Revised: 10/12/2020

Sequence Types

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

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

Colors/Fonts

- Variable Names – Brown
- Standard data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Operators/Punctuation – Black
- Function Names – Purple
- Comments – Gray
- Module Names – Pink

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

String Character Indexes

- Strings are a sequence type and are comprised of characters.
  - Characters can be letters, numbers, symbols and whitespace.

- Every character in a string has an index.

```
example = "Example String"
```

```
E  x  a  m  p  l  e     S  t  r  i  n  g
0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

String Characters

- Characters in a string can be accessed using subscript notation.

```
example = "Example String"
first_character = example[0]
print(first_character)
print(example[8])
```

```
E
S
```

String Character Indexes

- Strings are immutable.
  - Characters in a string cannot be changed.

```
example = "Example String"
example[13] = "G"
print(example)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    example[13] = "G"
TypeError: 'str' object does not support item assignment
>>>
```

String Character Indexes

- Attempting to access an index that does not exist will raise in an IndexError exception.

```
example = "Example String"
character = example[20]
print(character)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    character = example[20]
IndexError: string index out of range
>>>
```

String Length

- A string's length is the total number of characters it contains.
  - Use Python's built-in len function to return the length of a string.

```
example = "Example String"
length = len(example)
print(length)
```

```
14
```

Iterating Over String Characters

- A for loop can iterate over a string's characters.
  - To loop through all or part of a string's characters:

```
example = "Example"
for index in range(0, len(example)) :
  print(example[index])
```

  - To loop through all of a string's characters:

```
example = "Example"
for character in example :
  print(character)
```

```
E
x
a
m
p
l
e
```

Testing if a String is Alphabetic

- The string's isalpha function determines if a string contains only letters.
  - Returns a Boolean

```
example1 = "Example"
example2 = "Example123"

if example1.isalpha() :
  print("example1 contains only letters")

if example2.isalpha() :
  print("example2 contains only letters")
```

```
example1 contains only letters
```

Testing if a String is Numeric

- The string's isdigit function determines if a string contains only numbers.
  - Returns a Boolean

```
example1 = "Example"
example2 = "Example123"

if example1.isdigit() :
  print("example1 contains only numbers")

if example2.isdigit() :
  print("example2 contains only numbers")
```

Testing if a String is Alphanumeric

- The string's isalnum function determines if a string contains only letters and/or numbers.
  - Returns a Boolean

```
example1 = "Example"
example2 = "Example123"

if example1.isalnum() :
  print("example1 contains only letters and/or numbers")

if example2.isalnum() :
  print("example2 contains only letters and/or numbers")
```

```
example1 contains only letters and/or numbers
example2 contains only letters and/or numbers
```

Testing if a String Contains Only Whitespace

- The string's isspace function determines if a string contains only spaces.
  - Returns a Boolean

```
example1 = "Example"
example2 = "   "

if example1.isspace() :
  print("example1 contains only spaces.")

if example2.isspace() :
  print("example2 contains only spaces.")
```

```
example2 contains only spaces.
```

Testing if a String is Uppercase

- The string's isupper function determines if a string contains only uppercase letters.
  - Returns a Boolean

```
example1 = "Example123"
example2 = "EXAMPLE123"

if example1.isupper() :
  print("example1 contains only uppercase letters.")

if example2.isupper() :
  print("example2 contains only uppercase letters.")
```

```
example2 contains only uppercase letters.
```

Testing if a String is Lowercase

- The string's islower function determines if a string contains only lowercase letters.
  - Returns a Boolean

```
example1 = "Example123"
example2 = "example123"

if example1.islower() :
  print("example1 contains only lowercase letters.")

if example2.islower() :
  print("example2 contains only lowercase letters.")
```

```
example2 contains only lowercase letters.
```

Testing if a *Character* is Alphabetic

```
example = "ABC 123 ! xyz"

if example[2].isalpha() :
  print("Character at index 2 is a letter.")

if example[5].isalpha() :
  print("Character at index 5 is a letter.")

if example[7].isalpha() :
  print("Character at index 7 is a letter.")

if example[8].isalpha() :
  print("Character at index 8 is a letter.")

if example[10].isalpha() :
  print("Character at index 10 is a letter.")
```

Testing if a *Character* is Numeric

```
example = "ABC 123 ! xyz"

if example[2].isdigit() :
  print("Character at index 2 is a number.")

if example[5].isdigit() :
  print("Character at index 5 is a number.")

if example[7].isdigit() :
  print("Character at index 7 is a number.")

if example[8].isdigit() :
  print("Character at index 8 is a number.")

if example[10].isdigit() :
  print("Character at index 10 is a number.")
```

Testing if a *Character* is Alphanumeric

```
example = "ABC 123 ! xyz"

if example[2].isalnum() :
  print("Character at index 2 is a letter/number.")

if example[5].isalnum() :
  print("Character at index 5 is a letter/number.")

if example[7].isalnum() :
  print("Character at index 7 is a letter/number.")

if example[8].isalnum() :
  print("Character at index 8 is a letter/number.")

if example[10].isalnum() :
  print("Character at index 10 is a letter/number.")
```

Testing if a *Character* is Whitespace

```
example = "ABC 123 ! xyz"

if example[2].isspace() :
  print("Character at index 2 is a space.")

if example[5].isspace() :
  print("Character at index 5 is a space.")

if example[7].isspace() :
  print("Character at index 7 is a space.")

if example[8].isspace() :
  print("Character at index 8 is a space.")

if example[10].isspace() :
  print("Character at index 10 is a space.")
```

Testing if a *Character* is an Uppercase Letter

```
example = "ABC 123 ! xyz"

if example[2].isupper() :
  print("Character at index 2 is an uppercase letter.")

if example[5].isupper() :
  print("Character at index 5 is an uppercase letter.")

if example[7].isupper() :
  print("Character at index 7 is an uppercase letter.")

if example[8].isupper() :
  print("Character at index 8 is an uppercase letter.")

if example[10].isupper() :
  print("Character at index 10 is an uppercase letter.")
```

Testing if a *Character* is a Lowercase Letter

```
example = "ABC 123 ! xyz"

if example[2].islower() :
  print("Character at index 2 is a lowercase letter.")

if example[5].islower() :
  print("Character at index 5 is a lowercase letter.")

if example[7].islower() :
  print("Character at index 7 is a lowercase letter.")

if example[8].islower() :
  print("Character at index 8 is a lowercase letter.")

if example[10].islower() :
  print("Character at index 10 is a lowercase letter.")
```

Replacing parts of a String

- The string's replace function replaces part of a string with new data.
- Two arguments (both strings)- first is the string to find, second is what to replace it with. *CASE SENSITIVE*

```
orig_string = "Today is Monday."
new_string = orig_string.replace("Monday", "Tuesday")
print(new_string)
```

```
Today is Tuesday.
```

Note the value of orig_string **does not change**.
The replace method returns a new string with the every sequence of the first argument replaced with the second argument.

Replacing parts of a String

- The string's replace function replaces all matches.

```
orig_string = "Today is Monday."
new_string = orig_string.replace("day", "night")
print(new_string)
```

```
Tonight is Monnight.
```

String Slicing

- String slicing selects a range of characters from a string.
  - String slices are commonly referred to as *substrings*.
- The general syntax for slicing a string is:

***string*[*startIndex*:*endIndex*]**

- This will return a string containing all characters between those indexes.
  - The start index is inclusive.
  - The end index is exclusive.

String Slicing

```
college = "Community College of Philadelphia"
slice = college[10:17]
print(slice)
```

```
College
```

String Slicing

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

```
college = "Community College of Philadelphia"
slice = college[10:]
print(slice)
```

```
College of Philadelphia
```

String Slicing

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

```
college = "Community College of Philadelphia"
slice = college[:17]
print(slice)
```

```
Community College
```

String Slicing

- String slicing is safe from IndexError exceptions.

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

```
college = "Community College of Philadelphia"
slice = college[13:7]
print(slice)
```

```
[]
```

String Slicing

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

```
college = "Community College of Philadelphia"
slice = college[21:100]
print(slice)
```

```
Philadelphia
```

String Slicing

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

```
college = "Community College of Philadelphia"
slice = college[-5:9]
print(slice)
```

```
Community
```

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

String Slicing

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

```
college = "Community College of Philadelphia"
slice = college[-12:]
print(slice)
```

```
Philadelphia
```

String Slicing

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

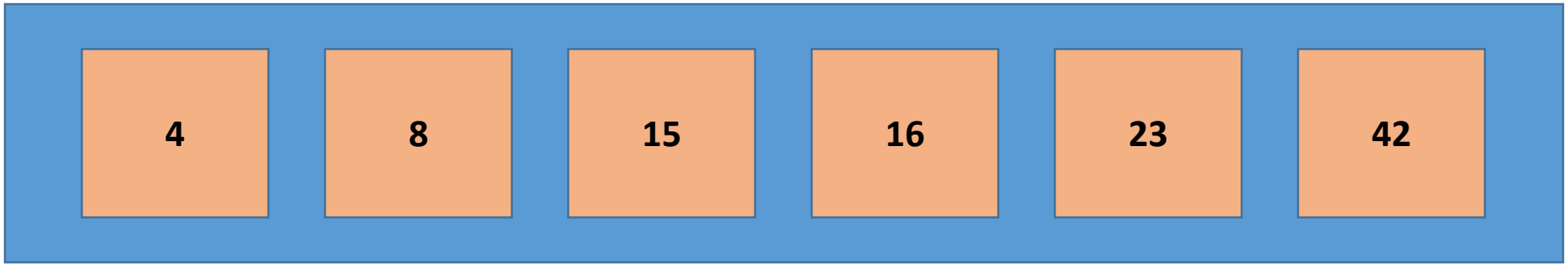
```
college = "Community College of Philadelphia"
slice = college[-17:-24]
print(slice)
```

```
College
```

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

What is a list?

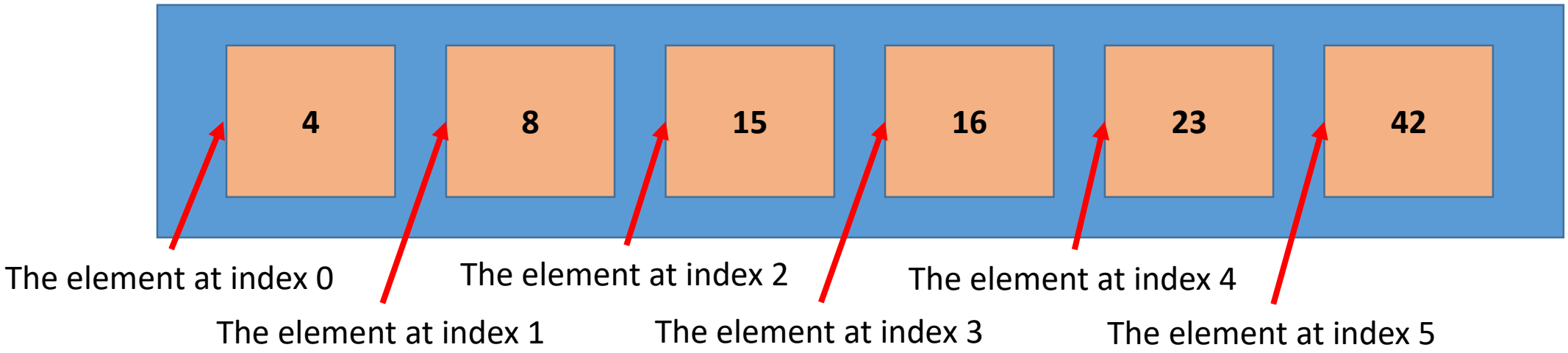
- An ***list*** is a sequence object that has multiple values.
  - Another way to look at it is a variable that has multiple values.



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
>>>
```

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
>>>
```

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
>>>
```

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]
```

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.

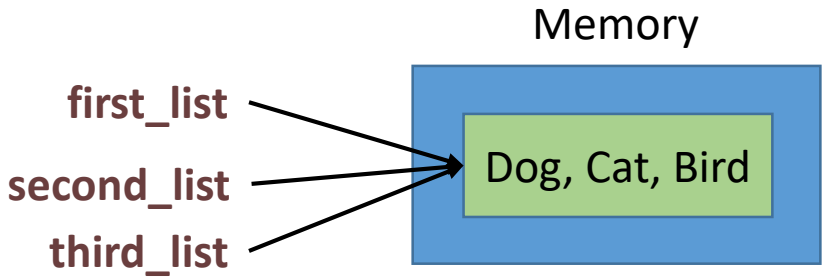
```
pets = ["dog", "cat", "bird", "fish"]
slice = pets[-2:-3]
print(slice)
```

```
[]
```

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 Lists

- 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
```

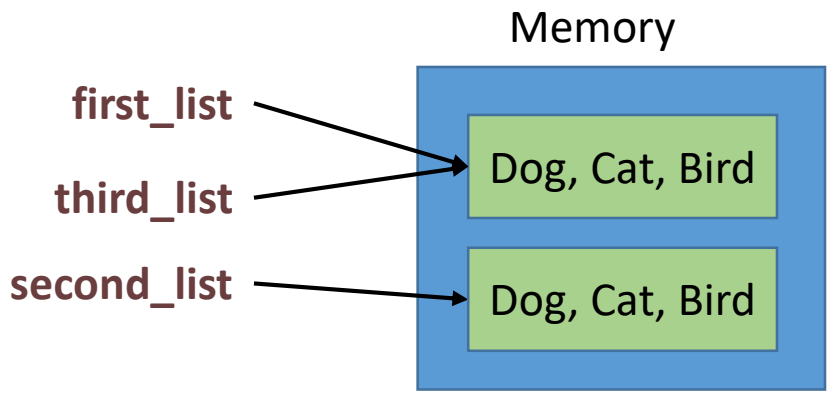
Testing Equality of Lists

- 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 a 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

Multidimensional 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]`?

Multidimensional Lists

- Rows in a multidimensional list 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 list.

```
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

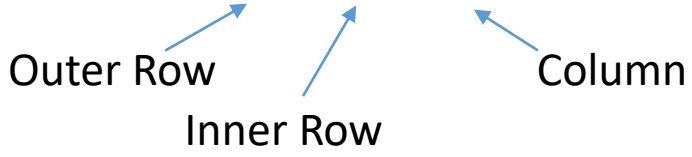
- There is no limit to the number of dimensions a list can have.

- A three-dimensional list:

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

- In the case of a three dimensional list, 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

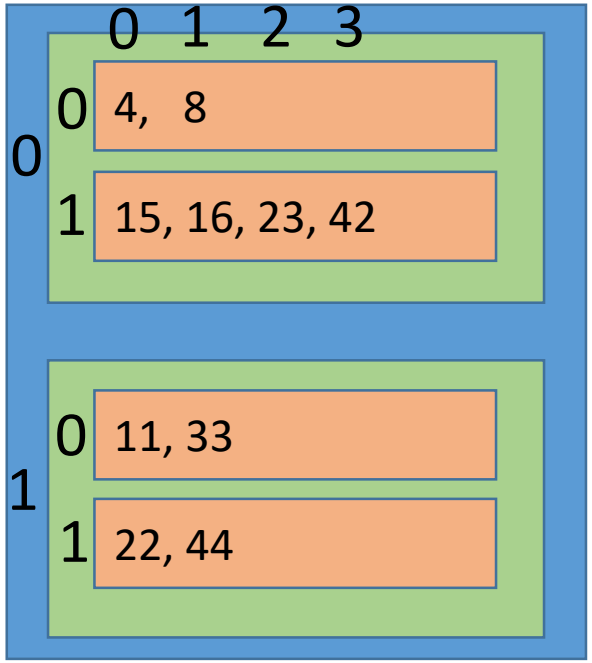
0

1

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)
```