Lecture 3: Python Containers

October 2nd, 2018

1 Containers

Background For any given problem, there are an infinitude of ways to solve the problem. One of the *types of decisions* we are faced with when writing a program is: *how to (internally) store* or represent our data in a way that lends itself to the problem at hand.

Motivation The practical implications of choosing the right (or wrong) data structure may be a program that runs orders of magnitudes slower than it should; i.e. knowing the contents of this lecture can help us *get started* with understanding how to answer the question, "why is my program so slow, and what can I do to make it faster?".

Definition We provide a formal definition, and a few concrete examples.

- *Containers* (Abstract Data Types) are objects that contain one or more other objects. They are sometimes called "collections" or "data structures".
- Last lecture, we introduced the Python `list` container. Today we will see Dictionaries, Sets, and Tuples.

1.1 Preview of Various Containers

Sequential containers store data items in a specified order. Think elements of a vector, names in a list of people that you want to invite to your birthday party. The most fundamental Python data type for this is called a `list`. Later in the course we will learn about containers that are more appropriate (and faster) for numerical data that come from NumPy.

We have seen lists in our prior lecture.

```python
my_list = [4, 8.95, 'list item', ["sub", "list"]]
print(my_list)
```

Python has another built-in sequential container called a `tuple`. Tuples are a lot like lists, but the elements cannot be modified after the tuple is created.

```
my_tup = (1, 2, "str", 99.99)
print(my_tup)
print(my_tup[1]) # Prints the second item. Happens to be the integer value two.

my_tup[1] = 42   # Error: 'tuple' object does not support item assignment.
```

Tuples are said to be immutable. We'll see why this is important later.

Associative containers store data, organized by a unique *key*. Think of a dictionary of word definitions. They unique *key* is a word, the value associated with the key is the definition. In Python, this is represented with the built-in `dict` or Dictionary type. You will soon learn the greatness of dictionaries in Python.

```
emails = dict()
emails['andreas'] = 'andreas@mail.com'
emails['nick'] = 'nwh@stanford.net'
print(emails)
```

Access a single element:

```
emails['nick']
```

We'll learn what happens if we query a key which does not exist in our dictionary soon.

Set containers store *unique* data items. They are related to dictionaries, because dictionaries require the keys to be unique.

```
dinos = set()
dinos.add('triceratops')
dinos.add('t-rex')
dinos.add('raptor')
print(dinos)
dinos.add('pterodactyl')
dinos.add('t-rex')          # Nothing actually gets added here...
print(dinos)
```

So, What About Making My Program Faster? Each of these data structures have different properties associated with them which make them amenable to certain operations. E.g., the fact that a *tuple* is immutable means that the programmer may be able to place certain guarantees on the (existence of) contents; a *list* is "fast" to append to, which might be nice if you want a growing collection (whose final size is not known apriori); a *dictionary* allows us to not only associate keys with values, but to look them up "quickly" in the future. We'll learn precisely what the quoted terms mean as we progress through the quarter.

2 Dictionaries

Dictionaries are an *associative container*: they map *keys* to *values*.

Use Case You want to repeatedly "look up" values that are associated with identifiers. E.g. your contacts list, which might pair a name with a phone-number. Want to look up a friends phone number? Just type their name and you get the result *instantly*.

Python Syntax They *can* be denoted by curly braces.

- Create an empty dictionary: `empty_dict = {}` or `empty_dict = dict()`
- Create a dictionary with some data: `ages` $= \{\texttt{"}\underbrace{\texttt{brad}}_{\text{key}}\texttt{"} : \underbrace{51}_{\text{value}}, \texttt{"angelina"} : 40\}$.

Values can be *any* python object. E.g. numbers, strings, lists, other dictionaries

Keys must be *immutable*. E.g. Numbers, strings, tuples (with *only* immutable data).[1]

Access values associated with a key with square brackets: `value = dictionary[key]`

Essential Caveat: Unordered Collection There is no sense of order in a python dictionary. E.g. when used in a loop, the key-value pairs may come out in any order.

2.1 Creation

```
simple_dict = {1 : "the number 1", "42" : 42, "a list" : [1,2,3]}
print(simple_dict)
```

Here is a slightly more useful dictionary associating names with ages. We start with an empty dictionary and add to it.

```
ages = {} # or ages = dict()
ages['brad'] = 51
ages['angelina'] = 40
ages['leo'] = 40
ages['bruce'] = 60
ages['cameron'] = 44
ages
```

We can get the size of the dictionary with `len`, i.e. query the number of keys.

```
len(ages)
```

[1] If we try to use a tuple as key, where one of the elements of the tuple is a mutable type, e.g. a list, we will get a `TypeError: unhashable type`. By the way, this also informs us of the implementation of our associative container.

2.2 Access

Retrieval Given a key, we can fetch the corresponding values quickly. When the key does not exist, we get a `KeyError`.

```python
ages['leo']     # Corresponding value returned.
ages['helen']   # KeyError: 'helen'
```

Or you can use the `get()` method:

```python
tmp = ages.get('brad')
print(tmp)                # Prints the integer 51.
tmp = ages.get('helen')  # Returns None.
print(tmp)
```

Updating Values We can change the value associated with a key.

```python
ages['brad'] = 52   # Perhaps Brad aged one year.
print(ages)
```

Querying Existence of a Key Suppose we want to ask if a key is presently contained?

```python
print(ages)  # Prints all key-value pairs; possibly overwhelming.
```

The `in` operator returns a Boolean indicating whether the key is already present.

```python
'nick' in ages   # Returns a corresponding boolean. E.g. False here.
'leo'  in ages   # And True here.
```

2.3 Iteration

Iterate through the keys as follows.

```python
for key in ages:                        # More explicitly: ages.keys().
    print("{} = {}".format(key,ages[key]))
```

We can also **iterate over values**.

```python
for value in ages.values():
    print(value)
```

Additionally, we can **iterate through key-value pairs** together.[2]

[2] The above syntax is more efficient in Python 3 compared with Python 2. To achieve equivalent performance in Python 2, it is best to ask for an *iterator* over the key-value pairs via dir.iteritems(). In Python3,

```python
for k, v in ages.items():
    print('{} is {} years old'.format(k,v))
```

Revisiting: Implications of an Unordered Collection We emphasize that the order of iteration is *not guaranteed* when using an associative data structure. E.g. if you observe a particular ordering when iterating over a fixed dictionary in a python interpreter, this ordering is not guaranteed to persist when the same dictionary is created in a new session.

2.4 Methods

See `help(dict)` to get a summary of all dictionary methods. Py-doc for dictionaries.

```
clear(...)
    D.clear() -> None.  Remove all items from D.

copy(...)
    D.copy() -> a shallow copy of D

get(...)
    D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.

items(...)
    D.items() -> a set-like object providing a view on D's items

keys(...)
    D.keys() -> a set-like object providing a view on D's keys

pop(...)
    D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised

setdefault(...)
    D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update(...)
    D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
    If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
    If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
    In either case, this is followed by: for k in F:  D[k] = F[k]

values(...)
    D.values() -> an object providing a view on D's values
```

`dir.items()` returns an object that provides access to the data in a container in a sequential fashion **without** requiring the creation of a new data structure and copying of data. This is sort of like the `range()` function.

3 Tuples

Tuples are essentially immutable lists, containing possibly heterogeneous elements.

Use Case You have a set of (possibly non-unique or heterogeneous) data that you wish to remain *ordered* and *immutable* for the lifetime of the object. E.g. we may expect that a person's name should be invariant over the course of a peron's lifetime, and therefore may choose to represent first and last names via a length-two tuple.[3]

Syntax They are denoted by parentheses: `tup = (1,2,3)`.

- Create an empty tuple: `empty_tup = ()`, or `empty_tup = tuple()`.
- Create a tuple with some data: `tup = (42, 3.14, "abc")`.

Values themselves can be of *any* type.

Immutability The number of items in a tuple, or what those fixed number of items reference, are not allowed to change after creation. Put differently, the *size* and *type signature* of a tuple are fixed at the time of creation.

Ordered Collection Since tuples are immutable, their elements are ordered. We can therefore access items via *indexing* and *slicing*, similar to lists.

Similarities with Lists Suppose we have a tuple of homogeneous data. There are some similarities in how we can access the elements relative to a `list`.

```
my_tuple = (1, 2, 3, 4)
print(my_tuple[1])   # Prints the second value, happens to be integer two.
print(my_tuple[1:3]) # Print out second and third value, up to (but excl.) fourth.
```

This all feels quite similar to a `list`. We can even loop over values with `for`:

```
for item in my_tuple:      # Identical to list-iteration.
    print(item)
```

[3] An alternative choice may be to hold two separate objects in memory, each a `list` of equal length; the first list could hold first names, and the second could hold last names; there is an assumed invariant that the indexing between first and last names is consistent, i.e. that each individual appears exactly once in each object and the index at which they appear is the same across both objects. While the assumed invariant is perhaps dicey, this may allow for slicing *across* (first or last) names.

3.1 Immutability

However, recall that while `lists` are mutable, tuples are not!

```
my_list = ["I", "am", "a", "list"]
my_list[0] = "I still"              # Modifying elements of a list is OK.
print("my_list:", my_list)          # --> my_list: ["I still", "am", "a", "list"].

my_tuple = ("I", "am", "a", "tuple")
my_tuple[2] = "still a"  # TypeError: 'tuple' object does not support item assignment
```

Caveat However, the underlying elements of a tuple which are mutable in nature may be modified. E.g. if one of the items of a tuple is a list, and since lists are mutable, it's possible to change the *contained* object.

```
t = (42, 3.14, ['a', 'b', 'c'])       # Tuple of length 3: tuple<int, float, list>
t[2] = ["another", "list", "of 3"]    # We cannot re-assign data.
t[2].append('z')                      # Mutate third elem, itself a list, via 'append'.
print(t)                              # 't' is "unchanged": tuple<int, float, list> ...
                                      # ... but! underlying data are different.
```

Check out this type of behavior in the following Python Tutor example, alongside a visual.

References Like all variables in Python, items in a tuple are actually references to objects in memory. Once a tuple has been created, its references to objects cannot be reassigned. Tuple items may reference an object that is mutable (say, a list). In this case the referenced mutable object may be changed in someway.

3.1.1 A note on (im)mutability

It is natural to wonder why have immutable objects at all. One answer to this is practical: in Python, only immutable objects are allowed as keys in a dictionary or items in a set.

The more detailed answer requires knowledge of the underlying data structure behind Python dictionary and set objects. In the context of a Python dictionary, the memory location for a key-value pair is computed from a *hash* of the key. If the key were modified, the *hash* would change, likely requiring a move of the data in memory. Thus, Python requires immutable keys in dictionaries to prevent unnecessary movement of data.

It is possible to associate a value with a new key with the following code:

```
ages = {'cameron': 44, 'angelina': 40, 'bruce': 60, 'brad': 51, 'leo': 40}
print(ages)
ages['BRUCE'] = ages['bruce']
print(ages)
```

3.2 Tuple (Un)-Packing

Tuple packing and unpacking is very convenient Python syntax. In packing, a tuple is automatically created by combining values or variables with commas:

```python
t = 1, 2, 3
print("type(t):", type(t))
print("t:", t)
```

Tuple unpacking can store the elements of a tuple into multiple variables in one line of code.

```python
my_tuple = ("a string", 43, 99.9)
my_str, my_int, my_flt = my_tuple

### Equivalently:
my_str = my_tuple[0]
my_int = my_tuple[1]
my_flt = my_tuple[2]
```

3.2.1 Swapping Variables Via Tuple Unpacking

Tuple packing and unpacking allows swapping of variables without (syntactically) creating a temporary variable:

```python
x, y = 1001, 'random string'
x, y = y, x
print("x:", x)
print("y:", y)
```

4 Sets

In Python, a set is an unordered, mutable collection of unique items.

Use Case We care about existence, and not duplicity; testing for existence is "fast". E.g. When a new-user signs up for a service, we want to check if their proposed username is available *instantly*, without scanning through existing users.[4]

Syntax They *can* be denoted by braces.

- Create a set with: `my_set = set([1, 2, 3])`
- Or create a set with: `my_set = {5, 8, "str", 49.2}`

Values of a set must be immutable (same reason keys in a dictionary must be immutable).

[4] If we stored our usernames in a `list` instead, and still checked for existence with the `in` operator, then as our service becomes more popular it would take new users *longer and longer* to sign-up! Ouch.

4.1 Creation, Adding Elements

```python
# create and update using add method
myclasses = set()
myclasses.add("math")
myclasses.add("chemistry")
myclasses.add("literature")

# create using {} notation
yourclasses = {"physics", "gym", "math"}
```

4.2 Set Operations

Test for membership (or existence) with the `in` operator:

```python
"gym" in myclasses      # Returns False.
"gym" in yourclasses    # Returns True.
```

Compute set intersections or unions.

```python
myclasses & yourclasses  # Returns only elements in both sets.
myclasses | yourclasses  # Returns elements found in either myclasses OR yourclasses.
```

We remark that the output of a set operation is itself a new set.

4.3 Application: Finding Unique Items in a List

Let's create a list with non-unique elements.

```python
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
```

Phone-screen: what are the unique elements?

Answer: from this list, we can easily create a `set`, and this answers the very question posed.

```python
fruit = set(basket)
print(fruit)
```

4.4 Methods

For more information and examples see the `set` documentation in the official Python Tutorial and Library Reference.

Also see `help(set)`:

```
add(...)
    Add an element to a set.

    This has no effect if the element is already present.

clear(...)
    Remove all elements from this set.

copy(...)
    Return a shallow copy of a set.

difference(...)
    Return the difference of two or more sets as a new set.
    (i.e. all elements that are in this set but not the others.)

discard(...)
    Remove an element from a set if it is a member.
    If the element is not a member, do nothing.

intersection(...)
    Return the intersection of two sets as a new set.
    (i.e. all elements that are in both sets.)

isdisjoint(...)
    Return True if two sets have a null intersection.

issubset(...)
    Report whether another set contains this set.

issuperset(...)
    Report whether this set contains another set.

pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.

remove(...)
    Remove an element from a set; it must be a member.

    If the element is not a member, raise a KeyError.

symmetric_difference(...)
    Return the symmetric difference of two sets as a new set.
    (i.e. all elements that are in exactly one of the sets.)

union(...)
    Return the union of sets as a new set.
    (i.e. all elements that are in either set.)
```

5 List Comprehensions

It is a common programming task to produce a list whose elements are the result of a function applied to another list. For example.

```
def square(x):
    return x*x

my_list = [1,2,3,4,5,6,7,8]

new_list = []
for x in my_list:
    new_list.append(square(x))

print(new_list)
```

This is so common that Python has special syntax to achieve the same thing.

```
list_comp = [x*x for x in my_list]
print(list_comp)
```

This is called a list comprehension. It creates a new list by applying an operation to each item of another list.

It is also possible to filter out elements of list in a comprehension:

```
my_list = [1,2,3,4,5,6,7,8,9,10,11]
odds = [x for x in my_list if x % 2 != 0]
odds
```

Python also has set and dictionary comprehensions. For further reference, see the List Comprehensions in Python Tutorial.

6 Exercises

6.1 References, Tuples, and Mutable Objects

Execute and understand the following code in Python Tutor.

```
sub_list_1 = [1,3,8]
sub_list_2 = ['z','y','x']
my_list = [2, 'a string', sub_list_1]
my_list[2] = sub_list_2
my_list[2][0] = 'new string'

my_tup = (2, 'a string', sub_list_1)

# cannot modify tuple references
# my_tup[2] = sub_list_2

# we can modify a mutable object referenced by a tuple
my_tup[2][0] = 'from my_tup'

# can can look at object ids
print(id(sub_list_1))
print(id(my_tup[2]))
```

6.2 Applied Problems using Census Data

This problem will contrast the use of `sets` with `lists`.

Goal: write program to predict *male* or *female* given name.[5]

Data: can be found here, Frequently Occurring Surnames from Census 1990

Algorithm: the following steps incorrectly bias the results to be skewed "M", but we will use it is a skeleton first pass implementation.

1. If input name is in list of males, return `"M"`
2. Else if input name is in list of females, return `"F"`
3. Otherwise, return `"NA"`

[5] Thanks to Patrick LeGresley for this example.

Understanding our Data After obtaining our data, we inspect its format.

```
pwd
/Users/asantucci/cme211-notes/lecture-03

 ls -1 *.first
 dist.female.first
 dist.male.first

 head -n 5 dist.female.first
MARY            2.629  2.629       1
PATRICIA        1.073  3.702       2
LINDA           1.035  4.736       3
BARBARA         0.980  5.716       4
ELIZABETH       0.937  6.653       5

 head -n 4 dist.male.first
JAMES           3.318  3.318       1
JOHN            3.271  6.589       2
ROBERT          3.143  9.732       3
MICHAEL         2.629 12.361       4
```

Notes:

- The unix `head` command prints out the first number lines of a text file based on the number after the `-n` argument.
- First column of the data file contains the name in uppercase.
- Following columns contain frequency data and rank, which we won't use in this lecture.

6.2.1 Set Exercise

Write a Python script `names_set.py` to implement the name to gender algorithm specified above using the Python `set` container. Also print out some information about the data sets.

The program should take data filenames and test names from the command line. If no command line arguments are provided, the script should print out a helpful usage message.

```
$ python3 names_set.py
Usage:
  $ python3 names_set.py FEMALE_DATA MALE_DATA [TEST NAMES]
Example:
  $ python3 names_set.py dist.female.first dist.male.first Nick
```

If data filenames and test names are provided, the script should behave as follows:

```
$ python3 names_set.py dist.female.first dist.male.first Nick Sally Bicycle
There are 4275 female names and 1219 male names.
There are 331 names that appear in both sets.
Nick: M
Sally: F
Bicycle: NA
```

The word `Bicycle` does not appear in either the male or female dataset, so `NA` is printed. A solution can be found in our Github, here.

6.2.2 List Exercise

Write a Python script `names_list.py` to implement the name to gender algorithm specified above using the Python `list` container. Also print out some information about the data sets. The script should behave the same as `names_set.py`. A solution can be found here.

6.2.3 Dictionary Exercise

Some names appear in both **male** and **female** lists. Some names might not appear in either list. Let's write a new algorithm to handle this uncertainty:

Given an input name: - return `0.0` if male - return `1.0` if female - return `0.5` if uncertain or name does not appear in dataset

Exercise: write a Python script `names_dict.py` to implement the name to gender algorithm specified above using the Python `dict` container. Also print out some information about the data sets. The behavior should follow:

Usage message:

```
$ python3 names_dict.py
Usage:
  $ python3 names_dict.py FEMALE_DATA MALE_DATA [TEST NAMES]
Example:
  $ python3 names_dict.py dist.female.first dist.male.first Nick
```

Let's examine `names_dict.py` in action.

```
$ python3 names_dict.py dist.female.first dist.male.first Nick Sally Billy
There are 5163 names in our reference data.
Nick: 0.0
Sally: 1.0
Billy: 0.5
```

The name "Billy" appears in both male and female datasets, so `0.5` is printed after the name to indicate uncertainty. Solution.