Comp 3005: Data Science Bridge Course: Computer Science Programming Basics

Class 6: More Data Structures

Dalton Crutchfield
University of Denver

Quick Review

Data Structures

- All the variables we have seen so far have been names for single values.

- Oftentimes, we need to store more than one value, and naming each individual value is impractical.

- **Data Structures** allow us to conveniently manipulate large amounts of data.

Data Structures

- We will look at four types of data structures in Python:

1. **List**: ordered, changeable, and allows duplicate members

2. **Tuple**: ordered, unchangeable, and allows duplicate members

3. **Set**: unordered, changeable, no duplicates, and not indexed

4. **Dictionary**: unordered, changeable, and indexed

Tuples

Initialization

- Tuples are also sequences like lists but are **immutable**.
- Once you create a tuple, you cannot modify its values.
- You would need to create a new tuple if you wanted different values.
- You specify a tuple with comma-separated values, and you can optionally add parentheses ( ) around them:

```
empty = () # a tuple with no values
OneElement = (1, ) # the comma is required
TwoElements = (1, 2) # two elements
```

- A one-element tuple must contain an extra comma to distinguish it from a single value.
- We can also omit the parentheses in nonempty tuples in most cases.

Indexing and Slicing

- Indexing, len function, and slicing work exactly like lists.
- Slicing a tuple returns a new tuple as you would expect:

```
t = (0, 1, 2)
print(len(t))
print(t[0], t[1], t[2])
print(t[1:2])
```

Immutability

- Tuples are immutable, so the following produces an error:

```
t = (0, 1, 2)
t[0] = 4 # ERROR!
```

- We must create a new tuple if we want to change their values:

```
t = (0, 1)
s = t
t += (2, ) # Creates a new tuple!!
print(s) # prints (0, 1)
print(t) # prints (0, 1, 2)
```

Immutability

- If a tuple contains a mutable collection like a list, for example, then the list could change!

```
l = [1, 2]
t = (l, l)
print(t) # prints ([1, 2], [1, 2])
l[0] = 3
l[1] = 4
print(t) # prints ([3, 4], [3, 4])
```

- This is because the alias to the list didn't change, but the list to which the **alias** is pointing changed.

Enumerate

- The for-loops we wrote to iterate over a list either used values or indices.
- When the loop requires both indices and values, then we need the square brackets to access the element:

```
l = [24, 45, 83, 54, 19, 70]
for i in range(len(l)):
    print('index: ',i, 'value:', l[i])
```

- Tuples and the enumerate function make it easier to use a loop that accesses the value and the index:

```
l = [24, 45, 83, 54, 19, 70]
for (i, element) in enumerate(l):
    print('index: ',i, 'value:', element)
```

- Note that the variable in the for-loop is a tuple.
- The enumerate function returns a collection of tuples.

Multiple Assignments and Return

- We can use tuples to do multiple assignments:

```
# declares three new variables: a=1, b=2, c=3
a, b, c = 1, 2, 3
print(a, b, c)
# reassigns those variables
a, b, c = 4, 5, 6
print(a, b, c)

#number of variables of left-hand side must match
#number on the right-hand side
a, b, c = 1, 2 # ERROR, not enough values
a, b = 1, 2, 3 # ERROR, too many values
```

Multiple Assignments and Return

- This process is called unpacking. The entire right-hand side is evaluated before anything on the left-hand side is updated, which can shorten code:

```
# prints Fibonacci numbers
a, b = 0, 1
for i in range(0, 20):
    print(a)
    a, b = b, a+b # right-hand side evaluated before any assignment
occurs!
```

- We can also use this to return multiple values!

Sets

Sets

- Sets in Python are just like mathematical sets.
- They are unordered collections of unique elements.
- They are typically used for categorization.
    - Set of English words
    - Set of ASCII characters
- We are interested in memberships in sets but not order.
- Note that element uniqueness reflects this emphasis since an element is either in the set or not.
- Sets can only store hashable types. For built-in Python types, this means that we can only store immutable objects!
- An immutable set is called a **frozenset**.

Initialization

- Creating an empty set:

```
emptySet = set()
emptyFrozen = frozenset()
```

- Creating a non-empty set:

```
s1 = {1, 2, 3, 4, 5}
s2 = {'a', 'b', 'c'}
```

- Creating a non-empty frozenset:

```
f1 = frozenset({1, 2, 3, 4})
f2 = frozenset({'a', 'b', 'c'})
```

Set Comprehension

- Unsurprisingly at this point, set comprehension works like list comprehension:

```
s1 = { i**2 for i in range(-5, 5) }
s2 = set(abs(i) for i in range(-5, 5))
```

Set Membership

- We can use ‘**in**’ and ‘**not in**’ to check membership in a set:

```
fruits = {'apple', 'banana', 'orange'}
nuts = {'walnut', 'almond', 'cashew'}
test_string = 'bus'
if test_string in fruits:
      print('a', test_string, 'is a type of fruit!')
Else:
      print('a', test_string, 'is not a fruit!')
if test_string not in nuts:
      print('a', test_string, 'is not a nut!')
Else:
      print('a', test_string, 'is a type of nut!')
```

Operations on Sets

- Sets and frozensets support the same operations as mathematical sets.
    - **Set union (A union B):** creates a new set that combines the elements in A and B
    - **Set intersection (A intersection B):** creates a new set that contains elements that A and B have in common
    - **Set difference (A minus B):** creates a new set resulting from removing elements of B from A

Set Union

```
# union: elements in one set or the other
s = {5, 2, 6}
t = {1, 2, 3}
# 1) method: returns a NEW SET
print(s.union(t))
print(s)
# 2) operator: returns a NEW SET
print(s | t) #this is the vertical bar
print(s)
# 3) augmented operator: updates s
s |= t
print(s)
```

Set Intersection

```
# intersection: elements in one set and the other
s = {1, 2, 3}
t = {2, 3, 4}
# 1) method: returns a NEW SET
print(s.intersection(t))
print(s)
# 2) operator: returns a NEW SET
print(s & t)
print(s)
# 3) augmented operator: updates s
s &= t
print(s)
```

Set Difference

```
# difference: removes the elements of one set from another
s = {1, 2, 3, 4, 5}
t = {2, 4}
# 1) method: returns a NEW SET
print(s.difference(t))
print(s)
# 2) operator: returns a NEW SET
print(s - t)
print(s)
# 3) augmented operator: modifies s
s-= t
print(s)
```

Comparing Two Sets

• In mathematics, sets are compared using containment. Is one set a subset or superset of another? Python has changed the standard comparison operators to reflect that terminology. For sets A and B:

Python Comparison	Math Notation	Meaning
A <= B	$A \subseteq B$	A is a subset of B. All elements of A are in B.
A < B	$A \subset B$	A is a strict subset of B. All elements of A are in B, and A is not equal to B.
A >= B	$A \supseteq B$	A is a superset of B. All elements of B are in A.
A > B	$A \supset B$	A is a strict superset of B. All elements of B are in A, and A is not equal to B.

Comparison Example

```
def check(s, t):
    if s <= t:
        print(s, '<=', t)
    if s < t:
        print(s, '<', t)
    if s >= t:
        print(s, '>=', t)
    if s > t:
        print(s, '>', t)
check({1, 2, 3}, {1, 2, 3}) # '<=' and '>='
check({1, 2}, {1, 2, 3}) # '<=' and '<'
check({1, 2, 3}, {2, 3}) # '>=' and '>'
check({1, 2}, {3, 4}) ## nothing. neither is a subset of the other
```

Iteration

• Both sets and frozensets can be iterated using a for-loop. However, since sets are unordered, we can only use the element form.

• The order in which elements are printed will vary from run to the next:

```
s = {'a', 'b', 'c', 'd'}
for c in s:
    print(c)
```

Dictionaries

Dictionaries

• Lists and tuples are ordered sequences of values, so elements are ordered by their indices.
• Dictionaries are not ordered data structures, and instead of indexing values by their order in the sequence, we use a **key** to index the value instead.
• Therefore, a dictionary stores pairs of keys and values and maps each key to a value.
• A common example for a dictionary is the phone book, where the key is the name and the value is the phone number.
• The keys within a dictionary must be **unique**.
• There are other names for dictionaries that you can find in other languages: associative arrays, maps, hashes, and hash tables.

Dictionary Example

- The syntax for creating a dictionary is similar to that of other collections:

```
# create an empty phonebook
phonebook = {}
# fill it with entries
phonebook['Alice'] = '1-234-567-8910'
phonebook['Bob'] = '1-098-765-4321'
phonebook['Charlie'] = '1-111-111-1111'
# look up numbers by name
print("Alice's number is", phonebook['Alice'])
print("Bob's number is", phonebook['Bob'])
print("Charlie's number is", phonebook['Charlie'])
```

Initialization

- We can create an empty dictionary in one of two ways:

```
emptyDict1 = {}
emptyDict2 = dict()
```

- We can also create the same phone book by specifying key-value pairs:

```
phoneBook = {'Alice': '1-234-567-8910',
'Bob': '1-098-765-4321',
'Charlie': '1-111-111-1111'}
```

- The dict(…) function, which takes iterable collection of tuples, can be used:

```
phoneBook = dict([('Alice', '1-234-567-8910'),
('Bob', '1-098-765-4321'),
('Charlie', '1-111-111-1111')])
```

Keys and Values

- Values can be of any type, but keys must be a hashable type.
- For the built-in Python types, the key must be an immutable type.
- Can use: ints, floats, strings, bools, frozensets, and tuples
- Cannot use: lists and dictionaries

```
d = {}
# allowed
d[2] = "two"
d["three"] = 3
d[3.0] = "three point oh"
d[(0, 1, 2)] = "value for the key: (0, 1, 2)"
# not allowed: these raise TypeError's
d[[1, 2, 3]] = "cannot use a list as a key..."
d[{}] = "cannot use a dict as a key..."
```

Accessing Values

- Since keys are the indices into the dictionary, they are used to access the values with square brackets, like sequences:

```
print(phonebook['Alice'])
print(phonebook['Bob'])

# trying to use an unknown key is an error (strings must match EXACTLY)

print(phonebook['Dave']) # unknown key: never added
print(phonebook['bob']) # unknown key: wrong capitalization
print(phonebook['Bob ']) #unknown key: extra space at the end
```

Modifying and Adding Values

- Dictionaries are **mutable**, so we can modify the values or add new ones:

```
# square braces used to set values
# updates Alice's number
phonebook['Alice'] = '1-222-222-2222’

# Adds an entry for Dave
phonebook['Dave'] = '1-333-333-3333'
```

Removing Values

- Key-value pairs can also be removed from the dictionary:

```
# mappings can be removed using 'del'
# Remove the entry for 'Dave'
del phonebook['Dave']
phonebook['Dave'] # so now this is an error (there is no mapping)
```

Dictionary Comprehension

- Just like lists and sets, we can create a dictionary using dictionary comprehension:

```
# {0: 0, 2: 4, 4: 16}
dict ={n:n**2 for n in range(5) if n%2 == 0}
```

Dictionary Methods

- Using the in keyword, we can check if a key is in the dictionary:

```
if 'Dave' in phonebook:
    print("Dave's number is", phonebook['Dave'])
else:
    print("Who is this 'Dave' you speak of?")
```

Dictionary Methods

- There are many methods on a dictionary. All keys, values, and key-value pairs can be accessed using the following methods:

    - **keys()** returns a list of all keys
    - **values()** returns a list of all values
    - **items()** returns a list of tuples for each key and value pair

Iteration

- We can use these methods to iterate over elements of the dictionary:

```
# iterate over keys
for key in phonebook.keys():
    print(key)

# iterate over values
for val in phonebook.values():
    print(val)

# iterate over both at the same time
for (key, val) in phonebook.items():
    print(key, ":", val)
```

Iteration

- You can also iterate over the keys using the following:

```
# if no method is used, the keys are iterated
for key in phonebook:
    print(key)
```

Today in Tech History

October 5, 2011

After a long battle with pancreatic cancer, technology visionary and founder of Apple Computer, Steve Jobs passes away. Jobs' contributions to the technology industry are undeniable. Together with Steve Wozniak, Steve Jobs started the personal computer revolution with their Apple II computer. After being forced out of Apple, Jobs went on to found NeXT, Inc. and then purchase Pixar, the company that would redefine the animated motion picture industry. In 1997, Apple purchased NeXT which brought Jobs back to Apple and the technology developed at NeXT was used as the foundation for Apple's future operating systems, Mac OS X and iOS. By introducing the iPhone and iPad, he ended the PC era he created, kickstarted The New World of Technology and led Apple from the brink of collapse to the most valuable