

CS303E: Elements of Computers and Programming

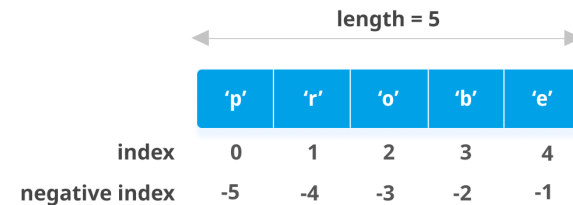
Lists

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: November 3, 2023 at 12:29

Lists

The `list` class is one of the most useful in Python.



Both strings and lists are sequence types in Python, so share many similar methods. Unlike strings, lists are *mutable*.

If you change a list, it doesn't create a new copy; *it changes the input list*.

Value of Lists

Suppose you have 30 different test grades to average. You could use 30 variables: `grade1`, `grade2`, ..., `grade30`. Or you could use one list with 30 elements: `grades[0]`, `grades[1]`, ..., `grades[29]`.

In file `AverageScores.py`:

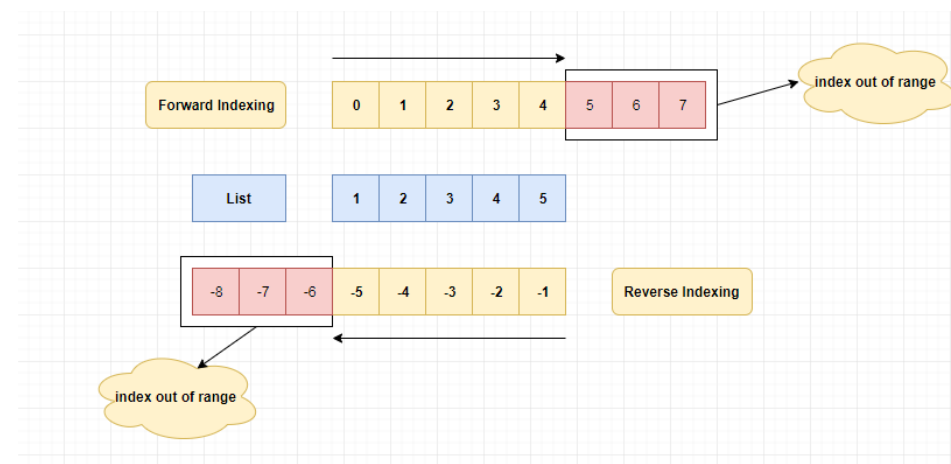
```
grades = [ 67, 82, 56, 84, 66, 77, 64, 64, 85, 67, \
          73, 63, 98, 74, 81, 67, 93, 77, 97, 65, \
          77, 91, 91, 74, 93, 56, 96, 90, 91, 99 ]

sum = 0
for score in grades:
    sum += score
average = sum / len(grades)
print("Class average:", format(average, ".2f"))
```

```
> python AverageScores.py
Class average: 78.60
```

Indexing and Slicing

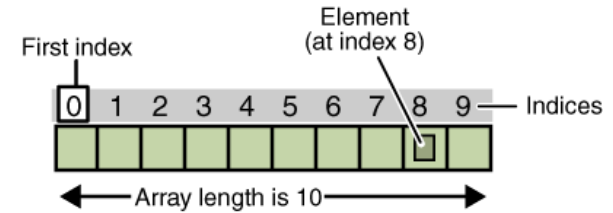
Indexing and slicing on lists are as for strings, including negative indexes.



Lists can be created with the `list` class constructor or using special syntax.

```
>>> list()           # create empty list, with constructor
[]
>>> list([1, 2, 3]) # create list [1, 2, 3]
[1, 2, 3]
>>> list(["red", 3, 2.5]) # create heterogeneous list
['red', 3, 2.5]
>>> ["red", 3, 2.5]   # create list, no explicit constructor
['red', 3, 2.5]
>>> range(4)         # not an actual list
range(0, 4)
>>> list(range(4))   # create list using range
[0, 1, 2, 3]
>>> list("abcd")     # create character list from string
['a', 'b', 'c', 'd']
```

Many programming languages have an **array** type. Python doesn't have native arrays (though some Python libraries add arrays).



Arrays are:

- homogeneous (all elements are of the same type)
- fixed size
- permit very fast access time

Python lists are:

- heterogeneous (can contain elements of different types)
- variable size
- permit fast access time

Like strings, lists are sequences and inherit various functions from sequences.

Function	Description
<code>x in s</code>	<code>x</code> is in sequence <code>s</code>
<code>x not in s</code>	<code>x</code> is not in sequence <code>s</code>
<code>s1 + s2</code>	concatenates two sequences
<code>s * n</code>	repeat sequence <code>s</code> <code>n</code> times
<code>s[i]</code>	<code>i</code> th element of sequence (0-based)
<code>s[i:j]</code>	slice of sequence <code>s</code> from <code>i</code> to <code>j-1</code>
<code>len(s)</code>	number of elements in <code>s</code>
<code>min(s)</code>	minimum element of <code>s</code>
<code>max(s)</code>	maximum element of <code>s</code>
<code>sum(s)</code>	sum of elements in <code>s</code>
for loop	traverse elements of sequence
<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	compares two sequences
<code>==</code> , <code>!=</code>	compares two sequences

```
>>> l1 = [1, 2, 3, 4, 5]
>>> len(l1)
5
>>> min(l1)           # assumes elements are comparable
1
>>> max(l1)           # assumes elements are comparable
5
>>> sum(l1)           # assumes summing makes sense
15
>>> l2 = [1, 2, "red"]
>>> sum(l2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> min(l2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'

>>>
```

Since lists are actual objects in class `list`, shouldn't `len`, `max`, etc. be *methods* instead of functions? Yes and no!

Remember from earlier that `len` is actually syntactic sugar for the method `__len__`.

```
>>> len([1, 2, 3])
3
>>> [1, 2, 3].__len__()
3
```

The others (`sum`, `max`, `min`) are actually functions defined on the class, for user convenience.

You just have to remember which operators are functions and which are methods.

We could rewrite `AverageScores.py` as follows:

```
grades = [ 67, 82, 56, 84, 66, 77, 64, 64, 85, 67, \
          73, 63, 98, 74, 81, 67, 93, 77, 97, 65, \
          77, 91, 91, 74, 93, 56, 96, 90, 91, 99 ]
average = sum(grades) / len(grades)
print("Class average:", format(average, ".2f"))
```

```
> python AverageScores.py
Class average: 78.60
```

Traversing Elements with a For Loop

General Form:

```
for u in list:
    body
```

In file `test.py`:

```
for u in range(3):           # not really a list
    print(u, end=" ")
print()

for u in [2, 3, 5, 7]:
    print(u, end=" ")
print()

for u in range(15, 1, -3):   # not really a list
    print(u, end=" ")
print()
```

```
> python test.py
0 1 2
2 3 5 7
15 12 9 6 3
```

Comparing Lists

Compare lists using the operators: `>`, `>=`, `<`, `<=`, `==`, `!=`. Uses *lexicographic* ordering: Compare the first elements of the two lists; if they match, compare the second elements, and so on. The elements must be of *comparable* classes.

```
>>> list1 = ["red", 3, "green"]
>>> list2 = ["red", 3, "grey"]
>>> list1 < list2
True
>>> list3 = ["red", 5, "green"]
>>> list3 > list1
True
>>> list4 = [5, "red", "green"]
>>> list3 < list4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and
'int'
>>> ["red", 5, "green"] == [5, "red", "green"]
False
```

BTW: the book's comparisons in 10.2.8 seem wrong.

List comprehension gives a compact syntax for building lists.

```
>>> range(4)                # not actually a list
range(0, 4)
>>> [ x for x in range(4) ]  # create list from range
[0, 1, 2, 3]
>>> [ x ** 2 for x in range(4) ]
[0, 1, 4, 9]
>>> lst = [ 2, 3, 5, 7, 11, 13 ]
>>> [ x ** 3 for x in lst ]
[8, 27, 125, 343, 1331, 2197]
>>> [ x for x in lst if x > 2 ]
[3, 5, 7, 11, 13]
>>> [s[0] for s in ["red", "green", "blue"] if s <= "green"]
['g', 'b']
>>> from IsPrime3 import *
>>> [ x for x in range(100) if isPrime(x) ]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
 59, 61, 67, 71, 73, 79, 83, 89, 97]
```



More List Methods

These are methods from class `list`. Since lists are mutable, these actually change `l`.

Function	Description
<code>l.append(x)</code>	add <code>x</code> to the end of <code>l</code>
<code>l.extend(l2)</code>	append elements of <code>l2</code> to <code>l</code>
<code>l.insert(i, x)</code>	insert <code>x</code> into <code>l</code> at position <code>i</code>
<code>l.pop()</code>	remove and return the last element of <code>l</code>
<code>l.pop(i)</code>	remove and return the <code>i</code> th element of <code>l</code>
<code>l.remove(x)</code>	remove the first occurrence of <code>x</code> from <code>l</code>
<code>l.reverse()</code>	reverse the elements of <code>l</code>
<code>l.sort()</code>	order the elements of <code>l</code>
<code>l.count(x)</code>	number of times <code>x</code> appears in <code>l</code>
<code>l.index(x)</code>	index of first occurrence of <code>x</code> in <code>l</code>

List Examples

```
>>> l1 = [1, 2, 3]
>>> l1.append(4)           # add 4 to the end of l1
>>> l1                     # note: changes l1
[1, 2, 3, 4]
>>> l1.count(4)           # count occurrences of 4 in l1
1
>>> l2 = [5, 6, 7]
>>> l1.extend(l2)         # add elements of l2 to l1
>>> l1
[1, 2, 3, 4, 5, 6, 7]
>>> l1.index(5)           # where does 5 occur in l1?
4
>>> l1.insert(0, 0)       # add 0 at the start of l1
>>> l1                     # note new value of l1
[0, 1, 2, 3, 4, 5, 6, 7]
>>> l1.insert(3, 'a')     # lists are heterogeneous
>>> l1
[0, 1, 2, 'a', 3, 4, 5, 6, 7]
>>> l1.remove('a')        # what goes in can come out
>>> l1
[0, 1, 2, 3, 4, 5, 6, 7]
```

```

>>> l1.pop()           # remove and return last element
7
>>> l1
[0, 1, 2, 3, 4, 5, 6]
>>> l1.reverse()      # reverse order of elements
>>> l1
[6, 5, 4, 3, 2, 1, 0]
>>> l1.sort()         # elements must be comparable
>>> l1
[0, 1, 2, 3, 4, 5, 6]
>>> l2 = [4, 1.3, "dog"]
>>> l2.sort()         # elements must be comparable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and
'float'
>>> l2.pop()         # put the dog out
'dog'
>>> l2
[4, 1.3]
>>> l2.sort()         # int and float are comparable
>>> l2
[1.3, 4]

```

A useful method on lists is `random.shuffle()` from the `random` module.

```

>>> import random
>>> list1 = [ x for x in range(9) ]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> random.shuffle( list1 )
>>> list1
[7, 4, 0, 8, 1, 6, 5, 2, 3]
>>> random.shuffle( list1 )
>>> list1
[4, 1, 5, 0, 7, 8, 3, 2, 6]
>>> random.shuffle( list1 )
>>> list1
[7, 5, 2, 6, 0, 4, 3, 1, 8]

```

Splitting a String into a List

Recall our `SplitFields` function from Slideset 8 to split up a comma separated value (csv) string. Python provides an easier approach with the `split` method on strings.

```

>>> str1 = "abc, def , ghi"
>>> str1.split(",")    # split on comma
['abc', ' def ', ' ghi'] # keeps whitespace
>>> str2 = " abc def ghi "
>>> str2.split()      # split on whitespace
['abc', 'def', 'ghi']
>>> str3 = "\tabc\ndef\r ghi\n"
>>> str3.split()      # split on whitespace
['abc', 'def', 'ghi']
>>> str4 = "abc / def / ghi"
>>> str4.split("/")   # split on slash
['abc ', ' def ', ' ghi']

```

Note `split` with no arguments splits on whitespace.

Processing CSV Lines

Suppose grades for a class were stored in a list of csv strings, such as:

```

studentData = ["Charlie,90,75",
               "Frank,8,77",
               "Susie,60,80"]

```

Here the fields are: Name, Midterm grade, Final Exam grade.

Compute the average for each student and print a nice table of results. *Remember that we solved a version of this problem in Slideset 3, where the data was entered by the user.*

```
def ProcessStudentData ( studentData ):
    """ Process list of csv student records. """
    # Print header line:
    print( "Name          MT    FN    Avg")
    print( "-----")

    for line in studentData:
        fields = line.split(',')
        if (len(fields) < 3):
            print( "Bad student record for ", fields[0] )
            continue
        else:
            name, midterm, final = fields[0].strip(), \
                                   int(fields[1].strip()), \
                                   int(fields[2].strip())

            avg = (midterm + final) / 2
            print( format(name, "10s"), \
                   format(midterm, "4d"), \
                   format(final, "4d"), \
                   format(avg, "7.2f") )
```

```
def main():
    studentData = ["Charlie,90,75",
                   "Frank,8,77",
                   "Johnnie,40",
                   "Susie,60,80"]

    ProcessStudentData( studentData )

main()
```

```
> python ExamExample2.py
Name          MT    FN    Avg
-----
Charlie        90    75    82.50
Frank          8     77    42.50
Bad student record for Johnnie
Susie          60    80    70.00
```

Copying Lists

Suppose you want to make a copy of a list. *The following won't work!*

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1 is lst2      # there's only one list here
True
>>> print(lst1)
[1, 2, 3, 4]
>>> print(lst2)
[1, 2, 3, 4]
>>> lst1.append(5)    # changes to lst1 also change lst2
>>> print(lst2)
[1, 2, 3, 4, 5]
```

But you can do the following:

```
>>> lst2 = [x for x in lst1]    # creates a new copy
```

Passing Lists to Functions

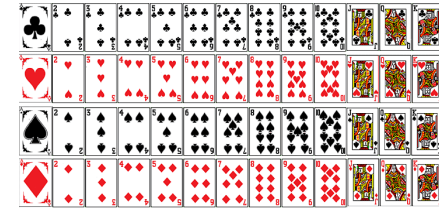
Like any other *mutable* object, when you pass a list to a function, you're really passing a reference (pointer) to the object in memory.

```
def alter( lst ):
    lst.pop()

def main():
    lst = [1, 2, 3, 4]
    print( "Before call: ", lst )
    alter( lst )
    print( "After call: ", lst )

main()
```

```
> python ListArg.py
Before call:  [1, 2, 3, 4]
After call:  [1, 2, 3]
```



In Slideset 7 we introduced the Card class. Let's now define a Deck of Cards. Remember we defined some functions: `isRank`, `isSuit`, `cardRankToIndex`, `cardIndexToRank`, etc.

It would be much easier to just add the following constant definitions to `Card.py`.

```
RANKS = ['Ace', '2', '3', '4', '5', '6', '7', '8', \
         '9', '10', 'Jack', 'Queen', 'King']
SUITS = ['Spades', 'Diamonds', 'Hearts', 'Clubs']
```

Think of how you'd redefine the functions listed above with those lists available.

```
RANKS = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', \
         '10', 'Jack', 'Queen', 'King']
SUITS = ['Spades', 'Diamonds', 'Hearts', 'Clubs']

def isRank( r ):
    return r in RANKS

def isSuit( s ):
    return s in SUITS

def cardRankToIndex( r ):
    return RANKS.index( r )

def cardSuitToIndex( s ):
    return SUITS.index( s )
```

A deck of cards "is" a list of Card objects, one for each combination of rank and suit.



Data: a list of Card objects, initially all possible combinations of rank and suit.

Methods:

- Print the deck in order.
- Shuffle the deck.
- Deal a card from deck.
- How many cards are left in the deck (after dealing)?

In file `Deck.py`:

```
import random
from Card import *

class Deck:
    """ Defines the Deck class. Each Deck contains
    a list of cards, one for each rank and suit """

    def __init__(self):
        """Return a new deck of cards."""
        self.__cards = []
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                c = Card(rank, suit)
                self.__cards.append(c)
```

Other things we might want to do with a deck are:

- 1 shuffle the deck
- 2 deal a card from the deck
- 3 ask how many cards are left in the deck
- 4 print the deck in order

Since the deck “is” a list, shuffling just means calling the `random.shuffle` function.

```
def shuffle(self):
    """Shuffle the cards."""
    random.shuffle(self.__cards)
```

Since lists are mutable, this shuffles *in place*, i.e., it doesn’t create a new deck.

Dealing a Card and Deck Length

Dealing a Card means removing the top card from the Deck and returning that card:

```
def deal(self):
    """Remove and return the top card, or None
    if the deck is empty."""
    if len(self) == 0:
        print("Deck is empty.")
        return None
    else:
        return self.__cards.pop(0)
```

Notice that we’re calling `len(self)` to check whether the Deck is empty. This only works if we define the `__len__` method for the class:

```
def __len__(self):
    """Returns the number of cards left in the deck."""
    return len(self.__cards)
```

Printing a Deck

Finally, we can use the `print` method for Deck class instances only if we’ve defined a `__str__` method to generate an appropriate string value:

```
def __str__(self):
    result = ""
    for c in self.__cards:
        # Here we ask each card how it
        # wants to be printed.
        result = result + str(c) + "\n"
    return result
```

Notice that `str(c)` only works because we defined the `__str__` method within class Card.


```
>>> from Deck import *
>>> d = Deck()           # create a new deck
>>> print( d )         # print, notice order
Ace of Spades
2 of Spades
...
Jack of Clubs
Queen of Clubs
King of Clubs

>>> d.shuffle()        # randomly shuffle deck
>>> print( d )
Queen of Spades
5 of Diamonds
4 of Clubs
...
Jack of Diamonds
8 of Clubs
```

```
>>> c1 = d.deal()      # deal top card
>>> print( c1 )
Queen of Spades
>>> c2 = d.deal()     # deal next card
>>> print( c2 )
5 of Diamonds
>>> len( c1 )         # didn't define len for Card
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'Card' has no len()
>>> len( d )         # deck now 50 cards
50
>>> d.__len__()      # len same as __len__
50
>>> d.__cards        # can't access private field
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Deck' object has no attribute '__cards'
```

Designing the Hand Class

Recall that our initial goal (from the Object slideset) was playing Poker. Now that we have Cards and Decks, we can define Hands; a poker hand is five cards.

Data: a list of five Card objects, dealt from a Deck object.

Methods:

- Print the hand in order.
- (Later) evaluate the hand as a poker hand.



The Hand Class

From file Hand.py:

```
import Card
from Deck import *

class Hand:
    """ Five cards dealt from a Deck object. """
    def __init__(self, deck):
        """ A hand is simply a list of 5 cards, dealt
            from the deck. """
        if ( len(deck) < 5 ):
            print ( "Not enough cards left!" )
            return None
        self.__cards = []
        for i in range(5):
            card = deck.deal()           # deal next card
            self.__cards.append(card)    # append to hand

    def __str__(self):
        result = ""
        for card in self.__cards:
            result = result + str(card) + "\n"
        return result
```

Finally, we allow looking at the cards in the Hand object:

```
def getCard( self, i ):
    """ Get the ith card from the hand, where
        i in [0..4]. """
    if (0 <= i <= 4):
        return self.__cards[i]
    else:
        return None
```

```
>>> from Hand import *
>>> h1 = Hand()           # can't deal without a deck
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument
: 'deck'
>>> d = Deck()           # so create a new deck
>>> d.shuffle()          # shuffle it
>>> print( d )
7 of Clubs
King of Diamonds
6 of Diamonds
Queen of Spades
8 of Clubs
Jack of Hearts
8 of Hearts
...
7 of Spades
10 of Clubs
```

```
>>> h1 = Hand( d )       # deal a hand from Deck d
>>> print( h1 )
7 of Clubs
King of Diamonds
6 of Diamonds
Queen of Spades
8 of Clubs

>>> h2 = Hand( d )       # deal another hand
>>> print( h2 )
Jack of Hearts
8 of Hearts
Jack of Clubs
9 of Clubs
8 of Diamonds

>>> len( d )
42

>>> len( h1 )           # we didn't define len on Hand
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'Hand' has no len()
```

It would be nice to be able to evaluate a hand as a poker hand, and perhaps compare two hands.

That would be a pretty good project!



Next stop: More on Lists.