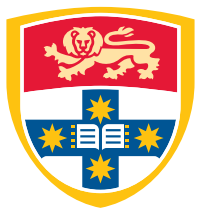
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



Copyright Warning

Lecture 24: Multi-dimensional arrays

Representing data in multiple dimensions

Multiple dimensions

Declaring a multi-dimensional list is easy

Each dimension has another index []

e.g. 1D `line = np.zeros((5)).astype('int32')`

e.g. 2D `grid = np.zeros((2,2)).astype('int32')`

e.g. 3D `volume = np.zeros((3,3,3)).astype('int32')`

e.g. 6D `hyperspace = np.zeros((3,3,3,3,3,3).astype('float64')`

We will keep it simple and use 2D. The same ideas apply when extending to higher dimensions

2 dimensions initialisation

Initialise a 2D array

```
1    grid = np.ones((2,2)).astype('int32')
```

creates a new array of $(2 \times 2 =)$ 4 `int32`s

We access them in the same way as for 1-dimensional arrays:

```
1  grid[0][0] = 7
2  grid[0][1] = 7
3  grid[1][0] = 7
4  grid[1][1] = 7
```

Which we can imagine, it might look like this:

$$\begin{pmatrix} 7 & 7 \\ 7 & 7 \end{pmatrix}$$

2 dimensions traversal

We need a procedure to visit each element in a 2D array and print the value

Such a procedure is termed a *traversal*

```
1  # initialise
2  grid = np.zeros((2,2)).astype('int32')
3
4  # our traversal to assign the initial value
5  for x in range(2):
6      for y in range(2):
7          grid[ x ] [ y ] = 1
8
9  # our traversal to display values
10 for x in range(2):
11     for y in range(2):
12         print(grid[x][y], end=" ")
13     print("")
```

Two-D Arrays — example program

```
1  ''' Create and display a 2D array '''
2  import numpy as np
3  # initialise
4  grid = np.zeros((5,10)).astype('int32')
5
6  # our traversal to assign the initial value
7  for x in range(5):
8      for y in range(10):
9          grid[ x ] [ y ] = 1
10
11 # our traversal to display values
12 for x in range(5):
13     for y in range(10):
14         print(grid[x][y], end=" ")
15     print("")
```

Running `TwoDArray`

```
~> python TwoDArray.py
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
~>
```

. . . which is rather dull, but works.

Printing 2D array contents

How we imagine the memory contents and how we print those contents to screen are two different ideas

Here is *somebody's* print out of a 2D array:

```
0 3
7 0
```

What are the values of 2D array elements [0][1] and [1][0]?

This depends on which traversal is used and if they are the same for both setting and printing.

Which traversal

Representation:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Output:

```
1 2
3 4
```

```
1  def init_trace(grid, size):
2      count = 1
3      for _?_ in range(size):
4          for _?_ in range(size):
5              grid[ _?_ ] [ _?_ ] = count
6              count += 1
7
8  def print(grid, size):
9      for _?_ in range(size):
10         for _?_ in range(size):
11             print(grid[ _?_ ][ _?_ ], end=" ")
12         print("")
```

Which traversal

Representation:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

Output:

```
1 2
3 4
```

```
1  def init_trace(grid, size):
2      count = 1
3      for _?_ in range(size):
4          for _?_ in range(size):
5              grid[ _?_ ] [ _?_ ] = count
6              count += 1
7
8  def print(grid, size):
9      for _?_ in range(size):
10         for _?_ in range(size):
11             print(grid[ _?_ ][ _?_ ], end=" ")
12         print("")
```

Which makes more sense? i.e. what is the correct order to visit each element?

Consider this example:

```
1  ''' Create and display a multiplication table :) '''
2  import numpy as np
3  grid = np.zeros((5,10)).astype('int32')
4
5  for i in range(5):
6      for j in range(10):
7          grid[i][j] = i * j
8
9  for i in range(5):
10     for j in range(10):
11         print(grid[i][j], end=" ")
12     print("")
```

Running TwoDArrayFilled

```
~> python TwoDArrayFilled.py
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
~>
```

Size of a multi-dimensional array

There's an obvious question that you might be wondering about now: what is the value of `grid.length` in the above?

Let's find out. With these lines inserted into our program...

```
1     print(len(grid))
2     print(len(grid[2])
```

we get

```
~> python TwoDArrayFilled.py
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
5
10
```

2 dimensions non-square

Initialise a 2D array

```
np.zeros((4, 2)).astype('float16')
```

creates a new array of (2 × 4 =) 8 elements of type `float16`'s

What does it look like? 2 × 4 or 4 × 2?

2 × 4

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

4 × 2

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

2 dimensions non-square (cont.)

```
1 import numpy as np
2
3 a = np.zeros((2, 4)).astype('int8')
4 print("2 x 4 is:\n" + str(a))
5
6 b = np.zeros((4, 2)).astype('int8')
7 print("4 x 2 is:\n" + str(b))
```

```
2 x 4 is:
[[0. 0. 0. 0.]
[0. 0. 0. 0.]]
4 x 2 is:
[[0. 0.]
[0. 0.]
[0. 0.]
[0. 0.]]
```

Arrays of arrays

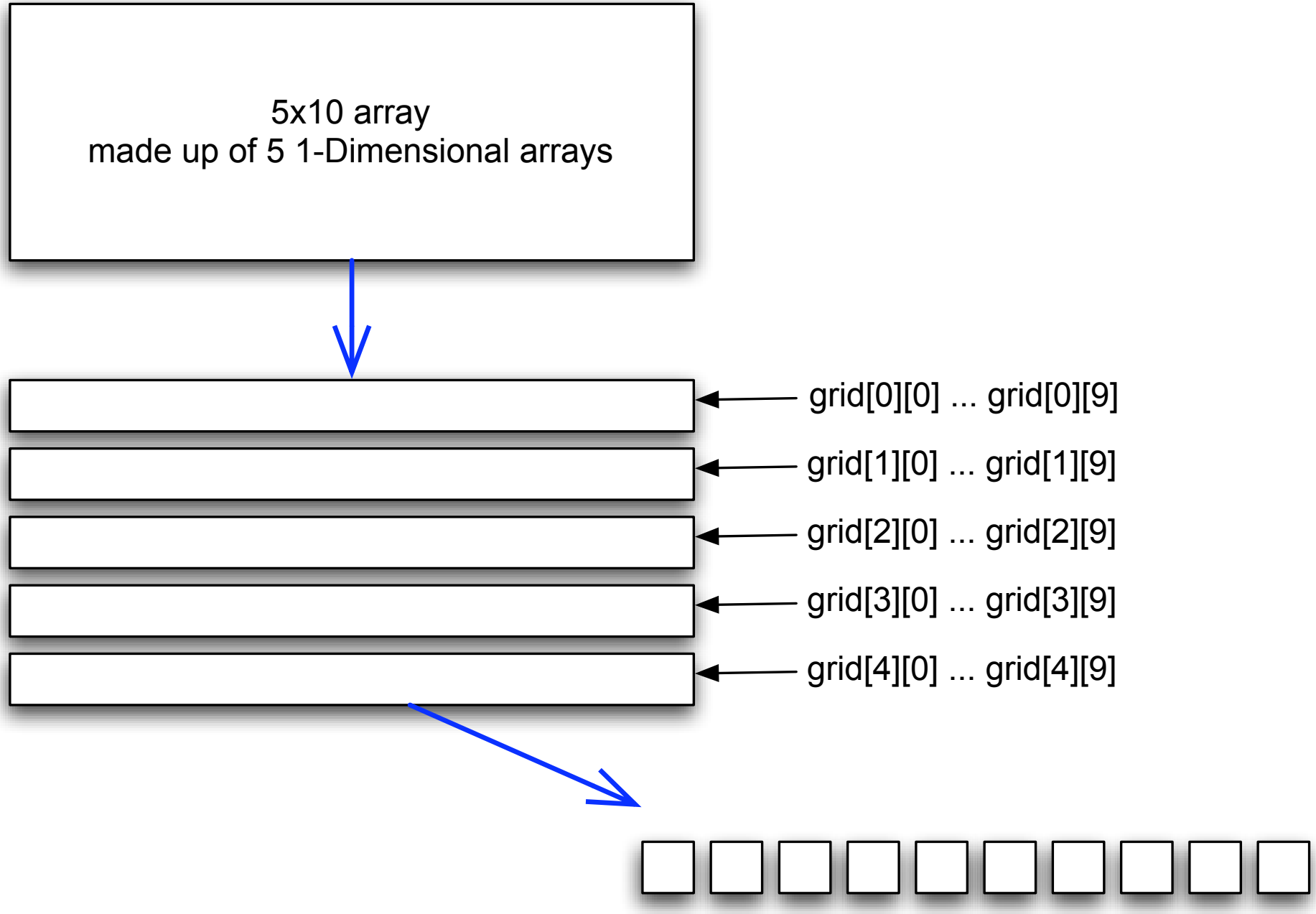
To understand what's going on you need to know an important fact: multidimensional arrays are stored as arrays of arrays.

An array of k dimensions is stored as a 1-dimensional array of arrays of $(k-1)$ *dimensions each.*

```
grid = np.zeros((5,10)).astype('int32')
print("dim 1:  " + str(len(grid)))
print("dim 2:  " + str(len(grid[0])))
```

So in the example above, `grid.length` is the length of the first dimension: it's the number of 1-dimensional arrays

`grid[0][0]...grid[0][9]` is the first "row", `grid[1][0]...grid[1][9]` the second, etc.


5x10 array
made up of 5 1-Dimensional arrays
grid[0][0] ... grid[0][9]
grid[1][0] ... grid[1][9]
grid[2][0] ... grid[2][9]
grid[3][0] ... grid[3][9]
grid[4][0] ... grid[4][9]

Exception with 2D arrays

```
1 grid = np.zeros((5,10)).astype('int32')
2 grid[3][2000] = 1 # oops
```

```
Traceback (most recent call last):
   File "badarray.py", line 4, in <module>
      grid[3][2000] = 1 # oops
IndexError: index 2000 is out of bounds for axis 0 with size 10
```

There's an *exception thrown* when python tries to access `grid[3][2000]`, as the correct range of indices for `c[3]` is `c[3][0]` . . . `c[3][9]`.

Summary: Multidimensional array

The ordering of rows, columns, (depth?) and other dimensions should be very carefully considered.

Traversals through the multi-dimensional array should always be the same for the same effect.

Arrays have strict types for elements, some of which are strict bit width (you can still have strings as types)

Arrays are best used for achieving *memory* efficiency.

That's all, folks!

This is the end of the lecture material covered in Week 12.