------------------|
| <code>range(5)</code> | stop | 0, 1, 2, 3, 4 |
| <code>range(5, 10)</code> | start, stop | 5, 6, 7, 8, 9 |
| <code>range(10, 20, 2)</code> | start, stop, step | 10, 12, 14, 16, 18 |



abusing range ★

- Avoid range() for iterating over lists
- Python is not C
- Don't be un-pythonic!



- Prefer direct iteration over iterable objects, such as lists

not using range – enumerate

- Prefer `enumerate()` for counters
- `enumerate()` yields *(index, value)* tuples

- Often combined with tuple unpacking



list

heterogeneous mutable sequence

| 0 | 1 | 2 | 3 | 4 | 5 |
|------|-----|----|-------|------|-----------|
| show | how | to | index | into | sequences |



| | | | | | |
|------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| show | how | to | index | into | sequences |

list

- Zero and positive integers for indexing from the front

| | | | | | |
|------|-----|----|-------|------|-----------|
| -6 | -5 | -4 | -3 | -2 | -1 |
| show | how | to | index | into | sequences |

list

- Negative integers index from the end
- The last element is at index -1
- Avoid `seq[len(seq) - 1]`

| start | | | | stop | |
|-------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| show | how | to | index | into | sequences |

list

- *Slicing* extracts part of a list
- `slice = seq[start:stop]`
- Slice range is half-open – stop not included

| start | | | | | stop |
|-------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| -6 | -5 | -4 | -3 | -2 | -1 |
| show | how | to | index | into | sequences |

list

- Slicing works with negative indexes

| start | | | stop | | |
|-------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| show | how | to | index | into | sequences |

list

- Omitting the *stop* index slices to the end
`slice_to_end = seq[start:]`

| start | | | stop | | |
|-------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| show | how | to | index | into | sequences |

list

- Omitting the *start* index slices from the beginning
`slice_from_beginning = seq[:stop]`

`s[:3]`

| start | | | stop | | |
|-------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| show | how | to | index | into | sequences |

`s[3:]`

| | | | start | | | stop |
|------|-----|----|-------|------|-----------|------|
| 0 | 1 | 2 | 3 | 4 | 5 | |
| show | how | to | index | into | sequences | |

list

- Half-open ranges give complementary slices

$$s[:x] + s[x:] == s$$

start

stop

| | | | | | |
|------|-----|----|-------|------|-----------|
| 0 | 1 | 2 | 3 | 4 | 5 |
| -6 | -5 | -4 | -3 | -2 | -1 |
| show | how | to | index | into | sequences |

list



- Omitting the *start* and *stop* indexes slices from the beginning to the end – a *full slice*

```
full_slice = seq[:]
```

- Important idiom for copying lists

3 ways for copying Lists.

list ★

Copying lists

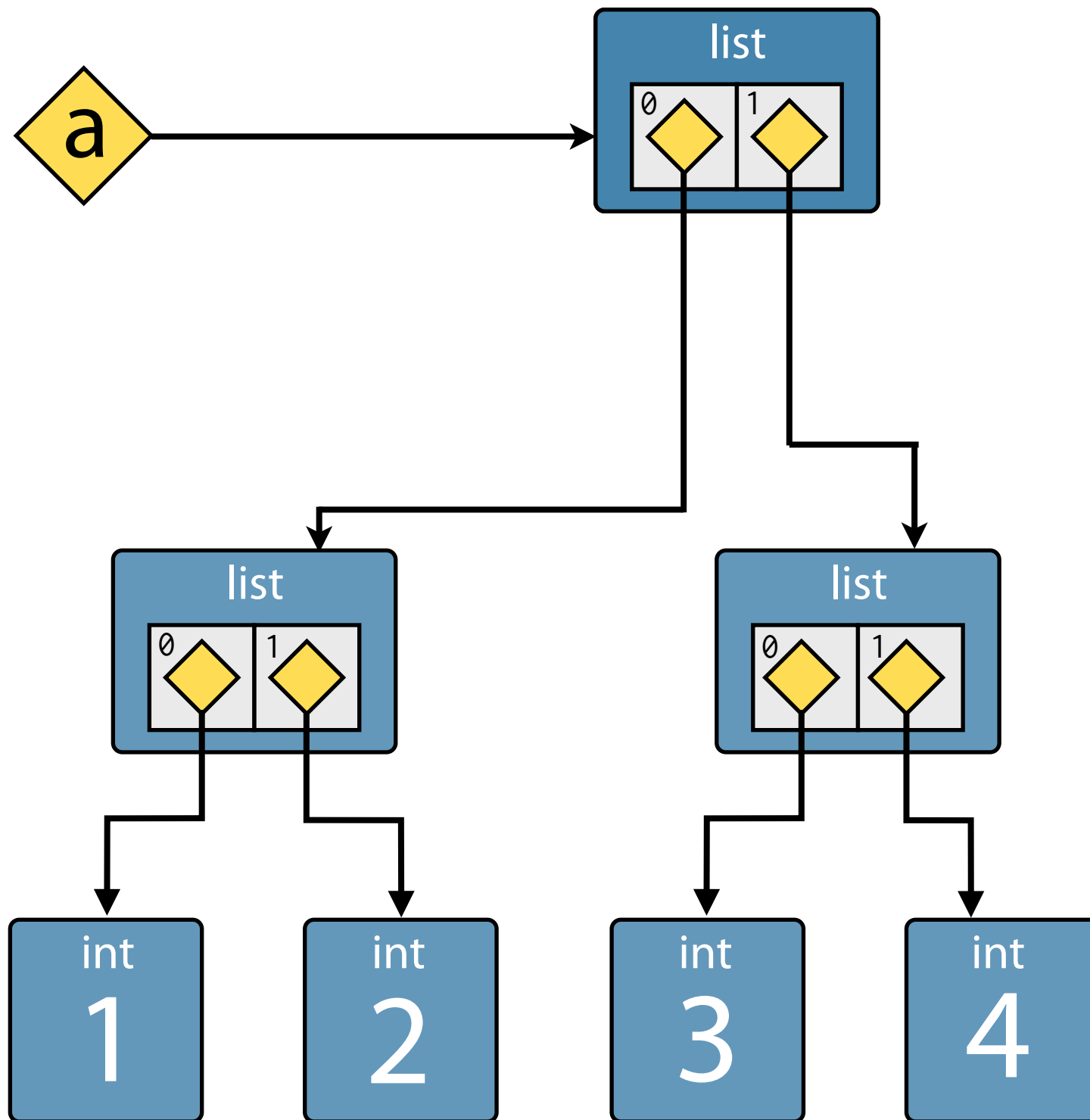
Full slice: `t = seq[:]`

`copy()` method: `u = seq.copy()`

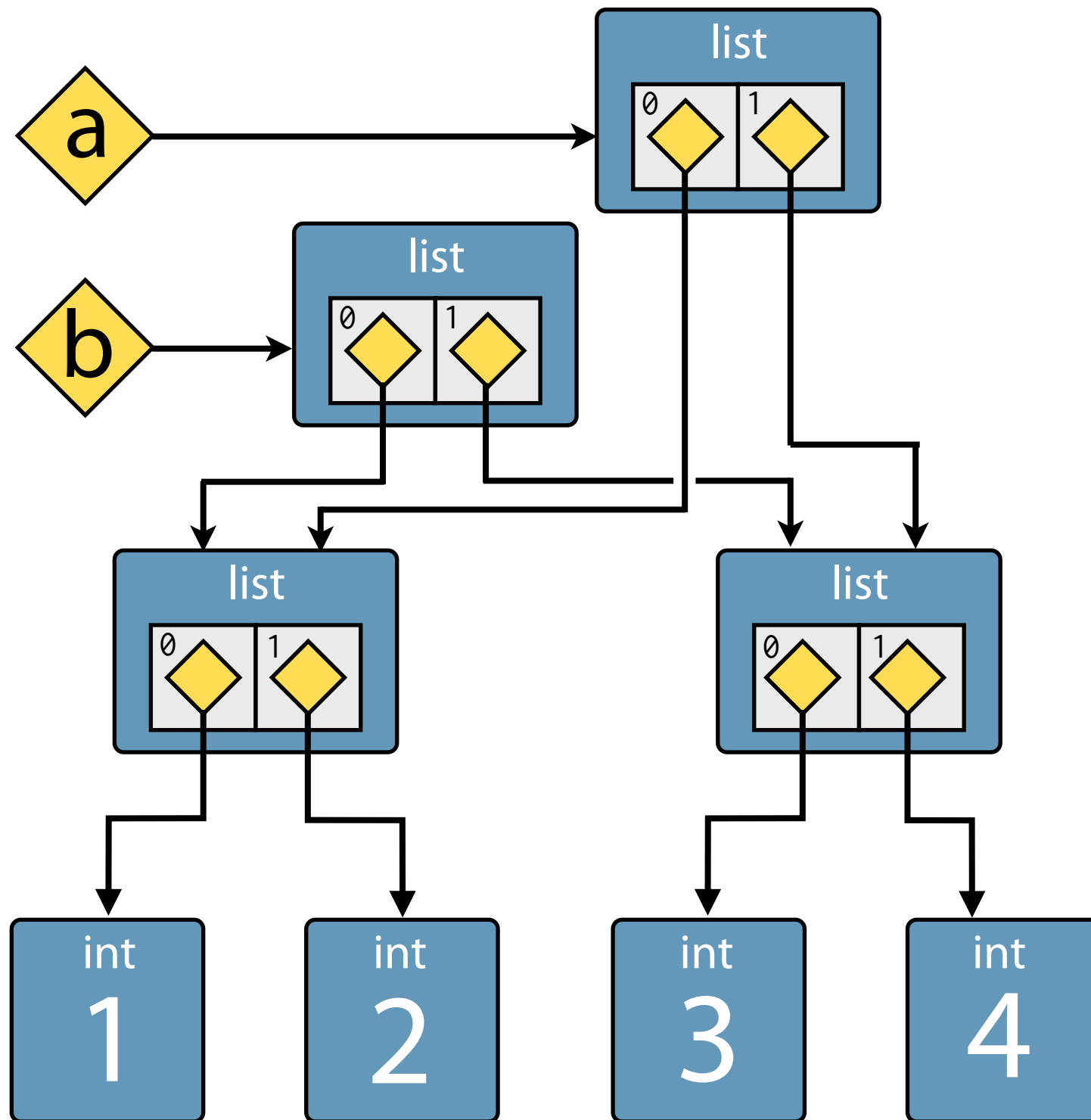
`list()` constructor: `v = list(seq)`

Copies are shallow

```
>>> a = [ [1, 2], [3, 4] ]  
>>> b = a[:]
```



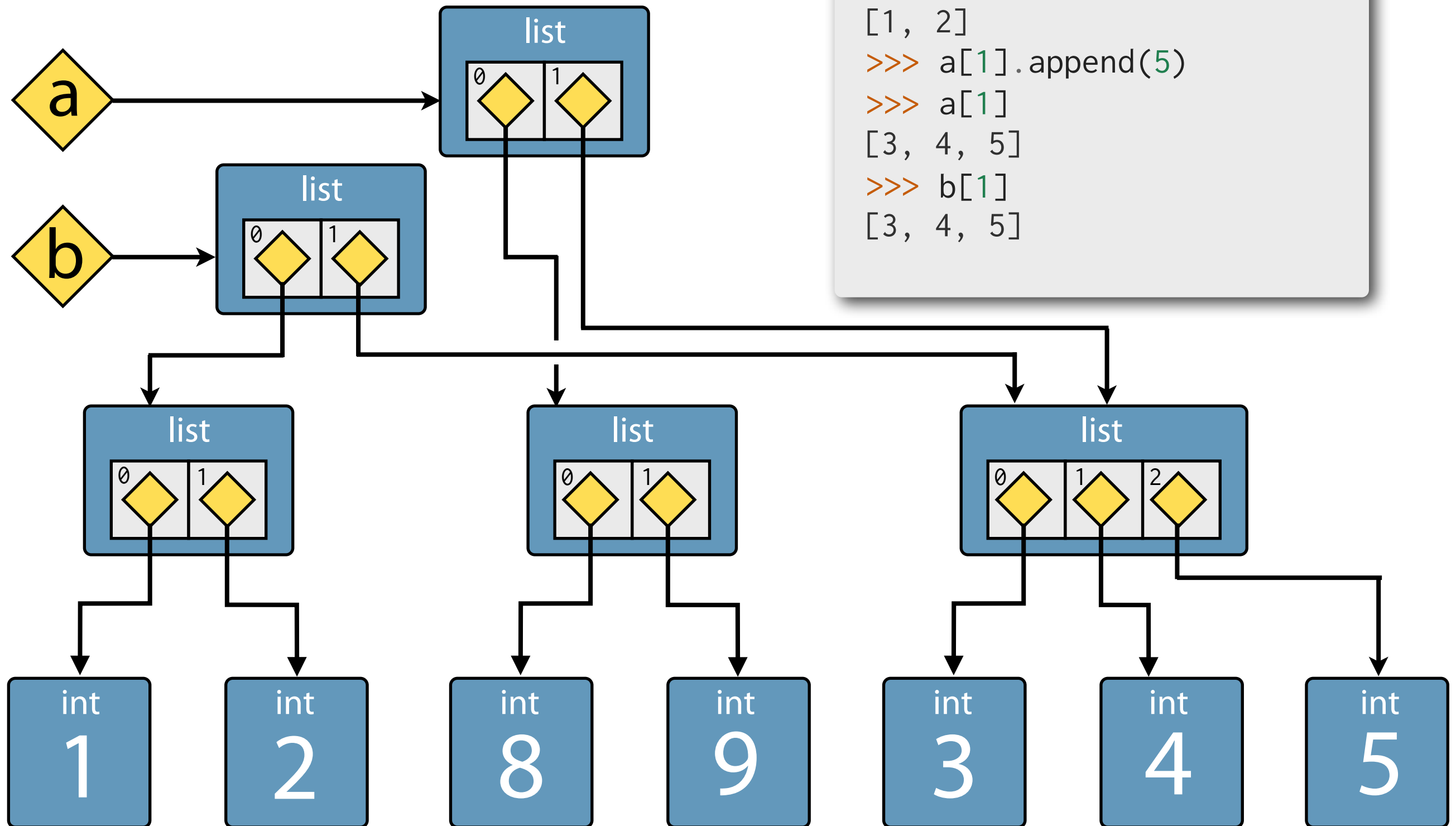
Copies are shallow



```
>>> a = [ [1, 2], [3, 4] ]
>>> b = a[:]
>>> a is b
False
>>> a == b
True
>>> a[0]
[1, 2]
>>> b[0]
[1, 2]
>>> a[0] is b[0]
True
```

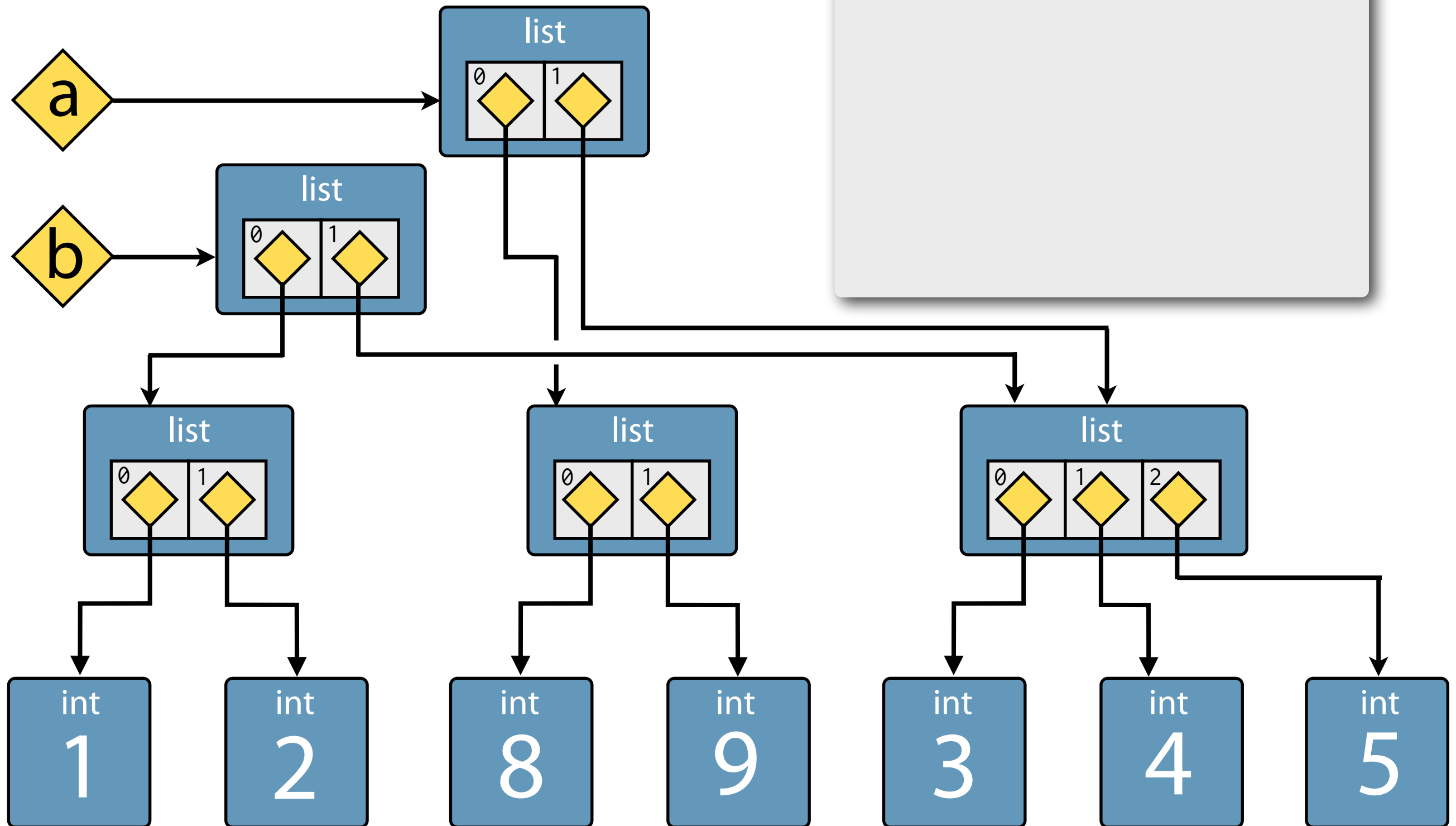
Copies are shallow

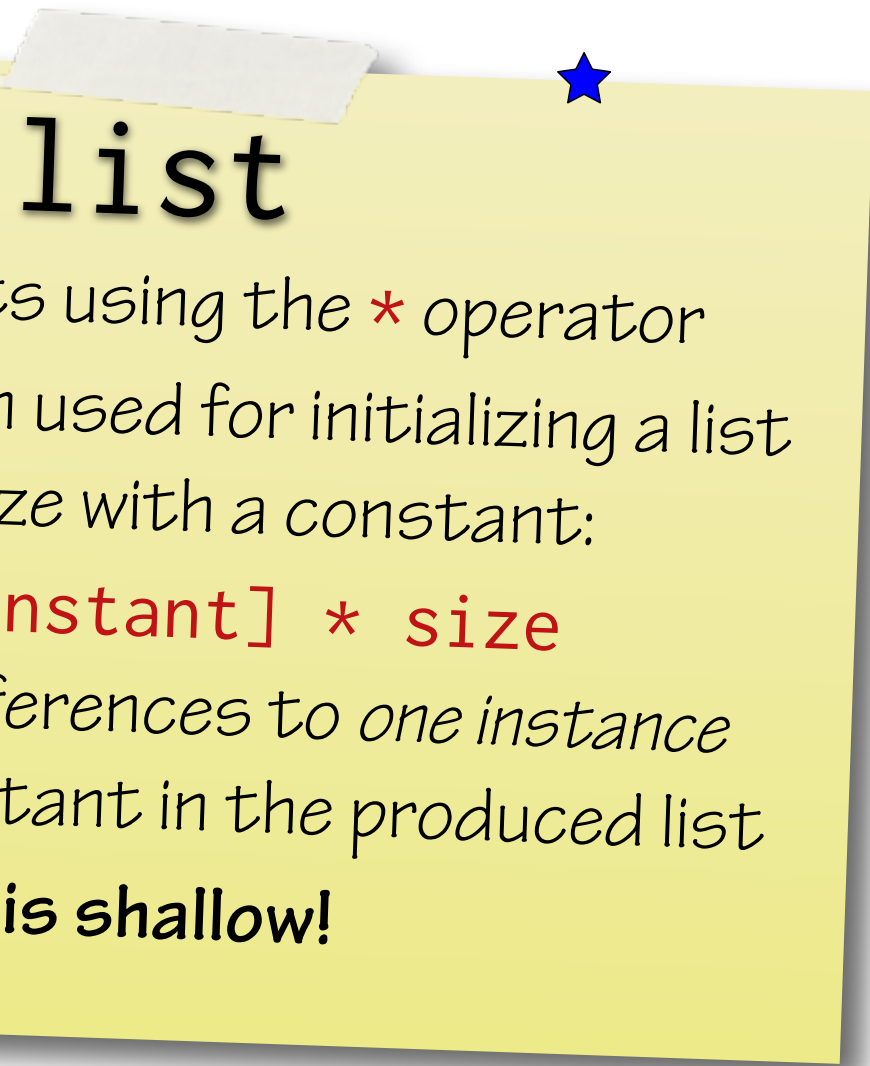
```
>>> a[0] = [8, 9]
>>> a[0]
[8, 9]
>>> b[0]
[1, 2]
>>> a[1].append(5)
>>> a[1]
[3, 4, 5]
>>> b[1]
[3, 4, 5]
```



Copies are shallow

```
>>> a
[[8, 9], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
```





list

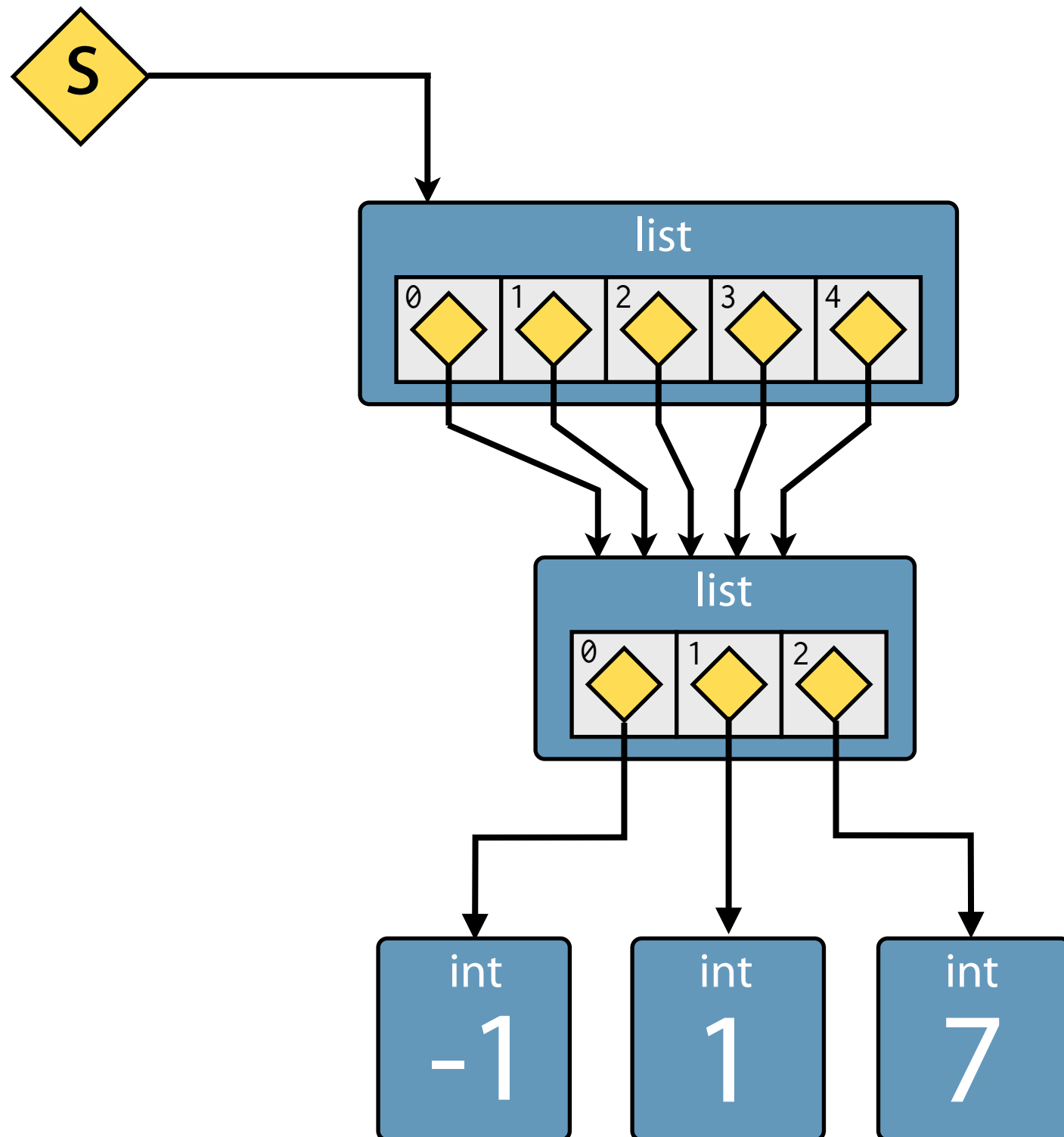
- Repeat lists using the *** operator
- Most often used for initializing a list of known size with a constant:

*s = [constant] * size*


Multiple references to one instance of the constant in the produced list

- **Repetition is shallow!**

Repetition is Shallow



```
>>> s = [ [-1, +1] ] * 5
>>> s
[[-1, 1], [-1, 1], [-1, 1],
 [-1, 1], [-1, 1]]
>>> s[3].append(7)
>>> s
[[-1, 1, 7], [-1, 1, 7], [-1,
 1, 7], [-1, 1, 7], [-1, 1, 7]]
```



★ list

Finding elements

- `index(item)` returns the integer index of the first equivalent element raises `ValueError` if not found
- `count(item)` returns the number of matching elements
- The `in` and `not in` operators test for membership

★ list

- `del seq[index]` to remove by index
- `seq.remove(item)` to remove by value;
raises `ValueError` if not present
- `remove()` equivalent to
`del seq[seq.index(item)]`

list



- Insert items with

```
seq.insert(index, item)
```

list ★

- Concatenate lists with **+** operator
- In-place extension with **+=** operator or **extend()** method.
- All accept any iterable series on the right-hand side.



list

- `k.reverse()` reverses in place

- `k.sort()` sorts in place
- `k.sort(reverse=True)` gives descending sort

list

- **key** argument to **sort()** method accepts a function for producing a sort key from an item

- be aware of unintentional side-effects with *in situ* rearrangements

list

- **sorted()** built-in function sorts any iterable series and returns a list



- **reversed()** built-in function reverses any iterable series
- returns a reverse iterator



dict

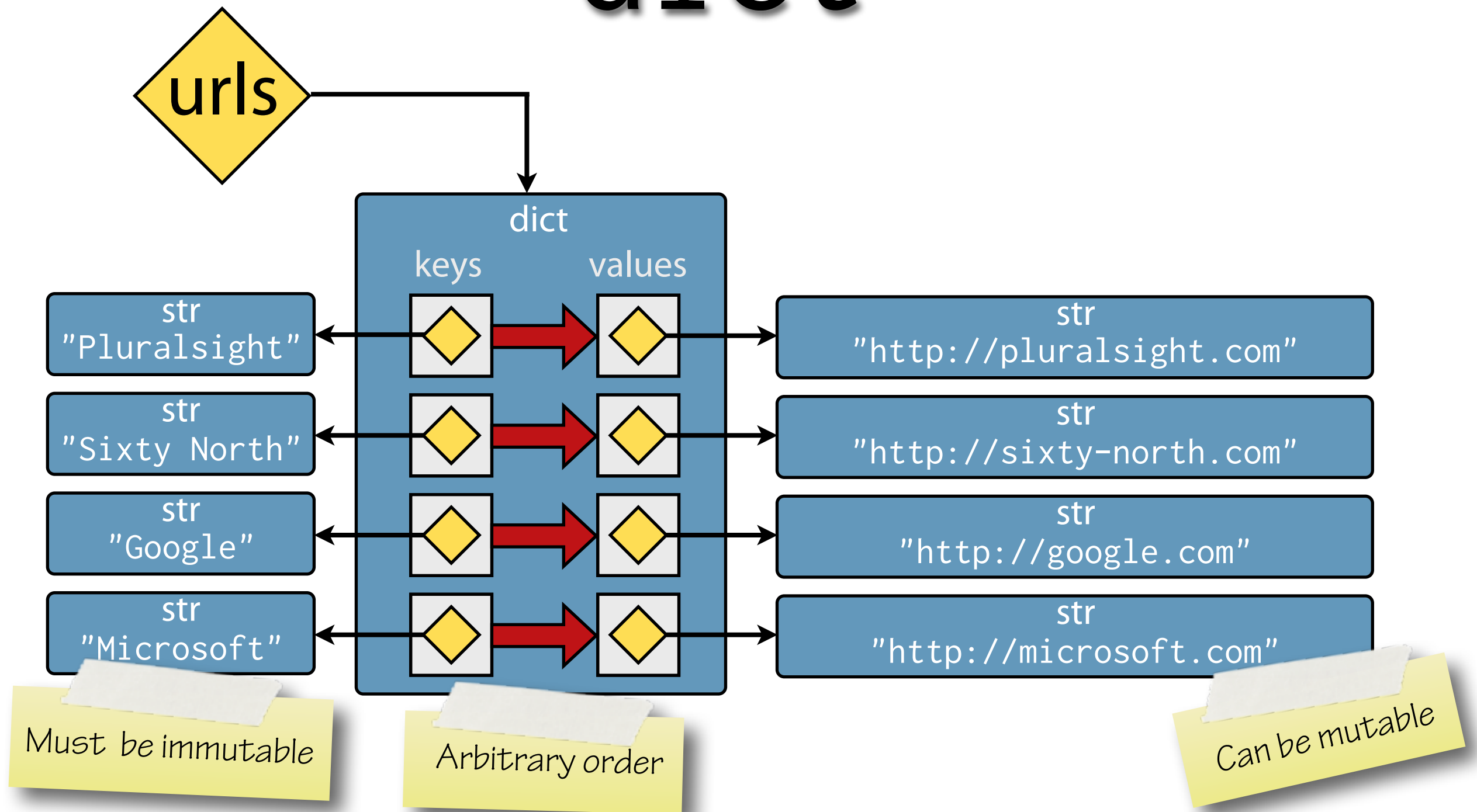
unordered mapping from
unique, immutable keys
to mutable values

dict

Recap literals:

- delimited by { and }
- key-value pairs comma separated
- corresponding keys and values joined by colon
- keys must be unique

dict



dict

`dict()` constructor accepts:

- iterable series of key-value 2-tuples
- keyword arguments – requires keys are valid Python identifiers
- a mapping, such as another dict

dict

Copying

- `d.copy()` for copying dictionaries
- or simply `dict(d)` constructor

dict

Updating a dictionary

- Extend a dictionary with `update()`
- `update` replaces values corresponding to duplicate keys



dict

Iteration

- Iteration is over *keys*
- Get corresponding value with `d[key]` lookup
- **Remember! Order is arbitrary!**

dict

Iteration

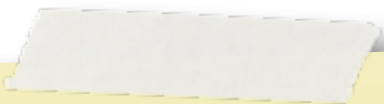
- Use **values()** for an iterable view onto the series of *values*.
- No efficient way to get the key corresponding to a value
- **keys()** method gives iterable view onto keys - not often needed.

dict

Iteration

- Use `items()` for an iterable view onto the series of *key-value* tuples.
- Use with tuple unpacking.






dict

Membership

- The **in** and **not in** operators work on the *keys*.





dict

Removal



- Use **del** keyword to remove by key

```
del d[key]
```




dict

Mutability

- *keys must be immutable*
- *values may be mutable*
- *The dictionary itself is mutable*

Pretty printing

- Python Standard Library `pprint` module
- Be careful not to rebind the module reference!
- Knows how to pretty-print all built-in data structures, including `dict`





set.★

unordered collection of
unique, immutable objects

set

Literals

- delimited by { and }
- single comma separated items
- empty {} makes a dict, so for empty set use the set() constructor.

set

`set()` constructor accepts:

- iterable series of values
- duplicates are discarded
- often used specifically to remove duplicates – not order preserving

set

Membership / containment

- Fundamental operation for sets
- Use **in** and **not in** operators



set

Adding elements

- `add(item)` inserts a single element
- Duplicates are silently ignored
- For multiple elements use `update(items)` passing any iterable series





set

Removing elements


- `remove(item)` requires that item is present, otherwise raises `KeyError`.
- `discard(item)` always succeeds.



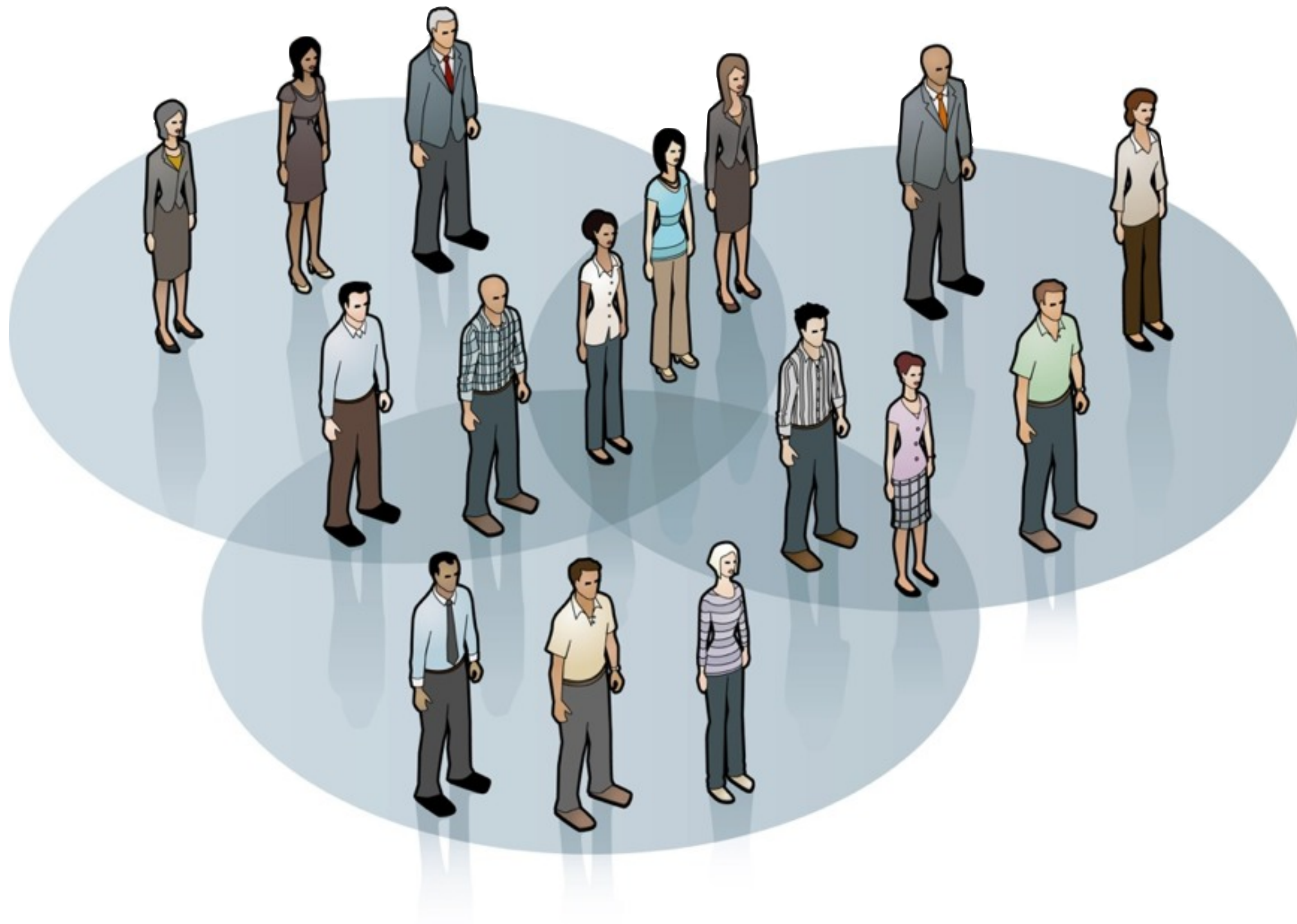


set

Copying

- `s.copy()` method. 
- Use constructor: `set(s)`
- **Copies are shallow!**

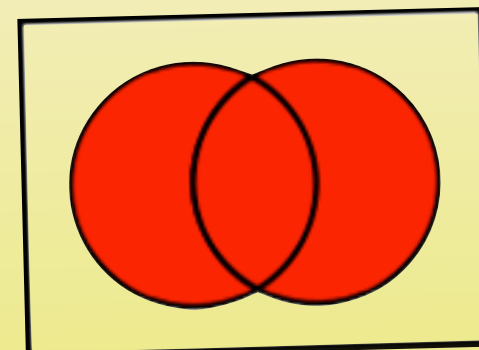
python™ set algebra



set

Union

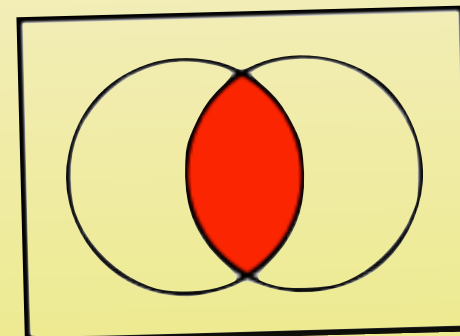
- `s.union(t)` method.
- commutative



set

Intersection

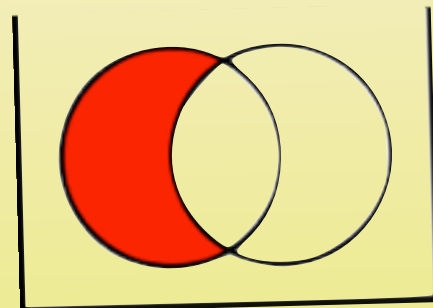
- `s.intersection(t)` method.
- commutative



set

- `s.difference(t)` method.
- non-commutative

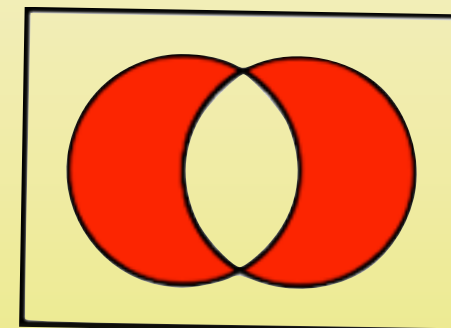
Difference



set

Symmetric Difference

- `s.symmetric_difference(t)` method.
- commutative

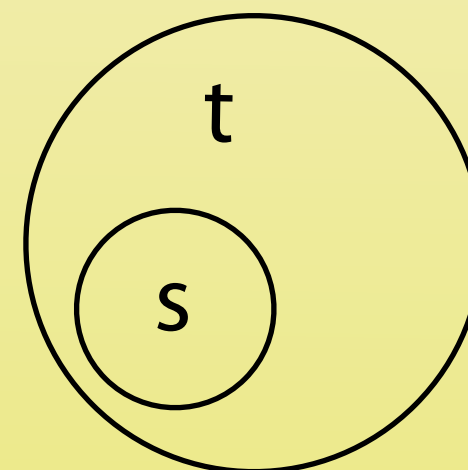




set

Subset

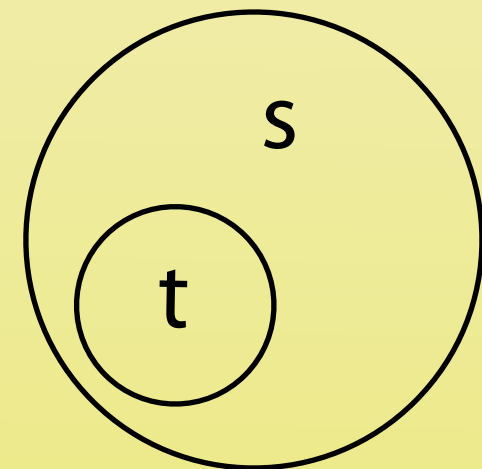
`s.issubset(t)` method.

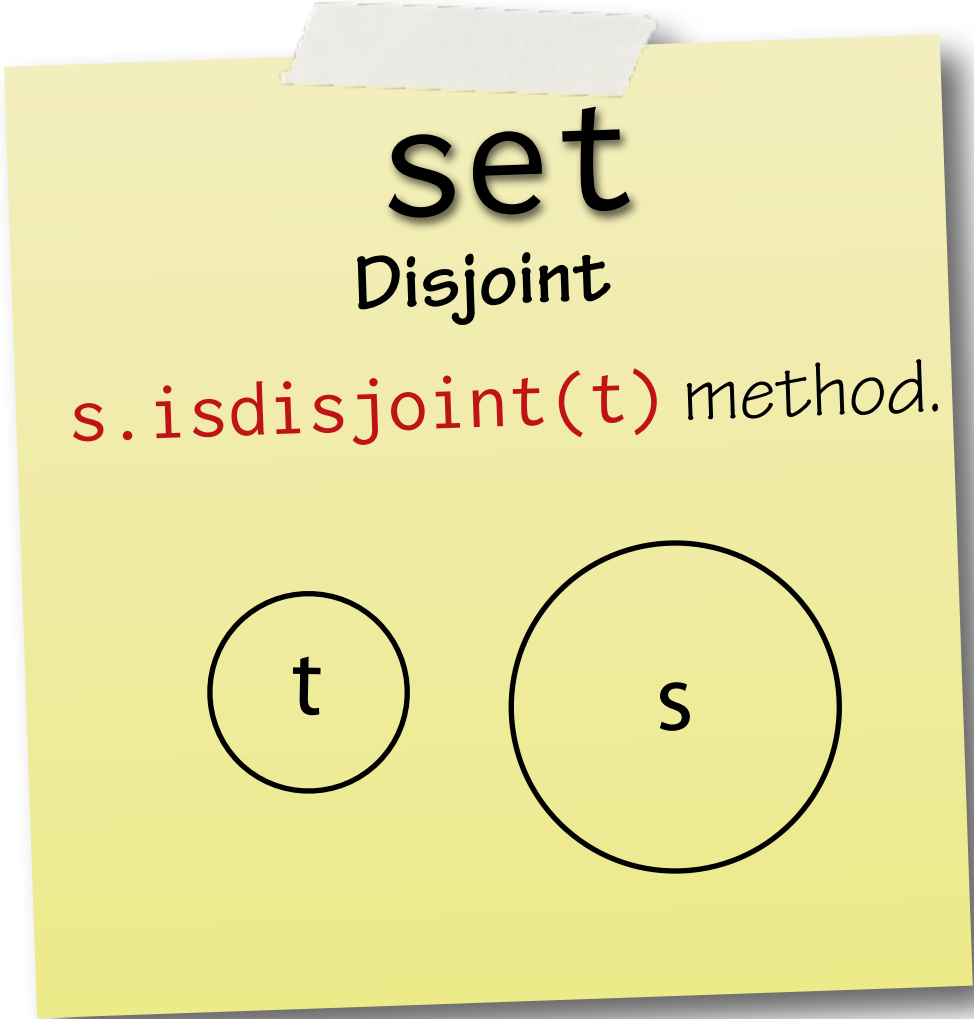


set

Superset

`s.issuperset(t)` method.

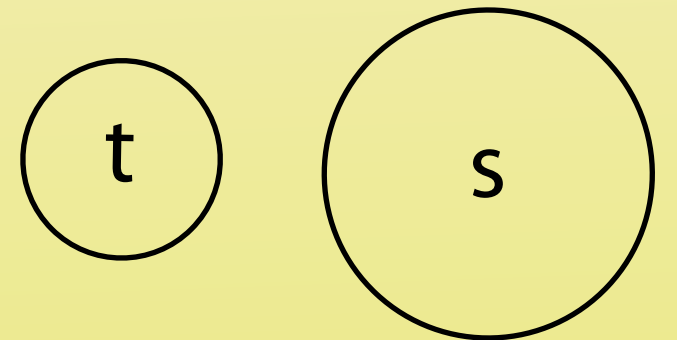




set

Disjoint

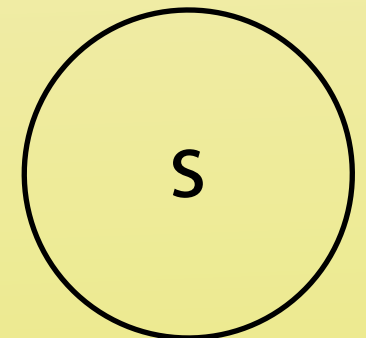
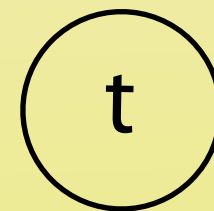
`s.isdisjoint(t)` method.



set

Disjoint

`s.isdisjoint(t)` method.





tuple

str

range

list

dict

set



Collection Protocols

| Protocol | Implementing Collections |
|------------------|--|
| Container | str, list, range, tuple , bytes, set, dict |
| Sized | str , list, range, tuple, bytes, set, dict |
| Iterable | str, list, range , tuple, bytes, set, dict |
| Sequence | str, list , range, tuple, bytes |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |





Collection Protocols

| Protocol | Implementing Collections |
|------------------|---|
| Container | str, list, range, tuple, bytes, set, dict |
| Sized | str, list, range, tuple, bytes, set, dict |
| Iterable | str, list, range, tuple, bytes, set, dict |
| Sequence | str, list, range, tuple, bytes |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |

Protocols

- To implement a protocol, objects must support certain operations.
- Most collections implement *container*, *sized* and *iterable*.
- All except *set* and *dict* are *sequences*.



Collection Protocols

| Protocol | Implementing Collections |
|------------------|---|
| Container | str, list, range, tuple, bytes, set, dict |
| Sized | str, list, range |
| Iterable | |
| Sequence | |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |

Container Protocol

- Membership testing using **in** and **not in**



Collection Protocols

| Protocol | Implementing Collections |
|------------------|---|
| Container | str, list, range, tuple, bytes, set, dict |
| Sized | str, list, range, tuple, bytes, set, dict |
| Iterable | str, list, range, tuple, bytes, set, dict |
| Sequence | str, list, range, tuple, bytes, set, dict |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |

Sized Protocol

- Determine number of elements with `len(s)`



Collection Protocols

| Protocol | Implementing Collections |
|------------------|---|
| Container | str, list, range, tuple, bytes, set, dict |
| Sized | str, list, range, tuple, bytes, set, dict |
| Iterable | str, list, range, tuple, bytes, set, dict |
| Sequence | str, list, range, tuple |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |

Iterable Protocol

- Can produce an *iterator* with `iter(s)`

```
for item in iterable:  
    do_something(item)
```




Collection Protocols

| Protocol | Implementing Collections |
|------------------|---|
| Container | str, list, range, tuple, bytes, set, dict |
| Sized | str, list, range, tuple, bytes, set, dict |
| Iterable | str, list, range, tuple, bytes, set, dict |
| Sequence | str, list, range, tuple, bytes |
| Mutable Sequence | list |
| Mutable Set | set |
| Mutable Mapping | dict |

Sequence Protocol

- Retrieve elements by index
`item = seq[index]`
- Find items by value
`index = seq.index(item)`
- Count items
`num = seq.count(item)`
- Produce a reversed sequence
`r = reversed(seq)`



Collections Summary

- **Tuples are immutable sequence types**
 - Literal syntax: optional parentheses around a comma separated list
 - Single element tuples must use trailing comma
 - Tuple unpacking - return values and idiomatic swap
- **Strings are immutable sequence types of Unicode codepoints**
 - String concatenation is most efficiently performed with `join()` on an empty separator
 - The `partition()` method is a useful and elegant string parsing tool.
 - The `format()` method provides a powerful way of replacing placeholders with values.
- **Ranges represent integer sequences with regular intervals**
 - Ranges are arithmetic progressions
 - The `enumerate()` function is often a superior alternative to `range()`



Collections Summary

- **Lists are heterogeneous mutable sequence types**
 - Negative indexes work backwards from the end.
 - Slicing allows us to copy all or part of a list.
 - The full slice is a common idiom for copying lists, although the `copy()` method and `list()` constructor are less obscure.
 - List (and other collection) copies are shallow.
 - List repetition is shallow.
- **Dictionaries map immutable keys to mutable values**
 - Iteration and membership testing is done with respect to the keys.
 - Order is arbitrary
 - The `keys()`, `values()` and `items()` methods provide views onto different aspects of a dictionary, allowing convenient iteration.
- **Sets store an unordered collection of unique elements**
 - Sets support powerful and expressive set algebra operations and predicates.
- **Protocols such as *iterable*, *sequence* and *container* characterise the collections.**



- Lists

- M
- S
- T
- 1
- M
- L

- Dictionaries

- It
- C
- T
- d

- Sets

- S

- Pro
collections.

Next time in Python Fundamentals

- ❧ errors
- ❧ exceptions
- ❧ forgiveness

and

ects of a