1. **Processing Lists: builtins**

- **`sum(x)`**: Adds up all the elements in the list `x`.
  - They must all be numbers!

- **`min(x)`** or **`max(x)`**: Find the min/max value in list `x`.
  - They use the same ordering as `sort()`.

- **`range(a, b, c)`**: Produces `\[a, a+c, a+2c, \ldots, a+c((b-a)/c)\]`.
  - Starts at `a`, increases by `c` each time, until `b` (or less).
  - The argument `c` is optional; `c = 1` by default.

- **`list(x)`**: Converts `x` (such as a string) to a list.
  - Example: `list('mimsy')` produces `['m', 'i', 'm', 's', 'y']`.

2. **The Map Function**

- **`map(function, list)`**
  - Function has to have exactly 1 parameter.
  - Otherwise, get an error.
  - Returns a new list.
  - Does the same thing as `def map(f, x):`
    - `result = [] # empty list`
    - `for y in x:`
      - `result.append(f(y))`
    - `return result`

  - `map(f, x)` produces `\[f(x[0]), f(x[1]), \ldots, f(x[n-1])\]`.

3. **Two Dimensional Lists**

   - **Table of Data**
   - **Images**
   - **Storing them as lists (row-major order)**

   - Each row, col has a value
   - Each row, col has an RGB value

4. **Overview of Two-Dimensional Lists**

   - **Access value at row 3, col 2:**
     - `d[3][2]`.

   - **Assign value at row 3, col 2:**

   - **An odd symmetry**
     - Number of rows of `d`: `len(d)`
     - Number of cols in row `r` of `d`: `len(d[r])`

5. **How Multidimensional Lists are Stored**

   - **b = [[9, 6, 4], [5, 7, 7]]**
     - `b` holds name of a one-dimensional list
     - `len(b)` elements
     - Its elements are (the names of) 1D lists
     - `b[i]` holds the name of a one-dimensional list (of ints)
     - `len(b[i])` elements

6. **Image Data: 2D Lists of Pixels**

   - `b[0][0]` is a white pixel
   - `red = [255, 255, 255]`
Slices and Multidimensional Lists

• Only “top-level” list is copied.
• Contents of the list are not altered
• \( b = \{[9, 6], [4, 5], [7, 7]\} \)

\[
x = b[:2]
\]

Functions and 2D Lists

```python
def transpose(table):
    # Precondition: table is a (non-ragged) 2d List
    numrows = len(table)
    numcols = len(table[0])

    result = []
    # Result accumulator
    for m in range(numcols):
        row = []
        # Single row accumulator
        for n in range(numrows):
            row.append(table[n][m])
        # Build up row
        result.append(row)
    # Add result to table
    return result
```

Dictionaries (Type dict)

<table>
<thead>
<tr>
<th>Description</th>
<th>Python Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>• List of key-value pairs</td>
<td>• Create with format:</td>
</tr>
<tr>
<td>• Keys are unique</td>
<td>{k1:v1, k2:v2, ...}</td>
</tr>
<tr>
<td>• Values need not be</td>
<td>• Keys must be non-mutable</td>
</tr>
<tr>
<td>• Example: net-ids</td>
<td>* ints, floats, bools, strings</td>
</tr>
<tr>
<td>• net-ids are unique (a key)</td>
<td>* Not lists or custom objects</td>
</tr>
<tr>
<td>• names need not be (values)</td>
<td>• Values can be anything</td>
</tr>
<tr>
<td>• Example:</td>
<td>• Example:</td>
</tr>
<tr>
<td>js1 is John Smith (class '13)</td>
<td>d = {'js1':'John Smith',</td>
</tr>
<tr>
<td>js2 is John Smith (class '16)</td>
<td>'js2':'John Smith',</td>
</tr>
<tr>
<td>• Many other applications</td>
<td>'wmw2':'Walker White'}</td>
</tr>
</tbody>
</table>

Using Dictionaries (Type dict)

• Access elts. like a list
  * \( d['js1'] \) evaluates to ‘John’
  * But cannot slice ranges!
• Dictionaries are **mutable**
  * Can reassign values
  * \( d['js1'] = 'Jane' \)
  * Can add new keys
  * \( d['aa1'] = 'Allen' \)
  * Can delete keys
  * del \( d['wmw2'] \)

Using Dictionaries and For-Loops

• Dictionaries != sequences
  * Cannot slice them
• **Different** inside for loop
  * Loop variable gets the key
  * Then use key to get value
• Has **methods** to convert dictionary to a sequence
  * Seq of keys: \( d.keys() \)
  * Seq of values: \( d.values() \)
  * key-value pairs: \( d.items() \)

for k in d:
    # Loops over keys
    print k  # key
    print d[k]  # value

# To loop over values only for v in d.values():
    print v  # value

See grades.py