Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei
gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
October 22, 2018

Lecture 4: Python's Built-in Data Types (2)

Data Type Hierarchy

- Python's built-in data types can be grouped into several classes.
- We use the same hierarchy scheme used in the official Python documentation, which defines the following classes:
    - **numeric**, **sequences**, **sets** and **mappings** (and a few more not discussed further here).
- A special mention goes to two particular data types: `bool` and `NoneType`.

In the previous lecture...

- Built-in data types:
    - `bool` and `NoneType` (`None`)
    - numeric: `int`, `float`, `complex` (*immutable*)
    - sequences: `str`, `bytes` (*immutable*), `bytearray` (*mutable*)

In *this* lecture

- We finalize the discussion on Python's built-in data types:
    - sequences: `list` (*mutable*) and `tuple` (*immutable*)
    - sets: `set` (*mutable*)
    - mappings: `dict` (*mutable*)

Lists: Type `list` (*mutable*)

Properties

- An object of type `list` represents a sequence of possibly heterogeneous Python objects.
- Lists are one of the basic data structure used to build more complex data types.
- Being it mutable, any list object can be modified *in place*.
- Operations defined on lists are the same we have already seen for any other sequence type (e.g., `str`).

Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

In [1]:
```
# Define a reference to an empty list object
a_list = []
# Rebind the above reference to another list object
a_list = [1, 2, 'foo', None]
# Print the length of the list
print(len(a_list))
# Access the i-th element of the list (remember, the first element is indexed by 0)
print(a_list[2])
# a_list[-i] stands for a_list[n-i], where n = len(a_list)
# In the example below, therefore, we are accessing the very last element of the list
print(a_list[-1]) # same as a_list[len(a_list)-1]
# Change an element of the list (in place)
a_list[1] = 'bar'
print(a_list)
# Trying to access an element outside of the index range
print(a_list[7])
```

```
4
foo
None
[1, 'bar', 'foo', None]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-4bc7f67583df> in <module>()
     14 print(a_list)
     15 # Trying to access an element outside of the index range
---> 16 print(a_list[7])

IndexError: list index out of range
```

In [2]:
```
# Insert a new element at the end of an existing list using the 'append' method
a_list.append(42)
print(a_list)
# Insert a new element at a specific position of an existing list
# a_list.insert(pos, element) where pos is the position where element should be inserted
a_list.insert(2, 'red')
print(a_list)
# Is insert robust?
a_list.insert(100, 'blue')
print(a_list)
# -pos is a shortcut for len(a_list)-pos, therefore -1 means the new element
# will replace the current last one, which is then properly shifted to the right
a_list.insert(-1, 73)
print(a_list)
# a_list.insert(len(a_list), element) is equivalent to a_list.append(element)
a_list.insert(len(a_list), 'cyan')
print(a_list)
```

```
[1, 'bar', 'foo', None, 42]
[1, 'bar', 'red', 'foo', None, 42]
[1, 'bar', 'red', 'foo', None, 42, 'blue']
[1, 'bar', 'red', 'foo', None, 42, 73, 'blue']
[1, 'bar', 'red', 'foo', None, 42, 73, 'blue', 'cyan']
```

Checkpoint Quiz

How would you insert the list `[4, 5]` as the first element of our original list?

In [3]:
```
a_list.insert(0, [4,5])
print(a_list)
```

```
[[4, 5], 1, 'bar', 'red', 'foo', None, 42, 73, 'blue', 'cyan']
```

Notes on `insert` *vs.* `append`

- `insert(pos, element)` is computationally expensive compared to `append(element)`.
- This is because references to elements at positions `pos, pos+1, ..., n-1` (where `n` is the total number of elements currently in the list) have to be shifted internally to make room for the new element.
- If you need to insert elements at both the beginning and end of a sequence, you may explore `collections.deque`, a double-ended queue, for this purpose.

In [4]:
```
# The inverse operation of 'insert' is 'pop',
# which removes and returns an element at a particular index.
# If no index is passed in, the last element is popped out by default
elem = a_list.pop()
print(elem)
print(a_list)
# Otherwise you can pass the index as argument
# Here we pop out the fourth element without storing it into a variable
a_list.pop(3)
print(a_list)
```

```
cyan
[[4, 5], 1, 'bar', 'red', 'foo', None, 42, 73, 'blue']
[[4, 5], 1, 'bar', 'foo', None, 42, 73, 'blue']
```

Checkpoint Quiz

What happens if we try to do the following `a_list.pop(123)`, namely if we try to pop out an element using an out-of-range index?

In [5]:
```
# Trying to pop out an element using an out-of-range index
a_list.pop(123)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-5-9312c34ca1c2> in <module>()
      1 # Trying to pop out an element using an out-of-range index
----> 2 a_list.pop(123)

IndexError: pop index out of range
```

In [6]:
```
# Elements can be removed by index with 'del' or by value using 'remove',
# 'remove' locates the first value passed as input and removes it from the list.
# Let's append another 'foo' at the end of the list
a_list.append('foo')
print(a_list)
# Then, remove the 2nd element (index=1)
del a_list[1]
print(a_list)
# Now, remove the first occurrence of 'foo'
a_list.remove('foo')
print(a_list)
# What if we try to remove an element which is not in the list?
a_list.remove('baz')
print(a_list)
```

```
[[4, 5], 1, 'bar', 'foo', None, 42, 73, 'blue', 'foo']
[[4, 5], 'bar', 'foo', None, 42, 73, 'blue', 'foo']
[[4, 5], 'bar', None, 42, 73, 'blue', 'foo']

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-bce0b8060d6c> in <module>()
     11 print(a_list)
     12 # What if we try to remove an element which is not in the list?
---> 13 a_list.remove('baz')
     14 print(a_list)

ValueError: list.remove(x): x not in list
```

In [7]:
```
# Although not very efficient, we can also check if a list contains an element
'red' in a_list
```

Out[7]: False

Notes on the usage of `in` with lists

- Checking whether a list contains an element is a lot **slower** than using `dict` and `sets` (to be introduced shortly).
- Using `list`, Python has to make a **linear scan** across the values of the list (time complexity is $O(n)$ if n is the number of elements of the list).
- Using `dict` and `sets` - which are based on *hash tables* - can make the check in constant time, i.e., $O(1)$.

List Concatenation

In [8]:
```
# Lists can be added together using '+'
[42, True, 'bar'] + [False, None, 'foo'] + ['baz', 48, '']
```

Out[8]:
```
[42, True, 'bar', False, None, 'foo', 'baz', 48, '']
```

In [9]:
```
# If you have a list already defined,
# you can append multiple elements to it using 'extend' in place.
a_list = [42, True, 'bar']
print(a_list)
a_list.extend([False, None, 'foo'])
print(a_list)
a_list.extend(['baz', 48, ''])
print(a_list)
```

```
[42, True, 'bar']
[42, True, 'bar', False, None, 'foo']
[42, True, 'bar', False, None, 'foo', 'baz', 48, '']
```

Notes on list concatenation

- Concatening lists using `+` is generally an expensive operation:
    - A new list must be created and the objects copied over for each concatenation (similar to string concatenation).
- Using **`extend`** to append elements to an existing list is the preferred way to go, especially if you are building up a large list.

```python
# This approach uses the '+' operator (slower)
result_list = []
for a_list in list_of_list:
    result_list += a_list

# This approach uses 'extend' method (faster)
result_list = []
for a_list in list_of_list:
    result_list.extend(a_list)
```

List Sorting

In [10]:
```
# Lists can be sorted in-place (without creating a new object) by calling `sort`
a_list = [7, 2, 5, 1, 3]
print(a_list)
a_list.sort()
print(a_list)
# sort has a few options that will occasionally come in handy.
# For example, you can pass a secondary sort key,
# i.e., a function that produces a value to use for sorting the objects.
# The following example shows how to sort a list of strings by their lengths:
str_list = ['saw', 'small', 'He', 'foxes', [123], 'six']
print(str_list)
str_list.sort(key=len)
print(str_list)
```

```
[7, 2, 5, 1, 3]
[1, 2, 3, 5, 7]
['saw', 'small', 'He', 'foxes', [123], 'six']
[[123], 'He', 'saw', 'six', 'small', 'foxes']
```

Notes on sorting

- Whenever you sort a list using `sort()`, remember that this happens in-place (i.e., you can not recover the original order).
- If you want to display a list in sorted order, but preserve the original order, you can use the `sorted()` function, instead.
- `sorted()` function also accepts the optional `reverse=True` argument.

In [11]:
```
students = ['bob', 'alice', 'carl']

# Display students in alphabetical order, but keep the original order.
print("Here is the list in alphabetical order:")
print(sorted(students))

# Display students in reverse alphabetical order, but keep the original order.
print("Here is the list in reverse alphabetical order:")
print(sorted(students, reverse=True))

print("Here is the list in its original order:")
# Show that the list is still in its original order.
print(students)
```

```
Here is the list in alphabetical order:
['alice', 'bob', 'carl']
Here is the list in reverse alphabetical order:
['carl', 'bob', 'alice']
Here is the list in its original order:
['bob', 'alice', 'carl']
```

List Slicing

In [12]:
```
# As for any other sequence types (tuples, NumPy arrays, pandas Series),
# you can select sections of lists using [start:stop] indexing notation
a_list = [7, 2, 3, 7, 8, 6, 0, 1]
print(a_list[1:5])
```

```
[2, 3, 7, 8]
```

Notes on slicing

- Element at the `start` index is included, whilst the `stop` index is not.
- Therefore, the total number of elements in the result is `start - stop`.
- Either the `start` or `stop` can be omitted.
  - if so, `start` will default to `0` and `stop` to `n` (where `n` is the length of the list).

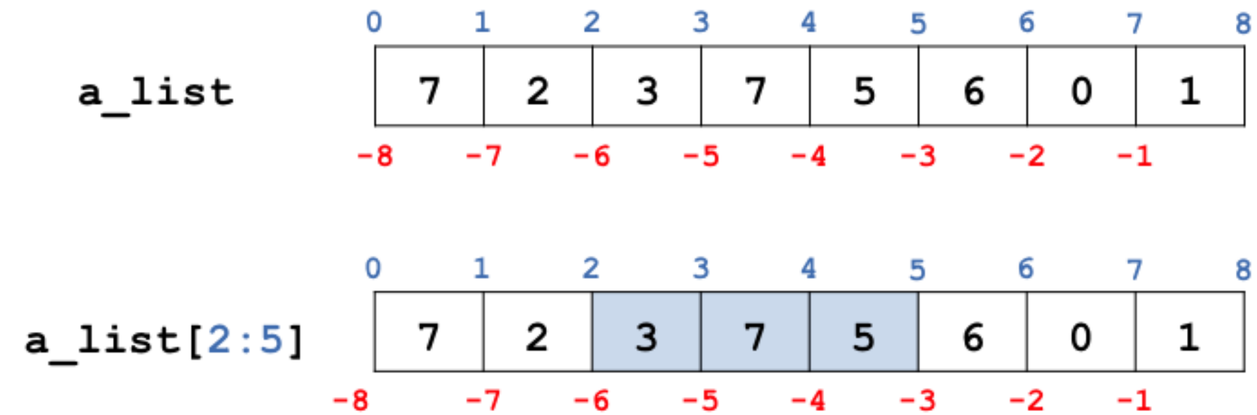
In [13]:
```
# Slicing without specifying the start index
# From the 1-st (index=0) to the 5-th (index=4) element
print(a_list[:5])
# Slicing without specifying the stop index
# From the 5-th (index=4) to the last (index=len(seq)-1)element
print(a_list[4:])
# Negative indices slice the sequence relative to the end
# Slice the last three elements
print(a_list[-3:])
# Slice the 7-th element from last (included) up to the 2-nd from last (excluded)
print(a_list[-7:-2])
```

```
[7, 2, 3, 7, 8]
[8, 6, 0, 1]
[6, 0, 1]
[2, 3, 7, 8, 6]
```

In [14]:
```
print(a_list)
# A step can also be used after a second colon to, say, take every other element
print(a_list[::2])
# A clever use of this is to pass -1, which has the useful effect of reversing the list
print(a_list[::-1])
```

```
[7, 2, 3, 7, 8, 6, 0, 1]
[7, 3, 8, 0]
[1, 0, 6, 8, 7, 3, 2, 7]
```

Again on slicing



Looping over a List

Accessing all the elements in a list

- One of the most important concepts related to lists.
- We use a **loop** (more on this later) to access *all* the elements in a list.
- A loop is a block of code that repeats itself until it runs out of items to work with, or until a certain condition is met.
- In this case, our loop will run once for every item in our list (e.g., if a list has three items, our loop will run three times).

In [15]:
```
# Define a list containing dog breeds
dogs = ['border collie', 'golden retriever', 'german shepherd']
# Print each dog breed contained in the list above
for dog in dogs:
    print(dog)
```

```
border collie
golden retriever
german shepherd
```

How does looping work?

- The keyword `for` tells Python to "get ready" to use a loop.
- The variable `dog` is a *temporary placeholder* variable where Python will place each item of the list, one at a time, at each loop iteration:
    - At the first iteration, `dog` references the string '`border collie`'.
    - At the second iteration, `dog` references '`golden retriever`'.
    - At the third iteration, `dog` references '`german shepherd`'.
    - Finally, after this there are no more items in the list, and the loop terminates.

In [16]:
```
# Once we hold a reference to a list item we are not just limited to print it!
# In fact, we can perform any supported operation on it
# For example, we can print it yet with its first letters capitalized
for dog in dogs:
    # We can call the string method title() on the current referenced string item
    # and use it within a predefined string pattern using the format method
    print('My favourite dog breed is: {0:s}'.format(dog.title()))

# Note that this statement is NOT indented as the previous one
# (i.e., it is outside the loop!)
# Therefore, it is printed ONLY once after the loop terminates
print('That\'s all I have to say about dogs!')
```

```
My favourite dog breed is: Border Collie
My favourite dog breed is: Golden Retriever
My favourite dog breed is: German Shepherd
That's all I have to say about dogs!
```

Enumerating a List

- When looping over a list, it might be useful to know the **index** of the current item.
- This can be achieved using `list.index(value)` syntax, but there is a simpler way.
- The `enumerate()` function tracks the index of each item for you, as it loops through the list.

In [17]:
```
# enumerate takes the sequence (list) as input
# and returns the index and the reference to the current item in the list
for index, dog in enumerate(dogs):
    print('My n.{0:d} favourite dog breed is: {1:s}'.format(index + 1, dog.title()))

# Note that this statement is NOT indented as the previous one
# (i.e., it is outside the loop!)
# Therefore, it is printed ONLY once after the loop terminates
print('That\'s all I have to say about dogs!')
```

```
My n.1 favourite dog breed is: Border Collie
My n.2 favourite dog breed is: Golden Retriever
My n.3 favourite dog breed is: German Shepherd
That's all I have to say about dogs!
```

List Comprehension

In [18]:
```
"""
Consider the following code snippet that, given a list of words, produces a new list
containing only those words containing at least 2 'a'
"""
# Input list of words
words = ['banana', 'kiwi', 'apple', 'melon', 'pineapple', 'papaya', 'strawberry', 'mango']
# Prepare the list containing the result
result = []
# Loop through all the words
for word in words:
    # Check if the current word contains at least 2 'a'
    if word.count('a') >= 2:
        # If so, just append it to the result list
        result.append(word)

# Finally, print the result (should be ['banana', 'papaya'])
print(result)
```

```
['banana', 'papaya']
```

In [19]:
```
# List comprehension allows us to write the same thing yet in a more compact way
# Let's start from scratch with an empty list (this step is not really needed)
result = []
# Using list comprehension you can do it in just a single line!
result = [word for word in words if word.count('a') >= 2]
# Finally, print the result
print(result)
```

```
['banana', 'papaya']
```

In [20]:
```
# Note that list comprehension works also when you have nested lists
# For example, consider the following list of lists
data = [['banana', 'kiwi', 'apple', 'melon'],['pineapple', 'papaya', 'strawberry', 'mango']]
# If you want to obtain a list of words starting with the letter 'm'
words_starting_with_m = [word for word_list in data for word in word_list
                         if word.startswith('m')]
# Finally, print the final list
print(words_starting_with_m)
```

```
['melon', 'mango']
```

Tuples: Type `tuple` (*immutable*)

Properties

- Tuples are basically ***immutable*** lists.
- Lists are great for containing highly dynamic information, as you can append/insert/remove/modify items in a list.
- However, sometimes we may want to ensure that no user nor part of a program can change a list. That's exactly what tuples are for!
- Allowed operations are the same as those of any other sequence type (i.e., `list`, `str`, `bytes`, etc.).

In [21]:
```
# Defining a tuple is like defining a list, except you use parentheses
# instead of square brackets
colors = ('red', 'green', 'blue')
# Once you have a tuple, you can access individual elements just like you can with a list...
print('The second color is: ' + colors[1])
# ... and you can loop through the tuple with a for loop:
print('\nHere is the list of primary colors:')
for color in colors:
    print(color.title())

# What happens if we try to add an item to the tuple?
colors.append('black')
```

```
The second color is: green

Here is the list of primary colors:
Red
Green
Blue
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-36c2c5feb193> in <module>()
     10
     11 # What happens if we try to add an item to the tuple?
---> 12 colors.append('black')

AttributeError: 'tuple' object has no attribute 'append'
```

Checkpoint Quiz

What happens if we try to do the following: `colors.sort()`, namely if we try to sort the tuple in-place?
And what would you expect to get if we did something like `sorted(colors)`?

In [22]:
```
colors.sort()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-22-f763f168a5eb> in <module>()
----> 1 colors.sort()

AttributeError: 'tuple' object has no attribute 'sort'
```

In [23]:
```
sorted(colors)
```

Out[23]:
```
['blue', 'green', 'red']
```

Hash Tables

- So far, we have seen data types which are able to store Python objects which are indexed by **integers**, such as `str` (*immutable*) or `list` (*mutable*).
- However, Python objects can also be collected into **hash tables**.
- Python provides two built-in types which corresponds to hash tables: `set` and `dict`

What is *hashing*?

- Hashing is the application of a **hash function**.
- A hash function maps a set of objects to a set of integers satisfying *some* properties.
- Any hash function h must be an *actual* function, that is, if two objects x and x' are the same (i.e., $x = x'$), then their hash should also be same, namely $h(x) = h(x')$.
- Also, a hash function should be easy to compute but hard to invert.

Domain *vs.* Codomain of a Hash Function

- Usually, the set of integers that the hash function maps to (i.e., the **codomain**) is **much smaller** than the set of objects (i.e., the **domain**)
- So that there will be multiple objects that hash to the same value (**hash collision**).
- In practice, hash functions operate on large enough codomains, and the function is designed so that if two objects hash to the same value, then they are *very likely* equal.

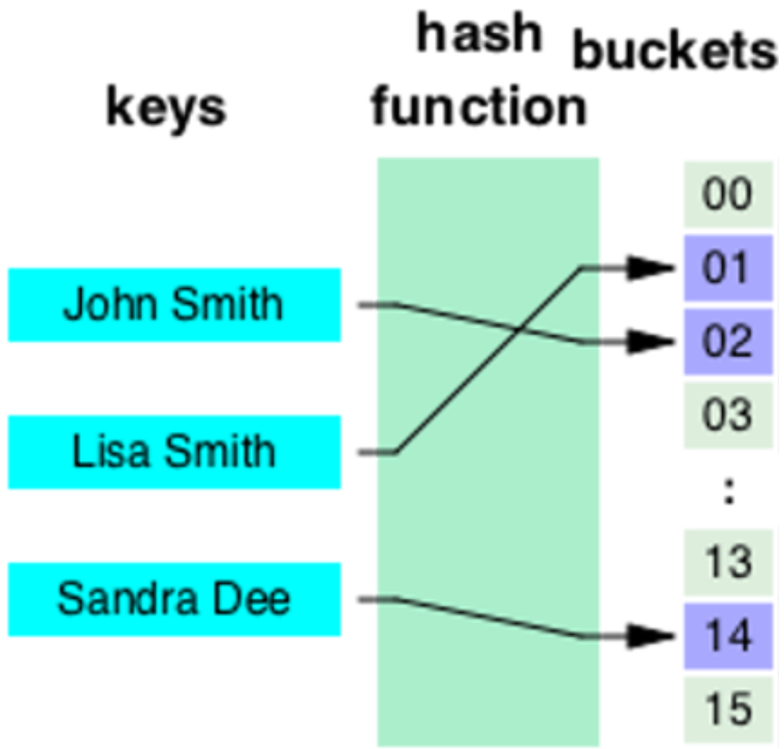
What are Hash Functions used for?

- We can leverage hash functions to organize a collection of objects into a new data structure, called **hash table**.
- Example: Suppose we have a collection of objects, and given *any* object, we want to be able to compute **very quickly** if that object belongs to our collection.
- **First solution**: Store objects in a list. But then to determine if an element is in the list, we might need to scan the *whole* list (time complexity $O(n)$, where n is the number of elements in the list).
- **Better solution**: Use hashing!

Hash Table

- Instead of storing the objects in a list, we create a list of "**buckets**", each one *indexed* by some hash value.
- We then compute the hash of each object, and store it into the list entry corresponding to its hash value.
- If there are more hash values than buckets (as usually is the case), we distribute them using a second hash function, which can be as simple as taking the modulus with respect to the number of buckets.

Hash Table



Source: [Wikipedia]

Lookup

- To determine if an object is in a hash table, we only have to hash the object, and look in the bucket corresponding to that hash.
- This is a $O(1)$ (i.e., constant time) operation which **does not** depend on the size of the input
- Of course, assuming the hash function **evenly** distributes objects in the available buckets (collisions).

Hashing in Python

- Python has a built-in function that performs a hash called `hash()`.
- For many objects, the hash is not very surprising.
- Python hashing depends on the architecture of the machine you are running on, and, in newer versions of Python, hashes are randomized for security purposes.

In [24]:
```
# Hashing an integer (immutable)
print("Hash of 42 is: {}".format(hash(42)))
# Hashing a string (immutable)
print("Hash of \"Aloha\" is: {}".format(hash("Aloha")))
# Hashing an empty tuple (immutable)
print("Hash of empty tuple () is: {}".format(hash(())))
```

```
Hash of 42 is: 42
Hash of "Aloha" is: 7768763457878087930
Hash of empty tuple () is: 3527539
```

In [25]:
```
# Not every Python object is hashable!
# Hashing a list (mutable)
print("Hash of list [1, 3, 5] is: {}".format(hash([1, 3, 5])))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-ea2ff4dcf778> in <module>()
      1 # Not every Python object is hashable!
      2 # Hashing a list (mutable)
----> 3 print("Hash of list [1, 3, 5] is: {}".format(hash([1, 3, 5])))

TypeError: unhashable type: 'list'
```

Hashability of Python Objects

- For an object to be hashable, it must be **immutable** (as well as any of its nested object)!
- This guarantees that the hash of an object remains the same across the object's lifetime.
- If the object is mutable and changes, then its hash will also have to change accordingly.
- This (design) restriction simplifies hash tables (i.e., `set` and `dict` introduced below), which otherwise should change the bucket where an object is store at runtime.

Sets: Type `set` (*mutable*)

Properties

- A *set* is an **unordered** collection of **unique** elements.
- Internally, they are stored into a hash table.
- A set can be created in two ways: using a *set literal* with curly braces or via the `set` function.

In [26]:
```
# Defining a set using curly braces
s = {3,5,6,5,5,2,1,4,3}
print(s)
# Defining a set using the 'set' built-in function
s = set([3,5,6,5,5,2,1,4,3])
print(s)
# Note that this means that we can also transform a list into a set
a_list = ['apple', 'kiwi', 'banana', 'apple', 'ananas', 'kiwi', 'pear', 'apple']
s = set(a_list)
print(s)
```

```
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
{'pear', 'kiwi', 'apple', 'banana', 'ananas'}
```

In [27]:
```
"""
Note that the following is legitimate because the objects used to create the set s
can be stored in an 'iterable' (like the mutable list below),
provided that each individual element in the iterable is hashable
(i.e., immutable) like the integers below.
"""
s = set([3,5,6,5,5,2,1,4,3])
print(s)
"""
But what if I do the following?
"""
s = set([[3,5,6,5],[5,2,1,4,3]])
print(s)
```

```
{1, 2, 3, 4, 5, 6}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-ab1b06aa92ec> in <module>()
     10 But what if I do the following?
     11 """
---> 12 s = set([[3,5,6,5],[5,2,1,4,3]])
     13 print(s)

TypeError: unhashable type: 'list'
```

Operations

Sets support mathematical set operations like **union**, **intersection**, **difference**, and **symmetric difference**. See Table below for a list of commonly used set methods.

Operations

Function	Alternate 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`.

Operations

`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.

In [28]:
```
# Defining two sets: A and B
A = {1, 2, 3, 4, 5}
B = {3, 4, 5, 6, 7, 8}
# Set Union (A or B)
print("Set Union: A \/ B = {}".format(A | B))
# Alternatively, you can invoke the 'union' method
print("Set Union: A \/ B = {}".format(A.union(B)))
# Set Intersection (A and B)
print("Set Intersection: A /\ B = {}".format(A & B))
# Alternatively, you can invoke the 'intersection' method
print("Set Intersection: A /\ B = {}".format(A.intersection(B)))
# Set Difference (A - B)
print("Set Difference: A - B = {}".format(A - B))
# Alternatively, you can invoke the 'difference' method
print("Set Difference: A - B = {}".format(A.difference(B)))
# Set Symmetric Difference (A xor B)
print("Set Symmetric Difference: A ^ B = {}".format(A ^ B))
# Alternatively, you can invoke the 'symmetric_difference' method
print("Set Symmetric Difference: A ^ B = {}".format(A.symmetric difference(B)))
```

```
Set Union: A \/ B = {1, 2, 3, 4, 5, 6, 7, 8}
Set Union: A \/ B = {1, 2, 3, 4, 5, 6, 7, 8}
Set Intersection: A /\ B = {3, 4, 5}
Set Intersection: A /\ B = {3, 4, 5}
Set Difference: A - B = {1, 2}
Set Difference: A - B = {1, 2}
Set Symmetric Difference: A ^ B = {1, 2, 6, 7, 8}
Set Symmetric Difference: A ^ B = {1, 2, 6, 7, 8}
```

Note on set operations

- Each of the logical set operations have **in place** counterparts.
- They can either be invoked using `op=` (where `op = {|, &, -, ^}`) or by calling the corresponding method with the `_update` suffix.
- Those replace the contents of the set on the left side of the operation with the result.
- For very large sets, this will be more efficient.

In [29]:
```
# Make a copy of set A
C = A.copy()
# In-place Set Union (C or B)
C |= B
print("Set Union: C \/ B = {}".format(C))
# Make another copy of set A
D = A.copy()
# In-place Set Intersection (D and B)
D &= B
print("Set Intersection: A /\ B = {}".format(D))
# ... similarly for the other operations
# Eventually, the original set A is unchanged
print("Original set A = {}".format(A))
```

```
Set Union: C \/ B = {1, 2, 3, 4, 5, 6, 7, 8}
Set Intersection: A /\ B = {3, 4, 5}
Original set A = {1, 2, 3, 4, 5}
```

In [30]:
```
# You can also check if a set is a subset of (is contained in)
# or a superset of (contains all elements of) another set
X = {1, 2, 3, 4, 5}
Y = {1, 3, 5}
print("'Y is subset of X' = {}".format(Y.issubset(X)))
print("'X is superset of Y' = {}".format(X.issuperset(Y)))
Y = {1, 3, 5, 7}
print("'Y is subset of X' = {}".format(Y.issubset(X)))
print("'X is superset of Y' = {}".format(X.issuperset(Y)))
# Finally, two sets are equal iff they have exactly the same content
Z = {5, 1, 7, 3}
print("'Y is equal to Z = {}'".format(Y == Z))
```

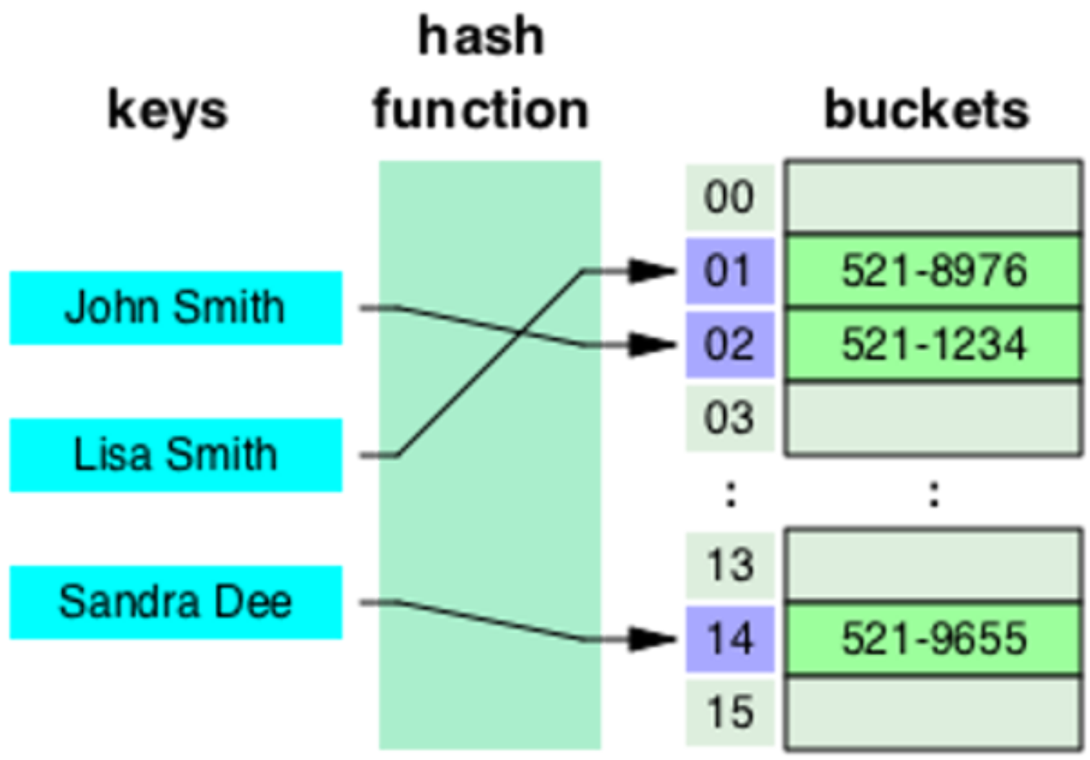
```
'Y is subset of X' = True
'X is superset of Y' = True
'Y is subset of X' = False
'X is superset of Y' = False
'Y is equal to Z = True'
```

Mappings: Type `dict` (*mutable*)

Properties

- Very likely, `dict` is the most important built-in Python data structure.
- A more common name for it is **hash map** or **associative array**.
- It is a hash table where each element of the hash table (**key**) points to another object (**value**); the object representing the value itself is not hashed.
- Keys and Values are of course Python objects! :)
- The easiest way to create one is by using curly braces `{}` and using colons to separate keys and values

Associative Array



Source: [Wikipedia]

In [31]:
```
# Create an empty dictionary
d = {}
# Define a dictionary containing some elements
d = {'a': 1, 'b': 2, 'c': [3, 4]}
# Values can be accessed/added/updated using the same list notation []
# Instead of accessing values by index (int), dictionary's values are accessed by key
# Retrieve the value associated with the key 'b' in the dictionary above
print("Retrieve the value associated with the key 'b' = {}".format(d['b']))
# Add a new value associated with a new key
d['z'] = 'some string'
print("After adding a new entry, the dictionary is: {}".format(d))
# Update the value associated with an existing key
d['a'] = (5, 42)
print("After updating the value of an existing entry, the dictionary is: {}".format(d))
# You can check if a dict contains a key using the same syntax
# as with checking whether a list or tuple contains a value
print("Q: The key 'c' is in the dictionary? A: {}".format('c' in d))
```

```
Retrieve the value associated with the key 'b' = 2
After adding a new entry, the dictionary is: {'a': 1, 'b': 2, 'c': [3, 4], 'z': 'some stri
ng'}
After updating the value of an existing entry, the dictionary is: {'a': (5, 42), 'b': 2,
'c': [3, 4], 'z': 'some string'}
Q: The key 'c' is in the dictionary? A: True
```

In [32]:
```
# Values can be deleted either using the 'del' keyword or the 'pop' method
# (the latter simultaneously returns the value and deletes the key)
del d['b']
print("After deleting an existing entry, the dictionary is: {}".format(d))
val = d.pop('z')
print("After popping out an existing entry, the dictionary is: {}".format(d))
print("The value popped out is: '{}'".format(val))
```

```
After deleting an existing entry, the dictionary is: {'a': (5, 42), 'c': [3, 4], 'z': 'som
e string'}
After popping out an existing entry, the dictionary is: {'a': (5, 42), 'c': [3, 4]}
The value popped out is: 'some string'
```

Useful methods: `keys` and `values`

- The `keys` and `values` methods give you iterators of the dictionary's keys and values, respectively as sets.
- While the key-value pairs are **not** in any particular order, these functions output the keys and values in the same order.

In [33]:
```
# Print the set of keys
print("The set of dictionary's keys is: {}".format(d.keys()))
# Print the set of values
print("The set of dictionary's values is: {}".format(d.values()))
# One dictionary can be merged into another using the update method (in-place)
d.update({'b' : 'foo', 'c' : 12})
print("After updating, the dictionary is: {}".format(d))
```

```
The set of dictionary's keys is: dict_keys(['a', 'c'])
The set of dictionary's values is: dict_values([(5, 42), [3, 4]])
After updating, the dictionary is: {'a': (5, 42), 'c': 12, 'b': 'foo'}
```

Creating Dictionaries from Sequences

In [34]:
```
# It’s common to end up with two sequences that you want to pair up element-wise in a dict.
# As a first cut, you might write code like this
# The list of keys
key_list = ['foo', 'bar', 'baz']
# The list of values
value_list = [15, 73, 42]
# Prepare the dictionary (at the beginning this is empty)
mapping = {}
# Populate the dictionary using the 'zip' function
# The 'zip' function takes two lists X = [x_1, ..., x_m] and Y = [y_1, ..., y_n]
# and returns a list of tuples [(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)],
# where k = min(m, n)
for key, value in zip(key_list, value_list):
    mapping[key] = value
print("The mapping dictionary is: {}".format(mapping))
```

```
The mapping dictionary is: {'foo': 15, 'bar': 73, 'baz': 42}
```

Checkpoint Quiz

What happens if the list of keys contains duplicates, i.e., if we change the definition of `key_list` as follows:

```
key_list = ['foo', 'bar', 'bar']
```

In [35]:
```
# A dictionary is roughly a collection of 2-tuples (one for the keys and one for the values),
# you can create one using the 'dict' type function and pass to it a list of 2-tuples
key_list = ('foo', 'bar', 'bar')
value_list = (15, 73, 42)
mapping = dict(zip(key_list, value_list))
print("The mapping dictionary is: {}".format(mapping))
```

```
The mapping dictionary is: {'foo': 15, 'bar': 42}
```

Default Values

In [36]:
```
# It’s quite common to have logic as follows:
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
# Luckily, the dict methods 'get' and 'pop' can take a default value to be returned,
# so that the above if-else block can be written simply as:
value = some_dict.get(key, default_value)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-36-3f280ab9c236> in <module>()
      1 # It’s quite common to have logic as follows:
----> 2 if key in some_dict:
      3     value = some_dict[key]
      4 else:
      5     value = default_value

NameError: name 'some_dict' is not defined
```

In [37]:
```
# Try to get the value associated with the key 'let',
# if this is not present fall back to 0
# 1. Using 'get'
value = mapping.get('let', 0)
print("Value returned = {}".format(value))
# 2. Using 'pop'
value = mapping.pop('let', 0)
print("Value returned = {}".format(value))
# If no default value is specified 'get' and 'pop' have 2 different behaviors:
# The 'get' method by default will return None if the key is not present
value = mapping.get('let')
print("Value returned = {}".format(value))
# ... whilst 'pop' will raise an exception.
value = mapping.pop('let')
print("Value returned = {}".format(value))
```

```
Value returned = 0
Value returned = 0
Value returned = None
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-37-102ec6e0ed3d> in <module>()
     12 print("Value returned = {}".format(value))
     13 # ... whilst 'pop' will raise an exception.
---> 14 value = mapping.pop('let')
     15 print("Value returned = {}".format(value))

KeyError: 'let'
```

In [38]:
```
# Another typical situation happens when trying to set values in a dictionary.
# Sometimes those values are other collections, like lists.
# Suppose you want to categorize a list of words by their first letters as a dict of lists.
# List of words
words = ['apple', 'bat', 'bar', 'atom', 'book', 'car', 'charlie', 'zoo']
# Initializing your empty dictionary
index = {}
# Loop through all the words in the list
for word in words:
    first_letter = word[0] # extract the first letter from the current word
    if first_letter not in index: # if the key (first_letter) is not in the dictionary
        index[first_letter] = [word] # just create a new entry, i.e., a list with one word
    else:
        # otherwise, append the current word to the list associated with the existing key
        index[first_letter].append(word)

print("The index dictionary is: {}".format(index))
```

```
The index dictionary is: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book'], 'c': ['ca
r', 'charlie'], 'z': ['zoo']}
```

In [39]:
```
# The if-else code block above can be easily rewritten using the 'setdefault' dict method.
# List of words
words = ['apple', 'bat', 'bar', 'atom', 'book', 'car', 'charlie', 'zoo']
# Initializing your empty dictionary
index = {}
# Loop through all the words in the list
for word in words:
    first_letter = word[0] # extract the first letter from the current word
    # either set an empty list ([]) with the current word
    # or append it to the existing entry
    index.setdefault(first_letter, []).append(word)

print("The index dictionary is: {}".format(index))
```

```
The index dictionary is: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book'], 'c': ['ca
r', 'charlie'], 'z': ['zoo']}
```

Valid Types for Dictionary Keys

- Although the **values** of a `dict` can be ***any*** Python object, the **keys** have to be **hashable**
- Therefore **keys** must be ***immutable*** objects like scalar types (`int`, `float`, `str`) or `tuple` (**note:** all the objects in the tuple need to be immutable, too!).
- Again, you can check whether an object is hashable (i.e., can be used as a key in a dictionary) with the `hash()` function.

In [40]:
```
# Check if an object of type str is 'hashable'
print(hash('string key'))
# Check if an object of type tuple is 'hashable'
print(hash((1, 2, (2, 3))))
# Check if a composite object of type tuple is 'hashable'
print(hash((1, 2, [2, 3]))) # fails because list are 'unhashable'
```

```
8606079362090932906
10976365022763477782

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-8d51b2f45daf> in <module>()
      4 print(hash((1, 2, (2, 3))))
      5 # Check if a composite object of type tuple is 'hashable'
----> 6 print(hash((1, 2, [2, 3]))) # fails because list are 'unhashable'

TypeError: unhashable type: 'list'
```

In [41]:
```
# 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.
d = {}
d[tuple([1, 2, 3])] = 'foo'
print(d)
d[tuple([1, 2, 1])] = 'bar'
print(d)
d[tuple([1, 2, [42, 73]])] = 'baz' # fails as the third element of the list is itself a list
```

```
{(1, 2, 3): 'foo'}
{(1, 2, 3): 'foo', (1, 2, 1): 'bar'}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-ab7bcf469625> in <module>()
      6 d[tuple([1, 2, 1])] = 'bar'
      7 print(d)
----> 8 d[tuple([1, 2, [42, 73]])] = 'baz' # fails as the third element of the list is its
elf a list

TypeError: unhashable type: 'list'
```

Summary

- In the last two lectures we have covered our (not-exhaustive) overview of Python's built-in data types.
- Python allows the programmer to easily define **new** data types (i.e., **classes**) supporting object-oriented paradigm.
- Still, the data types we have seen so far (or optimized variants of those) are the main building blocks, which most of the times are enough to develop your data science applications.
- Therefore, you are strongly encouraged to familiarize with them so that you know which are the most appropriate to use for achieving your specific task (https://wiki.python.org/moin/TimeComplexity)