Python
Lists

Learning Objectives

- Lists
- Object syntax

Variables and Values

- **Recall**: there are these things called values. Pieces of information (data) which have some characteristic (a data type), which can be operated on.
  - Ex: 3.1415926 (a floating-point number or simply a float), 5 (an integer), “Hello mom” (a string)
- Values can be stored in variables, named locations of memory used to store data.
  - Variables are created using the assignment operator =
  - Ex: salary = 100000

Bigger Data

- Variables have a bit of a scale problem.
  - Ex: How would we store all of the salaries for all of the workers of a company?

Bigger Data

- Variables have a bit of a scale problem.
  - Ex: How would we store all of the salaries for all of the workers of a company?
- We need some mechanism to store large amounts of related data
  - Python has a few, we'll start with the simplest: lists

Lists

- Lists are containers designed to hold a sequence of pieces of related data
  - Each piece of data in a list is referred to as an element of the list or an item in a list
  - Lists are intrinsic (built-in) to Python and a key feature of the language
  - A list value is created by enclosing a set of data in square brackets, separating each piece of data by a comma

```
1 [4, 5, 6]
2 [4.33, 5.44, 6.55]
3 ['Huey', 'Dewey', 'Louie']
4 [[4,5], [6,7],['This', 'is', 'weird']]
```

  - Note: Lists are just considered as value; therefore, lists can contain lists
    - The inner lists are said to be nested in the outer lists

Lists and Variables

- Predictably, lists can be stored in a variable using the assignment operator

```
1  i = [4, 5, 6]
2  f = [4.33, 5.44, 6.55]
3  s = ['Huey', 'Dewey', 'Louie']
4  n = [[4,5], [6,7],['This', 'is', 'weird']]
```

- Accessing an element of a list is done using the square bracket operator
- The number in the square bracket is called the index

```
1  print(i[2])
2  print(f[0])
3  print(s[1])
```

```
6
4.33
Dewey
```

Lists and Variables

- Predictably, lists can be stored in a variable using the assignment operator

```
1  i = [4, 5, 6]
2  f = [4.33, 5.44, 6.55]
3  s = ['Huey', 'Dewey', 'Louie']
4  n = [[4,5], [6,7],['This', 'is', 'weird']]
```

- Accessing an element of a list is done using the square bracket operator
- The number in the square bracket is called the index

Note: indices start at 0, not 1, in Python

How would we print "weird" from n?

```
1  print(i[2])
2  print(f[0])
3  print(s[1])
```

```
6
4.33
Dewey
```

Lists and Variables

- Predictably, lists can be stored in a variable using the assignment operator
- Accessing an element of a list is done using the square bracket operator
- The number in the square bracket is called the index

Note: indices start at 0, not 1, in Python

How would we print "weird" from n?
Print the 3rd element of the 3rd element

```
1  n = [[4,5], [6,7],['This', 'is', 'weird']]
2  print(n[2][2])
```

```
weird
```

Indices

- An index is simply an integer, so any expression which generates an integer can be used as an index

```
1  i = 2
2  seq = [2, 4, 6, 8, 10, 12, 14]
3  print(seq[i])
4  print(seq[i*2])
5  print(seq[i%2])
```

```
6
10
2
```

- Negative indices return values counting from the end of the list

```
1  print(seq[-1])
2  print(seq[6])
```

```
14
14
```

Mutability

- Lists are mutable, the members can be changed
  - Individual members are accessed using the same square bracket operator

```
1  stooges = ['Moe', 'Larry', 'Curly']
2  print(stooges)
3  stooges[2] = 'Shemp'
4  print(stooges)
```

```
['Moe', 'Larry', 'Curly']
['Moe', 'Larry', 'Shemp']
```

in Operator

- The in operator reports true if a given value is an element of a list

```
1  stooges = ['Moe', 'Larry', 'Curly']
2  if 'Curly' in stooges:
3      print("Where's Shemp?")
```

```
Where's Shemp?
```

Traversing a list

- Traversing a list, iterating over a list element by element and performing some set of operations on the element is most easily done with the in operator and a for loop

```
1 stooges = ['Moe', 'Larry', 'Curly']
2 for stooge in stooges:
3     print(stooge)
```

```
Moe
Larry
Curly
```

- This construct is useful when the programmer needs to access, though not update each member of the list

Traversing a list

- To update, use the indices

```
1  stooges = ['Moe', 'Larry', 'Curly']
2  for i in range(len(stooges)):
3      if stooges[i] == 'Curly':
4          stooges[i] = 'Shemp'
5  print(stooges)
```

```
['Moe', 'Larry', 'Shemp']
```

- Note: the len function returns the number of elements in a list

Other list Operators

- + addition concatenates two lists

```
1  a = [1, 3, 5]
2  b = [2, 4, 6]
3  a + b
```

```
[1, 3, 5, 2, 4, 6]
```

- * replicates a list n times

```
1  a = [1, 3, 5]
2  3 * a
```

```
[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Slice Operator

```
1 x = [1, 3, 5, 7, 9]
2 print(x[1:3])
```

```
[3, 5]
```

```
1 x = [1, 3, 5, 7, 9]
2 print(x[:3])
3 print(x[3:])
```

```
[1, 3, 5]
[7, 9]
```

```
1 print(x[3:2])
2 print(x[1:-1])
```

```
[]
[3, 5, 7]
```

- It is possible to extract sublists using the slice operator, :
- The slice operator returns a list comprised of the elements from some index to some other index in the form of list[x:y]
- Note: the sublist is comprised of values from index x to index (y – 1)
- If x is omitted, the beginning of the list is assumed. If y is omitted, the end of the list is assumed. If both are omitted, the whole list is returned
- If y doesn't come after x, an empty list is returned, though negative indices are allowed to find y

List methods

- Methods are functions which operate on some item and are a part of that item. These items are called objects. Objects will be more formally introduced later in the course.
- Methods are invoked using the . (dot) operator
- Lists have a large number of methods

```
1 help(list)
```

```
append(...)
    L.append(object) -> None -- append object to end

clear(...)
    L.clear() -> None -- remove all items from L

copy(...)
    L.copy() -> list -- a shallow copy of L

count(...)
    L.count(value) -> integer -- return number of occurrences of value

extend(...)
    L.extend(iterable) -> None -- extend list by appending elements from the iterable

index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value.
    Raises ValueError if the value is not present.
```

List methods

- A few useful methods
  - list.sort(), sorts the list
    - Note: the list was sorted in place. list.sort() returns none
  - list.append(value) adds an element to the end of a list
  - list.extend(list2) takes all the values of list2 and appends them to list

```
1  stooges = ['Moe', 'Larry', 'Curly']
2  print(stooges)
3  stooges.sort()
4  print(stooges)
```

```
['Moe', 'Larry', 'Curly']
['Curly', 'Larry', 'Moe']
```

```
1  stooges.append('Shemp')
2  print(stooges)
```

```
['Curly', 'Larry', 'Moe', 'Shemp']
```

```
1  new_stooges = ['Bryan', 'Eric']
2  stooges.extend(new_stooges)
3  print(stooges)
```

```
['Curly', 'Larry', 'Moe', 'Shemp', 'Bryan', 'Eric']
```

Reduction

- Many times, it is useful to reduce a list to some value. This requires iterating over the list, performing some reducing operation, such as adding all the value in a list to some sum

```
1  grades = [90, 92, 96, 81]
2  total = 0
3  for grade in grades:
4      total += grade
5  print("Grade = " + str(total/len(grades)))
```

```
Grade = 89.75
```

- *Side note*: this algorithm is called accumulation and the variable total is called the accumulator
  - It is so common, that there is a built-in function to handle it: sum(list)

```
1  print("Grade = " + str(sum(grades)/len(grades)))
```

```
Grade = 89.75
```

Mapping

- Sometimes you want to traverse one list while building another. For example, the following function takes a list of integers and returns a new list that contains the squares of integers:

```
def square_all(t):
    res = []
    for s in t:
        res.append(s**2)
    return res
```

- res is initialized with an empty list; each time through the loop, we append the next element.
- So res is another kind of accumulator.
- An operation like square_all is sometimes called a **map** because it “maps” a function (in this case the method square) onto each of the elements in a sequence.

Filtering

- Another common operation is to select some of the elements from a list and return a sublist.
- For example, the following code segment takes a number range [-5,5) and returns all negative elements by filtering out the positive ones:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)
```

- Note: list function creates a list from any collection of items i.e. range of numbers or strings
- An operation like this code segment is called a **filter** because it selects some of the elements and filters out the others.
- Most common list operations can be expressed as a combination of map, filter and reduce.

Removing elements

- If the element to be removed location is known and the value is desired, use the pop method. pop without an index number, returns the last element of the list

```
1  seq = [1, 1, 2, 3, 5]
2  x = seq.pop(3)
3  print(seq)
4  print(x)
```

```
[1, 1, 2, 5]
3
```

- If the value isn't desired, use the del operation

```
1  seq = [1, 1, 2, 3, 5]
2  del seq[3]
3  print(seq)
```

```
[1, 1, 2, 5]
```

- If the location isn't known, yet the value is, use the remove method.
  - Note: it only removes the first instance of the value

```
1  seq = [1, 1, 2, 3, 5]
2  seq.remove(1)
3  print(seq)
```

```
[1, 2, 3, 5]
```

Functions and lists

- Lists can be passed to a function just like any value
- Unlike scalar values, however, lists are passed by reference
  - For simple scalar values, Python copies the value into the receiving variable
  - Notice how x in __main__ doesn't change

```
1  def adder(x, y):
2      x = x + y
3      print("In adder: " + str(x))
4
5  x = 5
6  y = 7
7  print(x)
8  adder(x, y)
9  print(x)
```

```
5
In adder: 12
5
```

Functions and lists

- Instead of copying all of the values of x to the local x for adder, Python hands the function a reference
- For now, this means changes to a list in a function affect the list from the calling function
  - Notice how x in __main__ does change for a list
  - We'll cover this in more detail in the future

```
1  def adder(x, y):
2      for i in range(len(x)):
3          x[i] = x[i] + y
4      print("In adder: " + str(x))
5
6  x = [1, 2, 3]
7  y = 7
8  print(x)
9  adder(x, y)
10 print(x)
```

```
[1, 2, 3]
In adder: [8, 9, 10]
[8, 9, 10]
```

Resources

- Bryan Burlingame's notes
- Downey, A. (2016) *Think Python, Second Edition* Sebastopol, CA: O'Reilly Media
- (n.d.). 3.7.0 Documentation. *6. Expressions — Python 3.7.0 documentation*. Retrieved September 11, 2018, from http://docs.python.org/3.7/reference/expressions.html