Revised: 2/17/2018

Lists

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- List Basics
  - Retrieving and Changing Elements
  - Iterating Over a List
  - Adding Elements to Lists
  - Deleting Elements from Lists
- List Functions
- List Slicing
- Copying Lists
- Testing the Equality of Lists
- Two Dimensional Lists
- Tuples

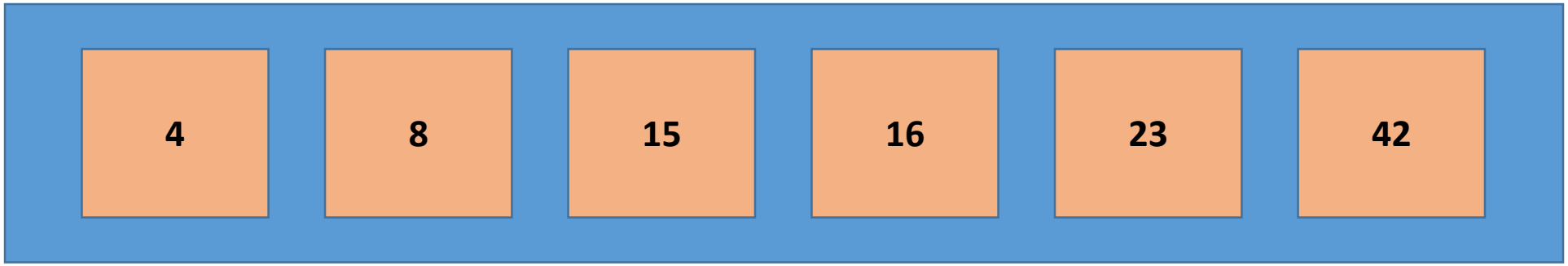
Colors/Fonts

- Global Variable Names – Brown
- Local Variable Names – Lt Blue
- Literals – Blue
- Keywords – Orange
- Operators/Punctuation – Black
- Functions – Purple
- Parameters – Gold
- Comments – Gray
- Modules – Pink
- Object/Class Names – Green

Source Code – **Consolas**
Output – Courier New

What is a list?

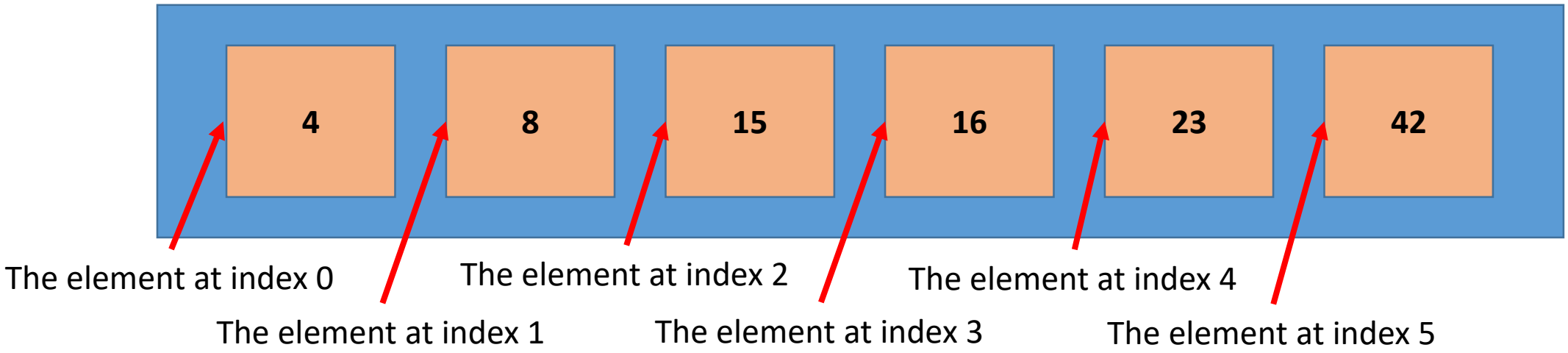
- An ***list*** is a sequence object that has multiple values.
  - Another way to look at it is a variable that has multiple values.



A list of ints

List Terminology

- An ***index*** (or ***subscript***) is the number representing the position of a list element.
  - First index is always zero.
  - The index is always an int.
- An ***element*** is the data or object referenced by an index.



Creating a List

- The elements are comma separated, enclosed in square brackets.

```
numbers = [4, 8, 15, 16, 23, 42]
values = [35.6, 32.76, 51.4]
pets = ["dog", "cat", "bird", "fish"]
```

- The data types of a list may vary.

```
mixed_values = [35.6, 15, "cat"]
```

- An empty list:

```
example = []
```

Printing a List

- When passed to the print function, the entire list is printed.
  - Includes commas and brackets.
  - Useful for testing/debugging.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42]
```

Getting the Length of a List

- A list's *length* is the total number of elements contained within it.
  - Python's built-in len function returns the length of a sequence data type.

```
pets = ["dog", "cat", "bird", "fish"]
length = len(pets)
print(length)
```

```
4
```

Retrieving an Element from a List

- Elements of a list are referenced using *subscript notation*.
  - Specify the index of the list's element.

```
numbers = [4, 8, 15, 16, 23, 42]
test_value1 = numbers[0]
print(test_value1)

test_value2 = numbers[4]
print(test_value2)

print(numbers[2])
```

```
4
23
15
```

Changing an Element from a List

- Lists are *mutable*, meaning the elements can be changed.
  - Specify the index of the list's element and assign to it a new value.

```
values = [35.6, 32.76, 51.4]
print(values[1])
values[1] = 27.21
print(values[1])
```

```
32.76
27.21
```

Relative Indexes

- Negative indexes retrieve elements relative to the end of the list.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers[-1])
numbers[-4] = 100
print(numbers[-4])
print(numbers[2])
```

Length – 1 = 6 – 1 = **5**

Length – 4 = 6 – 4 = **2**

```
42
100
100
```

IndexError Exception

- An IndexError exception will be raised if you try to access an index that does not exist.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers[10])
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    print(numbers[10])
IndexError: list index out of range
>>>
```

IndexError Exception

- IndexError exceptions can be handled using a try…except statement.

```
numbers = [4, 8, 15, 16, 23, 42]
try :
  print(numbers[10])
except IndexError :
  print("Value not found")
```

```
Value not found
```

Iterating Over a List

- For loops can iterate over the values of a list.

```
numbers = [4, 8, 15, 16, 23, 42]
for number in numbers :
  print(number)

print()

pets = ["dog", "cat", "bird", "fish"]
for animal in pets :
  print(animal)
```

```
4
8
15
16
23
42

dog
cat
bird
fish
```

Iterating Over a List

- For loops (using the range function) can iterate over the entire list or a segment of the list.

```
numbers = [4, 8, 15, 16, 23, 42]
for i in range(0, 3) :
  print(numbers[i])

print()

pets = ["dog", "cat", "bird", "fish"]
for i in range(1, len(pets)) :
  print(pets[i])
```

```
4
8
15

cat
bird
fish
```

Determining if an Element Exists in a List

- An if statement can be used to find if an element/value is present in a list.
  - Utilizes the **in** keyword.
  - Does not tell us where (what index) the element was found.

```
numbers = [4, 8, 15, 16, 23, 42]
value = 7
if value in numbers :
  print("Value exists in list")
else :
  print("Value does not exist in list")
```

```
Value does not exist in list
```

Index Function

- A list's index function returns the index of an element/value.
  - Returns the index of the first matching element/value.
  - Case sensitive.

```
numbers = [4, 8, 8, 15, 16, 23, 42]
found_index = numbers.index(8)
print(found_index)
```

```
1
```

Index Function

- If the list's index function does not find a matching element/value a ValueError exception will be raised.

```
pets = ["dog", "cat", "bird"]
found_index = pets.index("fish")
print(found_index)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 9, in <module>
    foundIndex = pets.index("fish")
ValueError: 'fish' is not in list
>>>
```

Index Function

```
pets = ["dog", "cat", "bird"]
try :
  found_index = pets.index("fish")
  print(found_index)
except ValueError :
  print("Value not found")
```

```
Value not found
```

Adding to Lists

- Values can be added/concatenated to a list using the addition operator ONLY if the values are in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + 100
print(numbers)
```

Error

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100]
```

Adding to Lists

- Two lists are merged/concatenated together when combined using the addition operator.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100, 101, 102]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100, 101, 102]
```

Adding to Lists

- Another way to add values to a list is with the addition combined assignment operator.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers += [100, 101, 102]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100, 101, 102]
```

Append Function

- A list's append function can add a single element to the end of a list.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers.append(100)
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100]
```

Append Function vs Addition Operator

- A list's append function can only add a single element to the end of a list.
  - Does not have to be in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers.append(100)
print(numbers)
```

- Concatenating data to the end of a list using the addition/combined assignment operator can be used to add one or multiple elements.
  - Must be in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100]
print(numbers)
```

```
numbers = [4, 8, 15, 16, 23, 42]
numbers += [100, 101, 102]
print(numbers)
```

Deleting Elements from Lists

- To remove an element by index, use the **del** (delete) keyword to remove it.
  - Any subsequent elements will be shifted over.

```
numbers = [4, 8, 15, 16, 23, 42]
del numbers[3]
print(numbers)
```

```
[4, 8, 15, 23, 42]
```

Deleting Elements from Lists

- To remove an element by value, the list's remove function will delete the element.
  - Only removes the first match.
  - Case-sensitive.

```
pets = ["dog", "cat", "bird", "cat", "fish"]
pets.remove("cat")
print(pets)
```

```
["dog", "bird", "cat", "fish"]
```

Deleting Elements from Lists

- If the element is not found, a ValueError exception will be raised.

```
pets = ["dog", "cat", "bird", "cat", "fish"]
pets.remove("CAT")
print(pets)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 9, in <module>
    pets.remove("CAT")
ValueError: list.remove(x): x not in list
>>>
```

Deleting Elements from Lists

```
pets = ["dog", "cat", "bird", "cat", "fish"]
try :
  pets.remove("CAT")
  print(pets)
except ValueError :
  print("Value not found")
```

```
Value not found
```

Remove Function vs del Keyword

- The value to delete must be known in order to use the list's remove function.
  - May raise a ValueError exception.

- When deleting using the del keyword, only the index must be known.
  - May raise an IndexError exception if the index does not exist.

Insert Function

- A list's insert function places a value at a specified index.
  - The existing elements are shifted over to make room.
  - First argument is the index.
    - If the specified index is beyond the length of the list, the value will be inserted at the end of the list.
  - Second argument is the value to insert.

```
numbers = [10, 20, 40, 50]
numbers.insert(2, 30)
print(numbers)
```

```
[10, 20, 30, 40, 50]
```

Sort Function

- A list's sort function rearranges a list so the values are in ascending order.

```
numbers = [30, 40, 20, 10]
numbers.sort()
print(numbers)
```

```
[10, 20, 30, 40]
```

  - Strings are sorted in ascending, lexicographical order

```
pets = ["dog", "cat", "bird", "Cat", "fish"]
pets.sort()
print(pets)
```

```
['Cat', 'bird', 'cat', 'dog', 'fish']
```

Reverse Function

- A list's reverse function rearranges a list so the values are in reverse order.

```
numbers = [20, 40, 10, 30]
numbers.reverse()
print(numbers)
```

```
[30, 10, 40, 20]
```

Reverse Function

- We can use the sort and reverse functions together to sort a list in descending order.

```
numbers = [20, 40, 10, 30]
numbers.sort()
print(numbers)                  [10, 20, 30, 40]
numbers.reverse()
print(numbers)                  [40, 30, 20, 10]
```

Min Function

- Python's built-in min function returns the smallest value from a list.

```
numbers = [20, 40, 10, 30]
min_value = min(numbers)
print(min_value)
```

```
10
```

  - For strings, the smallest value would be the one that is lexicographically first.

```
pets = ["dog", "cat", "bird", "Cat", "fish"]
min_value = min(pets)
print(min_value)
```

```
Cat
```

Max Function

- Python's built-in max function returns the largest value from a list.

```
numbers = [20, 40, 10, 30]
max_value = max(numbers)
print(max_value)
```

```
40
```

  - For strings, the largest value would be the one that is lexicographically last.

```
pets = ["dog", "cat", "bird", "Cat", "fish"]
max_value = max(pets)
print(max_value)
```

```
fish
```

List Slicing

- *Slicing* selects a range of elements from a sequence type.
- The general syntax for slicing a list is:

```
list[startIndex:endIndex]
```

- This will return a list containing the values between those indexes.
  - The start index is inclusive.
  - The end index is exclusive.

List Slicing

```
numbers = [4, 8, 15, 16, 23, 42]
slice = numbers[1:4]
print(slice)
```

```
[8, 15, 16]
```

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[0:2]
print(slice)
```

```
['dog', 'cat']
```

List Slicing

- Specifying only start index will return a slice beginning with the start index's element through the end of the list.

```
numbers = [4, 8, 15, 16, 23, 42]
slice = numbers[2:]
print(slice)
```

```
[15, 16, 23, 42]
```

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[1:]
print(slice)
```

```
['cat', 'bird', 'fish']
```

List Slicing

- Specifying only an ending index will return a slice beginning with the start of the list up to, but not including, the ending index.

```
numbers = [4, 8, 15, 16, 23, 42]
slice = numbers[:3]
print(slice)
```

```
[4, 8, 15]
```

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[:2]
print(slice)
```

```
['dog', 'cat']
```

List Slicing

- Slicing is safe from IndexError exceptions.

- If the starting index is greater than the ending index, an empty list will be returned.

```
numbers = [4, 8, 15, 16, 23, 42]
slice = numbers[3:1]
print(slice)
```

```
[]
```

List Slicing

- If the ending index is beyond the length of the list, Python will use the length as the ending index.

```
numbers = [4, 8, 15, 16, 23, 42]
slice = numbers[2:100]
print(slice)
```

```
[15, 16, 23, 42]
```

List Slicing

- If the starting index is negative, Python will use 0 as the starting index.

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[-5:2]
print(slice)
```

```
['dog', 'cat']
```

- This is not the case if there is no ending index or the ending index is negative.

List Slicing

- When only a negative starting index is specified, the slice will begin relative to the end of the list.

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[-3:]
print(slice)
```

```
['cat', 'bird', 'fish']
```

List Slicing

- When both starting and ending indexes are negative, the slice will begin and end relative to the end of the list.

```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[-3:-2]
print(slice)
```

```
['cat']
```

List Slicing

- If a negative starting index is greater (closer to zero) than the negative ending index, an empty list will be returned.

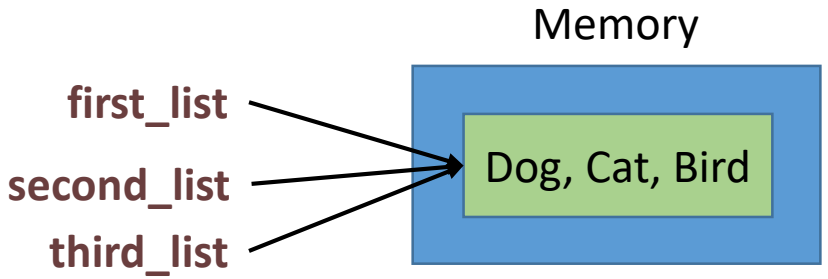
```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[-2:-3]
print(slice)
```

```
[]
```

Copying Lists

- Copying a list like the example below creates a ***shallow copy.***
  - Shallow copies are multiple variables referencing the same data.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = first_list
third_list = first_list
```



Shallow Copies

- Since the variables reference the same list, changing one appears to change any others.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = first_list
print(first_list[0])                Dog

second_list[0] = "Fish"
print(first_list[0])                Fish
```

Copying Arrays

- To create a second, separate list with the same contents you need to perform a ***deep copy***.
  - A deep copy copies the contents of one list into a second list.

```
original = [3, 5, 7, 9]
copy = []
```

Empty List

```
for element in original :
  copy.append(element)
```

Deep Copies

- Since the variables reference different lists, changing one does not alter the original.

```
original = [3, 5, 7, 9]
copy = []

for element in original :
  copy.append(element)

print(original[0])                3
copy[0] = 99
print(original[0])                3
```

Deep Copies

- An alternative, simpler way to deep copy a list.
  - Concatenate the original list with an empty list.

```
original = [3, 5, 7, 9]
copy = [] + original

for element in copy :
  print(element)
```

```
3
5
7
9
```

Testing Equality of Lists

- The equality operator (==) compares lists to determine if they are equal.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["Dog", "Cat", "Bird"]

if first_list == second_list :
  print("The lists are equal")
else :
  print("The lists are not equal")
```

```
The lists are equal
```

Testing Equality of Lists

- Order of the elements matter when determining equality.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["Cat", "Bird", "Dog"]

if first_list == second_list :
  print("The lists are equal")
else :
  print("The lists are not equal")
```

```
The lists are not equal
```

Testing Equality of Lists

- Elements must be exact matches.
  - Not the case for numeric types. 10 and 10.0 would be a match.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["DOG", "CAT", "BIRD"]

if first_list == second_list :
  print("The lists are equal")
else :
  print("The lists are not equal")
```

```
The lists are not equal
```

Testing Equality of Lists

- Must be the same length.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["Dog", "Cat"]

if first_list == second_list :
  print("The lists are equal")
else :
  print("The lists are not equal")
```

```
The lists are not equal
```

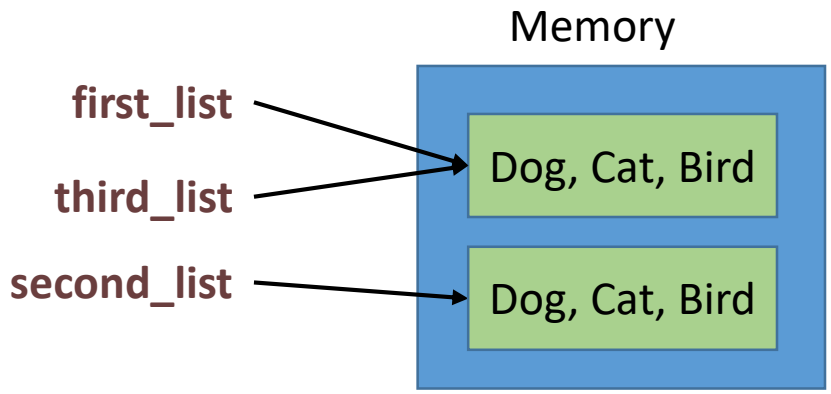
Testing Equality of Arrays

- The **is** keyword tests if two variables reference the same object.
  - In other words, the is keyword will determine if two list variables are shallow copies.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["Dog", "Cat", "Bird"]
third_list = first_list #Shallow Copy

if first_list is third_list :
  print("These lists are shallow copies")

if first_list is second_list :
  print("These lists are shallow copies")
```



Multidimensional Lists

- When an list contains lists, it is called ***multidimensional***.
  - A one dimensional list:

```
my_1d_list = [2, 4, 6]
```

  - A two dimensional list:

```
my_2d_list = [[8, 3, 7], [1, 9, 9], [5, 6, 9]]
```

Multidimensional Lists

- It's often better to write two dimensional lists like this:

```
my_2d_list = [[8, 3, 7],
              [1, 9, 9],
              [5, 6, 9]]
```

- This way, it's easier to see each row (first dimension) and column (second dimension).

Multidimensional Lists

- Elements in a two dimensional list are referenced by row and column.
  - Row and column numbers start at zero.

```
my_2d_list = [[8, 3, 7],
              [1, 9, 9],
              [5, 6, 9]]

my_2d_list[1][2] = 2 #Assignment
print(my_2d_list[0][1]) #Retrieval
```

3

Multi-dimensional Lists

```
my_2d_list = [[2, 4, 6],
              [1, 3, 5],
              [3, 6, 9],
              [1, 2, 3]]
```

What element is at `my_2d_list[0][2]`?

What element is at `my_2d_list[3][1]`?

What element is at `my_2d_list[1][0]`?

Multi-dimensional Lists

- Rows in a multidimensional array do not have to be the same length.
  - This is called a ***Ragged List***.

```
my_2d_list = [[2, 4, 6],
              [1, 3],
              [9],
              [1, 2, 3, 4]]
```

- Be careful with ragged lists as not all rows have the same number of columns.

`my_2d_list[2][1]` does not exist, even though every other row has a column 1.

Multidimensional Lists

- Two for loops are required to iterate through a two dimensional array.

```
my_2d_list = [[8, 3],
              [1, 9]]

for row in range(0, len(my_2d_list)) :
  for col in range(0, len(my_2d_list[row])) :
    print(my_2d_list[row][col])
```

Rows

Columns

Multidimensional Lists

- Iteration without using indexes.

```
my_2d_list = [[8, 3],
              [1, 9]]

for row in my_2d_list :
  for col in row :
    print(col)
```

Rows

Columns

Multidimensional Lists

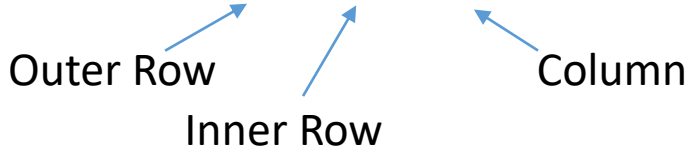
- There is no limit to the number of dimensions an list can have.

- A three dimensional list:

```
my_3d_list = [[[4,8],[15,16,23,42]],[[11,33],[22,44]]]
```

- In the case of a three dimensional array, the rows themselves have rows.

```
my_3d_list[2][2][3] = 100
```



Multidimensional Lists

```
                    0   1    2    3
my_3d_list = [[[4, 8],                0
               [15,16,23,42]],        1
              [[11,33],               0
               [22,44]]]              1
```

0

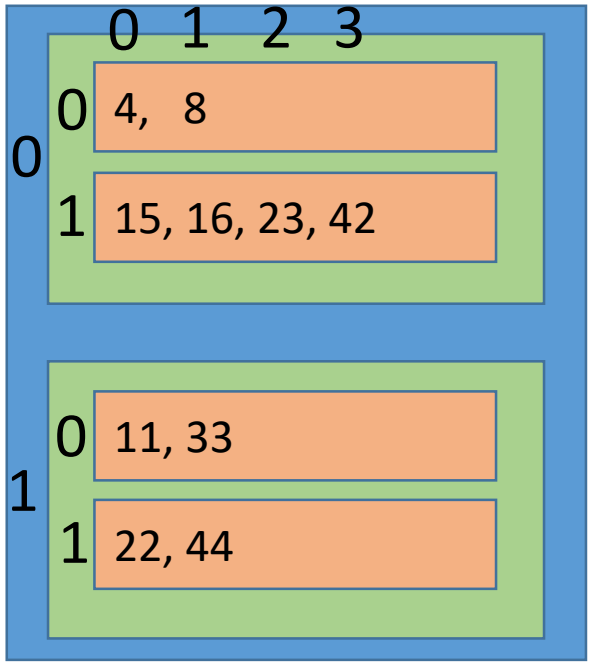
1

0

1

What element is at `my_3d_list[0][1][2]`?

What element is at `my_3d_list[1][0][0]`?



Multidimensional Lists

- Three for loops are required to iterate through a three dimensional list.

```
my_3d_list = [[[4, 8],
               [15,16,23,42]],
              [[11,33],
               [22,44]]]

for outer in range(0, len(my_3d_list)) :
  for inner in range(0, len(my_3d_list[outer])) :
    for col in range(0, len(my_3d_list[outer][inner])) :
      print(my_3d_list[outer][inner][col])
```

Columns

Inner Rows

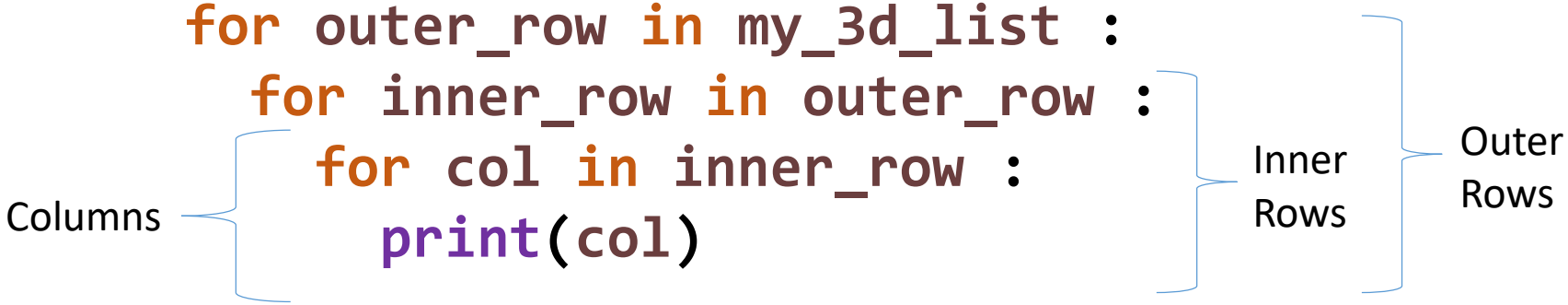
Outer Rows

Multidimensional Lists

- Iteration without using indexes.

```
my_3d_list = [[[4, 8],
               [15,16,23,42]],
              [[11,33],
               [22,44]]]

for outer_row in my_3d_list :
  for inner_row in outer_row :
    for col in inner_row :
      print(col)
```



Tuples

- A ***tuple*** is a sequence type and is very much like a list, however tuples are immutable.
  - The elements in a tuple cannot be changed.

- Tuples contain comma separated values in parentheses.
  - The elements/values can be of different types.

```
numbers = (4, 8, 15, 16, 23, 42)
values = (35.6, 32.76, 51.4)
pets = ("dog", "cat", "bird", "fish")
mixed_values = (35.6, 15, "cat")
```

Tuples

- Tuples that contain only one element must include a trailing comma.
  - Python interpreter treats the parentheses as part of an arithmetic expression:

```
example = (4)  #Creates an int
```

  - Python interpreter treats the parentheses as part of a tuple:

```
example = (4,) #Creates a tuple
```

Getting the Length of a Tuple

- A tuple's length, like a list, is the total number of elements contained within it.
  - Python's built-in len function returns the length of a sequence data type.

```
pets = ("dog", "cat", "bird", "fish")
length = len(pets)
print(length)
```

```
4
```

Retrieving an Element from a Tuple

- Elements of a tuple are referenced using subscript notation.
  - Specify the index of the tuple's element.

```
numbers = (4, 8, 15, 16, 23, 42)
test_value1 = numbers[0]
print(test_value1)

test_value2 = numbers[4]
print(test_value2)

print(numbers[2])
```

```
4
23
15
```

IndexError Exception

- An IndexError exception will be raised if you try to access an index that does not exist.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers[10])
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 14, in <module>
    print(numbers[10])
IndexError: tuple index out of range
>>>
```

IndexError Exception

- IndexError exceptions can be handled using a try…except statement.

```
numbers = (4, 8, 15, 16, 23, 42)
try :
  print(numbers[10])
except IndexError :
  print("Value not found")
```

```
Value not found
```

Relative Indexes

- Negative indexes retrieve elements relative to the end of the tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers[-1])
print(numbers[-4])
print(numbers[2])
```

Length – 1 = 6 – 1 = **5**

Length – 4 = 6 – 4 = **2**

```
42
15
15
```

Printing a Tuple

- When passed to the print function, the entire tuple is printed.
  - Includes commas and parentheses.
  - Useful for testing/debugging.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers)
```

```
(4, 8, 15, 16, 23, 42)
```

Iterating Over a Tuple

- For loops can iterate over the values of a tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
for number in numbers :
  print(number)

print()

pets = ("dog", "cat", "bird", "fish")
for animal in pets :
  print(animal)
```

```
4
8
15
16
23
42

dog
cat
bird
fish
```

Iterating Over a Tuple

- For loops can iterate over the values of a tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
for i in range(0, 3) :
  print(numbers[i])

print()

pets = ("dog", "cat", "bird", "fish")
for i in range(1, len(pets)) :
  print(pets[i])
```

```
4
8
15

cat
bird
fish
```

Combining Tuples

- While values cannot be appended to tuples, tuples can be concatenated together.

```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + 100
print(numbers)
```

Error

```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + (100)
print(numbers)

(4, 8, 15, 16, 23, 42, 100)
```

Combining Tuples

- Two tuples are concatenated together when combined using the addition operator.

```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + (100, 101, 102)
print(numbers)
```

```
(4, 8, 15, 16, 23, 42, 100, 101, 102)
```

Determining if an Element Exists in a Tuple

- An if statement can be used to find if an element/value is present in a tuple.
  - Utilizes the **in** keyword.
  - Does not tell us where (what index) the element was found.

```
numbers = (4, 8, 15, 16, 23, 42)
value = 7
if value in numbers :
  print("Value exists in tuple")
else :
  print("Value does not exist in tuple")
```

```
Value does not exist in list
```

Index Function

- A tuple's index function returns the index of an element/value.
  - Returns the index of the first matching element/value.
  - Case sensitive.

```
numbers = (4, 8, 8, 15, 16, 23, 42)
found_index = numbers.index(8)
print(found_index)
```

```
1
```

Index Function

- If the tuple's index function does not find a matching element/value a ValueError exception will be raised.

```
pets = ("dog", "cat", "bird")
found_index = pets.index("fish")
print(found_index)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    foundIndex = pets.index("fish")
ValueError: tuple.index(x): x not in tuple
>>>
```

Index Function

```
pets = ("dog", "cat", "bird")
try :
  found_index = pets.index("fish")
  print(found_index)
except ValueError :
  print("Value not found")
```

```
Value not found
```

Min Function

- Python's built-in min function returns the smallest value from a tuple.

```
numbers = (20, 40, 10, 30)
min_value = min(numbers)
print(min_value)
```

```
10
```

  - For Strings, the smallest value would be the first, lexicographically.

```
pets = ("dog", "cat", "bird", "Cat", "fish")
min_value = min(pets)
print(min_value)
```

```
Cat
```

Max Function

- Python's built-in max function returns the largest value from a tuple.

```
numbers = (20, 40, 10, 30)
max_value = max(numbers)
print(max_value)
```

```
40
```

  - For Strings, the largest value would be the last, lexicographically.

```
pets = ("dog", "cat", "bird", "Cat", "fish")
max_value = max(pets)
print(max_value)
```

```
fish
```

Converting Lists to/from Tuples

- Python's built-in tuple function returns a list argument as a tuple.

```
numbers = [20, 40, 10, 30]
numbers = tuple(numbers)
print(numbers)
```

```
(20, 40, 10, 30)
```

- Python's built-in list function returns a tuple argument as a list.

```
pets = ("dog", "cat", "bird", "Cat", "fish")
pets = list(pets)
print(pets)
```

```
['dog', 'cat', 'bird', 'Cat', 'fish']
```