1 Tuples

A ***tuple*** is a new kind of object that is like a list (it has elements that you can index and slice), but it is immutable.

We define a tuple with parentheses:

```
t = ('one', 'two', 'three')
```

and access it with square brackets:

```
t
('one', 'two', 'three')
t[0]
'one'
t[1]
'two'
t[2]
'three'
t[1:]
('two', 'three')
```

Q. Since it's ***immutable***, what can you NOT do with it?

A. We cannot:

1.
```
t[0] = 'seven' # error
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'tuple' object does not support item assignment
```

cannot assign to an element

2.
```
t.append('four') # error
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AttributeError: 'tuple' object has no attribute 'append'
```

cannot call methods or functions that change it

In fact, there are no tuple ***methods*** you can call. Even these list methods that ***do not*** change a list are not provided for tuples:

```
dir(tuple)
['__add__', '__class__', '__contains__',
 '__delattr__', '__doc__', '__eq__', '__f
ormat__', '__ge__', '__getattribute__', '
__getitem__', '__getnewargs__', '__ge
tslice__', '__gt__', '__hash__', '__init_
_', '__iter__', '__le__', '__len__', '__lt_
_', '__mul__', '__ne__', '__new__', '__r
educe__', '__reduce_ex__', '__repr__',
'__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'count', '
index']

t.index(1)
Traceback (most recent call last):
  File "<string>", line 1, in
<fragment>
ValueError: tuple.index(x): x not in
tuple
t.index('two')
1
t.count('two')
1
```

However, you can still call function **`len`** and ***loop*** through a tuple:

```
len(t)
3
for item in t:
        print item

one
two
three
```

```
years = [1799, 1800, 1801, 1802, 1902, 2002]
emissions = [1, 70, 74, 79, 82, 215630, 1733297] # metric tons of carbon, 1000s
```

2 Dictionaries

2.1 Motivation

Suppose we need to represent years and the total North American fossil fuel CO2 emissions for those years.

Q. How should we do this?

Option one:

```
# to add an entry: need to insert or append to both lists
# to edit the emissions for a particular year: need to find
# the year in the years list and modify the corresponding
# element in the emissions list
```

Option two:

```
#option two: list of lists
# better, but still a bit of pain to look up a year
# must search the list to find the year
```

There is a better way: a new type of object called dictionary (type **`dict`**).

2.2 Dictionary Basics

A dictionary keeps track of associations for you. A dictionary entry has two parts: a key and a value. This is similar to a regular dictionary where a ***key*** is a word and the ***value*** is the definition of that word.

```
emission_by_year = {1799: 1, 1800: 70, 1801: 74, 1802: 82, 1902: 215630, 2002: 1733297}
emission_by_year[1801]
74
# 1801 is a key and 74 is its value
# keys must be unique
# values can be duplicated
emission_by_year[1950] =734914
emission_by_year
{1799: 1, 1800: 70, 1801: 74, 1802: 82, 1902: 215630, 2002: 1733297, 1950: 734914}
# dictionary is mutable
emission_by_year[1802] = 100
emission_by_year
{1799: 1, 1800: 70, 1801: 74, 1802: 100, 1902: 215630, 2002: 1733297, 1950: 734914}
```

To define a `dict`:

```
mydictionary = {key1:value1, key2:value2, etc.}
```

To look up a `key`'s `value`:

```
mydictionary[key]
```

To add another (key, value) pair:

```
mydictionary[new_key] = new_value
```

`key` can be any ***immutable*** object (string, int, tuple).
`value` can be any object.

Some dictionary examples:

```
# braces indicate that you are defining a dictionary
```

see the stuffs above

```
# Look up the emissions for the given year


# Add another year to the dictionary



# Key's don't have to be numbers, but they have to be immutable.
```

```
>>> d = {1:5, 3:45, 4:10}
>>> d[[1, 2, 3]] = 100    # error
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: unhashable type: 'list'
>>> d[(1, 2, 3)] = 100   # tuples are immutable, so this is fine.
>>> d
{1: 5, 3: 45, 4: 10, (1, 2, 3): 100}
```

```
# values can be any type, mutable or not
```

Dictionaries themselves are mutable:

```
>>> d[5] = ["Diane", "Paul", "Karen"]
>>> d['weird'] = ['hello', 'bye']
>>> d['nested'] = {'diane':4236, 'paul':4234}
>>> d
{1: 5, 3: 45, 4: 10, 5: ['Diane', 'Paul', 'Karen'], (1, 2, 3): 100, 'weird':
['hello', 'bye'], 'nested': {'paul': 4234, 'diane': 4236}}
>>> d['nested']
{'paul': 4234, 'diane': 4236}
>>> d['nested']['paul']
4234
```

```
>>> id(d)
17224304
>>> d['me'] = 'you'
>>> id(d)
17224304
```

```
>>> emissions_by_year
{1799: 2, 1800: 70, 1801: 74, 1802: 82, 1803: 100, 1902: 215630, 2002: 1733297}
```

2.3 Dictionary Operations

```
emissions_by_year

# extend (add a new key and its value)
   >>> # extend
   emissions_by_year[2009] = 1000000
# update (change the value associated with a key)
  >>> # update
  emissions_by_year[2002] = 10


# check for membership
 >>> # check for membership
 >>> 1950 in emissions_by_year
 False
 >>> 2002 in emissions_by_year
 True

# remove a key-value pair
 >>> # remove a key-value pair
 >>> del emissions_by_year[1803]
 >>> 1803 in emissions_by_year
 False
# determine length (number of key-value pairs)
 >>> # determine length
 >>> len(emissions_by_year)
 7

# Iterating over the dictionary
```

```
>>> # iterate over a dict
for key in emissions_by_year:
 print key

1799
1800
1801
1802
1902
2002
2009

>>> for key in d:
 print key

me
1
3
4
5
(1, 2, 3)
weird
nested
>>> # Dictionaries are unordered.
The order that the keys
>>> # are traversed is arbitrary: no
guarantee that it
>>> # will be the order that they
were added to it.

>>> for key in emissions_by_year:
 print emissions_by_year[key]

2
70
74
82
215630
10
1000000
```

```
>>> # Methods
>>> dir(dict)
['__class__', '__cmp__', '__contai
ns__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__ge
titem__', '__gt__', '__hash__', '__i
nit__', '__iter__', '__le__', '__len_
_', '__lt__', '__ne__', '__new__', '_
_reduce__', '__reduce_ex__', '__re
pr__', '__setattr__', '__setitem__', '
__sizeof__', '__str__', '__subclass
hook__', 'clear', 'copy', 'fromkeys',
'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'va
lues', 'viewitems', 'viewkeys', 'vie
wvalues']
```

2.4 Dictionary Methods

Use **`dir(dict)`** to see a list of dictionary methods.

```
# keys
 >>> emissions_by_year.keys()
 [1799, 1800, 1801, 1802, 1902, 2002, 2009]

# values
 >>> emissions_by_year.values()
 [2, 70, 74, 82, 215630, 10, 1000000]


# items: the (key, value) pairs as a list of tuples
 >>> emissions_by_year.items()
 [(1799, 2), (1800, 70), (1801, 74), (1802, 82), (1902, 215630), (2002, 10), (2009, 1000000)]
```

```
>>> empty_dict = {}
>>> empty_dict.keys()
[]
>>> empty_dict.values()
[]
>>> empty_dict.items()
[]
```

```
# get: Same as [] notation, but doesn't crash when the key is not there.

>>> emissions_by_year.get(1802)
82
>>> emissions_by_year.get(1805)
>>> print emissions_by_year.get(1805)
None
>>> emissions_by_year[1805]
Traceback (most recent call last):
```

```
 File "<string>", line 1, in <fragment>
KeyError: 1805
>>> emissions_by_year.get(1805)
>>> emissions_by_year.get(1805, -1)  # asking to get -1 return if key not there
-1
>>> emissions_by_year.get(1805, "not there")
'not there'
```

```
# update: copies values from one dictionary into another
>>> dict1 = {'978-6025': 'Diane', '978-3965' : 'Tom'}
>>> dict2 = {'978-6360' : 'UG office', '978-6025' : 'main office'}
>>> dict1.update(dict2)  # add dict2 contents to dict1; updates keys from dict1 with key-value pairs from dict2
>>> dict1
{'978-6025': 'main office', '978-3965': 'Tom', '978-6360': 'UG office'}
>>> help(dict.update)
Help on method_descriptor:
update(...)  # Can use update to extend a dictionary (as we just saw) or to modify it
    D.update(E, **F) -> None.  Update D from dict/iterable E and F.
    If E has a .keys() method, does:     for k in E: D[k] = E[k]
    If E lacks .keys() method, does:     for (k, v) in E: D[k] = v
    In either case, this is followed by: for k in F: D[k] = F[k]

>>> d1 = {1:1, 2:2}
>>> d2 = {2: 222, 3:3}
>>> d1.update(d2)
>>> d1
{1: 1, 2: 222, 3: 3}
>>> d2
{2: 222, 3: 3}
```

```
# clear: removes the dictionary's contents
   >>> d2.clear()
   >>> d2
   {}
```

2.5 Iterating Through A Dictionary

```
phone = {'555-7632': 'Paul', '555-9832': 'Andrew', '555-6677': 'Dan', \
'555-9823': 'Michael', '555-6342' : 'Cathy', '555-7343' : 'Diane'}
```

- Going through the keys

```
>>> # The proper way:
>>> for key in phone:
 print key

555-7632
555-9999
555-3333
555-2222
555-1111
```

```
>>> # The equivalent, but not considered good style.
>>> for key in phone.keys():
 print key

555-7632
555-9999
555-3333
555-2222
555-1111
```

- Going through the values

```
>>> for value in phone.values():
 print value

Paul
Andrew
Karen
Michael
Dan
```

```
>>> for item in phone.items():
 print item

('555-7632', 'Paul')
('555-9999', 'Andrew')
('555-3333', 'Karen')
('555-2222', 'Michael')
('555-1111', 'Dan')
```

```
>>> # You can pull the pieces of the tuple out as you go:
>>> for (number, name) in phone.items():
 print "Name: %s; Phone number: %s" % (name, number)

Name: Paul; Phone number: 555-7632
Name: Andrew; Phone number: 555-9999
Name: Karen; Phone number: 555-3333
Name: Michael; Phone number: 555-2222
Name: Dan; Phone number: 555-1111
>>> x = 4
>>> y = 3.4
>>> print 'hello %d, by %f' % (x, y)
hello 4, by 3.400000
>>> print 'hello ' + str(x) + ', by ' + str(y)
hello 4, by 3.4
```

- Going through the key-value pairs

2.6 Inverting A Dictionary

The dictionary **phone** maps phone numbers to names. Some people have more than one phone number.

```
phone = {'555-7632': 'Paul', '555-9832': 'Andrew', '555-6677': 'Dan', \
'555-9823': 'Michael', '555-6342' : 'Karen', '555-2222': 'Paul',\
 '555-7343' : 'Anna'}
```

Suppose we want to create a list of all of Paul's phone numbers:

```
>>> phone
{'555-7632': 'Paul', '555-9832': 'Andrew', '555-7343': 'Anna', '555-9823': 'Michael', '555-6342': 'Karen', '555-6677': 'Dan', '555-2222': '
Paul'}
>>> # we want Paul's phone numbers

>>> phone.values()
['Paul', 'Andrew', 'Anna', 'Michael', 'Karen', 'Dan', 'Paul']
>>> for name in phone.values():
 # no easy way to get num associated with name
 print name

Paul
Andrew
Anna
Michael
Karen
Dan
Paul
>>> # Instead loop over keys
>>> for number in phone:
 if phone[number] == 'Paul':
  print number

555-7632
555-2222
>>> paul = []
>>> for key in phone:
 if phone[key] == 'Paul':
  paul.append(key)

>>> paul
['555-7632', '555-2222']
```

Q. But what if we want to be able to do this for all people? Is there some object you could create to make this easy?

A.

We call this an ***inverted*** dictionary.

```
>>> # Invert the dictionary
>>> # Goal: new dict with names as keys, numbers as values
>>> new_phone = {}
>>> phone
{'555-7632': 'Paul', '555-9832': 'Andrew', '555-7343': 'Anna', '555-9823': 'Michael', '555-6342':
 'Karen', '555-6677': 'Dan', '555-2222': 'Paul'}
>>> phone.items()
[('555-7632', 'Paul'), ('555-9832', 'Andrew'), ('555-7343', 'Anna'), ('555-9823', 'Michael'),
('555-6342', 'Karen'), ('555-6677', 'Dan'), ('555-2222', 'Paul')]
>>> for (number, name) in phone.items():
        new_phone[name] = number


>>> new_phone
{'Dan': '555-6677', 'Michael': '555-9823', 'Andrew': '555-9832', 'Karen': '555-6342', 'Paul': '555
-2222', 'Anna': '555-7343'}
>>> new_phone = {}
>>> for (number, name) in phone.items():
        if name in new_phone:
                new_phone[name].append(number)
        else:
                new_phone[name] = [number]
```

```
>>> new_phone
{'Dan': ['555-6677'], 'Michael': ['555-9823'], 'Andrew': ['555-9832'], 'Karen': ['555-6342'], 'Paul':
['555-7632', '555-2222'], 'Anna': ['555-7343']}
>>> [3]
[3]
>>> x = 78
>>> [x]
[78]
>>> x = 'hello'
>>> [x]
['hello']
```

```
>>> phone
{'555-7632': 'Paul', '555-9832': 'Andrew', '555-7343':
 'Anna', '555-9823': 'Michael', '555-6342': 'Karen', '55
5-6677': 'Dan', '555-2222': 'Paul'}
>>> for (number, name) in phone.items():
        if name in new_phone:
                new_phone[name].append(number)
        else:
                new_phone[name] = [number]
```