Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

```
l = [1,2,3,4]
s = str(l)
print(s[2])
```

```
,
```

In Python we have mainly two different types of loops :

- for loop : In the context of most data science work, Python for loops are used to loop through an iterable object (like a list, tuple, set, etc.) and perform the same action for each entry. For example, a for loop would allow us to iterate through a list, performing the same action on each item in the list.
- while loop : The while loop is somewhat similar to an if statement, it executes the code inside, if the condition is True. However, as opposed to the if statement, the while loop continues to execute the code repeatedly as long as the condition is True.

for loops

A `for` loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Iterable is an object, which one can iterate over. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

Let's go ahead and work through several example of `for` loops using a variety of data object types. We'll start simple and build more complexity later on.

Example 1

Let us print each element of our list of strings using a for loop statement

```
# Consider a list of strings

got_houses = ['Stark', 'Arryn', 'Baratheon', 'Tully', 'Greyjoy', 'Lannister', 'Tyrell', 'Martell', 'Targaryen']
```

```
# A simple for loop to print the houses of GOT universe
for house in got_houses[::-1]:
  print(f"House {house}")
```

```
House Targaryen
House Martell
House Tyrell
House Lannister
House Greyjoy
House Tully
House Baratheon
House Arryn
House Stark
```

Another interesting way to loop through the elements of a list is to use the `enumerate()` function. Using enumerate requires us two iterators index and element

```
got_houses
```

```
['Stark',
 'Arryn',
 'Baratheon',
 'Tully',
 'Greyjoy',
 'Lannister',
 'Tyrell',
 'Martell',
 'Targaryen']
```

```
# Using enumerate function to loop through the elements of a list
for number,house in enumerate(got_houses):
  print(f"The house no of house {house} is {number + 1}")
```

```
The house no of house Stark is 1
The house no of house Arryn is 2
The house no of house Baratheon is 3
The house no of house Tully is 4
The house no of house Greyjoy is 5
The house no of house Lannister is 6
The house no of house Tyrell is 7
The house no of house Martell is 8
The house no of house Targaryen is 9
```

```
x = 1,2
```

```
x
```

```
(1, 2)
```

```
list(enumerate(got_houses))
```

```
[(0, 'Stark'),
 (1, 'Arryn'),
 (2, 'Baratheon'),
 (3, 'Tully'),
 (4, 'Greyjoy'),
 (5, 'Lannister'),
 (6, 'Tyrell'),
 (7, 'Martell'),
 (8, 'Targaryen')]
```

Example 2

Suppose you are given a list of numbers. You need to find the corresponding squares of these numbers and zip them together in a dictionary

```
# The list of numbers
list_of_numbers = [1, 2, 4, 6, 11, 14, 17, 20]
```

```
# Let us first print the squares
for number in list_of_numbers:
  squared_number = number**2
  print(f"The square of {number} is {squared_number}")
```

```
The square of 1 is 1
The square of 2 is 4
The square of 4 is 16
The square of 6 is 36
The square of 11 is 121
The square of 14 is 196
The square of 17 is 289
The square of 20 is 400
```

So this was a pretty straight forward way to print out these squares.

Example 3

Imagine a scenario where we not only needed to print these numbers for each iteration but also we need to store these elements somewhere else

```
list_of_numbers
```

```
[1, 2, 4, 6, 11, 14, 17, 20]
```

```
# Let us first initialize a list where we will be appending the squares in each iteration

squared_numbers = []

for number in list_of_numbers:
  square = number**2
  # Use the append method to add the numbers one by one to our list
  squared_numbers.append(square)
  # print(squared_numbers)

print(f"The list of squared numbers is {squared_numbers}")
```

```
The list of squared numbers is [1, 4, 16, 36, 121, 196, 289, 400]
```

```
print(list_of_numbers)
print(squared_numbers)
```

```
[1, 2, 4, 6, 11, 14, 17, 20]
[1, 4, 16, 36, 121, 196, 289, 400]
```

```
# Let us zip the original numbers and squares together and get the dictionary
zipped_dict = dict(zip(list_of_numbers,squared_numbers))

# Let us print the dictionary where the key is the number and the value is the square of that number
print(zipped_dict)
```

```
{1: 1, 2: 4, 4: 16, 6: 36, 11: 121, 14: 196, 17: 289, 20: 400}
```

Example 4

Now suppose we only want to print the squares of those numbers which are even. Let us see how we can do this

```
print(list_of_numbers)
```

```
[1, 2, 4, 6, 11, 14, 17, 20]
```

```
# Let us first print the squares
for number in list_of_numbers:
  if number%2 == 0:
    squared_number = number**2
    print(f"The square of {number} is {squared_number}")
  else:
    print(f"I am an odd number {number}.My master prohibits me from printing the squares of odd numbers. Strange but okay!")

# print("\n")
# print("I am finished with my iteration")
```

```
I am an odd number 1.My master prohibits me from printing the squares of odd numbers. Strange but okay!
The square of 2 is 4
The square of 4 is 16
The square of 6 is 36
I am an odd number 11.My master prohibits me from printing the squares of odd numbers. Strange but okay!
The square of 14 is 196
I am an odd number 17.My master prohibits me from printing the squares of odd numbers. Strange but okay!
The square of 20 is 400
```

`range()` function is a pretty useful function to get a sequence of numbers. It takes three arguments : start, stop, step

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

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

```
# Let us first print the squares
for number in range(0,10,2):
  squared_number = number**2
  print(f"The square of {number} is {squared_number}")
```

```
The square of 0 is 0
The square of 2 is 4
The square of 4 is 16
The square of 6 is 36
The square of 8 is 64
```

```
for i in range(2,21,2):
  print(i)
```

```
2
4
6
8
10
12
14
16
18
20
```

Example 5

Next suppose we wanted to find the sum of the squares of our numbers in the list

```
# Print our list
print(list_of_numbers)
```

```
[1, 2, 4, 6, 11, 14, 17, 20]
```

```
# Let us the initialize the sum of the squares with 0. This makes sense right!
sum_squares = 0

for number in range(1,11):
  sum_squares = sum_squares + number
  print(sum_squares)

# Now we have added the squares of all the numbers in our list
print(f"The sum of the squares of our list of numbers is {sum_squares}")
```

```
1
3
6
10
15
21
28
36
45
55
The sum of the squares of our list of numbers is 55
```

Up till now, we have only implemented for loops for list objects. From the last week, we know that there are other objects in Python which are sequence of elements such as strings, tuples etc. Let us try and apply for loop to iterate over their elements.

Example 6

We are given a sentence : "I am the one who knocks!"

```
# Let us first store the sentence in a string variable
heisenberg_quote =  "I am the one who knocks!"
```

The next step is to print all the characters which are separated by whitespace

```
for char in heisenberg_quote:
  print(f" The character is {char}")
```

```
 The character is I
 The character is
 The character is a
 The character is m
 The character is
 The character is t
 The character is h
 The character is e
 The character is
 The character is o
 The character is n
 The character is e
 The character is
 The character is w
 The character is h
 The character is o
 The character is
 The character is k
 The character is n
 The character is o
 The character is c
 The character is k
 The character is s
 The character is !
```

```
# Lets apply a for loop to do the above task

for index,char  in enumerate(heisenberg_quote):
  print(f" The index is {index} and the character is {char}")
```

```
 The index is 0 and the character is I
 The index is 1 and the character is
 The index is 2 and the character is a
 The index is 3 and the character is m
 The index is 4 and the character is
 The index is 5 and the character is t
 The index is 6 and the character is h
 The index is 7 and the character is e
 The index is 8 and the character is
 The index is 9 and the character is o
 The index is 10 and the character is n
 The index is 11 and the character is e
 The index is 12 and the character is
 The index is 13 and the character is w
 The index is 14 and the character is h
 The index is 15 and the character is o
 The index is 16 and the character is
 The index is 17 and the character is k
 The index is 18 and the character is n
 The index is 19 and the character is o
 The index is 20 and the character is c
 The index is 21 and the character is k
 The index is 22 and the character is s
 The index is 23 and the character is !
```

We saw how we can iterate through each element of our string. What if you wanted to iterate through each word of the sentence and not each element. There is an important method available with strings known as `split()` method.

This method returns the list of words separated by the character we want to separate them by.

Example 7

```
heisenberg_quote = "It ceases to exist without me. No, you clearly don't know who you're talking to, so let me clue you in. I am not in dang

words_by_walter = heisenberg_quote.split(' ')
print(words_by_walter)
```

```
['It', 'ceases', 'to', 'exist', 'without', 'me.', 'No,', 'you', 'clearly', "don't", 'know', 'who', "you're", 'talking', 'to,', 'so', 'le
```

```
# Now we can print each word by iterating through this list
for word in words_by_walter:
  if word not in ['I','you']:
    print(f"The word is {word}")
```

```
The word is It
The word is ceases
The word is to
The word is exist
The word is without
The word is me.
The word is No,
The word is clearly
The word is don't
The word is know
The word is who
The word is you're
The word is talking
The word is to,
The word is so
The word is let
The word is me
The word is clue
The word is in.
The word is am
The word is not
The word is in
The word is danger,
The word is Skyler.
The word is am
The word is the
The word is danger.
```

- We can also iterate through each element of tuple as well.

Example 8

```
# Suppose we have a tuple of days
days = ('Monday' ,'Tuesday' ,'Wednesday' ,'Thursday' ,'Friday','Saturday','Sunday')

for day in days:
  print(f"Today is {day}")
```

```
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
Today is Saturday
Today is Sunday
```

Tuples have a special quality when it comes to `for` loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the `for` loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

- Remember from earlier we had a list of tuples of country-city

Example 9

```
country_city_river_list = [('India','New Delhi','Ganga'),('Australia','Canberra','Rovers'),('United States','Washington DC','Missouri'),('En


# Let us iterate through each tuple element of this list and unpack each item

for country,city,river in country_city_river_list:
  # print(country_city)
  print(f"The capital of the country {country} is {city} and it also has the river {river}.")
```

```
The capital of the country India is New Delhi and it also has the river Ganga.
The capital of the country Australia is Canberra and it also has the river Rovers.
```

```
The capital of the country United States is Washington DC and it also has the river Missouri.
The capital of the country England is London and it also has the river Thames.
```

- Finally we can iterate through the items of a dictionary as well.

Example 10

- This just prints the keys of the dictionary

Next we can iterate through the key-value pairs using the `items()` method which you should remember from earlier

Since the `items()` method supports iteration, we can perform *dictionary unpacking* to separate keys and values

```
country_city_dict = {'India':'New Delhi', 'Australia':'Canberra','United States':'Washington DC','England':'London'}


for country,city in country_city_dict.items():
  print(f"The capital of {country} is {city}")
```

```
The capital of India is New Delhi
The capital of Australia is Canberra
The capital of United States is Washington DC
The capital of England is London
```

```
for elem in {1,2,3}:
  print(elem)
```

```
1
2
3
```

And of course you can iterate through the list of keys and list of values independently depending upon the scenario

While Loops

The `while` statement in Python is one of the most general ways to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
 while test:
     code statements
 else:
     final code statements
```

Let's look at a few simple `while` loops in action.

```
x = 0

while x < 10:
  print('x is currently: ',x)
  print('x is still less than 10, adding 1 to x')
  x=x+1
```

```
x is currently:  0
x is still less than 10, adding 1 to x
x is currently:  1
x is still less than 10, adding 1 to x
x is currently:  2
x is still less than 10, adding 1 to x
x is currently:  3
x is still less than 10, adding 1 to x
x is currently:  4
```

```
    x is still less than 10, adding 1 to x
    x is currently:  5
    x is still less than 10, adding 1 to x
    x is currently:  6
    x is still less than 10, adding 1 to x
    x is currently:  7
    x is still less than 10, adding 1 to x
    x is currently:  8
    x is still less than 10, adding 1 to x
    x is currently:  9
    x is still less than 10, adding 1 to x
```

Notice how many times the print statements occurred and how the `while` loop kept going until the False condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an `else` statement:

```
x = 0

while x < 5:
  print('x is currently: ',x)
  print(' x is still less than 5, adding 1 to x')
  x+=1

print("All done")
print("I am done with the iterations")
```

```
x is currently:  0
 x is still less than 5, adding 1 to x
x is currently:  1
 x is still less than 5, adding 1 to x
x is currently:  2
 x is still less than 5, adding 1 to x
x is currently:  3
 x is still less than 5, adding 1 to x
x is currently:  4
 x is still less than 5, adding 1 to x
All done
I am done with the iterations
```

break, continue, pass

We can use `break`, `continue`, and `pass` statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

`break` and `continue` statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an `if` statement to perform an action based on some condition.

```
x = 0

while x < 10:
  print('x is currently: ',x)
  print(' x is still less than 10, adding 1 to x')
```

```
  # x+=1
  if x==3:
    print('x==3')
    break
  else:
    print('continuing...')
    continue
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

```
x = 0

while x < 10:
  print('x is currently: ',x)
  print(' x is still less than 10, adding 1 to x')
  x+=1
  break

else:
  print('continuing...')
```

```
x = 0

while x < 10:
  print('x is currently: ',x)
  print(' x is still less than 10, adding 1 to x')
  x+=1
  pass

else:
  print('continuing...')
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

After these brief but simple examples, you should feel comfortable using `while` statements in your code.

A word of caution however! It is possible to create an infinitely running loop with `while` statements.

```
for elem in [2,1]:
  while elem > 3:
    pass
  else:
    print(elem)
```

```
2
1
```

```
l = []
for elem in l :
```

```
  print(elem)
```

Start coding or generate with AI.