C Arrays and Python Lists

2 Oct 2009
CMPT140
Dr. Sean Ho
Trinity Western University

What's on today

- Type hierarchy, M2/C vs. Python
    - Enumeration types
- Python lists vs. M2/C arrays
- Lists as function parameters
- Multidimensional arrays/lists

M2 type hierarchy (partial)

- Atomic types
  - Scalar types
    - Real types (REAL, LONGREAL)
    - Ordinal types (CHAR)
      - Whole number types (INTEGER, CARDINAL)
      - Enumerations (§5.2.1) (BOOLEAN)
      - Subranges (§5.2.2)
- Structured (aggregate) types
  - Arrays (§5.3)
    - Strings (§5.3.1)
  - Sets (§9.2-9.6)
  - Records (§9.7-9.12)
- Also can have user-defined types

Python type hierarchy (partial)

- Atomic types
  - Numbers
    - Integers (int, long, bool): 5, 500000L, True
    - Reals (float) (only double-precision): 5.0
    - Complex numbers (complex): 5+2j
- Container (aggregate) types
  - Immutable sequences
    - Strings (str): "Hello"
    - Tuples (tuple): (2, 5.0, "hi")
  - Mutable sequences
    - Lists (list): [2, 5.0, "hi"]
  - Mappings
    - Dictionaries (dict): {"apple": 5, "orange": 8}

Enumeration types in M2 / C

```
TYPE
    DayName = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR
    today : DayName;
BEGIN
    today := Mon;
```

- We could have used CARDINALs instead (indeed, the underlying implementation does)
  - But the logical semantic of today's type is a DayName type, not a CARDINAL
- Can be thought of as Sun=0, Mon=1, Tue=2, ...
- No explicit enumeration scheme in Python

C Arrays

- Most languages (C, M2, Java, etc.) have arrays:
  - C: float myWages[5] = {0., 25.75, 0., 0., 0.};
  - M2: myWages: ARRAY [0..4] OF REAL;
- Compound data type, sequential storage
  - Fixed length: must declare length (5)
  - Uniform type: same type for all elements
  - Static type: can't change type of elements
- Indexing: myWages[2] = 15.85;

Python Lists

- Python doesn't have a built-in type exactly like arrays, but it does have lists:

```
nelliesWages = [0.0, 25.75, 0.0, 0.0, 0.0]
nelliesWages[1]                # returns 25.75
```

- Under the covers, Python often implements lists using arrays, but lists are more powerful:
  - Can change length dynamically
  - Can store items of different type
  - Can delete/insert items mid-list
- For now, we'll treat Python lists as arrays

Using lists

- We know one way to generate a list: range()

```
range(10)      # returns [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Or create directly in square brackets:

```
myApples = ["Fuji", "Gala", "Red Delicious"]
```

- We can iterate through a list:

```
for idx in range(len(myApples)):
    print "I like", myApples[idx], "apples!"
```

- Even easier:

```
for apple in myApples:
    print "I like", apple, "apples!"
```

Lists as parameters

```
def average(vec):
    """Return the average of the vector's values.
    pre: vec should have scalar values (float, int)
      and not be empty.
    """
    sum = 0
    for elt in vec:
        sum += elt
    return sum / len(vec)

myList = range(9)
print average(myList)                    # prints 4
```

- What happens when we pass an empty array? An atomic value?

Type-checking list parameters

- Since Python is dynamically-typed, the function definition doesn't specify what type the parameter is, or even that it needs to be a list
  - Easy way out: state expected type in precondition
  - Or do type checking in the function:

```
if type(vec) != type([]):
    print "Need to pass this function a list!"
    return
```

  - May also want to check for empty lists:

```
if len(vec) == 0:
```

- for, len(), etc. don't work on atomic types

Array parameters in M2/C/etc.

- In statically-typed languages like M2, C, etc., the procedure declaration needs to specify that the parameter is an array, and the type of its elements:
  - M2:

    ```
    PROCEDURE Average(myList: ARRAY of REAL) :
      REAL;
    ```

  - C:

    ```
    float average(float* myList, unsigned int len) {
    ```

- In M2, HIGH(myList) gets the length
- In C, length is unknown (pass in separately)

Multidimensional arrays

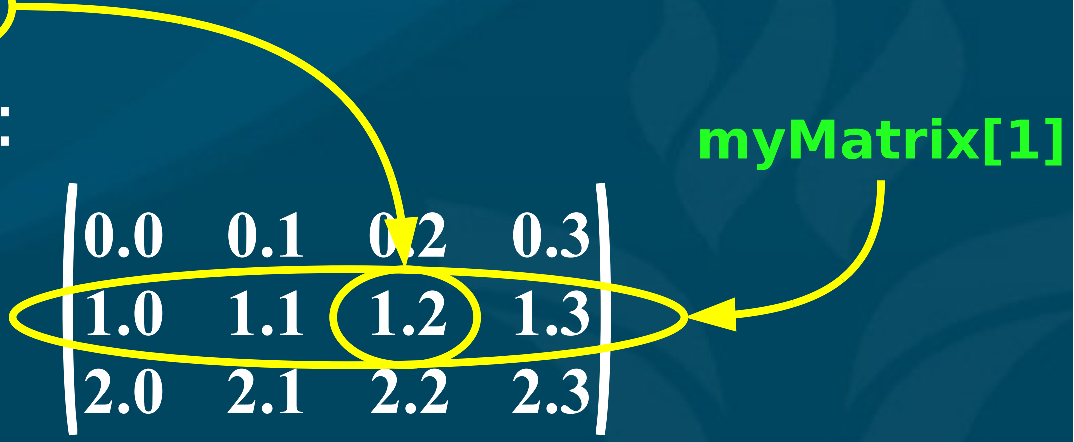
- Multidimensional arrays are simply arrays of arrays:

```
myMatrix = [ [0.0, 0.1, 0.2, 0.3],
             [1.0, 1.1, 1.2, 1.3],
             [2,0, 2.1, 2.2, 2.3] ]
```

- Accessing:

```
myMatrix[1][2] = 1.2
```

- Row-major convention:



myMatrix[1]

$$\begin{vmatrix} 0.0 & 0.1 & 0.2 & 0.3 \\ 1.0 & 1.1 & 1.2 & 1.3 \\ 2.0 & 2.1 & 2.2 & 2.3 \end{vmatrix}$$

Iterating in multidim arrays

```
def matrix_average(matrix):
    """Return the average value from the 2D
      matrix.
    Pre: matrix must be a non-empty 2D array of
      scalar values."""
    sum = 0
    num_entries = 0
    for row in range(len(matrix)):
        for col in range(len(matrix[row])):
            sum += matrix[row][col]
        num_entries += len(matrix[row])
    return sum / num_entries
```

- What if rows are not all equal length?