Sequence Types

- Lists

- A list is a **mutable** sequence of values of any type.

Lists are **mutable**, i.e., we can change their contents.

```
numbers[1] = 99
numbers      #-> [10, 99, 20, 40]
```

Play

We can even change a sublist.

```
numbers[2:3] = [98, 97]
numbers      #-> [10, 99, 98, 97, 40]
```

Lists have several methods to change their contents.

```
lst = [1, 2]
lst.append(3)        # appends 3 to end of lst → [1, 2, 3]
x = lst.pop()        # lst → [1, 2], x → 3
lst.extend([4, 5])   # lst → [1, 2, 4, 5]
lst.insert(1, 6)     # lst → [1, 6, 2, 4, 5]
x = lst.pop(0)       # lst → [6, 2, 4, 5], x → 1
```

- List Methods

List objects have several useful methods.

Play

`lst.append(item)`	Add item to the end
`lst.insert(pos, item)`	Insert item at the given position
`lst.extend(collection)`	Add all the items in the argument
`lst.pop()`	Remove last item
`lst.pop(pos)`	Remove item in given position
`lst.remove(item)`	Remove first occurence of given item (if any)
`lst.sort()`	Sort the items in the list
`lst.reverse()`	Reverse the order of items in the list
`lst.index(item)`	Position of first occurrence of given item
`lst.count(item)`	Number of occurrences of given item

Traversing

- The most common way to traverse the elements of a sequence is with a **`for`** loop.

```
for f in fruits:
    print(f)
```

banana
pear
orange

Play

- We may also traverse a sequence using the indexes.

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

- In this case, we may use a **`while`** loop instead.

```
i = 0
while i < len(numbers):
    print(i, fruits[i])
    i += 1
```

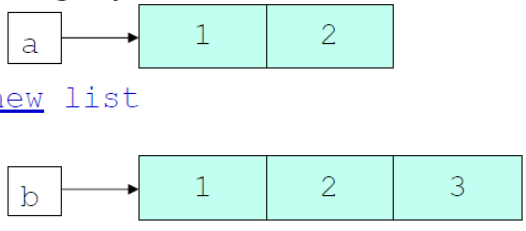
- Or traverse the indexes and items simultaneously.

```
for i, f in enumerate(fruits):
    print(i, f)
```

Cloning

- Sometimes, we need to make a copy of an object, so we can change it without changing the original.
- To clone lists, we may use the slicing operator **`[:]`**.

```
a = [1, 2]
b = a[:]    # slicing creates a new list
b is a      #-> False
b.append(3)
```



- We could also use the more general **`copy`** method.

```
b = a.copy() # clone a
b is a       #-> False
```

- Other mutable types (such as sets and dictionaries) also have a `copy` method.
- Immutable types (tuples, strings) don't need one.

- Strings

- Strings are **immutable**

String - traversal

- One way to traverse strings is with a `for` loop:

```
fruit = 'banana'
for char in fruit:
    print(char)
```

- Another way:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- Another example:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

Examples

- The following program counts the number of times the letter `'a'` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

- This prints the common characters in two strings. (For strings, the **`in`** operator returns `True` iff the first string appears as a substring in the second.)

```
for letter in word1:
    if letter in word2:
        print(letter)
```

String methods

- Strings have a lot of useful methods.

Play

`str.isalpha()`
`str.isdigit()`
`str.is...`
	True if all characters are alphabetic.
	True if all characters are digits.
	...

`str.upper()`
`str.lower()`
`...`
	Convert to uppercase.
	Convert to lowercase.
	...

`str.strip()`
`str.lstrip()`
`str.rstrip()`
	Remove leading and trailing whitespace.
	Remove leading whitespace.
	Remove trailing whitespace.

`str.split()`	Split str by the whitespace characters.
`str.split(sep)`	Split str using sep as the delimiter.
`sep.join(lst)`	Join the strings in lst using delimiter sep.

- Tuples

- Tuples are **immutable**

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Get your own Python Server

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Try it Yourself »

Add Items

Since tuples are immutable, they do not have a built-in `append()` method, but there are other ways to add items to a tuple.

1. **Convert into a list**: Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Try it Yourself »

2. **Add tuple to a tuple**. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Try it Yourself »

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Try it Yourself »

Or you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Try it Yourself »

Zipping and Enumerating

- The built-in function `zip` takes two or more sequences and generates a sequence of tuples, each containing one element from each sequence.

```
s = 'abc'
t = [4, 3, 2]
list(zip(s, t))      # → [('a', 4), ('b', 3), ('c', 2)]
```

- `enumerate` generates a sequence of (index, item) pairs.

```
enumerate('abc')     # → (0, 'a'), (1, 'b'), (2, 'c')
```

- You can use tuple assignment in a **`for`** loop to traverse a sequence of tuples:

```
s = 'somestuff'
for i, c in enumerate(s):
    print(i, c)
```

Play