

Compound data types

- Have seen a sampling of different classes of algorithms
 - Exhaustive enumeration
 - Guess and check
 - Bisection
 - Divide and conquer
- All have been applied so far to simple data types
 - Numbers
 - Strings

Compound data types

- Tuples
- Lists
- Dictionaries

Tuples

- Ordered sequence of elements (similar to strings)
- Elements can be more than just characters

```
t1 = (1, 'two', 3)  
print(t1)
```

```
t2 = (t1, 'four')  
print(t2)
```

Operations on tuples

```
t1 = (1, 'two', 3)
```

```
t2 = (t1, 'four')
```

- Concatenation
- Indexing
- Slicing

```
print(t1+t2)
```

```
print((t1+t2)[3])
```

```
print((t1+t2)[2:5])
```

- Singletons

```
t3 = ('five',)
```

```
print(t1+t2+t3)
```

Manipulating tuples


- Can iterate over tuples just as we can iterate over strings

```
def findDivisors(n1, n2):  
    """assumes n1 and n2 positive ints  
    returns tuple containing  
    common divisors of n1 and n2"""  
    divisors = () # the empty tuple  
    for i in range(1, min(n1, n2) + 1):  
        if n1%i == 0 and n2%i == 0:  
            divisors = divisors + (i,)   
    return divisors
```

Manipulating tuples

- Can iterate over tuples just as we can iterate over strings

```
divs = findDivisors(20, 100)
total = 0
for d in divs:
    total += d
print(total)
```



Lists

- Look a lot like tuples
 - Ordered sequence of values, each identified by an index
 - Use [1, 2, 3] rather than (1, 2, 3)
 - Singletons are now just [4] rather than (4,)
- **BIG DIFFERENCE**
 - Lists are mutable!!
 - While tuple, int, float, str are immutable
 - So lists can be modified after they are created!

Why should this matter?

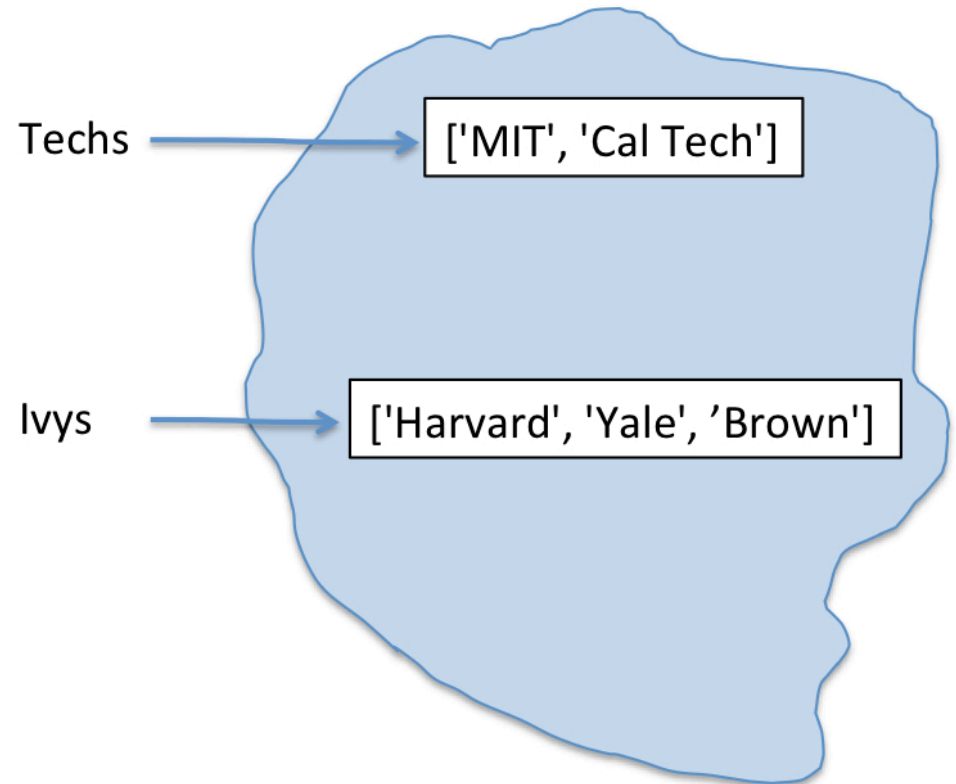
- Some data objects we want to treat as fixed
 - Can create new versions of them
 - Can bind variable names to them
 - But don't want to change them
 - Generally valuable when these data objects will be referenced frequently but elements don't change
- Some data objects may want to support modifications to elements, either for efficiency or because elements are prone to change
- Mutable structures are more prone to bugs in use, but provide great flexibility

Visualizing lists

```
Techs = [ 'MIT',  
          'Cal Tech' ]
```

```
Ivys = [ 'Harvard',  
         'Yale', 'Brown' ]
```

```
>>> Ivys[1]  
'Yale'
```



Structures of lists

- Consider

```
Univs = [Techs, Ivys]
```

```
Univs1 = [[ 'MIT', 'Cal Tech' ],  
           [ 'Harvard', 'Yale', 'Brown' ]]
```

- Are these the same thing?
 - They print the same thing
 - But let's try adding something to one of these

Mutability of lists

- Let's evaluate

```
Techs.append( 'RPI' )
```

- Append is a method (hence the .) that has a **side effect**
 - It doesn't create a new list, it mutates the existing one to add a new element to the end
- So if we print Univs and Univs1 we get different things

```
print(Univs)
```

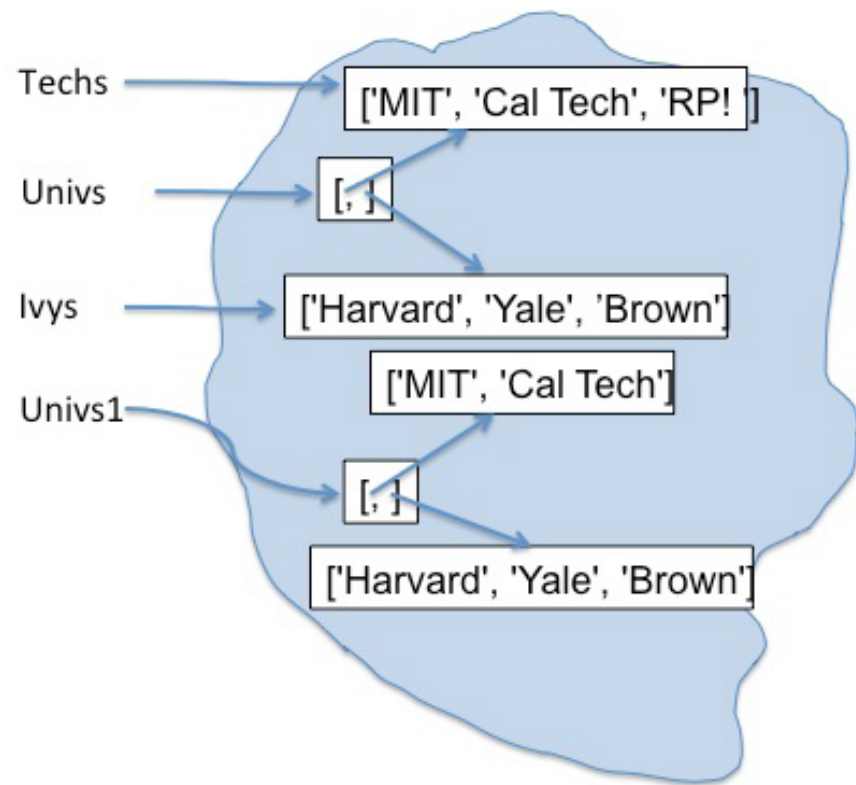
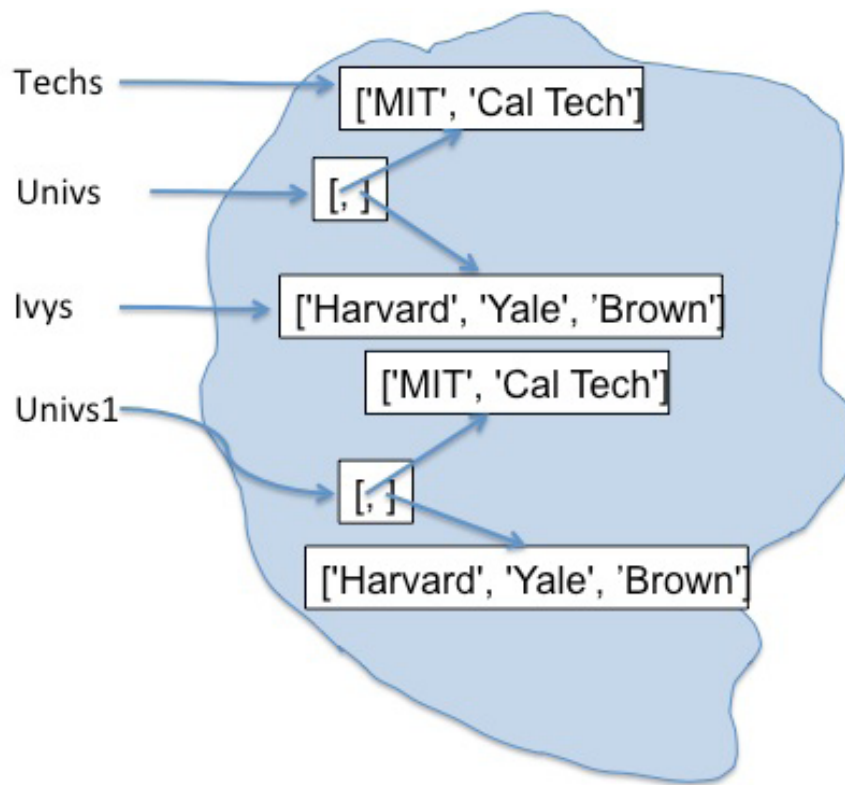
```
Univs = [['MIT', 'Cal Tech',  
         'RPI'], ['Harvard', 'Yale',  
         'Brown']]
```

```
Print(Univs1)
```

```
Univs1 = [['MIT', 'Cal Tech'],  
          ['Harvard', 'Yale', 'Brown']]
```

Why?

- Bindings before append
- Bindings after append



Observations

- Elements of `Univs` are not copies of the lists to which `Techs` and `Ivys` are bound, but are the lists themselves!
- This effect is called **aliasing**:
 - There are two distinct paths to a data object
 - One through the variable `Techs`
 - A second through the first element of list object to which `Univs` is bound
 - Can mutate object through either path, but effect will be visible through both
 - Convenient but **treacherous**

We can directly change elements

```
>>> Techs  
[ 'MIT', 'Cal Tech', 'RPI' ]
```

```
>>> Techs[2] = 'WPI'
```

Cannot do this with tuples!

```
>>> Techs  
[ 'MIT', 'Cal Tech', 'WPI' ]
```

Operations on lists

- Iteration

```
for e in Univs:
    print( 'Univs contains ' )
    print(e)
    print( '      which contains' )
    for u in e:
        print( '          ' + u)
```


Append versus flatten

```
Techs.append(Ivys)
```

Side Effect

Then Techs returns

```
['MIT', 'Cal Tech', 'RPI',  
 ['Harvard', 'Yale', 'Brown']]
```

```
flat = Techs + Ivys
```

Creates a new list

Then flat returns

```
['MIT', 'Cal Tech',  
 'RPI', 'Harvard', 'Yale', 'Brown']
```

In more detail

```
>>>Techs
[ 'MIT', 'Cal Tech',
  'RPI' ]
```

```
>>>Techs.append(Ivys)
```

```
>>>Techs
[ 'MIT', 'Cal Tech',
  'RPI', ['Harvard',
          'Yale', 'Brown' ] ]
```

```
>>>Techs
[ 'MIT', 'Cal Tech',
  'RPI' ]
```

```
>>>flat = Techs + Ivys
```

```
>>>flat
[ 'MIT', 'Cal Tech',
  'RPI', 'Harvard',
  'Yale', 'Brown' ]
```

```
>>>Techs
[ 'MIT', 'Cal Tech',
  'RPI' ]
```

Cloning

- Avoid mutating a list over which one is iterating
- Example:

`L1 = [1, 2, 3, 4]`

`L2 = [1, 2, 5, 6]`

`removeDups(L1, L2)`

```
def removeDups(L1, L2):  
    for e1 in L1:  
        if e1 in L2:  
            L1.remove(e1)
```

Then

`print(L1)`

returns

`[2, 3, 4]`

Why?

```
def removeDups(L1, L2):  
    for e1 in L1:  
        if e1 in L2:  
            L1.remove(e1)
```

- Inside for loop, Python keeps track of where it is in list using internal counter
- When we mutate a list, we change its length but Python doesn't update counter

Better is to clone

```
def removeDupsBetter(L1, L2):  
    L1Start = L1[:]  
    for e1 in L1Start:  
        if e1 in L2:  
            L1.remove(e1)
```

L1 = [1,2,3,4]

L2 = [1,2,5,6]

removeDupsBetter(L1,
L2)

Then

print(L1)

returns

[3, 4]

Note that using `L1Start = L1` is not sufficient

Functions as Objects


- Functions are **first class objects**:
 - They have types
 - They can be elements of data structures like lists
 - They can appear in expressions
 - As part of an assignment statement
 - As an argument to a function!!
- Particular useful to use functions as arguments when coupled with lists
 - Aka **higher order programming**

Example

```
def applyToEach(L, f):  
    """assumes L is a list, f a function  
        mutates L by replacing each element,  
        e, of L by f(e)"""  
    for i in range(len(L)):  
        L[i] = f(L[i])
```

Example

```
def applyToEach(L, f):          L = [1, -2, 3.4]
    for i in
        range(len(L)):
            L[i] = f(L[i])
```



```
applyToEach(L, abs)
print(L)
```

```
applyToEach(L, int)
print(L)
```

```
applyToEach(L, fact)
print(L)
```

```
applyToEach(L, fib)
print(L)
```


Example

```
def applyToEach(L, f):  
    for i in  
        range(len(L)):  
        L[i] = f(L[i])
```

L = [1, -2, 3.4]

```
applyToEach(L, abs)  
print(L)
```



[1, 2,
3.3999999999999999]

```
applyToEach(L, int)  
print(L)
```

```
applyToEach(L, fact)  
print(L)
```

```
applyToEach(L, fib)  
print(L)
```

Example

```
def applyToEach(L, f):  
    for i in  
        range(len(L)):  
        L[i] = f(L[i])
```

`L = [1, -2, 3.4]`

```
applyToEach(L, abs)  
print(L)
```

`[1, 2,
3.3999999999999999]`

```
applyToEach(L, int)  
print(L)
```



`[1, 2, 3]`

```
applyToEach(L, fact)  
print(L)
```

```
applyToEach(L, fib)  
print(L)
```

Example

```
def applyToEach(L, f):  
    for i in  
        range(len(L)):  
        L[i] = f(L[i])
```

```
applyToEach(L, abs)  
print(L)
```

```
applyToEach(L, int)  
print(L)
```

```
applyToEach(L, fact,  
print(L)
```

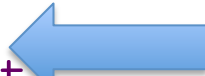
```
applyToEach(L, fib)  
print(L)
```

L = [1, -2, 3.4]

[1, 2,
3.3999999999999999]

[1, 2, 3]

[1, 2, 6]



Example

```
def applyToEach(L, f):  
    for i in  
        range(len(L)):  
        L[i] = f(L[i])
```

```
applyToEach(L, abs)  
print(L)
```

```
applyToEach(L, int)  
print(L)
```

```
applyToEach(L, fact)  
print(L)
```

```
applyToEach(L, fib)  
print(L)
```

L = [1, -2, 3.4]

[1, 2,
3.3999999999999999]

[1, 2, 3]

[1, 2, 6]

[1, 2, 13]



Lists of functions

```
def applyFuns(L, x):  
    for f in L:  
        print(f(x))
```

```
applyFuns([abs, int, fact, fib], 4)  
4  
4  
24  
5
```

Generalizations of higher order functions

- Python provides a general purpose HOP, `map`
- Simple form – a unary function and a collection of suitable arguments
 - `map(abs, [1, -2, 3, -4])`
 - `[1, 2, 3, 4]`
- General form – an n-ary function and n collections of arguments
 - `L1 = [1, 28, 36]`
 - `L2 = [2, 57, 9]`
 - `map(min, L1, L2)`
 - `[1, 28, 9]`

Dictionaries

- `Dict` is generalization of lists, but now indices don't have to be integers – can be values of any immutable type
- Refer to indices as **keys**, since arbitrary in form
- A `dict` is then a collection of `<key, value>` pairs
- Syntax
 - `monthNumbers = { 'Jan':1, 'Feb':2, 'Mar':3, 1:'Jan', 2:'Feb', 3:'Mar' }`

We access by using a key

```
monthNumbers =  
    { 'Jan':1, 'Feb':2,  
      'Mar':3, 1:'Jan',  
      2:'Feb', 3:'Mar' }
```

```
monthNumbers[ 'Jan' ]  
returns  
1
```

```
monthNumbers[ 1 ]  
returns  
'Jan'
```

Entries in a dict are unordered,
and can only be accessed by
a key, not an index

Operations on dicts

- Insertion

```
monthNumbers[ 'Apr' ] = 4
```

- Iteration

```
collect = []
```

```
for e in monthNumbers:  
    collect.append(e)
```

collect is now

```
[1, 2, 'Mar', 'Feb', 'Apr', 'Jan', 3]
```

Compare to

```
monthNumbers.keys()
```

Keys can be complex

```
myDict = {(1,2): 'twelve',  
          (1,3): 'thirteen'}
```

```
myDict[(1,2)]
```

returns

```
'twelve'
```

Note that keys must be immutable, so have to use a tuple, not a list

We will return to dicts and their methods later