PYTHON INTERNSHIP MANUAL

Batch A - 2025 **Week#4 (Basic Python)**

Important Instructions:

- Read the manual carefully and understand every topic before starting the assignment.
- Follow the deadlines properly; marks will be deducted for late submissions.
- Avoid to use AI Tools and make your own logic.
- Completing all tasks in the assignment is compulsory to receive full marks.

- **Manual Posted Date:**

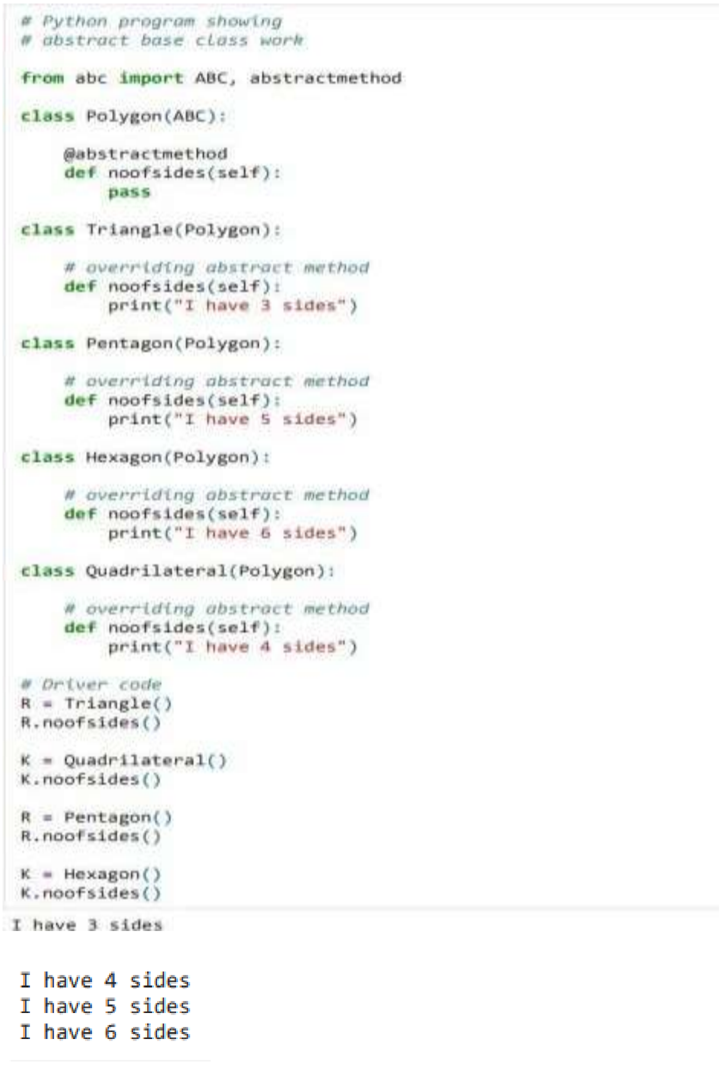
  24th March, 2025 (Monday)

- **Assignment-4 Deadline:**

  2nd April, 2025 (Wednesday 11:59pm)

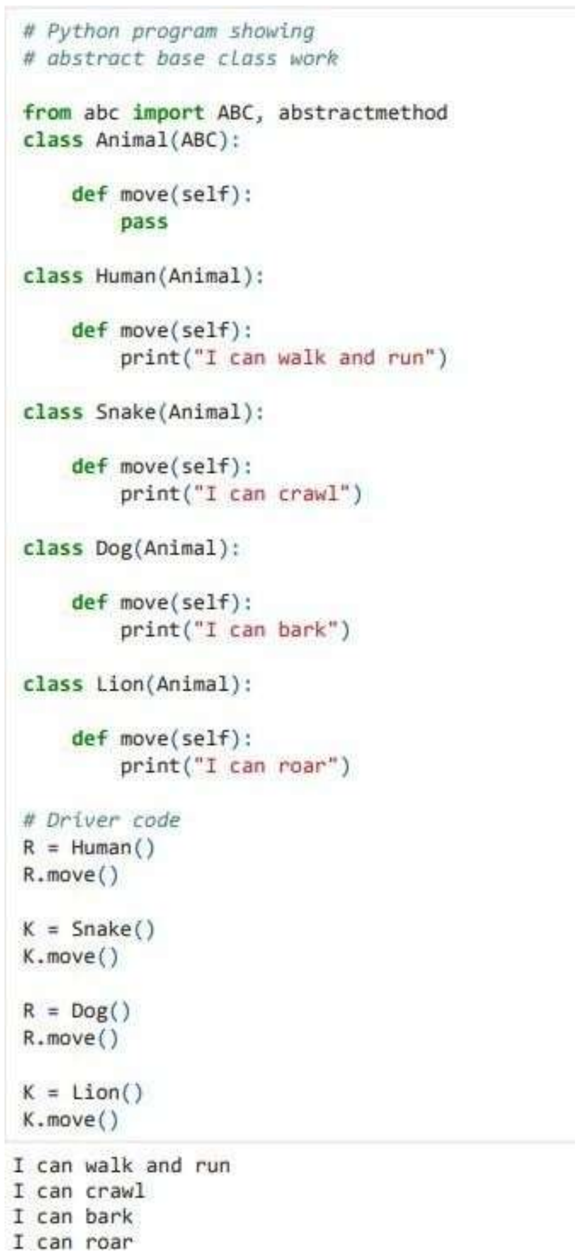
Abstraction

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC. ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.



```python
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass

class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")

# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

```
I have 3 sides

I have 4 sides
I have 5 sides
I have 6 sides
```

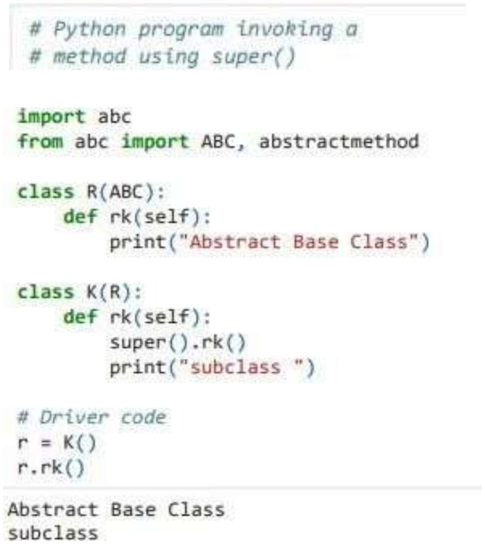


```python
# Python program showing
# abstract base class work

from abc import ABC, abstractmethod
class Animal(ABC):

    def move(self):
        pass

class Human(Animal):

    def move(self):
        print("I can walk and run")

class Snake(Animal):

    def move(self):
        print("I can crawl")

class Dog(Animal):

    def move(self):
        print("I can bark")

class Lion(Animal):

    def move(self):
        print("I can roar")

# Driver code
R = Human()
R.move()

K = Snake()
K.move()

R = Dog()
R.move()

K = Lion()
K.move()
```

```
I can walk and run
I can crawl
I can bark
I can roar
```

Concrete Methods in Abstract Base Class

Concrete classes contain only concrete (normal)methods whereas abstract classes may contain both concrete methods and abstract methods. The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super().

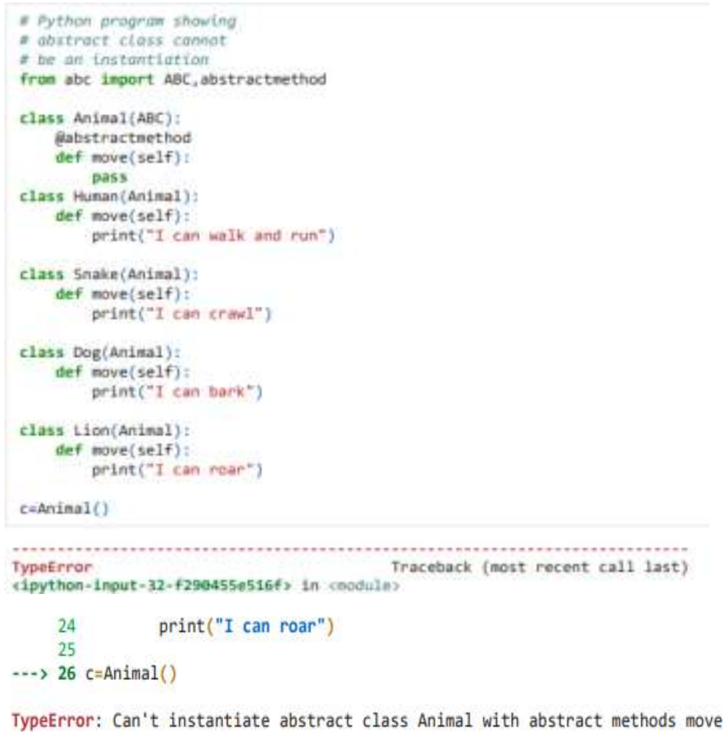


```
# Python program invoking a
# method using super()

import abc
from abc import ABC, abstractmethod

class R(ABC):
    def rk(self):
        print("Abstract Base Class")

class K(R):
    def rk(self):
        super().rk()
        print("subclass ")

# Driver code
r = K()
r.rk()
```

```
Abstract Base Class
subclass
```

Abstract Class Instantiation

Abstract classes are incomplete because they have methods that have nobody. If python allows creating an object for abstract classes then using that object if anyone calls the abstract method, but there is no actual implementation to invoke. So we use an abstract class as a template and according to the need, we extend it and build on it before we can use it. Due to the fact, an abstract class is not a concrete class, it cannot be instantiated. When we create an object for the abstract class it raises an error.



```
# Python program showing
# abstract class cannot
# be an instantiation
from abc import ABC,abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass
class Human(Animal):
    def move(self):
        print("I can walk and run")

class Snake(Animal):
    def move(self):
        print("I can crawl")

class Dog(Animal):
    def move(self):
        print("I can bark")

class Lion(Animal):
    def move(self):
        print("I can roar")

c=Animal()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-f290455e516f> in <module>

     24         print("I can roar")
     25
---> 26 c=Animal()

TypeError: Can't instantiate abstract class Animal with abstract methods move
```

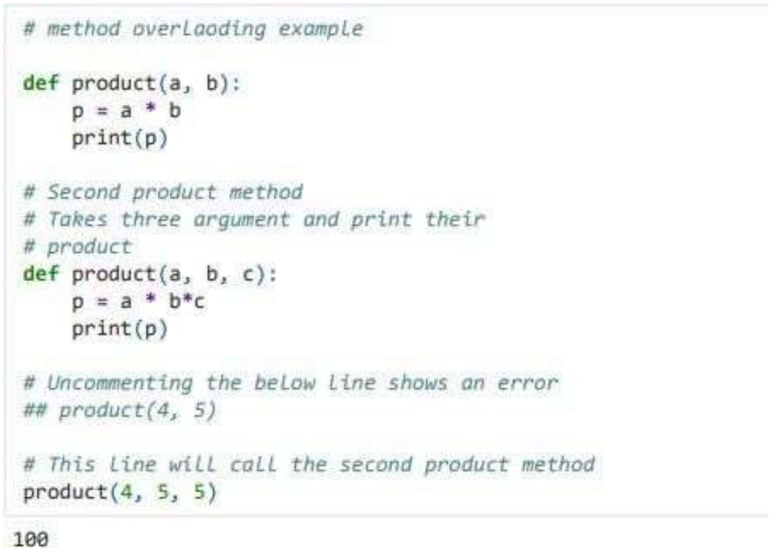
Polymorphism

Polymorphism means having many or different forms. In the programming world, Polymorphism refers to the ability of the function with the same name to carry different functionality altogether. It creates a structure that can use many forms of objects.

-> A Boy behave like a Student in a School
-> A Boy behave like a Customer in Market or Shopping Mall
-> A Boy behave like a Passenger in a Bus
-> A Boy behave like a Son in Home

❖ Method Overloading or Compile Time Polymorphism

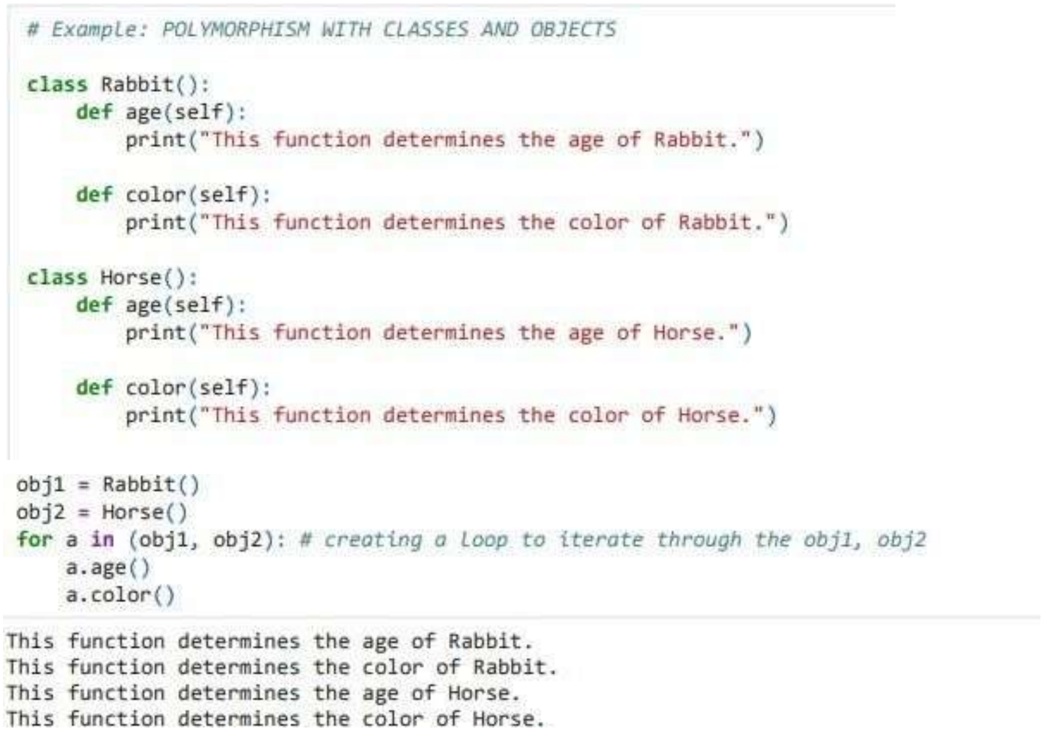
Unlike many other popular object-oriented programming languages such as Java, Python doesn’t support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.



```
# method overlaoding example

def product(a, b):
    p = a * b
    print(p)

# Second product method
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
## product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

```
100
```

❖ Polymorphism with Classes and Objects



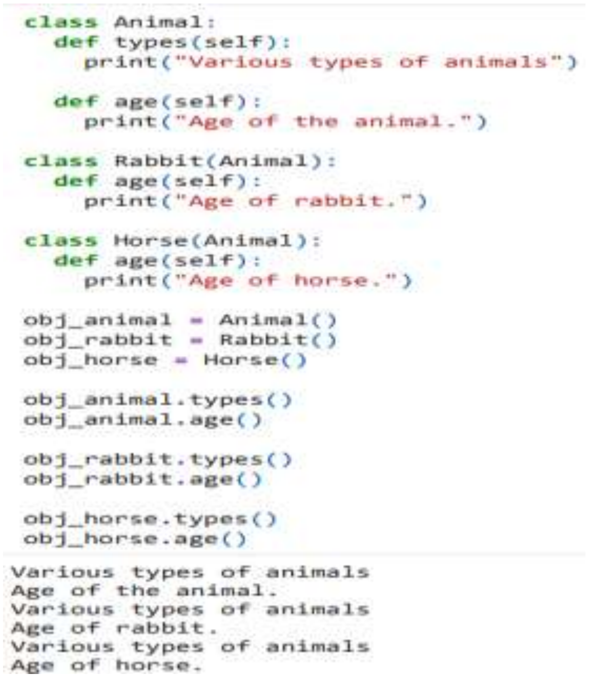
```
# Example: POLYMORPHISM WITH CLASSES AND OBJECTS

class Rabbit():
    def age(self):
        print("This function determines the age of Rabbit.")

    def color(self):
        print("This function determines the color of Rabbit.")

class Horse():
    def age(self):
        print("This function determines the age of Horse.")

    def color(self):
        print("This function determines the color of Horse.")
```

```
obj1 = Rabbit()
obj2 = Horse()
for a in (obj1, obj2): # creating a loop to iterate through the obj1, obj2
    a.age()
    a.color()
```

```
This function determines the age of Rabbit.
This function determines the color of Rabbit.
This function determines the age of Horse.
This function determines the color of Horse.
```

❖ Method Overriding or Polymorphism with Inheritance

We will be defining functions in the derived class that has the same name as the functions in the base class. Here, we re-implement the functions in the derived class. The phenomenon of reimplementing a function in the derived class is known as Method Overriding.



```
class Animal:
  def types(self):
    print("Various types of animals")

  def age(self):
    print("Age of the animal.")

class Rabbit(Animal):
  def age(self):
    print("Age of rabbit.")

class Horse(Animal):
  def age(self):
    print("Age of horse.")

obj_animal = Animal()
obj_rabbit = Rabbit()
obj_horse = Horse()

obj_animal.types()
obj_animal.age()

obj_rabbit.types()
obj_rabbit.age()

obj_horse.types()
obj_horse.age()
```

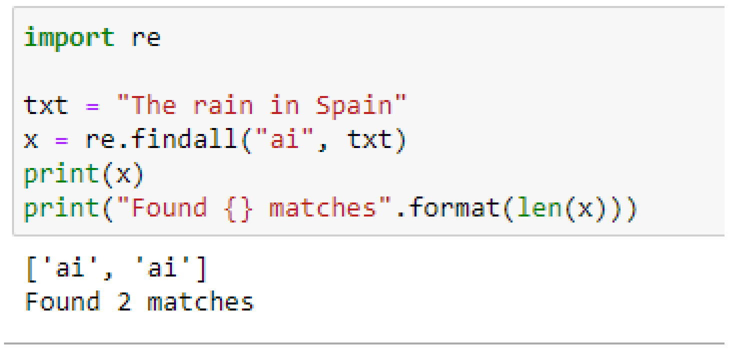
```
Various types of animals
Age of the animal.
Various types of animals
Age of rabbit.
Various types of animals
Age of horse.
```

Regular Expressions (RegEx)

A Regular Expression (RegEx) is a sequence of characters that defines a search pattern. For example, RegEx can be used to check if a string contains the specified search pattern. Python has a built-in package called **re**, which can be used to work with Regular Expressions.

RegEx Functions

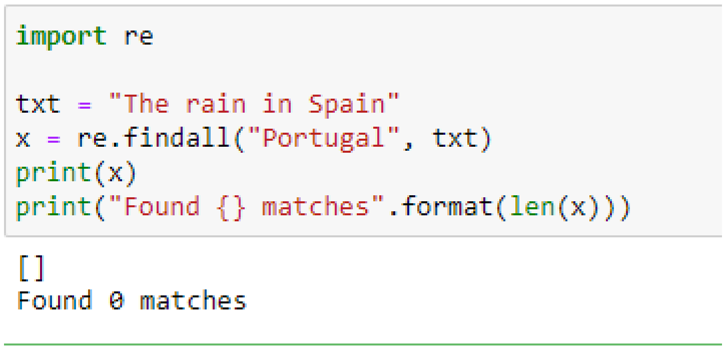
1) **findall:** Returns a list containing all matches:



```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
print("Found {} matches".format(len(x)))
```

```
['ai', 'ai']
Found 2 matches
```

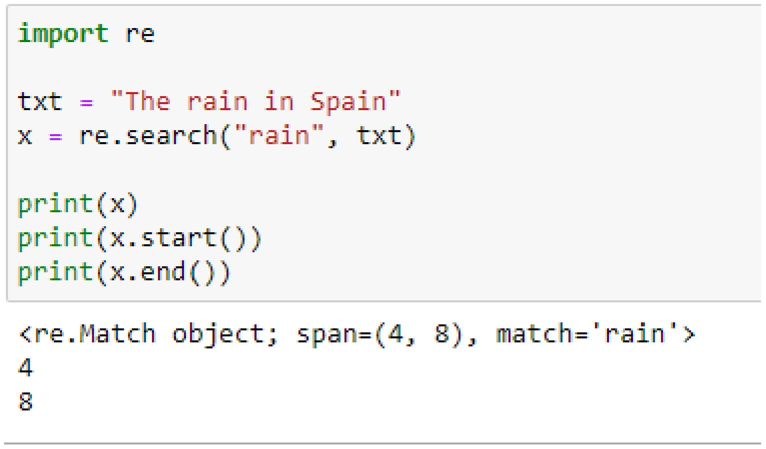
Returns an empty list if no match was found:



```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
print("Found {} matches".format(len(x)))
```

```
[]
Found 0 matches
```

2) **search:** Returns a Match object if there is a match anywhere in the string:



```
import re

txt = "The rain in Spain"
x = re.search("rain", txt)

print(x)
print(x.start())
print(x.end())
```

```
<re.Match object; span=(4, 8), match='rain'>
4
8
```

If no matches are found, the value **None** is returned:

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

```
None
```

3) split: Returns a list where the string has been split at each match.

```
import re

txt = "The rain in Spain"
x = re.split(" ", txt)
print(x)
```

```
['The', 'rain', 'in', 'Spain']
```

You can control the number of occurrences by specifying the maxsplit parameter:

```
import re

txt = "The rain in Spain"
x = re.split(" ", txt, 2)
print(x)
```

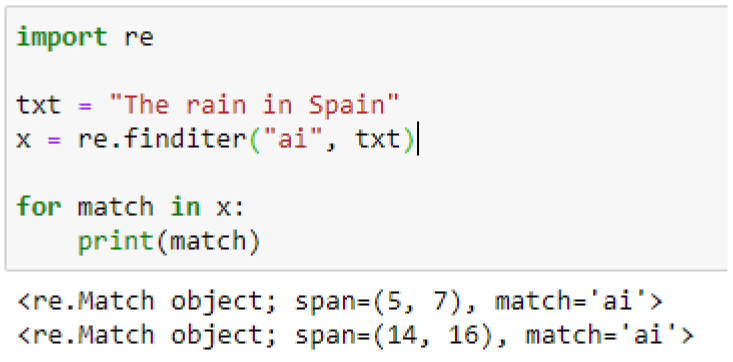
```
['The', 'rain', 'in Spain']
```

4) sub: Replaces one or many matches with a string.

```
import re

txt = "The rain in Spain"
x = re.sub("i", "j", txt)
print(x)
```

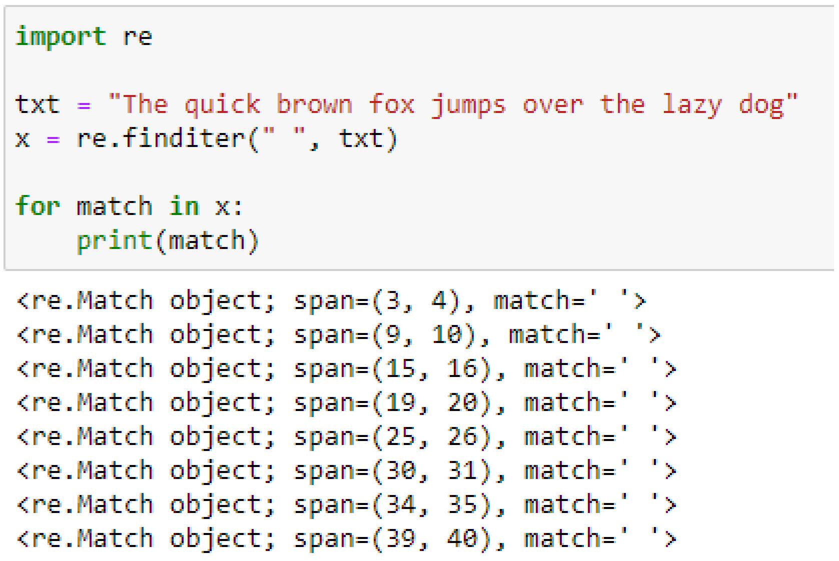
```
The rajn jn Spajn
```

5) **finditer:** returns an iterator that yields Match objects for all non-overlapping matches found anywhere in the string.



```
import re

txt = "The rain in Spain"
x = re.finditer("ai", txt)

for match in x:
    print(match)
```

```
<re.Match object; span=(5, 7), match='ai'>
<re.Match object; span=(14, 16), match='ai'>
```



```
import re

txt = "The quick brown fox jumps over the lazy dog"
x = re.finditer(" ", txt)

for match in x:
    print(match)
```

```
<re.Match object; span=(3, 4), match=' '>
<re.Match object; span=(9, 10), match=' '>
<re.Match object; span=(15, 16), match=' '>
<re.Match object; span=(19, 20), match=' '>
<re.Match object; span=(25, 26), match=' '>
<re.Match object; span=(30, 31), match=' '>
<re.Match object; span=(34, 35), match=' '>
<re.Match object; span=(39, 40), match=' '>
```

*Match Object

A Match Object is an object containing information about the search and the result. If there is nomatch, the value **None** will be returned, instead of the Match Object.
The Match object has properties and methods used to retrieve information about the search, andthe result:

- **.span():** Returns a tuple containing the start-, and end positions of the match.
- **.string:** Returns the string passed into the function.
- **.group():** Returns the part of the string where there was a match.