Data Structures

Built-in Data Structures

Values can be collected in data structures:

- Lists
- Tuples
- Dictionaries
- Sets

This lecture just an overview. See the Python documentation for complete details.

Lists

A list is a mutable indexed sequence of Python objects.

- Create a list with square brackets

```
>>> boys = ['Stan', 'Kyle', 'Cartman', 'Kenny']
```

- Create an empty list with empty square brackets or `list()` function

```
>>> empty = []
>>> leer = list()
```

- Add to a list with the `append` method.

```
>>> boys.append("Tweak")
>>> boys
['Stan', 'Kyle', 'Cartman', 'Kenny', 'Tweak']
```

Accessing List Elements

Individual list elements are accessed by index.

- First element at index 0

```
>>> boys = ['Stan', 'Kyle', 'Cartman', 'Kenny']
>>> boys[0]
'Stan'
```

- Negative indexes offset from the end of the list backwards

```
>>> boys[-1]
'Kenny'
```

- Lists are mutable, meaning you can add, delete, and modify elements

```
>>> boys[2] = 'Eric'
>>> boys
['Stan', 'Kyle', 'Eric', 'Kenny']
```

Lists are Heterogeneous

Normally you store elements of the same type in a list, but you can mix element types

```
>>> mixed = [1, 'Two', 3.14]
>>> type(mixed[0])
<class 'int'>
>>> type(mixed[1])
<class 'str'>
>>> type(mixed[2])
<class 'float'>
```

- What's the length of the second element of `mixed` ?

Creating Lists from Strings

- Create a list from a string with `str`'s `split()` method:

```
>>> grades_line = "90, 85, 92, 100"
>>> grades_line.split()
['90,', '85,', '92,', '100']
```

- By default `split()` uses whitespace to delimit elements. To use a different delimiter, pass as argument to `split()`:

```
>>> grades_line.split(',')
['90', ' 85', ' 92', ' 100']
```

- The `list()` function converts any iterable object (like sequences) to a list. Remember that strings are sequences of characters:

```
>>> list('abcdefg')
['a', 'b', 'c', 'd', 'e', 'f', 'g]
```

List Operators

The `in` operator tests for list membership. Can be negated with not:

```
>>> boys
['Stan', 'Kyle', 'Cartman', 'Kenny']
>>> 'Kyle' in boys
True
>>> 'Kyle' not in boys
False
```

- The + operator concatenates two lists, producing a new list:

```
>>> girls = ['Wendy', 'Annie', 'Bebe', 'Heidi']
>>> kids = boys + girls
>>> kids
['Stan', 'Kyle', 'Cartman', 'Kenny', 'Wendy', 'Annie', 'Bebe', 'Heidi']
```

- The * operator repeats a list to produce a new list:

```
>>> ['Ni'] * 5
['Ni', 'Ni', 'Ni', 'Ni', 'Ni']
```

Functions on Lists

Python provides several built-in functions that take list parameters.

- `len(xs)` returns the number of elements in `xs`, where `xs` is any object which has a number of elements.

```
>>> kids
['Stan', 'Kyle', 'Cartman', 'Kenny', 'Wendy', 'Annie', 'Bebe', 'Heidi']
>>> len(kids)
8
```

- `min(xs)` returns the least element of `xs`, `max(xs)` returns the greatest.

```
>>> min([8, 6, 7, 5, 3, 0, 9])
0
>>> max([8, 6, 7, 5, 3, 0, 9])
9
```

- What is `min(kids)`?

The `del` Statement

The `del` statement unbinds a variable from a value.

- Each element of a list is a variable whose name is formed by placing an `int` index in square brackets after a list expression. Here, `boys` is a list expression and `boys[3]` is a variable referring to the fourth element of the list.

```
>>> boys = ['Stan', 'Kyle', 'Cartman', 'Kenny']
>>> boys[3]
'Kenny'
```

- Applying `del` to a list element has the effect of removing it from the list.

```
>>> del boys[3]
>>> boys
['Stan', 'Kyle', 'Cartman'] # You killed Kenny!
```

- A list variable is a variable, so you can delete the whole list

```
>>> del boys
>>> boys
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'boys' is not defined
```

List Methods

- `xs.count(x)`: number of occurrences of `x` in the sequence `xs`

```
>>> surfin_bird = "Bird bird bird b-bird's the word".split()
>>> surfin_bird
['Bird', 'bird', 'bird', "b-bird's", 'the', 'word']
>>> surfin_bird.count('bird')
2
```

- `xs.remove(x)` removes the first occurrence of `x` in `xs`, or raises a `ValueError` if `x` is not in `xs`

```
>>> boys.remove('Kenny')
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Jimmy']
>>> boys.remove('Professor Chaos')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Using a List as a Stack

Use the `append` and `pop` methods to use a list as a stack.

```
>>> rpn = []
>>> rpn.append(3)
>>> rpn.append(2)
>>> rpn.append(int.__mul__)
```

- `xs.pop()` removes and returns the last element of the list

```
>>> op = rpn.pop()
>>> op(rpn.pop(), rpn.pop())
6
```

Slices

Slicing lists works just like slicing strings (they're both sequences)

- Take the first two elements:

```
>>> boys = ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
>>> boys[0:2]
['Stan', 'Kyle']
```

- Take every second element, starting with the first:

```
>>> boys[::2]
['Stan', 'Cartman', 'Tweak']
>>> boys[0:5:2] # same as above
['Stan', 'Cartman', 'Tweak']
```

- Take the second from the end:

```
>>> boys[-2]
'Butters'
```

Note that slice operations return new lists.

- What's the value of `boys[-1:1]` ?
- What's the value of `boys[-1:1:-1]` ?
- What's the value of `boys[::-1]` ?

Aliases

Aliasing occurs when two or more variables reference the same object

- Assignment from a variable creates an alias

```
>>> brats = boys
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
>>> brats
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
```

Now boys and brats are aliases.

- Changes to one are reflected in the other, becuase they reference the same object

```
>>> brats.append('Timmy')
>>> brats
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

Copies

Operators create copies

```
>>> brats + ['Bebe', 'Wendy']
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy', 'Bebe',
'Wendy']
>>> brats
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

You have to reassign to the list to make an update:

```
>>> brats = brats + ['Bebe', 'Wendy'] # could also use shortcut +=
>>> brats
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy', 'Bebe',
'Wendy']
```

Notice that after the reassignment, `brats` is no longer an alias of `boys`

```
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

Slicing

- Slice on the right hand side of an assignment creates a copy:

```
>>> first_two = boys[:2]
>>> first_two
['Stan', 'Kyle']
>>> first_two[0] = 'Stan the man'
>>> first_two
['Stan the man', 'Kyle']
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

- Slices on the left hand side allow for flexible assignment. Here we splice in 4 new elements in place of first 2 elements of `boys`:

```
>>> boys[0:2] = ['Randy', 'Sharon', 'Gerald', 'Sheila']
>>> boys
['Randy', 'Sharon', 'Gerald', 'Sheila', 'Cartman', 'Butters',
'Tweak', 'Timmy']
```

A Few More List Operations

You can combine the elements of a list to form a string with `str`'s `join()` method.

```
>>> aretha = ['R', 'E', 'S', 'P', 'E', 'C', 'T']
>>> "-".join(aretha)
'R-E-S-P-E-C-T'
```

`sorted()` function returns a new list

```
>>> sorted(aretha)
['C', 'E', 'E', 'P', 'R', 'S', 'T']
>>> aretha # Notice original is unchanged
['R', 'E', 'S', 'P', 'E', 'C', 'T']
```

`sort()` method modifies the list it is invoked on

```
>>> aretha.sort()
>>> aretha
['C', 'E', 'E', 'P', 'R', 'S', 'T']
```

Active Review

Given a list representing a line from a gradebook file:

```
1 >>> grades_line = ['Chris', 100, 90, 95]
```

- Use a slice to assign the grades to a variable named `grades`.
- Sum the grades using Python's built-in `sum()` function.
- Combine the sum of the grades with the length of the grades to find the average.

Tuples

Tuples are like lists, but are immutable.

```
Tuples are created by separating objects with commas
>>> pair = 1, 2
>>> pair
(1, 2)
```

Tuples can be used in assignments to “unpack” a sequence

```
>>> a, b = [1, 2]
>>> a
1
>>> b
2
```

Tuple assignment can be used to swap values

```
>>> b, a = a, b
>>> a, b
(2, 1)
```

Dictionaries

A dictionary is a map from keys to values.

Create dictionaries with `{}`

```
>>> capitals = {}
```

Add key-value pairs with assignment operator

```
>>> capitals['Georgia'] = 'Atlanta'
>>> capitals['Alabama'] = 'Montgomery'
>>> capitals
{'Georgia': 'Altanta', 'Alabama': 'Montgomery'}
```

Keys are unique, so assignment to same key updates mapping

```
>>> capitals['Alabama'] = 'Birmingham'
>>> capitals
{'Georgia': 'Altanta', 'Alabama': 'Birmingham'}
```

Dictionary Operations

Remove a key-value mapping with `del` statement

```
>>> del capitals['Alabama']
>>> capitals
{'Georgia': 'Atlanta'}
```

Use the `in` operator to test for existence of key (not value)

```
>>> 'Georgia' in capitals
True
>>> 'Atlanta' in capitals
False
```

Extend a dictionary with `update()` method, get values as a list with values method

```
>>> capitals.update({'Tennessee': 'Nashville', 'Mississippi':
'Jackson'})
>>> capitals.values()
dict_values(['Jackson', 'Nashville', 'Atlanta'])
```

Conversions to `dict`

Any sequence of two-element sequences can be converted to a `dict`

A list of two-element lists:

```
>>> dict([[1, 1], [2, 4], [3, 9], [4, 16]])
{1: 1, 2: 4, 3: 9, 4: 16}
```

A list of two-element tuples:

```
>>> dict([('Lassie', 'Collie'), ('Rin Tin Tin', 'German
Shepherd')])
{'Rin Tin Tin': 'German Shepherd', 'Lassie': 'Collie'}
```

Even a list of two-character strings:

```
>>> dict(['a1', 'a2', 'b3', 'b4'])
{'b': '4', 'a': '2'}
```

Notice that subsequent pairs overwrote previously set keys.

Sets

Sets have no duplicates, like the keys of a `dict`. They can be iterated over (we'll learn that later) but can't be accessed by index.

- Create an empty set with `set()` function, add elements with `add()` method

```
>>> names = set()
>>> names.add('Ally')
>>> names.add('Sally')
>>> names.add('Mally')
>>> names.add('Ally')
>>> names
{'Ally', 'Mally', 'Sally'}
```

- Converting to set a convenient way to remove duplicates

```
>>> set([1,2,3,4,3,2,1])
{1, 2, 3, 4}
```

Set Operations

Intersection (elements in `a` *and* `b`)

```
>>> a = {1, 2}
>>> b = {2, 3}
>>> a & b # or a.intersetion(b)
{2}
```

Union (elements in `a` *or* `b`)

```
>>> a | b # or a.union(b)
{1, 2, 3}
```

Difference (elements in `a` that are not in `b`)

```
>>> a - b # or a.difference(b)
{1}
```

Symmetric difference (elements in `a` or `b` but not both)

```
>>> a ^ b # or a.symmetric_difference(b)
{1, 3}
```

Set Predicates

A predicate function asks a question with a `True` or `False` answer.

Subset of:

```
>>>a <= b # or a.issubset(b)
False
```

Proper subset of:

```
>>> a < b
False
```

Superset of:

```
>>> a >= b # or a.issuperset(b)
False
```

Proper superset of:

```
>>> a > b
False
```

Closing Thoughts

Typical Python programs make extensive use of built-in data structures and often combine them (lists of lists, dictionaries of lists, etc)

- These are just the basics
- Explore these data structures on your own
- Read the books and Python documentation

This is a small taste of the expressive power and syntactic convenience of Python's data structures.