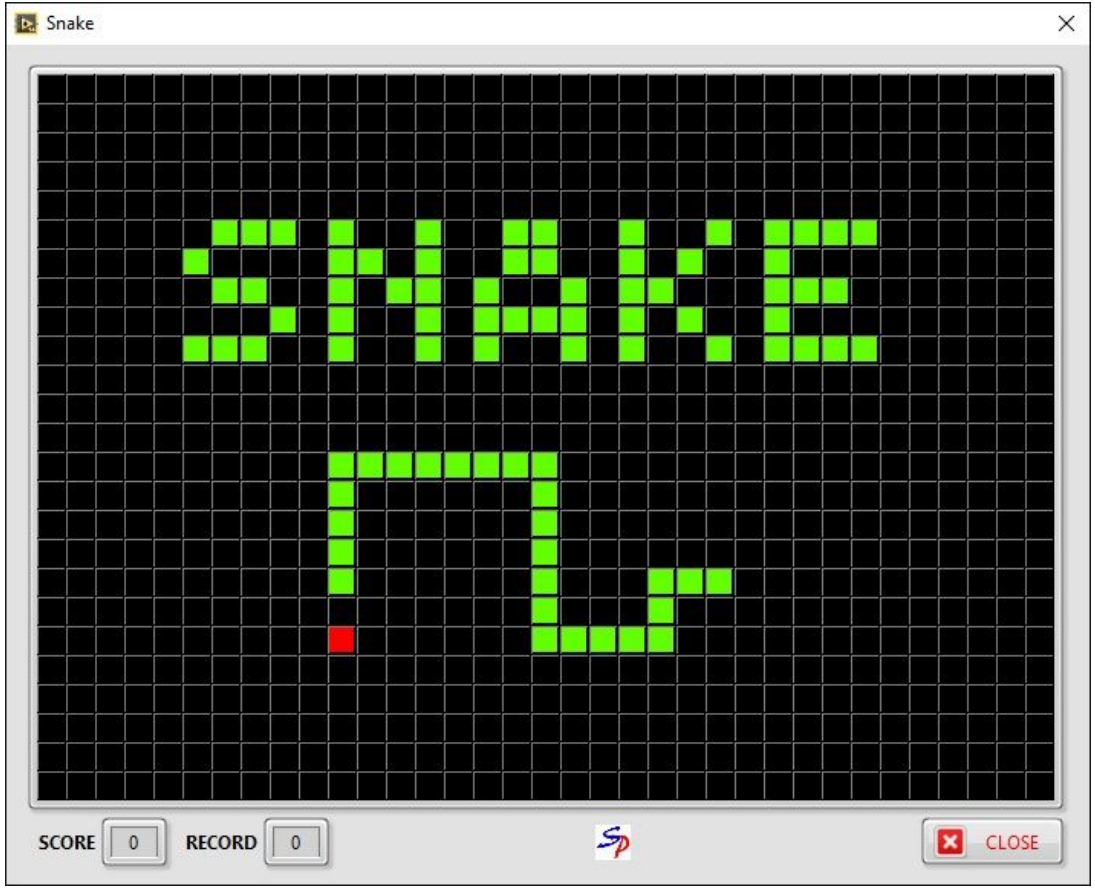
Programming Fundamentals

With Python

Chapter 5

01

Lists



Basic Structure

Index 0 , 1 , 2 , 3 , 4

```
nums = [6,9,0,7,8]
```

This is the name of the list

The **Index** is just a way for the compiler to quickly read the value for any position in the array you ask for.

NOTE: The indexing starts at 0, so largest index is always **# of elements** - 1 or **length of the array** - 1

Lists Properties

1. It is mutable, which means you can change the values in the list even after initialization.
2. It is ordered, which means whatever order you put the elements in, it will stay in that order forever.
3. You can have duplicate values in a list.

Access List Elements

```
1  nums = [6,9,0,7,8]
2
3  print(nums[0])
4  print(nums[3])
5  print(nums[5])
```

You have to tell the compiler what list you want to access and which index. This will give a single value and can be treated like a variable.

Output:

```
6
7
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 5, in <module>
IndexError: list index out of range
```

The last index is actually out of bounds.

Change List Elements

```
1  nums = [6,9,0,7,8]
2
3  print(f"nums list before: {nums}")
4
5  nums[2] = nums[1] + 1
6
7  print(f"nums list after: {nums}")
```

```
nums list before: [6, 9, 0, 7, 8]
nums list after: [6, 9, 10, 7, 8]
```

Add List Elements

append() function allows you to add an element to the end of the list.

```
1  alphabet = ["A","B","C","D"]
2
3  print(f"before: {alphabet}")
4
5  alphabet.append("E")
6
7  print(f"after: {alphabet}")
```

```
before: ['A', 'B', 'C', 'D']
after: ['A', 'B', 'C', 'D', 'E']
```

Add List Elements

insert() function allows you to add an element at the index you want.

```
1  alphabet = ["A","B","D"]
2
3  print(f"before: {alphabet}")
4
5  alphabet.insert(2,"C")
6
7  print(f"after: {alphabet}")
```

```
before: ['A', 'B', 'D']
after: ['A', 'B', 'C', 'D']
```

Delete List Elements

remove() function allows you to delete an element when you don't know which index it is but you know its value.

```
1  alphabet = ["A","B","E","C","D"]
2
3  print(f"before: {alphabet}")
4
5  alphabet.remove("E")
6
7  print(f"after: {alphabet}")
```

```
before: ['A', 'B', 'E', 'C', 'D']
after: ['A', 'B', 'C', 'D']
```

Delete List Elements

pop() function allows you to delete an element at a specific index. If you do not specify an index, it will delete the last element in the list by default.

```
1  alphabet = ["A","B","E","C","D"]
2
3  print(f"before: {alphabet}")
4
5  alphabet.pop(2)
6
7  print(f"after: {alphabet}")
```

```
before: ['A', 'B', 'E', 'C', 'D']
after: ['A', 'B', 'C', 'D']
```

List Comprehension

The format is ⇒ newlist = [**body of for loop** for loop **if statement**]. It allows you to create a list based on another list.

Classic Method

```
1  letters = ["A","B","C","D","E"]
2  vowels = []
3
4  for a in letters:
5      if a == 'A' or a == 'E' or a == 'I' or a == 'O' or a ==
           "U":
6          vowels.append(a)
7
8  print(vowels)
```

```
['A', 'E']
```

List Comprehension

The format is ⇒ newlist = [expression **for** element **in** iterable **if** condition == True]

Another Method

```
1  letters = ["A","B","C","D","E"]
2
3  vowels = [a for a in letters if a == "A" or a == "E" or a ==
       "I" or a == "O" or a == "U"]
4
5  print(vowels)
```

```
['A', 'E']
```

When to use lists?

- A list is a collection of elements. You can imagine 1 element = 1 variable.
- You don’t use lists just to store a bunch of random variables as one collection. The variables have to be related to each other.

When to use lists?

Good example of lists

Imagine you are creating the classic snake game! You have to keep track of every part of the body of a snake by storing its cell location in a grid so that when the user moves it, all parts of the body move accordingly.

1 body part could be represented as a variable storing its location 

Several body parts could be represented as a list of body parts storing the location of each body part 

Example
User clicks right arrow

Loop through the snake body parts and increment the location of each body part by 1 cell to the right.

Looping through Lists

```
1  nums = [6,9,0,7,8]
2
3  for i in range(len(nums)):
4      print(nums[i])
```

Output

```
6
9
0
7
8
```

Len() is a function that accepts an array as an argument and returns the length of that array.

Looping through Lists

```
1   nums = [6,9,0,7,8]
2
3 - for num in nums:
4       print(num)
```

Output
6
9
0
7
8

You use this method usually when you just want to read the values of the list without editing the array.

Looping through Lists Example

```
1  nums = [6,9,0,7,8]
2
3  for i in range(len(nums)):
4      nums[i] = nums[i + 1] + 5
5
6  print(f"The array now looks like {nums}")
```

Output:

Be careful! You should never leave a program accessing an index out of bounds of the list.

When i = 4 is reached, nums array is asked to access i = 5, which is out of bounds!

Looping through Lists Example

```
1  nums = [6,9,0,7,8]
2
3  for i in range(len(nums) - 1):
4      nums[i] = nums[i + 1] + 5
5
6  print(f"The array now looks like {nums}")
```

Output:

```
The array now looks like [14, 5, 12, 13, 8]
```

List Exercise 1

Your task is to find the largest number in an array.

Guidelines

- Create a list of numbers
- Print the largest number in the list

Example: Assume we have nums = [6,9,0,7,8], the output would be `The largest number in the list is 9`

List Exercise 1

```
1  nums = [6,9,0,7,8]
2
3  max_num = nums[0]
4
5  for num in nums:
6      if num > max_num:
7          max_num = num
8
9  print(f"The largest number in the list is {max_num}")
```

List Exercise 2

Your task is to create a todo list program. In this program, you will have a menu option where the user can chose to add a task, delete a task, or exit the program.

Note

When there are no tasks, make sure user has no option to delete a task and program tells user he has no tasks yet. But, when there are tasks, user can see all the tasks.

02

Strings

Access Certain Characters

```
1  greet = "Hello"
2
3  print(greet[3])
4  print(greet[1:3])
5  print(greet[1:])
6  print(greet[:3])
7  print(greet[-1])
8  print(greet[-2:])
```

Output

```
l
el
ello
Hel
o
lo
```

The index range is from start to everything but not including end

Find if a substring exists

```
1  word = "bulldog"
2
3  print("dog" in word)
```

```
Output

True
```

Modify the String

NOTE: You cannot just call an index in the string and give it another value like lists.

```
1  greet = "Hello"
2
3  print(greet.replace("l","p"))
```

```
Output

Heppo
```

Modify the String

You can split a string into 2 strings based on a certain character. This is especially useful for datasets that are split by commas or space.

```
1  person = "Joseph,21,white, red hair"
2  print(person.split(","))
```

Output

```
['Joseph', '21', 'white', ' red hair']
```

Concatenate Strings

```
1  name = "Joseph"
2  age = 21
3
4  print("Joseph is "+str(age)+" years old")
```

Output

```
Joseph is 21 years old
```

Strings Exercise

Check if a word is a palindrome. A word is a palindrome if you can read from the last letter to first letter of the word and it would still spell the same word. For example, kayak is a palindrome because if you read from last letter to first letter, it still spells kayak.

Guidelines

- Make a function called isPalindrome that returns False if it is not a palindrome and True if it is a palindrome.

Test cases

Test case 1

```
word = "stressed"
```

```
Output
False
```

Test case 2

```
word = "racecar"
```

```
Output
True
```

Strings Exercise

Method 1

```
def isPalindrome(word):
    for i in range(len(word)):
        if word[i] != word[len(word) - 1 - i]:
            return False
    return True

word = "stressed"

print(isPalindrome(word))
```

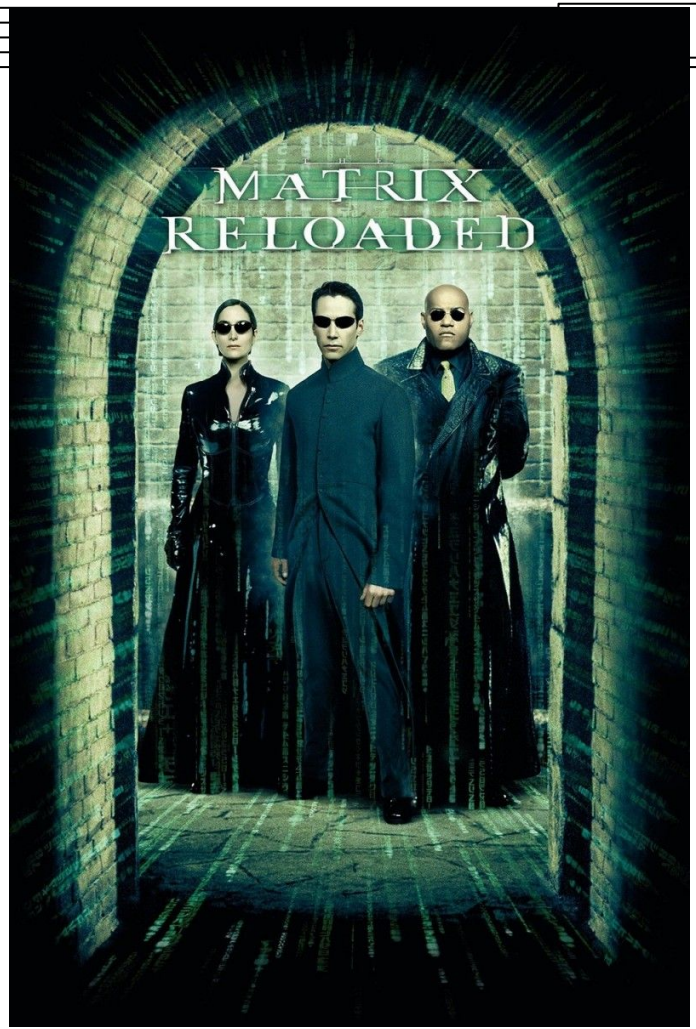
Strings Exercise

Method 2

```python
def isPalindrome(word):
    left = 0
    right = len(word) - 1

    while left <= right:
        if word[left] != word[right]:
            return False
        left += 1
        right -= 1
    return True

word = "racecar"

print(isPalindrome(word))
```

03

2D Lists



Basic Structure

```
1 nums = [
2       [1,2],
3       [3,4],
4       [5,6]
5       ]
```

Col 0 Col 1

Row 0

Row 1

Row 2

	0	1
0	1	2
1	3	4
2	5	6

Think of it like a list of lists. Each element can have different sizes of lists.

2D Lists Access Elements

```
1  nums = [
2       [1,2],
3       [3,4],
4       [5,6]
5       ]
6
7  nums[0][1] = nums[2][0]
8  nums[2][1] += nums[1][1]
9
10 print(nums)
```

Output

```
[[1, 5], [3, 4], [5, 10]]
```

	0	1
0	1	2
1	3	4
2	5	6

2D Lists Access Elements

```
1 nums = [
2          [1,2],
3          [3,4]
4       ]
5
6  print(nums[0])
```

You can access a whole row by specifying only one index. What you get is a 1D list.

2D Lists Add/Delete Elements

```
nums = [
      [1,2],
      [3,4],
      [5,6]
      ]

nums[1].append(9)
nums.append([3,2,1])
nums[3].remove(2)

print(nums)
```

Output

```
[[1, 2], [3, 4, 9], [5, 6], [3, 1]]
```

2D Lists - Print all the elements in the 2D List in a nice format

```
nums = [[1,2],[3,4],[5,6]]
```

```
1  nums = [[1,2],[3,4],[5,6]]
2
3  for i in range(len(nums)):
4      for j in range(len(nums[i])):
5          print(f"{nums[i][j]} ",end = '')
6      print('')
```

Output

```
1 2
3 4
5 6
```

2D Lists - Add all the elements in the 2D List

```
nums = [[1,2],[3,4],[5,6]]
```

```
1  nums = [[1,2],[3,4],[5,6]]
2  sum = 0
3
4  for i in range(len(nums)):
5      for j in range(len(nums[i])):
6          sum += nums[i][j]
7
8  print(sum)
```

Output

21

Exercise

```
nums = [
        [1,1,1],
        [0,3,0],
        [2,1,1]
        ]
```

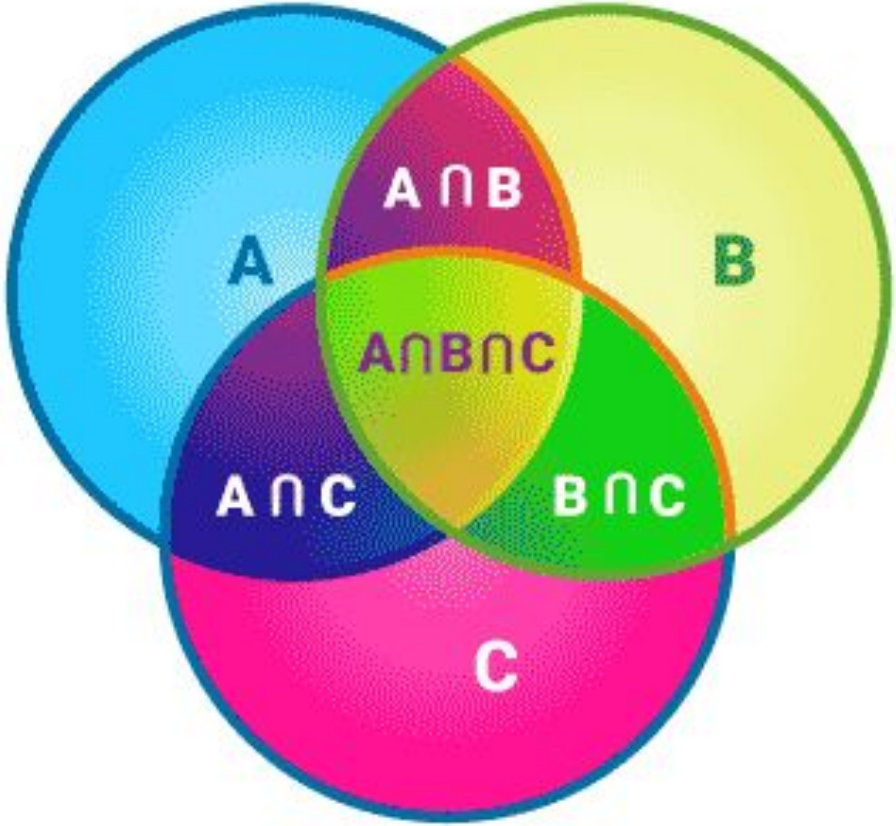
Find the sum of the diagonal and anti diagonal of this matrix.

Output

```
The sum of the diagonal is 5
The sum of the anti diagonal is 6
```

04

Sets



Sets Properties

1. It is unordered, which means whatever order you add elements in, it can change later.
2. You can add or remove elements, but you cannot change the value of an element.
3. You cannot have 2 elements with the same value.
4. You cannot read the value of a specific element by calling its index.

Sets Access Elements

You can iterate through all the elements in the set

```
1  color_set = {"Red","Blue","Yellow"}
2
3  for color in color_set:
4      print(color)
```

Output

```
Blue
Yellow
Red
```

You can check if an element exists in anywhere in the set

```
1  color_set = {"Red","Blue","Yellow"}
2
3  print("Yellow" in color_set)
```

Output

```
True
```

Sets Add Elements

You can add elements to the set (not in order).

```
1  color_set = {"Red","Blue","Yellow"}
2
3  color_set.add("Green")
4
5  print(color_set)
```

Output

```
{'Blue', 'Yellow', 'Green', 'Red'}
```

Sets Remove Elements

You can remove a certain element in the set.

```
1  color_set = {"Red","Blue","Yellow"}
2
3  color_set.remove("Blue")
4
5  print(color_set)
```

Output

```
{'Red', 'Yellow'}
```

Sets Join Methods

Union keeps all elements in both sets.

```
1  set1 = {1,2,3,4}
2  set2 = {1,3,5,8,9}
3
4  print(set1.union(set2))
```

```
Output

{1, 2, 3, 4, 5, 8, 9}
```

Intersection keeps elements that are shared between the 2 sets.

```
1  set1 = {1,2,3,4}
2  set2 = {1,3,5,8,9}
3
4  print(set1.intersection(set2))
```

```
Output

{1, 3}
```

Difference keeps all elements in the first set that are not shared with the second set.

```
1  set1 = {1,2,3,4}
2  set2 = {1,3,5,8,9}
3
4  print(set2.difference(set1))
```

```
Output

{8, 9, 5}
```

Sets Cannot Have Duplicate Values

```
1  list1 = [1,2,1,2,1]
2
3  print(set(list1))
```

Output
{1, 2}

05

Tuples

Tuples Properties

1. Tuple elements are in order
2. Tuples cannot be changed after initialization
3. Tuples can have duplicates

Tuples - Access Elements

You can access any element by its index like a list.

```
1  dark_purple = (48, 25, 52)
2
3  print(f"dark purple has {dark_purple[1]} green")
```

Output

```
dark purple has 25 green
```

Tuples - Unpack tuples

You can put every element in the tuple in a variable

```
1  dark_purple = (48, 25, 52)
2
3  red, green, blue = dark_purple
4
5  print(f"dark purple rgb value is ({red},{green},{blue})")
```

Output

```
dark purple rgb value is (48,25,52)
```

Tuples - Concatenate tuples

You can join tuples together

```
1  tuple1 = (1,2,3)
2  tuple2 = (4,5,6)
3  tuple3 = tuple1 + tuple2
4
5  print(tuple3)
```

Output

```
(1, 2, 3, 4, 5, 6)
```

You can duplicate a tuple

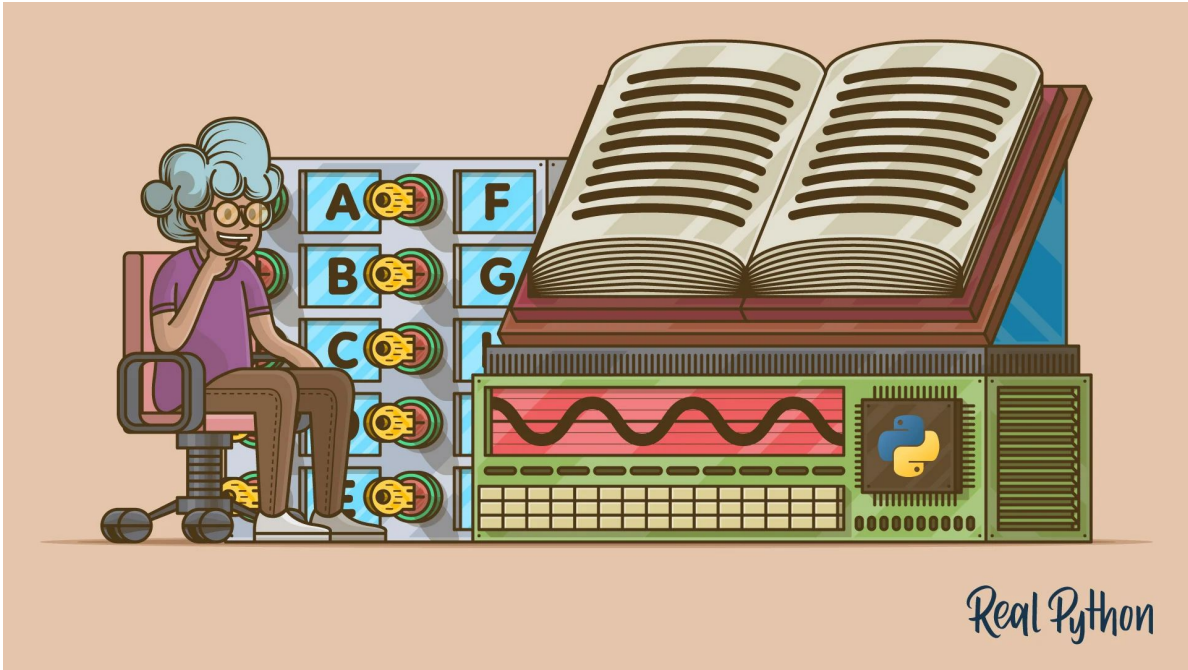
```
1  tuple1 = (1,2,3)
2  tuple2 = tuple1 * 3
3
4  print(tuple2)
```

Output

```
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

06

Dictionaries



Dictionaries Properties

1. You can change any key or value
2. You cannot have duplicate values
3. The key value pairs are in order

Dictionaries - Access Keys and Values

You can get the value of any key by calling the key.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

print(canadian_provinces["QC"])
```

Output

```
Quebec
```

Dictionaries - Access Keys and Values

You can get a list of all the keys in the dictionary.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

province_abbreviations = canadian_provinces.keys()

print(list(province_abbreviations))
```

Output

```
['AB', 'BC', 'QC', 'ON']
```

Dictionaries - Access Keys and Values

You can get a list of all the values in the dictionary.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

provinces = canadian_provinces.values()

print(list(provinces))
```

Output

```
['Alberta', 'British Columbia', 'Quebec', 'Ontario']
```

Dictionaries - Change Values

You can change the value of any key in the dictionary.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

canadian_provinces["ON"] = "ONTARIO"

print(canadian_provinces)
```

Output

Clear

```
{'AB': 'Alberta', 'BC': 'British Columbia', 'QC': 'Quebec', 'ON':
    'ONTARIO'}
```

Dictionaries - Add a Key,Value Pair

You can add a new key,value pair.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

canadian_provinces["MB"] = "Manitoba"

print(canadian_provinces)
```

Output

Clear

```
{'AB': 'Alberta', 'BC': 'British Columbia', 'QC': 'Quebec', 'ON':
    'Ontario', 'MB': 'Manitoba'}
```

Dictionaries - Delete a Key,Value Pair

You can delete a new key,value pair.

```
canadian_provinces = {
    "AB": "Alberta",
    "BC": "British Columbia",
    "QC": "Quebec",
    "ON": "Ontario",
}

canadian_provinces.pop("BC")

print(canadian_provinces)
```

Output

```
{'AB': 'Alberta', 'QC': 'Quebec', 'ON': 'Ontario'}
```

08

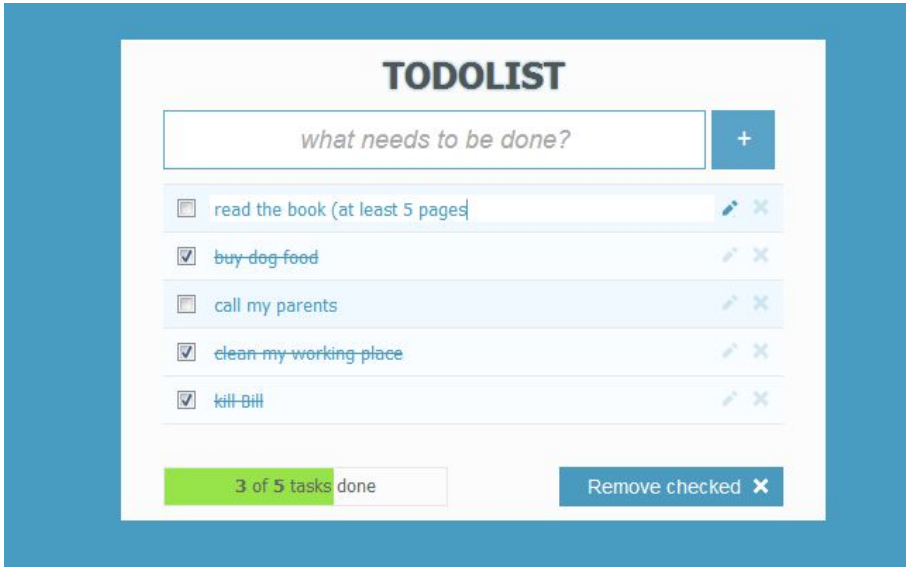
What Data Structure Do I use?

Lists vs. Tuples vs. Sets

	Mutable	Ordered	Indexing / Slicing	Duplicate Elements
List	✓	✓	✓	✓
Tuple	✗	✓	✓	✓
Set	✓	✗	✗	✗

Lists vs. Tuples

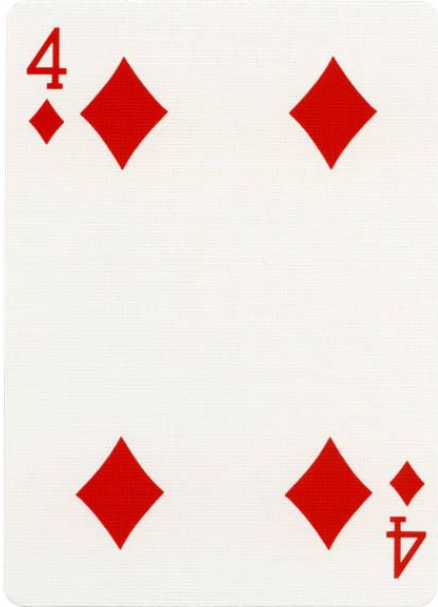
Let’s go back to the todo list program we made previously. You can add tasks and delete tasks. So, You are changing the collection. The best data structure is obviously a list.



Lists vs. Tuples

Now, let’s imagine we are creating a program that wants to represent the 52 deck of cards. So, we have a player with a red 4 of diamonds card.

```
1  card = (4, "Red", "Diamonds")
2
3  value, color, suit = card
4
5  print(f"The player has a {color} {value} of {suit}")
```



The best data structure is obviously a tuple. This is because this type of card will always have a value of 4, red color, and of type diamonds. This data will never change.

Lists vs. Tuples

In general

Lists
If you have a collection that represents data that has to be dynamic where you add or delete elements or you change the value of certain elements.

Tuples
If you have a collection that represents data of something that will never change. For example, a color, a card in a deck, or a record in a database.

Lists vs. Sets

Let’s go back again to the todo list program we made previously.

Both lists and sets allow you to add and delete elements, so it depends on the requirements of the program.

- Let’s say the user wants to edit the name of a certain task. You would have to locate that element and then change it. In sets, you can check if the task is there, but you cannot locate where exactly and you cannot change the value of that element.
- Let’s say the user wants the tasks to be in order. You can only have that with lists.
- Let’s say the program only allows users to delete tasks by specifying its index. You can only do that with lists.

Thus, a list is better for this program.

Lists vs. Sets

Let's see the difference between lists and sets when searching for an item.

List

```
1  nums = [1,11,5,3,44]
2  target = 3
3  exists = False
4
5  for i in nums:
6      if (target == i):
7          exists = True
8
9  print(exists)
```

OR

Time Complexity: **O(n)**

```
1  nums = [1,11,5,3,44]
2  target = 3
3  exists = False
4
5  if target in nums:
6      exists = True
7
8  print(exists)
```

Set

Time Complexity: **O(1)**

```
1  nums = {1,11,5,3,44}
2  target = 3
3  exists = False
4
5  if target in nums:
6      exists = True
7
8  print(exists)
```

It might see ridiculous that all I did was change [] to {}. But, the real difference is the time complexity because of the way lists and sets work behind the scenes.

Output

True

Lists vs. Sets

```python
import time

# A collection with numbers 0 to 1000000
nums_list = list(range(1000001))
nums_set = set(range(1000001))

# Let's make the number we are looking for the worst case scenario
target = 1000000

# Search in List
start_time = time.time()
found_in_list = target in nums_list
list_time = time.time() - start_time

# Search in Set
start_time = time.time()
found_in_set = target in nums_set
set_time = time.time() - start_time

print(f"{target} was found in the list after {list_time} seconds")
print(f"{target} was found in the set after {set_time} seconds")
print(f"The set was around {round(list_time/set_time)} times faster")
```

Output

Clear

```
1000000 was found in the list after 0.009879112243652344 seconds
1000000 was found in the set after 6.198883056640625e-06 seconds
The set was around 1594 times faster
```

The time will vary after every execution, but you can clearly see the big difference!

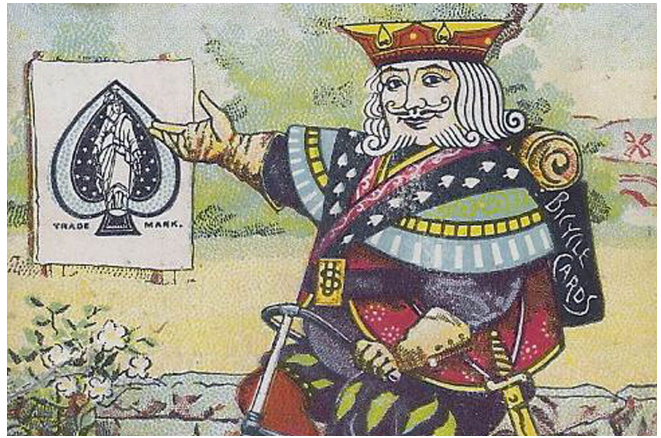
Lists vs. Sets

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

1. What cards do they have in common
2. What are all the cards they both collected from the deck
3. What cards does player 1 have that player 2 does not have.

NOTE: When I am talking about time complexity, assume

```
N = len(player1)
M = len(player2)
```



Lists vs. Sets

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

1. What cards do they have in common

List: Method 1

Time Complexity:
O(n*m)

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          ]
```

```
common_cards = []

for i in range(len(player1)):
    for j in range(len(player2)):
        if player1[i] == player2[j]:
            common_cards.append(player1[i])

print(common_cards)
```

Output

```
[('Red', 'Queen', 'Hearts'), ('Red', 10, 'Diamonds')]
```

Lists vs. Sets

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

1. What cards do they have in common

List: Method 2

Time Complexity: **O(n*m)**

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          ]
```

```
common_cards = [card for card in player1 if card in player2]

print(common_cards)
```

Output

```
[('Red', 'Queen', 'Hearts'), ('Red', 10, 'Diamonds')]
```

Lists vs. Sets

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

1. What cards do they have in common **Set**

Time Complexity:
O(min(n,m))

```
player1 = {
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          }

player2 = {
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          }
```

```
common_cards = player1.intersection(player2)

print(common_cards)
```

Output

```
{('Red', 'Queen', 'Hearts'), ('Red', 10, 'Diamonds')}
```

Lists vs. Sets

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

2. What are all the cards they both collected from the deck List: Method 1

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          ]
```

```
all_cards = []

for i in player1:
    all_cards.append(i)

for i in player2:
    if i not in player1:
        all_cards.append(i)

print(all_cards)
```

Time Complexity:
O(n*m)

Output Clear

```
[('Black', 'Ace', 'Spades'), ('Red', 'Queen', 'Hearts'), ('Red', 10,
    'Diamonds'), ('Red', 3, 'Diamonds'), ('Black', 5, 'Spades'),
    ('Black', 'King', 'Clubs')]
```

Lists vs. Sets

Time Complexity:
O(n*m)

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

2. What are all the cards they both collected from the deck <u>List: Method 2</u>

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
           ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
           ]
```

```
all_cards = player1 + [card for card in player2 if card not
    in player1]

print(all_cards)
```

Output Clear

```
[('Black', 'Ace', 'Spades'), ('Red', 'Queen', 'Hearts'), ('Red', 10,
    'Diamonds'), ('Red', 3, 'Diamonds'), ('Black', 5, 'Spades'),
    ('Black', 'King', 'Clubs')]
```

Lists vs. Sets

Time Complexity:
O(n + m)

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

2. What are all the cards they both collected from the deck Set

```
player1 = {
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          }

player2 = {
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          }
```

```
all_cards = player1.union(player2)

print(all_cards)
```

Output Clear

```
{('Red', 'Queen', 'Hearts'), ('Black', 5, 'Spades'), ('Red', 10,
    'Diamonds'), ('Black', 'King', 'Clubs'), ('Black', 'Ace',
    'Spades'), ('Red', 3, 'Diamonds')}
```

Lists vs. Sets

Time Complexity:
O(n*m)

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

3. What cards does player 1 have that player 2 does not have. List: Method 1

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          ]
```

```
player1_unique_cards = []

for i in player1:
    if i not in player2:
        player1_unique_cards.append(i)

print(player1_unique_cards)
```

Output

```
[('Black', 'Ace', 'Spades'), ('Red', 3, 'Diamonds')]
```

Lists vs. Sets

Time Complexity:
O(n*m)

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

3. What cards does player 1 have that player 2 does not have. List: Method 2

```
player1 = [
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          ]

player2 = [
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          ]
```

```
player1_unique_cards = [card for card in player1 if card not
    in player2]

print(player1_unique_cards)
```

Output

```
[('Black', 'Ace', 'Spades'), ('Red', 3, 'Diamonds')]
```

Lists vs. Sets

Time Complexity:
O(n)

Now, let's say we want to create a program that stores the cards that 2 players have collected from the deck of 52 cards. Then, we want to see:

3. What cards does player 1 have that player 2 does not have. Set

```
player1 = {
            ("Black","Ace","Spades"),
            ("Red","Queen","Hearts"),
            ("Red",10,"Diamonds"),
            ("Red", 3, "Diamonds")
          }

player2 = {
            ("Red",10,"Diamonds"),
            ("Black",5,"Spades"),
            ("Red","Queen","Hearts"),
            ("Black","King","Clubs"),
          }
```

```
player1_unique_cards = player1.difference(player2)

print(player1_unique_cards)
```

Output

```
{('Black', 'Ace', 'Spades'), ('Red', 3, 'Diamonds')}
```

Lists vs. Sets

In summary, when you want to search for a particular item from a collection **or** you want to join two collections, use sets because:

1. The code is much simpler
2. It is much faster to perform than using lists

Sets vs. Dictionaries

Similarities

- Both are very fast when searching for something (O(1) time complexity)
- You cannot have duplicate values

Differences

- In a dictionary, when you search for a key, you get a value mapped to it, and you can change its value

Sets vs. Dictionaries

Let’s say we wanna create a simple scrabble points game where the program reads the word player 1 and player 2 chose, give them a score, and determine who won.

- I want to search for each letter in the word quickly. So, I ruled out lists!
- When I search for the letter, I want to know how many points it is valued at. I can only map one value to another with dictionaries.

Thus, dictionaries is the clear winner!

Sets vs. Dictionaries

Let's say we wanna create a simple scrabble points game where the program reads the word player 1 and player 2 chose, give them a score, and determine who won.

```
scrabble_points = {
    'a': 1, 'b': 3, 'c': 3, 'd': 2,
    'e': 1, 'f': 4, 'g': 2, 'h': 4,
    'i': 1, 'j': 8, 'k': 5, 'l': 1,
    'm': 3, 'n': 1, 'o': 1, 'p': 3,
    'q': 10, 'r': 1, 's': 1, 't': 1,
    'u': 1, 'v': 4, 'w': 4, 'x': 8,
    'y': 4, 'z': 10
}
```

Now, I want you to implement this game!!

Example:

```
player1_word = "zebra"
player2_word = "apples"
player1_score = 0
player2_score = 0
```

Output

```
Player 1 score = 16, Player 2 score = 10
Player 1 won!
```

09

In Depth Understanding

References

Whenever a data structure is created, python knows where it lives in memory!

Address	Value
4000	5
4001	9
4002	8
4003	2

Let’s make it simple and say this is how memory looks like. Then, if I ask you what lives at address 4002, you should say 8.

References with Lists

NOTE: This is simplifying it and not exactly how it works!

Start address of the List

```
1  nums = [6,9,0,7,8]
2
3  print(nums[3])
```

Address	Value
4000	6
4004	9
4008	0
4012	7
4016	8
4020	Random value

NOTE: Each number takes 4 addresses.

Let's say compiler reads **nums[3]**. So, compiler will do this calculation:
list_start_address (4000) + index(3) * bytes_required_for_the_type (4)

NOTE: In a list, all elements are right below each other

References with Dictionaries

NOTE: This is simplifying it and not exactly how it works!

```
nums = {
      5 : 20,
      9 : 7,
      20 : 18,
      12 : 5,
      21 : 2,
      11 : 40
}
```

Each Address would be a list

Notice that the addresses are not right under each other!

This is a list stored somewhere in memory

Keys
Key0 = 5
Key1 = 9
Key2 = 20
Key3 = 12
Key4 = 21
Key5 = 11

Hash Function

Slot # = Key % 6

References

Slot #	Address	Value
0	4005	[5]
1	4020	[]
2	3900	[18]
3	6070	[7,2]
4	1002	[]
5	2026	[20,40]

References with Dictionaries and Sets

- Sets are managed in a similar way with the difference that the keys are just numbers. So, first element of the set would have key = 0, second element would have key = 1, and so on.
- Now you could see why searching for an element in dictionaries and sets are quick!! Because you can just pass any key into a hash function, which just completes a math formula and finds the location in **O(1)** time.
- You might ask how about the cases where one slot has 2 values? In this case it is **O(n)**, but the chance of all the keys being in the same slot are very low. So, let's say you have a huge dictionary with 1000 keys, you would only end up with like 1-4 values in one slot.

The Copy() Method

```
1  nums1 = [1,2,3]
2
3  nums2 = nums1
4  nums2[1] = 5
5
6  print(nums1)
7  print(nums2)
```

Output

```
[1, 5, 3]
[1, 5, 3]
```

In line 3, you told python let the start address of nums2 be equal to the start address of nums1. So, both lists reference the same collection in memory!

In the case of a dictionary or set, you told python let the start address of the **slot#** list be equal for both dictionaries or both sets.

The Copy() Method

To fix this, you have to use the copy() method in line 3. What this method does is it creates a new list and copies all the values in first list into the new list.

```
1  nums1 = [1,2,3]
2
3  nums2 = nums1.copy()
4  nums2[1] = 5
5
6  print(nums1)
7  print(nums2)
```

Output

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

The Copy() Method

The same goes for all other data structures

Set

```
1  set1 = {1,2,3}
2
3  set2 = set1
4  set2.add(4)
5
6  print(set1)
7  print(set2)
```

Output

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

```
1  set1 = {1,2,3}
2
3  set2 = set1.copy()
4  set2.add(4)
5
6  print(set1)
7  print(set2)
```

Output

```
{1, 2, 3}
{1, 2, 3, 4}
```

Dictionary

```
1  dict1 = {"A": 1, "B": 2}
2
3  dict2 = dict1
4  dict2["B"] = 3
5
6  print(dict1)
7  print(dict2)
```

Output

```
{'A': 1, 'B': 3}
{'A': 1, 'B': 3}
```

```
1  dict1 = {"A": 1, "B": 2}
2
3  dict2 = dict1.copy()
4  dict2["B"] = 3
5
6  print(dict1)
7  print(dict2)
```

Output

```
{'A': 1, 'B': 2}
{'A': 1, 'B': 3}
```

The deepCopy() Method

The copy() method is not gonna work with a 2D list. Instead, you have to use the deepcopy() method

```
nums1 = [
            [1,2],
            [3,4]
        ]

nums2 = nums1
nums2[0][1] = 10

print(nums1)
print(nums2)
```

Output

```
[[1, 10], [3, 4]]
[[1, 10], [3, 4]]
```

```
import copy
nums1 = [
            [1,2],
            [3,4]
        ]

nums2 = copy.deepcopy(nums1)
nums2[0][1] = 10

print(nums1)
print(nums2)
```

Output

```
[[1, 2], [3, 4]]
[[1, 10], [3, 4]]
```

Referencing a Data Structure from a function

```
1 def change_nums(nums):
2     for i in range(len(nums)):
3         nums[i] += 1
4
5 nums_list = [6,9,0,7,8]
6
7 change_nums(nums_list)
8
9 print(nums_list)
```

Output

```
[7, 10, 1, 8, 9]
```

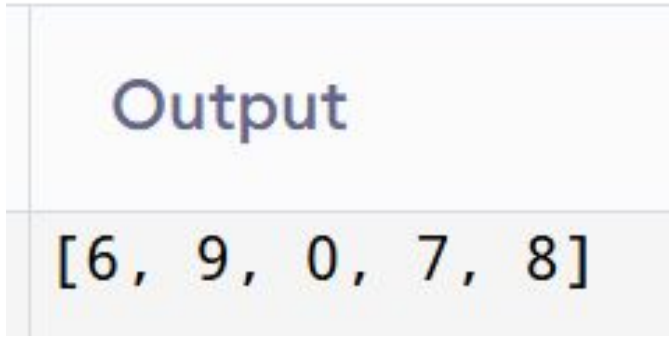
What you did here is you referenced the **nums** as a parameter for change_nums function. So, whatever you do to **nums** in the function will also change the **nums_list** that is outside of the function.

This is like saying **nums = nums_list**

NOTE: This is the same for all other data structures!

Referencing a Data Structure from a function

```
1 def change_nums(nums):
2     for i in range(len(nums)):
3         nums[i] += 1
4
5 nums_list = [6,9,0,7,8]
6
7 change_nums(nums_list.copy())
8
9 print(nums_list)
```



This time you created a new list. So, nums_list did not change.

NOTE: This is the same for all other data structures!

Returning multiple values from a function

```
1 def find_location():
2     return (500,600)
3
4 x,y = find_location()
5
6 print(x,y)
```

Output
500 600

When you are returning a value from a function, it will never change again because the function is over. So, the best data structure to use to return more than one value is a tuple.