NIAPythonDay2

May 16, 2017

NIA Intro to Python Class - May 16, 2017

1 Day 2: Iterables and Operators

- An iterable is a container object capable of returning its members one at a time.
- An operator is a type of function that returns a value based on the values its next to.
- In this notebook we explore of some of the most important iterable types built into Python.

1.0.1 Table Of Contents

1.1 Lists

- Container for a group of values
- Can all be the same type or different, doesn't matter.
- Declared by bracket outside, comma delimited within.
- The order of the values in the list is remembered
- Is mutable = Can append/insert new values to the list, and remove values.
- As with all iterables in Python, the values in a list are indexed starting with 0.

```
In [1]: a_list = [1, 2, 3, "a dog"]

In [2]: a_list

Out[2]: [1, 2, 3, 'a dog']
```

1.1.1 Append another value to the list

```
In [3]: a_list.append( "Hi there!" )

In [4]: a_list

Out[4]: [1, 2, 3, 'a dog', 'Hi there!']
```

1.1.2 Get the Nth value out of a list

Use open/close brackets [] to get at the Nth thing in the list.

```
In [5]: a_list[2]

Out[5]: 3
```

1.1.3 Delete a value in the list

use the keyword del to delete an item in the list.

```
In [6]: del a_list[2] # Remember, the count starts at 0

In [7]: a_list

Out[7]: [1, 2, 'a dog', 'Hi there!']
```

1.1.4 Empty list

A list can have one or zero values in it.

```
In [8]: empty_list = []

In [9]: len( empty_list )

Out[9]: 0

In [10]: empty_list.append( "sumn" )

In [11]: len( empty_list )

Out[11]: 1
```

1.2 Tuples

- The name "tuple" comes from math concept, i.e., quadruple, quintuple, sextuple, ... n-tuple
- A tuple is just an immutable list: once you make it, you can't change it.
- The order of the values in the list is remembered.
- Usually declared by surrounding a comma-separated sequence with optional parentheses

```
In [12]: a_tuple = 1, 2, 3, "four"

In [13]: readable_tuple = ( 1, 2, 3, "four" ) # Parentheses are optional
```

```
In [14]: empty_tuple = ()

In [15]: tuple_with_one_item = ("an item",)

In [16]: len( tuple_with_one_item )

Out[16]: 1

In [17]: # Index notation works for tuples too
         a_tuple[3]

Out[17]: 'four'
```

1.3 Dictionaries

- A dict is associative array, where "keys" are mapped to "values."
- DOES NOT KEEP TRACK OF ORDER OF KEY-VALUE PAIRS (for that you need collections.OrderedDict)

1.3.1 Create an empty dict

Declare empty dict with {}, or dict().

```
In [18]: a_dict = {}

In [19]: type( a_dict )

Out[19]: dict
```

1.3.2 Create a dict with stuff in it

The keys are separated by the values by a colon (:), and the key-value pairs are separated by commas.

```
In [20]: a_dict = {"key 1":"value 1", 2:3, 4:['a','list'],
                   "expects spanish inquisition": "hi"}
```

To get a value out of a dict, put the key into the brackets.

```
In [21]: print( a_dict['key 1'], ",", a_dict['expects spanish inquisition'] )

value 1 , hi
```

1.3.3 Get a list of all the keys of a dict

Return a list of keys by calling the .keys() function on any dict.

```
In [22]: a_dict.keys()

Out[22]: dict_keys(['key 1', 2, 4, 'expects spanish inquisition'])
```

1.3.4 Get a list of all the values of a dict

Use .values()

```
In [23]: # Return a list of values:
         a_dict.values()
```

```
Out[23]: dict_values(['value 1', 3, ['a', 'list'], 'hi'])
```

1.3.5 KeyError exception

If you reference a key that's not there, a KeyError exception is raised

```
In [24]: a_dict[3]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)

<ipython-input-24-30b56d6b7ab8> in <module>()
----> 1 a_dict[3]


KeyError: 3
```

1.3.6 Advanced topic: key-value directionality

dicts only go in one direction, i.e., you can't put in a value and get out a key. Keys map to values, but not vice versa

Here's a little code snipped reversing the directionality using a something called a dict comprehension

```
In [25]: keys_to_values = {1:'a', 2:'b', 3:'c', 4:'d'}
```

```
In [26]: keys_to_values[3]
```

```
Out[26]: 'c'
```

```
In [27]: keys_to_values['c']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)

<ipython-input-27-a9eaa25dc3f2> in <module>()
----> 1 keys_to_values['c']


KeyError: 'c'
```

```
In [28]: values_to_keys = \
             { (second, first) for first, second in keys_to_values.items() }

In [29]: values_to_keys

Out[29]: {('a', 1), ('b', 2), ('c', 3), ('d', 4)}
```

1.4 Sets

- Similar to math concept of sets; has operations like union, intersection, etc.
- Sets are unindexed, unordered, and contains no duplicates.
- My personal favorite of the Python standard types!

1.4.1 Create an empty set

Make an empty using set().

```
In [35]: empty_set = set() # not {}, that would be an empty dict

In [36]: len(empty_set)

Out[36]: 0
```

1.4.2 Create a set with stuff in it

Declare a set by putting values inside braces.

```
In [33]: a_set = {'set', 'of', 'words'}

In [34]: some_set = {0, (), False}

In [10]: type( some_set )

Out[10]: set
```

1.4.3 The Union operator for sets

The union: all unique things in the Venn diagram. All regions.
Use the pipe character | to take the union of two sets.

```
In [37]: set1 = {12,34,56,78,90,42}

In [38]: set2 = {1,23,45,67,89,42}

In [39]: set3 = set1 | set2

In [40]: set3

Out[40]: {1, 12, 23, 34, 42, 45, 56, 67, 78, 89, 90}
```

1.4.4 The Intersection operator for sets

The intersections is just the overlapped region in the Venn diagram. Use the ampersand (&) operator.

```
In [41]: set4 = set1 & set2

In [42]: set4

Out[42]: {42}
```

1.5 Slicing Iterables

You can use the slice notation on list, tuples and strings. Strings are like a tuple of chararcters.

```
In [43]: full_statement = \
             'The answer to life, the universe, and everything is 42'
```

1.5.1 Subsets: slicing iterables into smaller ones

Return a substring using a brackets separated by a colon.

```
In [44]: full_statement[3:22]

Out[44]: ' answer to life, th'
```

1.5.2 Slicing an iterable doesn't change the original iterable

Just because you just returned a substring from a string doesn't mean you changed the original string. Python created a new string and returned that
full_statement

1.5.3 Slicing syntax

[begin index:end index:step]

```
In [45]: full_statement[3:32:3] # take every 3rd letter

Out[45]: ' sroi,huvs'
```

1.5.4 Slice from the beginning to the middle somewhere

Leave out the start index and Python assumes you want a slice starting from the beginning.

```
In [76]: full_statement[:25]

Out[76]: 'The answer to life, the u'
```

1.5.5 Slice from the middle somewhere to the end

Leave out the end index and Python assumes you want a slice that goes straight to the end.

```
In [77]: full_statement[25:]
```

```
Out[77]: 'niverse, and everything is 42'
```

1.5.6 Negative slice indices mean count from the end

If i is negative, index is relative to end of string:

```
In [ ]: full_statement[-25:]
```

1.5.7 Reverse a the order of an iterable using the step parameter

Reverse a string by using a negative step value

```
In [54]: "a man, a plan, a canal, panama"[::-1]
```

```
Out[54]: 'amanap ,lanac a ,nalp a ,nam a'
```

1.6 Iterables of Iterables

- It's fine as long as you don't violate the mutable rules

```
In [ ]: list1 = [1,2,3,4,5]
```

```
In [ ]: list2 = [6,7,8,9,10]
```

```
In [ ]: tuple1 = list1, list2
```

```
In [62]: tuple1
```

```
Out[62]: ([1, 2, 3, 4, 5], [6, 7, 8, 9, 10])
```

```
In [63]: # I guess this is legal:
         tuple1[0][0] = 99 # compound index notation
         tuple1
```

```
Out[63]: ([99, 2, 3, 4, 5], [6, 7, 8, 9, 10])
```

```
In [ ]: dict1 = {'list1': list1, 'list2': list2}
```

```
In [ ]: dict1['list2'][4] = 66
```

```
In [ ]: dict1['list1'].append( 555 )
```

```
In [64]: dict1
```

```
Out[64]: {'list1': [99, 2, 3, 4, 5, 555], 'list2': [6, 7, 8, 9, 66]}
```

1.7 Operators and Operations on Iterables

```
In [48]: test_list = [1, 2, "3", "four", (5,), set((6,))]
```

1.7.1 The in operator

"x in s" - a boolean expression to test if the value or substring x is in iterable s.

```
In [49]: 2 in test_list
```

```
Out[49]: True
```

1.7.2 The not in operator

"x not in s" - Opposite of in.

```
In [50]: 2 not in test_list
```

```
Out[50]: False
```

1.7.3 Concatenate iterables

The plus operator works on some iterable types but not others

```
In [51]: [1,2,3] + [4,5,6]
```

```
Out[51]: [1, 2, 3, 4, 5, 6]
```

```
In [52]: "abc" + "def"
```

```
Out[52]: 'abcdef'
```

1.7.4 The len() operator

len() returns the length of an iterable.

```
In [53]: len( test_list )
```

```
Out[53]: 6
```

1.7.5 The math operators

- <
- <=
-
- >=
- ==

Note the double equal signs is an operator, not an assignment!!

1.7.6 The min() and max() operators

Gives the minumum/maximum value. Uses the math operators to compare.

```
In [56]: min(test_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-56-f15a2238f6de> in <module>()
----> 1 min(test_list)


TypeError: '<' not supported between instances of 'str' and 'int'
```

1.7.7 The .count() method

some_list.count(some_value) The number of times some_value appears in some_list

```
In [61]: my_new_list = [0,1,2,3,3,2,1]

In [58]: my_new_list.count( 3)

Out[58]: 2
```

1.8 Basic Sorting

- Use the sorted function to sort basic Python iterable types WITHOUT modifying the original.
- Use the .sort() method to sort an iterable in place.

```
In [59]: sorted( my_new_list )

Out[59]: [0, 1, 1, 2, 2, 3, 3]

In [62]: my_new_list

Out[62]: [0, 1, 2, 3, 3, 2, 1]

In [63]: my_new_list.sort()

In [64]: my_new_list

Out[64]: [0, 1, 1, 2, 2, 3, 3]
```