CptS 111, Spring 2023
Lect. #7, Feb. 1, 2023
Class Notes

Today's Agenda:

1. Data structures in Python
2. Lists
3. Tuples
4. Dictionaries

Ch. 3 (cont.)

1. Python's Data Structures (strings, lists, tuples, sets, and dictionaries)

Thus far we've worked with integers, floats, and strings. Python also provides four other variable types which are called data structures because of the special actions that can be performed with them. Actually, in Python, strings are also a data structure which makes them very easy to manipulate and is one of the reasons why Python is so popular.

In addition to strings, the four data structures are lists, tuples, sets, and dictionaries. Before discussing these, let's define some new terms:

- A ***container*** is something that ***contains*** literals or variables (or even data structures).
- An ***iterable*** is a data type that can be iterated :); we'll use these in `for` -loops which we'll cover in Ch. 6.
- A ***sequence*** is an ordered iterable that allows items to be indexed.

Because lists, tuples, set, and dictionaries all contain other variables, they're all containers. Let's compare these different containers together with strings.

Data Structure	Iterable	Sequence	Mutable	Creation	Example
string	yes	yes	no	`str()`	'Go, Cougs!'
list	yes	yes	yes	`list()`	[1, 1.618, ['a', 'b']]
tuple	yes	yes	no	`tuple()`	(1, 1.618, ['a', 'b'])
set	yes	no	yes	`set()`	{1, 1.618, 'a', 'b'}
dictionary	yes	no	yes	`dict()`	{1:'one', 'two':2, (1, 2):'tuple'}

Recall that if something is **mutable**, it can be changed; if it is **immutable**, it can't be changed.

Lists, tuples, sets, and dictionaries are all containers that are iterables. However, they aren't all sequences or mutable. Strings are iterable sequences that are immutable.

We won't use sets very much or cover named tuples in CptS 111 (other than in the reading assignments you've completed), but it's good for you to know about them.

2. Lists

1) About lists:

- together with strings, are probably the most useful variable type in Python
- are sequences so can index like strings
- indexes are always enclosed by square brackets `[ ]`
- indexing always begins at 0
- are mutable so can change values
- can have lists of lists, lists of tuples, mixed lists, and so forth

In [1]:
```
# List example with a commonly used index value

prime_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
last_prime = prime_list[-1]
print(last_prime)
```

```
43
```

An aside:

- ***Functions*** and ***methods*** are both ways of performing actions.
- Functions are usually broader in scope than methods.
- Both functions and methods may or may not have arguments.
- Functions stand on their own, but methods follow objects preceded by a dot.
- Methods are typically associated with a specific data structure.
- ***You must spend time memorizing which commands are functions and which are methods!***

Some list functions and methods:

- `len()` : finds list length, i.e., the number of elements in a list
- `max()` : finds maximum value
- `min()` : finds minimum value
- `.append()` : adds element to end of a list <- **very useful!**
- `.pop()` : removes element w/ given index or last element w/ no index
- `.remove()` : removes first element w/ a given value
- `.sort()` : sorts list from lowest to highest value
- `.count()` : provides count of number of occurrences of given value

2) To initialize an empty list, we use, e.g.,:

```
nums = []
```

In [2]:
```
# Initializing an empty list

names = []
print(names)
```

```
[]
```

3) We can use operator overloading to combine two lists.

In [4]:
```
# Combining two lists using operator overloading
# Result is order dependent!

primes1 = [2, 3, 5, 7]
primes2 = [11, 13, 17]
primes = primes2 + primes1
print('primes:', primes)
```

```
primes: [11, 13, 17, 2, 3, 5, 7]
```

4) We use indexing to change the value of an element.

In [5]:
```
# We can change list elements because LISTS ARE MUTABLE

print('primes1:', primes1)
print('primes1[0]:', primes1[0])  # Print first element
primes1[0] = 11                    # Change first element
print()

print('primes1:', primes1)
print('primes1[0]:', primes1[0])  # Print first element
```

```
primes1: [2, 3, 5, 7]
primes1[0]: 2

primes1: [11, 3, 5, 7]
primes1[0]: 11
```

5) Examples for some list functions:

In [6]:
```
# Note well!
# Use sum() and len() to find _average_ value of numbers in list

avg = sum(primes) / len(primes)
print(avg)
```

```
8.285714285714286
```

In [7]:
```
# Use max() and min() to find maximum and minimum values

print(max(primes))
print(min(primes))
```

```
17
2
```

6) Examples for some list methods; note the differences between how functions and methods are used.

In [8]:
```
# Use .append() method to add 5 to end of list; we'll use this method a lot!

primes.append(5)  # name of list before dot!
print(primes)
```

```
[11, 13, 17, 2, 3, 5, 7, 5]
```

In [9]:
```
# Use .count() method to find number of 5's in list

primes.count(5)
```

Out[9]:
```
2
```

In [10]:
```
# Use .pop() method wo/ arg to remove last element

primes.pop()
primes
```

Out[10]:
```
[11, 13, 17, 2, 3, 5, 7]
```

In [11]:
```
# Use .append() method again; note that element is placed
# at the end

primes.append(2)
primes
```

Out[11]:
```
[11, 13, 17, 2, 3, 5, 7, 2]
```

In [12]:
```
# Use .pop() method w/ arg to remove first element

primes.pop(0)
primes
```

Out[12]:
```
[13, 17, 2, 3, 5, 7, 2]
```

In [13]:
```
# Use .append() method again

primes.append(2)
primes
```

Out[13]:
```
[13, 17, 2, 3, 5, 7, 2, 2]
```

In [14]:
```
# Use .remove() method to remove _first_ occurrence of 2

primes.remove(2)
primes
```

Out[14]:
```
[13, 17, 3, 5, 7, 2, 2]
```

In [15]:
```
# Use .sort() method to sort list from low to high values

primes.sort()
primes
```

Out[15]:
```
[2, 2, 3, 5, 7, 13, 17]
```

Actions performed on lists change the lists "in place" which means when you perform an action on a list, you lose the original list.

3. Tuples

About tuples:

- are very similar to lists
- **major difference is that tuples are immutable**
- are very common in Python
- often use is hidden because they're easily unpacked (more about unpacking later)
- can use many of the functions and methods used with lists

In [16]:
```
# We can use some of the functions with tuples that we used
# with lists

primes = (2, 3, 5, 7, 11, 13)
print('Max is:', max(primes))
print('Length is:', len(primes))
print('2 count is:', primes.count(2))  # Method!
print('Sum is:', sum(primes))
```

```
Max is: 13
Length is: 6
2 count is: 1
Sum is: 41
```

However, we can't use any of the functions or methods that alter a tuple. Why? Also, while we can initialize an empty tuple, it isn't of any practical use. Why? However, we can use operator overloading to create a new tuple.

In [17]:
```
# We can't change a tuple, but we can use operator overloading
# with tuples

evens = (2, 4, 6, 8, 10)
odds = (1, 3, 5, 7, 9)
nums = odds + evens
nums
```

Out[17]:
```
(1, 3, 5, 7, 9, 2, 4, 6, 8, 10)
```

In [18]:
```
# Can't sort because tuples are immutable!

nums.sort()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [18], in <cell line: 3>()
      1 # Can't sort because tuples are immutable!
----> 3 nums.sort()

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

4. Dictionaries

Dictionaries are useful data structures, and we'll have more to say about them in a later lecture. ***For now, we'll cover the material that you need for PA #5.***

About dictionaries:

- differ from lists and tuples because they aren't sequences -> can't be indexed
- are made up of key-value pairs {key1: value1, key2: value2}
- keys must be immutable
- values can be either immutable or mutable

To initialize an empty dictionary, we simply do the following, e.g.,

In [19]:
```
# Initialize an empty dictionary

faves = {}
faves     # Empty!
```

Out[19]:
```
{}
```

Then to add to the dictionary, we use new key-value pairs. **Important: To add these, we use the form >>> `dict_name[key] = value` <<< .**

In [20]:
```
# Dictionary values can be anything: lists, tuples, dictionaries, ...
# Let's add a list value ('goodies' is the key; the list is the value)

faves['goodies'] = ['cookies', 'HD coffee ice cream', 'scones'] # value is list
faves     # goodies is the key; the value occurs after the colon
```

Out[20]:
```
{'goodies': ['cookies', 'HD coffee ice cream', 'scones']}
```

In [21]:
```
# Add another list value; here we assign the list to the lvalue nums and
# then use the lvalue as the value

nums = [42, 1.618, 3.141592653]
faves['numbers'] = nums         # value is a list
faves     # goodies and numbers are the keys
```

Out[21]:
```
{'goodies': ['cookies', 'HD coffee ice cream', 'scones'],
 'numbers': [42, 1.618, 3.141592653]}
```

In [22]:
```
# Add an integer value

faves['integer'] = 88       # value is an integer
faves     # goodies, numbers, and integer are the keys
```

Out[22]:
```
{'goodies': ['cookies', 'HD coffee ice cream', 'scones'],
 'numbers': [42, 1.618, 3.141592653],
 'integer': 88}
```

In [23]:
```
# This dictionary value is a string literal ('husband' is the key)

faves['husband'] = 'John Schneider'  # value is a string
faves     # We've added a new key-value pair: 'husband': 'John Schneider'
```

Out[23]:
```
{'goodies': ['cookies', 'HD coffee ice cream', 'scones'],
 'numbers': [42, 1.618, 3.141592653],
 'integer': 88,
 'husband': 'John Schneider'}
```

Notice that a value can be any type of variable (including another dictionary). We can change a value using a key.

In [24]:
```
# USE A KEY TO CHANGE A VALUE

faves['integer'] = 99  # Can think of a key as being a substitute for an index
faves     # The value for the key 'integer' has been changed from 88 to 99
```

Out[24]:
```
{'goodies': ['cookies', 'HD coffee ice cream', 'scones'],
 'numbers': [42, 1.618, 3.141592653],
 'integer': 99,
 'husband': 'John Schneider'}
```

In [25]:
```
# What am I doing here? Hint: .append() is used with lists

faves['goodies'].append('chocolate croissants')
faves

# faves['goodies'] gives the value which is a list; we can append
# an element to a list
```

Out[25]:
```
{'goodies': ['cookies',
  'HD coffee ice cream',
  'scones',
  'chocolate croissants'],
 'numbers': [42, 1.618, 3.141592653],
 'integer': 99,
 'husband': 'John Schneider'}
```

For PA #5, you'll also need to use the `in` command as part of a test statement to see whether a key exists in your dictionary.

In [26]:
```
if 'goodies' in faves:
    print('My favorite goodies are:', faves['goodies'])
```

```
My favorite goodies are: ['cookies', 'HD coffee ice cream', 'scones', 'chocolate croissants']
```

We remove key-value pairs using the `del` (for delete) command.

In [27]:
```
# Use del to delete a key-value pair

del faves['goodies']
faves
```

Out[27]:
```
{'numbers': [42, 1.618, 3.141592653],
 'integer': 99,
 'husband': 'John Schneider'}
```

Finally, we can remove all the entries from a dictionary using the `.clear()` method. Be careful with this method if your dictionary is large. You don't want to wipe out a dictionary you wanted to keep!

In [28]:
```
# Use the .clear method to empty the dictionary of everything

faves.clear()
faves     # And we're back to our empty dictionary
```

Out[28]:
```
{}
```