# CERTIFIED ASSOCIATE IN PYTHON PROGRAMMING

BY: IMRAN

# Strings

- Python's are **immutable sequences**.

```
# Example 1
word = 'by'
print(len(word))

# Example 2
empty = ''
print(len(empty))

# Example 3
i_am = 'I\'m'
print(len(i_am))
```

# Multiline strings

- multiline = '''Line #1
- Line #2'''

- print(len(multiline))


# Operations on strings

- In general, strings can be:
  - **concatenated** (joined)
  - **replicated**.
- The first operation is performed by the + operator (note: it's not an addition) while the second by the * operator (note again: it's not a multiplication).
- The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called **overloading** (as such an operator is overloaded with different duties).

# Operations on strings

- str1 = 'a'
- str2 = 'b'

- print(str1 + str2)
- print(str2 + str1)
- print(5 * 'a')
- print('b' * 4)


# Operations on strings

- `ord()`
- If you want **to know a specific character's ASCII/UNICODE code point value**, you can use a function named ord() (as in *ordinal*).
- The function needs a **one-character string as its argument** - breaching this requirement causes a `TypeError` exception, and returns a number representing the argument's code point.

# Operations on strings

- `ord()`
- # Demonstrating the ord() function.
- char_1 = 'a'
- char_2 = ' '  # space
- print(ord(char_1))
- print(ord(char_2))


# Operations on strings

- `chr()`

- If you know the code point (number) and want to get the corresponding character, you can use a function named chr().
- The function **takes a code point and returns its character**.
- Invoking it with an invalid argument (e.g., a negative or invalid code point) causes `ValueError` or `TypeError` exceptions.

# Operations on strings

- `chr()`
- # Demonstrating the chr() function.
- print(chr(97))
- print(chr(945))


# Strings as sequences: indexing

- Strings aren't lists, but **you can treat them like lists in many particular cases**.
- For example, if you want to access any of a string's characters, you can do it using **indexing**, just like in the example below. Run the program:

```
# Indexing strings.
the_string = 'silly walks'
for ix in range(len(the_string)):
print(the_string[ix], end=' ')
print()
```

# Strings as sequences: iterating

- **Iterating through the strings** works, too. Look at the example below:

```
# Iterating through a string.
the_string = 'silly walks'
for character in the_string:
print(character, end=' ')
print()
```

# Slices

```
# Slices
alpha = "abdefg"

print(alpha[1:3])
print(alpha[3:])
print(alpha[:3])
print(alpha[3:-2])
print(alpha[-3:4])
print(alpha[::2])
print(alpha[1::2])
```

# The in and not in operators

- **The `in` operator**
- The in operator shouldn't surprise you when applied to strings - it simply **checks if its left argument (a string) can be found anywhere within the right argument (another string)**.
- The result of the check is simply True or False.
- Look at the example program below. This is how the in operator works:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
print("f" in alphabet)
print("F" in alphabet)
print("1" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```

# The in and not in operators

- **The `not in` operator**
- As you probably suspect, the not in operator is also applicable here.
- This is how it works:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

# Python strings are immutable

- The first important difference **doesn't allow you to use the del instruction to remove anything from a string**.
- The example here won't work:
- `alphabet = "abcdefghijklmnopqrstuvwxyz"`
- `del alphabet[0]`

- The only thing you can do with del and a string is to **remove the string as a whole**. Try to do it.
- Python strings **don't have the append() method** - you cannot expand them in any way.
- **the insert() method is illegal**, too:

# Operations on strings: continued

- alphabet = "bcdefghijklmnopqrstuvwxy"
- alphabet = "a" + alphabet
- alphabet = alphabet + "z"
- print(alphabet)


# Operations on strings

- `min()`
- The function **finds the minimum element of the sequence passed as an argument**. There is one condition - the sequence (string, list, it doesn't matter) **cannot be empty**, or else you'll get a `ValueError` exception.
- # Demonstrating min() - Example 1:

- print(min("aAbByYzZ"))
- # Demonstrating min() - Examples 2 & 3:
- t = 'The Knights Who Say "Ni!"'
- print('[' + min(t) + ']')
- t = [0, 1, 2]
- print(min(t))

# Operations on strings

- Note: It's an upper-case *A*. Why? Recall the ASCII table - which letters occupy first locations - upper or lower?

# Operations on strings

- `max()`

- Similarly, a function named max() **finds the maximum element of the sequence**.

```
# Demonstrating max() - Example 1:
print(max("aAbByYzZ"))
# Demonstrating max() - Examples 2 & 3:
t = 'The Knights Who Say "Ni!"'
print('[' + max(t) + ']')
t = [0, 1, 2]
print(max(t))
```

# Operations on strings

- **the `index()` method**

- The index() method (it's a method, not a function) **searches the sequence from the beginning, in order to find the first element of the value specified in its argument**.
- Note: the element searched for must occur in the sequence - **its absence will cause a `ValueError` exception**.
- The method returns the **index of the first occurrence of the argument** (which means that the lowest possible result is `0`, while the highest is the length of argument decremented by `1`).
-

# Operations on strings

- **the `index()` method**
- # Demonstrating the index() method:
- print("aAbByYzZaA".index("b"))
- print("aAbByYzZaA".index("Z"))
- print("aAbByYzZaA".index("A"))

# Operations on strings

- **the list() function**

- The list() function **takes its argument (a string) and creates a new list containing all the string's characters, one per list element**.
- Note: it's not strictly a string function - list() is able to create a new list from many other entities (e.g., from tuples and dictionaries).

```
# Demonstrating the list() function:
print(list("abcabc"))
# Demonstrating the count() method:
print("abcabc".count("b"))
print('abcabc'.count("d"))
```

# Operations on strings

- **the `count()` method**

- The count() method **counts all occurrences of the element inside the sequence**. The absence of such elements doesn't cause any problems.

# Operations on strings

- .