Biniam Abebe 03/27/2024

Week 2 Assignment Python Exercises

Complete the following sections:

- Basic Techniques of Programming
- Built-In Basic Data Types
- Python Data Structures: Lists
- Python Data Structures: Ranges
- Python Data Structures: Strings
- Python Data Structures: Tuples

Basic Techniques of Programming: Programming for Data Science with Python

Overview

To write a useful program, the developer needs to use various techniques of programming. Basic Techniques of Programming cover the techniques below.

Some of these techniques are considered as the core of programming to create interactive software applications:

- Input and output
- Selections
- Loops

1. Read Data from Console

Scenario

Write a short Python program that reads an integer value from the console and then prints out the value.

IMPORTANT NOTES: It is assumed that the user would not make any mistake while entering the value. Therefore, it is not necessary to check the input after reading it.

Syntax

To read data from the console in Python, use the built-in function:

- input(prompt_string)

Where prompt_string is the text used to prompt the user to enter the data.

IMPORTANT NOTES: We need to declare a new variable to store the text read from the console.

Run the following code:

In [ ]:

```
anIntValue = input("Enter an interger value:")
print ("The user has entered this value: ", anIntValue)
```

```
The user has entered this value:  5
```

2. Print Data to the Console

- To print data to the console, use the built-in function:

  > print(a_string)

- Where a string can be only one string or multiple sub-strings and values separated by commas','.
- To print the values in one line, using the end-of-line character'\n':

  > print (...,'\n')

Run the following code:

In [ ]:

```
print ("Examples of using print() function", "\n")

x = 15

print ("This is the value of x:", x, "\n")

y = 25

print ("This is the value of x:", x, "; This is the value of y:", y, ". \n")
```

```
Examples of using print() function

This is the value of x: 15

This is the value of x: 15 ; This is the value of y: 25 .
```

PRACTICE

- Change the value of x to 1001 and the value of y to 2001
- Print the following sentence using the print(a_string) built-in function

- The value of x is 1001 and y is 2001

In [ ]:

```python
#TO DO Add practice code here

x = 1001
y = 2001

print ("The value of x is", x, "and y is", y, "\n")
```

```
The value of x is 1001 and y is 2001
```

3. Selections

Scenario: A Problem

- Let's review the problem of calculating the diameter and circumference of a circle
- It is assumed that a software developer is asked to write a Python program that can calculate and print out the diameter and the circumference of a circle. The user enters data of the radius and its measurement unit (in, ft, cm, or m) from the console.

Let's imagine this scenario:

The user inadvertently enters a negative value of the radius, which raises the following question:

- Should we let the program ignore this error?
- The answer is definitely "NO."
- So, what should we do?
- We should add selections into our program to check the sign of the input.

Let's write a better pseudo-code:

```
1. Start
2. Read the input of the radius from the console
   if(radius<0): ←selection
   inform the user about the error
   request to read again
3. Read the measurement unit of the radius (in, ft, cm, m)
4. Calculate the diameter of the circle
```

diameter = 2 * radius

5. Calculate the circumference of the circle

   Circumference = diameter * PI (3.14159)

6. Print out the diameter

7. Print out the circumference

8. End

4. if Statements

1. Simple if:

if(boolean expression):

> //... ... ... statement(s)

Example:

numCredits = ... # number of credits an undergaduate student completed

if (numCredits >=90):

> studentLevel = "Senior"

2. if Statements: if...else:

if(boolean expression):

> //... ... ... statement(s)

else:

```
                    //... ... ... statement(s)
```

Example:

if(numCredits >= 120):

```
                readyToGraduate = True;
```

else:

```
                    readyToGraduate = False;
```

3. if ... elif ... elif ... else

if(boolean expression):

```
            // ... ... ... statement(s)
```

elif (boolean expression):

```
          // ... ... ..... statement(s)
```

elif (boolean expression):

```
          // ... ... ..... statement(s)
```

else:

```
          // ... ... ..... statement(s)
```

Example:

It is assumed that the Registrar Office of a university asks one analyst to provide a solution to the following problem:

Write a Python program that can read input from the console. The user enters a student's name and his/her level (freshmen, ..., senior). The program is expected to assign a numeric code that represents his/her priority to register for courses. Students with higher priority can register for courses before those with lower priority. The code starts from 1 (highest) assigned to seniors and increments by 1 for each lower level. Finally, the program prints out the student name, his/her level, and the code of priority to register courses in the same line.

Pseudo-Code with if ... elif ... elif ... else:

```
1. START
2. ... more code here ...
3. Perform the selection
    • If "senior", priorityToRegister = 1 // highest
    • If "junior", priorityToRegister = 2
```

  - If "sophmore", priorityToRegister = 3
  - If "freshman" , priorityToRegister = 4
  - If (not any above), print out warning of errors

4. ... more code here ...

5. END

Code:

studentLevel = ... #level: freshman, sophmore, junior, senior

if(studentLevel == "Senior")

```
priorityToRegister = 1
```

if(studentLevel == "Junior")

```
priorityToRegister = 2
```

if(studentLevel == "Sophmore")

```
priorityToRegister = 1
```

Run the following code:

In [ ]:

```
studentLevel= "Senior" # level: freshman, sophomore, junior, senior
if(studentLevel =="Senior"):
    prioritytoRegister = 1
elif(studentLevel =="Junior"):
    prioritytoRegister = 2
elif(studentLevel =="Sophomore"):
    prioritytoRegister = 3
elif(studentLevel =="Freshman"):
    prioritytoRegister = 4
else:
    print("Invalid studentLevel!!!")

print("studentLevel:", studentLevel, "; Priority to register",prioritytoRegister, "\n")
```

```
studentLevel: Senior ; Priority to register 1
```

5. Loops

Scenario: A Problem

Let's review the problem of calculating the diameter and circumference of a circle

```
It is assumed that a software developer is asked to write a Python
program that can calculate and print out the diameter and the
circumference of a circle. The user enters data of the radius and its
measurement unit (in, ft, cm, or m) from the console.
```

Let's write a Pseudo-Code:

```
1. Start
2. Read the input of the radius from the console
    • if (radius<0),
        ▪ inform the user about the error
        ▪ request to read again
3. Read the measurement unit of the radius (in, ft, cm, m)
    • if (unit is not among (in, ft, cm, m)),
        ▪ inform the user about the error
        ▪ request to read again
4. Calculate the diameter of the circle
    • diameter = 2 * radius
5. Calculate the circumference of the circle
    • Circumference + diameter * PI (3.14159)
6. Print out the diameter
7. Print out the circumference
8. End
```

Let's focus on this piece of pseudo-code:

```
Read the input of the radius from the console
• if (radius < 0),
    ▪ inform the user about the error
    ▪ request to read again
```

What happens if the user makes mistakes while entering the radius data again and again?

```
The program must perform the checking again and again until it can
read a valid piece of data._
```

In other words, the program must repeatedly check the input until it gets the correct one → the program uses LOOPS.

1. while Loop

Syntax

while (<loop-continuation condition.):

```
// ... ... ... statements(s)
```

Example:

radius = ... # radius of the circle

while (radius < 0):

```
print ("Radius cannot be negative!!!")

print ("Enter radius:")
```

radius = input("Enter radius: ")

IMPORTANT NOTES:

- In Python, WHILE loop also has an "ELSE" statement as IF does.
- However, it is strongly discouraged from using the ELSE statement of WHILE because it causes confusion and can make the code too complicated.
- For more details, see the example in the following section of FOR loop.

2. for Loop

The Python for loop is an iterator based for loop.

It steps through the items of lists, tuples, strings, the keys of dictionaries, and other iterables.

The Python for loop starts with the keyword for followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through.

Syntax

for (variable) in (sequence):

```
// ... ... ... statement(s)
```

Run the following code:

In [ ]:

```
language = ["C", "C++", "Java", "Python", "Perl", "Ruby", "Scala"]
for x in language:
    print (x)
```

```
C
C++
Java
Python
Perl
Ruby
Scala
```

PRACTICE

- Create a for Loop that displays four or more characteristics of Big Data
- Hint: Volume, Velocity, and so on.
- Check the lecture slides if you cannot remember

In [ ]:

```
language = ["Volume", "Velocity", "Variety","Veracity", "Variability", "Value"]
for x in language:
    print (x)
```

```
Volume
Velocity
Variety
Veracity
Variability
Value
```

IMPORTANT NOTES:

In Python, FOR loop also has an optional "ELSE" statement as the IF statement does.

However, it is strongly discouraged from using the ELSE statement of FOR loop because it causes confusion and complicates the code.

(Remember the Zen of Python: ... ***Simple is better than complex! Complex is better than complicated!***)

Semantically, the optional ELSE of FOR loop works exactly as the optional ELSE of a WHILE loop:

- It will be executed only if the loop hasn't been "broken" by a BREAK statement.
- So it will only be executed after all the items of the sequence in the header have been iterated through.

If a BREAK statement has to be executed in the program flow of the for loop:

- The loop will be exited

- The program flow will continue with the first statement following the FOR loop if there are any.

Usually, BREAK statements are wrapped into conditional statements.

Run the following code:

In [ ]:

```
edibles = ["ham", "Spam", "eggs", "nuts"]

for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
        print ("reat, delicious " + food)
    else:
        print("I am so glad: No spam!")
        print ("Finally, I finished stuffing myself")
```

```
I am so glad: No spam!
Finally, I finished stuffing myself
I am so glad: No spam!
Finally, I finished stuffing myself
I am so glad: No spam!
Finally, I finished stuffing myself
I am so glad: No spam!
Finally, I finished stuffing myself
```

Let's consider the "else" statement in the FOR loop

for (variable) in (sequence):

```
#statements ...
```

else:

```
#statements ...
```

What does it mean by the "ELSE" statement?

Based on the syntax, it seems that the execution of the ELSE block is only based on the state of the conditional expression of the FOR statement - no other conditions.

However, semantically, it is not true!

The execution of the ELSE block only becomes meaningful due to the existence of a BREAK statement embedded inside another conditional statement like IF.

In the above piece of code, intuitively, the ELSE block would be executed if the conditional expression of FOR statement, i.e., "food is edibles", gets a "False" value - when the FOR loop has iterated through the whole sequence.

Let's execute the following piece of code that lets the FOR loop iterate through its sequence:

Run the following code:

In [ ]:

```
edibles = ["ham", "spam", "eggs", "nuts"]
for food in edibles:
    print("No break in FOR loop statements")
else:
    print ("I am so glad: No spam!")
print ("Finally, I finished stuffing myself")
```

```
No break in FOR loop statements
No break in FOR loop statements
No break in FOR loop statements
No break in FOR loop statements
I am so glad: No spam!
Finally, I finished stuffing myself
```

NOTES: From the results of the above piece of code, the FOR loop iterated through the whole sequence. Therefore, the ELSE block is executed. However, the output of the ELSE block is not only meaningless but also misleading! There is "Spam" in the sequence! Let's find out what the developer really wants to achieve with the above ELSE statement - He/she wants to find out if "spam" is listed in the list of edibles by using FOR loop to iterate through the sequence. - If "spam" is found, he/she prints out a dialog to inform that. - Otherwise, he/she is so happy to print out that there is no "spam" in the list. Let's write another much simpler piece of code to achieve what he/she wants

Run the following 2 blocks of code:

In [ ]:

```
edibles = ["ham", "spam", "eggs","nuts"]
spam=False

for food in edibles:
    if food == "spam":
        spam = True
        print("No more spam please!")
        break
    print ("Great, delicious " + food)

if (not spam):
    print("I am so glad: No spam!")

print("Finally, I finished stuffing myself")
```

```
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
```

In [ ]:

```
#NO spam

edibles =["ham","eggs","nuts"]
spam = False
```

```
for food in edibles:
    if food == "spam":
        spam = True
        print("No more spam please!")
        break
    print("Great, delicious " + food)
if (not spam):
        print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious ham
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
```

IMPORTANT NOTES:

By not using the ELSE statement, the code is much easier to understand, cleaner, and simpler - following the Zen of Python:

> ***Simple is better than complex!***
>
> ***Complex is better than complicated!***

3. Break

"Break" is used to terminate a loop, i.e., completely get out of the loop immediately.

Run the following 4 blocks of code:

In [ ]:

```
for x in range (10,20):
       if (x == 15): break
       print (x)
```

```
10
11
12
13
14
```

In [ ]:

```
for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
s
t
r
The end
```

In [ ]:

```
for letter in 'Python':
   if letter == 'h':
      break
   print ('Current Letter :', letter)
print ("Good bye!")
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Good bye!
```

In [ ]:

```
# You will need to enter a price once you run the code.  Use high numbers since the break
#  Hit return once you enter an amount

numItems=0
totalSales=0
totalSoldItems=5000

while(numItems<totalSoldItems):
    price=int(input("Enter the price of the next sold item: "))
    totalSales=totalSales+price
    if(totalSales>=1000000):
        break;
    numItems=numItems+1
print(totalSales)
```

```
1999766
```

4. Continue

"Continue" is used to skip the rest of an itinerary and start a new one.

Run the following 2 blocks of code:

In [ ]:

```
for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
s
t
r
n
g
The end
```

In [ ]:

```
var = 10
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

```
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Built-In Basic Data Types:

Programming for Data Science with Python

Overview

Python: Everything is an Object

In Python, everything is an object

- All values are objects
- Anything which can be used as a value (int, str, float, functions, etc.) are implemented as objects.

Built-In Basic Data Types

- NUMERIC:

> - Integers: int
> - Floating point numbers: float
> - Complex numbers: complex

- LOGICAL/BOOLEAN: Boolean values: bool The focus: the integers and the floats

1. Numeric Data Types

1.1 Integers: int

Run the following code:

In [ ]:

```
x = 3

y = x

print ("Data type of x: ", type(x), '\n')
print ("Data type of y: ", type(y), '\n')
```

```
Data type of x:  <class 'int'>

Data type of y:  <class 'int'>
```

1.2 Floating-Point Numbers: float

Run the following code:

In [ ]:

```
x = 3.5

y = x

print ("Data type of x: ", type(x), '\n')
print ("Data type of y: ", type(y), '\n')
```

```
Data type of x:  <class 'float'>

Data type of y:  <class 'float'>
```

1.3 Complex Numbers: complex

A complex number is represented by “x + yj”.

- Python converts the real numbers x and y into complex using the function complex (x,y).
- The real part can be accessed using the function real() and imaginary part can be represented by imag(). ### **Run the following code:**

In [ ]:

```
x = 5

y = 3

aComplex = complex(5,3)

print ("aComplex is a complex number: ", aComplex, '\n')
print ("Data type of aComplex: ", type(aComplex), '\n')
```

```
aComplex is a complex number:  (5+3j)

Data type of aComplex:  <class 'complex'>
```

2. Logical Data Types/Boolean Values: bool

Run the following code:

In [ ]:

```
boolVar = True
print ("boolVar is a boolean variable: ", boolVar, '\n')
print ("Data type of boolVar: ", type(boolVar), '\n')
```

```
boolVar is a boolean variable:  True

Data type of boolVar:  <class 'bool'>
```

IMPORTANT NOTES:

Any values that are ***NOT 0*** or null can be used the “True” Boolean value in Python.

Run the following 2 code blocks:

In [ ]:

```
boolVar = 5

if (boolVar):
    print ("Data type of boolVar: ", type(boolVar), '\n')
```

```
Data type of boolVar:  <class 'int'>
```

In [ ]:

```
boolVar = False
print ("boolVar is a boolean variable: ", boolVar, '\n')
print ("Data type of boolVar: ", type(boolVar), '\n')
```

```
boolVar is a boolean variable:  False

Data type of boolVar:  <class 'bool'>
```

IMPORTANT NOTES:

Any zero values like ***0 or null*** can be used the “False” Boolean value in Python.

In [ ]:

```
boolVar=0

if (boolVar):
    print ("Data type of boolVar: ", type(boolVar), '\n')
```

IMPORTANT NOTES:

In the above code, the value 0 can be used as “False.” Therefore, nothing is printed out when the code in the above cell is executed.

3. Character Data Types

NOTES about character data types

- Python does not support character data type (char).
- It supports string and the characters as string of length one.

  ### **Run the following code:**

In [ ]:

```
aChar = 'a'
print ("aChar is a String variable, NOT a Character variable: ", aChar, "\n")
print ("ata type of aChar:",type(aChar),'\n')
```

```
aChar is a String variable, NOT a Character variable:  a
```

```
ata type of aChar: <class 'str'>
```

Python Data Structures: Lists

Programming for Data Science with Python

1. Overview

In Python, ***lists*** are the objects of the class list that has the ***constructor list()***.

A list is a ***mutable*** sequence data type/structure, i.e., its ***contents can be changed*** after being created.

List literals are written within square brackets [ ].

Lists work similarly to strings:

- Use the len() function for the length of a list
- Use square brackets [ ] to access data, with the first element at index 0
- The range of indices: 0 .. len(a list) - 1

1.1 Properties of Lists

The ***main properties*** of Python lists:

- List elements are ordered in a sequence.
- List contain objects of different data types
- Elements of a list can be accessed by an index - as other sequence data type/structures like strings, tuples
- Lists are arbitrarily nestable, i.e. they can contain other lists as sublists
- Lists are ***mutable***, i.e. their elements can be changed after the list has been created.

  #### Examples:

Empty list

> my_list=[]

List of integers

> my_list = [1,2,3]

List with mixed datatypes

> my_list = [1, "Hello", 3.4]

Nested list

> my_list =["mouse", [8,4,6], ['a']]

1.2. Elements of a list

Index range of list elements

Forward index range of list elements: ***0 .. len(list) - 1*** Forward: starting from the 1st element

Backward index range of list elements: -1 .. -len(list) Backward : Starting from the last element

1.3. Constructor list(iterable)

The ***constructor list()*** builds a list whose items are the same and in the same order as iterable's items.

- ***iterable*** may be either a sequence, a container that supports iteration, or an iterator object.
- If ***iterable is already a list, a copy*** is made and returned, similar to iterable[:].

For example:

- list('abc') returns ['a', 'b', 'c']
- list((1, 2, 3)) returns [1, 2, 3].

If ***no argument*** is given, the constructor creates a ***new empty list, []***.

Run the following 3 code blocks:

In [ ]:

```
list("abc")
```

Out[ ]:

```
['a', 'b', 'c']
```

In [ ]:

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

Out[ ]:

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

In [ ]:

```
list ([1, 3, 5, 7, 9])
```

Out[ ]:

```
[1, 3, 5, 7, 9]
```

2. Create Lists

2.1 Overview

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the ***empty list: []***
- Using square brackets with values separating from each others with commas: [a], [a, b, c]
- Using a ***list comprehension:*** [x for x in iterable]
- Using the ***list constructor:*** list() or list(iterable)

2.2 Create empty lists

Run the following code block:

In [ ]:

```
empty_list = []
another_empty_list = list()
print(len(empty_list))
print(len(another_empty_list))
```

```
0
0
```

2.3 Create lists by converting other data structures/types to lists: Using list()

> ### 2.3.1 Create list from strings or tuples using the constructor list()
>
> ## **Run the following 3 code blocks:**

In [ ]:

```
# Convert a string of one word to a list of characters
list("house")
```

Out[ ]:

```
['h', 'o', 'u', 's', 'e']
```

In [ ]:

```
# Convert a a string of words to a list of characters
list("This word")
```

Out[ ]:

```
['T', 'h', 'i', 's', ' ', 'w', 'o', 'r', 'd']
```

In [ ]:

```
# Convert a tuple of a list
# Notice the parentheses vs. the square brackets
aTuple = ('ready', 'fire', 'aim')
list(aTuple)
```

Out[ ]:

```
['ready', 'fire', 'aim']
```

> ### 2.3.2 Create lists from strings using split() method
>
> ## **Run the following 2 code blocks but change the date to today:**

In [ ]:

```
#Convert a string of words to a list of words: Using split() to chop the string with ' '

aStringOfWords= "This is a string of words"
aList=aStringOfWords.split(' ')
print(aList)
```

```
['This', 'is', 'a', 'string', 'of', 'words']
```

In [ ]:

```
#Convert a string to a List: Using split() to chop the string with some separator
aDayString = "2/1/2022"
alist = aDayString.split('/')
print(alist)
```

```
['2', '1', '2022']
```

> ### 2.3.3 Create lists by using list comprehension and slicing an existing list
>
> **Run the following code block:**

In [ ]:

```
# NOTES: MUST use List slice--> CANNOT use any other function to delete/remove

l_lists=[[1,2,3],[2,3,4],[3,4,5]]

new_llists=[element[1:] for element in l_lists]

i=0
for element in new_llists:
    print(element)
    i=i+1
    if i==3:
        break
```

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

3. Access List Elements

3.1 Access single elements

- As other sequence data types/structures, list elements can be accessed via their indices.
- We can use the index operator [] to access an item in a list. ***Index starts from 0.***
- So, a list having 5 elements will have index from O to 4.
- Trying to access an element other than this will raise an IndexError.

- ***The index must be an integer.***

- We can't use float or other types, this will result into TypeError.

Nested list are accessed using ***nested indexing [][]*** that is similar to index of 2-D array elements.

Run the following 6 code blocks:

In [ ]:

```
my_list = ['p','r','o','b','e']


print(my_list[0])

print(my_list[2])

print(my_list[4])
```

```
p
o
e
```

In [ ]:

```
# Nested List

n_list = ["Happy", [2,0,1,5]]

# Nested indexing

print(n_list[0][1])

print(n_list[1][3])
```

```
a
5
```

In [ ]:

```
aTuple=('ready','fire','aim')
aList=list(aTuple)

print (aList)
print("Length of the list:",len(aList))
```

```
['ready', 'fire', 'aim']
Length of the list: 3
```

In [ ]:

```
# Access using forward index

aTuple=('ready','fire','aim')
aList=list(aTuple)

list_element1=aList[0]
list_element2=aList[1]
list_element3=aList[2]

print(list_element1)
print(list_element2)
print(list_element3)
```

```
ready
fire
aim
```

In [ ]:

```
# Access using backward index
aTuple=('ready','fire','aim')
```

```
aList=list(aTuple)

list_element_last=aList[-1]
list_element_next_to_last=aList[-2]
list_element_first=aList[-3]

print(list_element_last)
print(list_element_next_to_last)
print(list_element_first)
```

```
aim
fire
ready
```

In [ ]:

```
languages= ["Python", "C", "C++", "Java", "Perl"]
print(languages[0] +" and "+ languages[1] +" are quite different!")
```

```
Python and C are quite different!
```

PRACTICE

- Change the languages to English, Spanish, French, Chinese, Hindi, and Arabic
- Print the following sentence
- English and Spanish are the most common languages in the great state of Texas!

In [ ]:

```
#To Do add your code here
languages= ["English", "Spanish", "French", "Chinese", "Hindi",  "Arabic"]
print(languages[0] +" and "+ languages[1] +" are the most common languages in the great s
```

```
English and Spanish are the most common languages in the great state of Texas!
```

3.2 Access a slice of lists

Run the following code block:

In [ ]:

```
# We can access a range of items in a List by using the slicing operator (colon).
# This is a very important concept for when we start working with algorithms in the 2nd h

my_list = ['p','r','o','g','r','a','m','i','z']

# elements 3rd up to the 5th (but not including)
print(my_list[2:5])

# elements backward from (but not inclucing) the negative 5th element ("r")
print(my_list[:-5])

# elements 6th to end
# Remember the count starts at zero, not one
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

```
['o', 'g', 'r']
['p', 'r', 'o', 'g']
```

```
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

4. Modify Lists

4.1 Add/Change elements of lists

> ### 4.1.1 Update/Change single elements or a sub-list of lists

Run the following code block:

In [ ]:

```
odd= [2, 4, 6, 8]

# change the 1st item
odd[0] = 1
print(odd)

# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
print(odd)
```

```
[1, 4, 6, 8]
[1, 3, 5, 7]
```

> ### 4.1.2 Add single items or a sub-list into a list - using append() or extend() respectively

Run the following code block:

In [ ]:

```
# We can add one item to a List using append() method
# or add several items using extend() method.
odd= [1, 3, 5]

odd.append(7)
print(odd)

odd . extend([9, 11, 13])
print(odd)
```

```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

> ### 4.1.3 Insert single elements or sub-lists into an existing list

Run the following code block:

In [ ]:

```
# We can insert one item at a desired Location by using the method insert()
# or insert multiple items by squeezing it into an empty slice of a List.

odd= [1, 9]
odd.insert( 1,3)
print(odd)
```

```
odd[2:2] = [5, 7]
print(odd)
```

```
[1, 3, 9]
[1, 3, 5, 7, 9]
```

4.2 Delete/Remove elements of lists

> ### 4.2.1 Delete/Remove elements of lists - using the del() function

Run the following code block:

In [ ]:

```
# We can delete one or more items from a List using the keyword del.

my_list = ["p","r", "o", "b", "l", "e", "m"]

# delete one item
del my_list[2]
print("3rd element has been removed: ", my_list)

# delete multiple items
del my_list[1:5]
print("Elements from index 1 until 4 have been removed: ", my_list)
```

```
3rd element has been removed:  ['p', 'r', 'b', 'l', 'e', 'm']
Elements from index 1 until 4 have been removed:  ['p', 'm']
```

> ### 4.2.2 Delete/Remove elements of lists - using the functions remove() or pop{)

Run the following code block:

In [ ]:

```
# We can use remove() method to remove the given item or pop() method to remove an item a
# The pop() method removes and returns the last item if index is not provided.
# This helps us implement lists as stacks (first in, last out data structure).
# We can also use the clear() method to empty a List.

my_list=['p','r','o','b','l','e','m']

# Remove p, p is gone. ("r", "o", "b", "l", "e", "m") is left.
my_list.remove('p')

# Will now remove the first element ("o"). ("r","b","l","e","m") is left.
my_list.pop(1)

# Will now remove the last element
my_list.pop()

print(my_list)
```

```
['r', 'b', 'l', 'e']
```

> ### 4.2.3 Delete/Remove elements of a list - assigning an empty list [] to a slice of the list

Run the following code block:

In [ ]:

```
my_list=['p','r','o','b','l','e','m']

# remove 'o'
my_list[2:3]=[]

# remove 'b', 'l', 'e'
my_list[2:5]=[]

print(my_list)
```

```
['p', 'r', 'm']
```

> ### 4.2.4 Delete/Remove all the elements of a list - using the clear() function

Run the following code block:

In [ ]:

```
my_list=['p','r','o','b','l','e','m']
my_list.clear()

print(my_list)
```

```
[]
```

5. Copy Lists

5.1 Shallow copy

- ***Shallow copy*** means that only the reference to the object is copied. No new object is created.
- ***Shallow Copy*** means defining a new collection object and then populating it with references to the child objects found in the original.
- The ***Shallow Copy*** process is not recursive. This means that the child objects won't be copied. In case of shallow copy, a reference of object is copied in other object. It means that any changes made to a copy of object do reflect in the original object. In python, this is implemented using “copy()” function.

Run the following code block:

In [ ]:

```
# importing "copy" for copy operations
import copy

# initializing list 1
i1 = [1, 2, [3,5], 4]

# using copy to shallow copy
s2 = copy.copy(i1)
```

```
# original elements of list
print ("The original elements before shallow copying")
for i in range(0,len(i1)):
        print (i1[i],end=" ")

print("\n")

# modifying the new list (shallow copy)
s2[2][0] = 7

# checking if change is reflected
print ("The original elements after shallow copying")
for i in range(0,len( i1)):
        print (i1[i],end=" ")
```

```
The original elements before shallow copying
1 2 [3, 5] 4

The original elements after shallow copying
1 2 [7, 5] 4
```

5.2 Deep copy

- The ***Deep Copy*** process is where the copying process occurs recursively.
- ***Deep copy*** means a new collection will first be created and then that copy will recursively be populated with copies of the child objects found in the original list.
- A ***Deep Copy*** stores copies of an object's values, but a ***Shallow Copy*** stores references to the original object(list, dict, etc)
- A ***Deep Copy** does ***NOT*** reflect any changes made to the new (copied) object from the original object; however, the ***Shallow Copy*** does reflect any modifications.
- A ***Deep Copy*** is the ***real copy*** of the orginal.
- Deep copying lists can be done using the ***deepcopy()*** function of the ***module copy*** in Python 3.

Run the following code block:

In [ ]:

```
# importing "copy" for copy operations
import copy

# initializing list 1
i1 = [1, 2, [3,5], 4]

# using deepcopy() to deep copy initial list (il)
d2 = copy.deepcopy(i1)

# original elements of list
print ("The original elements before deep copying")
for i in range(0,len(i1)):
        print (i1[i],end=" ")

print("\n")
```

```
# adding and element to new list
d2[2][0] = 7

# Change is reflected in l2
print ("The new list of elements after deep copying ")
for i in range(0,len( i1)):
        print (d2[i],end=" ")

print("\n")

# Change is NOT reflected in original list
# as it is a deep copy
print ("The original elements after deep copying")
for i in range(0,len( i1)):
        print (i1[i],end=" ")
```

```
The original elements before deep copying
1 2 [3, 5] 4

The new list of elements after deep copying
1 2 [7, 5] 4

The original elements after deep copying
1 2 [3, 5] 4
```

6. Delete Lists

To delete a list, using the built-in function del().

Run the following 3 code blocks:

In [ ]:

```
list1 = [1, 2, [3,5], 4]
print(list1)
```

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

In [ ]:

```
del(list1)
print("list1 has been deleted.")
```

```
list1 has been deleted.
```

In [ ]:

```
print(list1)
# You will get an error since list1 has been deleted.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-476b7d506017> in <module>
----> 1 print(list1)
      2 # You will get an error since list1 has been deleted.

NameError: name 'list1' is not defined
```

To Do: Please state why you received an error. I received an error because ...

7. Operations on List

Lists implement all of the common and mutable sequence operations.

7.1 Concatenate lists

Using + to concatenate strings

Run the following 2 code blocks:

In [ ]:

```
list1 = [1, 2, [3,5], 4]
list2 = ["Hello", "World"]
print(list1 + list2)
```

```
[1, 2, [3, 5], 4, 'Hello', 'World']
```

In [ ]:

```
# We can also use+ operator to combine two lists.
#This is also called concatenation.
#The * operator repeats a list for the given number of times.

odd= [1, 3, 5]


print(odd + [9, 7, 5])
```

```
[1, 3, 5, 9, 7, 5]
```

7.2 Replicate lists

Run the following 2 code blocks:

In [ ]:

```
aList = [1, 2]

print (aList * 3)
```

```
[1, 2, 1, 2, 1, 2]
```

In [ ]:

```
print(["re"] * 3)
```

```
['re', 're', 're']
```

7.3 Test elements with "in" and "not in"

Run the following 2 code blocks:

In [ ]:

```
list1 = [1, 2, [3,5], 4]
print (2 in list1)
```

```
True
```

In [ ]:

```
list1 = [1, 2, [3,5], 4]
print ([3] in list1)
```

```
False
```

7.4 Compare lists: <, >, <=, >=, ==, !=

Run the following code block:

In [ ]:

```
list1 = [1, 2, [3,5], 4]
list2 = [1, 2, 4]
print (list1 == list2)
```

```
False
```

7.5 Iterate a list using for loop

Run the following 4 code blocks:

In [ ]:

```
list1 = [1, 2, [3,5], 4]
for i in list1:
    print (i)
```

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

In [ ]:

```
list1 = [1, 2, [3,5], 4]

for i in list1:
    print(i, end="")
```

```
12[3, 5]4
```

In [ ]:

```
list1 = [1, 2, [3,5], 4]
for i in list1:
    print (i, end="\n")
```

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

In [ ]:

```
for fruit in ["apple","banana","mango"]:
    print("I like",fruit)
```

```
I like apple
I like banana
I like mango
```

7.6 Sort lists

> ### 7.6.1 Using the sort method of the class list: sort (*, key = none, reverse = false)

This method list.sort():

- Sort the list in ***place***
- Use only < comparisons between items.

By default, sort() doesn't require any extra parameters . However, it has two optional parameters :

- reverse - If true, the sorted list is reversed (or sorted in descending order)
- key - function that serves as a key for the sort comparison

IMPORTANT NOTES:

This method modifies the sequence in place for economy of space when sorting a large sequence. Exceptions are not suppressed.

- if any comparison opertions fail, the entire sort operation will fail
- the list will likely be left in a partially modified state.

  ### **Run the following code block:**

In [ ]:

```
# vowels list
vowels= ['e', 'a', 'u', 'o', 'i']

# sort the vowels
vowels.sort()

# print vowels
print('Sorted list:', vowels)
```

```
Sorted list: ['a', 'e', 'i', 'o', 'u']
```

> ### 7.6.2 Using the built-in sorted() function: sorted(iterable, *, key = None, reverse = False)

The built-in sorted() function returns a new sorted list from the items in iterable.

## **Run the following code block:**

In [ ]:

```
# vowels list
vowels= ['e', 'a', 'u', 'o', 'i']

# sort the vowels
sortedVowels = sorted(vowels)

# print vowels
print('Sorted list:', sortedVowels)

#A new list has been created and returned by the built-in sorted function
id(vowels), id(sortedVowels)
```

```
Sorted list: ['a', 'e', 'i', 'o', 'u']
```

Out[ ]:

```
(2415548082368, 2415561158976)
```

8. Class list

8.1 Count()

count(x): return the number of elements of the tuple that are equal to x

Run the following code block:

In [ ]:

```python
list1 = ['a','p','p','l','e']
print(list1.count('p'))
```

```
2
```

8.2 index (x)

index(x) returns the index of the first element that is equal to x

Run the following code block:

In [ ]:

```python
list1 = ['a','p','p','l','e']
print(list1.index('p'))
```

```
1
```

Python Data Structures: Range

Programming for Data Science with Python

1. Overview

In Python, ranges are the objects of the class range that has the constructor range().

Range is an immutable sequence data type/structure, i.e., its contents can not be changed after being created.

The range type:

- Represent an immutable sequence of numbers
- Is commonly used for looping a specific number of times in for loops.

### **Run the following code block:**

In [ ]:

```
range(10)
```

Out[ ]:

```
range(0, 10)
```

1.1 Properties of ranges

The advantage of the range type over a regular list or tuple:

- A ***range*** object always takes the ***same (small) amount of memory*** (no matter the size of the range it represents because it only stores the start, stop, and step values).

1.2 Constructors

> ### 1.2.1 Constructor: range(stop)
>
> ### 1.2.2 Constructor: range (start, stop, [step])

The arguments to the range constructor must be integers:

- Either built-in int or any object that implements the index special method.
- If the step argument is omitted, it defaults to 1.
- If the start argument is omitted, it defaults to 0.
- If step is zero, ValueError is raised.

For a positive step, the contents of a range r are determined by the formula:

```
- ***r[i]=start+step*I where i>=0 and r[i] < stop***
```

- Start: The value of the start parameter (or 0 if the parameter was not supplied)
- Stop: The value of the stop parameter.
- Step: The value of the step parameter (or 1 if the parameter was not supplied).

2. Examples:

Using range() in creating other sequence objects

Run the following 7 code blocks:

In [ ]:

```
list(range(10))
```

Out[ ]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [ ]:

```
list(range(1, 11))
```

Out[ ]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [ ]:

```
list(range(0, 30, 5))
```

Out[ ]:

```
[0, 5, 10, 15, 20, 25]
```

In [ ]:

```
list(range(0, 10, 3))
```

Out[ ]:

```
[0, 3, 6, 9]
```

In [ ]:

```
list(range(0, -10, -1))
```

Out[ ]:

```
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

In [ ]:

```
list(range(0))
```

Out[ ]:

```
[]
```

In [ ]:

```
list(range(1, 0))
```

Out[ ]:

```
[]
```

PRACTICE

- Create a sequence from 0 to 101 with a step parameter of 5

In [ ]:

```
#To Do add code here
list(range(0, 101, 5))
```

Out[ ]:

```
[0,
 5,
 10,
 15,
 20,
 25,
 30,
 35,
 40,
 45,
 50,
 55,
 60,
 65,
 70,
 75,
 80,
 85,
 90,
 95,
 100]
```

Python Data Structures: Strings

Programming for Data Science with Python

1. Overview

- In Python, ***strings*** are the objects of the class str that has the ***constructor str()***.
- Strings are one of the most popular data types/data structures in Python.
- We can create them simply by enclosing characters in quotes (single or double).
- Python treats single quotes the same as double-quotes.
- Creating strings is as simple as assigning a value to a variable.

  ### **Run the following 2 code blocks:**

In [ ]:

```
aStr = "Hello"
print(aStr)
```

```
Hello
```

In [ ]:

```
aStr2 = 'Hello'
print(aStr2)
```

```
Hello
```

1.1 Length of Strings

- The ***length*** of a string is the number of characters of the string.
- The ***length*** of a string can be obtained using the built-in function ***len()***.

IMPORTANT NOTES: ***len()*** is a ***built-in** function of Python, not a method of class str.

Run the following code block:

In [ ]:

```
# Declare a string
aStr = "This is a string . "
print ("The length of this string - or the number of characters: ", len(aStr))
```

```
The length of this string - or the number of characters:  19
```

1.2 String Indices

- **String is a sequence data type/structure** in Python.
- Like any other sequence data type in Python, the **indices** of a string always start with 0.
- The range of indices of a string: 0 ... len(string) - 1

  ### **Run the following 5 code blocks:**

In [ ]:

```
aStr = "This is a string . "
print("The length of this string: ", len(aStr))
```

```
The length of this string:  19
```

In [ ]:

```
print(aStr[0])
```

```
T
```

In [ ]:

```
print(aStr[1])
```

```
h
```

In [ ]:

```
print(aStr[16])
# Notice: This will return a blank space.
```

In [ ]:

```
print (aStr[17])
#Notice:  This will return a period.
```

```
.
```

2. Create Strings

2.1 Using String Literals

Run the following 4 code blocks:

In [ ]:

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)
```

```
Hello
```

In [ ]:

```
my_string = "Hello"
print(my_string)
```

```
Hello
```

In [ ]:

```
my_string = '''Hello'''
print(my_string)
```

```
Hello
```

In [ ]:

```
# triple quotes string can extend multiple Lines
my_string ="""Hello. Welcome to
Python World!"""
print(my_string)
```

```
Hello. Welcome to
Python World!
```

2.2 Create Strings from Lists - Using join() method

Run the following 2 code blocks:

In [ ]:

```
# VERSION 1: List of strings--> A string

alist = ["This", "is", "a", "string"]
print ("This is a list: ", alist)
```

```
This is a list:  ['This', 'is', 'a', 'string']
```

In [ ]:

```
aString =" " . join(alist)
# aString is a string and so is alist
print(aString)
```

```
This is a string
```

2.3 Create Strings from Lists - Using str() and join ()

Run the following 2 code blocks:

In [ ]:

```
# Version 2: List of numbers--> A string

# A List of numbers
alist = [20, 30, 40, 50, 60]

# Convert aList into a List of strings - Using the constructor str()
aStrList = [str(element) for element in alist]

print ("This is a list of strings: ", aStrList)
```

```
This is a list of strings:  ['20', '30', '40', '50', '60']
```

In [ ]:

```
# Using join() to create a new string
aString =" " . join(aStrList)

# aString = "20 30 40 50 60"
print("This is a string : ", aString)
```

```
This is a string :  20 30 40 50 60
```

2.4 Create Strings from Lists - Using map() and join()

Run the following code block:

In [ ]:

```
# Generate the combination from the list
# Then transform each element of the list into a string

from itertools import combinations
L = [1, 2, 3, 4]

print(combinations(L, 3))

# Using map() and join() to convert each numeric combination into as string
# Thanks to this technique, we can display the List of combinations

[",".join(map(str, comb)) for comb in combinations(L, 3)]
```

```
<itertools.combinations object at 0x000002326AABD6D0>
```

Out[ ]:

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

3. Access Characters in Strings

3.1 Access Single Characters

Run the following 4 code blocks:

In [ ]:

```
# Python allows negative indexing for its sequences.
# The index of -1 refers to the last item, -2 to the second to the last item, and so on.
# We can access a range of items in a string by using the slicing operator (colon).
str = 'programiz'
print('str = ', str)
```

```
str =  programiz
```

In [ ]:

```
# first character
print('str[0] = ', str[0])
```

```
str[0] =  p
```

In [ ]:

```
# Third character
print('str[0] = ', str[2])
```

```
str[0] =  o
```

In [ ]:

```
#Last character
print('str[-1] = ', str[-1])
```

```
str[-1] =  z
```

3.2 Access a Slice of Strings

Run the following 6 code blocks:

In [ ]:

```
#slicing 2nd to 5th character

str='programiz'

print('str[1:5]= ', str[1:5])
```

```
str[1:5]=  rogr
```

In [ ]:

```
#slicing 6th to 2nd Last character
print('str[5:-2] = ', str[5:-2])
```

```
str[5:-2] =  am
```

In [ ]:

```
sample_str = 'Python String'

# Print a range of character starting from index 3 to index 4
print (sample_str[3:5])
```

```
ho
```

In [ ]:

```
# Print all characters from index 7
print (sample_str[7:])
```

```
String
```

In [ ]:

```
# Print all characters before index 6
print(sample_str[:6])
```

```
Python
```

In [ ]:

```
# Print all characters from index 7 to the index -4 (count from)
print (sample_str[7:-4])
```

```
St
```

4. Modify Strings

IMPORTANT NOTES:

- ***Strings are immutable***, i.e. they cannot be changed after being created.
- Any attempt to change or modify the contents of strings will lead to errors.

Run the following code block:

In [ ]:

```
sample_str = 'Python String'
sample_str[2] = 'a'
# Do you know why you have an error in your output?
# can not change String after it is created.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-3-0fc4e724124c> in <module>
      1 sample_str = 'Python String'
----> 2 sample_str[2] = 'a'
      3 # Do you know why you have an error in your output?
      4 # can not change String after it is created.

TypeError: 'str' object does not support item assignment
```

To Do: Explain why you got an error. I received an error because ...

IMPORTANT NOTES:

Strings are immutable.

- This means that elements of a string cannot be changed once it has been assigned.
- But an existing string variable can be re-assigned with a brand new string.

Run the following 2 code blocks:

In [ ]:

```
str2 = "This is a string . "
print ("str2: ", str2)
```

```
str2:  This is a string .
```

In [ ]:

```
# Reassign a new tuple to tuple1
str2 = "This is a new string."
print("str2 after being re-assinged : ", str2)
```

```
str2 after being re-assinged :  This is a new string.
```

5. Copy Strings

5.1 Shallow copy

- Shallow copy means that only the reference to the object is copied. No new object is created.
- Assignment with an = on string does not make a copy.
- Instead, assignment makes the two variables point to the same list in memory.

5.2 Deep copy

Deep copy means that a new object will be created when the copying has done.

IMPORTANT NOTES: Strings are ***immutable sequence objects***. Strings **cannot be deep-copied***.

6. Delete Strings

To ***delete a string***, using the built-in function ***del()***.

Run the following 2 code blocks:

In [ ]:

```
sample_str = "Python is the best scripting language."
del (sample_str)
```

In [ ]:

```
# to show that the string has been deleted, Let's print it
# --> ERROR
print (sample_str)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-83-58e70779f7c9> in <module>
      1 # to show that the string has been deleted, Let's print it
      2 # --> ERROR
----> 3 print (sample_str)

NameError: name 'sample_str' is not defined
```

To Do: Explain why you got an error. I received an error because ...

7. Operations on Strings

7 .1 Concatenate Strings

Using **+** to ***concatenate*** strings

Run the following code block:

In [ ]:

```
str1 = "Hello"
str2 = "World!"
str3 = str1 + " " + str2
#using +
print(str3)
```

```
Hello World!
```

7.2 Replicate Strings

Using ***** to ***replicate*** a string

Run the following code block:

In [ ]:

```
str = "Hello "
replicatedStr = str * 3
print ("The string has been replicated three times: ", replicatedStr)
```

```
The string has been replicated three times:  Hello Hello Hello
```

In [ ]:

```
str1 = "Welcome"
print("come" in str1)
```

```
True
```

7.3 Test substrings with "in" & "not in"

Run the following 2 code blocks:

In [ ]:

```
print("come" not in str1)
```

```
False
```

7.4 Compare strings: <, >, <=, >=, ==, !=

Run the following 3 code blocks:

In [ ]:

```
# TRUE: "apple" comes before "banana"
print("apple" < "banana")
```

```
True
```

In [ ]:

```
print("apple" < "Apple")
```

```
False
```

In [ ]:

```
print("apple" == "Apple")
```

```
False
```

7.5 Iterate strings using for loops

Run the following 3 code blocks:

In [ ]:

```
aStr = "Hello"
for i in aStr:
    print(i)
```

```
H
e
l
l
o
```

In [ ]:

```
aStr = "Hello"
for i in aStr:
    print(i, end="")
```

```
Hello
```

In [ ]:

```
aStr = "Hello"
for i in aStr:
    print(i, end="\n")
```

```
H
e
l
l
o
```

7.6 Test Strings

Method Name	Method Description
isalnum()	Returns "True" if string is alpha-numeric
isalpha()	Returns "True" if string contains only alphabets
isidentifier()	Returns "True" if string is valid identifier
isupper()	Returns "True" if string is in uppercase
islower()	Returns "True" if string is in lowercase
isdigit()	Returns "True" if string only contains digits
isspace()	Returns "True" if string only contains whitespace

Run the following 7 code blocks:

In [ ]:

```
s = "welcome to python"
s. isalnum()
```

Out[ ]:

```
False
```

In [ ]:

```
"Welcome".isalpha()
```

Out[ ]:

```
True
```

In [ ]:

```
"first Number".isidentifier()
```

Out[ ]:

```
False
```

In [ ]:

```
"WELCOME".isupper()
```

Out[ ]:

```
True
```

In [ ]:

```
"Welcome".islower()
```

Out[ ]:

```
False
```

In [ ]:

```
s.islower()
```

Out[ ]:
```
True
```

In [ ]:
```python
" \t". isspace()
```

Out[ ]:
```
True
```

8. Class string

8.1 count (x)

count(x): return the number of elements of the tuple that are equal to x

Run the following code block:

In [ ]:
```python
strl = "This is a string: Hello . . . Hello Python World!"
print (strl.count("Hello"))
```

```
2
```

8.2 index (x)

index(x) returns the index of the first element that is equal to x

Run the following code block:

In [ ]:
```python
strl = "This is a string: Hello ... Hello Python World!"
print (strl.index('s'))
```

```
3
```

PRACTICE

- Print the index for 'y' in the same string

In [ ]:
```python
#To Do add your code here
strl = "This is a string: Hello ... Hello Python World!"
print (strl.index('y'))
```

```
35
```

Python Data Structures: Tuples

Programming for Data Science with Python

Overview

In Python, **tuples** are the objects of the class tuple that has the **constructor tuple()**.

Tuple is an **immutable** sequence data type/structure, i.e., its **contents cannot be changed** after being created.

Tuples work similarly to strings and lists:

- Use the len() function for the length of a tuple
- Use square brackets [] to access data, with the first element at index 0
- The range of indices: 0 .. len(a tuple) - 1

IMPORTANT NOTES:

What are the benefit of tuples?

- Tuples are **faster than lists**.
- If you know that some **data doesn't have to be changed**, you should **use tuples** instead of lists (because this protects your data against accidental changes.)
- Tuples can be used as **keys in dictionaries**, while lists can't.
- We generally use **tuple** for **heterogeneous (different) datatypes** and list for homogeneous (similar) datatypes.

1.1 Properties of Tuples

A tuple is an **immutable list**, i.e., a tuple cannot be changed in any way once it has been created.

A tuple is defined analogously to lists except that the set of elements is enclosed in parentheses instead of square brackets.

The rules for indices are the same as for lists. Once a tuple has been created, you can't add elements to a tuple or remove elements from a tuple.

IMPORTANT NOTES:

It is actually the comma which makes a tuple, not the parentheses:

- The parentheses are optional, except in the empty tuple case **OR** when they are needed to avoid syntactic ambiguity.

For example:

- f(a, b, c) is a function call with three arguments
- f((a, b, c)) is a function call with a 3-tuple as the sole argument.

  ### **Run the following code:**

In [ ]:

```python
t = ("tuples", "are", "immutable")
t[0]
```

Out[ ]:

```
'tuples'
```

1.3 Elements of Tuples

Index range of list elements

Forward index range of list elements: **0 .. len(list) – 1** Forward: starting from the 1st element

Backward index range of list elements: **-1 .. -len(list**) Backward: Starting from the last element

Run the following code:

In [ ]:

```python
t = ("tuples", "are", "immutable")
print(t[0])
print(t[-1])
print (t[-3])
print(len(t))
```

```
tuples
immutable
tuples
3
```

1.4 Constructor: tuple ([iterable])

The constructor builds a tuple whose items are the same and in the same order as iterable's items.

- Iterable may be either a sequence, a container that supports iteration, or an iterator object.
- If iterable is already a tuple, it is returned unchanged.

If no argument is given, the constructor creates a new empty tuple:().

Run the following 2 code blocks:

In [ ]:

```python
tuple = ("a","b","c")
print(tuple)
```

```
('a', 'b', 'c')
```

In [ ]:

```python
tuple_a = ([1,2,3])
print(tuple_a)
```

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

2. Create Tuples

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: a, or (a,)
- Separating items with commas: a, b, c or (a, b, c)
- Using the tuple() built-in: tuple() or tuple(iterable)

A tuple is created by placing all the items (elements) inside a parentheses(), separated by comma. The parentheses are optional but is a good practice to write it.

Run the following 4 code blocks:

In [ ]:

```
# empty tuple
# Output: ()
my_tuple = ()
print(my_tuple)
```

```
()
```

In [ ]:

```
# tuple having integers
# Output: (1, 2, 3)
my_tuple = (1, 2, 3)
print(my_tuple)
```

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

In [ ]:

```
# tuple with mixed datatypes
# Output: (1, "Hello", 3.4)
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
```

```
(1, 'Hello', 3.4)
```

In [ ]:

```
# nested tuple
# Output: ("mouse", [B, 4, 6}, (1, 2, 3))
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

```
('mouse', [8, 4, 6], (1, 2, 3))
```

2.1 Create tuples with only ONE element

Creating a tuple with one element is a bit tricky.

Placing one element within parentheses is not enough. We must add a **trailing comma** to indicate that it is in fact a tuple.

Run the following 3 code blocks:

In [ ]:

```
# only parentheses is not enough
```

```
my_tuple = ("hello")
print(type(my_tuple))
```

```
<class 'str'>
```

In [ ]:

```
# need a comma at the end

my_tuple = ("hello",)
print(type(my_tuple))
```

```
<class 'tuple'>
```

In [ ]:

```
# parentheses are optional

my_tuple = "hello",
print(type(my_tuple))
```

```
<class 'tuple'>
```

3. Access List Elements

As other sequence data types/structures, list elements can be accessed via their indices.

We can use the index operator [] to access an item in a list. **Index starts from 0**. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other than this will raise an IndexError.

The index must be an integer. We can't use float or other types, this will result into TypeError.

Nested list are accessed using nested indexing [][] that is similar to index of 2-D array elements.

3.1 Access Single Elements of Tuples

Run the following 8 code blocks:

In [ ]:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[0])
```

```
p
```

In [ ]:

```
print(my_tuple[5])
```

```
t
```

In [ ]:

```
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3])
```

```
s
```

In [ ]:

```
print(n_tuple[1][1])
```

```
4
```

In [ ]:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
len(my_tuple)
```

Out[ ]:

```
6
```

In [ ]:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[-1])
```

```
t
```

In [ ]:

```
print(my_tuple[-6])
```

```
p
```

In [ ]:

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
# Range of the indices: 0 … len(my_tuples) -1: 0 … 6
# Index must be in range
# Or you will get an ERROR: since the index is out of range
print (my_tuple[6])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-12-a99fb7cd5e81> in <module>
      3 # Index must be in range
      4 # Or you will get an ERROR: since the index is out of range
----> 5 print (my_tuple[6])

IndexError: tuple index out of range
```

TO Do: Please state why you received an error. I received an error because ...

3.2 Access a slice of Tuples

Run the following 4 code blocks:

In [ ]:

```
# elements 2nd to 4th
my_tuple =('p','r','o','g','r','a','m','i','z')

print(my_tuple[1:4])
```

```
('r', 'o', 'g')
```

In [ ]:

```
# elements beginning to 2nd
print(my_tuple[:-7])
```

```
('p', 'r')
```

In [ ]:

```
# elements 8th to end
# Output: ('i ', 'z')
print(my_tuple[7:])
```

```
('i', 'z')
```

In [ ]:

```
# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

```
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

4. Modify Tuples

4.1 All elements are immutable objects (integers, floats, strings, etc.)

IMPORTANT NOTES:

Tuples are immutable, i.e., they cannot be changed after being created. Any attempt to change or modify contents of tuples will lead to errors.

## **Run the following code block:**

In [ ]:

```
# Here you see that a tuple can not be modified; you get an error.
aTuple = ('Python', 'C', 'C++', 'Java', 'Scala')
aTuple[2] = 'Ruby'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-66a351dac12a> in <module>
      1 # Here you see that a tuple can not be modified; you get an error.
      2 aTuple = ('Python', 'C', 'C++', 'Java', 'Scala')
----> 3 aTuple[2] = 'Ruby'

TypeError: 'tuple' object does not support item assignment
```

TO Do: Please state why you received an error. I received an error because...

4.2 One or more elements are mutable objects: lists, byte arrays, etc.

Tuples are **immutable**.

- This means that elements of a tuple cannot be changed once it has been assigned.
- If the element is itself a mutable datatype, like list, its nested items can be changed.

  ### **Run the following code block:**

In [ ]:

```
my_tuple = (4, 2, 3, [6, 5])
# An item of mutable element (list) can be changed

my_tuple[3][0] = 9
print(my_tuple)
```

```
(4, 2, 3, [9, 5])
```

4.3 Tuples can be Reassigned

Tuples are immutable.

- This means that elements of a tuple cannot be changed once it has been assigned, but an existing tuple variable can be reassigned with a brand new tuple.

Run the following 2 code blocks:

In [ ]:

```
tuple_1 = (4, 2, 3, [6, 5])
print ("tuple_1: ", tuple_1)
```

```
tuple_1:  (4, 2, 3, [6, 5])
```

In [ ]:

```
# Reassign a new tuple to tuple1

my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
tuple_l = my_tuple
print("tuple_l after being reassinged: ", tuple_l)
```

```
tuple_l after being reassinged:  ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

5. Copy Tuples

5.1 Shallow Copy

Shallow copy means that only the reference to the object is copied. No new object is created.

Assignment with an = on lists does not make a copy. Instead, assignment makes the **two variables point to the same tuple** in memory.

Run the following code block:

In [ ]:

```
tuple_l = "Hello"
tuple_2 = tuple_1
# Both the tuples refer to the same object, i.e., the same id value
id(tuple_1), id(tuple_2)
```

Out[ ]:

```
(2482379472960, 2482379472960)
```

5.2 Deep copy

Deep copy means that a new object will be created when the copying has done.

IMPORTANT NOTES:

Tuples are immutable sequence objects. Tuples **cannot** be deep-copied.

5.3 Delete Tuples

To delete a string, using the built-in function del().

Run the following 2 code blocks:

In [ ]:

```
aTuple = "Python is the best scripting language."
del (aTuple)
```

In [ ]:

```
# To show that the string has been deleted, let's print it
# You see you get an ERROR
print (aTuple)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-2b9917b0b8a8> in <module>
      1 # To show that the string has been deleted, let's print it
      2 # You see you get an ERROR
----> 3 print (aTuple)

NameError: name 'aTuple' is not defined
```

To Do: Please state why you received an error. I received an error because ...

6. Operations on Tuples

6.1 Concatenate Tuples

Run the following code block:

In [ ]:

```
tuple1 = 'Hello', # comma to indicate this is a tuple; parentheses are optional
tuple2 = ' ',
tuple3 ='World!',

# using +
print('tuple1 + tuple2 + tuple3 = ', tuple1 + tuple2 + tuple3)
```

```
tuple1 + tuple2 + tuple3 =  ('Hello', ' ', 'World!')
```

6.2 Replicate tuples

Using * to replicate a tuple

Run the following code block:

In [ ]:

```
Tuple1 = "Hello",
replicatedTuple = tuple1 * 3
print (replicatedTuple)
```

```
('Hello', 'Hello', 'Hello')
```

6.3 Test elements with "in" and "not in"

Run the following 3 code blocks:

In [ ]:

```
aTuple = (2, 4, 6, "This", "is", "a", "tuple")
print (2 in aTuple)
```

```
True
```

In [ ]:

```
print ('a' in aTuple)
```

```
True
```

In [ ]:

```
print ("This is" in aTuple)
```

```
False
```

6.4 Compare Tuples: <, >, <=, >=, ==, !=

Run the following code block:

In [ ]:

```
Tuple1 = "Hello World!"
tuple2 = "hello world!"
print (tuple1 == tuple2)
```

```
False
```

6.5 Iterate a tuple using for loop

Run the following 3 code blocks:

In [ ]:

```
tuple1 = ("This", "is", 1, "book")
for i in tuple1:
   print (i)
```

```
This
is
1
book
```

In [ ]:

```
Tuple1 = ("This", "is", 1, "book")
for i in tuple1:
   print (i, end="")
```

```
Thisis1book
```

In [ ]:

```
Tuple1 = ("This", "is", 1, "book")
for i in tuple1:
    print(i, end="\n")
```

```
This
is
```

```
1
book
```

7. Class Tuple

7.1. count()

count(x) returns the number of elements of the tuple that are equal to (x)

Run the following code block:

In [ ]:

```python
my_tuple = ('a', 'p', 'p', 'l', 'e',)
# Count
print(my_tuple.count('p'))
```

```
2
```

7.2 index (x)

index(x) returns the index of the first element that is equal to (x)

Run the following code block:

In [ ]:

```python
# Index
my_tuple=('a','p','p','l','e',)
print(my_tuple.index('l'))
```

```
3
```

PRACTICE

- Instead of the word apple change the word to texas and print the index of 'x' in texas

In [ ]:

```python
#To Do add your code
my_tuple=('t', 'e', 'x', 'a', 's',)
print(my_tuple.index('x'))
```

```
2
```

You are done. Great job!