

Lists/Tuples

Example

- Write a program to keep track of all students' scores on exam 1.
- Need a *list* of everyone's score
- Use 14 doubles
- What about next semester?

Lists

- Ordered set of values

```
scores=[90.5, 73, 87]
```

- Each value accessed by using [] and index

```
scores[1] #would be 73
```

- A list can contain different types -- including other lists

```
scores=[90.5, 73, 87, 'Mickey', [67, 89]]
```

- A list can be empty

```
scores=[]
```

Example

scores:	90.5	73	87
---------	------	----	----

`scores=[90.5, 73, 87]`

Subscripts

- Subscript describes which box of the array you are dealing with
- List of size N has subscripts
 - 0, 1, 2, ... (n-1)
 - list of size 3 – 0, 1, 2
- Subscript can be a variable or expression which produces an integer
 - `scores[i]`
 - `scores[i+5]`
- If subscript specified does not exist – runtime error

Range

- Lists of consecutive integers common
- Use `range()` to generate
 - `range(5)` #0, 1, 2, 3, 4
 - `range(1,5)` #1, 2, 3, 4
 - `range(1, 10, 2)` #1, 3, 5, 7, 9

Accessing List Elements

- Print all elements of the list *scores*
 - Use only one print statement

Accessing List Elements

- Print all elements of the list *scores*
 - Use only one print statement

```
scores = [90.5, 73, 82]
i=0
while(i<3):
    print scores[i]
    i = i+1
```

Accessing List Elements

- Using 3 as length is error prone

```
scores = [90.5, 73, 82]
i=0
while(i<len(scores)):
    print scores[i]
    i = i+1
```

Accessing List Elements

- Another method -- short cut

```
scores = [90.5, 73, 82]
for i in scores:
    print i
```

- Membership

```
scores[90.5, 73, 82]
if 100 in scores:
    print "There is a perfect score!"
```

Operators -- A lot like strings

- Concatenation

```
first_scores=[90,56,87]  
second_scores=[67,100,89]  
all_scores = first_scores+second_scores
```

- Slices

```
all_scores[2:4] #yields [87,67]  
more_scores=all_scores[:] #makes a copy of all_scores
```

Mutability

- Lists are mutable - they can be changed

```
scores[90.5, 73, 82]
```

```
scores[1] = 100 #scores is now [90.5, 100, 82]
```

- Lists can grow - using append function
 - append takes 1 argument

```
scores.append(12)
```

```
scores.append([56, 78, 34])
```

- Lists can shrink - using del function
 - del takes an element or slice

```
del scores[2]
```

```
del scores[1:3]
```

- Other functions - sort, reverse

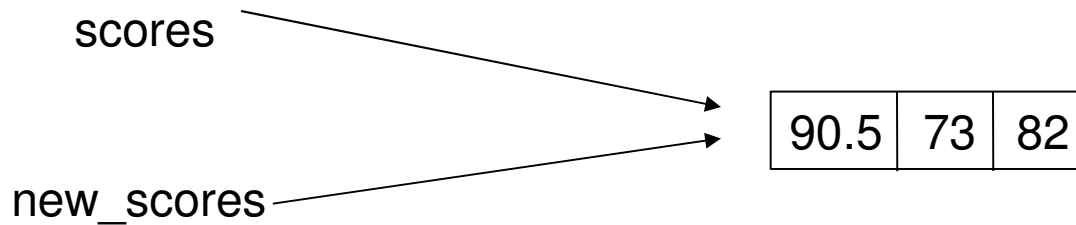
Aliasing

- Reference assignment

```
scores=[90.5, 73, 82]
```

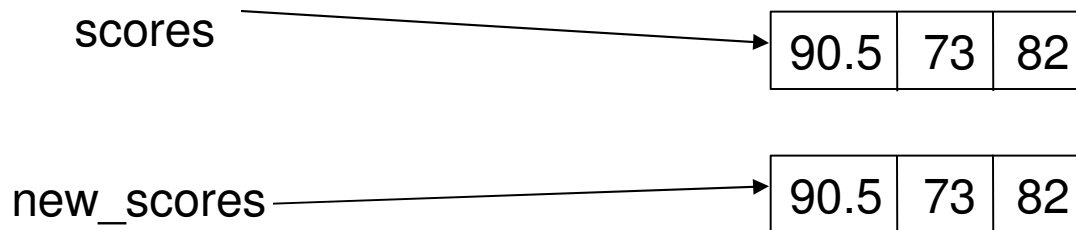
```
new_scores = scores
```

```
scores[0] = 100 #new_scores[0] changes
```



Cloning

```
scores=[90.5, 73, 82]  
new_scores = scores[:]  
scores[0] = 100 #new_scores[0] DOES NOT change
```



Exercises

1. Use the interpreter to help you determine the outcome of each of the following statements. Why are the invalid statements invalid?

```
list1 = [1,2,3]
list2 = list1
print list2
list2 = list1[2]
print list2
list2.append(8)
print list2
list2 = list1
list2.append(8)
print list2
list1[9] = "hello"
print list2
list1[2] = "hello"
print list2
```

List Elements as Parameters

- Write a function to add two elements of a list

List Elements as Parameters

- Write a function to add two elements of a list
 - How is the function called?

```
def adder(num1, num2):  
    return (num1 + num2)
```

List Elements as Parameters

- Write a function to add two elements of a list
 - How is the function called?

```
adder(scores[0], scores[1])
```

Passing Lists

- Would like to pass an entire list into a function
 - Sum all elements, prompt user for values

Example

```
sum(scores)
```

```
def sum(list):  
    sum = 0  
    for i in list:  
        sum += i  
    return sum
```

Nested Lists

```
exam_scores=[[90, 67, 34], [46, 99, 87], [89,78,67],[87,87,85]]  
#could represent three scores for four students
```

- Access second score of third student

Nested Lists

```
exam_scores=[[90, 67, 34], [46, 99, 87], [89,78,67],[87,87,85]]  
#could represent three scores for four students
```

- Access second score of third student

```
exam_scores[2][2]
```

	0	1	2
0	90	67	34
1	46	99	87
2	89	78	67
3	87	87	85

Tuples

- Very similar to lists, but immutable
 - You can copy, but you cannot modify (append, del, assign new values)

- Typically enclosed in ()

```
scores=(90.5, 73, 87)
```

- An integer - `scores = (70)`
- A tuple of one element - `scores = (70,)`
- Convert a list to a tuple

```
tuple_scores = tuple(list_scores)
```
- Convert a tuple to a list -

```
list_scores = list(tuple_scores)
```

Swapping

- Typical swap requires a temp variable

```
tmp = b
```

```
b = a
```

```
a = tmp
```

- Tuple assignment

```
a, b = b, a
```

```
x, y, z = 4, 6, 8
```

Return Values

```
def getnums() :  
    n1 = input("Enter number 1:")  
    n2 = input("Enter number 2:")  
    return n1, n2  
  
x, y = getnums()  
print x #prints first number entered  
print y #prints second number entered
```