Lecture 8-2

Pythonic Features

Week 8 Wednesday

Miles Chen, PhD

Named Tuples

Named tuples are a quick and simple way to define a new class if the Class definition only contains values and does not require its own methods.

Recall we defined a class Point with the following definition.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

A named tuple can be created that functions in a nearly identical fashion. You will need to import `namedtuple` from the `collections` module

In [1]:
```
from collections import namedtuple
```

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

In [2]:
```
Point = namedtuple('Point', ['x', 'y'])
```

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

In [2]:
```
Point = namedtuple('Point', ['x', 'y'])
```

In [3]:
```
Point
```

Out[3]:
```
__main__.Point
```

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

In [2]:
```
Point = namedtuple('Point', ['x', 'y'])
```

In [3]:
```
Point
```

Out[3]:
```
__main__.Point
```

With our namedtuple defined, we can create instances of it like we would any other class.

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

In [2]:
```
Point = namedtuple('Point', ['x', 'y'])
```

In [3]:
```
Point
```

Out[3]:
```
__main__.Point
```

With our namedtuple defined, we can create instances of it like we would any other class.

In [4]:
```
p = Point(1, 2)
```

Once we have imported namedtuple, we can create a named tuple.

We'll create a Point Class named tuple that contains two values, `x` and `y`.

In [2]:
```
Point = namedtuple('Point', ['x', 'y'])
```

In [3]:
```
Point
```

Out[3]:
```
__main__.Point
```

With our namedtuple defined, we can create instances of it like we would any other class.

In [4]:
```
p = Point(1, 2)
```

In [5]:
```
p
```

Out[5]:
```
Point(x=1, y=2)
```

Now that we have created an instance of the named tuple, we can access the values using dot notation. We can also access values using indexed square-bracket notation as well because it is a tuple.

Now that we have created an instance of the named tuple, we can access the values using dot notation. We can also access values using indexed square-bracket notation as well because it is a tuple.

In [6]:
```
p.x
```

Out[6]:
```
1
```

Now that we have created an instance of the named tuple, we can access the values using dot notation. We can also access values using indexed square-bracket notation as well because it is a tuple.

In [6]:
```
p.x
```

Out[6]:
```
1
```

In [7]:
```
p.y
```

Out[7]:
```
2
```

Now that we have created an instance of the named tuple, we can access the values using dot notation. We can also access values using indexed square-bracket notation as well because it is a tuple.

In [6]:
```
p.x
```

Out[6]:
```
1
```

In [7]:
```
p.y
```

Out[7]:
```
2
```

In [8]:
```
p[0]
```

Out[8]:
```
1
```

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

In [9]:

```
p1 = Point(0, 1)
p2 = Point(3, 4)
p3 = Point(2, 2)
```

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

In [9]:
```
p1 = Point(0, 1)
p2 = Point(3, 4)
p3 = Point(2, 2)
```

In [10]:
```
p1 > p2
```

Out[10]:
```
False
```

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

In [9]:
```
p1 = Point(0, 1)
p2 = Point(3, 4)
p3 = Point(2, 2)
```

In [10]:
```
p1 > p2
```

Out[10]:
```
False
```

In [11]:
```
p1 + p2
```

Out[11]:
```
(0, 1, 3, 4)
```

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

In [9]:
```
p1 = Point(0, 1)
p2 = Point(3, 4)
p3 = Point(2, 2)
```

In [10]:
```
p1 > p2
```

Out[10]:
```
False
```

In [11]:
```
p1 + p2
```

Out[11]:
```
(0, 1, 3, 4)
```

In [12]:
```
l = [p1, p2, p3]
l
```

Out[12]:
```
[Point(x=0, y=1), Point(x=3, y=4), Point(x=2, y=2)]
```

A named tuple will inherit all of the methods associated with tuples such as comparison and "addition"

In [9]:
```
p1 = Point(0, 1)
p2 = Point(3, 4)
p3 = Point(2, 2)
```

In [10]:
```
p1 > p2
```

Out[10]:
```
False
```

In [11]:
```
p1 + p2
```

Out[11]:
```
(0, 1, 3, 4)
```

In [12]:
```
l = [p1, p2, p3]
l
```

Out[12]:
```
[Point(x=0, y=1), Point(x=3, y=4), Point(x=2, y=2)]
```

In [13]:
```
sorted(l)
```

Out[13]:
```
[Point(x=0, y=1), Point(x=2, y=2), Point(x=3, y=4)]
```

If the class definition needs to become more complicated you can define a new class that inherits from the namedtuple.

In [14]:
```
class Uberpoint(Point):
    """A class based on the named tuple Point"""
    # add methods here
```

List comprehensions

List comprehensions allow us to create new lists concisely based on an existing collection

They take the form:

`[expr for val in collection if condition]`

This is basically equivalent to the following loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

In [15]:

```
# make a list of the squares
[x ** 2 for x in range(1, 11)]
```

Out[15]:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [15]:
```
# make a list of the squares
[x ** 2 for x in range(1, 11)]
```

Out[15]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [16]:
```
import numpy as np
np.array([x**2 for x in range(1, 11)])
```

Out[16]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In [15]:
```
# make a list of the squares
[x ** 2 for x in range(1, 11)]
```

Out[15]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [16]:
```
import numpy as np
np.array([x**2 for x in range(1, 11)])
```

Out[16]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In [17]:
```
# square only the odd numbers
[x**2 for x in range(1, 11) if x % 2 == 1]
```

Out[17]:
```
[1, 9, 25, 49, 81]
```

In [15]:
```
# make a list of the squares
[x ** 2 for x in range(1, 11)]
```

Out[15]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [16]:
```
import numpy as np
np.array([x**2 for x in range(1, 11)])
```

Out[16]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In [17]:
```
# square only the odd numbers
[x**2 for x in range(1, 11) if x % 2 == 1]
```

Out[17]:
```
[1, 9, 25, 49, 81]
```

In [18]:
```
# take a list of strings, and write the words that are over 2 characters long in uppercase.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

Out[18]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [19]:
```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[19]:
```
[9, 6, 3, 2, 5, 7]
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [19]:
```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[19]:
```
[9, 6, 3, 2, 5, 7]
```

In [20]:
```
# iterate over a dictionary's items
d = {'a':'apple', 'b':'banana', 'c':'carrots', 'd':'donut', 'e':'eggs'}
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [19]:
```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[19]:
```
[9, 6, 3, 2, 5, 7]
```

In [20]:
```
# iterate over a dictionary's items
d = {'a':'apple', 'b':'banana', 'c':'carrots', 'd':'donut', 'e':'eggs'}
```

In [21]:
```
list(d.items())  # recall what dict.items() returns: a list of tuples
```

Out[21]:
```
[('a', 'apple'),
 ('b', 'banana'),
 ('c', 'carrots'),
 ('d', 'donut'),
 ('e', 'eggs')]
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [19]:
```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[19]:
```
[9, 6, 3, 2, 5, 7]
```

In [20]:
```
# iterate over a dictionary's items
d = {'a':'apple', 'b':'banana', 'c':'carrots', 'd':'donut', 'e':'eggs'}
```

In [21]:
```
list(d.items())  # recall what dict.items() returns: a list of tuples
```

Out[21]:
```
[('a', 'apple'),
 ('b', 'banana'),
 ('c', 'carrots'),
 ('d', 'donut'),
 ('e', 'eggs')]
```

In [22]:
```
['%s is for %s' % (key, value) for key, value in d.items() if key not in ('b', 'd') ]
```

Out[22]:
```
['a is for apple', 'c is for carrots', 'e is for eggs']
```

Dictionary Comprehensions

A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```

Dictionary Comprehensions

A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```

In [23]:

```
# create a dictionary, where the key is the word capitalized, and the value is the length of the wor
fruits = ['apple', 'mango', 'banana', 'cherry']
{f.capitalize():len(f) for f in fruits}
```

Out[23]:

```
{'Apple': 5, 'Mango': 5, 'Banana': 6, 'Cherry': 6}
```

In [24]:

```
# create a dictionary where the key is the index, and the value is the string in the strings list.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [24]:
```
# create a dictionary where the key is the index, and the value is the string in the strings list.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [25]:
```
list(enumerate(strings))  # enumerate produces a collection of tuples, with index and value
```

Out[25]:
```
[(0, 'a'), (1, 'as'), (2, 'bat'), (3, 'car'), (4, 'dove'), (5, 'python')]
```

In [24]:
```
# create a dictionary where the key is the index, and the value is the string in the strings list.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [25]:
```
list(enumerate(strings))  # enumerate produces a collection of tuples, with index and value
```

Out[25]:
```
[(0, 'a'), (1, 'as'), (2, 'bat'), (3, 'car'), (4, 'dove'), (5, 'python')]
```

In [26]:
```
index_map = {index:val for index, val in enumerate(strings)}
index_map
```

Out[26]:
```
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [27]:

```
# note that enumerate returns tuples in the order (index, val)
# in the creation of a dictionary, you can swap those positions
# and even apply functions to them

# We create a dictionary where the key is the string, and the value is the index in the strings list
loc_mapping = {val : index for index, val in enumerate(strings)}
loc_mapping
```

Out[27]:

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

In [27]:

```
# note that enumerate returns tuples in the order (index, val)
# in the creation of a dictionary, you can swap those positions
# and even apply functions to them

# We create a dictionary where the key is the string, and the value is the index in the strings list
loc_mapping = {val : index for index, val in enumerate(strings)}
loc_mapping
```

Out[27]:

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

In [28]:

```
index_map['a']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-28-a566f0150b5c> in <module>
----> 1 index_map['a']

KeyError: 'a'
```

In [29]:
```
loc_mapping['a']
```

Out[29]:
```
0
```

In [29]:
```
loc_mapping['a']
```

Out[29]:
```
0
```

In [30]:
```
# combine dictionaries with kwargs
dd = {**loc_mapping, **index_map}
print(dd)
```

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5, 0: 'a', 1: 'as', 2:
'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [31]:

```
# even better... use dict.update(). This modifies the dictionary in place
loc_mapping.update(index_map)
loc_mapping
```

Out[31]:

```
{'a': 0,
 'as': 1,
 'bat': 2,
 'car': 3,
 'dove': 4,
 'python': 5,
 0: 'a',
 1: 'as',
 2: 'bat',
 3: 'car',
 4: 'dove',
 5: 'python'}
```

Generator Expressions

Generator Expressions are similar to List comprehensions.

You create them with parentheses instead of square brackets.

The result is a generator object. You can access values in the generator using `next()`

In [32]:

```
g = (n**2 for n in range(12))
```

In [32]:
```
g = (n**2 for n in range(12))
```

In [33]:
```
g
```

Out[33]:
```
<generator object <genexpr> at 0x0000019077A56F48>
```

In [32]:
```
g = (n**2 for n in range(12))
```

In [33]:
```
g
```

Out[33]:
```
<generator object <genexpr> at 0x0000019077A56F48>
```

In [34]:
```
next(g)
```

Out[34]:
```
0
```

In [32]:
```
g = (n**2 for n in range(12))
```

In [33]:
```
g
```

Out[33]:
```
<generator object <genexpr> at 0x0000019077A56F48>
```

In [34]:
```
next(g)
```

Out[34]:
```
0
```

In [35]:
```
next(g)
```

Out[35]:
```
1
```

In [32]:
```
g = (n**2 for n in range(12))
```

In [33]:
```
g
```

Out[33]:
```
<generator object <genexpr> at 0x0000019077A56F48>
```

In [34]:
```
next(g)
```

Out[34]:
```
0
```

In [35]:
```
next(g)
```

Out[35]:
```
1
```

In [36]:
```
next(g)
```

Out[36]:
```
4
```

In [32]:
```
g = (n**2 for n in range(12))
```

In [33]:
```
g
```

Out[33]:
```
<generator object <genexpr> at 0x0000019077A56F48>
```

In [34]:
```
next(g)
```

Out[34]:
```
0
```

In [35]:
```
next(g)
```

Out[35]:
```
1
```

In [36]:
```
next(g)
```

Out[36]:
```
4
```

In [37]:
```
next(g)
```

Out[37]:
```
9
```

In [38]:

```
for val in g:
    print(val)
```

```
16
25
36
49
64
81
100
121
```

In [38]:

```
for val in g:
    print(val)
```

```
16
25
36
49
64
81
100
121
```

In [39]:

```
next(g) # calling next after it has run out of iterations will result in an error
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-39-35efc4ce126e> in <module>
----> 1 next(g) # calling next after it has run out of iterations will result in
 an error

StopIteration:
```

List Comprehension vs Generator Expressions in Python

A Key difference between a list comprehension and a generator is that the generator is lazy.

The list comprehension will evaluate the entire sequence of iterations. The generator will only generate the next value when it is asked to do so.

Depending on the expression that needs to be evaluated, you may prefer to use a generator over the list comprehension.

The following examples are from: https://code-maven.com/list-comprehension-vs-generator-expression

In [40]:

```
l = [n*2 for n in range(1000)] # List comprehension
g = (n*2 for n in range(1000))  # Generator expression
```

In [40]:
```
l = [n*2 for n in range(1000)] # List comprehension
g = (n*2 for n in range(1000))  # Generator expression
```

In [41]:
```
print(type(l))  # 'list'
print(type(g))  # 'generator'
```

```
<class 'list'>
<class 'generator'>
```

In [40]:
```
l = [n*2 for n in range(1000)] # List comprehension
g = (n*2 for n in range(1000))  # Generator expression
```

In [41]:
```
print(type(l))  # 'list'
print(type(g))  # 'generator'
```

```
<class 'list'>
<class 'generator'>
```

In [42]:
```
import sys
print(sys.getsizeof(l))  # more space in memory
print(sys.getsizeof(g))  # less space in memory
```

```
9024
120
```

In [43]:

```
# cannot access values in a generator by index
print(l[4])   # 8
print(g[4])   # TypeError: 'generator' object is not subscriptable
```

```
8
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-e29ce47c972b> in <module>
      1 # cannot access values in a generator by index
      2 print(l[4])   # 8
----> 3 print(g[4])   # TypeError: 'generator' object is not subscriptable

TypeError: 'generator' object is not subscriptable
```

In [44]:
```
g
```

Out[44]:
```
<generator object <genexpr> at 0x00000190779DC948>
```

In [44]:
```
g
```

Out[44]:
```
<generator object <genexpr> at 0x00000190779DC948>
```

In [45]:
```
sum(g) # sum demands that all elements of g be calculated so the generator evaluates all of them
```

Out[45]:
```
999000
```

In [44]:
```
g
```

Out[44]:
```
<generator object <genexpr> at 0x00000190779DC948>
```

In [45]:
```
sum(g) # sum demands that all elements of g be calculated so the generator evaluates all of them
```

Out[45]:
```
999000
```

In [46]:
```
sum(l) # the list already has the values in memory ready to be summed
```

Out[46]:
```
999000
```

map and lambda functions

The `map(function, iterable)` function takes a particular function and maps it to each element of an iterable. The object it returns is a map object which itself is iterable.

A lambda function allows you to create and use a new short function without having to formally define it.

map and lambda functions

The `map(function, iterable)` function takes a particular function and maps it to each element of an iterable. The object it returns is a map object which itself is iterable.

A lambda function allows you to create and use a new short function without having to formally define it.

In [47]:
```
# the module re is used for regular expressions
import re
```

map and lambda functions

The `map(function, iterable)` function takes a particular function and maps it to each element of an iterable. The object it returns is a map object which itself is iterable.

A lambda function allows you to create and use a new short function without having to formally define it.

In [47]:
```
# the module re is used for regular expressions
import re
```

In [48]:
```
# re.sub substitutes one pattern of text with another.
# Here we define a function that replaces multiple instances of white space (\s+) with one space:
def replace_space(x):
    return(re.sub('\s+', ' ', x))
```

map and lambda functions

The `map(function, iterable)` function takes a particular function and maps it to each element of an iterable. The object it returns is a map object which itself is iterable.

A lambda function allows you to create and use a new short function without having to formally define it.

In [47]:
```
# the module re is used for regular expressions
import re
```

In [48]:
```
# re.sub substitutes one pattern of text with another.
# Here we define a function that replaces multiple instances of white space (\s+) with one space:
def replace_space(x):
    return(re.sub('\s+', ' ', x))
```

In [49]:
```
replace_space('Hello     Alabama ')
```

Out[49]:
```
'Hello Alabama '
```

In [50]:

```
text = ['Hello     Alabama', 'Georgia!', 'Georgia', 'georgia',
          'FlOrIda', 'south  carolina##', 'West virginia?']
```

In [50]:
```
text = ['Hello     Alabama', 'Georgia!', 'Georgia', 'georgia',
          'FlOrIda', 'south  carolina##', 'West virginia?']
```

In [51]:
```
# we can use the map function to map the replace_space() function to each element of the list text
for item in map(replace_space, text):
    print(item)
```

```
Hello Alabama
Georgia!
Georgia
georgia
FlOrIda
south carolina##
West virginia?
```

In [50]:
```
text = ['Hello     Alabama', 'Georgia!', 'Georgia', 'georgia',
          'FlOrIda', 'south  carolina##', 'West virginia?']
```

In [51]:
```
# we can use the map function to map the replace_space() function to each element of the list text
for item in map(replace_space, text):
    print(item)
```

```
Hello Alabama
Georgia!
Georgia
georgia
FlOrIda
south carolina##
West virginia?
```

In [52]:
```
# we can also put the map results inside a list
list(map(replace_space, text))
```

Out[52]:
```
['Hello Alabama',
 'Georgia!',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south carolina##',
 'West virginia?']
```

In [53]:

```
# however, because the code for the function is so short, it might be easier to just create
# a quick function without a formal name. These 'anonymous' functions are also known as lambda funct
list(map(lambda x: re.sub('\s+',' ', x), text))
```

Out[53]:

```
['Hello Alabama',
 'Georgia!',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south carolina##',
 'West virginia?']
```

In [53]:
```
# however, because the code for the function is so short, it might be easier to just create
# a quick function without a formal name. These 'anonymous' functions are also known as lambda funct
list(map(lambda x: re.sub('\s+',' ', x), text))
```

Out[53]:
```
['Hello Alabama',
 'Georgia!',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south carolina##',
 'West virginia?']
```

In [54]:
```
# here's a similar function that turns the text into title case.
list(map(lambda string: string.title(), text))
```

Out[54]:
```
['Hello     Alabama',
 'Georgia!',
 'Georgia',
 'Georgia',
 'Florida',
 'South  Carolina##',
 'West Virginia?']
```

lambda functions are written in the form:

```
lambda argument1, argument2, etc: expression to return
```

lambda functions are written in the form:

```
lambda argument1, argument2, etc: expression to return
```

In [55]:

```
# lambda functions can also accept multiple arguments
# if you use it with map, you'll need to provide a list for each argument
list(map(lambda x, y: x + y, [1, 2, 3], [100, 200, 300]))
```

Out[55]:

```
[101, 202, 303]
```