MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Computational Thinking and Algorithms
159.172
Python Lists

Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin & Giovanni Moretti

Python Sequences

List examples:

```
>>> []
>>> [2, 4, 6, 8]
>>> [159172, 'John Smith', [85, 78, 91] ]
>>> ['one element']
```

Tuple examples:

THIS IS A TRAP
The list elements, but not which list can be changed.

```
>>> 1,2,3
>>> (1, 2, 3)
>>> (159172, 'John Smith', [85, 78, 91])
>>> ( )
>>> (42,)       <<<<<<< Comma is needed to indicate a tuple
>>> 42,
```

Indexing examples

```
>>> vowels = ['a,'e','i','o','u']
>>> vowels[0]
'a'

>>> vowels[-1]
'u'
>>> vowels[-3]
'i'
>>> ['a,'e','i','o','u'][1]
'e'
```

items in a sequence are numbered from zero

i.e. the first item in a sequence has index zero (not 1)

last item has index of -1

Indexing examples

```
def mylistfun(x):
    return [1, 2, 3, x*2]
```

```
>>> q = mylistfun(9)
>>> print q[0]
1
>>> mylistfun(9) [3]
18
```

Thing returned by the function is a list

we can subscript directly (without using assignment)

Slicing

Use indexing to access individual elements in a sequence,
Use slicing to access ranges of elements

Slice examples:

```
>>> vowels = ['a’, 'e', 'i', 'o', 'u']
>>> vowels[0:2]
['a', 'e']

>>> vowels[-3:-1]
['i', 'o']

>>>vowels[-3:-4]        # Step size is +1, towards end
[]
```

Slicing – using a step length

More slice examples, using a third parameter, the step length :

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
>>> numbers[::4]
[1, 5, 9]

>>> numbers[5::-2]      #If step is –ve, swap start/end
[6, 4, 2]
>>> numbers[:5:-2]      #If step is –ve, start from end
[10, 8]
>>> numbers[8:2:-1]
[9, 8, 7, 6, 5, 4]
>>> numbers[0:10:-2]
[]
```

Concatenation

We can concatenate two sequences together,
as long as they are sequences of the same type, say lists,
using the + operator.

Concatenation examples:

```
>>> 'my ' + 'big ' + 'fat ' + 'wedding'
'my big fat wedding'

>>> [0, 1, 2, 3] + [4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]

>>>['a', 'b', 'c'] + [1, 2, 3]
['a', 'b', 'c', 1, 2, 3]
```

Multiplication

Multiplying a sequence by a number **n**,
gives a new sequence, the original repeated **n** times.

Multiplication examples:

```
>>> 'name ' * 5
'name name name name name '
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]
```

Membership

The **in** operator returns a Boolean value, True or False.

```
>>> expression = '$$$ Get rich quick!! $$$'
>>> '$$$' in expression
True

>>> numbers = [1, 2, 3, 4, 5]
>>> [4, 5] in numbers
False

>>> 4 in numbers
True
>>> moreNumbers = [1, 2, 3, [4, 5]]
>>> [4,5] in moreNumbers
True
```

The **in** operator is "overloaded" for strings

len( ), max( ), min( )

```
>>> len([1, 1, 1, 1, 1, 1])
6

>>> max([1, 2, 3])
3

>>> min(['a', 'b', ’c'])
'a'

>>> max( [1, 'John', [2,3,4] ] )
????
```

be careful to use max and min only on sequences where it makes sense to compare elements for relative size, **otherwise results are arbitrary**

polymorphism vs overloading

In this context:

Polymorphism

polymorphic function does the same job on different types

eg. **`len()`** works the same for all sequences:

tuples, lists, strings

Overloading

overloaded function or operator does a different, but related, job on different types

eg. **`1 + 3`** **`‘play’ + ‘ing’`**

`You will learn more about Polymorphism and Overloading next year in 159.272!`

List operations

Functions & operations so far work for all Python sequences.
Since some sequences types are immutable,
these functions **cannot change** the original sequence.

You can turn an immutable sequence into a list,
which is mutable.

```
>>> list((1, 2, 3))
[1, 2, 3]
>>> list('john')
['j', 'o', 'h', 'n']
```

Item assignment

Assign to a specific, existing position in the list
changes the item at the index location, rest of the list unchanged.

```
>>> x = [1, 1, 1]

>>> x[1] = 2
>>> x[1, 2, 1]

>>> x[3] = 2
Traceback (most recent call last):  File "", line 1, in
IndexError: list assignment index out of range
```

Why?

del - Item deletion

Deletes the item at a specific, existing position in the list
rest of the items remain in order, but indices changed.

```
>>> x = [1, 2, 1]
>>> del x[1]                    # NOT del(x[1])
>>> x
>>> x[1, 1]

>>> del x[4]
Traceback (most recent call last):
  File "", line 1, in
IndexError: list assignment index out of range

>>> x[0:2]
???
```

Slice assignments

Alter multiple items in a list using a slice assignment.

Can replace a slice with a sequence of different length,

so can use a slice assignment to insert or delete elements.

```
>>> name = list('Pearl')
>>> name
['P', 'e', 'a', 'r', 'l']
>>> name[2:4]
['a', 'r']
>>> name[2:] = list(‘xy')
>>> name
['P', 'e', ‘x', ‘y']
```

Slice assignments

Alter multiple items in a list using a slice assignment.

Can replace a slice with a sequence of different length,

so can use a slice assignment to insert or delete elements.

```
>>> name[2:3] = list('ppe')
['P', 'e', 'p', 'p', 'e', 'r’]

>>> name[2:]= []
['P', 'e’]

>>> name[1:1] = list('opey')
[‘P', 'o', 'p', 'e', 'y', 'e']
```

Slice assignment with varying stepsize

If you're not replacing one contiguous sequence with another, ensure that the new has the same number of elements as the old, else not clear which bits of the new sequence replace which bits of the old.

```
>>> mylist = [1,2,3,4,5,6,7,8,9,10]
>>> mylist[0:10:2] = ['a', 'b', 'c', 'd', 'e']
>>> mylist
['a', 2, 'b', 4, 'c', 6, 'd', 8, 'e', 10]

>>> mylist[9:0:-3] = [22, 22, 22]
>>> mylist
['a', 2, 'b', 22, 'c', 6, 22, 8, 'e', 22]
```

Slice assignment with varying stepsize

If you're not replacing one a contiguous sequence with another,
ensure that the new has the same number of elements as the old,
else not clear which item of the new sequence replace which of the old.

```
>>> mylist = ['a', 2, 'b', 4, 'c', 6, 'd', 8, 'e', 10]
>>> mylist[9:0:-3] = [22, 22, 22, 22]  # ONLY 3 items
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: attempt to assign sequence of size 4 to extended slice
   of size 3
```

Leaving out an index defaults to end of the list

```
>>> mylist[9::-3] = [22, 22, 22, 22]
[22, 2, 'b', 22, 'c', 6, 22, 8, 'e', 22]
```

append

Append a single item onto the end of a list.

```
>>> lst
['my', 'big', 'fat']

>>> lst.append('bottom')
>>> lst
['my', 'big', 'fat', 'bottom']

>>> '@'.join(lst)
'my@big@fat@bottom'
```

`join()` function takes a string and a list of strings as arguments

Turns the list into a single string, 1stargument is separator string..

insert

Insert an object into a list at a given index location, moving existing elements

`insert` takes two arguments, index and object.

```
>>> newlst
['my ', 'big ’, 'wedding ', 'next ', 'week ']

>>> newlst.insert(2, 'fat ')
>>> newlst
['my ', 'big ', 'fat ’, 'wedding ', 'next ', 'week
 ']
```

remove

Remove the **first** occurrence of a given value in list.

```
>>> newlst
['my ', 'big ', 'greek ', 'wedding ', 'day']

>>> newlst.insert(2, 'fat ')
>>> newlst
['my ', 'big ', 'fat ', 'greek ', 'wedding ', 'day']

>>> newlst.remove('greek ')
>>> newlst
['my ', 'big ', 'fat ', 'wedding ', 'day']
```

You'll get an exception (error) if you try to remove a value that is not in the list

.pop()

Removes an item from a list, **by default, the last one.**

Given an index argument, will remove the item at that position.

Both modifies the list *and* returns a value. All other list methods return **`None`**.

```
>>> newlst
['my ', 'big ', 'fat ', 'wedding ', 'next ', 'week ']
>>> newlst.pop()
'week '
>>> newlst
['my ', 'big ', 'fat ', 'wedding ', 'next ']

>>> newlst.pop(0)
'my '
>>> newlst
['big ', 'fat ', 'wedding ', 'next ']
```

Question: how to remove the second last item from any list?

reverse

Reverses the elements in an existing list.

```
>>> x = [1, 2, 3, 4]
>>> x.reverse()
>>> x
[4, 3, 2, 1]

>>> x.reverse()
>>> x
[1, 2, 3, 4]
```

`reversed(x)` **`returns a new list`** `with elements reversed`

index

Returns the index of the **first** occurrence of a given value.

```
>>> newlst
['big ', 'fat ', 'wedding ']

>>> newlst.index('fat ')
1

>>> newlst.index('greek ')
Traceback (most recent call last):
  File "", line 1, in
ValueError: 'greek ' is not in list
```

count

Returns the number of occurrences of a given value in the list.

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

>>> x.count(1)
3

>>> x.count('my ')
0
```

List comprehensions

Based on mathematical set comprehension.

Set comprehension: { x*x | x < 10 and (x mod 2 = 0) }

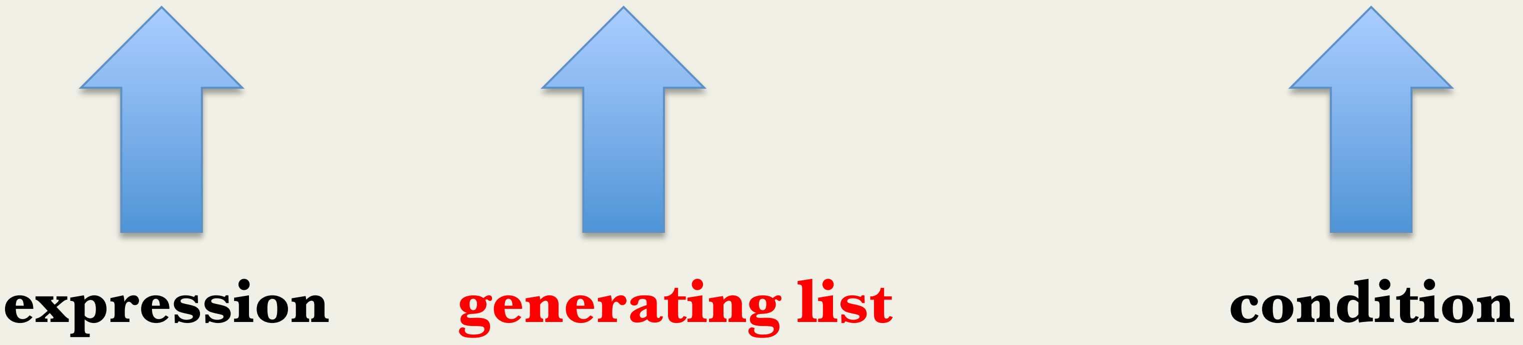
List comprehension: **[x*x for x range(10) if x % 2 == 0]**

A list comprehension consists of three parts

1. an expression,
2. a generating list
3. a condition.

List comprehensions

```
[ x*x  for x in range(10) if x % 2 ==0]
```



List comprehensions can be nested.

We can add more conditions.

We can add more generating lists,

let the expression take arguments from each of them.

List comprehensions

```
>>> [ x  for x in range(1,100)  if x % 19 == 0]
[19, 38, 57, 76, 95]
```

equivalent to:

```
s = [ ]
for x in range(1,100):
    if x % 19 == 0:
        s.append(x)
```

```
>>> [ x*x*x for x in range(1,100) if x % 19 == 0]
[6859, 54872, 185193, 438976, 857375]
```

List comprehensions

```
>>> [x+2*y for x in range(4) for y in range(3)]
[0, 2, 4, 1, 3, 5, 2, 4, 6, 3, 5, 7]
```

More generating lists gives us the equivalent of nested for loops. First generating list is iterated through in the outermost loop, the last in the innermost loop.

```
result = []
for x in range(4):
    for y in range(3):
        result.append(x+2*y)
```

Multiple List comprehensions

```
>>> [x+y for x in [1,2,3,4] for y in [100, 200, 300]]
```

```
[101, 201, 301, 102, 202, 302, 103, 203, 303, 104, 204, 304]
```

More generating lists are equivalent to nested for loops.

The later (inner) loop goes through all of its values for before outer loop changes to next value

```
result = [ ]
for x in [1,2,3,4]:
      for y in [100, 200, 300]:
          result.append(x+y)
  print(result)
```

map( )

Passes all elements of a sequence through a given function.

Equivalent to a list comprehension with a single generating list, where the expression is the function applied to a list item.

```
>>> def even(x):
        return x % 2 == 0

>>> list(map(even,  [0,1,2,3,4] ))
[True, False, True, False, True]

>>> list(even(x)   for x in [0,1,2,3,4] )
[True, False, True, False, True]
```

filter( )

Filter out items based on a Boolean function.

Equivalent to a list comprehension,

where the condition is the Boolean function applied to a list item.

```
>>> def negative(x):
        return x < 0

>>> filter(negative, [-1, 2, -3, 5, 7, -9])
[-1, -3, -9]

>>> [x for x in [-1, 2, -3, 5, 7, -9] if negative(x)]
[-1, -3, -9]
```

zip function

You can zip sequences together, returning a list of tuples.
The zip function works with as many sequences as you like.
If the sequences are of different lengths,
    it stops when the shortest is used up.

```
>>> mylist  = [ 1,  2,  3,  4]
>>> letters = ['a','b','c','d']

>>> zip(mylist, letters)
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

>>> zip(range(5), range(1000))
[(0,0), (1,1), (2,2), (3,3), 4,4)]
```

Using Lists

A simple example:

```
def tallest(classlist):
    (tallest, name) = classlist[0]
    for (height, person) in classlist:
        if height > tallest:
            tallest = height
            name = person
    return(tallest, name)
```

How much "work" does this algorithm do for a classlist of size n?

Removing duplicates

```
Creating a new list that contain only new items.

def removedups(mylist):
    newlist = [ ]
    for x in mylist:
        if x not in newlist:
            newlist.append(x)
    return newlist

print (“old list”, mylist)

print (“new list”, removedups(mylist))
```

Removing duplicates

```
Checking if there is more than one copy of each
   item.

def removedups1(mylist):
    for x in mylist:
        if mylist.count(x) > 1:
           indx = mylist.index(x)
           del mylist[indx]
    return mylist

print (“old list”, mylist)

print (“new list”, removedups(mylist))
```

Removing duplicates

```
>>> mylist = [1,2,3,3,4,4,5,5,6,7,8,8,8]

>>> print removedups(mylist)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> print mylist
[1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 8, 8, 8]

>>> print removedups1(mylist)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> print mylist
[1, 2, 3, 4, 5, 6, 7, 8]
```

Summing a list

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

or use the built-in Python function

```
>>> t = [1,2,3]
>>> sum(t)
6
```

Summing a list of lists

```
def nested_sum(t):
    total = 0
    for x in t:
        total += sum(x)
    return total

>>> t = [[1,2,3], [4,5,6], [7,8,9]]
>>> nested_sum(t)
45
```

Minimum value of a list

```
def min_val(t):
    min_so-far = t[0]
    for x in t:
        if x < min_so-far:
            min_so_far = x
    return min_so_far

>>> t = [1,2,3,4,5]
>>> min_val(t)
1
```

Minimum value of a list of lists

```
def nested_min(t):
    min_so_far = min_val(t[0])
    for x in t:
        if min_val(x) < min_so_far:
            min_so_far = min_val(x)
    return min_so_far
```

An operation that combines a sequence into a single value sometimes called **reduce**.

Pascal's triangle

```
        1
      1   1
    1   2   1
  ?   2   ?   1
?   ?   ?   ?   ?
```

Pascal's triangle

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

Define a function **`pascal(n)`** that takes an integer **`n`** and returns a list of lists that form the first **`n`** rows of Pascal's triangle.

Pascal's triangle

```
def pascal(n):
   if n == 1:
      return [[1]]
   else:
     result = [[1]]
     x = 1
     while x < n:
       lastRow = result[-1]
       nextRow = [ a+b
              for a,b in zip([0]+lastRow,lastRow+[0])]
       result.append(nextRow)
       x += 1
   return result
```