Dictionaries

- Dictionaries are an *associative container*. They contain *keys* with associated *values*
- Dictionaries in Python are denoted by curly braces
- Create an empty dictionary: `empty_dict = {}` or `empty_dict = dict()`
- Create a dictionary with some data: `ages = {"brad": 51, "angelina": 40}`
- Values can be any python object: numbers, strings, lists, other dictionaries
- Keys can be any immutable object: numbers, strings, tuples (containing immutable data)
- No sense of order in a python dictionary. When used in a loop, the key-value pairs may come out in any order.
- Access values associated with a key with square brackets: `value = dictionary[key]`

Create a dictionary

```
simple_dict = {1:"ths number 1", "42": 42, "a list": [1,2,3]}
print(simple_dict)
```

Here is a slightly more useful dictionary associating names with ages. We start with an empty dictionary and add to it.

```
ages = {} # or ages = dict()
ages['brad'] = 51
ages['angelina'] = 40
ages['leo'] = 40
ages['bruce'] = 60
ages['cameron'] = 44
ages
```

We can get the size of the dictionary with `len`:

```
len(ages)
```

Access items

```
ages['leo']
```

When the key does not exist:

```
ages['helen']
```

Or you can use the `get()` method:

```
temp = ages.get('brad')
print(temp)
temp = ages.get('helen')
print(temp)
```

Updating values

We can change the value associated with a key.

```
ages['brad'] = 52
print(ages)
```

Checking for existence of a key

The `in` operator is used to check for the existence of a key in a dictionary.

```
print(ages)
```

```
'nick' in ages
```

```
'leo' in ages
```

Iteration

Iterate through the keys with:

```
for key in ages:
    print("{} = {}".format(key,ages[key]))
```

or:

```
for key in ages.keys():
    print("{} = {}".format(key,ages[key]))
```

Loop over values:

```
# loop over values
for value in ages.values():
    print(value)
```

Iterate through key-values pairs with:

```
for k, v in ages.items():
    print('{} is {} years old'.format(k,v))
```

Note: The above syntax is more efficient in Python 3 compared with Python 2. To achieve equivalent performance in Python 2, it is best to ask for an *iterator* over the key-value pairs.

```
# only in python 2
for k, v in ages.iteritems():
    print('{} = {}'.format(k,v))
```

What does `ages.items()` return?

```
ages.items()
```

This is a Python object that provides access to the data in a container in a sequential fashion **without** requiring the creation of a new data structure and copying of data. This is sort of like the `range()` function.

Dictionary methods

See `help(dict)` to get a summary of all dictionary methods:

```
clear(...)
    D.clear() -> None.  Remove all items from D.

copy(...)
    D.copy() -> a shallow copy of D

fromkeys(iterable, value=None, /) from builtins.type
    Returns a new dict with keys from iterable and values equal to value.
```

```
get(...)
    D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.

items(...)
    D.items() -> a set-like object providing a view on D's items

keys(...)
    D.keys() -> a set-like object providing a view on D's keys

pop(...)
    D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised

popitem(...)
    D.popitem() -> (k, v), remove and return some (key, value) pair as a
    2-tuple; but raise KeyError if D is empty.

setdefault(...)
    D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update(...)
    D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
    If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
    If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
    In either case, this is followed by: for k in F:  D[k] = F[k]

values(...)
    D.values() -> an object providing a view on D's values
```