ISYS90088 Introduction to Application Development

Week 6 – Contd. from week 5 Nested loops; While
Followed by Lists, tuples

Semester 2 , 2018

Dr Antonette Mendoza

s

Objectives

After completing this lecture, you will be able to:

- Work with nested loops – while

- Work with lists and tuples:

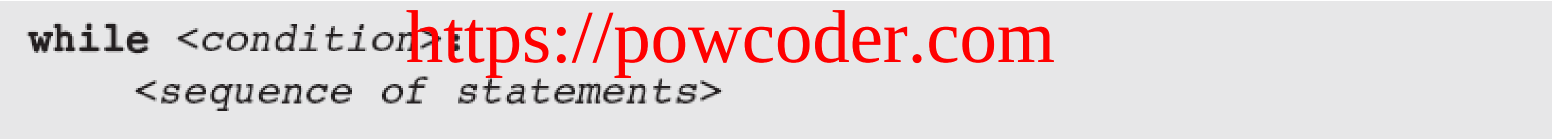
  - Construct lists and access items in those lists
  - Use methods to manipulate lists
  - Perform traversals of lists to process items in the lists
  - Tuples

Conditional Iteration: The `while` Loop

- The **while** loop can be used to describe conditional iteration
  - Example: A program’s input loop that accepts values until user enters a ‘**sentinel**’ that terminates the input

The Structure and Behavior of a `while` Loop

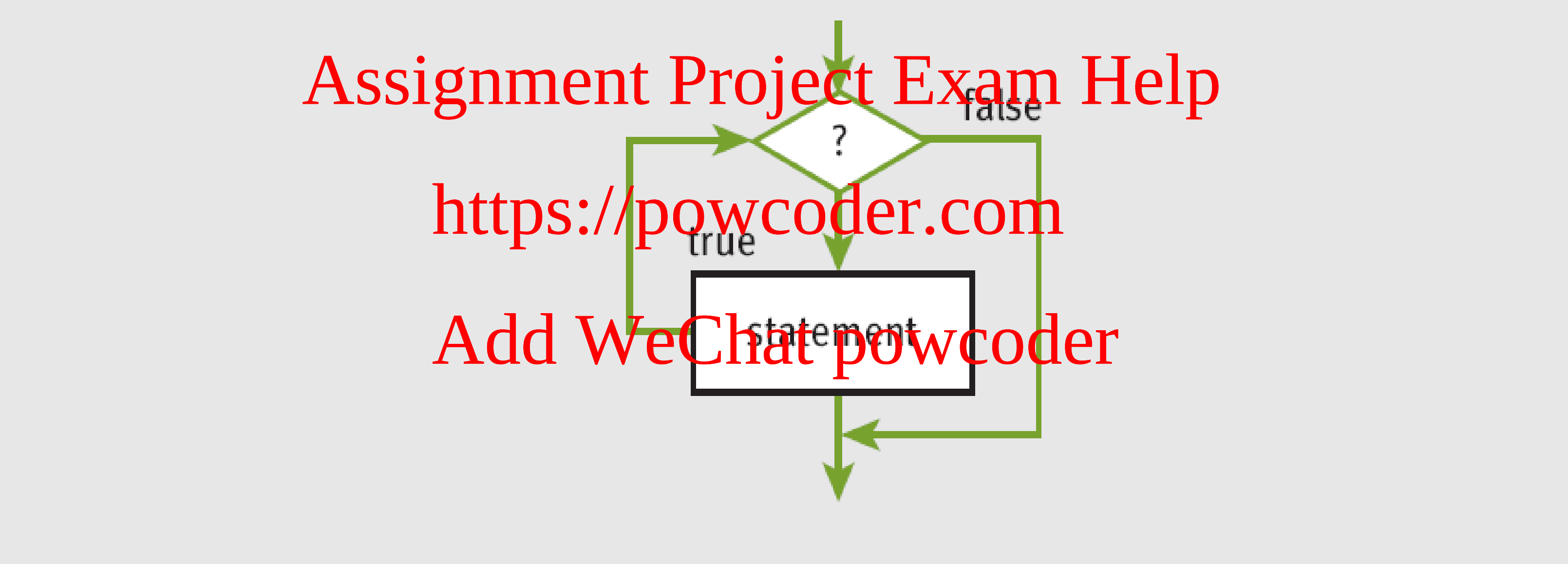
- Conditional iteration requires that condition be tested within loop to determine if it should continue
  - Called **continuation condition**



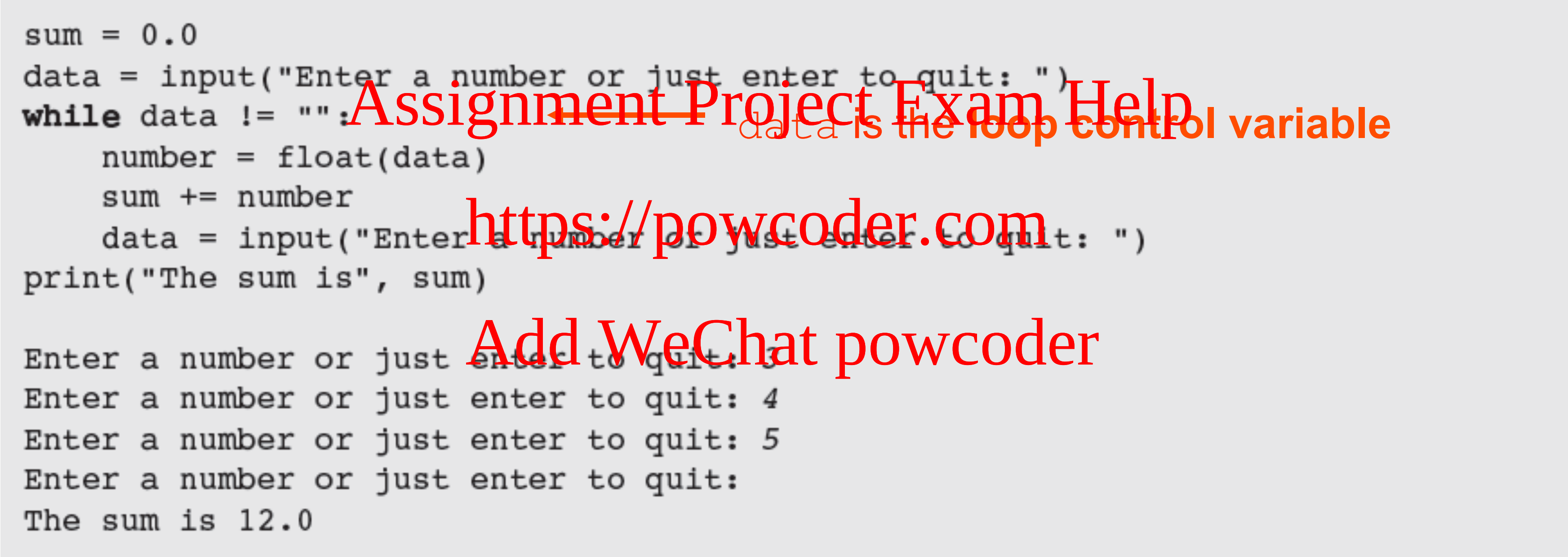
```
while <condition>:
     <sequence of statements>
```

  - Improper use may lead to **infinite loop**
- **while** loop is also called **entry-control loop**
  - Condition is tested at top of loop
  - Statements within loop can execute zero or more times

The Structure and Behavior of a `while` Loop (continued)



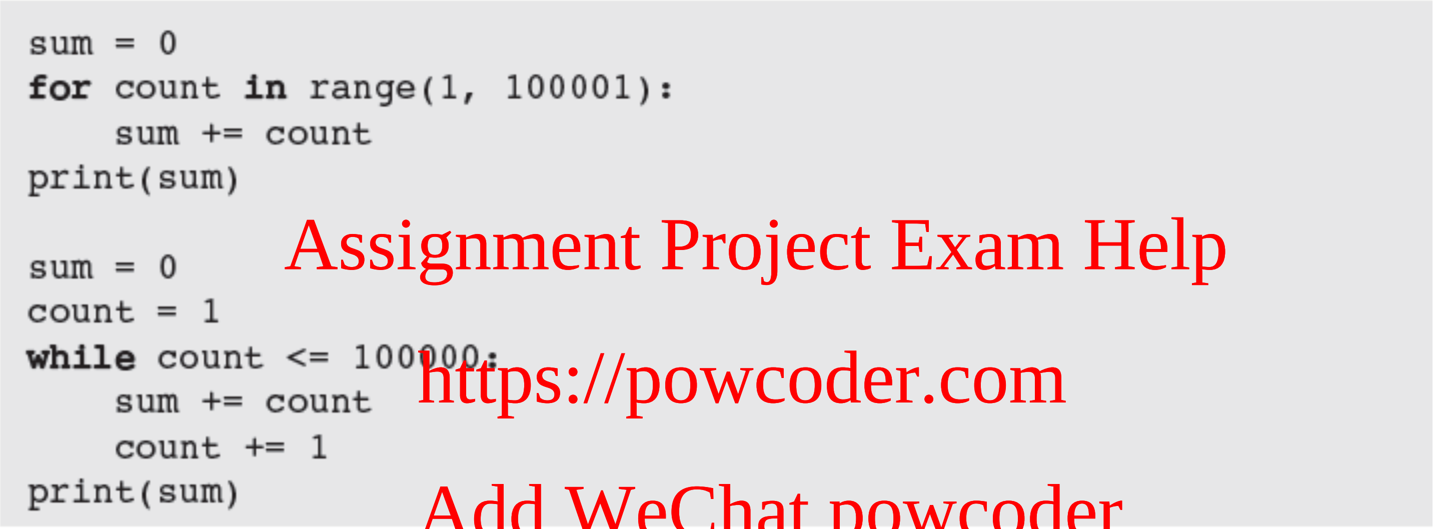
The Structure and Behavior of a `while` Loop (continued)



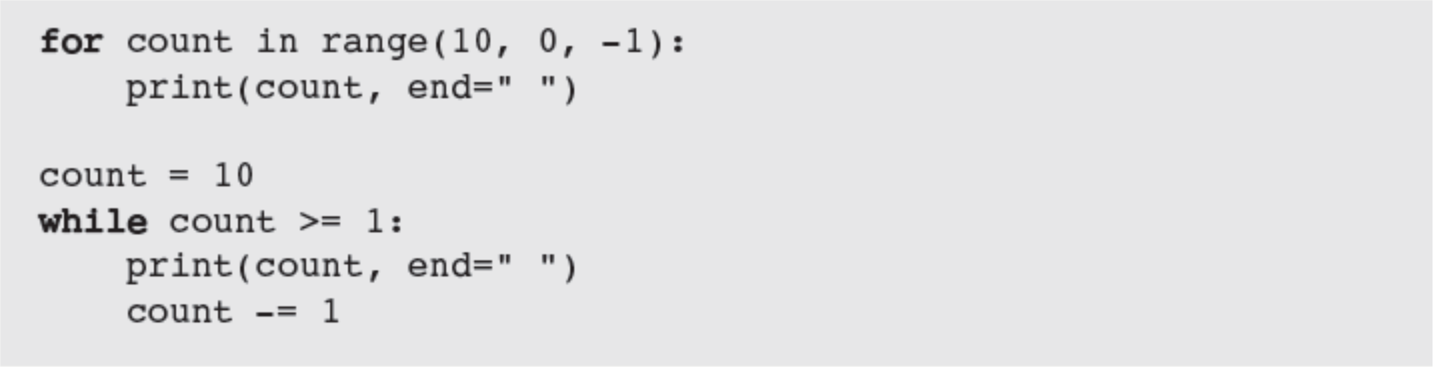
```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

`data` is the loop control variable

Count Control with a `while` Loop



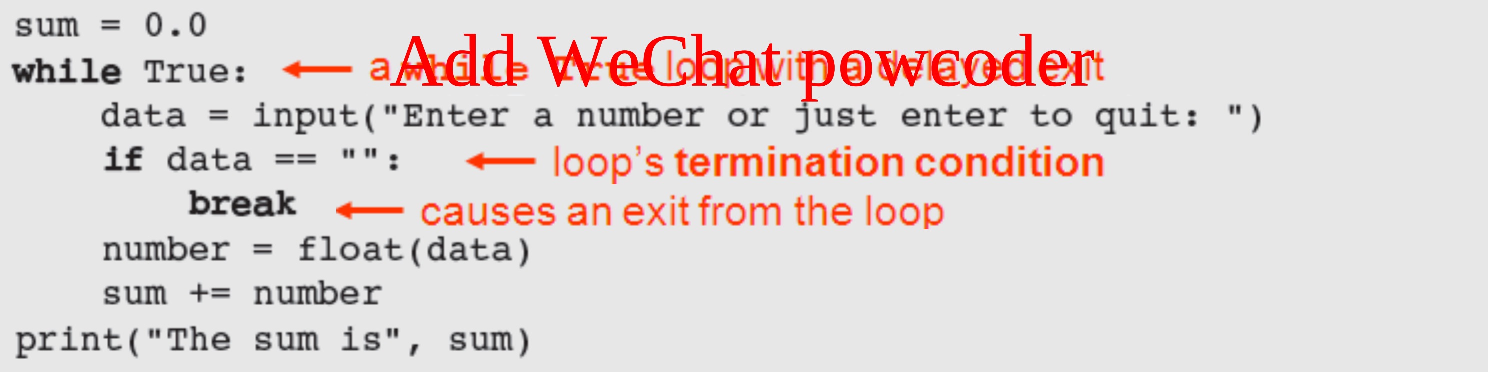
```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)

sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```



```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

The `while True` Loop and the `break` Statement

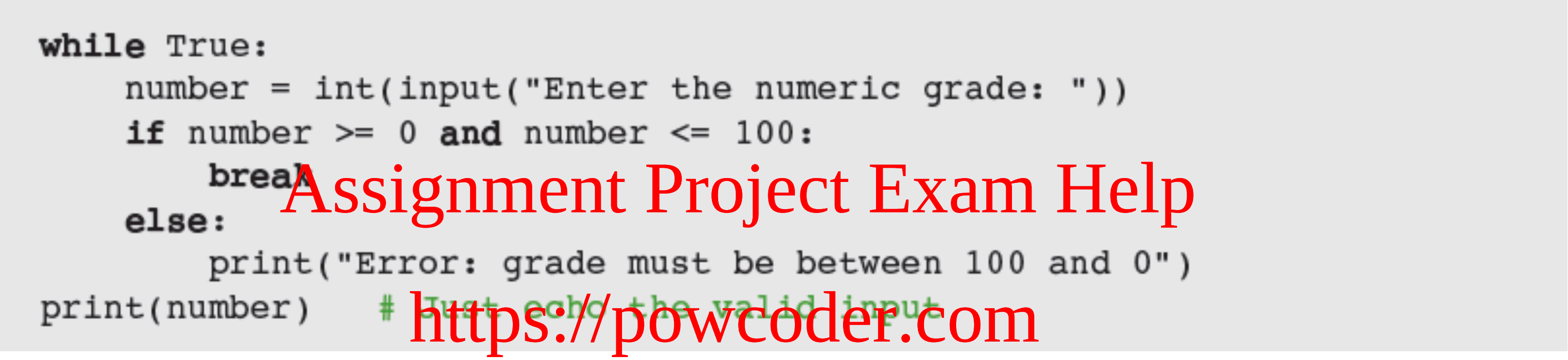
- **while** loop can be complicated to write correctly
  - Possible to simplify its structure and improve its readability



```
sum = 0.0
while True:
    data = input("Enter a number or just enter to quit: ")
    if data == "":
        break
    number = float(data)
    sum += number
print("The sum is", sum)
```

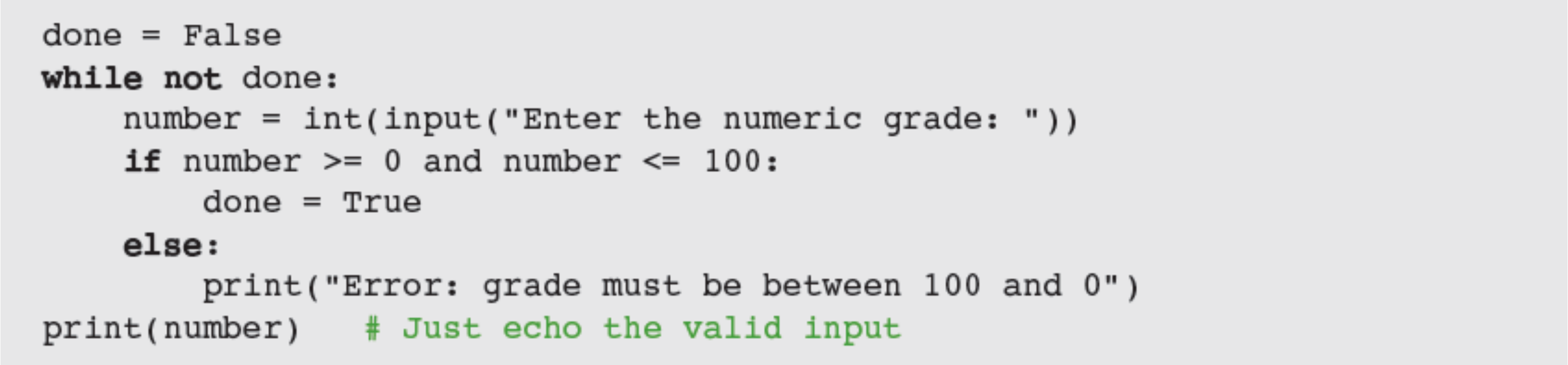
a while True loop with a delayed exit
loop's termination condition
causes an exit from the loop

The `while True` Loop and the `break` Statement (continued)



```
while True:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        break
    else:
        print("Error: grade must be between 100 and 0")
print(number)   # Just echo the valid input
```

- Alternative: Use a Boolean variable to control loop



```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)   # Just echo the valid input
```

Recap: break statement

- It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

- If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

```
>>> break
```

Introduction – lists, tuples and Dictionaries

- A **list** allows the programmer to manipulate a sequence of data values of any types
  - Indicate by enclosing its elements in []

- A **tuple** resembles a list, but is immutable

  - Indicate by enclosing its elements in ()

- A **dictionary** organizes data values by association with other data values rather than by sequential position
- Lists and dictionaries provide powerful ways to organize data in useful and interesting applications

Next week

Lists

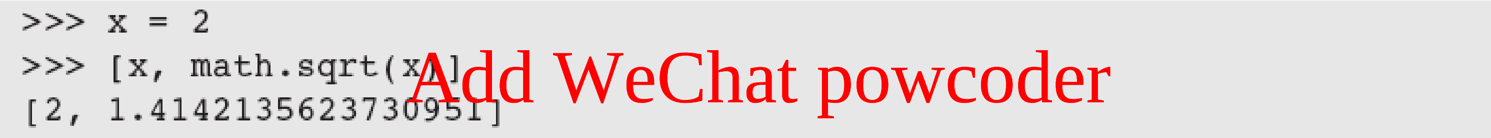
- List: Sequence of data values (**items** or **elements**)
- Some examples:
  - Shopping list for the grocery store
  - Guest list for a wedding
  - Recipe, which is a list of instructions
  - Text document, which is a list of lines
  - Words in a dictionary
- Each item in a list has a unique **index** that specifies its position (from 0 to length – 1)

List Literals and Basic Operators

- Some examples:

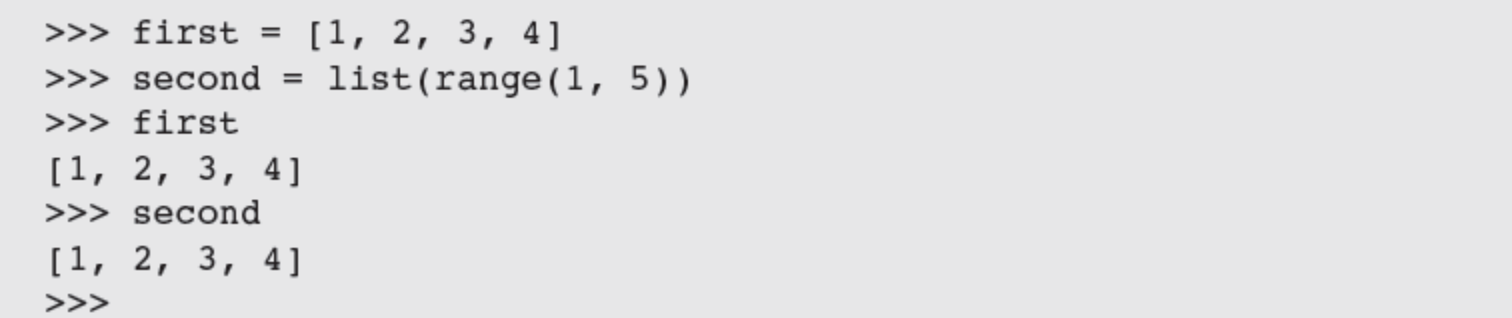
```
['apples', 'oranges', 'cherries']
[[5, 9], [541, 78]] – list of lists!
```

- When an element is an expression, its value is included in the list:



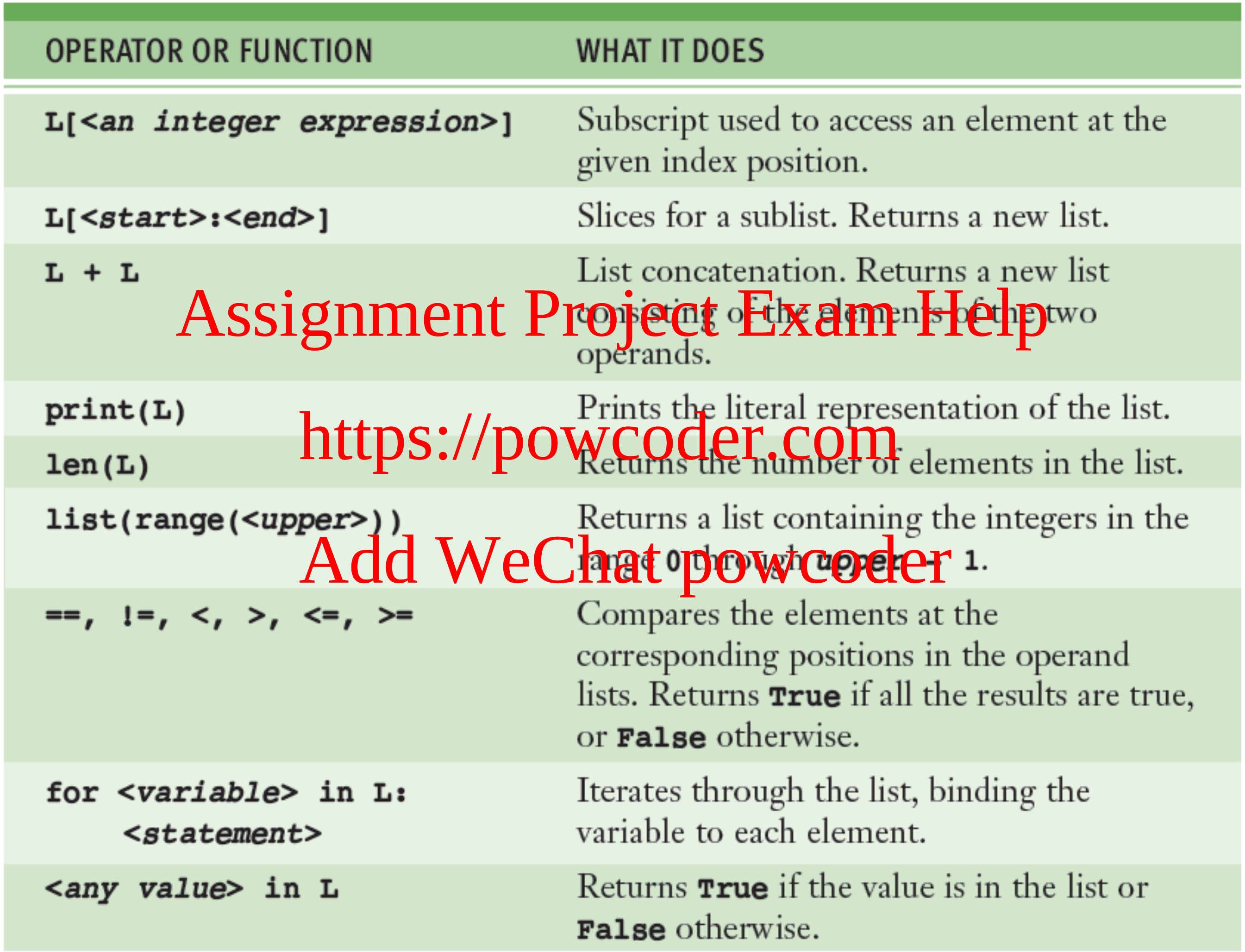
```
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
```

- Lists of integers can be built using **range**:



```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
>>>
```

List Literals & Basic Operators (cont.)



OPERATOR OR FUNCTION	WHAT IT DOES
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L + L`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range `0` through `upper - 1`.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns **`True`** if all the results are true, or **`False`** otherwise.

`for <variable> in L:`
`    <statement>`
	Iterates through the list, binding the variable to each element.

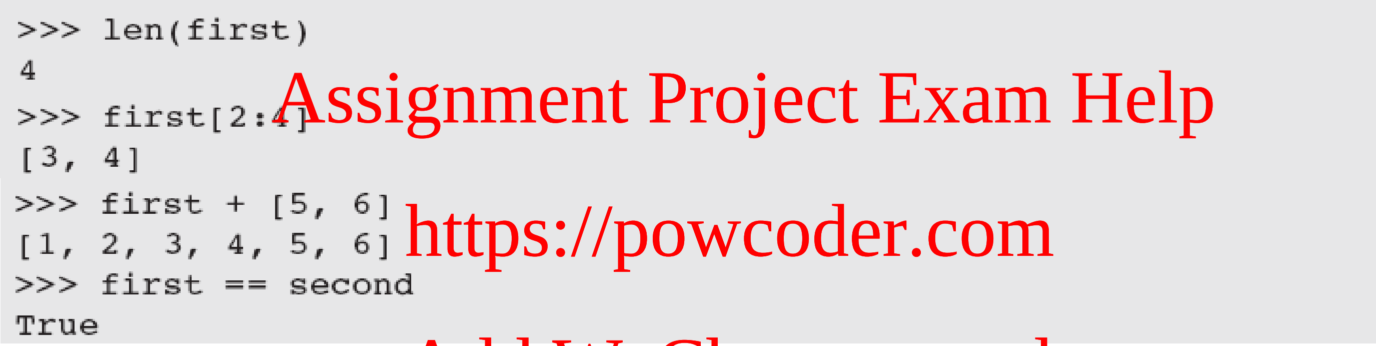
`<any value> in L`	Returns **`True`** if the value is in the list or **`False`** otherwise.

Taken from - Fundamentals of Python: First Programs

List Literals and Basic Operators (continued)

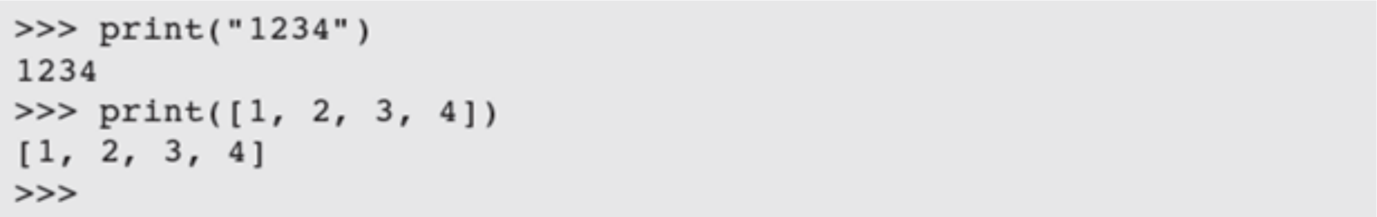
- **`len`**, **`[]`**, **`+`**, and **`==`** work on lists as expected:

```
>>> first = [1,2,3,4]
>>> second = list(range(1,5))
```



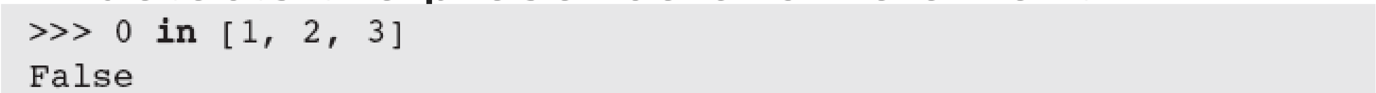
```
>>> len(first)
4
>>> first[2:4]
[3, 4]
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
```

- To print the contents of a list:



```
>>> print("1234")
1234
>>> print([1, 2, 3, 4])
[1, 2, 3, 4]
>>>
```

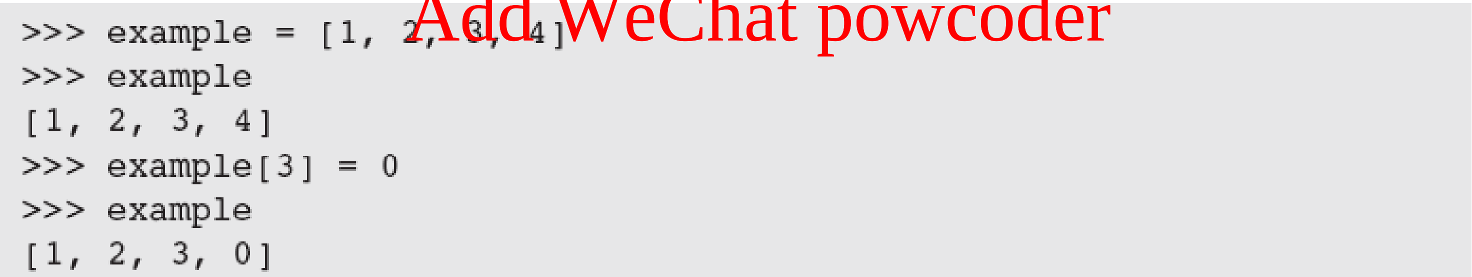
- **`in`** detects the presence of an element:



```
>>> 0 in [1, 2, 3]
False
```

Replacing an Element in a List

- A list is **mutable**
  - Elements can be inserted, removed, or replaced
  - The list itself maintains its identity, but its **state**—its length and its contents—can change
- Subscript operator is used to replace an element:

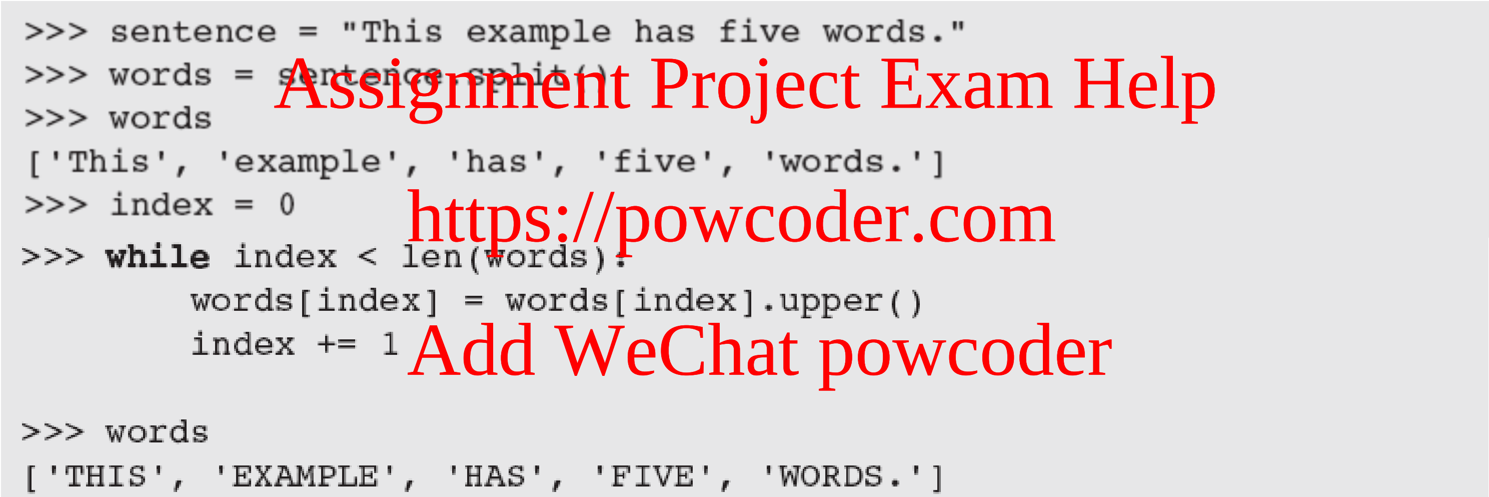


```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

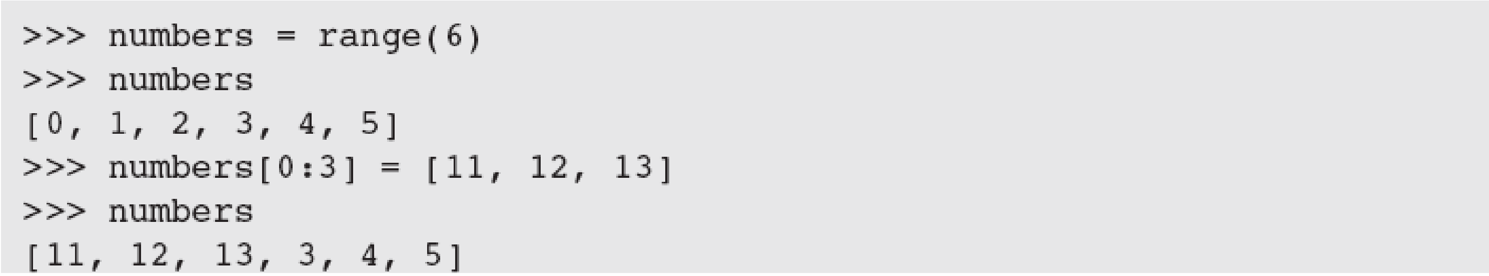
  - Subscript is used to reference the **target** of the assignment, which is not the list but an element's position within it

Replacing an Element in a List (continued)

- Examples: to make all words in the list uppercase



```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> index = 0
>>> while index < len(words):
        words[index] = words[index].upper()
        index += 1

>>> words
['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
```



```
>>> numbers = range(6)
>>> numbers
[0, 1, 2, 3, 4, 5]
>>> numbers[0:3] = [11, 12, 13]
>>> numbers
[11, 12, 13, 3, 4, 5]
```

Lists: index()

- **Index :** returns the index of the first element whose value is equal to the item. A ValueError exception is raised if the item is not found in the list.

- **Syntax:**

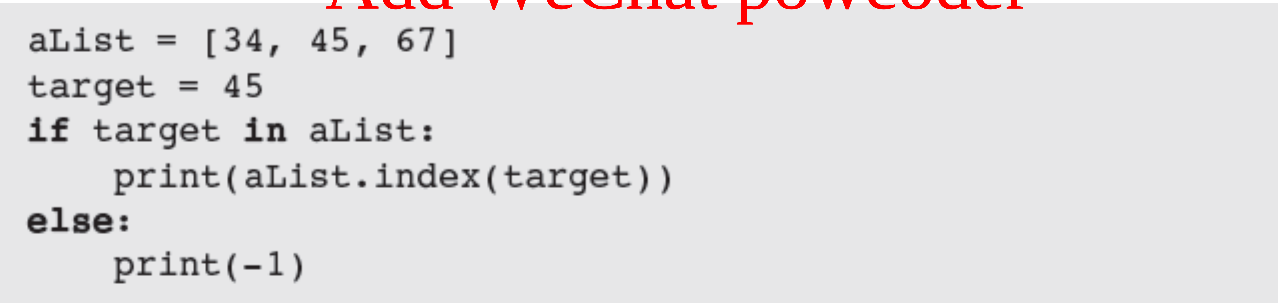
```
<list>.index(item)
```

Returns the first element whose value is equal to the item.

```
>>> n = [1,2,3,4]
>>> n.index(2)
1
```

Searching a List

- **in** determines an element’s presence or absence, but does not return position of element (if found)
- Use method **index** to locate an element’s position in a list
  - Raises an error when the target element is not found



```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

Try a couple on IDLE!!!!

Example: index ()

```
#example to illustrate the index(). This simple program
#replaces #an item in a list once the index is known

food = ['pizza', 'burger', 'chips’]
print(‘here are the list of items’)
print(food)
item = input('which item would you like to change:’)
#searching in the list for the item or value
if item not in food:
    print('the item is not in the list’)
else:
    item_index = food.index(item)
    print(item_index)
#enter the new value replacing the old one
    new_item = input('enter the new item:')
    food[item_index] = new_item
    print(food)
```

Lists: append ()

- **Append:** adds items into the list one by one - one item at a time to the end of the list

- **Syntax:**

```
<list>.append(item)
```

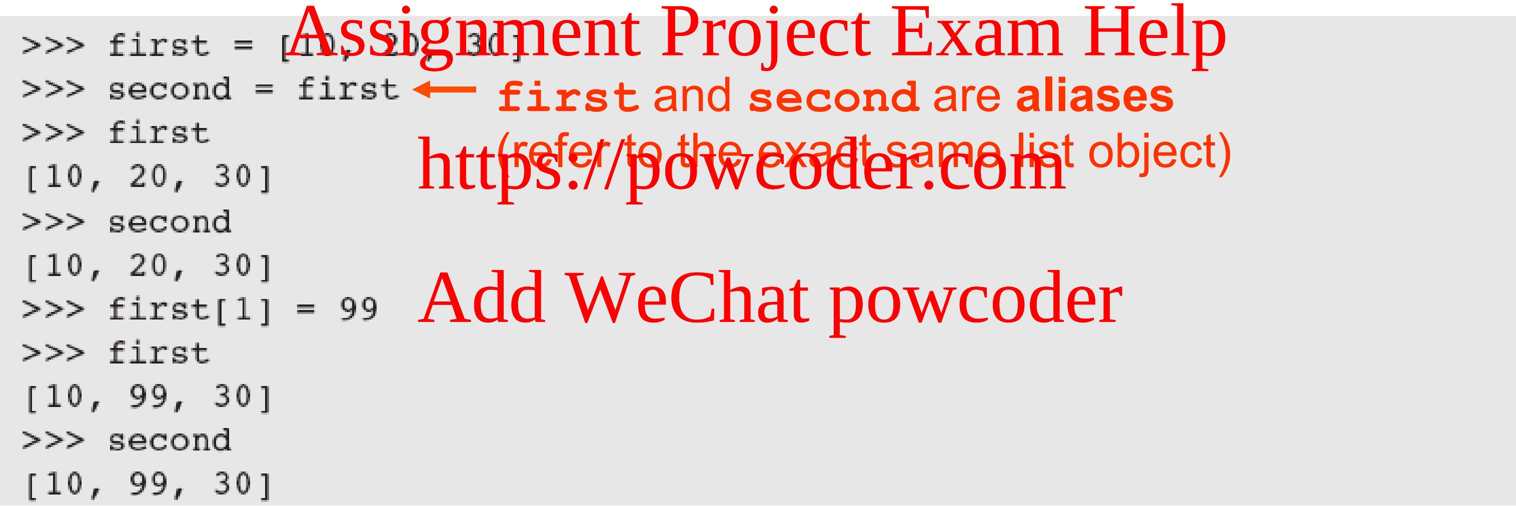
Returns a list with an item

Example: append()

```
name_list = []
again = 'y'
#add names into the list – adds it to the end of list
while again == 'y':
    name = input('enter the name:')
    name_list.append(name)
    #to add another name into the list
    print('do you want to add more name')
    again = input('y = yes, anything else = no:')

#display the names that were added
print('here are the names:')
print(name)
```

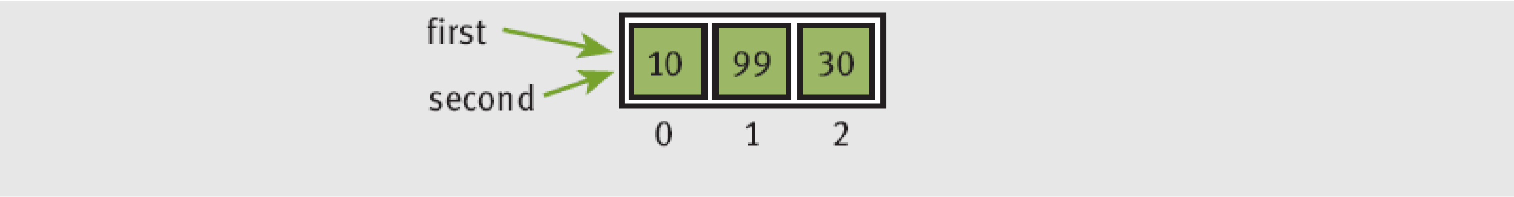
Aliasing and Side Effects

- Mutable property of lists leads to interesting phenomena:



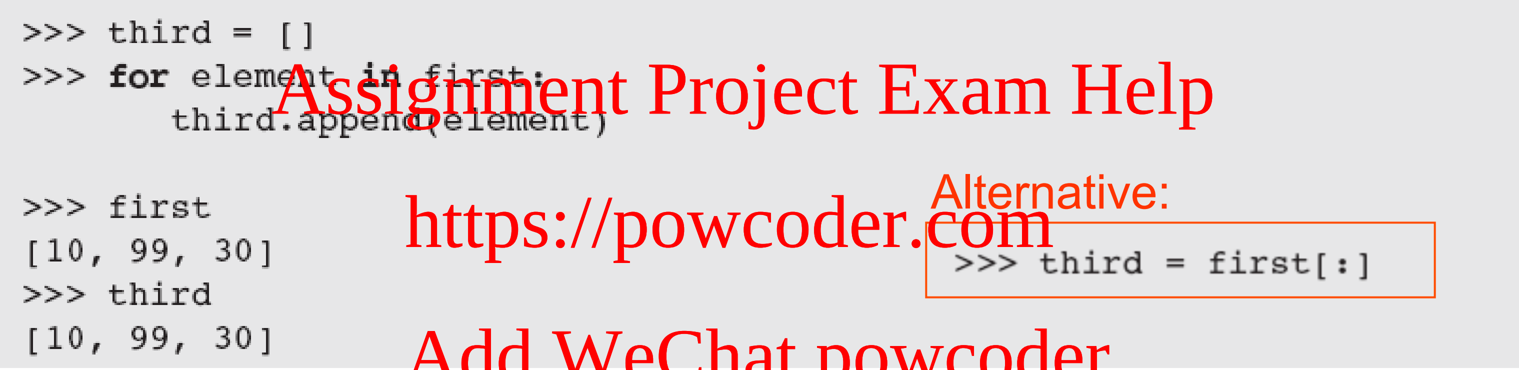
```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```

first and **second** are **aliases** (refer to the exact same list object)



Aliasing and Side Effects (continued)

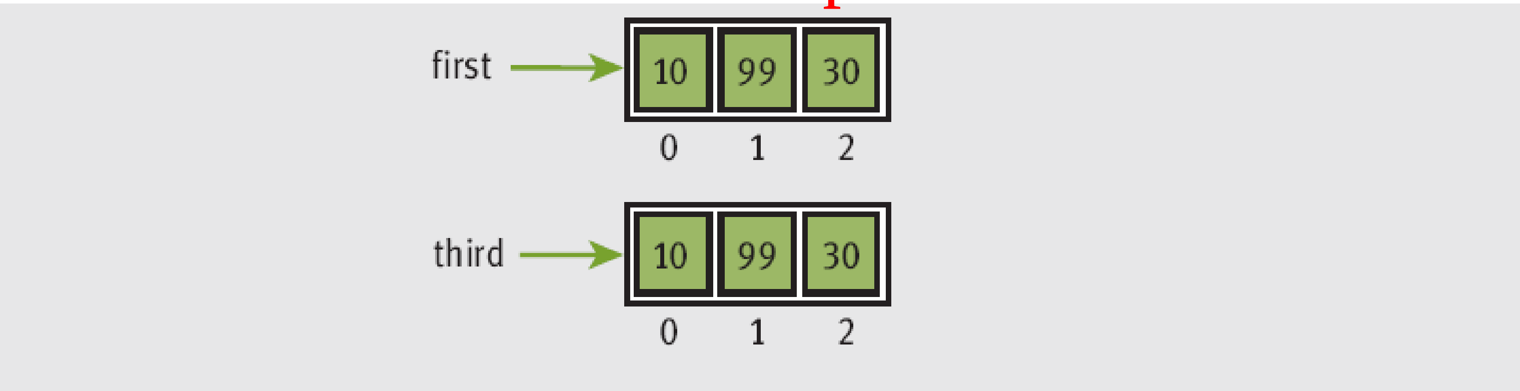
- To prevent aliasing, copy contents of object:



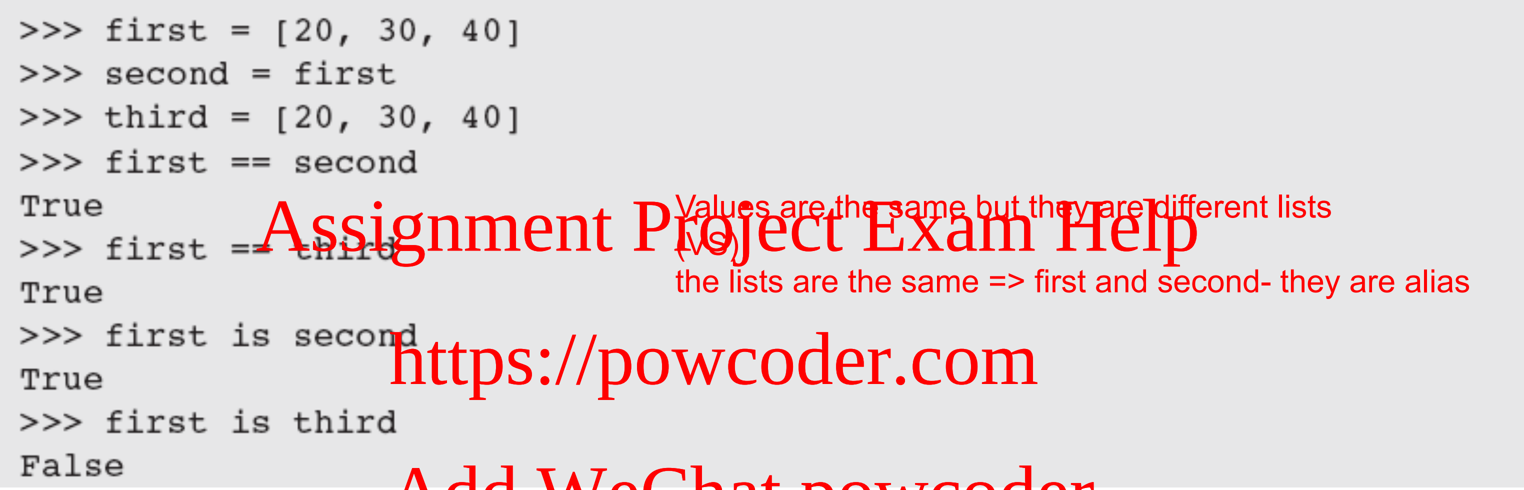
```
>>> third = []
>>> for element in first:
        third.append(element)

>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
```

Alternative:

```
>>> third = first[:]
```

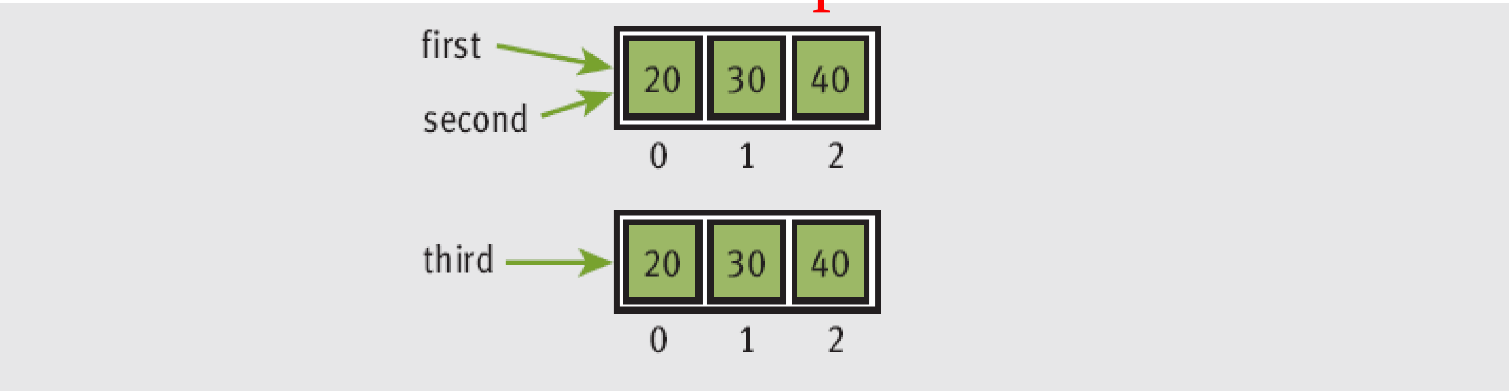


Equality: Object Identity and Structural Equivalence



```
>>> first = [20, 30, 40]
>>> second = first
>>> third = [20, 30, 40]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```

Values are the same but they are different lists
(VS)
the lists are the same => first and second- they are alias



Sorting a List

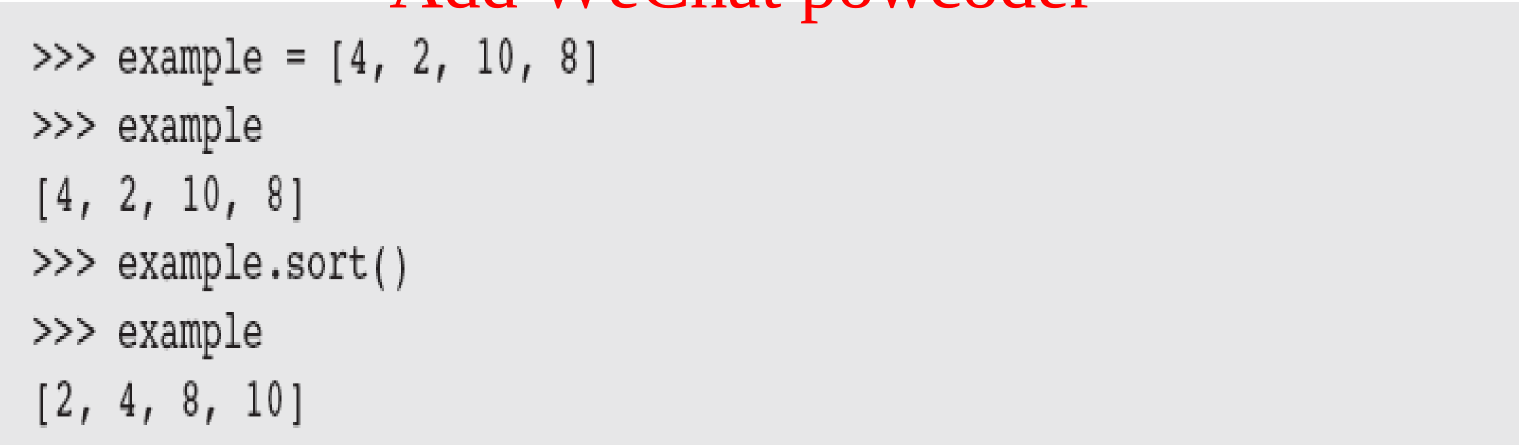
- A list’s elements are always ordered by position, but you can impose a **natural ordering** on them
  - For example, in alphabetical order or ascending order
- When the elements can be related by comparing them <, >, and ==, they can be sorted
  - The method **sort** mutates a list by arranging its elements in ascending order

Lists: sort()

- **sort:** it simply rearranges elements in a list so they appear to be ascending order.
- **Syntax:**

```
<list>.sort()
```

Returns the list sorted



```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

Lists: sort()

```
>>> name = ['anne', 'david', 'james', 'cathy',
'bob']
>>> name.sort()
>>> name
['anne', 'bob', 'cathy', 'david', 'james']
>>> list1 = [3,2, 1, 1, 2, 4, 54, 45]
>>> list1.sort()
>>> list1
[1, 1, 2, 2, 3, 4, 45, 54]
>>>
```

Example: Using a List to Find the Median of a Set of Numbers

```
#median of numbers in a list. Assume the input is a text - integers
listofnumbers = input ('enter a list of numbers:')
numbers = []
words = listofnumbers.split()
for word in words:
    numbers.append(float(word))
print(numbers)

#sort the list and print the median or its midpoint
#numbers.sort() or use it this way numbers = sorted(numbers)
numbers.sort()
print(numbers)
midpoint = len(numbers) // 2
print("the median is", end=" ")
if len(numbers) % 2 == 1:
    print(numbers[midpoint])
else:
    print((numbers[midpoint] + numbers[midpoint -1]) /2)
```

Lists: insert ()

- **Insert :** insert an item into the item at a specific position. Two arguments are provided to this method: the index specifying where the item should be inserted and; the item that you want to insert.
- **Syntax:**

```
<list>.insert(<index>,<item> )
```

- Returns a list with the item added.

Example: insert ()

```
>>> list1 = ['cat', 'dog', 'horse']
>>> list1.insert(3, 'bird')
>>> list1
['cat', 'dog', 'horse', 'bird']
>>> list1 = ['cat', 'dog', 'horse']
>>> list1.insert(3, 'bird')
>>> list1
['cat', 'dog', 'horse', 'bird']
>>> name = ['anne', 'david']
>>> name.insert(0, 'anto')
>>> name
>>> name.insert(4, '3')
>>> name
['anto', 'anne', 'david', '3']
>>> name.insert(4, 1)
>>> name
['anto', 'anne', 'david', '3', 1]
>>>
```

Lists: reverse()

- **reverse :** it simply reverses the order of the items in the list.
- **Syntax:**

```
<list>.reverse()
```

Returns the list reversed.

- Example:

```
>> name
['ant', 'bee', 'cat', 'dog', 'elephant']
>>> name.reverse()
>>> name
['elephant', 'dog', 'cat', 'bee', 'ant']
>>>
```

Lists: remove()

- **remove :** removes an item from the list. You pass an item as an argument and the first element containing that item is removed.
  - This reduces the size of the list one by one
  - All of the elements after the removed element are shifted one position towards the beginning of the list

- **Syntax:**

```
<list>.remove(item)
```

Returns a list with one less item .

Example: remove()

```
# example to illustrate the remove().

food = ['pizza', 'burger', 'chips']
print(food)
item = input('which item would you like to
remove:')
if item not in food:
    print('the item is not in the list')

else:
    food.remove(item)
    print('here is the new list:')
    print(food)
```

Lists: del()

- **del :** some situations require that you have to remove an element from a specific index in the list regardless of what item is actually stored in that index.

- **Syntax:**

```
del <list[index]>
```

Returns a list with one less item .

- Example:

```
>>> name = ['anne', 'david', 'james']
>>> del name[2]
>>> name
['anne', 'david']
>>>
```

Examples: reversing and sorting a List in loops

```
# example to reverse a list of items in loops
   listofvalues =[10,15,20,40]
   for i in reversed(listofvalues):
          print (i)

# example to sort a list of items
   listofvalues =[10,25,20,40, 11]
   for i in sorted(listofvalues):
          print (i)


# another way of using sort – example
   listofvalues.sort()
   for i in listofvalues:
          print(i)
```

Lists: max() and min() functions

max: takes in a list as an argument and returns the max value in that list.

min: takes in a list and returns the min value in that list

Syntax:

```
   min(<list>)
   max(<list>)
```

Examples:

```
>>> list1 = [3,2, 1, 1, 2, 4, 54, 45]
>>> max(list1)
54
>>> min(list1)
1
```

BREAK!

Lists: two-dimensional

A 2-dimensional list is a list that has others lists as its elements

Examples:

```
>>> students = [['joe', 'jack', 'mary'],
['sam', 'jane']]
>>> students
[['joe', 'jack', 'mary'], ['sam', 'jane']]
>>> students[0]
Joe
>>>student[0][1]
'jack'
```

Lists: two-dimensional

Useful when working with multiple lists. Example: write a program that calculates the grade-average for a teacher. Lets say we have 2 students each of who do three assessments. How can we represent and work with the lists?

Ass 1	Ass2	mid-sem test
10	15	8
5	10	6

Lists: two-dimensional

```
>>>Scores = [[10,15,8], [5,10,6]]
>>>scores[0][0]
10
```

Ass 1	Ass2	mid-sem test
10	15	8
5	10	6

Lists: two-dimensional – work on this program at home!!!

- Program to multiply two matrices using nested loops

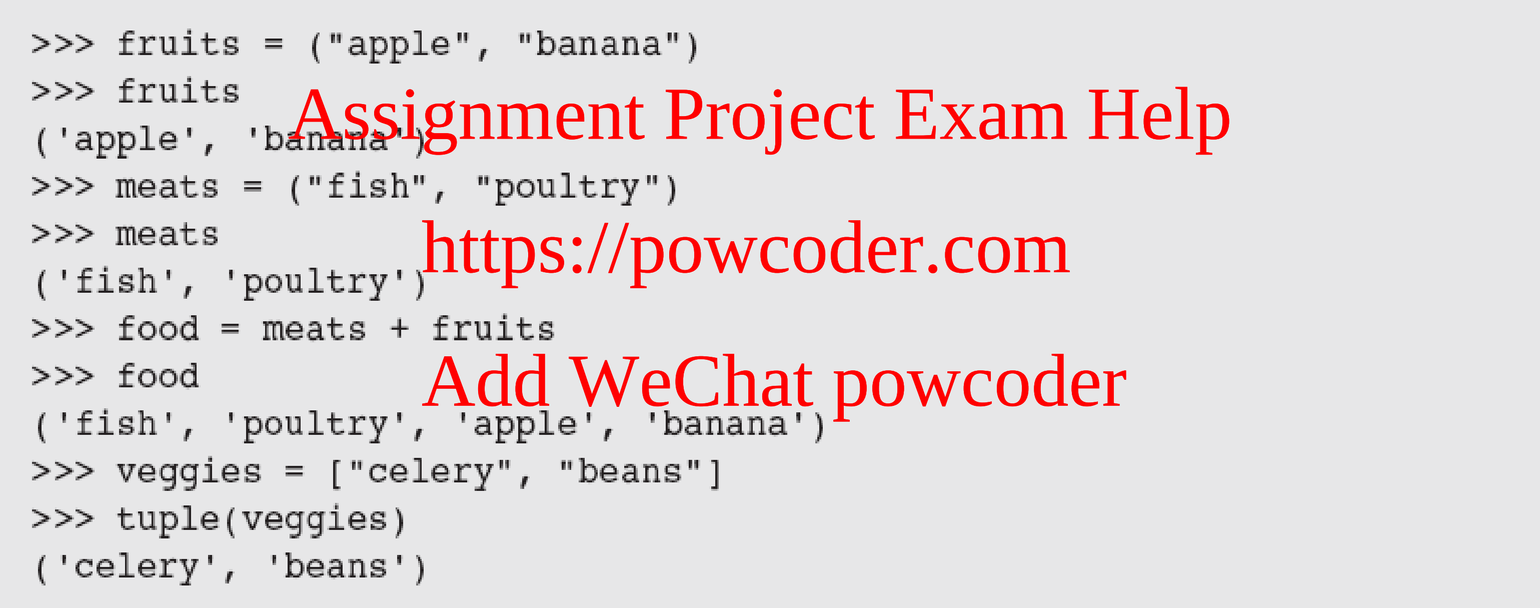
```
# add two 2x2 matrix
X = [[1,2], [2,1],[1,3]]
Y = [[4,1], [2,1], [2,2]]
result = [[0,0],[0,0],[0,0]]
# iterate through rows
for i in range(len(X)):
 # iterate through columns
    for j in range(len(X[0])):
         result[i][j] = X[i][j] + Y[i][j]
for r in result:
   print(r)
```

Tuples

- A **tuple** resembles a list, but is immutable
  - Indicate by enclosing its elements in ()
- The differences between tuples and lists are:
  - the tuples cannot be changed unlike lists
  - tuples use parentheses, whereas lists use square brackets

- Creating a tuple is as simple as putting different comma-separated values.

Tuples

- Lists can be converted to tuples; two sets of tuples can be concatenated
- 

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

- Most of the operators and functions used with lists can be used in a similar fashion with tuples

Tuples

- Most of the operators and functions used with lists can be used in a similar fashion with tuples:

  - The empty tuple is written as two parentheses containing nothing

```
    tup1 = ();
```

  - To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
    tup1 = (50,);
```

Tuples

- Most of the operators and functions used with lists can be used in a similar fashion with tuples:

  - The empty tuple is written as two parentheses containing nothing

```
      tup1 = ();
```

for lists: `list1 = [ ]`

  - To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
      tup1 = (50,);
```

```
   For lists: list1 = [50]
```

Tuples

- Like string indices, tuple indices start at 0. The operations performed are: concatenation, iteration, in, slicing and indexing

- Accessing Values in Tuples: use the square brackets for slicing along with the index or indices to obtain value available at that index.

- Updating Tuples - Tuples are immutable which means you cannot update or change the values of tuple elements.

- Delete Tuple Elements - Removing individual tuple elements is not possible.

Tuples

- To explicitly remove an entire tuple, just use the **del** statement. For example:

```
tuple1 = ('physics', 'chemistry', 1997, 2000)
print (tuple1)
del tuple1
print ("After deleting tuple : ")
print (tuple1)
```

- This produces the following result (check example). Note an exception raised, this is because after **del tup** tuple does not exist any more.

Example:

```
>>>tuple3 = (1,2,3)
>>>list(tuple3)
[1,2,3]
```

Tuples

Built-in Tuple Functions can be used:

```
## length, max and min in a tuple
   tuple1, tuple2 = ('zar', 'xyz', 'zara'), (100, 500, 20)
   print ("Max value element : ", max(tuple1))
   print ("Max value element : ", max(tuple2))
   print ("Min value element : ", min(tuple1))
   print ("Min value element : ", min(tuple2))
   print ("First tuple length : ", len(tuple1))
   print ("Second tuple length : ", len(tuple2))

#convert a list of items into tuples
   Listofitems = [23, 'years', 'dogs', 'cats'];
   toaTuple = tuple(Listofitems)
   print ("Tuple elements : ", toaTuple)
```

Difference between lists and tuples

- Lists are mutable. Lists however have this method called append. In order for most of your appends to be fast, python will actually create a larger array in memory *just in case* you append.
- This way, when you do append, it does not have to recreate a list every time. You can add items to the list .How would it know that you don't want to maybe add a 4th 5th 6th element? To play safe, we assume you might want more in the memory
- On the other hand, by using tuples, it tells python that you want an immutable structure. Give me space for 3 things, fill those slots up, and move on.

- Since tuples are immutable, this means that tuples are fixed. We can't do anything to them in memory.
- Performance: processing of tuples is said to be faster than list processing
- Using tuples is safe: Since they are immutable, we cant change content of the tuple. This can be useful when you don’t want any data modified by your code