Introduction to Python

Built-In Sequences: Lists

Topics

1) Lists
   a) Indexing and Slicing
   b) List Methods
3) List comprehensions
4) Iterables
5) Membership and Operations

Containers

Python includes several built-in sequences: *lists, tuples, strings*.

We discussed strings in the previous lecture. String is a flat sequence which holds item of one type. Flat sequences physically store the value of each item within its own memory space.

Lists and *tuples* are *container sequences*, which can hold items of different type. They hold references to objects they contain(more on this later).

Another way of grouping sequence types is by *mutability*. Lists are *mutable(can be modified)* sequences while strings and tuples are *immutable* sequences. We discuss lists in this lecture and tuples in the next.

Lists

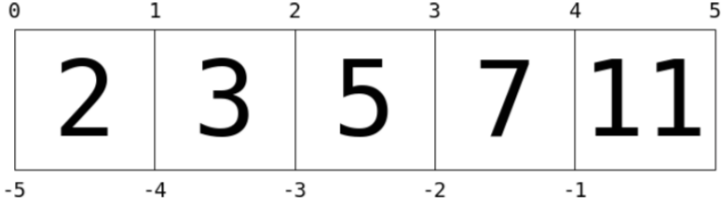
Lists are the basic *ordered* and *mutable* data collection type in Python. They can be defined with comma-separated values between square brackets.

```
In[1]:L = [2, 3, 5, 7]


In [2]: len(L) # len also worked with strings
Out [2]: 4


In [3]: L.append(11)  # append to the end of the list
In [4]: L
Out [4]: [2, 3, 5, 7, 11]
```

Indexing

Indexing is a means the fetching of a single value from the list. This is a 0-based indexing scheme. This is similar to strings from the previous lecture.

```
In[1]:L = [2, 3, 5, 7, 11]
In [2]: L[0]
Out [2]: 2
```




```
In [3]: L[1]
Out [3]: 3
```

```
In [4]: L[5]  # index out of bounds error.
```

Indexing

Negative index wraps around the end.

```
In[1]:L = [2, 3, 5, 7, 11]
In [2]: L[-1]
Out [2]: 11

In [3]: L[-2]
Out [3]: 7
```

Lists can contain different types of objects

List can contain different types and even other lists.

```
In [1]: L = [1, 'two', 3.14, [0, 3, 5]]
In [2]: L[0]
Out [2]: 3

In [3]: L[3]
Out [3]: [0, 3, 5]
```

Slicing

Similar to strings, lists also support slicing. *Slicing* is accessing multiple values from the list. It uses a colon to indicate the start point (inclusive) and end point (non-inclusive) of the subarray.

```
In[1]:  L = [1, 2, 3, 4, 5]
In [2]: L[0:3]
Out [2]: [1, 2, 3]
```

```
In [3]: L[:3]
Out [3]: [1, 2, 3]
```

Leaving out the first index defaults to 0.

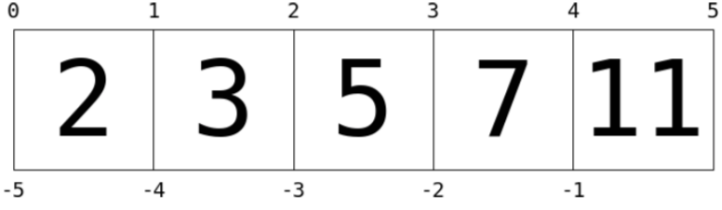
```
In [4]: L[-3:]
Out [4]: [3, 4, 5]
```

Leaving out the last index defaults to length of list.

Slicing

Finally, it is possible to specify a third integer that represents the step size.

```
In[1]:L = [2, 3, 5, 7, 11]
In [2]: L[::2] # equivalent to L[0:len(L):2]
Out [2]: [2, 5, 11]
```



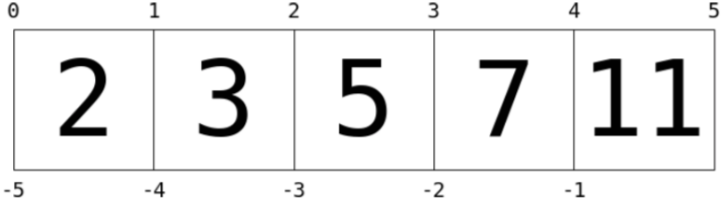
Slicing

A particularly useful version of this is to specify a negative step, which will reverse the list. In this case, the default start index is at the end of the list and the default stop index is at the beginning(inclusive).

```
In [1]: L = [1, 2, 3, 4, 5]
In [2]: L[::-1] # defaults end of array to beginning
Out [2]: [5, 4, 3, 2, 1]
In [3]: L[1::-1] # first two items reversed
Out [3]: [2, 1]
In [4]: L[:-3:-1] # last two items reversed
Out [4]: [5, 4]
In [5]: L[-3::-1] # everything except last two items reversed
Out [5]: [3, 2, 1]
```

Slicing

Both indexing and slicing can be used to set elements as well as access them.




```
In [1]: L[0] = 100
In [2]: print(L)
[100, 3, 5, 7, 11]
```

```
In[3]: L[1:3] = [99, 99]
In [4]: print(L)
[100, 99, 99, 7, 11]
```

Lists: Concatenation

List Concatenation: + operator will concatenate two lists.

```
In[1]:L = [2, 3, 5, 7, 11]
In [2]: L + [13, 17, 19]
Out [1]: [2, 3, 5, 7, 11, 13, 17, 19]
```

List Methods

The following is a short list of useful list methods.

append(value)	appends value to the end of the list
insert(index, value)	inserts value at position given by index, shifts elements to the right.
pop(index)	removes object at index from list, shifts elements left and returns removed object. Returns last element if index is omitted.
remove(value)	removes object from list by **value.**
index(value)	returns index of first occurrence of value if exists, otherwise throws a ValueError.
count(value)	returns the number times value appear in list.

List Methods

```
In [1]: L = [3, "hi", -4, 6]
In [2]: L.append(2) # [3,"hi",-4, 6, 2]
In [3]: L.insert(1, "hello") # [3,"hello","hi",-4,6,2]
In [4]: a = L.pop(3) # [3,"hello","hi",6,2]
In [5]: print(a)
-4

In [6]: L.pop() # [3,"hello","hi",6]
In [7]: L.remove("hi") # [3,"hello",6]
In [8]: ind = L.index(6) # 2
In [9]: L.count("hello") # 1
```

list()

Converting between a string and a list can be done using the list() constructor.

```
In [1]: message = “python”
        list(message)
Out [1]: [‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]
```

list()

A useful operation is to use list() with range() to create new lists easily.

```
In [1]: lst = list(range(1,7))
In [2]: lst
Out [2]: [1, 2, 3, 4, 5, 6]

In [3]: lst2 = list(range(1,1000))
```

List Comprehensions

List comprehension(or listcomp) is a concise and readable way of creating a new list from another iterable(list, tuple, string).

Suppose we want to square all numbers from a list.

```
In [1]: lst = [2, 3, 5, 7]
        new_lst = []
        for x in lst:
            new_lst.append(x ** 2)
        print(new_lst)
4 9 25 49
```

We can use list comprehension to do this much simpler and more Pythonic.

List Comprehensions

We can use list comprehension to do the previous problem much simpler and more Pythonic.

```
In[2]: lst = [2, 3, 5, 7]
       new_lst = [x**2 for x in lst]
       print(new_lst)
4 9 25 49
```

For loops are used to do many things: scan through a list, pick out items, compute sums, counts and averages, etc... Listcomp is a one trick pony: create new lists.

List Comprehensions

List comprehension can also contain conditions.

```
In[3]: lst = [x**2 for x in range(7) if x % 2 == 0]
       print(lst)
0 4 16 36
```

List Comprehensions

List comprehension can also contain nested loops. A list of lists or a list of tuples can be created in this way.

```
In[1]: colors = ["red", "black"]
In[2]: sizes = ["S, "M", "L"]
In[3]: shirts = [(color, size) for color in colors
                               for size in sizes]
In[4]: shirts
Out[4]:
[('red', 'S'),('red', 'M'),('red', 'L'),('black', 'S'),
 ('black', 'M'),('black', 'L')]
```

Iterables

An iterable is an object from which we can loop over. Lists and strings are iterables.

```
In[1]: lst = [2, 3, 5, 7]
       for x in lst:
           print(x, end=‘ ’)
2 3 5 7

In[2]: message = “python”
       for x in message:
           print(x, end=‘ ’)
p y t h o n
```

split()

split() method splits a string into a list. A separator can be specified. The default separator is any whitespace.

```
In [1]: fruits = "apple mango banana grape"
        fruits.split()
Out [1]: ['apple', 'mango', 'banana', 'grape']

In [2]: list = "hi, I am Mike, I just graduate."
        list.split(", ")
Out [2]: ['hi', 'I am Mike', 'I just graduate.']
```

split()

When using split() to split a string of integers into a list of integers, remember to use the int() constructor to convert the list elements into integers.

```
s = input("Enter a list of integers separated by spaces: ")
list_nums = [int(x) for x in s.split()]
print(list_nums) # list of integers instead of strings
```

Membership and Operations

Operator	Description
`a is b`	True if a and b are identical objects
`a is not b`	True if a and b are not identical objects
`a in b`	True if a is a member of b
`a not in b`	True if a is not a member of b

Identity Operations

The comparison operator == checks for *object equality*.

```
In [1]: a = [1,2,3]
        b = [1,2,3]
In[2]:  a == b
Out [2]: True
```

Identity Operations

The identity operators, is and is not, check for *object identity*.

```
In[3]: a is b
Out [3]: False

In[4]: a is not b
Out [4]: True

In [5]:   a = [1,2,3]
          b = a
          a is b
Out [5]: True
```

Membership Operations

Membership operators check for membership within compound objects like lists.

```
In [1]: 1 in [1, 2, 3]
Out [1]: True

In[2]: 2 not in [1,2,3]
Out [2]: False

In[3]: 'a' in "python"
Out [3]: False
```

Variables Are Pointers

```
In [1]: x = [1, 2, 3]
        y = x
In [2]: print(y)
[1, 2, 3]


In [3]: x.append(4) # append 4 to the list pointed to by x
        print(y)    # y's list is modified as well!
[1, 2, 3, 4]


In [4]: x = 'something else'
        print(y) # y is unchanged
[1, 2, 3, 4]
```

List items hold references

Items in lists and tuples hold references to objects they contain.

```
In [1]: x = [1, 2, [3, 4, 5]]
In [2]: y = x[2] # y and x[2] both references [3,4,5]
In [3]: y[1] = 44 # modifying y will modify x[2] also!
In [3]: y
[3, 44, 5]
In [5]: x
[1, 2, [3, 44, 5]]
```

Both x[2] and y reference(points to the address) to the same list object. Modifying one will affect the other!

References

1) Vanderplas, Jake, A Whirlwind Tour of Python, O’reilly Media.
2) Luciano, Ramalho, Fluent Python, O'reilly Media.