香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 5 List, Dictionary, and Tuple

Prof. Yunming XIAO

School of Data Science

Outline

- List
- Dictionary
- Tuple

Outline

- **List**
- Dictionary
- Tuple

List is a kind of collection

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many variables around in one convenient package

What is not a collection

- Most of our variables have only one value in them – when we put a new value in the variable, the old value will be over-written

```
>>> x = 2
>>> print(x)
2
>>> x = 4
>>> print(x)
4
```

List constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas

```
>>> print([1, 24, 76])
[1, 24, 76]
>>> print(['red', 'yellow', 'blue'])
['red', 'yellow', 'blue']
>>> print(['red', 24, 98.6])
['red', 24, 98.6]
```

- A list element can be any Python object – even another list

```
>>> print([1, [5, 6], 7])
[1, [5, 6], 7]
```

- A list can be empty

```
>>> print([])
[]
```

List and definite loop – best pal

Program

```
friends = ["Tom", "Jerry", "Bat"]
for friend in friends:
    print("Happy new year", friend)
print("Finished")
```

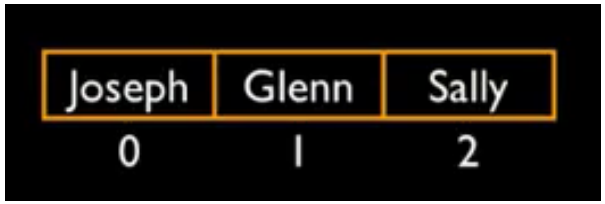
Outputs

```
lecture3 — yur
➜ lecture3 python3 lecture3-for2.py
Happy new year Tom
Happy new year Jerry
Happy new year Bat
Finished
➜ lecture3
```

Looking inside lists

- Just like strings, we can access any single element in a list using an index specified in square bracket

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(friends[1])
Glenn
```



Lists are mutable

- Strings are “immutable” – we cannot change the contents of a string unless we make a new string

```
>>> fruit = "Banana"
>>> fruit[0] = 'b'
Traceback (most recent call last):
  File "<python-input-25>", line 1, in <module>
    fruit[0] = 'b'
    ~~~~~^^^
TypeError: 'str' object does not support item assignment
```

- Lists are “mutable” – we can change an element of a list using index operator

```
>>> lotto = [2, 14, 26, 41, 63]
>>> lotto[0] = 3
>>> print(lotto)
[3, 14, 26, 41, 63]
```

How long is a list?

- The len() function takes a list as input and returns the number of elements in that list

- Actually len() tells us the number of elements in any sequence (e.g. strings)

```
>>> greet = "Hello Bob"
>>> print(len(greet))
9
>>> x = [1, 2, "Joe", 99]
>>> print(len(x))
4
```

Function range()

- The range() function returns a list of numbers

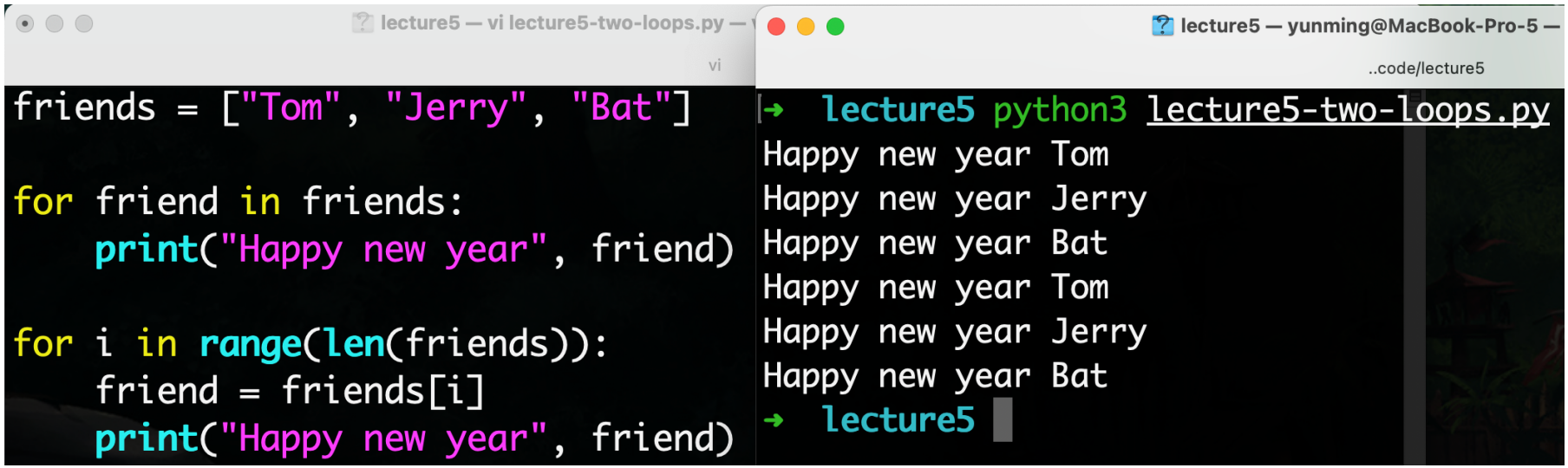
- We can construct an index loop using for and an integer iterator

```
>>> x = range(4)
>>> x
range(0, 4)
>>> x[0]
0
>>> x[1]
1
>>> x[3]
3
>>> x[4]
Traceback (most recent call last):
  File "<python-input-39>", line 1, in <module>
    x[4]
    ~^^^
IndexError: range object index out of range
```

A tale of two loops

Program

Outputs



Concatenating lists using “+”

- Similar to strings, we can add two existing lists together to create a new list

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3]
```

Lists can be sliced using ":"

- Remember: similar to strings, the second number is "up to but no including"

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41, 12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

List methods

```
>>> x = []
>>> type(x)
<class 'list'>
>>> dir(x)
['__add__', '__class__', '__class_getitem__', '__contains__', '__de
lattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '_
_gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_sub
class__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__
ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rev
ersed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '
__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', '
extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Building a list from scratch

- We can create an empty list using list(), and then add elements using append() method

- The list stays in order, and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append("book")
>>> stuff.append(99)
>>> stuff
['book', 99]
>>> stuff.append("cookie")
>>> stuff
['book', 99, 'cookie']
>>>
```

Is something in a list

- Python provides two operators to check whether an item is in a list: in and not in
- These are logical operators that return True or False
- They do not modify the list

```
>>> stuff
['book', 99, 'cookie']
>>> 9 in stuff
False
>>> 99 in stuff
True
>>> "cookie" in stuff
True
```

A list is an ordered sequence

- A list can hold many items and keeps them in the order until we do something to change the order
- A list can be sorted (i.e. change the order)
- The sort() method means "sort yourself"

```
>>> friends = ["Tom", "Jerry", "Bat"]
>>> friends.sort()
>>> friends
['Bat', 'Jerry', 'Tom']
>>>
>>> numbers = [1, 2, 5, 100, 32, 7]
>>> numbers.sort()
>>> numbers
[1, 2, 5, 7, 32, 100]
```

Built-in functions and lists

- There are a number of functions built into Python that take lists as inputs
- Remember the loops we built? These are much simpler

```
>>> numbers = [3, 41, 12, 9, 74, 15]
>>> len(numbers)
6
>>> max(numbers)
74
>>> min(numbers)
3
>>> sum(numbers)
154
>>> sum(numbers)/len(numbers)
25.666666666666668
```

Averaging with a list

```
total = 0
count = 0
while True:
    inp = input("Enter a number:")
    if inp == "done":
        break
    value = float(inp)
    total = total + value
    count = count + 1

avg = total / count
print("The average is", avg)
```

Practice

- Write a program to instruct the user to input several numbers and calculate their average using list methods

Best friends: strings and lists

- Use the split() method to break up a string into a list of strings
- We think of these as words
- We can access a particular word or loop through all the words

```
>>> str = "Catch me if you can"
>>> words = str.split()
>>> words
['Catch', 'me', 'if', 'you', 'can']
>>> len(words)
5
>>> for w in words:
...     print(w)
...
Catch
me
if
you
can
```

Best friends: strings and lists

- When you do not specify a delimiter, multiple spaces are treated like “one” delimiter

- You can specify what delimiter character to use in splitting

```
>>> line = "A lot       of spaces"
>>> etc = line.split()
>>> etc
['A', 'lot', 'of', 'spaces']
>>>
>>> line = "first;second;third"
>>> things = line.split()
>>> things
['first;second;third']
>>>
>>> things = line.split(";")
>>> things
['first', 'second', 'third']
```

Practice

- The header of an email takes the following format:

From professor.xman@cuhk.edu.cn Sun Oct 5 09:14:16 2025

For a given email header, write a program to find out the domain of email address, and the month in which this email is sent

The double split pattern

- Sometimes we split a line one way, and then grab one piece of the line and split it again

From professor.xman@cuhk.edu.cn Sun Oct 5 09:14:16 2025

```
words = header.split()
address = words[1].split('@')
```

Outline

- List
- **Dictionary**
- Tuple

A story of two collections

- List: a linear collection of values that stay in order
- Dictionary: a “bag” of values, each with its own label

Dictionary

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Associative arrays – Perl/PHP
  - Properties or Map or HashMap – Java
  - Property Bag – C#/.Net

Dictionary

- Lists index their entries based on the position in the list
- Dictionaries are like bags – no order
- We index the elements we put in the dictionary with a "lookup tag"

```
>>> purse = dict()
>>> purse["money"] = 12
>>> purse["candy"] = 3
>>> print(purse)
{'money': 12, 'candy': 3}
>>> print(purse["candy"])
3
>>> purse["candy"] = purse["candy"] + 2
>>> print(purse)
{'money': 12, 'candy': 5}
>>>
>>> purse[3] = 77
>>> print(purse)
{'money': 12, 'candy': 5, 3: 77}
```

Dictionary

```
>>> purse = dict()
>>> purse["money"] = 12
>>> purse["candy"] = 3
>>> purse["tissues"] = 75
>>> print(purse)
{'money': 12, 'candy': 3, 'tissues': 75}
```

List vs. dictionary

- Dictionaries are similar to lists, except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 183
>>> print(ddd)
{'age': 21, 'course': 183}
>>> ddd['age'] = 23
>>> print(ddd)
{'age': 23, 'course': 183}
```

List vs. dictionary

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 183
>>> print(ddd)
{'age': 21, 'course': 183}
>>> ddd['age'] = 23
>>> print(ddd)
{'age': 23, 'course': 183}
```

Dictionary literals (constants)

- Dictionary literals use curly braces and have list of key:value pairs
- You can make an empty dictionary using empty curly braces

```
>>> d = {"chuck": 1, "fred": 42, "jan": 100}
>>> print(d)
{'chuck': 1, 'fred': 42, 'jan': 100}
>>>
>>> dd = {}
>>> print(dd)
{}
```

Counting with a dictionary

- A common use of dictionary is counting how often we "see" something

```
>>> attendance = {}
>>> attendance["Yunming"] = 1
>>> attendance["Xiaozhuang"] = 0
>>> print(attendance)
{'Yunming': 1, 'Xiaozhuang': 0}
>>> attendance["Yunming"] += 1
>>> print(attendance)
{'Yunming': 2, 'Xiaozhuang': 0}
```

Dictionary tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> csc1001 = dict()
>>> print(csc1001['grad-stu'])
Traceback (most recent call last):
  File "<python-input-71>", line 1, in <module>
    print(csc1001['grad-stu'])
          ~~~~~~~^^^^^^^^^^^^
KeyError: 'grad-stu'
>>> 'grad-stu' in csc1001
False
```

Practice

- Write a program to instruct the user to continuously input some words, and use dictionary to count how many times a word has been inputted before.

The get() method

- This pattern of checking to see if a key is already in a dictionary, and assuming a default value if the key is not there is so common, that there is a method called get() that does this for us

```
>>> counts = {'aaa': 1, 'bbb': 2, 'ccc': 3}
>>> counts.get('aaa', 0)
1
>>> counts.get('ddd', 0)
0
```

Practice

- Write a program to instruct the user to input a line of texts, and use dictionary to count how many times a word has been seen in this line. You should use the get() method in this program.

Definite loops and dictionaries

- Even though dictionaries are not stored in order, we can write a for loop that goes through all elements in a dictionary – actually it goes through all the keys in that dictionary and looks up the values

```
>>> counts = {'aaa': 1, 'bbb': 2, 'ccc': 3}
>>> for key in counts:
...     print(key, counts[key])
...
aaa 1
bbb 2
ccc 3
```

Retrieving lists of keys and values

- You can get a list of keys, values or items (both) from a dictionary

```
>>> counts = {'aaa': 1, 'bbb': 2, 'ccc': 3}
>>> list(counts)
['aaa', 'bbb', 'ccc']
>>> list(counts.keys())
['aaa', 'bbb', 'ccc']
>>> list(counts.values())
[1, 2, 3]
>>> list(counts.items())
[('aaa', 1), ('bbb', 2), ('ccc', 3)]
```

Bonus: two iteration variables

- We loop through the key-value pairs in a dictionary using two iteration variables

- Each iteration, the first variable is the key, and the second variable is the corresponding value for the key

```
>>> counts = {'aaa': 1, 'bbb': 2, 'ccc': 3}
>>> for key, value in counts.items():
...     print(key, value)
...
aaa 1
bbb 2
ccc 3
```

Outline

- List
- Dictionary
- **Tuple**

Tuples

- Tuples are another type of sequence that function more like a list – they have elements which are indexed starting from 0

But, tuples are “immutable”

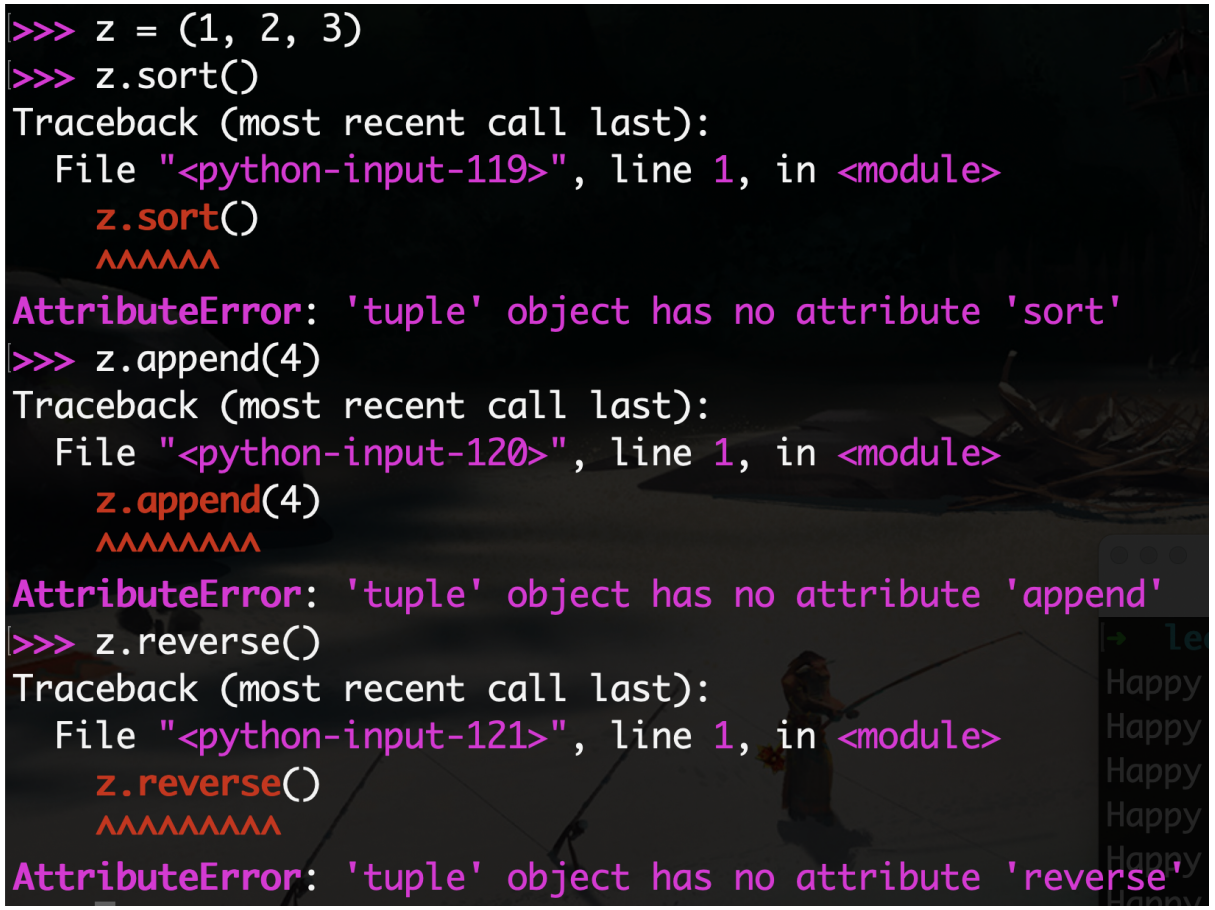
- Unlike a list, once you create a tuple, you cannot change its contents – similar to a string

```
>>> x = [1, 2, 3]
>>> x[0] = 10
>>> print(x)
[10, 2, 3]
```

```
>>> y = "abc"
>>> y[0] = "z"
Traceback (most recent call last):
  File "<python-input-114>", line 1, in <module>
    y[0] = "z"
    ~^^^
TypeError: 'str' object does not support item assignment
```

```
>>> z = (1, 2, 3)
>>> z[0] = 10
Traceback (most recent call last):
  File "<python-input-116>", line 1, in <module>
    z[0] = 10
    ~^^^
TypeError: 'tuple' object does not support item assignment
```

Things that you cannot do with tuples



Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists

- In our program when we are making “temporary variables” we prefer tuples over lists

```
L = list(range(10_000_000))
T = tuple(range(10_000_000))
```

```
List creation: 1.7211s
Tuple creation: 1.0783s
```

Tuples and dictionaries

- The item() method in dictionaries returns a list of (key, value) tuples

```
>>> d = dict()
>>> d['aaa'] = 1
>>> d['bbb'] = 2
>>> for k, v in d.items():
...     print(k, v)
...
aaa 1
bbb 2
```

```
>>> tup = tups[0]
Traceback (most recent call last):
  File "<python-input-146>", line 1, in <module>
    tup = tups[0]
          ~~~~^^^
TypeError: 'dict_items' object is not subscriptable
>>> tups = d.items()
>>> print(tups)
dict_items([('aaa', 1), ('bbb', 2)])
>>>
>>> tup = list(tups)[0]
>>> tup
('aaa', 1)
>>> type(tup)
<class 'tuple'>
```

Tuples are comparable

- The comparison operators work with tuples and other sequences if the first item is equal. Python goes on to the next element, until it finds the elements which are different

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 20) < (0, 3, 4)
True
>>> ("Jones", "Sally") < ("Jones", "Fred")
False
```

Sorting lists of tuples

- We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary

- First we sort the dictionary by the key using the items() method

```
>>> d = {'aaa': 10, 'bbb': 100, 'c': 1000}
>>> t = d.items()
>>> t = list(t)
>>> t
[('aaa', 10), ('bbb', 100), ('c', 1000)]
>>> t.sort()
>>> t
[('aaa', 10), ('bbb', 100), ('c', 1000)]
```

Using sorted()

- We can do this even more efficiently using a built-in function sorted() which takes a sequence as a parameter and returns a sorted sequence

```
>>> d = {'aaa': 10, 'bbb': 100, 'c': 1000}
>>> t = sorted(list(d.items()))
>>> t
[('aaa', 10), ('bbb', 100), ('c', 1000)]
```

Practice

- Write a program, which sorts the elements of a dictionary by the value of each element

Sort by values instead of key

- If we could construct a list of tuples of the form (key, value) we could sort by value
- We do this with a for loop that creates a list of tuples

```
>>> d = {'aaa': 10, 'bbb': 20, 'ccc': 30}
>>> tmp = list
KeyboardInterrupt
>>> d = {'aaa': 10, 'bbb': 2, 'ccc': 3}
>>> tmp = list()
>>> for k, v in d.items():
...     tmp.append((v, k))
...
>>> print(tmp)
[(10, 'aaa'), (2, 'bbb'), (3, 'ccc')]
>>> tmp.sort()
>>> print(tmp)
[(2, 'bbb'), (3, 'ccc'), (10, 'aaa')]
```

Example: finding 10 most common words in a file

```
file_handle = open('my_novel.txt', 'r')
counts = {}
for line in file_handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

lst = list()
for key, value in counts.items():
    lst.append((value, key))

lst.sort(reverse = True)

for value, key in lst[:10]:
    print(key, value)
```

A tale of two sequences

```
>>> l = list()
>>> dir(l)
['__add__', '__class__', '__class_getitem__', '__contains__', '__dela
ttr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__'
, '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__'
, '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '_
_rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__s
ubclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index'
, 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
>>> t = tuple()
>>> dir(t)
['__add__', '__class__', '__class_getitem__', '__contains__', '__dela
ttr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__ge
tattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt
__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__
', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__'
, '__str__', '__subclasshook__', 'count', 'index']
```

Practice

- Find the differences in the APIs of list and tuple. Tip: find APIs that (1) list has but tuple doesn't, and (2) list doesn't have but tuple does

Thanks