

Sequences

Built-in Data Structures

Values can be collected in data structures:

- ▶ Lists
- ▶ Tuples
- ▶ Dictionaries
- ▶ Sets

See the [Python documentation](#) for complete details.

By the end of this lesson you will

- ▶ know how to use lists and tuples

Lists

A list is an 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()
```

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()` function:

```
>>> 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('abcdefghijklmnopqrstuvwxyz')  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',  
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',  
'z']
```

- Use the `split()` method to separate an email address in to user name and host name.

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:

```
>>> 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 the list `xs` (more generally, the sequence `xs`)

```
>>> 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 deletes variables.

- ▶ Each element of a list is a variable whose name is formed by indexing into the list with square brackets.

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

- ▶ Like any variable, a list element can be deleted with `del`

```
>>> 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

Methods are invoked on an object (an instance of a class) by appending a dot, ., and the method name.

- ▶ `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.append(x)` adds the single element `x` to the end of `xs`

```
>>> boys.append('Butters')
>>> boys
['Stan', 'Kyle', 'Cartman', 'Kenny', 'Butters']
s.extend(t) adds the elements of t to the end of s
>>> boys.extend(['Tweak', 'Jimmy'])
>>> boys
['Stan', 'Kyle', 'Cartman', 'Kenny', 'Butters', 'Tweak', 'Jimmy']
```

List Methods

- ▶ `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
```

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

```
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak', 'Jimmy']
>>> boys.pop()
'Jimmy'
>>> boys
['Stan', 'Kyle', 'Cartman', 'Butters', 'Tweak']
```

Slicing

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, because 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 Creates Copies (Usually)

- ▶ 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

```
>>> 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']
```


Example: Grades

Start with a list representing a line from a gradebook file

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

Get the sublist containing just the grades by slicing

```
>>> grades = grades_line[1:]
>>> grades
[100, 90, 95]
```

Sum the grades using Python's built-in `sum()` function

```
>>> sum(grades)
285
```

► And get the average by dividing by the number of grades

```
>>> sum(grades) / len(grades)
95.0
```

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)
```

Conclusion

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.