08.2_Lists

March 22, 2023

1 Introduction to Python for Open Source Geocomputation

- Instructor: Dr. Wei Kang
- Class Location and Time: ENV 336, Mon & Wed 12:30 pm - 1:50 pm

Content:

- Lists - continued
- Tuples

2 Standard Data Types in Python - lists

Category of Data type	Data type	Example
Numeric, scalar	Integer	1
	Floats	1.2
	Complex	1.5+0.5j
	Booleans	True
Container	strings	"Hello World"
	List	[1, "Hello World"]
	Tuple	(1, "Hello World")
	Set	{1, "Hello World"}
	Dictionary	{1: "Hello World", 2: 100}

2.1 `in` operator with lists

similar to the behavior when working with strings

- Check whether an item/element occurs in a given list
- Returns a boolean value

2.1.1 Group Exercise

Write python code to check whether `3` occurs in the list `[1,2,3]`

When you are done, raise your hand!

[1]:
```
3 in [1,2,3]
```

[1]: True

[2]:
```
3 in [1,2,3]
```

[2]: True

[3]:
```
3.0 in [1,2,3]
```

[3]: True

2.1.2 Comparison operators across integers and floats

Numbers of built-in numeric types (Numeric Types — **int, float, complex**) and of the standard library types fractions.Fraction and decimal.Decimal can be compared **within and across their types***, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

Compare numerical values rather than focusing on data types

reference: https://docs.python.org/3/reference/expressions.html#value-comparisons

[4]:
```
3 == 3.0
```

[4]: True

[5]:
```
3.0 in [1,2,3]
```

[5]: True

[6]:
```
3 in [1,2,3.0]
```

[6]: True

[7]:
```
3 in [1,2,"3"]
```

[7]: False

Two ways to check whether an object is of the specified type:

- `type(2) == int`
- `isinstance(2, int)`

[8]:
```
type(2) == int
```

[8]:
```
True
```

[9]:
```
isinstance(2, int)
```

[9]:
```
True
```

2.2 Lists are Nestable

- A list can be an element/item in another list

[10]:
```
list_a = [1, "happy", 1+9j, 2.3, True]
list_b = [list_a, [1,2], True]
list_b
```

[10]:
```
[[1, 'happy', (1+9j), 2.3, True], [1, 2], True]
```

Indexing and slicing nested lists:

[11]:
```
list_b[0] #the first element is a list
```

[11]:
```
[1, 'happy', (1+9j), 2.3, True]
```

[12]:
```
list_intermediate = list_b[0]
list_intermediate[1]
```

[12]:
```
'happy'
```

[13]:
```
list_b[0][1]
```

[13]:
```
'happy'
```

Since the first element is a list, we can use indexing to get the second element of this list

[14]:
```
list_b[0][1]
```

[14]:
```
'happy'
```

Since the second element of the first element of `list_b` is a string which is ordered, we can use slicing to get the second and the third elements of the string:

[15]:
```
list_b
```

[15]:
```
[[1, 'happy', (1+9j), 2.3, True], [1, 2], True]
```

[16]:
```
list_b[0][1][2:4]
```

[16]:
```
'pp'
```

[17]:
```
list_b
```

[17]:
```
[[1, 'happy', (1+9j), 2.3, True], [1, 2], True]
```

[18]:
```
list_b[2]
```

[18]:
```
True
```

[19]:
```
list_b[2:2]
```

[19]:
```
[]
```

[20]:
```
list_b[2:5]
```

[20]:
```
[True]
```

2.2.1 Robust start and end indices with python slicing `[start:end]`

- **robust end index**: The above slicing on list (`list_b[2:4]`) works even though the end index slicing is greater than the maximal sequence index. This is because python scling has **robust end index**. The slice just takes all elements up to the maximal element.
- If the start index is out of bounds as well, it returns the empty slice. So `list_b[4:]` will return an empty list.

[21]:
```
list_b
```

[21]:
```
[[1, 'happy', (1+9j), 2.3, True], [1, 2], True]
```

[22]:
```
list_b[4:]
```

[22]:
```
[]
```

[23]:
```
a = list_b[0]
b = a[1]
b[2:4]
```

[23]:
```
'pp'
```

2.2.2 Group Exercise (nested list)

1. Write python code to obtain the second element of the nested list `list_b = [[1, 'happy', (1+9j), 2.3, True], [1, 2], True]`
2. Write python code to obtain the second element of the second element (a list) of the nested list `list_b = [[1, 'happy', (1+9j), 2.3, True], [1, 2], True]`:

When you are done, raise your hand!

[24]:
```
list_b = [[1, 'happy', (1+9j), 2.3, True], [1, 2], True]
```

[25]:
```
list_b[1]
```

[25]:
```
[1, 2]
```

[26]:
```
list_b[1][1]
```

[26]:
```
2
```

2.2.3 `in` operator with nested lists

- check whether an object is an element of a given list

[27]:
```
list_a = [3, [1, 'happy', (1+9j), 2.3, True], [1, 2], False]
```

[28]:
```
1 in list_a
```

[28]:
```
False
```

[29]:
```
[1,2] in list_a
```

[29]:
```
True
```

[30]:
```
list_a[1][1][0]
```

[30]:
```
'h'
```

[31]:
```
list_b = [[1, 'happy', (1+9j), 2.3, True], [1, 2], False]
```

[32]:
```
'happy' in list_b
```

[32]:
```
False
```

[33]:
```
'happy' in list_b[0]
```

[33]:
```
True
```

[34]:
```
list_b
```

[34]:
```
[[1, 'happy', (1+9j), 2.3, True], [1, 2], False]
```

[35]:
```
[1, 'happy', (1+9j), 2.3, True] in list_b
```

[35]:
```
True
```

[36]:
```
'happy' in list_b[0]
```

[36]:
```
True
```

[37]:
```
list_b[0]
```

[37]:
```
[1, 'happy', (1+9j), 2.3, True]
```

2.3 Lists are mutable

Unlike strings, however, lists are mutable - item assignment is feasible

[38]:
```
list_a = [1, "happy", True]
```

[39]:
```
list_a[1] = 2
```

[40]:
```
list_a
```

[40]:
```
[1, 2, True]
```

[41]:
```
list_a[2] = 3
```

[42]:
```
list_a
```

[42]:
```
[1, 2, 3]
```

2.3.1 List item assignment with slicing (assign values to a slice of list)

[43]:
```
list_a = [1, "happy", True]
```

[44]:
```
list_a[1:]
```

[44]:
```
['happy', True]
```

[45]:
```
list_a[1:] = [2,3]
list_a
```

[45]:
```
[1, 2, 3]
```

[46]:
```
list_a[1:] = [1]
list_a
```

[46]:
```
[1, 1]
```

[47]:
```
list_b = [1,2,3,4,5,6,1,2,46]
list_b
```

[47]:
```
[1, 2, 3, 4, 5, 6, 1, 2, 46]
```

[48]:
```
list_b[1:] = [100, 101]
list_b
```

[48]:
```
[1, 100, 101]
```

2.4 List Operations

- Concatenation with + (similar to strings)
- Repetition with * (similar to strings)

[49]:
```
"happy" + "happy"
```

[49]:
```
'happyhappy'
```

[50]:
```
list_a = [1, "happy", True]
list_a
```

[50]:
```
[1, 'happy', True]
```

[51]:
```
list_a + list_a
```

[51]:
```
[1, 'happy', True, 1, 'happy', True]
```

[52]:
```
[1, 2, 3] + [2, 3, 4]
```

[52]:
```
[1, 2, 3, 2, 3, 4]
```

[53]:
```
"happy" * 3
```

[53]:
```
'happyhappyhappy'
```

[54]:
```
list_a
```

[54]:
```
[1, 'happy', True]
```

[55]:
```
1 > "happy"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[55], line 1
----> 1 1 > "happy"

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

```
[ ]: list_a * 3
```

```
[ ]: max(list_a)
```

```
[ ]: sum([1,2,3])
```

```
[ ]: list_a
```

```
[ ]: sum(list_a)
```

```
[ ]: max([1,2,3])
```

```
[ ]: sum([1,2,3])
```

In a list with elements as string, **max( )** and **min( )** is applicable. **max( )** would return a string element whose ASCII value is the highest and the lowest when **min( )** is used. Note that only the first index of each element is considered each time and if they value is the same then second index considered so on and so forth.

```
[ ]: mlist = ['bzaa','ds','nc','az','z','klm']
```

```
[ ]: max(mlist)
```

```
[ ]: 'bzaa' > 'z'
```

```
[ ]: min(mlist)
```

```
[ ]: nlist = [1,94,93,1000]
```

```
[ ]: max(nlist)
```

```
[ ]: nlist = ['1','94','93','1000']
```

```
[ ]: max(nlist)
```

```
[ ]: min(nlist)
```

Even if the numbers are declared in a string the first index of each element is considered and the maximum and minimum values are returned accordingly.

2.5 Further readings

- Lists and Tuples in Python

2.6 Assginments

- HW4
- Mid-term exam

[ ]: