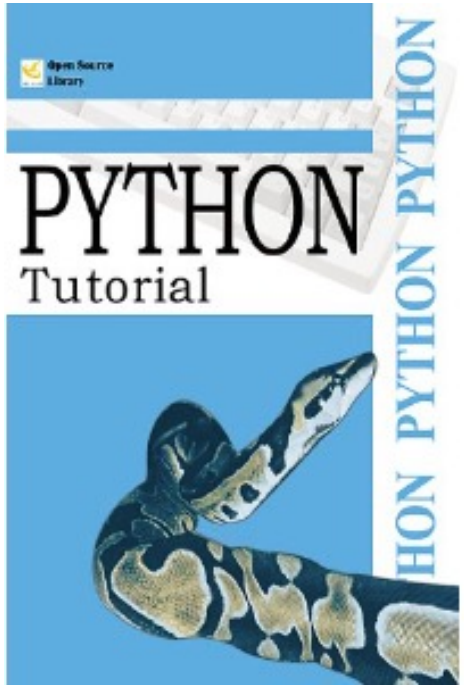
Data Structures in Python



Dra. Mª Dolores Rodríguez Moreno

Objectives

Specific Objectives

- Understanding the main data structures

Source

- https://docs.python.org/3/reference/datamodel.html
- Charles R. Severance (www.pythonlearn.com)
- https://ellibrodepython.com/
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)
- Abhijeet Anand on NumPy

Outline

- **Introduction**
- List
- Tuple
- Set
- Dictionary
- Enum
- Array

Introduction

- Programming focuses on the representation of information
- Simple data types like numbers, characters, and strings are straightforward to handle
- Real-world scenarios often involve more complexity
- A class can represent a single object, but we need to manage multiple objects effectively
- Representing complex data requires robust solutions
- This is where data structures come into play, providing powerful tools for organizing and storing information

Outline

- Introduction
- **List**
- Tuple
- Set
- Dictionary
- Enum
- Array

List

- List initialization: list = [item1, ..., itemN]
- Methods:
  - list.append(x)
  - list.insert(i, x)
  - list.remove(x)
  - list.pop()
  - list.index(x)
  - list.count(x)
  - list.sort()
  - list.reverse()

Example: List

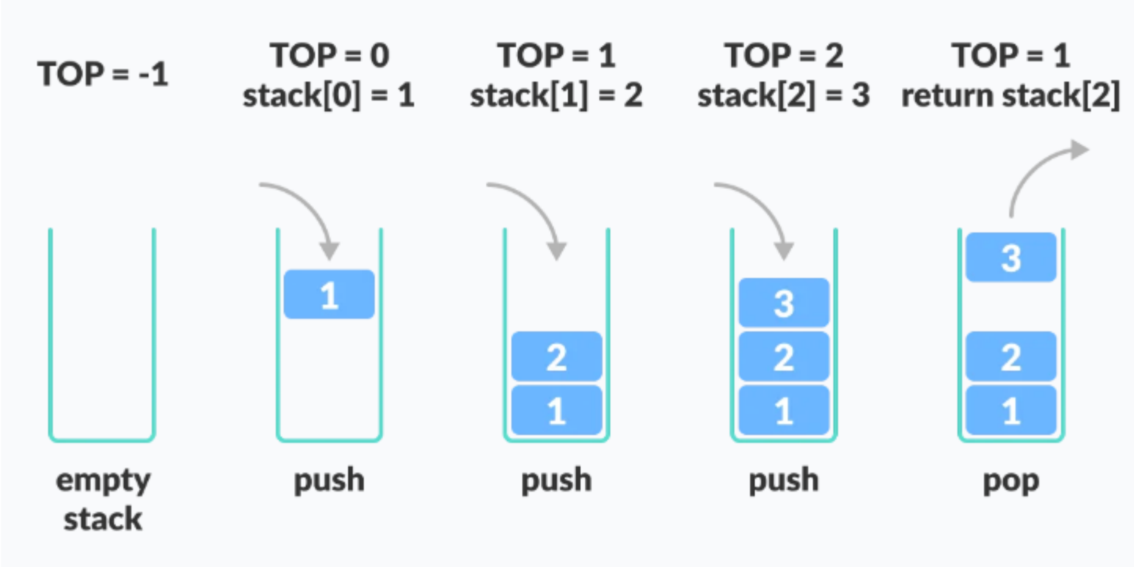
```
friends = ['Joseph', 'Glenn', 'Sally']

print(len(friends)) #3


print(list(range(len(friends))))

#[0,1,2]
```

Example: Loops in List

```
friends = ['Joseph', 'Glenn', 'Sally']

#Both loops do the same.

for friend in friends :

    print('Happy New Year:',  friend)

for i in range(len(friends)) :

    friend = friends[i]

    print('Happy New Year:',  friend)
```

Stack (as a list)

- Collection of elements that follows the LIFO (Last In, First Out) principle
- Last element added is the first one to be removed
- Main Operations:
  - push: adds an element to the top of the stack
  - pop: removes and returns the element at the top of the stack
  - peek: returns the element at the top of the stack without removing it
  - is_empty: checks if the stack is empty
  - size: returns the number of elements in the stack



source

Example: Stack

```python
# Initialize an empty stack
stack = []

# Push operation: Add elements to the top of the stack
stack.append(1)  # Stack: [1]

stack.append(2)  # Stack: [1, 2]

stack.append(3)  # Stack: [1, 2, 3]

# Pop operation: Remove and return the top element of the stack
top_element = stack.pop()  # Returns 3, Stack: [1, 2]

# Peek operation: Return the top element without removing it
top_element = stack[-1]  # Returns 2, Stack: [1, 2]

# Check if the stack is empty
is_empty = len(stack) == 0  # Returns False

# Get the size of the stack
stack_size = len(stack)  # Returns 2
```

Queue

- Collection of elements that follows the FIFO (First In, First Out) principle
- First element added is the first one to be removed
- Main Operations:
  - enqueue: adds an element to the end of the queue
  - dequeue: removes and returns the element at the front of the queue
  - front: returns the element at the front of the queue without removing it
  - is_empty: checks if the queue is empty
  - size: returns the number of elements in the queue
- Queue with List not efficient → Class deque

source

Example: Queue

```python
from collections import deque

# Initialize an empty queue

queue = deque()

# Enqueue operation: Add elements to the end of the queue

queue.append('Eric')    # Queue: ['Eric']

queue.append('John')    # Queue: ['Eric', 'John']

queue.append('Michael') # Queue: ['Eric', 'John', 'Michael']

# Dequeue operation: Remove and return the element at the front of the queue

front_element = queue.popleft()  # Returns 'Eric', Queue: ['John', 'Michael']
```

Exercise

- Create a class “Stack” with the functionality of the Stack previously explained

```
#Example Usage
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.pop())  # Output: 3
print(stack.peek())  # Output: 2
print(stack.is_empty())  # Output: False
print(stack.size())  # Output: 2
print(stack)
```

Outline

- Introduction
- List
- **Tuple**
- Set
- Dictionary
- Enum
- Array

Tuple

- Function much like a list with elements indexed starting at 0
- An **immutable** sequence of elements (similar to a string), typically used to store collections of heterogeneous data
- Main Operations:
  - Creation: can be created using parentheses () or without them
  - Access: elements can be accessed using indexing
  - Slicing: similar to lists, tuples support slicing
  - Unpacking: can be unpacked into individual variables

Example: Tuple

```
# Creating tuples

tup1 = (1, 2, 3)

tup2 = "a", "b", "c"

tup3 = (1, "a", 3.14)

# Accessing elements

first_element = tup1[0]  # Returns 1

# Slicing

sub_tuple = tup1[1:3]  # Returns (2, 3)

# Adding

print(tup1+tup2)  # (1, 2, 3, 'a', 'b', 'c')
```

Tuple: Unpacking

- You can assign a tuple of values to a tuple of variables in one step (i.e., it can be on the left side)
- Parentheses are optional in some contexts
- Python can infer that you are working with a tuple from the context
- Swapping values: is a common pattern without needing a temporary variable

Example with ()

```
(x, y) = (4, 'maria')

print(y)   # Output: maria
```

Example without ()

```
a, b = 99, 98

print(a)   # Output: 99
```

Example swapping

```
x, y = y, x
```

Example: Tuple (NOT to do)

```
x = (1, 2, 3)

x.sort()     #Traceback:AttributeError:  'tuple' object has no attribute 'sort'

x.append(5) #Traceback:AttributeError:  'tuple' object has no attribute 'append'

x.reverse() #Traceback:AttributeError:  'tuple' object has no attribute 'reverse’



dir(x) #['count', 'index’]

x.index(2) # Output: 1 (index of the first occurrence of 2 in the tuple x)

x.count(1) # Output: 1 (the number 1 appears once)
```

Tuples more efficient

- Since tuples don't need to be modifiable, they are simpler in design and translates to better memory usage and performance

- When your program needs temporary variables, reach for tuples instead of lists. Their efficient nature will give your code a boost!

Tuples are comparable

- You can compare tuples and other sequences
- If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ

Example with numbers

```
(2, 3, 4) < (3, 3, 4)          #True

(2, 3, 4000000) < (2, 4, 1) #True
```

Example with strings

```
('Anna', 'Bob') < ('Anna', 'Carol’) #True

('Anna', 'Bob’) > ('Alice', 'David’) #False
```

Outline

- Introduction
- List
- Tuple
- **Set**
- Dictionary
- Enum
- Array

Set

- An unordered collection of unique elements
- Mutables, for an immutable version, you can use *frozenset*
- It supports mathematical operations like union, intersection, and difference
- Main Operations:
  - Creation: sets can be created using curly braces {} or the set() function
  - Add: add an element to the set using add()
  - Remove: remove an element from the set using remove() or discard()
  - Membership: check if an element is in the set using the ***in*** keyword
  - Set Operations: union, intersection, and difference using operators or methods

Example: Set

```
# Creating sets

set1 = {1, 2, 3}

set2 = set([2, 3, 4])

# Adding elements

set1.add(4)  # set1 = {1, 2, 3, 4}

# Removing elements

set1.remove(1)  # set1 = {2, 3, 4}

# Membership check

is_member = 2 in set1  # Returns True
```

Example: Set

```
# Set operations

#set1 = {2, 3, 4}   #set2 = set([2, 3, 4])

# union_set = set1.union(set2)

union_set = set1 | set2  # Returns {2, 3, 4}

intersection_set = set1 & set2  # Returns {2, 3, 4}

difference_set = set1 - set2  # Returns set()

# Creating an immutable frozenset

frozen_set = frozenset([1, 2, 3])

# Attempting to modify the frozenset will raise an error

# frozen_set.add(4)  # Raises AttributeError
```

Outline

- Introduction
- List
- Tuple
- Set
- **Dictionary**
- Enum
- Array

What is a Collection

- Allows storage of multiple values in a single variable
- Contains multiple elements accessed via an index or key
- Convenient for managing groups of related data
- Types of Collections
  - List: ordered collection of elements
  - Dictionary: unordered collection of key-value pairs
- What is NOT a Collection?
  - Variables generally hold one value at a time
  - Assigning a new value overwrites the previous value

Dictionaries

- Also known as Associative Arrays, provide fast, flexible database-like operations
- Indexed by unique keys instead of numeric positions
- Other names in other languages
  - Dictionaries – Python, Objective-C, Smalltalk, REALbasic
  - Hashes – Ruby, Perl,
  - Maps – C++, Java, Go, Clojure, Scala, OCaml, Haskell
  - Property Bag - C#

Example: Dictionary

```
#Initialize an empty dictionary

purse = {}

#Add key-value pairs

purse['money'] = 15

purse['candy'] = 5

purse['tissues'] = 75

print(purse)  # Output: {'money': 15, 'candy': 5, 'tissues': 75}

#Modifying Dictionary Values. Access and update values using keys

print(purse['candy'])  # Output: 5

purse['candy'] += 2

print(purse)  # Output: {'money': 15, 'candy': 7, 'tissues': 75}
```

Missing Keys

- Accessing a non-existent key raises an error

example

```
c = {}

# print(c['unknown'])  # Raises KeyError

print('unknown' in c)  # Output: False
```

Counting with dictionaries

- Track occurrences of elements: get()
- Simplifies checking and updating dictionary entries

example

```
counts = {}

names = ['alice', 'bob', 'alice', 'eve', 'bob']

for name in names:

    counts[name] = counts.get(name, 0) + 1

print(counts)  # Output: {'alice': 2, 'bob': 2, 'eve': 1}
```

Example: Words Count

```
counts = {}
print('Enter a line of text:')
line = input()
words = line.split()
print('Words:', words)
print('Counting...')
for word in words:
    counts[word] = counts.get(word, 0) + 1
print('Counts:', counts)
```

Iterating

- Even if there is not order, we can iterate on dictionaries

example

```
counts = {'alice': 1, 'bob': 42, 'eve': 100}

for key in counts:

    print(key, counts[key])

# Output:

# alice 1

# bob 42

# eve 100
```

Retrieving Keys, Values, and Items

- Get lists of keys, values, or key-value pairs

example

```
dict1 = {'alice': 1, 'bob': 42, 'eve': 100}

print(list(dict1))  # Output: ['alice', 'bob', 'eve']

print(dict1.keys())  # Output: dict_keys(['alice', 'bob', 'eve'])

print(dict1.values())  # Output: dict_values([1, 42, 100])

print(dict1.items())  # Output: dict_items([('alice', 1), ('bob', 42),
('eve', 100)])
```

Example: Two iteration values. Loop through key-value pairs using two variables

```
jjj = {'alice': 1, 'bob': 42, 'eve': 100}

for key, value in jjj.items():

    print(key, value)

# Output:

# alice 1

# bob 42

# eve 100
```

Exercise

- Create a program that allows a teacher to input student names and their grades
  - Create an empty dictionary to store student names (keys) and their grades (values)
  - Write a loop that asks the user to input a student's name and grade. Add these to the dictionary. Continue until the user enters 'done'
  - Print out all student names and their grades
  - Calculate and print the average grade of the class
  - Find and print the name(s) of the student(s) with the highest grade

Summary

- List: li = [1, 2, 3]
- Tuple: tu = (1, 2, 3)

  tu = 1, 2, 3
- Set: se = {1, 2, 3}
- Dictionary: dic = {'abc': 1, 'bca': 2}

Outline

- Introduction
- List
- Tuple
- Set
- Dictionary
- **Enum**
- Array

What is an Enum?

- An enumeration is a set of symbolic names (members) bound to unique, constant values
- Useful when you have a collection of related constant values like days of the week, states in a process, etc.
- Part of the standard library: (*from enum import Enum)*
- Introduced in Python 3.4
- Purpose: define named constants that can be compared and iterated over

Why Use Enums?

- Code Readability: Enums make code more expressive
- Prevent Magic Numbers: No need to remember arbitrary values
- Safety: Enum members are immutable and unique
- Maintainability: Simplifies updates when constant values change

Example: Definition

```
from enum import Enum

class Weekday(Enum):

    MONDAY = 1

    TUESDAY = 2

    WEDNESDAY = 3

    THURSDAY = 4

    FRIDAY = 5
```

Example: Accessing

```
print(Weekday.MONDAY)
print(Weekday.MONDAY.name)
print(Weekday.MONDAY.value)

#Output: Weekday.MONDAY, 'MONDAY', 1
```

Iterating

Example:

```
for day in Weekday:
    print(day)



#Output: Weekday.MONDAY, Weekday.TUESDAY...
```

Comparing

Example:

```
if Weekday.MONDAY == Weekday.TUESDAY:
    print("Same day")
else:
    print("Different days")
```

Enums with Automatic Values

- Use auto() to automatically assign increasing values starting from 1

Example: Auto

```
from enum import Enum, auto

class Weekday(Enum):

    MONDAY = auto()

    TUESDAY = auto()

    WEDNESDAY = auto()

    THURSDAY = auto()

    FRIDAY = auto()
```

Useful Enum Methods

- Enum members can be accessed by name or value:

Example:

```
Weekday['MONDAY']
Weekday(1)
```

- You can get a dictionary of all enum member: *__members__*

Example:

```
print(Weekday.__members__)
```

Exercise

- Represent different states in a task management system
- Create a function that prints a message depending on the values
- For example:

  NOT_STARTED

  IN_PROGRESS

  COMPLETED

  BLOCKED
- Call: print_task_status(“Exam corrections", TaskStatus. NOT_STARTED)

#The task 'Exam corrections' has not started yet.

Outline

- Introduction
- List
- Tuple
- Set
- Dictionary
- Enum
- **Array**

Array

- An array is a collection of items stored at memory locations
- Python has an “array” library but it is not used much
  - import array
  - array_name = array.array(typecode, [elements])
- Instead, *numpy* library is used for creating and handling arrays
- Key Characteristics:
  - Fixed size
  - Homogeneous elements (all elements are of the same type)
  - Efficient numerical operations

NumPy

- NumPy stands for Numerical Python
- NumPy is a Python library used for working with arrays
- NumPy is the fundamental package for scientific computing in Python
- It has functions for working in the domain of linear algebra, Fourier transform, and matrices
- NumPy was created in 2005 by Travis Oliphant
- It is an open-source project and can be used freely

Why Use NumPy?

- In Python, we have lists that serve the purpose of arrays, but they are slow to process
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists
- NumPy's array class is called ndarray
- It provides a lot of supporting functions that make working with ndarray very easy
- Arrays are very frequently used in data science, where speed and resources are very important

1-D Array

- An array that has 0-D arrays as its elements is called a uni-dimensional or 1-D array

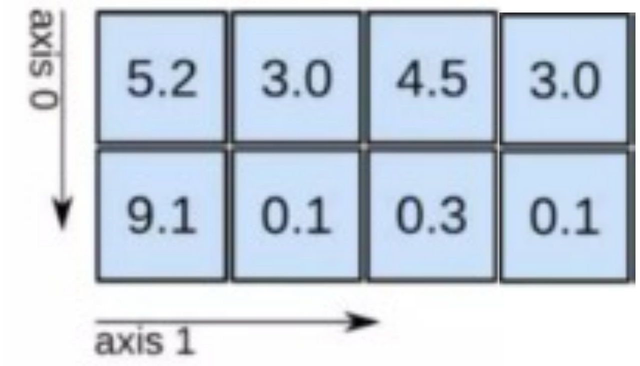
- Dimension: 7

Example: numpy – 1D

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

#Accessing Elements in 1D:

print(arr[0])  # Output: 1

#Negative Indexing:

print(arr[-1])  # Output: 5

#updating Elements

arr[1] = 10 # Output: [ 1 10  3  4  5]

#Appending Elements:

arr = np.append(arr, 6) # Output: [ 1 10  3  4  5  6]
```

2-D Array

- An array that has 1-D arrays as its elements is called a 2-D array
- These are often used to represent a matrix or 2nd order tensors



- Dimension: 2 X 4

Example: numpy – 2D

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

#Accessing Elements in 2D:

print(arr_2d[1, 2])  # Output: 6

#Slicing:

print(arr_2d[0:2, 1:3])

#updating Elements

arr_2d[0, 0] = 10

#Appending Elements:

arr_2d = np.append(arr_2d, [[10, 11, 12]], axis=0)
```

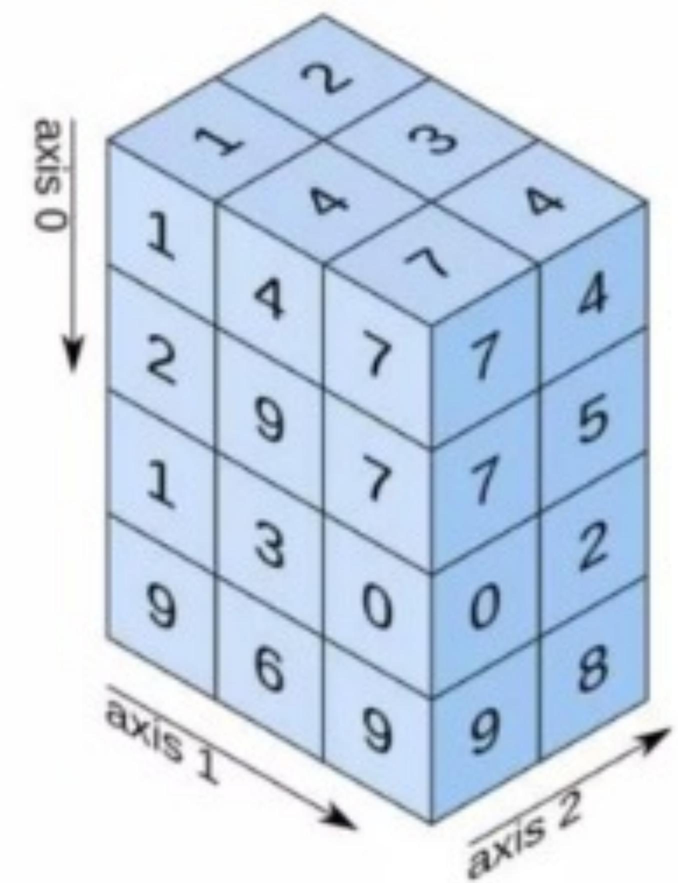
```
[[2 3]
 [5 6]]
```

```
[[10  2  3]
 [ 4  5  6]
 [ 7  8  9]]
```

```
[[10  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

3-D Arrays

- An array that has 2-D arrays (matrices) as its elements is called a 3-D array
- These are often used to represent a 3rd order tensor



Example: numpy – 3D

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

#Accessing Elements

print(arr[1, 0, 1])  # Output: 8

#Slicing

print(arr[:, 1, :])

#Updating Elements:

arr[0, 1, 1] = 20

#Appending Elements

arr = np.append(arr, [[[13, 14, 15], [16, 17, 18]]], axis=0)
```

```
[[ 4  5  6]
 [10 11 12]]
```

```
[[[ 1  2  3]
  [ 4 20  6]]
 [[ 7  8  9]
  [10 11 12]]
 [[13 14 15]
  [16 17 18]]]
```

Example: numpy – 3D

```
#Inserting Elements

arr = np.insert(arr, 1, [[[-1, -2, -3], [-4, -5, -6]]], axis=0)
```

```
[[[  1   2   3]
  [  4  20   6]]
 [[ -1  -2  -3]
  [ -4  -5  -6]]
 [[  7   8   9]
  [ 10  11  12]]
 [[ 13  14  15]
  [ 16  17  18]]]
```

```
#Deleting Elements

arr = np.delete(arr, 1, axis=0)
```

```
[[[ 1  2  3]
  [ 4 20  6]]
 [[ 7  8  9]
  [10 11 12]]
 [[13 14 15]
  [16 17 18]]]
```

Initialization

- np.array([1, 2, 3]): 1D array #You can specify type: np.array([1, 2, 3], dtype = float)
- np.array([[1, 2, 3], [4, 5, 6]]): 2D array
- np.arange(start, stop, step): Create an array (from start to stop with increments "steps"→ it will have steps elements)
- np.linspace(0, 2, 9): Add evenly spaced values between intervals to array of length
- np.zeros((1, 2)): Create an array filled with zeros (1x2)
- np.ones((1, 2)): Create an array filled with ones
- np.random.random((5, 5)): Create a random array (5x5)
- np.empty((2, 2)): Create an empty array (2x2)

Array Properties

- array.shape: Dimensions (Rows, Columns)
- len(array): Length of Array
- array.ndim: Number of Array Dimensions
- array.size: Number of Array Elements
- array.dtype: Data Type
- type(array): Type of Array

Copying & Sorting

- np.copy(array): Creates a copy of array
- array.sort(): Sorts an array
- array.sort(axis=0): Sorts axis of array

Operations

- np.add(x, y): Addition
- np.subtract(x, y): Subtraction
- np.divide(x, y): Division
- np.multiply(x, y): Multiplication
- np.sqrt(x): Square Root
- np.sin(x): Element-wise sine
- np.cos(x): Element-wise cosine
- np.log(x): Element-wise natural log
- np.dot(x, y): Dot producto (escalar product)
- np.roots([1, 0, -4]): Roots of given polynomial coefficients

Exercise

- Create a 2D Array to represent the grades of 3 students in 3 subjects
  - Retrieve the grade of the second student in the third subject
  - Modify the grade of the first student in the first subject
  - Add a new student with grades.
  - Insert a new subject at the second position
  - Remove the second student
  - Change the shape of the array to represent 2 students with 4 subjects each