

Python Workshop Series Session 3: *Iteration and Lists*

Nick Featherstone
Research Computing

Slides: https://github.com/ResearchComputing/Python_Spring_2018



Research Computing
UNIVERSITY OF COLORADO **BOULDER**

Be Boulder.

Outline

- Lists
- Tuples & Dictionaries
- Loops



Memory: ID Know (Really)

- The *id* function returns the “identity” of an object in Python.
- As we will soon see, everything in Python is an object...
- In many Python *implementations*, *id* returns object memory address.
- Different groups make Python interpreters
- They are free to choose how they *implement* those things not defined in the Python standard.
- In the meantime, try this:

```
a = 1 ; b = 2  
print( id(b) – id(a) )
```

- Semicolons = multiple statements per line
- Anyone not get 32?
- Hmm... we'll come back to this



Lists in Python

- Multiple values can be grouped together into lists
- Just use square brackets []
- Values can have different datatypes.
- Indexed starting with 0
- Values copied as necessary ...
- Try this ...

```
a = 1.0
b = [ 1, 2, a, 4 ]
print( b[0] )
print( b[2] )
print( b )
```

```
print ( '1', b[2] is a , id(b[2]), id(a))
a = 2
print ( '2', b[2] is a , id(b[2]), id(a))
c = 1.0
print ( 'c', b[2] is c , id(b[2]), id(c))
```



All in one python script or Jupyter cell!



Nested (Multi-dimensional) Lists

- We can have lists of lists:
- Indexing uses two square brackets

```
a = [ 1 , 2]  
b = [ 3 , 4]  
c = [ a , b , 5 ]
```

```
print( c[ 0 ] )      [1, 2]  
print( c[ 1 ] )      [3, 4]  
print( c[ 0 ][ 0 ] )  1  
print( c[ 0 ][ 1 ] )  2  
print( c[ 1 ][ 0 ] )  3  
print( c[ 1 ][ 1 ] )  4  
print( c[ 2 ] )       5
```

- `c[2]` is a scalar
- `c[0]` and `c[1]` are 2-element lists.



Nested Lists: Memory

- Be careful!
- Python **does not** automatically copy a and b...

```
a = [ 1 , 2 ]  
b = [ 3 , 4 ]  
c = [ a , b ]
```

```
print( a[ 0 ] , c[ 0 ][ 0 ] )  
print( id(a[0]) , id(c[0][0]) )  
c[ 0 ][ 0 ] = 4  
print( a[ 0 ] , c[ 0 ][ 0 ] )  
print( id(a[0]) , id(c[0][0]) )
```



Cloning Lists

- If we want copies, use the “slice” notation → : ←

```
a = [ 1, 2]
b = a
b[0] = 5.0
print(a[0], b[0])
```

```
a = [ 1, 2]
b = a[ : ]
b[0] = 5.0
print( a[0], b[0] )
```

```
a = [ 1, 2]
b = [ 3, 4]
c = [ a[ : ], b[ : ] ]
```

```
print( a[ 0 ], c[ 0 ][ 0 ] )
c[ 0 ][ 0 ] = 4
print( a[ 0 ], c[ 0 ][ 0 ] )
```



Sublists

- We can copy a portion of a list using the slice notation with bounds

```
a = [ 1 , 2 , 3 , 4 , 5 ]  
b = a[ 2 : 4 ]  
print( len( b ) )  
print( b )
```

len function:
returns number of
elements in a list

Slicing Convention:

- b is **essentially** a copy of [a[2] , a[3]]
- *b is not a copy of [a[2] , a[3] , a[4]]*



Essentially?

```
a = [ 1 , 2 , 3 , 4 , 5 ]  
b = a[ 2 : 4 ]  
print( id(a[2]) , id(b[0]) )  
b[0] = 85  
print( id(a[2]) , id(b[0]) )
```



... well, more or less.



Lists and functions

- Lists are passed by reference.
- Avoid unwanted side-effects by passing list clones instead

```
def modify( a ):
    a[ 0 ] = 2
```

Side Effect

```
b = [ 0 , 0 ]
modify( b )
print( b )
```

No Side Effect

```
b = [ 0 , 0 ]
modify( b[ : ] )
print( b )
```



append & del

- The **append** method grows a list
- Syntax: *listname* **dot** *append*()
- The **del** statement deletes elements or sublists

```
a = [ ] init empty list  
a.append( 1 )  
print( len( a ) , a )  
a.append( 4 )  
print( len( a ) , a )  
a.append( 8 )  
print( len( a ) , a )
```

```
a = [ 4 , 8 , 12 , 13 ]  
print( a )  
del a[ 0 : 2 ]  
print( a )  
del a[ 0 ]  
print( a )
```



List Initialization: Replication

- Occasionally useful to initialize a list with known value
- Use the * operator to replicate values from an existing list or list expression

1-dimensional list

```
a = [ 1 , 2 ]  
b = 3 * a  
print( b )
```

b is [1, 2, 1, 2, 1, 2]

Nested list

```
a = [ 1 , 2 ]  
b = 3*[ a ]  
print( b )
```

b is [[1, 2], [1, 2], [1, 2]]



List Initialization: Replication



- Extends naturally to higher dimensions

```
a = [ [ 1 , 2 ] , [ 3 , 4 ] ]  
b = 2 * a  
print( b )
```

b is [[1, 2], [3, 4], [1, 2], [3, 4]]

```
a = [ [ 1,2], [ 3, 4] ]  
c = 2*[a]  
print(c)
```

c is [[[1, 2], [3, 4]], [[1, 2], [3, 4]]]

c[0][0][1]  c[1][1][0] 



List-like Class: Tuples

- Similar to Lists, but immutable (can't change elements)
- Use () instead of [] during creation (**only**)

```
a = ( 1 , 2 )  
print( a[ 0 ] )  
a[ 0 ] = 2 not allowed
```

lists of tuples

```
a = ( 1 , 2 )  
b = ( 3 , 4 )  
c = [ a , b ]  
c[ 0 ] = 1      OK – replace tuple with int  
c[ 1 ][ 0 ] = 2 not OK – modifying tuple element
```



Tuples of lists

- A bit non-intuitive

```
a = [ 1 , 2 ]  
b = [ 2 , 3 ]  
c = ( a , b )
```

Allowed; modifying list element

```
c[0][0] = 4
```

Not allowed; modifying tuple element

```
c[0] = 2
```



Tuple Assignment

- Useful Python feature
- Values on right assigned to values on left

Create a,b,c and assign them values

```
( a, b, c ) = ( 1, 2, 3 )
```

Swap values

```
tmp = a  
a = b  
b = tmp
```



```
( a , b ) = ( b , a )
```

Question:

How do the object id's behave?



List-like Objects: Dictionaries

- Key-value pairs
- Key (i.e., the index) must be immutable (ints or strings)
- Initialize with { } (not [] or ())

```
var = { }  
var['Apple'] = 43  
var[8] = [ 'Orange', 2, 14.0]
```

```
print (var[ 'Apple' ]) 43
```

```
print( var[ 8 ] ) [ 'Orange', 2, 14.0]
```

```
print( var[ 8 ][ 2 ] ) 14.0
```



Lists: odds and ends

- Concatenation:

```
a = [ 1 , 2 ]  
b = [ 2 , 3 ]  
c = a + b  
print(c)
```

- Membership:

```
mylist = [ 'Mario' , 'Luigi' ]  
b = 'Mario' in mylist  
c = 'Zelda' in mylist  
print(b,c)
```



Lists: Final Remarks

- See online text, chapter 11 for more on lists.
- Strings act like lists
 - Immutable
 - For useful string methods see online text, ch. 8

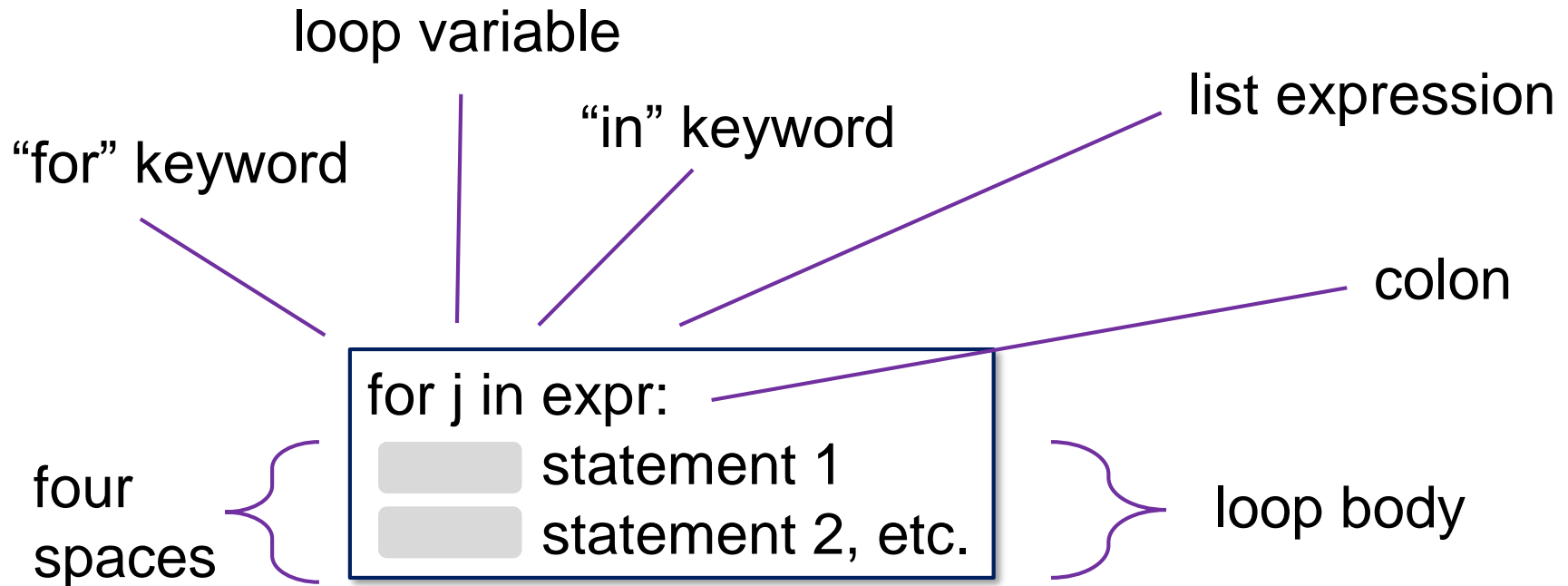


Iteration in Python

- Three commonly used loop constructs:
 - for
 - while
 - enumerate



For Loop Syntax



For each element in expr:

- Assign its value to j
- Execute statements in loop body



For Loop Examples

- Try these:

```
a = [1,2,3]
for j in a:
    print(j)
```

```
a = (1,2,3)
for j in a:
    print(j)
```

```
a = [ [1,2] , [3,4] ]
for j in a:
    print(j)
```

```
a = ['Peter', 'Paul', 'Mary']
for j in range(3):
    print( j )
    print( a[ j ] )
```

range function

- range(n)
 - Integer sequence 0 through n-1
- range(m,n)
 - Integer sequence m through n-1



Nesting Loops

- Just indent to begin a nested loop
- Try this

```
a = [ [ 1 , 2 , 0 ] , [ 3 , 4 , 7 ] ]  
alen = len(a)  
for j in range(alen):  
    jlen = len(a[ j ])  
    for k in range(jlen):  
        print( j , k , ' : ' , a[ j ][ k ] )
```



Exercise 1: For Loops

- Write a function that:
 - Accepts a list of numbers
 - Returns the sum of those numbers
- Be sure to use a for loop

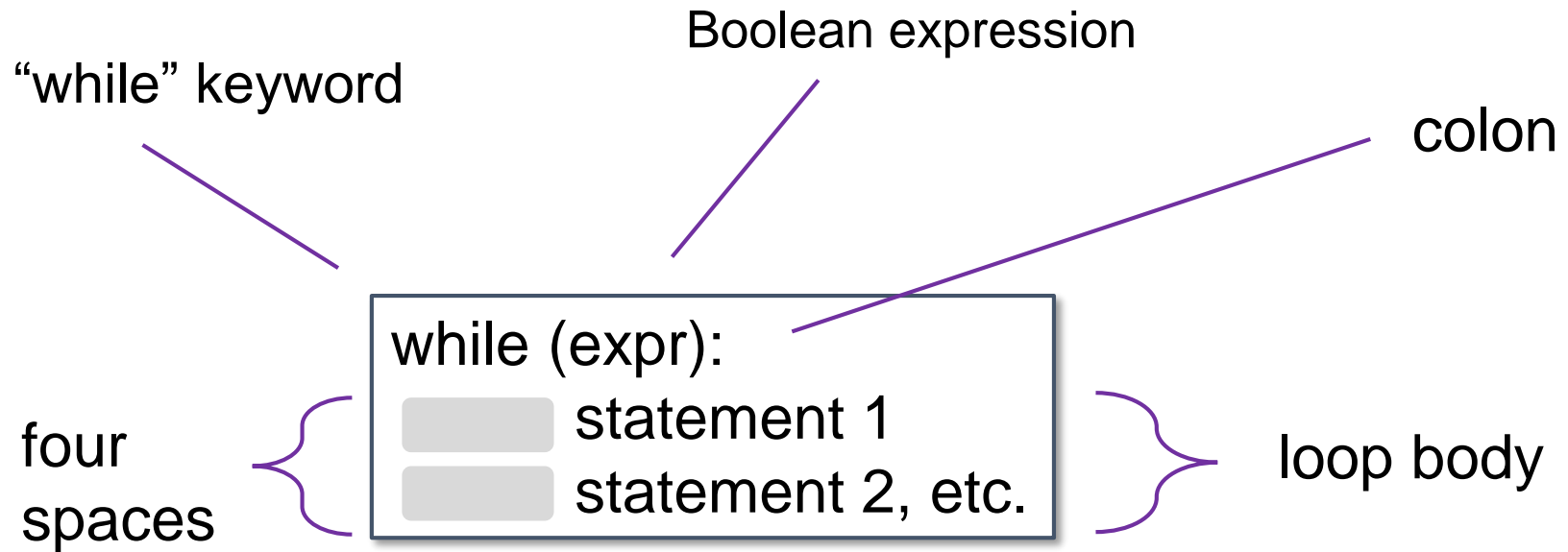


Exercise 2: For Loops

- Write a function that:
 - Accepts a single integer N
 - Returns a list of all odd numbers 1 through N
- Recall that the % operator is used to check for remainders (mod division)



While Loop Syntax



As long as `expr` is True:

- Execute statements in loop body



While Loop Examples

- Try these:

```
a = [ 1 , 2 , 3 ]  
j = 0  
n = len( a )  
while( j < n ):  
    print( a[ j ] )  
    j += 1
```

```
a = [ [ 1 , 2 ] , [ 3 , 4 ] ]  
n = len( a )  
j = 0  
while ( j < n ):  
    print( a[ j ] )  
    for b in a[ j ]:  
        print( b )  
    j += 1
```

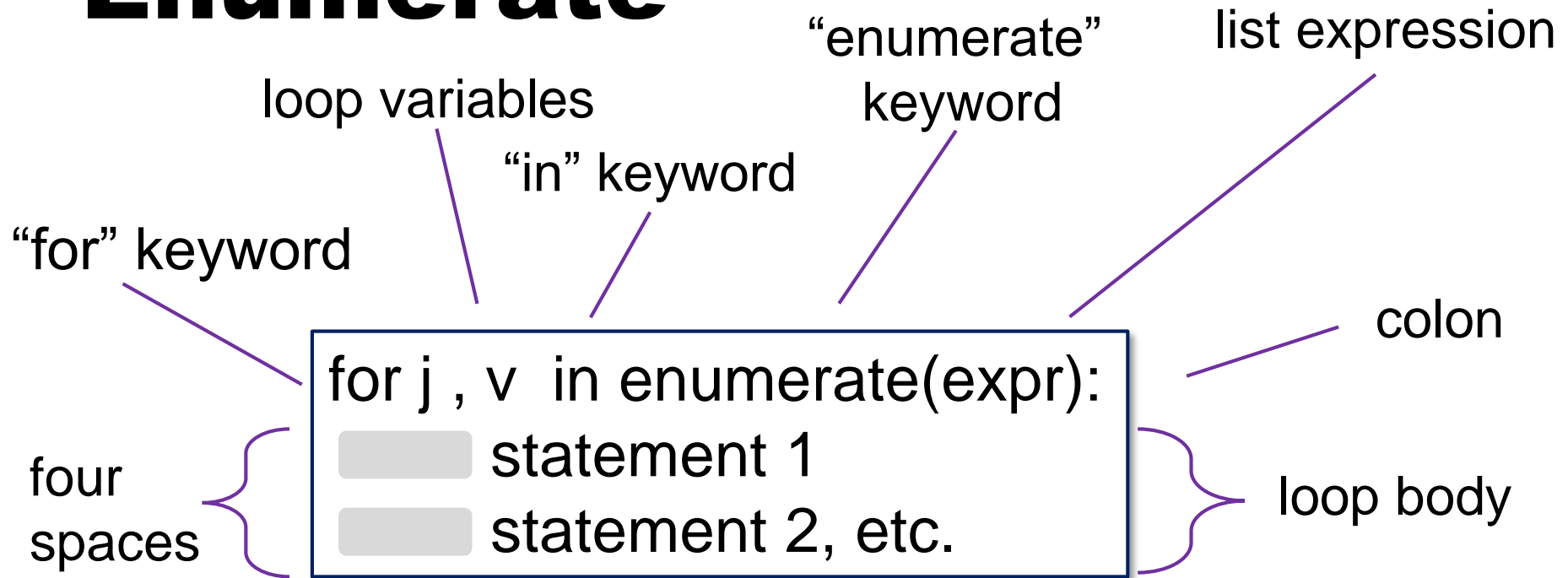


Exercise 3: While Loops

- Write a function that:
 - Accepts two lists a and b
 - Returns the sum of $a[j] * b[j]$ for all elements j in a and b .
 - Replaces $b[j]$ with $b[j] * a[j]$ as a *side effect*
 - a and b should have the same length – if not, return a `NoneType`
- Be sure to use while loops



Enumerate



For each element in `expr`:

- Assign its value to `v`
- Assign a value of 0 through `len(expr)-1` to `j`
- Execute statements in loop body



Enumeration Example

- Try this:

```
a = [ 'John' , 45.0 , 85 ]  
for j, v in enumerate(a):  
    print( j, v )
```



Exercise 4: Enumerate

- Write a function that:
 - Accepts a single parameter, assumed to contain a list of string values
 - Returns a list of string values with their element index appended.
 - For example:
 - Input = ['Hello' , 'There']
 - Return value = ['Hello 0' , 'There 1']
- Be sure to use enumerate



Exercise 5

- Use a loop and the `id` function to print the difference memory addresses each consecutive pair integers 0 through 260.
- E.g., `id(1) – id(0)`; `id(2) – id(1)`, etc.
- What do you notice?
- Try doing it backwards. Does it make a difference?



Next Time

- Defining Classes/OO Programming in Python
- Modules

