3. Collections

Based on Practical Python Programming

Lists

This section introduces lists, Python's primary type for holding an ordered collection of values.

Creating a List

Use square brackets to define a list literal:

```
names = [ 'Elwood', 'Jake', 'Curtis' ]
nums = [ 39, 38, 42, 65, 111]
```

Sometimes lists are created by other methods. For example, a string can be split into a list using the `split()` method:

```
>>> line = 'GOOG,100,490.10'
>>> row = line.split(',')
>>> row
['GOOG', '100', '490.10']
>>>
```

List operations

Lists can hold items of any type. Add a new item using `append()`:

```
names.append('Murphy')    # Adds at end
names.insert(2, 'Aretha') # Inserts in middle
```

Use `+` to concatenate lists:

```
s = [1, 2, 3]
t = ['a', 'b']
s + t           # [1, 2, 3, 'a', 'b']
```

Lists are indexed by integers. Starting at 0.

```
names = [ 'Elwood', 'Jake', 'Curtis' ]

names[0]  # 'Elwood'
names[1]  # 'Jake'
names[2]  # 'Curtis'
```

Negative indices count from the end.

```
names[-1] # 'Curtis'
```

You can change any item in a list.

```
names[1] = 'Joliet Jake'
names                    # [ 'Elwood', 'Joliet Jake', 'Curtis' ]
```

Length of the list.

```
names = ['Elwood','Jake','Curtis']
len(names)  # 3
```

Membership test (`in` , `not in`).

```
'Elwood' in names       # True
'Britney' not in names  # True
```

Replication (`s * n`).

```
s = [1, 2, 3]
s * 3   # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Iteration and Search

Use `for` to iterate over the list contents.

```
for name in names:
    # use name
    # e.g. print(name)
    ...
```

This is similar to a `foreach` statement from other programming languages.

To find the position of something quickly, use `index()` .

```
names = ['Elwood','Jake','Curtis']
names.index('Curtis')   # 2
```

If the element is present more than once, `index()` will return the index of the first occurrence.

If the element is not found, it will raise a `ValueError` exception.

List Removal

You can remove items either by element value or by index:

```
# Using the value
names.remove('Curtis')

# Using the index
del names[1]
```

Removing an item does not create a hole. Other items will move down to fill the space vacated. If there are more than one occurrence of the element, `remove()` will remove only the first occurrence.

List Sorting

Lists can be sorted “in-place”.

```
s = [10, 1, 7, 3]
s.sort()                    # [1, 3, 7, 10]

# Reverse order
s = [10, 1, 7, 3]
s.sort(reverse=True)        # [10, 7, 3, 1]

# It works with any ordered data
s = ['foo', 'bar', 'spam']
s.sort()                    # ['bar', 'foo', 'spam']
```

Use `sorted()` if you’d like to make a new list instead:

```
t = sorted(s)               # s unchanged, t holds sorted values
```

Data Structures

Real programs have more complex data. For example information about a stock holding:

```
100 shares of GOOG at $490.10
```

This is an “object” with three parts:

- Name or symbol of the stock (“GOOG”, a string)
- Number of shares (100, an integer)
- Price (490.10 a float)

Tuples

A tuple is a collection of values grouped together.

Example:

```
s = ('GOOG', 100, 490.1)
```

Sometimes the `()` are omitted in the syntax.

```
s = 'GOOG', 100, 490.1
```

Special cases (0-tuple, 1-tuple).

```
t = ()           # An empty tuple
w = ('GOOG', )   # A 1-item tuple
```

Tuples are often used to represent *simple* records or structures. Typically, it is a single *object* of multiple parts. A good analogy: *A tuple is like a single row in a database table.*

Tuple contents are ordered (like an array).

```
s = ('GOOG', 100, 490.1)
name = s[0]                 # 'GOOG'
shares = s[1]               # 100
price = s[2]                # 490.1
```

However, the contents can't be modified.

```
>>> s[1] = 75
TypeError: object does not support item assignment
```

You can, however, make a new tuple based on a current tuple.

```
s = (s[0], 75, s[2])
```

Tuple Packing

Tuples are more about packing related items together into a single *entity*.

```
s = ('GOOG', 100, 490.1)
```

The tuple is then easy to pass around to other parts of a program as a single object.

Tuple Unpacking

To use the tuple elsewhere, you can unpack its parts into variables.

```
name, shares, price = s
print('Cost', shares * price)
```

The number of variables on the left must match the tuple structure.

```
name, shares = s     # ERROR
Traceback (most recent call last):
...
ValueError: too many values to unpack
```

Tuples vs. Lists

Tuples look like read-only lists. However, tuples are most often used for a *single item* consisting of multiple parts. Lists are usually a collection of distinct items, usually all of the same type.

```
record = ('GOOG', 100, 490.1)       # A tuple representing a record in a portfolio

symbols = [ 'GOOG', 'AAPL', 'IBM' ]  # A List representing three stock symbols
```

Dictionaries

A dictionary is mapping of keys to values. It's also sometimes called a hash table or associative array. The keys serve as indices for accessing values.

```
s = {
    'name': 'GOOG',
    'shares': 100,
    'price': 490.1
}
```

Common operations

To get values from a dictionary use the key names.

```
>>> print(s['name'], s['shares'])
GOOG 100
>>> s['price']
490.10
>>>
```

To add or modify values assign using the key names.

```
>>> s['shares'] = 75
>>> s['date'] = '6/6/2007'
>>>
```

To delete a value use the `del` statement.

```
>>> del s['date']
>>>
```

Why dictionaries?

Dictionaries are useful when there are *many* different values and those values might be modified or manipulated. Dictionaries make your code more readable.

```
s['price']
# vs
s[2]
```