

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

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

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

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

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

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

Accessing List Elements

Individual list elements are accessed by index.

► First element at index 0

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

► Negative indexes offset from the end of the list backwards

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

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

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

Lists are Heterogeneous

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

```
1 >>> mixed = [1, 'Two', 3.14]
2 >>> type(mixed[0])
3 <class 'int'>
4 >>> type(mixed[1])
5 <class 'str'>
6 >>> type(mixed[2])
7 <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:

```
1 >>> grades_line = "90, 85, 92, 100"  
2 >>> grades_line.split()  
3 ['90,', ' 85,', ' 92,', ' 100']
```

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

```
1 >>> grades_line.split(',')  
2 ['90', ' 85', ' 92', ' 100']
```

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

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

List Operators

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

```
1 >>> boys
2 ['Stan', 'Kyle', 'Cartman', 'Kenny']
3 >>> 'Kyle' in boys
4 True
5 >>> 'Kyle' not in boys
6 False
```

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

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

► The `*` operator repeats a list to produce a new list:

```
1 >>> ['Ni'] * 5
2 ['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.

```
1 >>> kids
2 ['Stan', 'Kyle', 'Cartman', 'Kenny', 'Wendy', 'Annie', 'Bebe', 'Heidi']
3 >>> len(kids)
4 8
```

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

```
1 >>> min([8, 6, 7, 5, 3, 0, 9])
2 0
3 >>> max([8, 6, 7, 5, 3, 0, 9])
4 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.

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

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

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

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

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

List Methods

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

```
1 >>> surfin_bird = "Bird bird bird b-bird's the word".split()
2 >>> surfin_bird
3 ['Bird', 'bird', 'bird', "b-bird's", 'the', 'word']
4 >>> surfin_bird.count('bird')
5 2
```

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

```
1 >>> boys.remove('Kenny')
2 >>> boys
3 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Jimmy']
4 >>> boys.remove('Professor Chaos')
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   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.

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

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

```
1 >>> op = rpn.pop()
2 >>> op(rpn.pop(), rpn.pop())
3 6
```

Slices

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

► Take the first two elements:

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

► Take every second element, starting with the first:

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

► Take the second from the end:

```
1 >>> boys[-2]
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

```
1 >>> brats = boys
2 >>> boys
3 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
4 >>> brats
5 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
```

Now boys and brats are aliases.

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

```
1 >>> brats.append('Timmy')
2 >>> brats
3 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
4 >>> boys
5 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

Copies

Operators create copies

```
1 >>> brats + ['Bebe', 'Wendy']
2 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy', 'Bebe',
3  'Wendy']
4 >>> brats
5 ['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Timmy']
```

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

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

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

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

Slicing

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

```
1 >>> first_two = boys[:2]
2 >>> first_two
3 ['Stan', 'Kyle']
4 >>> first_two[0] = 'Stan the man'
5 >>> first_two
6 ['Stan the man', 'Kyle']
7 >>> boys
8 ['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`:

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

A Few More List Operations

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

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

`sorted()` function returns a new list

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

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

```
1 >>> aretha.sort()
2 >>> aretha
3 ['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.

```
1 Tuples are created by separating objects with commas
2 >>> pair = 1, 2
3 >>> pair
4 (1, 2)
```

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

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

Tuple assignment can be used to swap values

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

Dictionaries

A dictionary is a map from keys to values.

Create dictionaries with {}

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

Add key-value pairs with assignment operator

```
1 >>> capitals['Georgia'] = 'Atlanta'
2 >>> capitals['Alabama'] = 'Montgomery'
3 >>> capitals
4 {'Georgia': 'Atlanta', 'Alabama': 'Montgomery'}
```

Keys are unique, so assignment to same key updates mapping

```
1 >>> capitals['Alabama'] = 'Birmingham'
2 >>> capitals
3 {'Georgia': 'Atlanta', 'Alabama': 'Birmingham'}
```

Dictionary Operations

Remove a key-value mapping with `del` statement

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

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

```
1 >>> 'Georgia' in capitals
2 True
3 >>> 'Atlanta' in capitals
4 False
```

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

```
1 >>> capitals.update({'Tennessee': 'Nashville', 'Mississippi':
2 'Jackson'})
3 >>> capitals.values()
4 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:

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

A list of two-element tuples:

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

Even a list of two-character strings:

```
1 >>> dict(['a1', 'a2', 'b3', 'b4'])
2 {'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

```
1 >>> names = set()
2 >>> names.add('Ally')
3 >>> names.add('Sally')
4 >>> names.add('Mally')
5 >>> names.add('Ally')
6 >>> names
7 {'Ally', 'Mally', 'Sally'}
```

- Converting to set a convenient way to remove duplicates

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

Set Operations

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

```
1 >>> a = {1, 2}
2 >>> b = {2, 3}
3 >>> a & b # or a.intersection(b)
4 {2}
```

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

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

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

```
1 >>> a - b # or a.difference(b)
2 {1}
```

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

```
1 >>> a ^ b # or a.symmetric_difference(b)
2 {1, 3}
```

Set Predicates

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

Subset of:

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

Proper subset of:

```
1 >>> a < b
2 False
```

Superset of:

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

Proper superset of:

```
1 >>> a > b
2 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.