



COMP 1023 Introduction to Python Programming
Collections - Container Data Types (Part I)
Dr. James Cheung, Dr. Wilfred Ng, Dr. Desmond Tsoi

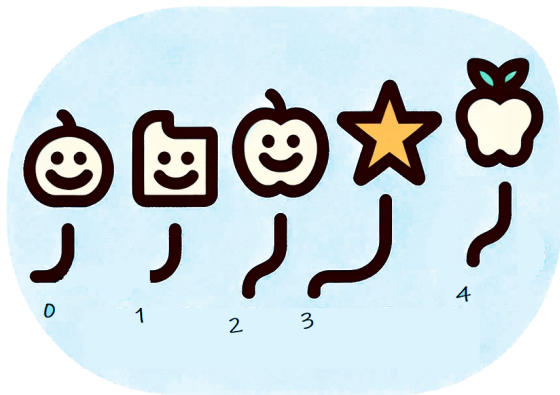
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China



Collections in Python

- **Collections** are fundamental data structures in Python that allow you to **store and manage multiple items efficiently**.
- They provide ways to **organize data, enabling easy access, modification, and manipulation**.
- Different types of collections serve different purposes:
 - **Lists**: Ordered, mutable collections that can hold mixed data types.
 - **Tuples**: Ordered, immutable collections that are typically used to group related data.
 - **Sets**: Unordered collections of unique elements, useful for membership testing and eliminating duplicates. **(Self-study)**
 - **Dictionaries**: Ordered collections (i.e., those that preserve insertion order) of key-value pairs allow for fast lookups and data storage based on unique keys.
- Understanding these collections is crucial for effective programming in Python, as they provide the foundation for data organization and manipulation.

Lists



Introduction

- **Programs** often need to **store a large number of values**.
- For instance, suppose you need to read 100 numbers, compute their average, and find out how many of the numbers are above the average.
- Your program first reads the numbers, computes their average, and then compares each number with the average to determine whether it is above the average.
- To accomplish this task, all numbers must be stored in variables. This would require **creating 100 variables** and writing almost identical code 100 times.
- Writing a program this way is impractical. So, how do you solve this problem?
- Python provides a data type called a **list** that **stores a sequential collection of elements**.



Not a Good Idea!

```
def main():
    v1 = float(input("Enter the 1st number: "))
    v2 = float(input("Enter the 2nd number: "))
    v3 = float(input("Enter the 3rd number: "))
    ...
    v99 = float(input("Enter the 99th number: "))
    v100 = float(input("Enter the 100th number: "))

    avg = (v1 + v2 + v3 + v4 + v5 +
           v6 + v7 + v8 + v9 + v10 +
           ...
           v96 + v97 + v98 + v99 + v100) / 100
    print(f"The average is {avg}.")

if __name__ == "__main__":
    main()
```



List Basics

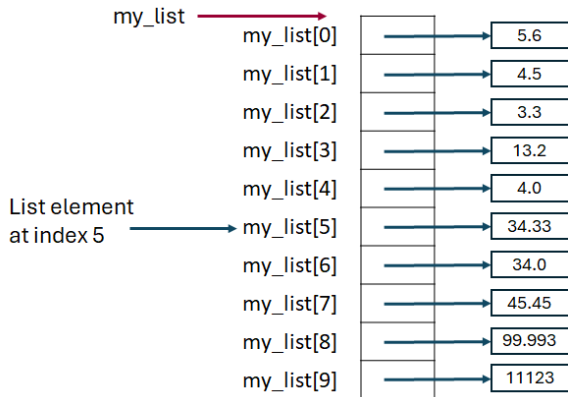
- A **list** is a **sequence of any elements (of any type)**.
- To create a list, you can use the following syntax:

```
list1 = [] # Create an empty list
list2 = [2, 3, 4] # Create a list with elements 2, 3, 4
list3 = ["red", "green", "blue"] # Create a list with strings
list4 = list(range(3, 6)) # Create a list with elements 3, 4, 5 using constructor
list5 = list("abcd") # Create a list with characters a, b, c, d
# using constructor
list6 = [2, "three", 4] # Create a list with elements of mixed types
```



List Demonstration

```
my_list = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]
```



- The list `my_list` has 10 elements with indexes from 0 to 9.

Common Error

- Accessing a list **out of bounds** is a common programming error that results in a **runtime 'IndexError'**.
- To avoid this error, ensure that you do not use an index beyond `len(my_list) - 1`.
- Here is an example of an **out-of-bounds error**:

Filename: list_out_of_bounds_error.py

```
def main():  
    my_list = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]  
    i = 0  
    while i <= len(my_list):  
        print(my_list[i])  
        i += 1  
  
if __name__ == "__main__":  
    main()
```



How to fix it?

Operations for Sequences

- The following shows some common operations for sequences such as lists, ranges, tuples (to be discussed), and strings (to be covered in the labs).

Operation	Description
<code>x in s</code>	True if element <code>x</code> is in sequence <code>s</code> .
<code>x not in s</code>	True if element <code>x</code> is not in sequence <code>s</code> .
<code>s1 + s2</code>	Concatenates two sequences <code>s1</code> and <code>s2</code> .
<code>s * n, n * s</code>	<code>n</code> copies of sequence <code>s</code> concatenated.
<code>s[i]</code>	<code>i</code> th element in sequence <code>s</code> .
<code>s[i:j]</code>	Slice of sequence <code>s</code> from index <code>i</code> to <code>j - 1</code> .
<code>len(s)</code>	Length of sequence <code>s</code> , i.e., the number of elements in <code>s</code> .
<code>min(s)</code>	Smallest element in sequence <code>s</code> .
<code>max(s)</code>	Largest element in sequence <code>s</code> .
<code>sum(s)</code>	Sum of all numbers in sequence <code>s</code> .
for loop	Traverses elements from left to right in a for loop.
<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>	Compares two sequences

Comparison Operators for Lists

- Equality (`==`):

- Checks if two lists have the **same elements in the same order**.

```
[1, 2, 3] == [1, 2, 3]    # True  
[1, 2, 3] == [3, 2, 1]    # False
```

- Inequality (`!=`):

- Checks if two lists **are not equal**.

```
[1, 2] != [1, 2, 3]      # True
```

- Less Than (`<`) and Greater Than (`>`):

- Compares lists lexicographically (like dictionary order).
- Compares element by element until a difference is found.**

```
[1, 2, 3] < [1, 2, 4]     # True  
[1, 2] < [1, 2, 0]       # True
```

- Less Than or Equal To (`<=`) and Greater Than or Equal To (`>=`):

- Similar to `<` and `>`, but include equality.

```
[1, 2, 3] <= [1, 2, 3]    # True  
[1, 2] >= [1, 2, 0]      # False
```

Functions for Lists

```
def main(): # Filename: functions_for_lists.py
    my_list1 = [1, 2, 3, 4, 5]
    my_list2 = [4, 5, 6, 7, 8]

    print("4 in my_list1:", 4 in my_list1)
    print("4 not in my_list1:", 4 not in my_list1)
    print("my_list1 + my_list2:\n", my_list1 + my_list2)
    print("my_list1 * 2:", my_list1 * 2)
    print("2 * my_list1:", 2 * my_list1)
    print("my_list1[3]:", my_list1[3])
    print("my_list1[3:5]:", my_list1[3:5])
    print("len(my_list1):", len(my_list1))
    print("min(my_list1):", min(my_list1))
    print("max(my_list1):", max(my_list1))
    print("sum(my_list1):", sum(my_list1))

    for i in my_list1:
        print(i, end=" ")
    print()

    print("my_list1 < my_list2:", my_list1 < my_list2)

if __name__ == "__main__":
    main()
```

Output:

```
4 in my_list1: True
4 not in my_list1: False
my_list1 + my_list2:
    [1, 2, 3, 4, 5, 4, 5, 6, 7, 8]
my_list1 * 2: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
2 * my_list1: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
my_list1[3]: 4
my_list1[3:5]: [4, 5]
len(my_list1): 5
min(my_list1): 1
max(my_list1): 5
sum(my_list1): 15
1 2 3 4 5
my_list1 < my_list2: True
```

Index Operator []

- An **element in a list** can be **accessed through the index operator**, using the following syntax:

```
my_list[index]
```

- List indexes are **0-based**, meaning they range from 0 to `len(my_list) - 1`.
- `my_list[index]` can be used just like a variable, so it is also known as an **indexed variable**.
- For example, the following code adds the values in `my_list[0]` and `my_list[1]` to `my_list[2]`:

```
my_list[2] = my_list[0] + my_list[1]
```
- The following loop assigns 0 to `my_list[0]`, 1 to `my_list[1]`, ..., 9 to `my_list[9]`:

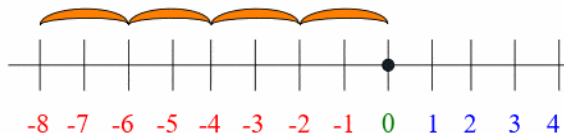
```
for i in range(len(my_list)):
    my_list[i] = i
```

Negative Numbers as Indexes

- Python allows the use of **negative numbers as indexes** to reference positions relative to the end of the list.
- The **actual position** is **obtained by adding the length of the list** to the negative index.
- For example:

```
my_list = [2, 3, 5, 2, 33, 21]
print(my_list[-1])  # Print 21
print(my_list[-3])  # Print 2
```

- $\text{my_list}[-1]$ is the same as $\text{my_list}[-1 + \text{len}(\text{my_list})]$ (i.e., $\text{my_list}[-1 + 6]$).
- $\text{my_list}[-3]$ is the same as $\text{my_list}[-3 + \text{len}(\text{my_list})]$ (i.e., $\text{my_list}[-3 + 6]$).



List Slicing

- The **slicing operator** returns a **slice of the list** using the syntax:

`my_list[start : end : step]`

- **start, end, step must be integers** (they cannot be floats, even if they represent whole numbers).
 - **step cannot be 0.**
- The **slice** is a **sublist** from index `start` to index `end - 1` with the specified step.
 - By default, step is 1.

```
my_list = [2, 3, 5, 7, 9, 1]
print(my_list[2 : 4])      # Print [5, 7]
print(my_list[0 : 5 : 2])  # Print [2, 5, 9]
```



List Slicing

- You can use a **negative index in slicing**.

```
my_list = [2, 3, 5, 7, 9, 1]
```

```
print(my_list[1 : -3])    # Print [3, 5]
```

```
print(my_list[-4 : -2])   # Print [5, 7]
```

- $\text{my_list}[1 : -3]$ is the same as $\text{my_list}[1 : -3 + \text{len}(\text{my_list})]$.

- $\text{my_list}[-4 : -2]$ is the same as

$\text{my_list}[-4 + \text{len}(\text{my_list}) : -2 + \text{len}(\text{my_list})]$.

- You can **assign values to a slice** of a list.

```
my_list = [2, 3, 5, 7, 9, 1]
```

```
my_list[1 : 3] = [91, 92, 93, 94]
```

```
print(my_list) # Print [2, 91, 92, 93, 94, 7, 9, 1]
```

- $\text{my_list}[1 : 3] = [91, 92, 93, 94]$ replaces $[3, 5]$ in my_list with $[91, 92, 93, 94]$.



List Slicing: Default Values and Edge Cases

- The **starting index or ending index may be omitted**. Then, **default values will be used**.
 - Positive step (i.e., $\text{step} > 0$):
 - If you omit the start index: Default is 0 (start from the beginning).
 - If you omit the end index: Default is the length of the list (i.e., `len(my_list)`).
 - If start index $\geq$ end index, the result will be an empty list.
 - Negative step (i.e., $\text{step} < 0$):
 - If you omit the start index: Default is the last index (i.e., `len(my_list)-1`).
 - If you omit the end index: Default is None (will go until the start of the list).
 - If end index $\geq$ start index, the result will be an empty list.
 - If you omit the step: Default is 1.
- If start or end specifies a **position beyond the end** of the list, Python will **use the length** of the list for start or end instead.



List Slicing Examples

- Positive steps (i.e., step > 0):

```
my_list = [2, 3, 5, 7, 9, 1]
print(my_list[ : 2 : 1]) # Equivalent to print(my_list[0 : 2 : 1]), Print [2, 3]
print(my_list[3 :    : 1]) # Equivalent to print(my_list[3 : 6 : 1]), Print [7, 9, 1]
print(my_list[3 : 1 : 1]) # Empty list
```

- Negative steps (i.e., step < 0):

```
my_list = [2, 3, 5, 7, 9, 1]
print(my_list[ : 2 : -1]) # Equivalent to print(my_list[5 : 2 : -1]), Print [1, 9, 7]
print(my_list[3 :    : -1]) # Equivalent to print(my_list[3 : None : -1]), Print [7, 5, 3, 2]
print(my_list[1 : 3 : -1]) # Empty list
```

- start or end specifies a position beyond the end of the list:

```
my_list = [2, 3, 5, 7, 9, 1]
print(my_list[3 : 8]) # Equivalent to print(my_list[3 : 6]), Print [7, 9, 1]
print(my_list[7 : 5]) # Equivalent to print(my_list[6 : 5]), Print []
print(my_list[7 : 8]) # Equivalent to print(my_list[6 : 6]), Print []
```

Slicing handles out-of-range indices gracefully!

Traversing Elements in a List

- Python lists are **iterable**.
- Python supports a **convenient for loop**, which enables you to **traverse the list sequentially** without using an index variable.

- For example, the following code displays all the elements in the list `my_list`:

```
my_list = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]
for u in my_list:
    print(u, end=' ')
# Print 5.6 4.5 3.3 13.2 4.0 34.33 34.0 45.45 99.993 11123
```

- You still have to use **an index variable** if you wish to **traverse the list in a different order or change the elements in the list**. For example, the following code displays the elements at even-numbered indices:

```
my_list = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]
for i in range(0, len(my_list), 2):
    print(my_list[i], end=' ')
# Print 5.6 3.3 4.0 34.0 99.993
```

Traversing Elements in a List

- To iterate over a list, we can use `enumerate()` function, which **adds a counter** to an iterable (such as list) and returns an enumerate object which can be used in loops to **access both the index and the value of each item**. This simplifies iteration when we need to keep track of the position of items without managing a manual counter variable.
- Syntax:

```
enumerate(<iterable>, start = 0)
```

where `iterable` is any object that supports iteration, such as list, tuple or string, and `start` is an optional index value from which the counter should begin. By default, it starts at 0.

```
my_list = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]
for index, item in enumerate(my_list):
    print(f"Index {index}: {item}", end=' ')
# Print Index 0: 5.6, Index 1: 4.5, Index 2: 3.3, Index 3: 13.2, Index 4: 4.0,
#      Index 5: 34.33, Index 6: 34.0, Index 7: 45.45, Index 8: 99.993, Index 9: 11123,
```

List Comprehensions

- **List comprehensions** provide a **concise syntax to create a list** by processing another sequence of data.
- A list comprehension **consists of brackets containing an expression followed by a for clause**, then zero or more for or if clauses.
- The list comprehension produces a list with the results from evaluating the expression.

```
list1 = [x for x in range(5)]  
print(list1)  # Print [0, 1, 2, 3, 4]
```

```
list2 = [0.5 * x for x in list1]  
print(list2)  # Print [0.0, 0.5, 1.0, 1.5, 2.0]
```

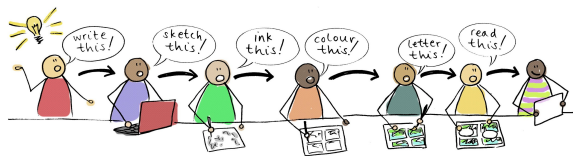
```
list3 = [x for x in list2 if x < 1.5]  
print(list3)  # Print [0.0, 0.5, 1.0]
```



List Methods

- Lists are defined using the **list class** in Python. Once a list is created, you can **use the list class's methods** to manipulate the list.

Method	Description
<code>append(x)</code>	Adds an element <code>x</code> to the end of the list.
<code>count(x)</code>	Returns the number of times element <code>x</code> appears in the list.
<code>extend(anotherList)</code>	Appends all the elements in <code>anotherList</code> to the list.
<code>index(x)</code>	Returns the index of the first occurrence of element <code>x</code> in the list.
<code>insert(index, x)</code>	Inserts an element <code>x</code> at a given index. Note that the first element in the list has index 0.
<code>pop(index)</code>	Removes the element at the given index and returns it. The parameter <code>index</code> is optional. If it is not specified, <code>list.pop()</code> removes and returns the last element in the list.
<code>remove(x)</code>	Removes the first occurrence of element <code>x</code> from the list.
<code>reverse()</code>	Reverses the elements in the list.
<code>sort()</code>	Sorts the elements in the list in ascending order.



Examples

```
list1 = [2, 3, 4, 1, 32, 4]
list1.append(19)
print(list1)           # Print [2, 3, 4, 1, 32, 4, 19]
print(list1.count(4))  # Return the count for number 4, print 2
list2 = [99, 54]
list1.extend(list2)
print(list1)           # Print [2, 3, 4, 1, 32, 4, 19, 99, 54]
print(list1.index(4))  # Print 2
list1.insert(1, 25)
print(list1)           # Print [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
print(list1.pop(2))    # Print 3
print(list1)           # Print [2, 25, 4, 1, 32, 4, 19, 99, 54]
print(list1.pop())     # Print 54
print(list1)           # Print [2, 25, 4, 1, 32, 4, 19, 99]
list1.remove(32)       # Equivalent to del list1[4]
print(list1)           # Print [2, 25, 4, 1, 4, 19, 99]
list1.reverse()
print(list1)           # Print [99, 19, 4, 1, 4, 25, 2]
list1.sort()
print(list1)           # Print [1, 2, 4, 4, 19, 25, 99]
list1.sort(reverse=True) # Sort the list in descending order
print(list1)           # Print [99, 25, 19, 4, 4, 2, 1]
del list1              # Delete the whole list, so list1 no longer exists
```



Splitting a String into a List

- To **split the characters in a string** `s` into a list, use `list(s)`:

```
my_list = list("abc")  
print(my_list)  # Print ['a', 'b', 'c']
```

- The `str` class contains the **split method**, which is useful for **splitting items in a string into a list**:

```
items1 = "COMP1023 is the best COMP course".split() # Delimited by spaces  
print(items1)  # Print ['COMP1023', 'is', 'the', 'best', 'COMP', 'course']
```

```
items2 = "12/25/2026".split("/") # Delimited by /  
print(items2)  # Print ['12', '25', '2026']
```



Inputting Lists

- To read data from the console into a list, you can enter one data item per line and append it to a list in a loop.

Filename: inputting_list.py

```
def main():
    total = 0.0
    counter = 0
    my_list = [] # Create an empty list
    print("Enter 10 numbers, one number per line: ")
    for i in range(10):
        my_list.append(float(input()))
        total += my_list[-1]
        counter += 1
    average = total / counter
    print("Average =", average)

if __name__ == "__main__":
    main()
```

Output:

```
Enter 10 numbers, one number per line:
52
79
67
21
56
11
99
41
69
47
Average = 54.2
```

Inputting Lists

- You can **enter the data in one line**, separated by spaces.
- The following code demonstrates how to read the input and calculate the average:

```
# Filename: inputting_list_comprehension.py
def main():
    s = input("Enter 10 numbers separated by spaces: ")
    items = s.split()           # Extract items from the string
    my_list = [float(x) for x in items] # Convert items to numbers
    average = sum(my_list) / len(my_list) # Calculate average
    print("Average =", average)

if __name__ == "__main__":
    main()
```

Output:

```
Enter 10 numbers separated by spaces: 52 79 67 21 56 11 99 41 69 47
Average = 54.2
```



Good Use of List

- Suppose you want to prompt the user to enter a month number and display its month name.
- A simple approach might look like this:

```
if monthNumber == 1:
    print("The month is January")
elif monthNumber == 2:
    print("The month is February")
...
else:
    print("The month is December")
```

- A “better” way to do this is as follows:

```
months = ["January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November", "December"]
monthNumber = int(input("Enter a month number (1 to 12): "))
if 1 <= monthNumber <= 12:
    print("The month is", months[monthNumber - 1])
else:
    print("Invalid month number! Please enter a number between 1 and 12.")
```

Copying Lists

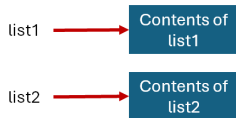
Question

Does `list2 = list1` duplicate a list?

- The statement `list2 = list1` does not copy the contents of the list referenced by `list1` to `list2`.
- Instead, it copies the reference from `list1` to `list2`.
- After this statement, `list1` and `list2` refer to the same list.

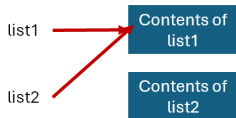
Before the assignment:

`list2 = list1`



After the assignment:

`list2 = list1`



```
list1 = [1, 2]
list2 = [3, 4, 5]
print(id(list1)) # Print 140415885168576
print(id(list2)) # Print 140415896311744
list2 = list1
print(id(list2)) # Print 140415885168576
```

- The list previously referenced by `list2` is no longer referenced. The memory space occupied by that list will be automatically collected and reused by the Python interpreter.

How can you duplicate a list?

Deep Copy vs. Shallow Copy in Python

- Shallow Copy

- Creates a **new object**, but inserts references to the objects found in the original.
- Changes to **mutable objects** in the copied list **affect** the original list.

- Deep Copy

- Creates a **new object** and recursively adds copies of **nested objects** found in the original.
- Changes to **mutable objects** in the copied list **do not affect** the original list.

Filename: shallow_vs_deep_copy.py

```
import copy
```

```
# Original list with a nested mutable object  
original_list = [[1, 2], [3, 4]]
```

```
# Shallow Copy
```

```
shallow_copied_list = original_list.copy()
```

```
# Modifies original_list as well
```

```
shallow_copied_list[0][0] = 99
```

```
# Deep Copy
```

```
deep_copied_list = copy.deepcopy(original_list)
```

```
# Does not affect original_list
```

```
deep_copied_list[1][0] = 88
```

```
print(original_list)           # Print [[99, 2], [3, 4]]
```

```
print(shallow_copied_list)     # Print [[99, 2], [3, 4]]
```

```
print(deep_copied_list)        # Print [[99, 2], [88, 4]]
```

Copying Lists

- Shallow Copy

- List Constructor

```
list1 = [1, 2]  
list2 = list(list1)
```

- List Comprehension

```
list1 = [1, 2]  
list2 = [x for x in list1]
```

- List Concatenation

```
list1 = [1, 2]  
list2 = [] + list1
```

- List Splicing

```
list1 = [1, 2]  
list2 = list1[:]
```

- Deep Copy

- `copy.deepcopy()`

```
import copy  
list1 = [1, 2]  
list2 = copy.deepcopy(list1)
```



Two-Dimensional Lists

- A two-dimensional list consists of rows.
- Each row is a list that contains values.
- The rows can be accessed using an index, called a row index.
- The values in each row can be accessed through another index, called a column index.

```
matrix = [  
    [1, 2, 3, 4, 5],  
    [6, 7, 0, 0, 0],  
    [0, 1, 0, 0, 0],  
    [1, 0, 0, 0, 8],  
    [0, 0, 9, 0, 3]  
]
```

	[0]	[1]	[2]	[3]	[4]
[0]	1	2	3	4	5
[1]	6	7	0	0	0
[2]	0	1	0	0	0
[3]	1	0	0	0	8
[4]	0	0	9	0	3

```
matrix[0] is [1, 2, 3, 4, 5]  
matrix[1] is [6, 7, 0, 0, 0]  
matrix[2] is [0, 1, 0, 0, 0]  
matrix[3] is [1, 0, 0, 0, 8]  
matrix[4] is [0, 0, 9, 0, 3]  
  
matrix[0][0] is 1  
matrix[4][4] is 3
```

- Each value can be accessed using `matrix[i][j]`, where *i* and *j* are the row and column indexes.

Summing All Elements and Elements by Column

Filename: summing_elements.py

```
def main():
    matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    # Summing All Elements
    total = 0
    for row in matrix:
        for value in row:
            total += value
    print("Total is", total)  # Print the total

    # Summing Elements by Column
    for column in range(len(matrix[0])):
        total = 0
        for row in range(len(matrix)):
            total += matrix[row][column]
        print("Sum for column", column, "is", total)

if __name__ == "__main__":
    main()
```

Output:

Total is 45

Sum for column 0 is 12

Sum for column 1 is 15

Sum for column 2 is 18



Printing Two-Dimensional Lists

```
def main(): # Filename: printing_two_dimensional_lists.py
    matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    # Method 1: Using indices
    for row in range(len(matrix)):
        for column in range(len(matrix[row])):
            print(matrix[row][column], end=" ")
        print() # Print a new line

    print() # Print another new line

    # Method 2: Using direct iteration
    for row in matrix:
        for value in row:
            print(value, end=" ")
        print() # Print a new line

if __name__ == "__main__":
    main()
```

Output:

```
1 2 3
4 5 6
7 8 9
```

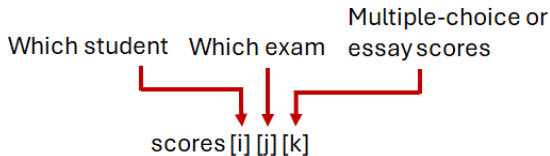
```
1 2 3
4 5 6
7 8 9
```



Multidimensional Lists

- Occasionally, you need to represent **n-dimensional data**, for any integer n .
- For example, you can use a **three-dimensional list** to store **exam scores** for a class of **6 students with 5 exams**, where each exam has **2 parts** (multiple-choice and essay).

```
scores = [  
    [[11.5, 20.5], [11.0, 22.5], [15, 33.5], [13, 21.5], [15, 2.5]],  
    [ [4.5, 21.5], [11.0, 22.5], [15, 34.5], [12, 20.5], [14, 11.5]],  
    [ [6.5, 30.5], [11.4, 11.5], [11, 33.5], [11, 23.5], [10, 2.5]],  
    [ [6.5, 23.5], [11.4, 32.5], [13, 34.5], [11, 20.5], [16, 11.5]],  
    [ [8.5, 26.5], [11.4, 52.5], [13, 36.5], [13, 24.5], [16, 2.5]],  
    [[11.5, 20.5], [11.4, 42.5], [13, 31.5], [12, 20.5], [16, 6.5]]  
]
```



The expression `scores[0][1][0]` refers to the multiple-choice score for the first student's second exam.

Key Terms

- List
- List comprehension

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. You can use the Python built-in functions _____, _____, _____, and _____ to return the length of a numeric list, the maximum and minimum elements in a numeric list, and the sum of all the elements in a numeric list.
2. You can use the index operator `[]` with integer values to reference an individual element in a _____, and a _____.
3. Programmers often mistakenly reference the first element in a list or tuple with index 1, but it should be _____.
4. You can use the concatenation operator `+` to concatenate two _____, the repetition operator `*` to duplicate elements, the slicing operator `[:]` to get a _____, and the `in` and `not in` operators to check whether an element is in a _____.

Answer: 1. `len`; `max`; `min`; `sum`, 2. list; tuple, 3. 0, 4. lists; sublist; sequence.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

5. A _____ object is mutable. You can use the methods _____, _____, _____, _____ to add and remove elements to and from a _____.
6. You can use the _____ method to split a string into a list.

Answer: 5. list; append; insert; pop; remove; list, 7. split.

Further Reading

- Read Chapters 7 & 14 of the textbook “Introduction to Python Programming and Data Structures”.



That's all!

Any question?

**I CAN'T
STOP
THINKING!!**

