Python Basics

for Data Analysis

By Armin Khayati

Python Basics : Built-in Data Structures, Functions, and Files

- Python's workhorse data structures are : tuples, lists, dicts, and sets.

- Python's data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

Python Basics : Tuple

- Python’s workhorse data structures are : tuples, lists, dicts, and sets.
- Python’s data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

Python Basics : Tuple

- A tuple is a fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values.

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```

- When you're defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Python Basics : Tuple

- You can convert any sequence or iterator to a tuple by invoking tuple.

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

- Elements can be accessed with square brackets [values] as with most other sequence types.

```
In [8]: tup[0]
Out[8]: 's'
```

Python Basics : Tuple

- While the objects stored in a tuple may be mutable themselves, once the tuple is created it’s not possible to modify which object is stored in each slot.

```
In [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

Python Basics : Tuple

- If an object inside a tuple is mutable, such as a list, you can modify it in-place.

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

- You can concatenate tuples using the + operator to produce longer tuples.

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Python Basics : Tuple

- Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple.

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Python Basics : Unpacking tuples

- If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the right-hand side of the equals sign.

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

- Even sequences with nested tuples can be unpacked:

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

Python Basics : Unpacking tuples

- Using this functionality you can easily swap variable names, a task which in many languages might look like

```
tmp = a
a = b
b = tmp
```

- But, in Python, the swap can be done like this

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

Python Basics : Unpacking tuples

- A common use of variable unpacking is iterating over sequences of tuples or lists.

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
    ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

- Another common use is returning multiple values from a function.

Python Basics : Unpacking tuples

- Another application of tuple unpacking is “pluking” few elements from the beginning of a tuple.
- This uses the special syntax `*rest`
- It is also used in function signatures to capture an arbitrarily long list of positional arguments
- .

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

Python Basics : Tuple Methods

- As a matter of convention, many Python programmers will use the underscore (_) for unwanted variables.

```
In [33]: a, b, *_ = values
```

- Since the size and contents of a tuple cannot be modified, it is very light on instance methods.
- A particularly useful one (also available on lists) is `count` , which counts the number of occurrences of a value.

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

Python Basics : List

- In contrast with tuples, lists are variable-length and their contents can be modified in-place.
- You can define them using square brackets [ ] or using the `list` type function

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

Python Basics : List

- Lists and tuples are semantically similar (though tuples cannot be modified) and can be used interchangeably in many functions.
- The list function is frequently used in data processing as a way to materialize an iterator or generator expression

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python Basics : Adding and removing elements

- Elements can be appended to the end of the list with the `append` method.
- Using `insert` you can insert an element at a specific location in the list.
- The insertion index must be between 0 and the length of the list, inclusive.

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']

In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

Python Basics : Adding and removing elements

- The inverse operation to `insert` is `pop` , which removes and returns an element at a particular index.

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

Python Basics : Adding and removing elements

- Elements can be removed by value with remove , which locates the first such value and removes it from the last.

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

Python Basics : Adding and removing elements

- If performance is not a concern, by using append and remove , you can use a Python list as a perfectly suitable “multiset” data structure.
- To check if a list contains a value using the `in` keyword.

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

- The keyword `not` can be used to negate in

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

Python Basics : Concatenating and combining lists

- Similar to tuples, adding two lists together with '`+`' concatenates them.
- If you have a list already defined, you can append multiple elements to it using the `extend` method.

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]

In [58]: x = [4, None, 'foo']

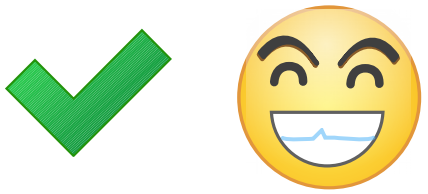
In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

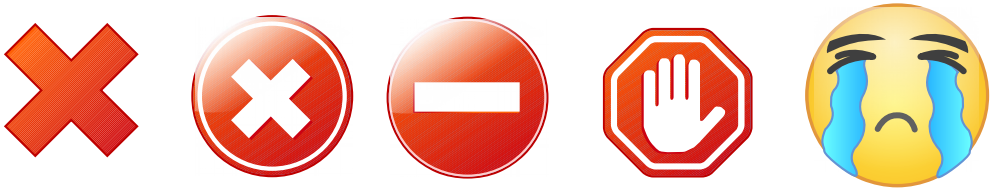
Python Basics : Concatenating and combining lists

- Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over.
- Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable.

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```



```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```



Python Basics : Sorting

- You can sort a list in-place (without creating a new object) by calling its sort function.

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```

Python Basics : Sorting

- sort has a few good options such as the ability to pass a secondary sort key—that is, a function that produces a value to use to sort the objects.

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

Python Basics : Binary search and maintaining a sorted list

- The built-in `bisect` module implements binary search and insertion into a sorted list.
- `bisect.bisect` finds the location where an element should be inserted to keep it sorted.
- `bisect.insort` actually inserts the element into that location

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6
```

```
In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Python Basics : Slicing

- You can select sections of most sequence types by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator [ ]

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

Python Basics : Slicing

- Slices can also be assigned to with a sequence

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

- While the element at the start index is included, the stop index is not included, so that the number of elements in the result is

`stop - start`.

Python Basics : Slicing

- Either the `start` or `stop` can be omitted, in which case they default to the start of the sequence and the end of the sequence.

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

- Negative indices slice the sequence relative to the end.

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

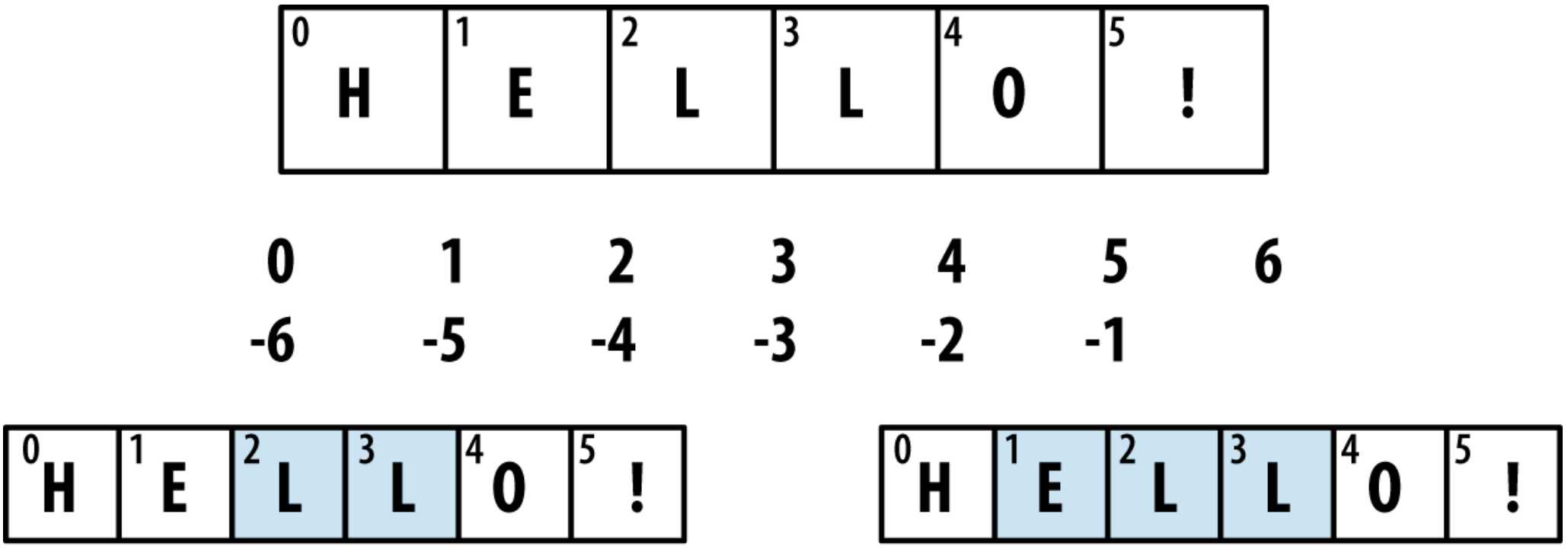
Python Basics : Slicing

- A step can also be used after a second colon to, say, take every other element.
- A clever use of this is to pass -1 , which has the useful effect of reversing a list or tuple.

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]

In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Python Basics : Slicing



Python Basics : Built-in Sequence Functions

- Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity

Python Basics : enumerate

- It’s common when iterating over a sequence to want to keep track of the index of the current item.
- A do-it-yourself approach would look like.

```
i = 0
for value in collection:
    # do something with value
    i += 1
```

- Python has a built-in function, enumerate , which returns a sequence of (i, value) tuples

```
for i, value in enumerate(collection):
    # do something with value
```

Python Basics : enumerate

- When you are indexing data, a helpful pattern that uses enumerate is computing a `dict mapping` the values of a sequence (which are assumed to be unique) to their locations in the sequence.

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

Python Basics : sorted

- The sorted function returns a new sorted list from the elements of any sequence.
- The sorted function accepts the same arguments as the sort method on lists.

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

Python Basics : zip

- `zip` “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples.
- `zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the shortest sequence.
- A very common use of zip is simultaneously iterating over multiple sequences, possibly also combined with enumerate.

Python Basics : zip

- ```
  In [89]: seq1 = ['foo', 'bar', 'baz']

  In [90]: seq2 = ['one', 'two', 'three']

  In [91]: zipped = zip(seq1, seq2)

  In [92]: list(zipped)
  Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
  ```
- ```
  In [93]: seq3 = [False, True]

  In [94]: list(zip(seq1, seq2, seq3))
  Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
  ```
- ```
  In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
     ....:     print('{0}: {1}, {2}'.format(i, a, b))
     ....:
  0: foo, one
  1: bar, two
  2: baz, three
  ```
  ```

Python Basics : zip

- Given a “zipped” sequence, zip can be applied in a clever way to “unzip” the sequence.
- Another way to think about this is converting a list of rows into a list of columns.
- The syntax, which looks a bit magical, is

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
    ....:                ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

Python Basics : reversed

- `reversed` iterates over the elements of a sequence in reverse order.

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

- Keep in mind that reversed is a generator, so it does not create the reversed sequence until materialized (e.g., with `list` or a `for` loop).

Python Basics : dict

- A more common name for it is `hash map` or `associative array`.
- One approach for creating one is to use curly braces `{ }` and colons to separate keys and values.
- `key` and `value` are Python objects.

```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Python Basics : dict

- You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple.

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

Python Basics : dict

- You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value.

```
In [107]: 'b' in d1
Out[107]: True
```

- You can delete values either using the `del` keyword or the `pop` method

Python Basics : dict

```
In [109]: d1
Out[109]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}

In [110]: d1['dummy'] = 'another value'
```

```
In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}

In [112]: del d1[5]
```

```
In [113]: d1
Out[113]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

Python Basics : dict

- The `keys` and `values` method give you iterators of the dict's keys and values.
- These functions output the keys and values in the same order.

```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

Python Basics : dict

- You can merge one dict into another using the `update` method.
- The update method changes dicts in-place, so any existing keys in the data passed to update will have their old values discarded.

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

Python Basics : Creating dicts from sequences

- It’s common to occasionally end up with two sequences that you want to pair up element-wise in a dict.
- As a firs/t cut, you might write code like this.

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

- Since a dict is essentially a collection of 2-tuples, the dict function accepts a list of 2-tuples

```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Python Basics : Default values

- It's very common to have logic like this.

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

- Thus, the dict methods `get` and `pop` can take a default value to be returned.

```
value = some_dict.get(key, default_value)
```

- `get` by default will return `None` if the `key` is not present, while `pop` will raise an `exception`.
- With setting values, a common case is for the values in a dict to be other collections, like lists.

Python Basics : Default values

- Imagine categorizing a list of words by their first letters as a dict of lists.

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [124]: by_letter = {}
In [125]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:

In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

Python Basics : Default values

- The `setdefault` dict method is for precisely this purpose. The preceding for loop can be rewritten as

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

- The built-in collections module has a useful class, `defaultdict` , which makes this even easier.

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

- To create one, you pass a type or function for generating the default value for each slot in the dict.

Python Basics : Valid dict key types

- While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too)
- The technical term here is `hashability`.
- You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function
- To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can

Python Basics : Valid dict key types

```
In [127]: hash('string')
Out[127]: 5023931463650008331

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # fails because lists are mutable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'


In [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(1, 2, 3): 5}
```

Python Basics : Set

- A set is an unordered collection of unique elements.
- You can think of them like dicts,but keys only, no values.
- A set can be created in two ways: via the `set` function or via a `set literal` with curly braces { }

```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```

Python Basics : Set

- Sets support mathematical set operations like union, intersection, difference, and symmetric difference.
- The union of two sets can be computed with either the `union` method or the `|` binary operator.
- The intersection of two sets can be computed with either the `&` operator or the `intersection` method.

Python Basics : Set

```
In [135]: a = {1, 2, 3, 4, 5}

In [136]: b = {3, 4, 5, 6, 7, 8}

In [137]: a.union(b)
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}

In [138]: a | b
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}

In [139]: a.intersection(b)
Out[139]: {3, 4, 5}

In [140]: a & b
Out[140]: {3, 4, 5}
```

Python Basics :
Python common set operations

Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to the set `a`
`a.clear()`	N/A	Reset the set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from the set `a`
`a.pop()`	N/A	Remove an arbitrary element from the set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	N/A	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	N/A	`True` if the elements of `b` are all contained in `a`
`a.isdisjoint(b)`	N/A	`True` if `a` and `b` have no elements in common

Python Basics : Set

- Like dicts, set elements generally must be immutable.
- You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set

```
In [147]: my_data = [1, 2, 3, 4]

In [148]: my_set = {tuple(my_data)}

In [149]: my_set
Out[149]: {(1, 2, 3, 4)}
```

```
In [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

- Sets are equal if and only if their contents are equal.

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: True
```

Python Basics : List, Set, and Dict Comprehensions

- `List comprehensions` are one of the most-loved Python language features.
- They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression.
- They take the basic form:

```
[expr for val in collection if condition]
```

- This is equivalent to the following for loop

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

Python Basics : List, Set, and Dict Comprehensions

- Example of List Comprehensions

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Python Basics : List, Set, and Dict Comprehensions

- A `dict` comprehension basic form

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

- A `set` comprehension basic form

```
set_comp = {expr for value in collection if condition}
```

Python Basics : List, Set, and Dict Comprehensions

- A set comprehension example

```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

- We could also express this more functionally using the map function

```
In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

Python Basics : List, Set, and Dict Comprehensions

- A dict comprehension example

```
In [159]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Python Basics : Nested list comprehensions

- Suppose we have a list of lists containing some English and Spanish names.

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
     .....:           ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

- suppose we wanted to get a single list containing all names with two or more `e`'s in them.
- `for` loop implementation:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

Python Basics : Nested list comprehensions

- We can wrap this whole operation up in a single nested list comprehension.

```
In [162]: result = [name for names in all_data for name in names
     .....:              if name.count('e') >= 2]

In [163]: result
Out[163]: ['Steven']
```

Python Basics : Nested list comprehensions

- Another example where we "`flatten`" a list of tuples of integers into a simple list of integers.

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [165]: flattened = [x for tup in some_tuples for x in tup]

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- *Keep in mind that the order of the for expressions would be the same if you wrote a nested for loop instead of a list comprehension*

Python Basics : Nested list comprehensions

- Produce a list of lists.

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```