😊 Introduction to Strings Indexing and Slicing

1. In this lab, we will look at the strings in Python. So, open a new notebook or you can use the notebook you got from GitHub just clear the outputs of all the cells so that you get a better understanding of running the notebook.
2. Let's understand what strings are, Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).
3. We can create a string either in single quotes or double quotes as you can see below in the snapshot.

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

```
[1]: # Single word
     'hello'
```

```
[1]: 'hello'
```

```
[2]: # Entire phrase
     'This is also a string'
```

```
[2]: 'This is also a string'
```

```
[3]: # We can also use double quote
     "String built with double quotes"
```

```
[3]: 'String built with double quotes'
```

4. As you can see in the example shown below you should be careful with quotes as well. In cell number 4 we get an error because it thought that our sentence has ended on the first word itself because of the single quote. Although we have used the single quote in the end it still finds that in the first word, the sentence has ended.
5. So, to resolve this error we can use double quotes instead of single.

```
[4]: # Be careful with quotes!
     ' I'm using single quotes, but this will create an error'
```

```
  File "<ipython-input-4-da9a34b3dc31>", line 2
    ' I'm using single quotes, but this will create an error'
       ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in `I'm` stopped the string. You can use combinations of double and single quotes to get the complete statement.

```
[5]: "Now I'm ready to use the single quotes inside a string!"
```

```
[5]: "Now I'm ready to use the single quotes inside a string!"
```

6. We can directly print a string in cells by writing them in quotes and running the cells but if there are two strings then only the last string will be printed.
7. To do so, we can use a print statement to print a string.

Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

```
[6]: # We can simply declare a string
     'Hello World'
```

```
[6]: 'Hello World'
```

```
[7]: # Note that we can't output multiple strings this way
     'Hello World 1'
     'Hello World 2'
```

```
[7]: 'Hello World 2'
```

We can use a print statement to print a string.

```
[8]: print('Hello World 1')
     print('Hello World 2')
     print('Use \n to print a new line')
     print('\n')
     print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line
```

8. This is the built-in length function which tells us the length of the string as you can see below. It also counts the spaces between the strings.

String Basics

We can also use a function called len() to check the length of a string!

```
[9]: len('Hello World')
```

```
[9]: 11
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

9. Now we will look at the string indexing. To use indexing, we use large brackets after an object to call its index.
10. First, we will define a string as you can see below in the snapshot.

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets `[]` after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called `s` and then walk through a few examples of indexing.

```
[10]: # Assign s as a string
      s = 'Hello World'
```

```
[11]: #Check
      s
```

```
[11]: 'Hello World'
```

```
[12]: # Print the object
      print(s)
```

```
Hello World
```

11. Below you can see that this is how indexing is done. We have defined our string to a variable and by using the brackets and then writing the number we can get the letter from our string.
12. So, the length starts with 0 up to the end of the string. It can go positively or negatively.

```
[13]: # Show first element (in this case a letter)
      s[0]

[13]: 'H'

[14]: s[1]

[14]: 'e'

[15]: s[2]

[15]: 'l'
```

13. As you can see below in the snapshot we can use the (colon) **:** to perform slicing that grabs everything up to a designated point.

We can use a `:` to perform *slicing* which grabs everything up to a designated point. For example:

```
[16]: # Grab everything past the first term all the way to the length of s which is len(s)
      s[1:]

[16]: 'ello World'

[17]: # Note that there is no change to the original s
      s

[17]: 'Hello World'

[18]: # Grab everything UP TO the 3rd index
      s[:3]

[18]: 'Hel'

[19]: #Everything
      s[:]

[19]: 'Hello World'
```

14. Below you can see that we used the negative indexing to go backwards.

We can also use negative indexing to go backwards.

```
[20]: # Last letter (one index behind 0 so it loops back around)
      s[-1]
```

```
[20]: 'd'
```

```
[21]: # Grab everything but the last letter
      s[:-1]
```

```
[21]: 'Hello Worl'
```

15. We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance, we can use two colons in a row and then a number specifying the frequency to grab elements.

```
[22]: # Grab everything, but go in steps size of 1
      s[::1]
```

```
[22]: 'Hello World'
```

```
[23]: # Grab everything, but go in step sizes of 2
      s[::2]
```

```
[23]: 'HloWrd'
```

```
[24]: # We can use this to print a string backwards
      s[::-1]
```

```
[24]: 'dlroW olleH'
```

16. It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it cannot be changed or replaced.
17. As you can see below we tried to change the letter H from Hello to X but it instantly gave us an error.

[25]:
```
s
```

[25]:
```
'Hello World'
```

[26]:
```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-976942677f11> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

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

Notice how the error tells us directly what we can't do, change the item assignment!

18. So, what we can do? We can concatenate the strings.

[27]:
```
s
```

[27]:
```
'Hello World'
```

[28]:
```
# Concatenate strings!
s + ' concatenate me!'
```

[28]:
```
'Hello World concatenate me!'
```

[29]:
```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

•••

[30]:
```
print(s)
```

```
Hello World concatenate me!
```

[31]:
```
s
```

[31]:
```
'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

[32]:
```
letter = 'z'
```

•••

[33]:
```
letter*10
```

[33]:
```
'zzzzzzzzzz'
```

19. Objects in Python usually have built-in methods. These methods are functions inside the object that can perform actions or commands on the object itself.
20. We call methods with a period and then the method name. Methods are in the form: **object.method(parameters)**
21. Where parameters are extra arguments, we can pass them into the method.

Here are some examples of built-in methods in strings:

```
[34]: s
```

```
[34]: 'Hello World concatenate me!'
```

```
[35]: # Upper Case a string
      s.upper()
```

```
[35]: 'HELLO WORLD CONCATENATE ME!'
```

```
[36]: # Lower case
      s.lower()
```

```
[36]: 'hello world concatenate me!'
```

```
[37]: # Split a string by blank space (this is the default)
      s.split()
```

```
[37]: ['Hello', 'World', 'concatenate', 'me!']
```

```
[38]: # Split by a specific element (doesn't include the element that was split on)
      s.split('W')
```

```
[38]: ['Hello ', 'orld concatenate me!']
```