MBSI Python Coding Workshop 5

Data Structures & For Loops

Revision Questions!

Which of the following is a correct way to import the random module and call the function that generates a random float between 0 and 1?

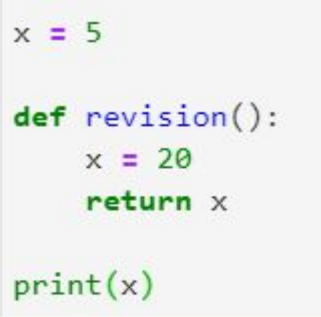
a) import random
print(random())

b) from Random import random
print(random)

[✓ c) import random as rd
print(rd.random())]

d) from random import *
print(randint(0, 1))

What will be the output for the following code?



```
x = 5

def revision():
    x = 20
    return x

print(x)
```

a) 20

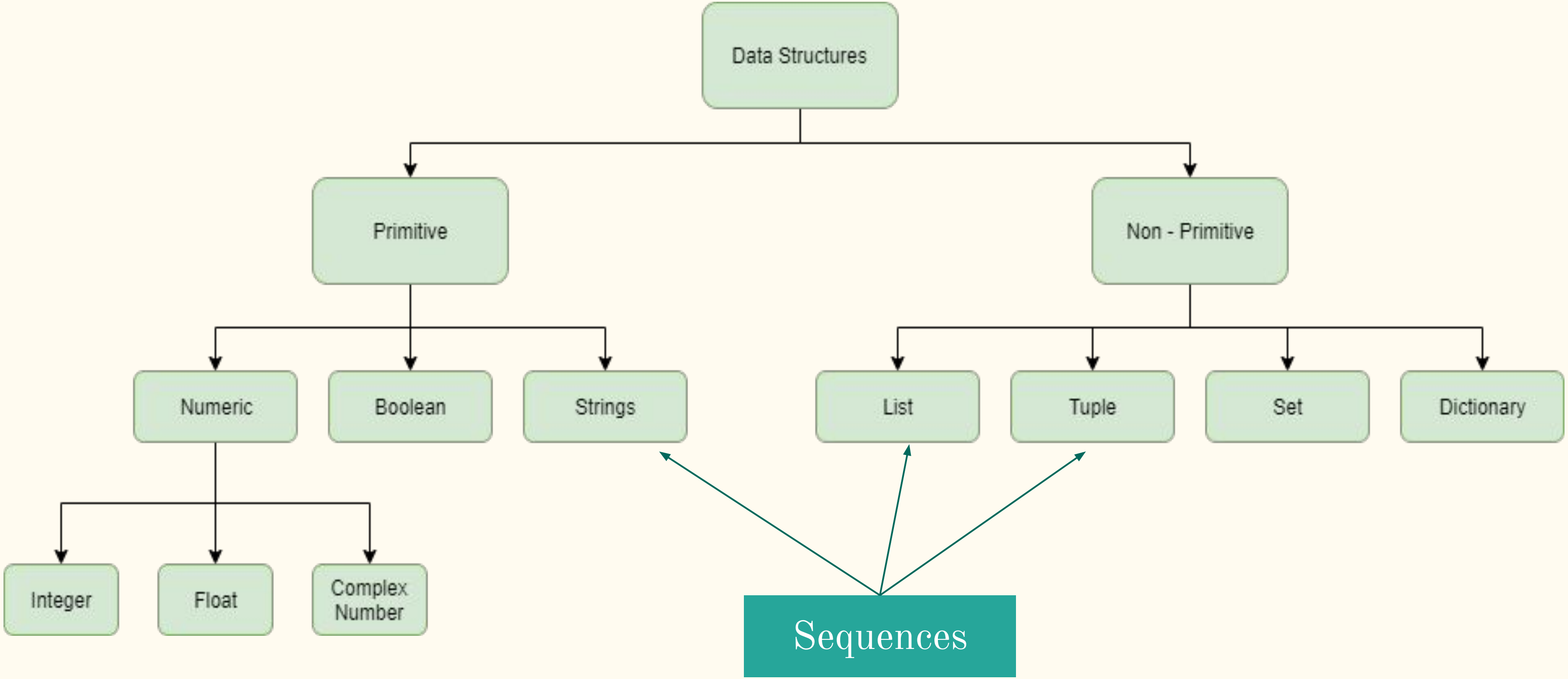
[✓ b) 5]

c) There will be an error

d) x

5.1 Data Structures

Data Structures

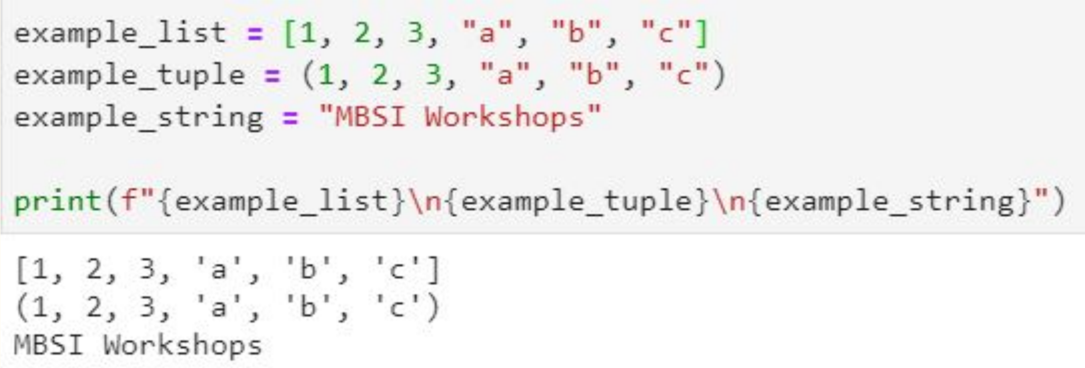


Sequences

- Lists are the most versatile sequence type. The elements of a list can be any object, and lists are **mutable** - meaning they can be changed.

- Tuples are like lists, but they are **immutable** - they can’t be changed.

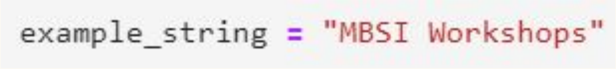
- Strings are a special type of sequence that can only store characters.



```
example_list = [1, 2, 3, "a", "b", "c"]
example_tuple = (1, 2, 3, "a", "b", "c")
example_string = "MBSI Workshops"

print(f"{example_list}\n{example_tuple}\n{example_string}")
```

```
[1, 2, 3, 'a', 'b', 'c']
(1, 2, 3, 'a', 'b', 'c')
MBSI Workshops
```

String Sequences



```
example_string = "MBSI Workshops"
```

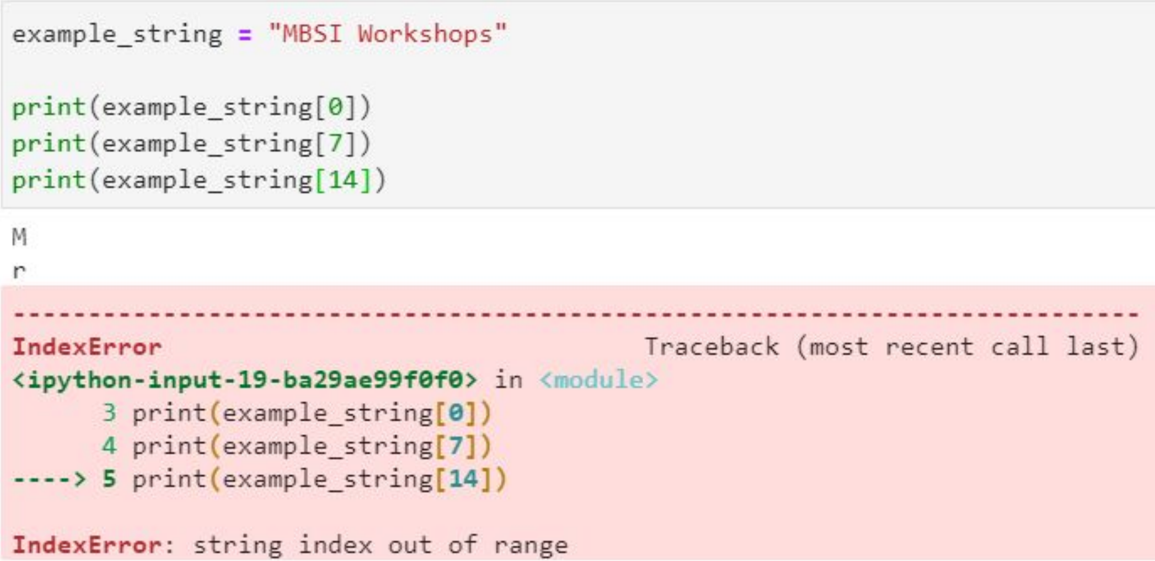
Character	M	B	S	I		W	o	r	k	s	h	o	p	s
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Indexing (from the left)

Character	M	B	S	I		W	o	r	k	s	h	o	p	s
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

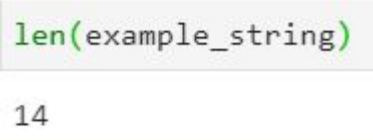
length - 1

- To access a specific element: **sequence[index]**



```
example_string = "MBSI Workshops"

print(example_string[0])
print(example_string[7])
print(example_string[14])
```

```
M
r
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-19-ba29ae99f0f0> in <module>
      3 print(example_string[0])
      4 print(example_string[7])
----> 5 print(example_string[14])

IndexError: string index out of range
```

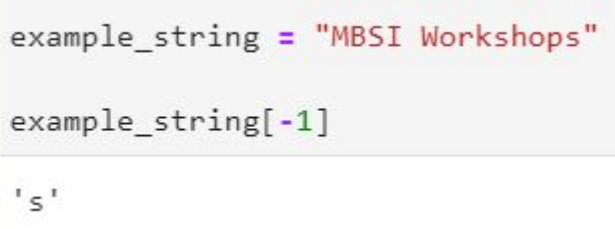


```
len(example_string)
```

```
14
```

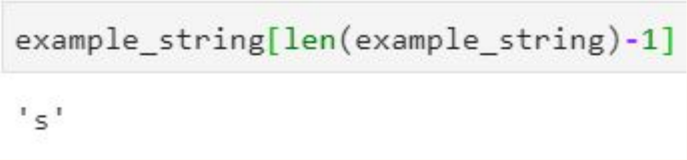
Indexing (from the right)

Character	M	B	S	I		W	o	r	k	s	h	o	p	s
Index	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



```
example_string = "MBSI Workshops"

example_string[-1]
```

```
's'
```

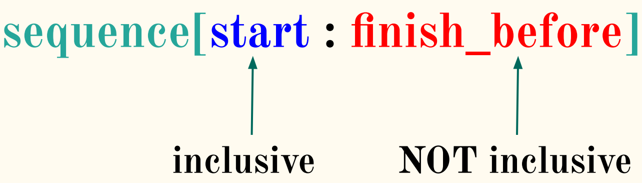


```
example_string[len(example_string)-1]
```

```
's'
```

Slicing

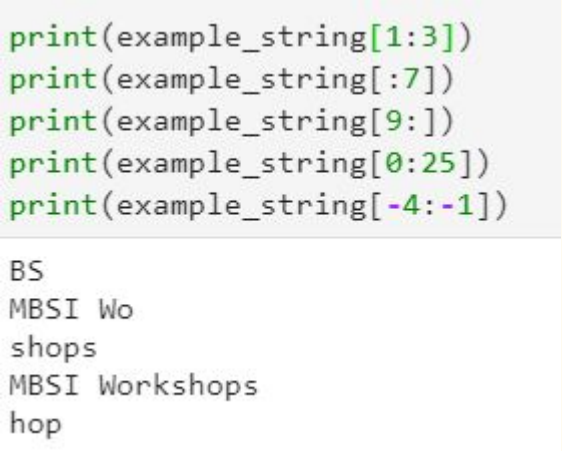
- Similar to indexing, except now we want to access multiple elements, or sub-sequences.

- 
sequence[start : finish_before]

inclusive

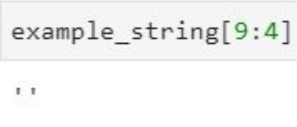
NOT inclusive

Char	M	B	S	I		W	o	r	k	s	h	o	p	s
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13



```
print(example_string[1:3])
print(example_string[:7])
print(example_string[9:])
print(example_string[0:25])
print(example_string[-4:-1])
```

```
BS
MBSI Wo
shops
MBSI Workshops
hop
```



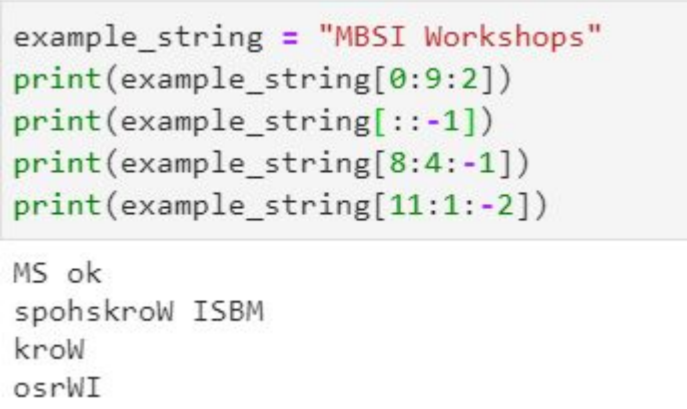
```
example_string[9:4]
```

```
''
```

Slicing with Steps & Direction

- sequence[start : finish_before : step_size and direction]

Char	M	B	S	I		W	o	r	k	s	h	o	p	s
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13



```
example_string = "MBSI Workshops"
print(example_string[0:9:2])
print(example_string[::-1])
print(example_string[8:4:-1])
print(example_string[11:1:-2])
```

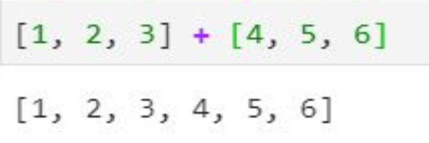
```
MS ok
spohskroW ISBM
kroW
osrWI
```

Sequence Operations

- Indexing and slicing.

- + combines two sequences in a process called **concatenation**.

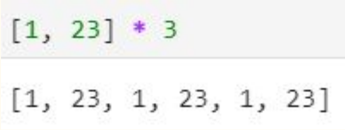
  For example:



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

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

- * repeats a sequence a number of times. For example:

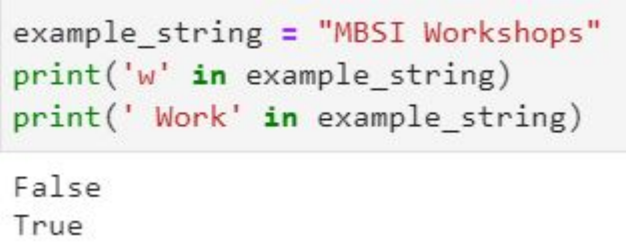


```
[1, 23] * 3
```

```
[1, 23, 1, 23, 1, 23]
```

- x **in** my_seq will return True if x is an element of my_seq, and False otherwise.

  For example:

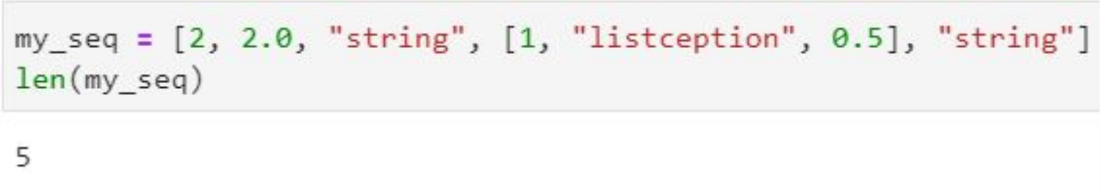


```
example_string = "MBSI Workshops"
print('w' in example_string)
print(' Work' in example_string)
```

```
False
True
```

Some Useful Sequence Functions

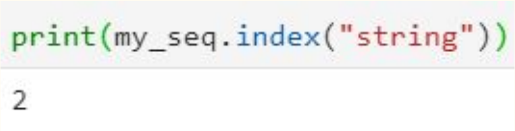
- **len(my_seq)** returns the number of elements in my_seq.



```
my_seq = [2, 2.0, "string", [1, "listception", 0.5], "string"]
len(my_seq)
```

```
5
```

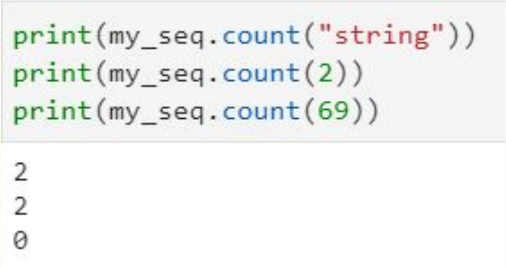
- **my_seq.index(x)** returns the index of the **first occurrence** of x in my_seq. Note that if x isn't in my_seq, an error will be returned.



```
print(my_seq.index("string"))
```

```
2
```

- **my_seq.count(x)** returns the number of occurrences of x in my_seq.



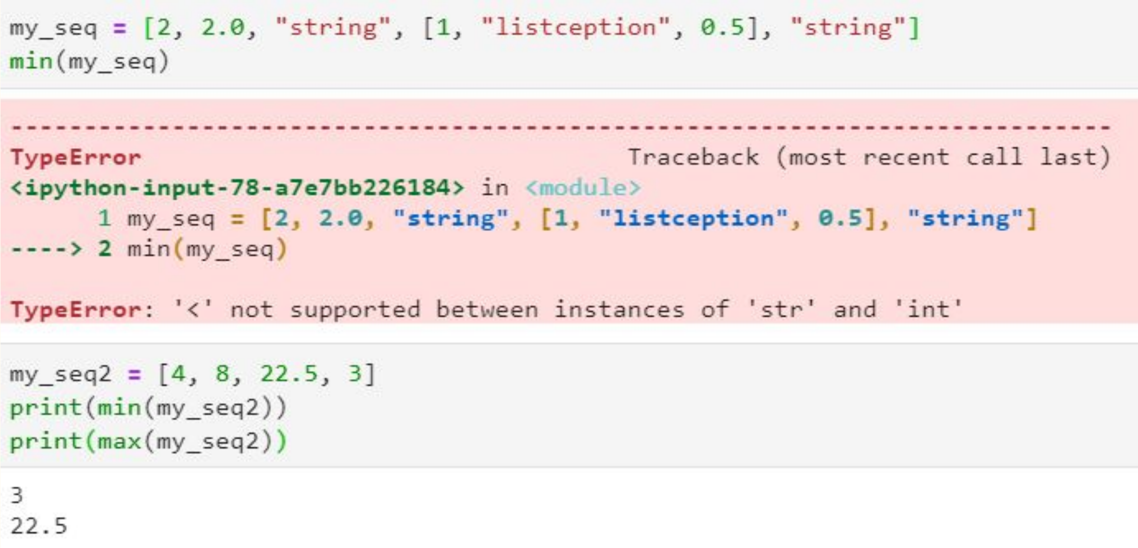
```
print(my_seq.count("string"))
print(my_seq.count(2))
print(my_seq.count(69))
```

```
2
2
0
```

Some Useful Sequence Functions

- **min(my_seq)** and **max(my_seq)** return the smallest and largest elements in my_seq, respectively.

- Note that if any two elements in my_seq are incomparable (e.g., a string and a number), min and max will return errors.



```
my_seq = [2, 2.0, "string", [1, "listception", 0.5], "string"]
min(my_seq)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-78-a7e7bb226184> in <module>
      1 my_seq = [2, 2.0, "string", [1, "listception", 0.5], "string"]
----> 2 min(my_seq)

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

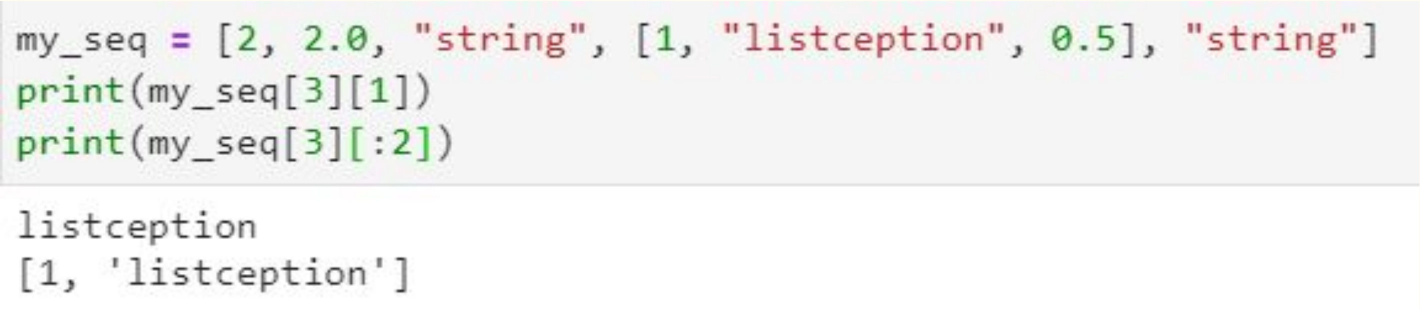
```
my_seq2 = [4, 8, 22.5, 3]
print(min(my_seq2))
print(max(my_seq2))
```

```
3
22.5
```

List Operations

- As mentioned, lists are the most versatile sequence, as they are mutable and can contain any object. There are some operations valid for lists that we didn’t see for strings.

- **Multiple indexing:** As lists can contain other sequences, you can access individual elements within nested sequences using multiple indexing.
  For example, double indexing:



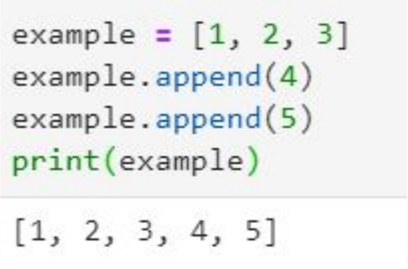
```
my_seq = [2, 2.0, "string", [1, "listception", 0.5], "string"]
print(my_seq[3][1])
print(my_seq[3][:2])
```

```
listception
[1, 'listception']
```

List Mutability

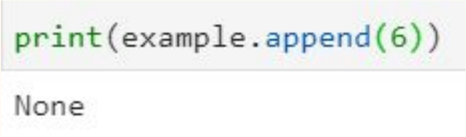
- List elements can be inserted, removed, or replaced. Not possible with tuples.

- **Add things with .append():**



```
example = [1, 2, 3]
example.append(4)
example.append(5)
print(example)
```

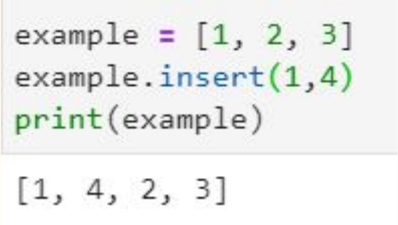
```
[1, 2, 3, 4, 5]
```



```
print(example.append(6))
```

```
None
```

- **Add things at a specific position with .insert():** my_list.insert(index, item)



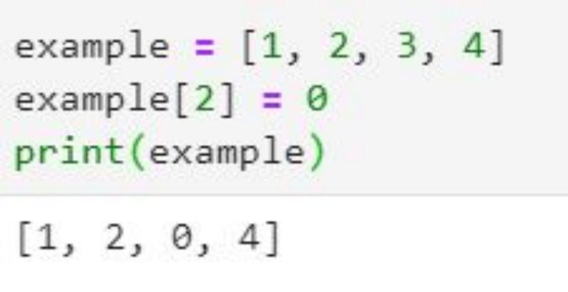
```
example = [1, 2, 3]
example.insert(1,4)
print(example)
```

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

List Mutability

- **Use the subscript operator [ ] to replace an element:**

  E.g., my_list[index] = new_element

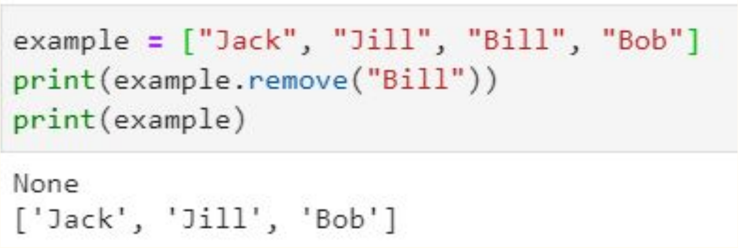


```
example = [1, 2, 3, 4]
example[2] = 0
print(example)
```

```
[1, 2, 0, 4]
```

List Mutability

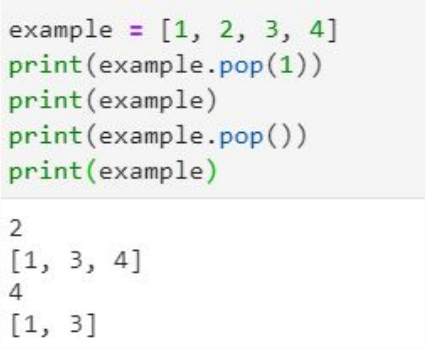
- **Remove things with .remove():** my_list.remove(item)



```
example = ["Jack", "Jill", "Bill", "Bob"]
print(example.remove("Bill"))
print(example)
```

```
None
['Jack', 'Jill', 'Bob']
```

- **Remove and obtain things with .pop():** my_list.pop(index) or my_list.pop()

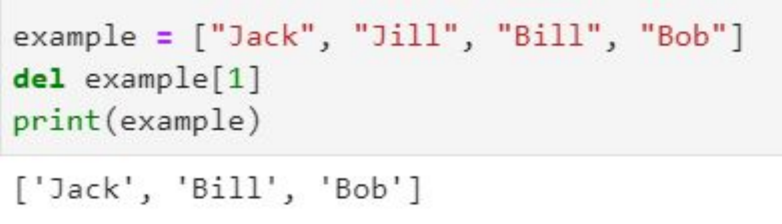


```
example = [1, 2, 3, 4]
print(example.pop(1))
print(example)
print(example.pop())
print(example)
```

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

List Mutability

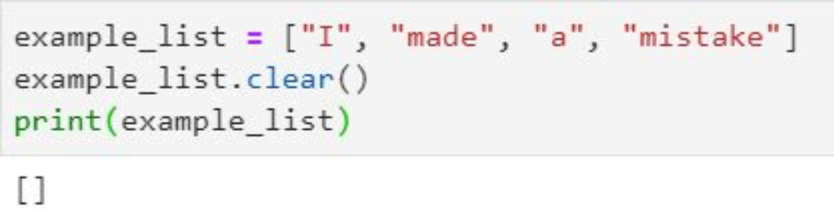
- **Remove any item at a specific index with del:** del my_list[index]



```
example = ["Jack", "Jill", "Bill", "Bob"]
del example[1]
print(example)
```

```
['Jack', 'Bill', 'Bob']
```

- **Remove all items using .clear():** my_list.clear()



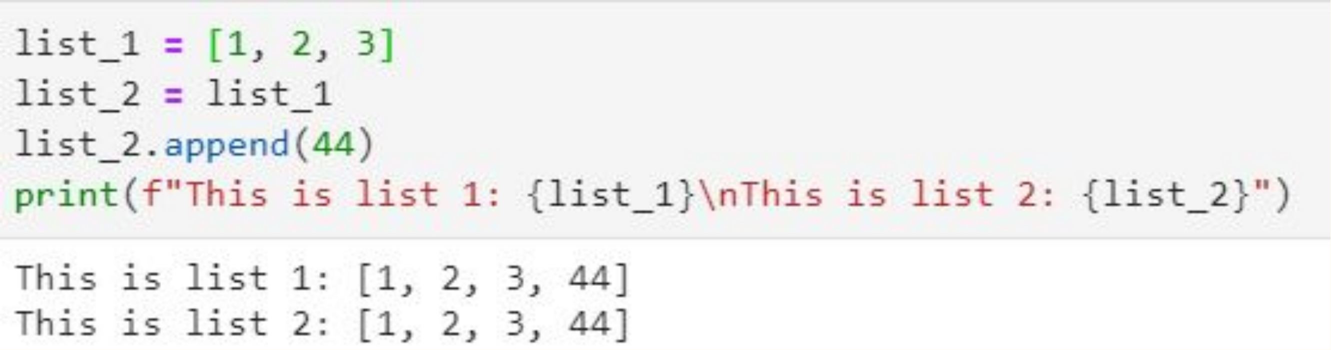
```
example_list = ["I", "made", "a", "mistake"]
example_list.clear()
print(example_list)
```

```
[]
```

Mutability - A Word of Caution

- Issues can be encountered when creating copies of a mutable object.

- **Aliasing** occurs when one mutable object’s value is assigned to another variable. If one of the variables is then mutated, the changes are applied to both variables!

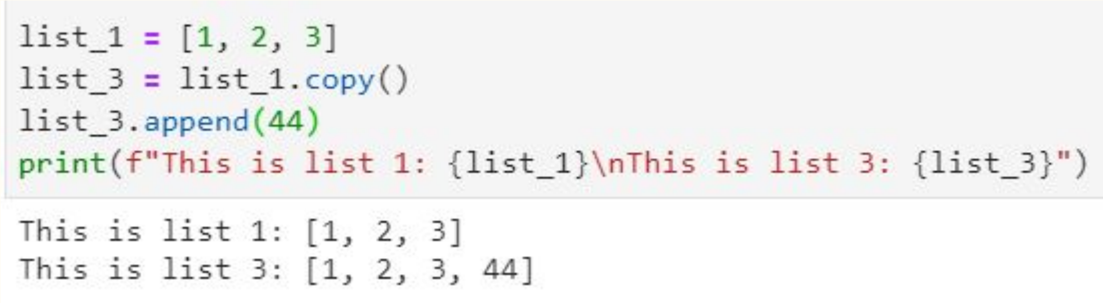


```python
list_1 = [1, 2, 3]
list_2 = list_1
list_2.append(44)
print(f"This is list 1: {list_1}\nThis is list 2: {list_2}")
```

```
This is list 1: [1, 2, 3, 44]
This is list 2: [1, 2, 3, 44]
```

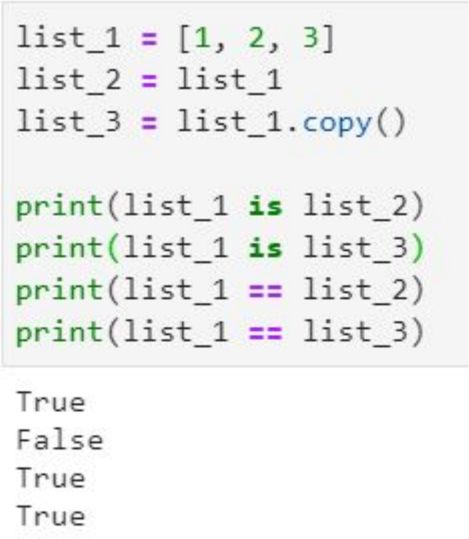
Mutability - A Word of Caution

- To avoid this, if we want to make a copy of a mutable object, we must use the **copy()** method: e.g., my_list.copy()



```
list_1 = [1, 2, 3]
list_3 = list_1.copy()
list_3.append(44)
print(f"This is list 1: {list_1}\nThis is list 3: {list_3}")
```

```
This is list 1: [1, 2, 3]
This is list 3: [1, 2, 3, 44]
```

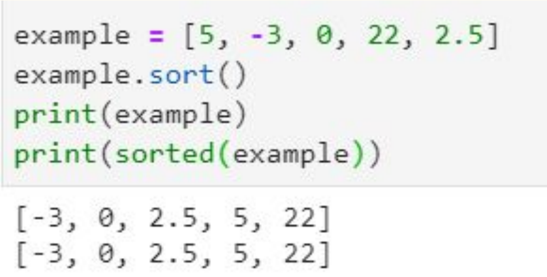


```
list_1 = [1, 2, 3]
list_2 = list_1
list_3 = list_1.copy()

print(list_1 is list_2)
print(list_1 is list_3)
print(list_1 == list_2)
print(list_1 == list_3)
```

```
True
False
True
True
```

Sorting

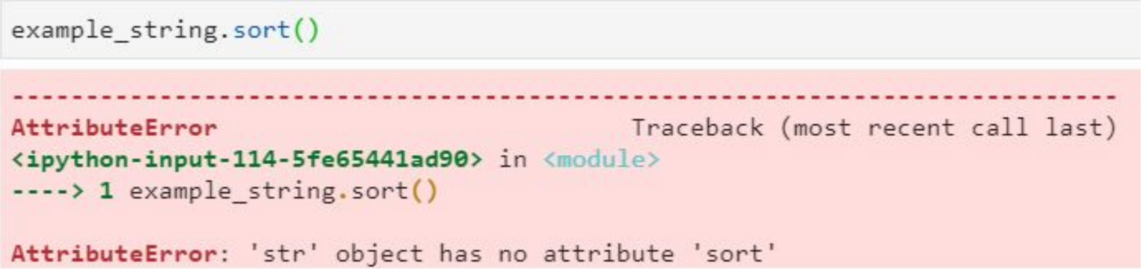
- **sorted() and .sort()** can be used to sort a list in ascending order.
- The elements all need to be comparable (i.e. can’t have numbers and strings).



```
example = [5, -3, 0, 22, 2.5]
example.sort()
print(example)
print(sorted(example))
```

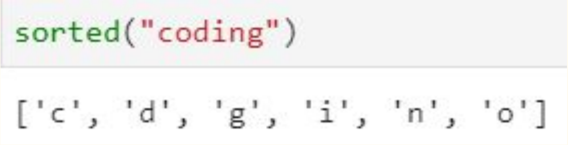
```
[-3, 0, 2.5, 5, 22]
[-3, 0, 2.5, 5, 22]
```

- **sorted()** returns a new list and can be used on strings. **sort()** mutates the list it is applied to and cannot be used on strings.



```
example_string.sort()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-114-5fe65441ad90> in <module>
----> 1 example_string.sort()

AttributeError: 'str' object has no attribute 'sort'
```

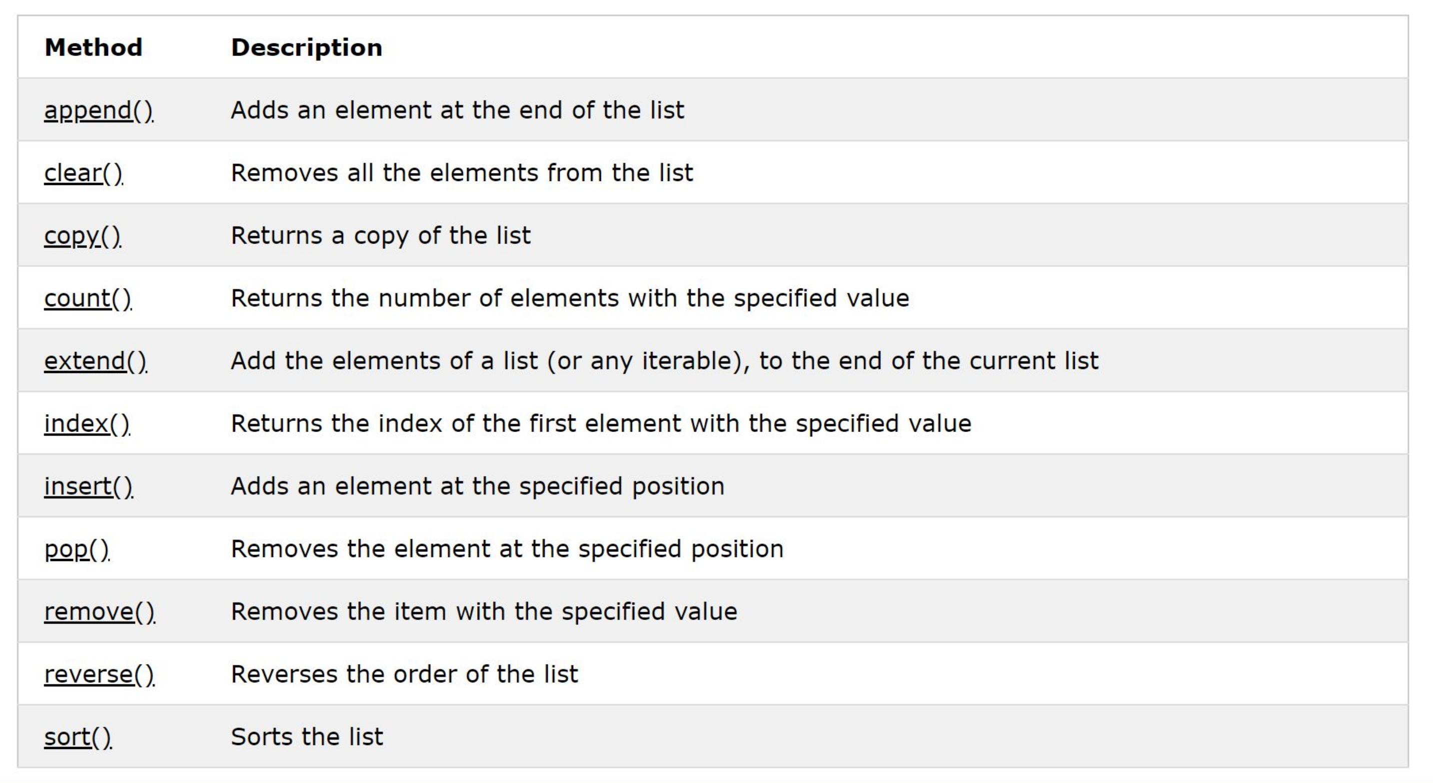


```
sorted("coding")
```

```
['c', 'd', 'g', 'i', 'n', 'o']
```

List Methods

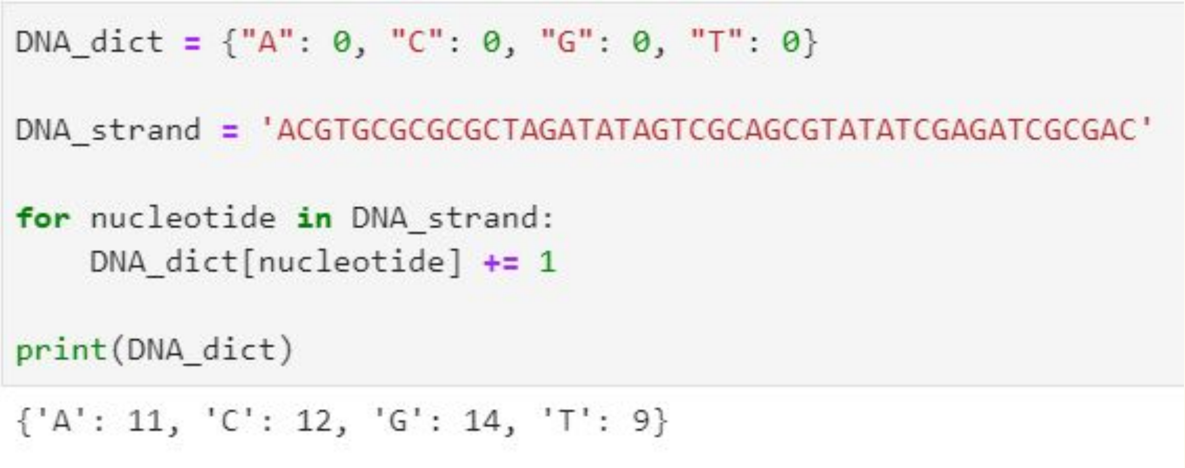
https://www.w3schools.com/python/python_lists.asp



Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Dictionaries

$$\text{my\_dict} = \{\underbrace{\text{'key1': 'value1'}}_{\text{item 1}},\ \underbrace{\text{'key2': 'value2'}}_{\text{item 2}}\}$$

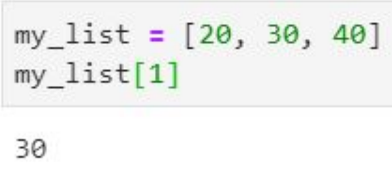


```
DNA_dict = {"A": 0, "C": 0, "G": 0, "T": 0}

DNA_strand = 'ACGTGCGCGCGCTAGATATAGTCGCAGCGTATATCGAGATCGCGAC'

for nucleotide in DNA_strand:
    DNA_dict[nucleotide] += 1

print(DNA_dict)
```

```
{'A': 11, 'C': 12, 'G': 14, 'T': 9}
```

Indexing in Dictionaries

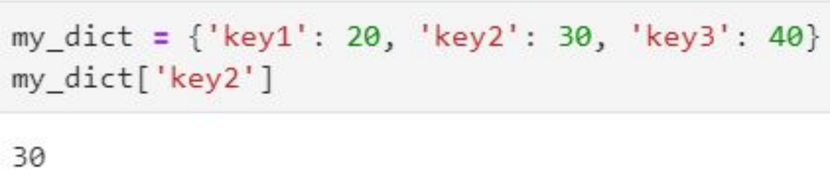
- We saw with sequences, such as lists, that we had numbered indexes.



```
my_list = [20, 30, 40]
my_list[1]
```

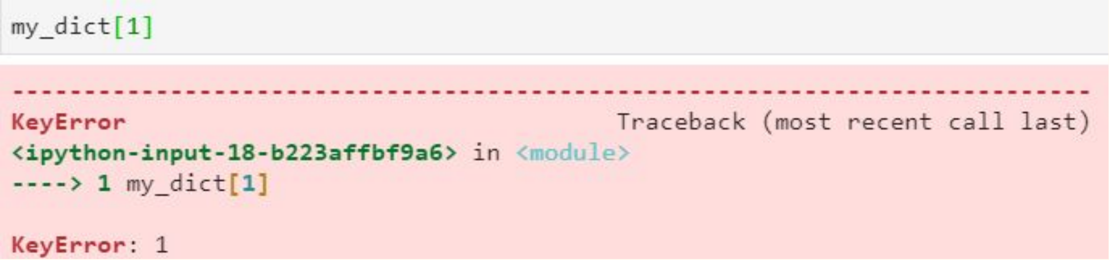
```
30
```

- However, with dictionaries, the keys are the indexes.



```
my_dict = {'key1': 20, 'key2': 30, 'key3': 40}
my_dict['key2']
```

```
30
```

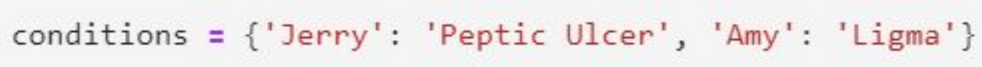


```
my_dict[1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-18-b223affbf9a6> in <module>
----> 1 my_dict[1]

KeyError: 1
```

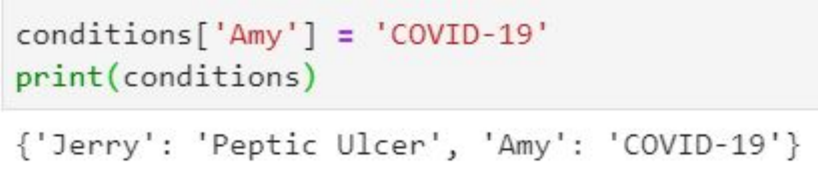
Updating Dictionaries

- Say we have a dictionary representing patients in a hospital ward and their conditions.



```
conditions = {'Jerry': 'Peptic Ulcer', 'Amy': 'Ligma'}
```

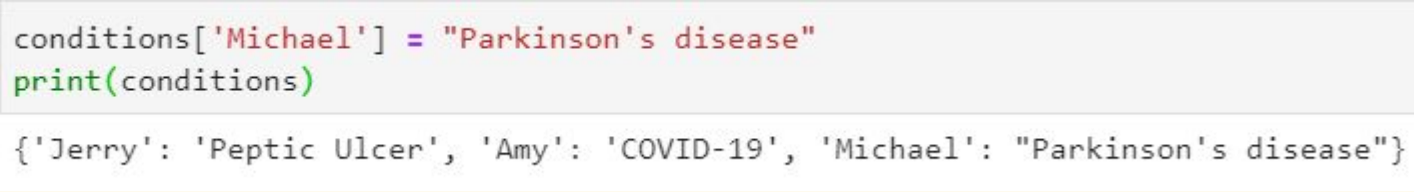
- Turns out Amy actually has COVID-19, so we have to update the info:



```
conditions['Amy'] = 'COVID-19'
print(conditions)
```

```
{'Jerry': 'Peptic Ulcer', 'Amy': 'COVID-19'}
```

- We also have to add a new patient, Michael, who has Parkinson's disease:



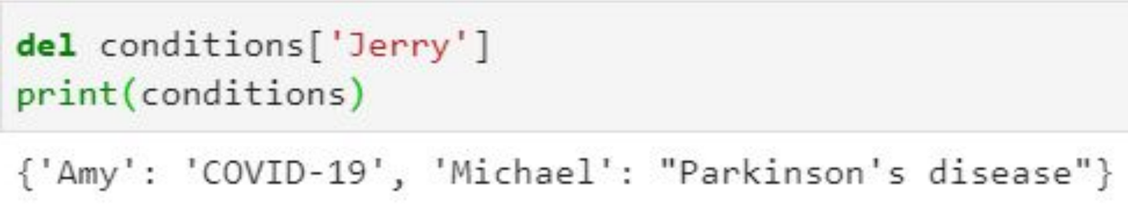
```
conditions['Michael'] = "Parkinson's disease"
print(conditions)
```

```
{'Jerry': 'Peptic Ulcer', 'Amy': 'COVID-19', 'Michael': "Parkinson's disease"}
```

Updating Variables

- Jerry has now recovered and we want to remove him from the dictionary of patients.

- We can use the **del** function that we saw earlier: del my_dict[“key”]

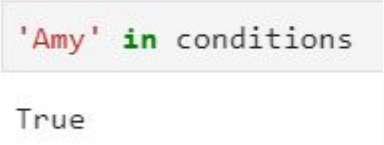


```
del conditions['Jerry']
print(conditions)
```

```
{'Amy': 'COVID-19', 'Michael': "Parkinson's disease"}
```

Looking Inside Dictionaries

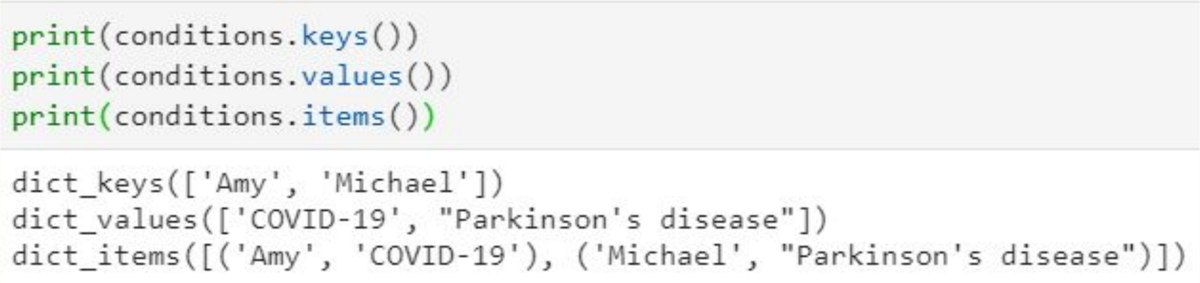
- Check if a key is in a dictionary, i.e., check if someone is a patient:



```
'Amy' in conditions
```

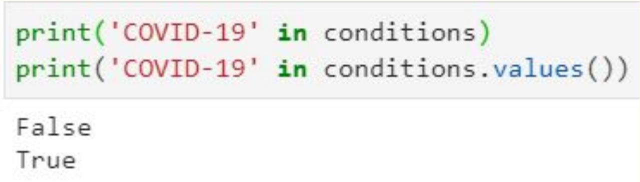
```
True
```

- Find out what's inside the dictionary:
  - my_dict.keys() - my_dict.values() - my_dict.items()



```
print(conditions.keys())
print(conditions.values())
print(conditions.items())
```

```
dict_keys(['Amy', 'Michael'])
dict_values(['COVID-19', "Parkinson's disease"])
dict_items([('Amy', 'COVID-19'), ('Michael', "Parkinson's disease")])
```



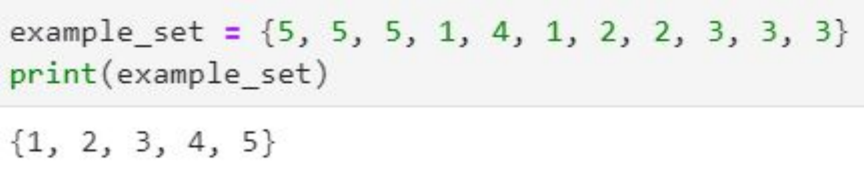
```
print('COVID-19' in conditions)
print('COVID-19' in conditions.values())
```

```
False
True
```

Sets

my_set = {'only', ‘unique’, ‘values’}

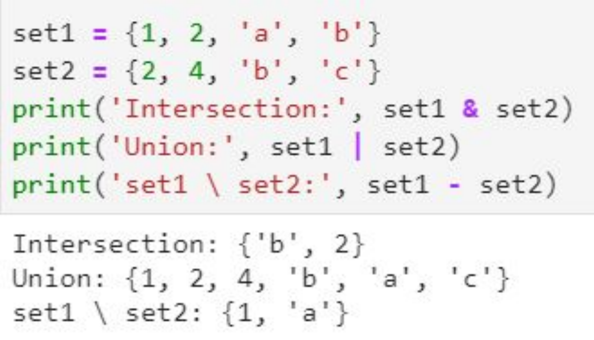
- Sets are unordered and cannot be referred to by index or key.



```
example_set = {5, 5, 5, 1, 4, 1, 2, 2, 3, 3, 3}
print(example_set)
```

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

- Sets can sometimes be useful for performing mathematical operations, such as finding intersections, unions, and differences.



```
set1 = {1, 2, 'a', 'b'}
set2 = {2, 4, 'b', 'c'}
print('Intersection:', set1 & set2)
print('Union:', set1 | set2)
print('set1 \ set2:', set1 - set2)
```

```
Intersection: {'b', 2}
Union: {1, 2, 4, 'b', 'a', 'c'}
set1 \ set2: {1, 'a'}
```

Summary of Non-Primitive Data Structures

Data Structure	**List**	**Tuple**	**Dictionary**	**Set**
Syntax	[1, “a”, etc]	(1, “a”, etc)	{key1: “value1”, etc}	{1, “a”, etc}
Usefulness	Very useful	so-so	Very useful	so-so
Indexing	Position number	Position number	Key	No indexing
Mutability	Mutable	Immutable	Mutable	Mutable