Lecture 3:

Python Programming Advanced Topics

Data Science Course with Python

Assertions

An assertion is similar to exceptions but it is used for debugging purposes.

```
temp=-10
assert(temp>=0), “Can not be colder than zero”
```

Define a function that takes one argument. Assert the argument to be positive.

```
def my_func(x):
        Assert x>0, “Error!!”
print(x)
```

Files- Access

Open File using open()

- myFile = open(“filename.txt”)
- myFile = open(“filename.txt”, ’w’)
- myFile = open(“filename.txt”,’r’)
- myFile = open(“filename.txt”,’a’)
- myFile = open(“filename.txt”, ’w+’)
- myFile = open(“filename.txt”, ’r+’)
- myFile = open(“filename.txt”,’wb’)
- myFile = open(“filename.txt”,’rb’)

Opening File in different modes

- Write mode
- Read mode
- ReadWrite Mode
- Append mode
- Binary write mode
- Binary read mode

Files- Closing

Close File using close()

```
>>> file = open(“filename.txt”, “w”)

>>> #do stuff in the file

>>> file.close()
```

Files- Closing

Closing File : using finally

```
try:
        f = open(‘filename.txt’)
        print(f.read())
except: ……………………
finally:
        f.close()                                    //so file is always closed even incase of any errors.
```

Files- Closing

Closing File : using with

```
with open(“filename.txt”) as f:
    print(f.read())
```

// file is automatically closed at then end of with statement even if exceptions occur within it

Files- Reading

Read File using read()

```
>>> file = open(“filename.txt”, “r”)

>>> content = file.read()

>>> print(content)

>>> file.close()
```

Files- Reading

Read File based on size

```
>>> file = open(“filename.txt”, “r”)

>>> content = file.read(16)

>>> print(content)

>>> file.close()
```

Files- Reading

Files are read only once

```
>>> file = open(“filename.txt”, “r”)

>>> content = file.read()

>>> print(content)

>>> print(file.read())

>>> file.close()
```

Files- Reading

Read lines in File

```
>>> file = open(“filename.txt”, “r”)

>>> content = file.readlines()

>>> print(content)

>>> file.close()
```

Files- Writing

Write into File using write()

```
>>> file = open(“newfile.txt”, “w”)                    // Existing content is deleted and new content is overwritten

>>> msg = file.write(“This is being written in file”)

>>> print(msg)                                         // Returns the number of bytes written to a file - 29

>>> file.close()

>>> file = open(“newfile.txt”, “r”)

>>> print(file.read())                                 // Returns the contents of the file

>>> file.close()
```

None - Object

None represents absence of value

```
foo = print()
If foo==None:
        print(1)
else :
        print(2)
```

o/p: 1

None Object is returned by any function that does not explicitly return anything else.

```
def some_func( ):
        print (“Hi”)
var = some_func()
print(var)
```

O/p: Hi
        None

Dictionaries

- Dictionaries is a type of data structures
- It is used to map arbitrary keys to values
- Dictionaries are indexed using square brackets [ ] .

Example

```
ages= {“Dave”: 24, “Mary”:42, “John”:58}

print(ages[“Dave”])

print(ages[“Mary”])
```

O/p:

```
24
42
```

Quiz??

```
print(ages[“Jake”])
```

```
KeyError: ‘Jake’
```

Dictionaries

- Dictionaries can be assigned to heterogeneous values

Example

```
squares = {1:1, 2:4, 3: “error”, 4:16}

squares[8]=64

print(squares)
```

Result

```
{8:64, 1:1, 2:4, 4:16}
```

Dictionaries

- get() method is used to access the index of the specified item
- If key is not found it returns another value as specified instead of showing error.

Example

```
Pairs ={1: ‘apple’, ‘orange’:[2,3,4]}

print(Pairs.get(“orange”))

print(Pairs.get(7))

print.(Pairs.get(12345, ”not in dictionary”))
```

Result

```
[2,3,4]

None

Not in dictionary
```

Tuples

- Tuple is a type of data structure.
- Unlike list, or dictionary, tuples are immutable.
- Tuple are created using parenthesis (…..)
- Tuples are faster than lists, but cannot be altered.

Example

```
words= (“spam”,”eggs”,”sausages”)

print(words[0])
```

Result

```
spam
```

Tuples

- ❖ Tuple items are accessed using [….]
- ❖ Reassigning a value in Tuple will cause a TypeError.
- ❖ Tuples can be nested within each other.

Example

```
words= (“spam”,”eggs”,”sausages”)

words[1]= “cheese”
```

Result

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

List Slices

- List Slicing involves indexing a list with two colon separated indices.
- Slicing a list returns a new list containing all the values in the old list between the indices.
- Slicing can also be done on tuples.

Squares = [0,1,4,9,16,25,36,49,64]

Result

```
print(squares[2:6])
```

[4,9,16,25]

```
print(squares[0:1])
```

[0]

List Slices

- List Slicing involves indexing a list with two colon separated indices.
- Slicing a list returns a new list containing all the values in the old list between the indices.
- Slicing can also be done on tuples.

Squares = [0,1,4,9,16,25,36,49,64,81,100]

Result

```
print(squares[ :7])
```

[0,1,4,9,16,25,36]

```
print(squares[7: ])
```

[49, 64, 81, 100]

List Slices

- List Slicing involves indexing a list with two colon separated indices.
- Slicing a list returns a new list containing all the values in the old list between the indices.
- Slicing can also be done on tuples.

Squares = [0,1,4,9,16,25,36,49,64,81,100]

```
print(squares[ : :2])

print(squares[2:8:3 ])
```

Result

```
[0,4,16,36,64,100]

[4,25]
```

List Slices

- List Slicing involves indexing a list with two colon separated indices.
- Slicing a list returns a new list containing all the values in the old list between the indices.
- Slicing can also be done on tuples.

Squares = [0,1,4,9,16,25,36,49,64,81,100]

	Result
print (squares[1:-1])	[1,4,9,16,25,36,49,64,81]
print (squares[: : -1])	[100,81,64,49,36,25,16,9,4,1,0]
print (squares[6: : -1]) =??	

List Comprehensions

- List comprehensions are used to create a simple list in one go provided all its contents obey a simple rule.

Example

```
#a list comprehension

Cubes = [i**3 for i in range(5)]

print(Cubes)
```

Result

```
[0,1,8,27,64]
```

List Comprehensions

- Using List comprehensions create a simple list with squares from 0 to 9 ifof the square of a number is even.

Example

```
#a conditional list comprehension

evens = [i**2 for i in range(10) if ((i**2)%2==0)]

print(evens)
```

Result

```
[0,4,16,36,64]
```

Converting List to String

- Using ***join*** - join a list of strings with a separator to return a string separated by the separator.

Example

```
#join

print(“!!!”.join([“Super”, “Sale”, “Register Now”]))
```

Result

```
“Super!!!Sale!!!Register Now!!!”
```

Converting String to List

- Using ***split*** - split string entities separated by a given separator to a list of strings values.

Example

```
#split

print(“Super!!!Sale!!!Register Now!!!”).split(“!!!”))
```

Result

```
[‘Super’ , ‘Sale’, ‘Register Now’]
```

All() & Any()

❖ Using ***all*** - takes a list as an argument and return True if all the values in the argument evaluate to True

Example

```
Nums =[55,44,33,22,11]

if all( [ i>5 for i in Nums] ):

    print(“All are larger than 5”)
```

Result

True

All() & Any()

- Using **any** - takes a list as an argument and return True if any of the values in the argument evaluate to True

Example

```
Nums =[55,44,33,22,11]

if any( [ i%5==0 for i in Nums] ):

      print(“At least one is multiple of 5”)
```

Result

True

Functional Programming

Anonymous functions

- Using ***lambda*** - lambda is used as a no-name-function. It gives an alternate way to create a function on the fly provided the function perform within them require a single expression.

```
def my_func(f, arg):
    return  f(arg)
y=my_func(lambda x: 2*x*x, 5)
print(y)
```

Anonymous functions - alternate approach

- Using ***lambda*** - lambda is used as a no-name-function. It gives an alternate way to create a function on the fly provided the function perform within them require a single expression.

```
double = lambda x:x*2

print(double(7))
```

O/P:

```
>>>
14
>>>
```

Map Function

- ***map*** - The function map takes a function and an iterable as arguments and returns a new iterable with the function applied to each argument.

```
def add_five(x):
        return x+5
nums = [11,22,33,44,55]
result = list(map(add_five,nums))
print(result)
```

O/P:

```
[16,27,38,49,60]
```

Map Function

- ***map*** - The function map takes a function and an iterable as arguments and returns a new iterable with the function applied to each argument.

```
nums = [11,22,33,44,55]
result = list(map(lambda x: x+5, nums))
print(result)
```

O/P:

```
[16,27,38,49,60]
```

Filter Function

❖ ***map*** - The function ***filter*** filters an iterable by removing items that do not match a condition.

```
nums = [11, 22, 33, 44, 55]
res = list(filter(lambda x: x%2==0, nums))
print(res)
```

O/P:

```
[22,44]
```

Generators

- **Generators** - are a type of iterable just like list, or tuples.
- Generators are created using yield statement inside functions.
- Generators yield only one item at a time.
- Generators allow you to declare a function that behaves like an iterator, so it can be used in a for loop directly.

```
>>>def countdown( ):
        i=5
        while i>0:
            yield i
            i = i-1
>>>for i in countdown( ):
        print(i)
Output:  5    4    3    2    1
```

Decorators

- ***Decorators*** - modify functions using other functions
- Are used to extend the functionality of existing functions that you do not want to modify.
- A single function can have multiple decorators.
- A function that is being decorated with a decorator is prepended with an @ operator followed by all the names of decorator functions that is being used to decorate the original function.

Decorators

```
def wrapdecor(display_func):
        def wrap():
            print(“|==|==|==|==|==|==|==|==|==|”)
            display_func()
            print(“|==|==|==|==|==|==|==|==|==|”)
        return wrap

@wrapdecor
def display( ):
    print(“Python is Easy!”)
```

Output:

```
|==|==|==|==|==|==|==|==|==|
Python is Easy
|==|==|==|==|==|==|==|==|==|
```

Sets

- **Sets** - are data structures
- Are created using {...} curly braces
- To create an empty set, use set().
- Sets are unordered; they cannot be indexed.
- Sets cannot contain duplicates.
- Sets are faster than list.
- Sets can be combined using union, intersection, difference, symmetric difference (returns unique elements from both sets)

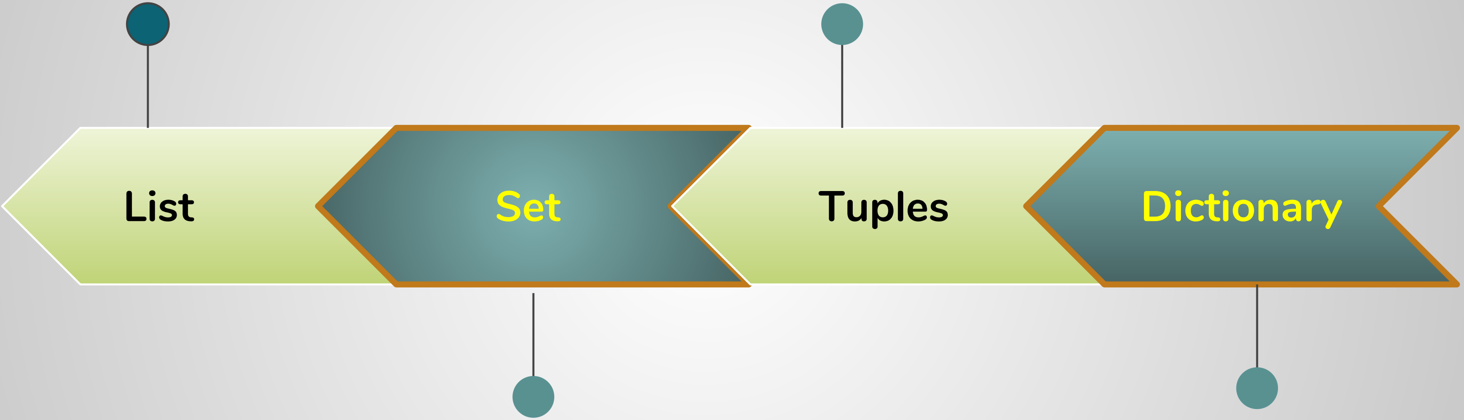
Sets

```
Nums = { 1,2,1,3,1,4,5,6,7}
print(Nums)
Nums.add(-7)
Nums.discard(3)
print(Nums)
```

```
Output:    { 1, 2 , 3 , 4 , 5 , 6 }
           {1, 2, 3, 4, 5, 6, -7}
           {1, 2, 4, 5, 6, -7}
```

Use lists if you have a collection of data that modifies frequently and does not need random access

Use tuples when you cannot change your data.



List

Set

Tuples

Dictionary

Use a set when you need uniqueness in your elements

Use dictionary when you need fast lookup based on unique keys and your data can be modified.

Quiz 1

What is the result of this code?

```
nums = {1,2,3,4,5,6}
nums= {0,1,2,3} & nums
nums= filter(lambda x:x>1, nums)
print(len(list(nums)))
```

Output: 2

Quiz 2

What is the result of this code?

```
def power(x,y):
        if y ==0:
        return 1
else:
        return x* power(x,y-1)
print(power(2,3))
```

Output: 8

Quiz 3

Fill in the blanks to calculate the expression x*(x+1) using an anonymous function and call it for the number 6.

```
a= (________x:x__())(__)
print(a)
```

Output: (lambda x: x*(x+1))(6)

Quiz 4

Fill in the blanks to leave only even numbers in the list?

```
nums = {1,2,8,3,7}
res= list(___(____x:x%___==0,nums))
print(res)
```

Output: list(*filter*(lambda x:x%2==0,nums))

Quiz 5

Write the statement to print only the items in set “a” that are not in the set “b”.

```
print(a-b)
```

Object Oriented Programming In Python

Classes

- ***Classes*** - are created using the keyword class.

```
>>>class Cat:
     def __init__(self, color, legs):
          self.color = color
          self.legs =legs

>>>felix= Cat(“ginger”, 4)
>>>rover= Cat(“White”, 4)
>>>stumpy =Cat(“brown”, 3)
>>>print(felix.color)
```

Output: ***ginger***

Classes

- ***_init_*** - are like constructors. They are the default method which is called when an object is instantiated using the class name.

```
>>>class Student:
    def __init__(self, name):
        self.name = name

>>>test =Student(“Bob”)
>>>print(test.name)
Output:    Bob
```

Methods

- ***Class methods*** - all class methods must have self as their parameter.
- To access a method, use - **object.method_name()**

```
class Dog:
    def __init__(self, name,color):
        self.name = name
        self.color=color
    def bark(self):
        print(“Fetch!”)
Bruno=Dog(“Bruno”, “black”)
print(Bruno.name)
Bruno.bark()
```

```
Output:    Bruno
           Fetch!
```

Methods

- ***Class methods*** - all class methods must have self as their parameter.
- To access a method, use - **object.method_name()**

```
class Student:
        def __init__(self, name):
                self.name = name
        def greet(self):
                print(“Hi from “+ self.name)
s1=Student(“Alex”)
s1.greet()
```

Output: ***Hi from Alex***

Inheritance

- ***Inheritance*** - when multiple subclasses share functionality from a superclass.

```
class Animal:
        def __init__(self, name,color):
                self.name = name
                self.color=color
class Cat(Animal):
        def meow(self):
                print(“Meow.....”)
class Dog(Animal):
        def bark(self):
                print(“Woof!”)
Tuffy =Dog(“Tuffy”, “white”)
print(Tuffy.color)
Tuffy.bark()
```

Output:

```
white
Woof!
```

Method Overriding

- ***Inheritance*** - if a class inherits from another class with the same attributes or methods, it overrides them.

```
class Wolf:
    def __init__(self, name,color):
        self.name = name
        self.color=color
     def bark(self):
        print(“Grrrrrr”.)
class Dog(Wolf):
    def bark(self):
        print(“Woof!”)
Husky =Dog(“Max”, “grey”)
Husky.bark()
```

Output:

```
Woof!
```

Super Function

- ***super*** - The function super refers to the parent class. It is used to access an object’s superclass.

```
class A:
    def spam(self):
        print(1)

class B(A):
    def spam(self):
        print(2)
        super().spam()
obj1=B()
obj1.spam()
```

Output:

```
2
1
```

Lifecycle of an Object

- **Creation**
  - Class Definition
  - Instantiation of an instance of class (when __init__ is called) - Memory is allocated to store the new instance

- **Manipulation**
  - Once memory is allocated, the object is ready to be used.
  - Other code can interact with the object by calling functions on the object and accessing its attributes.

- **Destruction**
  - Once it is finished being used, it is destroyed using garbage collection and allocated memory is freed.
  - Object Destruction occurs when its reference count reaches zero. Reference count is the number of variables that refer to an object.

Regular Expressions

Regular Expressions

- ***Regular Expressions are used for string manipulation.***

- ***They are mainly used to verify if strings match a specific pattern.***

  - String having the format of an email address, or a 16 digit credit card number.

- ***Performing substitutions***
  - Changing all occurrences of Gurgaon to Gurugram.

Regular Expressions

- ***Regular Expressions (re) module is part of standard library.***
- ***They are mainly used to verify if strings match a specific pattern.***

```
import re
pattern = r“Coffee”
If re.match(pattern, “CoffeeCoffeeCoffee”):
        print(“Match”)
else :
        print(“No Match”)
```

Output: ***Match***

Regular Expressions

- ***Regular Expressions (re) module is part of standard library.***
- ***They are mainly used to verify if strings match a specific pattern.***

```
import re
pattern = r“Coffee”
If re.search(pattern, “ToffeeCoffeeCoffee”):
        print(“Match”)
else :
        print(“No Match”)
```

Output: ***Match***

Regular Expressions

❖ ***Regular Expressions (re) module is part of standard library.***

❖ ***They are mainly used to verify if strings match a specific pattern.***

```
import re
pattern = r“Coffee”
print(re.findall(pattern, “ToffeeCoffeeCoffee”))
```

Output: **[‘Coffee’, ’Coffee’]**

Regular Expressions

- ***Regular Expressions (re) module is part of standard library.***
- ***They are mainly used to verify if strings match a specific pattern.***

```
import re
pattern = r“test”
match=re.search(pattern, “sometest”):
print(match.start())
print(match.end())
print(match.span())
print(match.group())
```

Output: **4** **8** **(4, 8)** **test**

Regular Expressions: sub method

- *Sub method is used to replace all occurrences of the pattern in string with a substitute string.*
- *Sub method returns the modifies string*

```
re.sub(pattern, repl,  string,  max=0)
```

Regular Expressions: sub method

- ***Sub method is used to replace all occurrences of the pattern in string with a substitute string.***
- ***Sub method returns the modifies string***

```
import re
str = “My name is Dhruv. Hi Dhruv. ”
pattern =r“Dhruv”
newstr= re.sub(pattern, “David”, str)
print(newstr)
```

Output: **My name is David. Hi David.**

Regular Expressions: sub method

- ***Sub method is used to replace all occurrences of the pattern in string with a substitute string.***
- ***Sub method returns the modifies string***

```
import re
num =”09795693229496”
pattern =r“9”
newstr= re.sub(pattern, “0”, num)
print(newstr)
```

Output: **00705603220406**

Regular Expressions: Meta characters

- The dot metacharacter - represents any (single) character

```
import re
pattern =r“gr.y”
if re.match(pattern, “grey”):
    print(“Match 1”)
if re.match(pattern, “blue”)
    print(“Match 2”)
```

Output: **Match 1**

Regular Expressions: Meta characters

- The ^ and $ metacharacter - represents start and end of a string

```
import re
pattern =r“^gr.y$”
if re.match(pattern, “grey”):
        print(“Match 1”)
if re.match(pattern, “stingray”)
        print(“Match 2”)
```

Output: **Match 1**

Regular Expressions: Meta characters

❖ The ^ and $ metacharacter - represents start and end of a string

Quiz: Fill in the blanks to create a pattern that matches strings that contain 3 characters, out of which the last character is an exclamation mark.

Output: r"..!$"

Regular Expressions: Character Classes

- The [somechars] metacharacter - represents a class of characters

```
import re
pattern =r“[aeiou]”
if re.search(pattern, “grey”):
        print(“Match 1”)
if re.search(pattern, “qwertyuiop”):
        print(“Match 2”)
if re.search(pattern, “rhythm myths”):
        print(“Match 3”)
```

Output: Match 1 Match 2

Regular Expressions: Character Classes

- The [a-z] class - represents lowercase alphabetic
- The [G-P] class - matches any uppercase character from G to P.
- The class [0-9] matches any digit.
- The class [A-za-z] matches a letter of any case.
- The class [A-Z][A-Z][0-9] matches a string that contains two uppercase letter followed by a digit.

Regular Expressions: Character Classes

```
import re
pattern =r“[A-Z][A-Z][0-9]”
if re.search(pattern, “LS8”):
        print(“Match 1”)
if re.search(pattern, “E3”):
        print(“Match 2”)
if re.search(pattern, “1ab”):
        print(“Match 3”)
```

Output: **Match 1**

Regular Expressions: Character Classes

Quiz: What would [1-5][0-9] match?

```
pattern =r“[1-5][0-9]”
if re.search(pattern, “85”):
        print(“Match 1”)
if re.search(pattern, “10”):
        print(“Match 2”)
if re.search(pattern, “59”):
        print(“Match 3”)
```

Output: **Match 2**   **Match 3**

Regular Expressions: Character Classes

Use ^ to invert a character class

```
pattern =r“[^A-Z]”
if re.search(pattern, “this is all lowercase”):
        print(“Match 1”)
if re.search(pattern, “This Is MIXed CAse AnD DigItS 899 AISo”):
        print(“Match 2”)
if re.search(pattern, “THISISALLUPPERCASE”):
        print(“Match 3”)
```

Output:

```
Match 1        Match 2
```

Regular Expressions: More Metacharacters

Use *, +, ?, { and } for repetitions.

* ⇒ Zero or more Repetitions of the Previous Character or Class

```
pattern =r”egg(spam)*”
if re.match(pattern, “egg”):
      print(“Match 1”)
if re.match(pattern, “spam”):
       print(“Match 2”)
```

Output: **Match 1**

Regular Expressions: More Metacharacters

Use *, +, ?, (and) for repetitions.

+ ⇒ One or more Repetitions of the Previous Character or Class

```
pattern =r”g+”
if re.match(pattern, “ggggg”):
      print(“Match 1”)
if re.match(pattern, “abc”):
      print(“Match 2”)
```

Output: **Match 1**

Regular Expressions: More Metacharacters

Use *, +, ?, (and) for repetitions.

? ⇒ Zero or one Repetitions of the Previous Character or Class

```
pattern =r”ice(-)?cream”
if re.match(pattern, “ice-cream”):
      print(“Match 1”)
if re.match(pattern, “sausage”):
      print(“Match 2”)
if re.match(pattern, “ice--cream”):
      print(“Match 3”)
```

Output: Match 1

Regular Expressions: More Metacharacters

Use *, +, ?, (and) for repetitions.

(x,y) ⇒ x to y Repetitions of the Previous Character or Class
By default, x =0, and y = infinity.

```
Import re
pattern =r”9?(1,3)$”
if re.match(pattern, “9”):
      print(“Match 1”)
if re.match(pattern, “999”):
      print(“Match 2”)
if re.match(pattern, “9999”):
      print(“Match 3”)
```

Output: **Match 1 Match 2**

Regular Expressions: More Metacharacters

Use *, +, ?, (and) for repetitions.

x|y⇒ matches either x or y.

```
Import re
pattern =r”gr(a|e)y”
If re.match(pattern, “gray”):
      print(“Match 1”)
if re.match(pattern, “grey”):
      print(“Match 2”)
if re.match(pattern, “griy”):
      print(“Match 3”)
```

Output: **Match 1   Match 2**

Regular Expressions: More Metacharacters

Quiz→ What would ‘([^aeious][aeious][^aeiou])+’ match?

⇒ One or more repetitions of a non vowel, a vowel and a non-vowel.

iba	**(not valid)**
cae	**(not valid)**
dah	**(valid)**

Regular Expressions: More Metacharacters

Quiz→ Which regex would match a valid phone number such that -

- Number should prefix either 0 or 91
- Number should contain 10-12 digits
- The first digit after the prefix should be 7,8 or 9.

Quiz→ Which regex would match “email@domain.com” such that -

- Email can contain any lowercase letter or digit
- Domain can be any any lowercase letter
- .com should come as it is.

Regular Expressions: More Metacharacters

Quiz→ Which regex would match “[01]+0$”?

⇒ 0101

⇒ 011101

⇒ 01011111001010

Regular Expressions: More Metacharacters

Quiz→ Which regex would match “(4{5,6})”?

⇒ 10 or 12 4s

⇒ 5 or 6 fours

⇒ 456

Summary

Variable

Stores a single value

List

Stores multiple values in ordered index

Tuple

Stores multiple fixed values in a sequence

Set

Stores multiple unique values in an unordered collection

Dictionary

Stores multiple unordered key:value pairs

Coding Assignment

Q1 What is the output of the following code?

```
re.findall('good', 'good is good')
re.findall('good', 'bad is good')
```

Q2 What is the output?

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

Further Reading

Guide To Chatbots: Journey From Past to Present

https://www.datasciencecentral.com/profiles/blogs/beginners-guide-to-chatbots