Sets

Lecture 6-1

Hyung-Sin Kim

SNU Graduate School of Data Science

Sets

- Differently from Lists, Sets store **unordered** and **distinct** items
- Like Lists, Sets are **mutable** and can be used as **function arguments**
- Just as int, float, str, bool, and list, set is also a class that has its own methods

- Declare a set using {xxxx}
  - >>> vowels = {“a”, “e”, “i”, “o”, “u”, “a”, “u”, “i”, “e”}
  - >>> vowels ⟹ {‘a’, ‘u’, ‘e’, ‘o’, ‘i’}
    - No particular order (different from that we typed)
    - No duplicate items
- Check if duplicates are ignored
  - >>> {“a”, “e”, “i”, “o”, “u”, “a”, “u”, “i”, “e”} == {‘a’, ‘u’, ‘e’, ‘o’, ‘i’}
  - True

Sets

- An empty set
  - >>> a = set() - Not {} !

- We can change a list to a set
  - >>> set([2, 3, 2, 5, 5, 5]) ➡ {2, 3, 5}

- We can loop over a set
  - >>> students = set(["inhoe", "yesong", "jaehyuk", "yeha", "suun", "nayeon"])
  - >>> for student in students:
  - … print(student)

Sets – Methods

- digits = set([0, 1, 2, 3, 4, 5]) / odds = set([1, 3, 5, 7, 9])
  - >>> digits.add(6)
  - >>> digits.remove(2)
  - >>> digits.clear()

  - >>> digits.issubset(odds) == digits <= odds
  - >>> digits.issuperset(odds) == digits >= odds

  - >>> digits.difference(odds) == digits - odds
  - >>> digits.intersection(odds) == digits & odds
  - >>> digits.symmetric_difference(odds) == digits ^ odds
  - >>> digits.union(odds) == digits | odds

Sets – Immutable Content

- Set supports "**in**" operator (value_A in set_A)

- Checking for set membership is fast (directly done) since it uses mathematical technique called ***hashing***
  - In contrast, checking for list membership needs scanning the whole items

- Hashing assumes that all elements of a set are **immutable** and a mutable element like List cannot be an element of a set
  - >>> S = set()
  - >>> L = [1, 2, 3]
  - >>> S.add(L)
  - Error ^0^

Tuples

Lecture 6-1

Hyung-Sin Kim

SNU Graduate School of Data Science

Tuples

- Like Lists, Tuples have **ordered** items
- Unlike Lists (and like Strings), Tuples are **immutable**

- Declare a tuple using ()
  - >>> nums = ()
  - >>> nums = (8,)
  - >>> nums = (5+3,) to differentiate from arithmetic operations
- Loop over a tuple
  - >>> nums = (1, 2, 3, 4)
  - >>> for num in nums:
  - … print(num)

Tuples

- Tuples **cannot** be mutated
  - >>> family = (“dad”, “mom”, “me”, “brother”)
  - >>> family[0] = “grand father”
  - Error ^0^

- Objects inside tuples **can** be mutated
  - >>> family = ([“dad”, 60] , [“mom”, 58], [“me”, 27], [“brother”, 24])
  - >>> family[0][1] = 61
  - >>> family ⟹ ([“dad”, 61] , [“mom”, 58], [“me”, 27], [“brother”, 24])

Dictionaries

Lecture 6-2

Hyung-Sin Kim

SNU Graduate School of Data Science

Dictionaries

- Dictionaries have **ordered mutable** items without **duplicates**
  - Insertion order
- Unlike those in sets, items in dictionaries are **key/value pairs**
  - Key is like index in Lists

- Declare using {}
  - >>> dict_empty = {}
  - >>> dict_grades = {“inhoe”: “A”, “yesong”: “A+”, “jaewook”: “A++”}
- Access using keys ➡
  - >>> dict_grades[“inhoe”] “A”

Dictionaries

- Updating and checking membership of Dictionaries
  - >>> dict_grades = {}
  - >>> dict_grades[“inhoe”] = “A” (adding a new key/value pair)
  - >>> dict_grades[“inhoe”] = “B” (modifying an existing pair, not adding another)
  - >>> “inhoe” in dict_grades (this is fast like sets)
  - >>> del dict_grades[“inhoe”]

Looping

- Looping over dictionaries
  - for <<variable>> in <<dictionary>>:
  - <<block>>
  - The loop variable is assigned each **key** from the dictionary

- Example
  - >>> for student in dict_grades:
  - >>> print(student, “got grade”, dict_grades[student])

  - inhoe got grade A
  - yesong got grade A+
  - jaewook got grade A++

Methods

- dict_grades.clear()
  - Remove all key/value pairs
- dict_grades.keys()
  - Returns all keys as a set-like object (unique entries)
- dict_grades.items()
  - Returns all key/value pairs as set-like objects (unique entries)
- dict_grades.values()
  - Returns all values as a list-like object (maybe not unique entries)
- dict_grades.get(k)
  - Returns the value associated with key k
- dict_grades.get(k, v)
  - Same + returns v if key k is not present

Methods

- dict_grades.pop(k)
  - Removes key k and returns the value associated with the key
- dict_grades.pop(k,v)
  - Same + returns v if key k is not present
- dict_grades.setdefault(k)
  - Returns the value associated with key k, if key k is not present, add it and set its associated value to None
- dict_grades.setdefault(k,v)
  - Returns the value associated with key k, if key k is not present, add it and set its associated value to v
- dict_grades.update(dict2)
  - Update dict_grades with the contents of dict2

“in” Operation

- grades = {“inhoe”: “A”, “yesong”: “A+”, “jaewook”: “A”, “hyung-sin”: “C”}

- “**in**” operator only checks if a value is one of the dictionary’s **keys** (ignoring values)
  - >>> “jaewook” in grades
  - True
  - >>> “A” in grades
  - False

Invert a Dictionary!

- grades = {“inhoe”: “A”, “yesong”: “A+”, “jaewook”: “A”, “hyung-sin”: “C”}

- Write a program that provides a dictionary **grades_inv** that has inverted key-value pairs (grades’ keys become values and values become keys)

- What is the main challenge?

Invert a Dictionary! (Solution)

- grades = {“inhoe”: “A”, “yesong”: “A+”, “jaewook”: “A”, “hyung-sin”: “C”}

- >>> grades_inv = {}
- >>> for student, grade in grades.items():
- >>> if grade in grades_inv:
- >>> grades_inv[grade].**append**(student)
- >>> else:
- >>> grades_inv[grade] = [student]

Mutability

Lecture 6-3

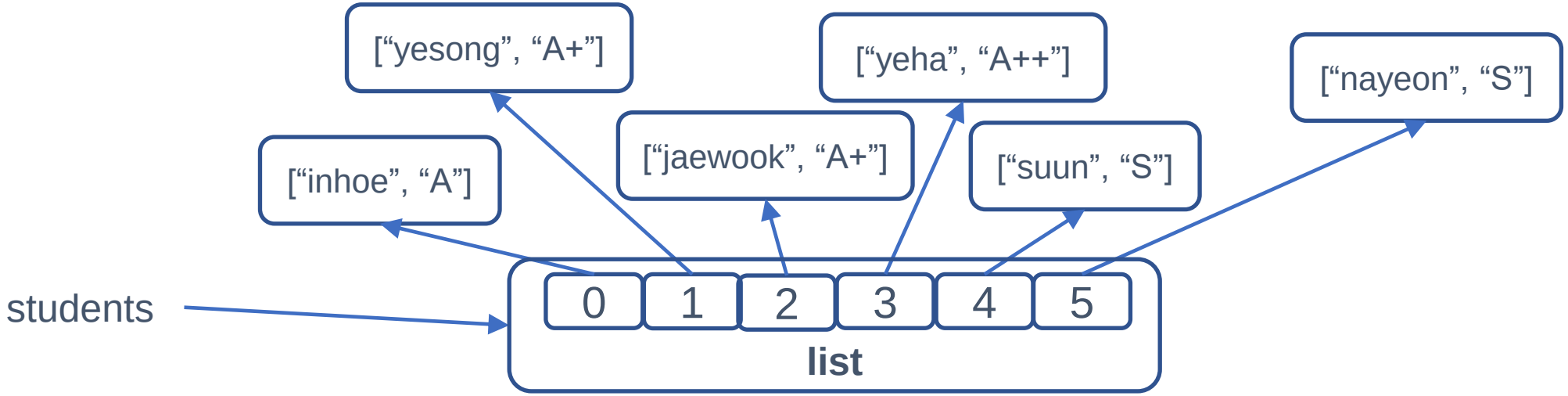
Hyung-Sin Kim

SNU Graduate School of Data Science

Lets review the various data structures we've learnt so far, in terms of ***mutability***

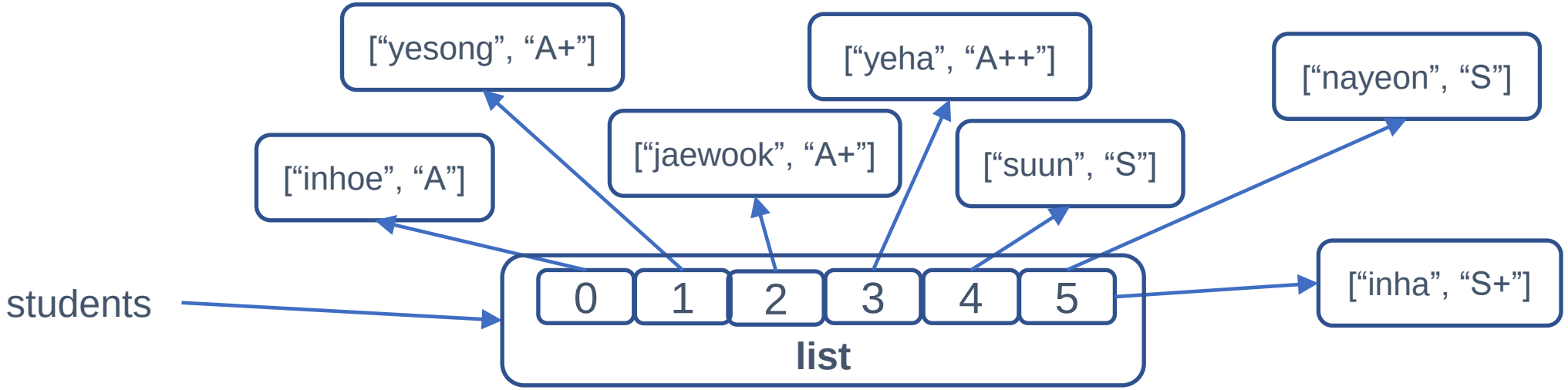
Mutability – List

- List (mutable container with mutable elements)
  - student[5] = [“inha”, “S+”]



Mutability – List

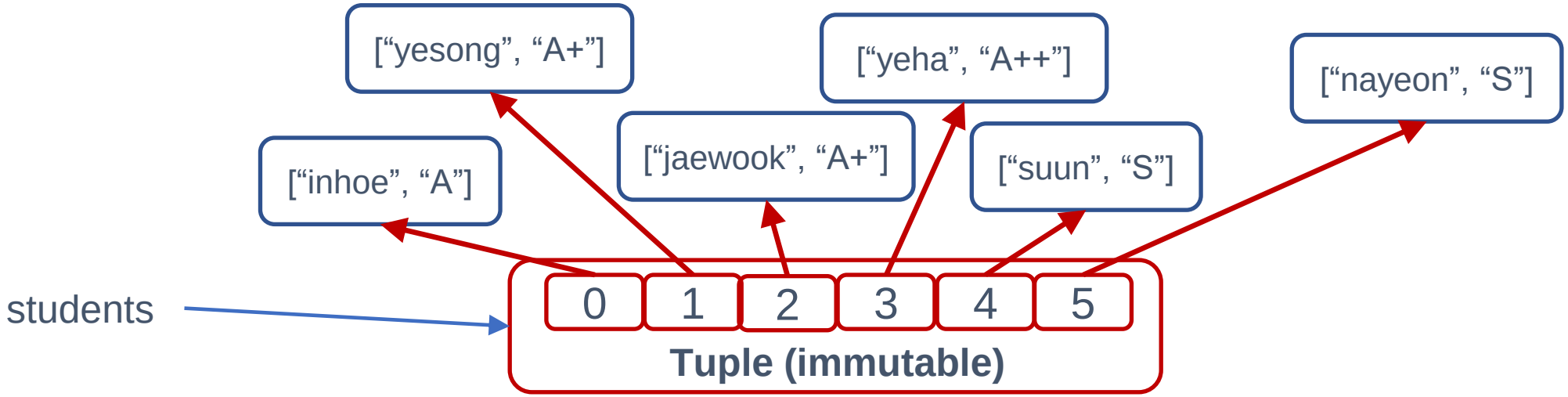
- List (mutable container with mutable elements)
  - student[5] = [“inha”, “S+”]
  - student[5] is pointing at a **different object (address)**



When a container (collection) is **mutable**, after it is defined
(1) its elements can be added or removed
(2) Its elements can change what **address (object)** to point at

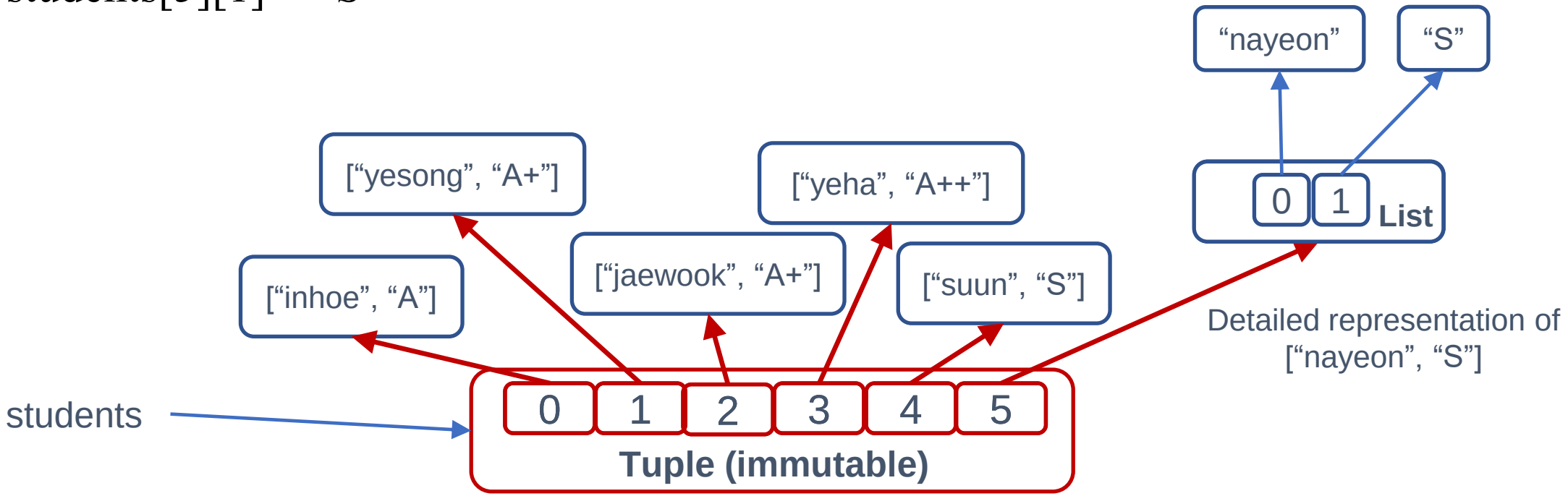
Mutability – Tuple

- Tuple (immutable container with mutable elements)
  - Once students is defined, # of elements and all the elements' arrows are **fixed**
  - students[5] = ["inha","S+"] does not work since students[5] **cannot** point at a **different object**



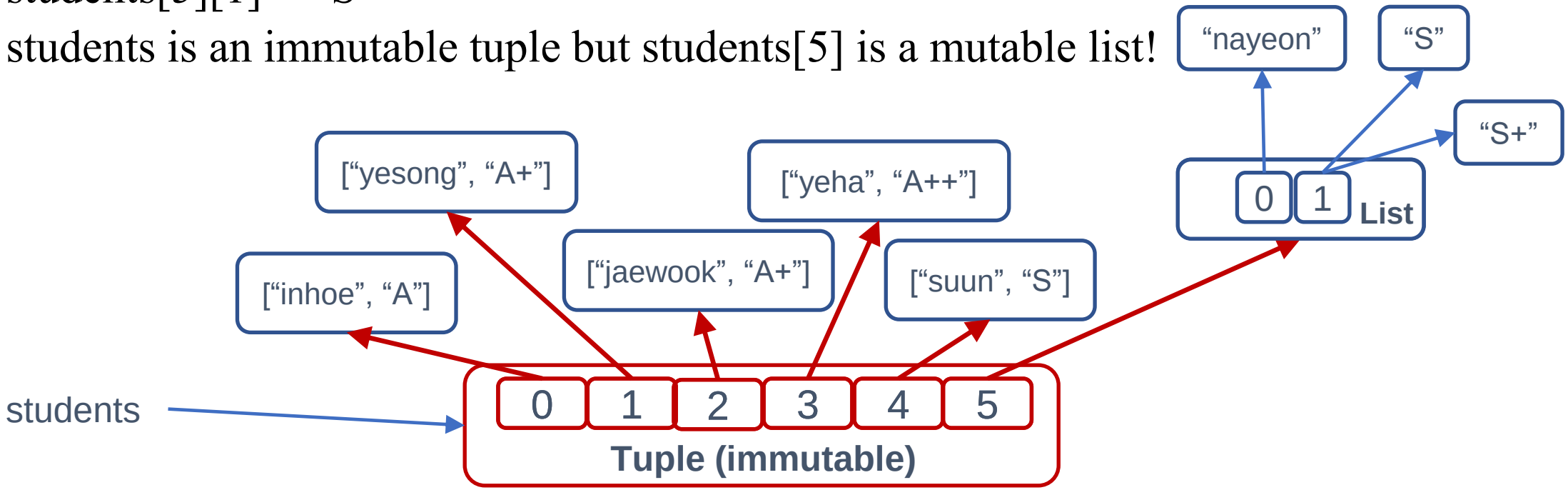
Mutability – Tuple

- Tuple (immutable container with mutable elements)
  - students[5][1] = “S+”



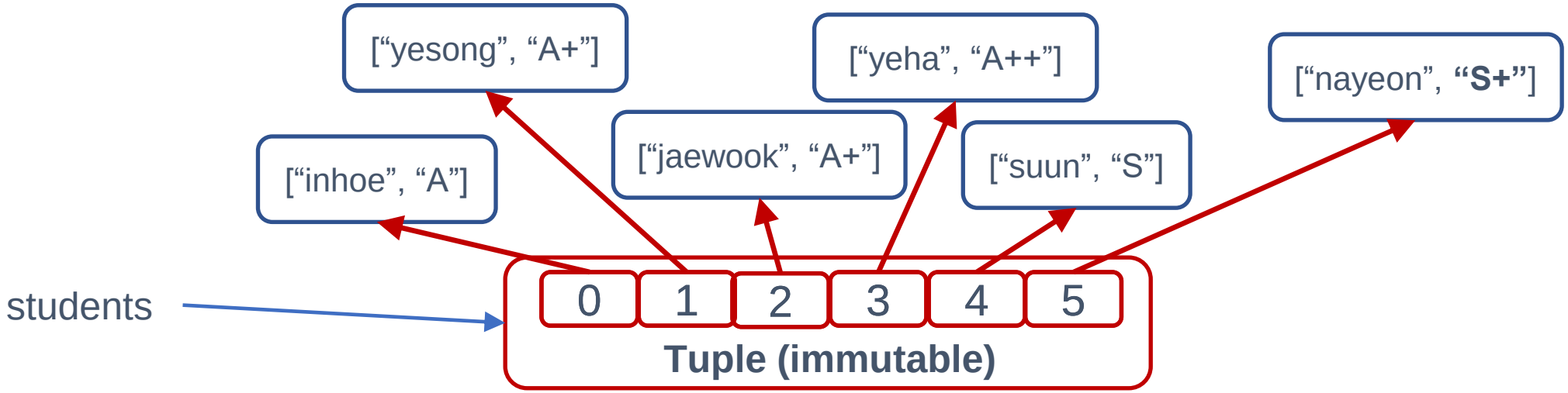
Mutability – Tuple

- Tuple (immutable container with mutable elements)
  - students[5][1] = “S+”
  - students is an immutable tuple but students[5] is a mutable list!



Mutability – Tuple

- Tuple (immutable container with mutable elements)
  - students[5][1] = “S+”
  - students is an immutable tuple but students[5] is a mutable list!

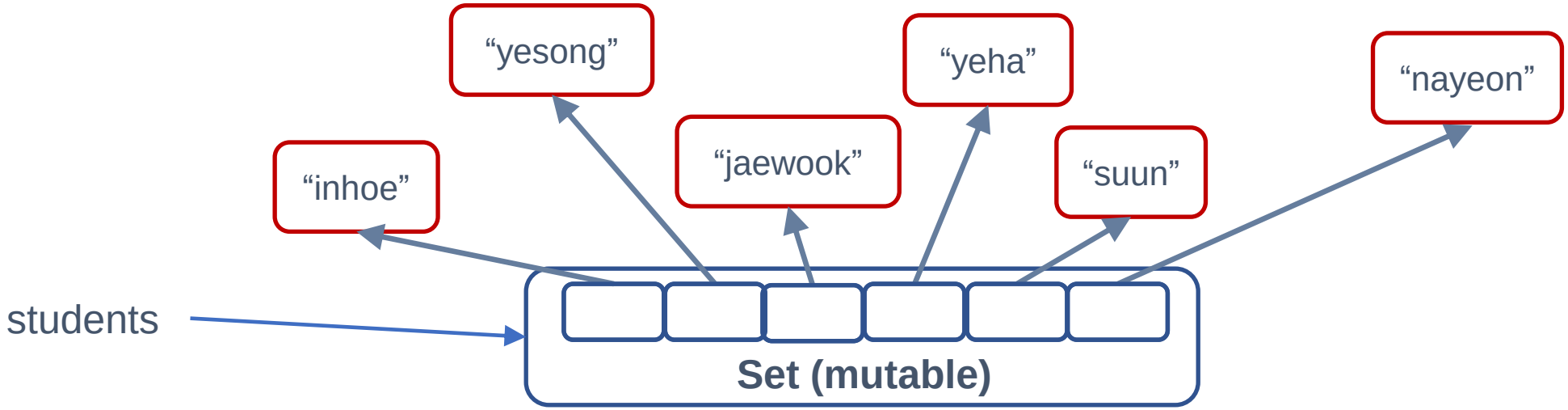


It is a valid operation because student[5] is still pointing at the **same** address (object)

Tuple is <u>immutable</u> but its elements can be <u>mutable</u> objects

Mutability – Set

- Set (mutable container with immutable elements)
  - Feel free to add (or remove) elements to (or from) students
  - But each element must point at an **immutable** (**hashable**) object



Summary

Collection	Mutable?	Ordered?	Use When…
str	No	Yes	You want to keep track of text
list	Yes	Yes	You want to keep track of an ordered sequence that you want update
tuple	No	Yes	You want to build an ordered sequence that you know won't change or that you want to use as a key in a dictionary or as a value in a set
set	Yes	No	You want to keep track of values, but order doesn't matter, and you don't want duplicates. The values must be immutable.
dictionary	Yes	Yes	You want to keep a mapping of keys to values. The keys must be immutable.

Thanks!