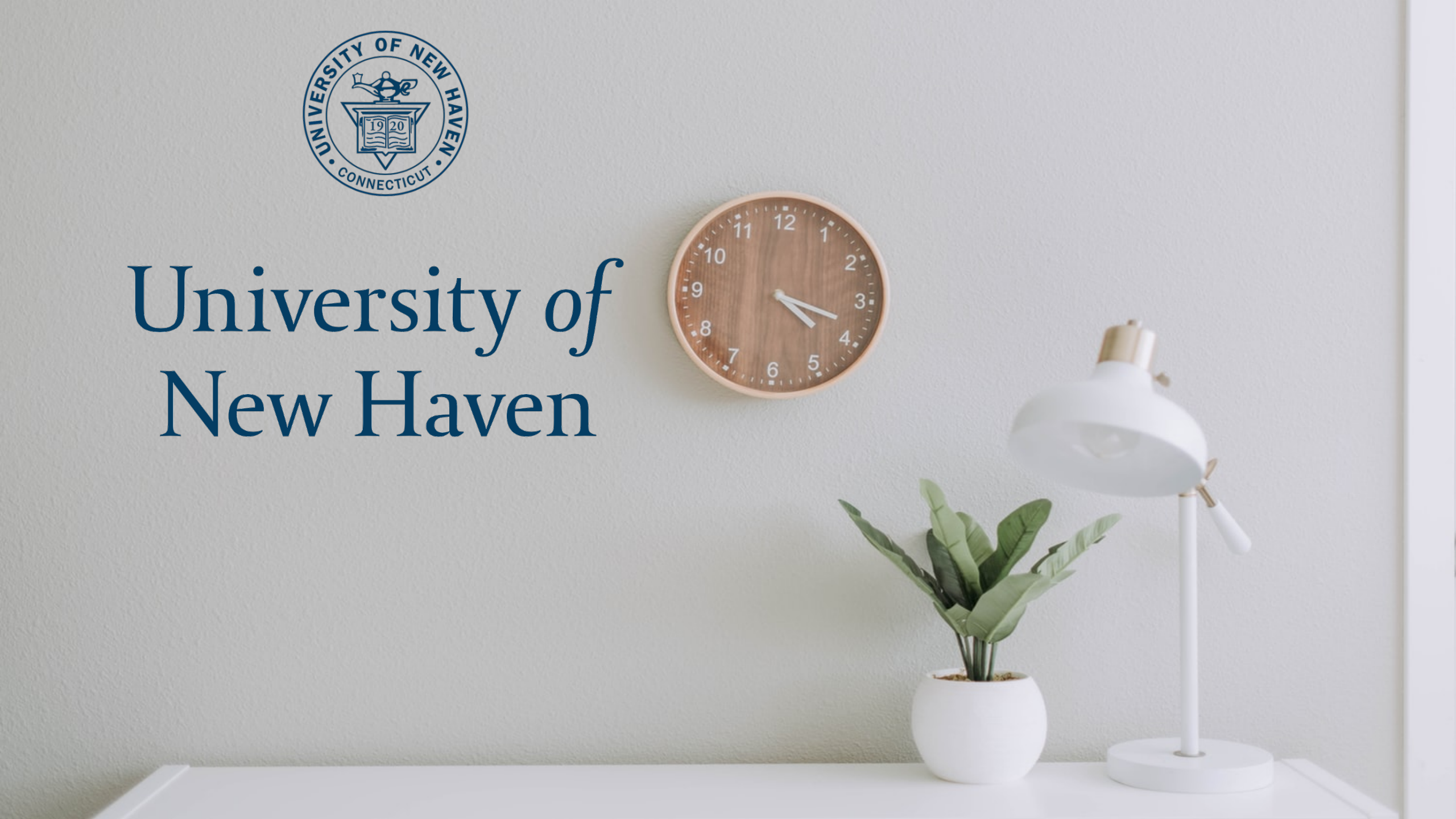
University of New Haven

University of New Haven

WELCOME
TO
Intro to Script Programming/Python

CSCI 6651-03 Spring 2022

Bibek Upadhayay

University of New Haven

University of New Haven

Functions and Sequences

Objectives

- **Functions**
  - Defining a function
  - Return
  - Default values for parameters
  - Functional programming support in Python – map, filter, reduce
  - Scope of variables and pitfalls
- **Sequences**
  - Tuples, lists and strings
  - Mutable and immutable sequences
  - Accessing elements

Defining Functions

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1

    line2
    return total_counter
```

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword ‘return’ indicates the value to be sent back to the caller.

Functions vs. methods

University of New Haven

Some operations are functions and others methods

- You just have to learn (and remember or lookup) which operations are functions and which are methods

len() is a function on collections that returns the number of things they contain

```
>>> len(['a', 'b',
  'c'])
3
>>> len(('a','b','c'))
3
>>> len("abc")
3
```

index() is a method on collections that returns the index of the 1st occurrence of its arg

```
>>> ['a','b','c'].index('a')
0
>>> ('a','b','c').index('b')
1
>>> "abc".index('c')
2
```

University of New Haven

Python and Types

Dynamic typing: Python determines the data types of variable bindings in a program automatically

Strong typing: But Python's not casual about types, it enforces the types of objects

For example, you can't just append an integer to a string, but must first convert it to a string

```
x = "the answer is "  # x bound to a string
y = 23                # y bound to an integer.
print x + y           # Python will complain!
```

Calling a Function

University of New Haven

The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

Functions without returns

University of New Haven

All functions in Python have a return value, even if no return line inside the code

Functions without a return, return the special value **None**

- None is a special constant in the language
- None is used like NULL, void, or nil in other languages
- **None is not equivalent to False**

Function overloading? No.

University of New Haven

There is no function overloading in Python

- Unlike C++, **a Python function is specified by its name alone**
- The number, order, names, or types of arguments cannot be used to distinguish between two functions with the same name
- Two different functions can't have the same name, even if they have different arguments (signatures).

What is Function Overloading?

University of New Haven

- OOP feature
- two or more functions can have the same name but different parameters.

Default Values for Arguments

University of New Haven

You can provide default values for a function's arguments

These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

Keyword Arguments

University of New Haven

Can call a function with some/all of its arguments out of order as long as you specify their names

```
>>> def foo(x,y,z): return(2*x,4*y,8*z)
>>> foo(2,3,4)
(4, 12, 32)
>>> foo(z=4, y=2, x=3)
(6, 8, 32)
>>> foo(-2, z=-4, y=-3)
(-4, -12, -32)
```

Can be combined with defaults, too

```
>>> def foo(x=1,y=2,z=3): return(2*x,4*y,8*z)
>>> foo()
(2, 8, 24)
>>> foo(z=100)
(2, 8, 800)
```

*args and **kwargs

University of New Haven

By convention the names are *args and **kwargs.

One would use *args when you're not sure how many arguments might be passed to your function, i.e. it allows you pass an arbitrary number of arguments to your function:

```
>>> def print_everything(*args):
    for count, thing in enumerate(args):
...         print( '{0}. {1}'.format(count,
thing))
...
>>> print_everything('apple', 'banana', 'cabbage')
```

*args and **kwargs

Similarly, **kwargs allows you to handle named arguments that you have not defined in advance:

```
>>> def table_things(**kwargs):
...     for name, value in kwargs.items():
...         print( '{0} = {1}'.format(name, value))
...
>>> table_things(apple = 'fruit', cabbage = 'vegetable')
```

*args and **kwargs

University of New Haven

One can use these along with named arguments too.

- The explicit arguments get values first and then everything else is passed to *args and **kwargs. The named arguments come first in the list. For example:

```
def table_things(titlestring, **kwargs)
```

You can also use both in the same function definition but *args must occur before **kwargs.

*args and **kwargs

University of New Haven

You can also use the * and ** syntax when calling a function. For example:

```
>>> def print_three_things(a, b, c):
...     print( 'a = {0}, b = {1}, c =
{2}'.format(a,b,c))
...
>>> mylist = ['aardvark', 'baboon', 'cat']
>>> print_three_things(*mylist)
a = aardvark, b = baboon, c = cat
```

In this case it takes the list (or tuple) of items and unpacks it. By this it matches them to the arguments in the function. Of course, you could have a * both in the function definition and in the function call.

Functions are first-class objects

Functions can be used as any other datatype, e.g.:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc.

```
>>> def square(x): return x*x

>>> def applier(q, x): return q(x)

>>> applier(square, 7)
49
```

Functional programming

University of New Haven

Python supports functional programming idioms

Built-ins for map, reduce, filter, etc.

University of New Haven

map

Applies a function to all the items in an input_list. Here is the blueprint:

```
map(function_to_apply, list_of_inputs)
```

Exercise – replace this code by using a map:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

http://book.pythontips.com/en/latest/map_filter.html

University of New Haven

Example: map

```
>>> def add1(x): return x+1
>>> map(add1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(lambda x: x+1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(+, [1,2,3,4],
[100,200,300,400])
map(+,[1,2,3,4],[100,200,300,400])
          ^
SyntaxError: invalid syntax
```

University of New Haven

filter

As the name suggests, filter creates a list of elements for which a function returns true. Here is a short and concise example:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0,
number_list))
print(less_than_zero)
```

University of New Haven

reduce

Reduce is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the product of a list of integers.

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
print(product)
>>> 24
```

Global versus Local Scopes

University of New Haven

- **Scopes**: Different areas of a program that are separate from each other
- Every function has its own scope
- Functions can't directly access each other's variables
- **But can exchange information through parameters and return values**

Global versus Local Variables

```
def func1():
        local_name1 = "Func1” #local variable
        print local_name1, global_name
        #can access global_name but not local_name2
def func2():
        local_name2 = "Func2"
        print local_name2, global_name
        #but can not access local_name1 here

global_name = "Global” #global variable
#can not access local_name1 & local_name2 here
func1()
func2()
```

Shadowing/Changing a Global Variable from Inside a Function

University of New Haven

```
def demo():
    global value1 #full access of global variable value1
    value1 = -value1
    value2 = -20 #a new variable with same name (shadow)
    print("Inside local scope:", value1, value2, value3)
value1 = 10
value2 = 20
value3 = 30
print("In the global scope:", value1, value2, value3)
demo() # value1 is changed; value2 and value3 not
print("Back in the global scope", value1, value2, value3)
```

Shadow: To hide a global variable inside a scope by creating a new local variable of the same name

Not a good idea to shadow a global variable

Python Data Types

Type	Classes
Text Type	str
Numeric types	int, float, complex
Sequence Types	list, tuple, range
Mapping Type	dict
Set Types	Set, frozenset
Boolean Type	bool
Binary Types	bytes, bytesarray, memoryview

University of New Haven

Sequence types

TUPLES, LISTS, AND STRINGS

University of New Haven

Sequence Types

Tuple: (‘john’, 32, [CMSC])

- A simple **immutable** ordered sequence of items
- Items can be of mixed types, including collection types

Strings: “John Smith”

- **Immutable**
- Conceptually very much like a tuple

List: [1, 2, ‘john’, (‘up’, ‘down’)]

- **Mutable** ordered sequence of items of mixed types

University of New Haven

Similar syntax

All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.

Key difference:

- Tuples and strings are immutable
- Lists are mutable

The operations shown in this section can be applied to all sequence types

- most examples will just show the operation performed on one

Sequence Types 1

Define tuples using parentheses and commas

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

Define lists are using square brackets and commas

```
>>> li = ["abc", 34, 4.34, 23]
```

Define strings using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."
```

Sequence Types 2

Access individual members of a tuple, list, or string using square bracket “array” notation
Note that all are 0 based…

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

University of New Haven

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
 >>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

University of New Haven

Slicing: return copy of a subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members.

Start copying at the first index, and stop copying before second.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

The ‘in’ Operator

Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

Be careful: the in keyword is also used in the syntax of for loops and list comprehensions

University of New Haven

The + Operator

The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

The * Operator

The * operator produces a new tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

University of New Haven

Lists are mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

We can change lists in place.

Name li still points to the same memory reference when we're done.

University of New Haven

Tuples are immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

The immutability of tuples means they're faster than lists.

University of New Haven

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a') # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The extend method vs +

+ creates a fresh list with a new memory ref

extend operates on list li in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Potentially confusing:

- **extend** takes a list as an argument.
- **append** takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

University of New Haven

Operations on Lists Only

Lists have many methos, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of 1st occurrence
1
>>> li.count('b')  # number of occurrences
2
>>> li.remove('b') # remove 1st occurrence
>>> li
  ['a', 'c', 'b']
```

University of New Haven

cont'd

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

https://docs.python.org/3/tutorial/datastructures.html

University of New Haven

Summary

Strings, lists, tuples, sets and dictionaries all deal with aggregates

Two big differences

- Lists and dictionaries are mutable
  - Unlike strings, tuples and sets
- Strings, lists and tuples are ordered
  - Unlike sets and dictionaries

University of New Haven

Remark

Any use of code from some source other than yourself has to be cited in the comments.

You should not use any aspects of the Python language beyond what we have discussed in class.

University of New Haven

Thank you!