Structured Types: Tuples, Ranges, Lists, Dictionaries

Object Types

Scalar objects: have no accessible internal structure

- Scalar types in Python: int, float

Structured objects:

- Data structures are *structures* which can hold some *data* together. They are used to store a collection of related data.
- The built-in data structures in Python 3.7 include: *str*, *list, tuple, range, and dictionary.*

Structured Types: str

Strings are structured because you can use indexing to extract individual characters from a string and slicing to extract substrings.

As discussed before, string objects have built-in functionality.

String objects are **immutable**, meaning that once the string is created, it cannot be modified.

Modifying a string always results in the creating of a new string.

Structured Types: tuples

Tuples are an ordered sequence of elements, which may contain a mix of element types

Tuples are **immutable**, you cannot change element values

Tuples are represented using parentheses ( )

Example with Tuples

```
#create an empty tuple
t1 = ()

#create a tuple containing 3 values
t2 = (1,"Two", 3)

#display the tuples
print(t1)
print(t2)

#display an element in a tuple
print( t2[1] )

#display the type of the element
print( type(t2[1]))

#tuples are immutable
#t2[0] = 5  -> TypeError: 'tuple' object does not support item assignment
```

Output:

```
()
(1, 'Two', 3)
Two
<class 'str'>
```

Structured Types: tuples

Like strings, tuples can be concatenated, indexed, sliced and repeated.

```
#concatenating tuples
t1 = ( 'a', 'b', 5)
t2 = (7,)
t1 = t1 + t2
print(t1)

#indexing with tuples
print(t1[2])

#slicing tuples
print(t1[1:3])

#repeating tuples
t3 = 2 * t1
print(t3)

#nesting tuples
t4 = ((1,'z'), 8, ('hi', 2, 'u'))
print(t4)
```

Output:

```
('a', 'b', 5, 7)
5
('b', 5)
('a', 'b', 5, 7, 'a', 'b', 5, 7)
((1, 'z'), 8, ('hi', 2, 'u'))
```

Examples with Tuples

```
te = ()
t = (2,"mit",3)
t[0]                    → evaluates to 2
(2,"mit",3) + (5,6)     → evaluates to (2,"mit",3,5,6)
t[1:2]     → slice tuple, evaluates to ("mit",)
t[1:3]     → slice tuple, evaluates to ("mit",3)
len(t)     → evaluates to 3
t[1] = 4   → gives error, can't modify object
```

empty tuple

extra comma means a tuple with one element

Example with Tuples

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print(t2)
print(t1 + t2)
print((t1+t2)[3])
print((t1+t2)[2:5])
```

Output:

```
((1, 'two', 3), 3.25)
(1, 'two', 3, (1, 'two', 3), 3.25)
(1, 'two', 3)
(3, (1, 'two', 3), 3.25)
```

TUPLES

- conveniently used to **swap** variable values

```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x // y
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(4,5)
```

integer division

Traversing a Tuple

Compute the sum of elements of a tuple.

Common pattern, iterate over tuple elements.

```
total = 0
for i in range(len(T)):
    total += T[i]
print total
```

```
total = 0
for i in T:
    total += i
print total
```

Note:

- Tuple elements are indexed `0` to `len(T)-1`
- range(`n`) goes from `0` to `n-1`

Exercises with Tuples and Function

Write a function to find the intersection of two tuples.

- `06_intersection.py`

Write a function to find the smallest common divisor greater than 1 and the largest common divisor of n1 and n2, assuming that n1 and n2 are positive integers. If no common divisor, return a tuple with two elements with the value (None, None).

- `06_divisors.py`

Homework: Write a function to find the union of two tuples.

- `06_union.py`

Structured Type – ranges

Like strings and tuples, ranges are **immutable**.

We can use the range() function to return a range of values, and the function takes 3 parameters (start, stop, step)

All operations on tuples can also be used with ranges, except for concatenation and repetition.

Range Examples

```
#create a range
r1 = range(1,11,2)
for i in r1:
    print(i, end=" ")
print()

#create a range - omit step
for j in range(2,12):
    print(j, end=" ")
print()

#create a range - omit start and step
for k in range(10):
    print(k,end=" ")
print()

#slicing/indexing ranges
print(range(10)[2:4][1])
```

Output:

```
1 3 5 7 9
2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
3
```

Comparing Ranges

The equality operator (==) can be used to compare range objects.

It returns True if the two ranges represent the same sequence of integers, and False if not.

When comparing ranges, the values and their order must be the same to be equal.

Example:

```
range(0, 7, 2) == range(0, 8, 2)
```
-> evaluates to true

```
range(0, 7, 2) == range(6, -1, -2)
```
-> evaluates to False

Structured Types: Lists

List are used to store a sequence of related values.

Lists are an ordered sequence of information, accessible by index.

A list is denoted by square brackets, [ ]

A list usually contains homogeneous elements.

List elements can be changed, so a list is **mutable.**

Lists:

```
#creating an empty list
a_list = []

#creating a list and initializing values
L = [2,8,3,6]

#displaying a list
print(L)

#display the number of elements in a list
print(len(L))

#indexing from zero - accessing an element
print(L[0])
print(L[2]+1)
print(L[3])
#print(L[4]) -> no element at index 4

#indexing with variables
i = 2
L[i-1]

#updating an element
L[1]= 12
print(L)
```

Output:

```
[2, 8, 3, 6]
4
2
4
6
[2, 12, 3, 6]
```

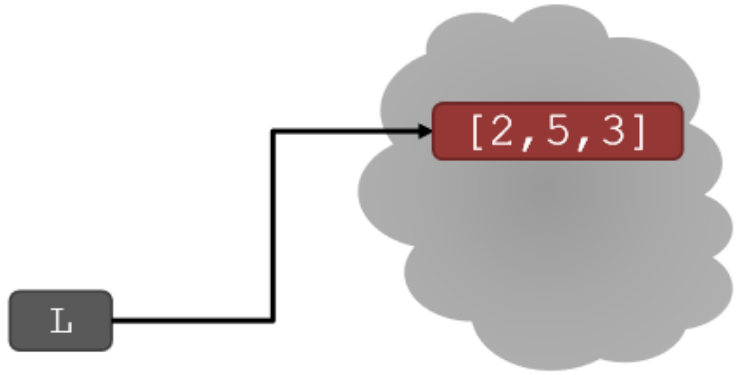
List and Mutability

Lists are mutable!

Assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

`L` is now `[2, 5, 3]`, note this is the same object



Traversing a List

Compute the sum of elements of a list.

Common pattern, iterate over list elements.

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

Note:

- list elements are indexed `0` to `len(L)-1`
- range(`n`) goes from `0` to `n-1`

List Operations - Appending

We can add elements to end of a list with the append function

```
L.append(e) -> adds the element,e, to the end of L
```

Mutates the list!

```
L = [2,1,3]
L.append(5) -> L is now [2,1,3,5]
```

List Operations - Inserting

We can insert elements at a specific position in the list

```
L.insert(i, e) -> inserts the element,e, into L at index i
```

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane']
friends.insert(1,'Cindy')
print(friends)
Output:
['Harry', 'Cindy', 'Emily', 'Bob', 'Jane']
```

List Operations – Finding an Element

We can find the position at which an element occurs.

```
L.index(e)
```

- returns the index of the first occurrence of e in L, gives a run-time error if e is not in `L`.

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane‘, ‘Emily’]
n = friends.index(‘Emily')
print(n)
n = friends.index(‘Emily‘, n+1)
print(n)
Output:
                1
                4
```

List Operations – Finding an Element

Because index will cause an error if the element does not exist in the list, it is usually a good idea to test with the `in` operator before calling the `index` function.

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane‘, ‘Emily’]
if ‘Emily’ in friends:
    n = friends.index(‘Emily‘)
else:
 n = None
```

List Operations – Finding an Element

Alternate solution: handling the exception.

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane‘, ‘Emily’]
try:
    n = friends.index(‘Emily‘)
except:
        n = None
```

List Operations - Removing

We can remove elements from a list by index:

```
L.pop(i)
-> removes and returns the item at index i in L,
-> if no index is specified it removes the last element
-> if i is not a valid index, a runtime error will occur
```

We can also remove an element by its value:

```
L.remove(e)
-> deletes the first occurrence of e from L.
-> if e does not exist in the list, a runtime ValueError will occur.
```

List Operations - Removing

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane']
friends.pop(1)
print(friends)
```

Output:

```
['Harry', 'Bob', 'Jane']
```

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane']
friends.pop()
print(friends)
```

Output:

```
['Harry', 'Emily', 'Bob']
```

Example:

```
friends = ['Harry', 'Emily', 'Bob', 'Jane']
friends.remove('Bob')
print(friends)
```

Output:

```
['Harry', 'Emily','Jane']
```

List Operations - Concatenation

The concatenation of two lists is a new list that contains the elements of the first list, followed by the elements of the second.

When we concatenate two lists, there are no side effects, meaning that a new list is created, and the original lists being concatenated are not mutated.

Example:

```
myFriends = ['Jane', 'Bob', 'Emily']
yourFriends = ['Cindy', 'John']
ourFriends = myFriends + yourFriends
```

Output:

```
['Jane', 'Bob', 'Emily', 'Cindy', 'John']
```

See: `06_friends.py`

Trace the below example:

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print('L3 =',L3)
L1.extend(L2)
print('L1 =',L1)
L1.append(L2)
print('L1 =',L1)
```

See: 06_extend_append.py

Trace the below example:

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print('L3 =',L3)
L1.extend(L2)
print('L1 =',L1)
L1.append(L2)
print('L1 =',L1)
```

Output:

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

See: 06_extend_append.py

List Operations – Equality Testing

The equality operator (==) can be used to compare whether two lists have the same elements, in the same order.

Example:

[1,4,9] == [1,4,9] is `True`, but

[1,4,9] == [4,1,9] is `False`

The opposite of == is !=.

Example:

[1,4,9] != [4,9] is `True`

List Operations – sum, max, min

You can use `sum, max, min,` functions whenever you want to find the sum, maximum element, minimum element of a list.

Example:

```
x=[1,16,9,4]
```

`sum(x)` will give 30

`max(x)` will give 16

`min(x)` will give 1

`x.sort()` will make x=[1,4,9,16]

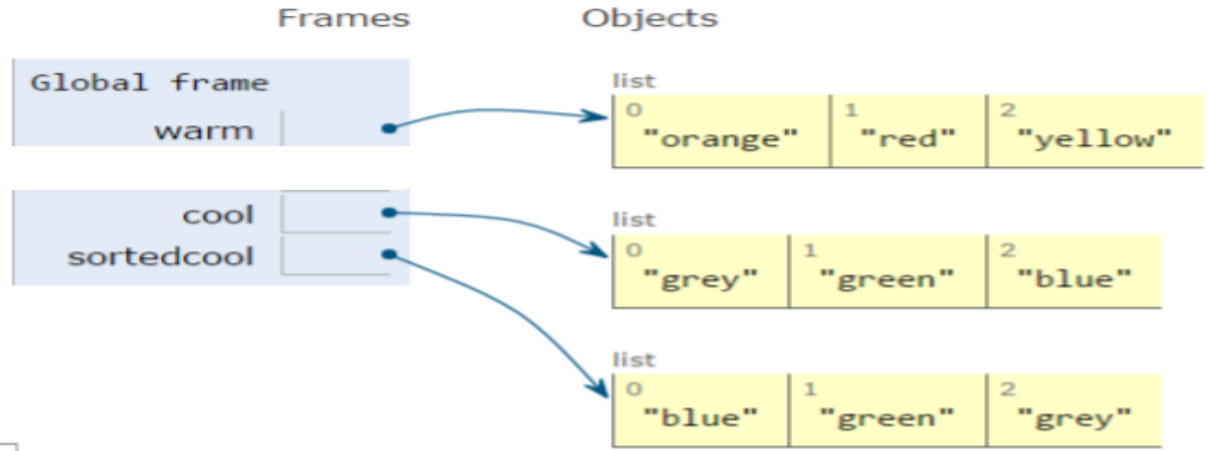
`x.sort(reverse=True)` will make x=[16,9,4,1]

List Operations – Sort

```
1  warm = ['red', 'yellow', 'orange']
2  warm.sort()
3  print(warm)
4
5
6  cool = ['grey', 'green', 'blue']
7  sortedcool = sorted(cool)
8  print(cool)
9  print(sortedcool)
```

```
['orange', 'red', 'yellow']

['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



Calling `sort()`: mutates the list, returns nothing

Two versions: `x.sort()` will make x=[1,4,9,16]

`x.sort(reverse=True)` will make x=[16,9,4,1]

Calling `sorted()`: does not mutate list, must assign result to a variable

List Function Summary

Function	Purpose
`len(L)`	Returns the number of items in `L`.
`L.append(e)`	Adds the object `e` to the end of `L`.
`L.count(e)`	Returns the number of times that `e` occurs in `L`.
`L.insert(i,e)`	Inserts the object `e` into L at index `i`.
`L.extend(L1)`	Adds the items in list `L1` to the end of `L`.
`L.remove(e)`	Deletes the first occurrence of `e` from `L`.
`L.index(e)`	Returns the index of the first occurrence of `e` in `L` and gives a runtime error if `e` is not in `L`.
`L.pop(i)`	Removes and returns the item at index `i` in L,and gives a runtime error if L is empty or the index is outside the bounds of the list. If `i` is omitted, it returns the last element (element at index -1)
`L.reverse()`	Reverses the order of the elements in L.

Exercises with Lists

1. Input values until the user enters -1, and store in a list. Then input a limit and display the index of the first element in the list that exceeds the limit and remove that element.
   - See: `06_listExercise1.py`
2. Input 5 words from the user, and store in a list.
   - Starting from the end of the list, display all Strings that begin with an uppercase letter.
   - Display the shortest word in the list.
   - See: `06_listExercise2.py`

Splitting Strings as Lists

`s.split(d)` – splits `s` using `d` as a delimiter. Returns a list of substrings of `s`. If `d` is omitted, the substrings are separated by arbitrary string of whitespace characters.

```
s = 'dog,cat,mouse,horse'
words = s.split(',')
print(words)
```

Output:

```
['dog','cat','mouse','horse']
```

```
words2 ='dog cat mouse horse'.split()
print(words2)
```

Output:

```
['dog','cat','mouse','horse']
```

Dictionaries

A dictionary is a container that stores associations between `keys` and `values`. Also known as a map, because it maps unique keys to values.

Every `key` is associated with a `value`.

`Keys` must be unique in a `dictionary`.

`Values` can be duplicated, but each `value` will be associated with a unique `key`.

Creating Dictionaries

Dictionary objects are created using curly braces { }

Syntax:

```
dict = {key1 : value1, key2 : value2, … keyN : valueN }
```

You can create an empty dictionary, using empty braces.

```
dict = {}
```

Example:

```
#create a dictionary
phone = { 'Evren':7445167, 'Ana':6413354,'Enes':6543210}
```

Accessing Dictionary Values

The subscript operator(`[]`) is used to return the `value` associated with a `key`.

Dictionary is not a sequence-type container like a list so although the subscript operator is used, you cannot access the items by index/position.

The given key must be in the dictionary, if it is not, a `KeyError` will be raised. Use `in/not in` to check if key values exist before accessing.

Syntax:

`dict[ key ]` -> returns the value associated with a given key.

Example:

```
phone = { 'Evren':7445167,
          'Ana':6413354,
          'Enes':6543210}
name = input('Enter name to search: ')

while name != 'quit':
    if name in phone:
        print(name,"'s contact number
is:",phone[name])
    else:
        print(name,' not in dictionary')
    name = input('Enter name to search: ')
```

Accessing Dictionary Values - Exceptions

A given key must be in the dictionary, if it is not, a `KeyError` will be raised.
You may use `in/not in` to check if key values exist before accessing.
An alternate solution is to use the Python exception handling mechanism.

Syntax:

```
try:
   #do this
except:
   #code to execute if statement in try block throws exception.
```

```
while name != 'quit':
    try:
        print(name,"'s contact number is:",phone[name])
    except:
        print(name,' not in dictionary')
    name = input('Enter name to search: ')
```

Adding/Updating Values

Dictionaries are mutable, you can change its contents after it has been created.

To change a value associated with a given key, set a new value using the `[]` operator on an existing key.

```
dict[key]= new_value
```

To add items to the dictionary, just specify the new value using a new unique key:

```
dict[new_key]= new_value
```

Example with Adding/Updating

```
name = input('Enter person to update: ')
number = input('Enter phone number: ')
if name in phone:
    phone[name] = number
    print(name, 'updated! (', phone[name],')')
else:
    phone[name] = number
    print(name, 'added! (', phone[name],')')
```

Removing Items – pop()

To remove a `key` / `value` pair from the dictionary, you can use the `pop`() function.

Syntax:

`dict.pop( key )` -> removes the `key/value` pair with the given `key`.

Example:

```
#remove keys from dictionary
name = input('Enter person to remove: ')
if name in phone:
phone.pop(name)
print(name, 'removed! ')
else:
print(name, 'not in phone book')
```

Traversing a Dictionary (Iteration)

The dictionary stores its items in an order that is optimized for efficiency, which may not be the order in which they were added.

You can iterate over the individual keys in a dictionary using a for loop.

Example:

```
#traversing a dictionary
print('Contact List: ')
for people in phone:
    print(people,phone[people])
```

Note: the above example is used to show iteration through the elements in a dictionary, but it can also be done without using a for loop.

Dictionary Function Summary

Function	Purpose
`len(d)`	Returns the number of items in `d`.
`d.keys()`	Returns a view of the `keys` in `d`.
`d.values()`	Returns a view of the `values` in `d`.
`k in d`	Returns true if `k` is in `d`.
`d[k]`	Returns the item in `d` with key `k`.
`d.get(k,v)`	Returns `d[k]` if `k` is in `d`, `v` otherwise.
`d[k] = v`	Associates the value `v` with the key `k` in `d`, if there is already a value associated with k, that value is replaced.
`d.pop(k)`	Removes the `key/value` pair with the given key, `k` in `d`.
`for k in d`	Iterates over the `keys` in `d`.

Dictionary Exercise

1. Write a program that uses a dictionary to store information about doctors and their patients.
2. Your program should use a function, read_doctors(), which does the following:
   - takes a file reference as a parameter and returns a dictionary containing doctor and patient information.
   - The keys in the dictionary are **tuples** containing the string doctor id and string name. The values in the dictionary are **lists** of **tuples** containing the id,name,telephone numbers of each patient.
   - Each line of the file contains the id/name of the doctor and the id/name/telephone number of the patient.
   - The doctors are not unique in the file, one doctor may have multiple patients. Each doctor should be added to the dictionary with their **list** of patients.
3. Your script should read the file data into a dictionary and do the following:
4. Input the name of a patient and list the information of the patient and their doctor. If there is more than one patient with the same name, display all.
5. See sample run on the next slide.

`See: 06_dictionary_exercise.py`

Sample Run:

```
Sevil Degirmenci : 265332
         717488 Selahattin Bardakci ( +90 242 023 4313 )
         600194 Muge Tiryaki ( +90 242 577 439 )
Mansur Binici : 379795
         965441 Rasim Karga ( +90 242 089 9441 )
Bunyamin Aksoy : 213286
         486976 Cemre Degirmenci ( +90 242 107 3041 )
         695664 Yeter Demirci ( +90 242 341 2984 )
Ceren Avci : 238200
         556805 Fidan Badem ( +90 242 116 4943 )
         916757 Belma Koc ( +90 242 147 3348 )
         128269 Mahmut Terzi ( +90 242 388 8937 )
Erkan Marangoz : 506263
         612160 Mahmut Terzi ( +90 242 132 4672 )
Nejla Koc : 442171
         445691 Sami Tiryaki ( +90 242 159 6412 )
         200030 Berrak Ekmekci ( +90 242 432 5372 )
Ezgi Uzun : 297021
         502803 Asli Kartal ( +90 242 204 0806 )
         181651 Ozturk Nacar ( +90 242 426 4289 )
Hasip Burakgazi : 730083
         151659 Munire Macar ( +90 242 216 8125 )
Tuncay Sadik : 925031
         202773 Necla Peynirci ( +90 242 276 0402 )

Enter patient name to search: Mahmut Terzi
128269 Mahmut Terzi +90 242 388 8937 Doctor: Ceren Avci
612160 Mahmut Terzi +90 242 132 4672 Doctor: Erkan Marangoz
```

Lists and Dictionaries in Memory

Lists and dictionaries are mutable

Mutable types behave differently than immutable types

Structured object are stored in memory and the variable name points to (references) the object

When an object is changed, any variable pointing to that object (referencing the object) is also affected

This is called ‘side effects’, meaning changing one variable may impact other variables.

Alias

Two or more references that refer to the same object are called *aliases* of each other

That creates an interesting situation: one object can be accessed using multiple reference variables

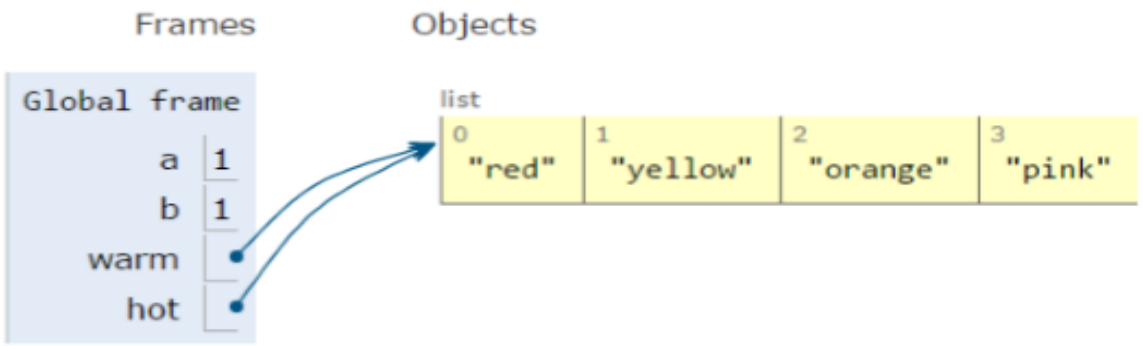
Aliases can be useful, but should be managed carefully

Changing an object through one reference changes it for all of its aliases, because there is really only one object

Alias Example

```
1  a = 1
2  b = a
3  print(a)
4  print(b)
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



In the example above, `hot` is an `alias` for warm – changing one changes the other!

Therefore the function append() has a side effect, applying it to `hot` impacts `warm`.

Because `a` and `b` are scalar values, there are no aliases/side effects.

Cloning

As you see the previous example, when you assign an object to another variable it creates an alias and not a copy.

Cloning an object involves create a new object by copy the data from an existing object, in this case a list.

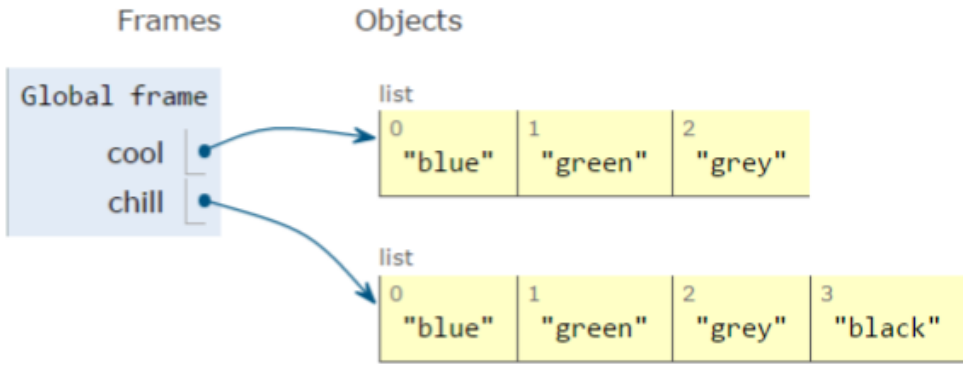
There are two ways to clone a list, either using a built-in function, list() or by using slicing.

Cloning Examples

Create a new list and copy every element using chill = cool[:]

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```



Create a new list using the list command:

```
chill = list(cool)
```

MUTATION AND ITERATION

- **avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

clone list first, note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2

Lists as Function Parameters – Trace the Following

```
from random import randrange
def generate_list(n):
    my_list = []
    for i in range(1,n+1):
        my_list.append(randrange(1,101))

    return my_list

def double_list(my_list):
    for i in range(len(my_list)):
        my_list[i] = my_list[i] * 2

def double_value(x):
    x = x / 2;
    print(x)

list_one = generate_list(5)
print(list_one)

double_list(list_one)
print(list_one)

double_value(list_one[0])
print(list_one)
```

See: 06_list_parameters.py

Common Operations on Sequence Types (str, tuple, list)

seq[i] returns the i^{th} element in the sequence.

len(seq) returns the length of the sequence.

seq1 + seq2 returns the concatenation of the two sequences.

n * seq returns a sequence that repeats seq n times.

seq[start:end] returns a slice of the sequence.

e in seq is True if e is contained in the sequence and False otherwise.

e not in seq is True if e is not in the sequence and False otherwise.

for e in seq iterates over the elements of the sequence.

Comparison of Sequence Types

Type	Type of elements	Examples of literals	Mutable
`str`	characters	`''`, `'a'`, `'abc'`	No
`tuple`	any type	`()`, `(3,)`, `('abc', 4)`	No
`list`	any type	`[]`, `[3]`, `['abc', 4]`	Yes

Higher Order Functions

A higher-order function is a function that does at least one of the following:

- takes one or more functions as arguments
- returns a function as its result.

Python supports functions as first class objects, meaning that we can use them like we use any other values (numbers, strings, lists)

We can pass functions as arguments to function calls, return function values as results from function calls, and embed function values in data structures.

Functions as Objects - Example

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
def fact(n):
    factorial = 1
    for i in range(1,n+1):
        factorial *= i
    return factorial

L = [1,-2,3.33]
print('L = ', L)
print('Apply abs to each element of L')
applyToEach(L, abs)
print('L = ', L)

print('Apply int to each element of L')
applyToEach(L, int)
print('L = ', L)

print('Apply factorial to each element of L')
applyToEach(L, fact)
print('L = ', L)
```

Output:

```
L =  [1, -2, 3.33]
Apply abs to each element of L
L =  [1, 2, 3.33]
Apply int to each element of L
L =  [1, 2, 3]
Apply factorial to each element of L
L =  [1, 2, 6]
```

See: 06_functions_as_objects.py

Tables – Lists of Lists

A table or a matrix is an arrangement consisting of rows and columns of values.

Sometimes it is necessary to store tables/matrices of data in our programs (for example scientific or financial applications).

Python does not have a data type for creating tables, but a two-dimensional tabular structure can be created using Python lists.

Creating Tables

Because tables are lists of lists, they can be created in the same way.

Either we can create a table by initializing its values, or by creating an empty list, and adding rows as needed.

Note that the statements shown below both produce the same table.

```
table = [[0,3,0],
         [0,0,1],
         [2,0,3],
         [3,1,0],
         [2,5,4]]
```

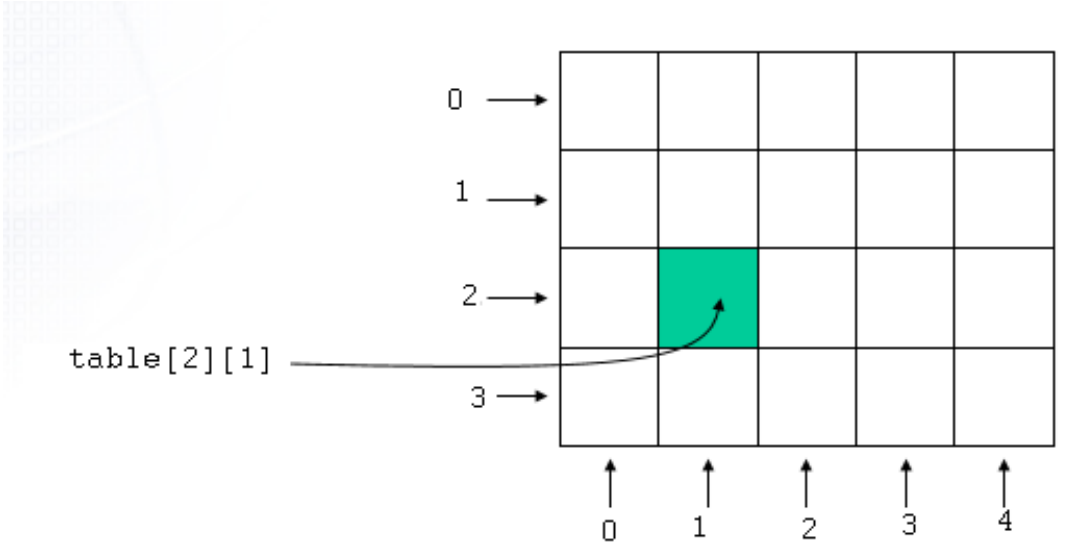
```
table = []
table.append([0,3,0])
table.append([0,0,1])
table.append([2,0,3])
table.append([3,1,0])
table.append([2,5,4])
```

Accessing an Element of a Table

To access a particular element in the table, you need to specify two index values in separate square brackets to select the row and column, respectively.

To calculate the number of rows in the table: len(table)

To calculate the length of a specific row in the table: len(table[row])



Accessing All Elements in a Table

To access all elements in a table, you use 2 nested loops.

You can either use the `range` function, or the `in` operator to access the elements.

```
for row in table:
    for col in row:
        print(col,end=" ")
    print()
```

```
for row in range(len(table)):
    for col in range(len(table[row])):
        print(table[row][col],end=" ")
    print()
```

Exercises:

Write a program that does the following:

- Input a table of values from the user (2x3)
- Using a method, `print_table()`, displays the table.
- Using a method `sum_rows()`, displays the sum of each row.
- Using a method `sum_cols()`, displays the sum of each column.
- `06_inputtable.py`

Write a function that takes a table of words and a string word as a parameter and replaces all occurrence of the word with an asterisk.

- `06_stringTable.py`