

# INST0002 PROGRAMMING 1 & INST0091 INTRODUCTION TO PROGRAMMING

LECTURE 05 – STRINGS

**Dr Daniel Onah**  
Lecturer  
Module Leader

# Copyright Note:

**Copyright Licence** of this lecture resources is with the Module Lecturer named in the front page of the slides. If this lecture draws upon work by third parties (e.g. Case Study publishers) such third parties also hold copyright. It must not be copied, reproduced, transferred, distributed, leased, licensed or shared with any other individual(s) and/or organisations, including web-based organisations, online platforms without permission of the copyright holder(s) at any point in time.

# Recap

- We discussed about Functions
- We looked at predefined and user-defined functions
- We looked parameter and argument passing
- We discussed about local and global scope
- We discussed two types of functions – lambda and recursive

Re: Cap☺



# Outline/Learning outcomes

- To be able to understand the various **types of string operations**
- Use of **built-in string** functions
- Use of **string formatting operator**
- Understand string comparison
- Understand string iterations
- Use of immutable strings

# Introduction to Strings

# Introduction to Strings

A string is a sequence of characters use within Python and programming in general. Strings are arrays of bytes in Python that represent Unicode characters. Since Python lacks a character data type, a single character is simply a string with one length.

A string literal is represented or enclosed using either single ('...') or double ("...") quotes. For example: 'HelloWorld' or "HelloWorld"

- In strings, the plus sign (+) means “concatenate”.
- When a string contains numbers, it is still a string
- We can convert numbers in a string into integer/whole numbers using the `int()` data type.

# Important Facts: Strings

Strings are important concept in Python. There are few important points about strings.

- Strings are amongst the most popular class types in Python
- Python can create strings simply by enclosing characters in quotes (single, double or triple).
- Python treats single-quotes the same as double-quotes.
- A string is a sequence of Unicode characters, and a character is simply a symbol.

# String Concatenation

String concatenation is the process of joining or gluing two or more strings together in a program by using the plus sign. The plus (+) serves two main purposes in programming (1) adding two numbers together (2) concatenating two strings together. This plus operator is considered as **operator overloaded** as it can be used to perform two different operations in programming.



# String Concatenation

String concatenation simply means combining two strings. If we have two strings and they are assigned to two different variables for example: `str1` and `str2`, these strings can be concatenated or combined as `str1` and `str2`.

There are four ways to concatenate a string:

- Using `+` operator
- Using `join()` method
- Using `%` string formatting operator
- Using `format()` function

# String Concatenation

When the `+` operator is applied to strings, it means “concatenation”

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere
>>> c = a + ' ' + 'There'
>>> print(c)
Hello There
>>>
```

# Using + operator

The plus (+) operator is simple way to concatenate two strings. It can be used to concatenate multiple strings together.

```
#!/usr/bin/env python  
  
# defining strings in python  
  
str1 = "Hello"  
  
str2 = "World!"  
  
concat = str1 + str2 # + operator is used to combine strings  
  
print(concat)
```

**Output:**  
HelloWorld!

# Using `join()` method

The `join()` method is a string method or function that returns a string in which the sequence elements have been joined using string(`str`) separator.

```
#!/usr/bin/env python

# defining strings in python

fname = "Danny"

lname = "Onah"

# join() method is used to join strings

print("".join([fname, lname])
```

**Output:**

DannyOnah

# Using `join()` method

The `join()` method is a string method or function that returns a string in which the sequence elements have been joined using string(`str`) separator.

```
#!/usr/bin/env python

# defining strings in python

fname = "Danny"

lname = "Onah"

# join() method with a separator space (" ")

name = (" ").join([fname, lname])

print(name)
```

**Output:**

Danny Onah

# Concatenate using `join()`

We can also concatenate two or more strings together using the `join()` function

```
#!/usr/bin/env python  
greeting = ["Hello,", "welcome to", "INST0002 Programming 1!"]  
  
print(" ".join(greeting))
```

**Output:**

Hello welcome to INST0002 Programming 1!

# Concatenate using `join()`

We can also concatenate two or more strings together using the `join()` function

```
#!/usr/bin/env python  
sun=("The"," ","sun"," ","will"," ","come"," ","up"," ","tomorrow.")  
print(" ".join(sun))
```

**Output:**

The sun will come up tomorrow.

# Using % Operator

Here using % operator simply means we are using a [string formatting operator](#) for concatenating the strings.

```
#!/usr/bin/env python

"""
A program using the string formatting operator (%) to concatenate strings
"""

fname = "Danny"
lname = "Onah"

# string formatting(%) operator
print("%s %s"%(fname,lname))
```

**Output:**  
Danny Onah



# Using `format()` Function

The `format()` function is an example of string **formatting method**. With this `format()` function, it is possible to perform multiple substitutions and value formatting. In order to concatenate a string, we use `str.format()` construct.

This method uses positional formatting to concatenate elements inside a series of string literal. For example, the `format()` function in the below code combines the string values stored in the `fname` and `lname` variables and then stores the values in another variable `name`.

The curly braces are used to position strings in a specific order. The first variable is stored in the first set of curly braces, while the second variable is stored in the second set of curly braces.

# Using `format()` Function

```
#!/usr/bin/env python

"""
A program using the string format() function to concatenate strings
"""

# defining strings
fname = "Danny"
lname = "Onah"

print("{} {}".format(fname, lname)) # using format() function

# using a variable 'name' to store the output of lname & fname
name = ("{} {}".format(fname, lname))
print(name)
```

**Output:**  
Danny Onah  
Danny Onah

# Concatenate strings and integer

We **cannot** concatenate a string literal with an integer literal.

```
#!/usr/bin/env python
# We cannot concatenate a string with an integer without first casting the string to integer

str3 = '12345' # string literal

str3 = str3 + 1 # concatenating a string literal with an integer literal

print(str3)
```

## Output:

Traceback (most recent call last):

File "concat\_string\_int.py", line 5, in <module>

str3 = str3 + 1

TypeError: cannot concatenate 'str' and 'int' objects

# Concatenate strings and integer:

## Solution

We **can** concatenate a string literal with an integer literal, if the string is cast into an integer. Example:

```
#!/usr/bin/env python
```

```
# We can concatenate a string with an integer by first casting the string to integer
```

```
str3 = '12345' # string object declaration or initialisation
```

```
str3 = int(str3) + 1 # string cast to integer and added to an integer 1
```

```
print(str3)
```

**Output:**

**12346**

# Another Conversion

## String Conversions

- You can also use `int()` and `float()` to convert between strings and integers
- You will get an **error** if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str'
and 'int'
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

# Reading and Converting

Normally, we prefer to read data-in using **strings** and then **parse** and **convert** the data as we need to the required data type.

This gives us more control over error situations and/or bad user input. Input numbers must be **converted** from string.

## Example

```
book = input('Enter:') # input entered 100  
value = book - 10
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

# Reading and Converting

Normally, we prefer to read data in using **strings** and then **parse** and **convert** the data as we need to the required data type.

This gives us more control over error situations and/or bad user input. Input numbers must be **converted** from string.

## Example

```
book = input('Enter:') # input entered 100  
value = int(book) - 10  
print(value)
```

Output: 90
---------------

# Accessing Strings

Square brackets are used to access the string's elements. We can look inside strings and can get any single character in a string using an index specified in square brackets. The index value must be an integer and starts at zero. This index value can also be an expression that is computed.

d	a	n	n	y
0	1	2	3	4



*Index values of 'HELLO WORLD'*

H	E	L	L	O		W	O	R	L	D
0	1	2	3	4	5	6	7	8	9	10
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Positive Index Values (+) →

Negative Index Values (-) →



# Accessing Strings

```
#!/usr/bin/env python
```

```
"""
```

```
A program accessing the elements of a string using index
```

```
"""
```

```
name = "Danny"
```

```
letter = name[1]
```

```
print(letter)
```

```
x = 2
```

```
char = name[x - 1]
```

```
print(char)
```



Output:

a

a

# Accessing Strings – Index Example

```
#!/usr/bin/env python
```

```
"""
```

```
A program accessing the elements of a string
```

```
"""
```

```
name = "Danny"
```

```
print(name[0])
```

```
print(name[1])
```

```
print(name[2])
```

```
print(name[3])
```

```
print(name[4])
```

**Output:**

D  
a  
n  
n  
y

# IndexError

You will get a **Python error** if you attempt to **index beyond the length of the string**. Care should be taken (be careful) when constructing index values and **slices**. The **index value** can only be set to the **length of the string**.

## Example

```
#!/usr/bin/env python
"""
```

```
A program accessing the elements of a string beyond the length of
the string
"""
```

```
name = "Danny"
print(name[5])
```



```
Traceback (most recent call last): File
"<stdin>", line 7, in <module>
    print(name[5])
IndexError: string index out of range
```

# Accessing Strings – *For Loop* Example

We can access the element of a string using a *for loop*, character by character.

```
#!/usr/bin/env python
```

```
"""
```

```
A program accessing the elements of a string using a for loop
```

```
"""
```

```
for i in "Danny":
```

```
    print(i)
```

**Output:**

D

a

n

n

y

# String Concepts

# Appending Strings

In Python, **appending to strings means adding one string to another**. For example, let say we have two strings, and they are assigned to two different variables *str1* and *str2*. These strings can be appended together which means we can add *str1* to *str2*. There are two ways we can append two strings together:

- By using ***+=*** operator and
- By using *join()*

# Using += operator

This process of appending strings together in Python is a very simple and straightforward process or way as compared to traditional approach used in other programming languages. Example of this traditional approach, might be using a dedicated function to perform the task.

Let us look at an example to understand the concept properly in Python.

# Using += operator: Example

```
#!/usr/bin/env python

"""
A program to append a string using += operator
"""

# initialising first string variable
str1 = "Danny"

# initialising second string variable
str2 = "Onah"

# display the first string
print("The first-string value is: " + str(str1))

# display the second string
print("The second-string value is: " + str(str2))

# appending the string together
str1 += str2 # Using the += operator to append the strings together

# display the outcome
print("The appended string is: " + str1)
```

**Output:**

The first-string value is: Danny  
The second-string value is: Onah  
The appended string is: DannyOnah



# Using `join()` Function

We can certainly use the `join()` method to append strings. The major advantage of this method as compared to the `+=` operator approach is that the `join()` function can be used to concatenate so *many strings* rather than just *only two* as done in the case of the `+=` operator.

**Note:** The `join()` method (`str.join()`) takes exactly only one argument in its function.

Let's look at an example to help us understand the concept.

# Using `join()` Function: Example

```
#!/usr/bin/env python
"""
A program to use the join() function to append so many strings
"""
# Initialising the first string
str1 = "Daniel"
# Initialising the second string
str2 = "Onah"
# Initialising the third string
str3 = "Danny"
# display the first string
print("The first name is: " + str(str1))
# display the second string
print("The last name is: " + str(str2))
# display the third string
print("Another name is: " + str(str3))
# appending the string to another
name = ("".join((str1, str2, str3))) # using join() method
print("The appended name is: " + name)
```

**Output:**

The first name is: Daniel

The last name is: Onah

Another name is: Danny

The appended name is: DanielOnahDanny

# Multiplying Strings

In Python, multiplying strings is much easier because it is like multiplying another primitive data type (e.g., integers, floating-point value etc.). There are several ways we could use to multiply strings together.

Let's look at an example to help us understand the concept.

# Multiplying Strings: Example

```
#!/usr/bin/env python

"""
A program to multiply string
"""

str1 = "INST0002 Python Programming"
str2 = 2*str1 # first method of string multiplication
print(str2)

str3 = 4*(str1) # second method of string multiplication
print(str3)

str4 = 3*('INST0002', 'Python', 'Programming')*3 # third method of string multiplication
print(str4) # 3 * 3 = 9

str5 = ('INST0002', 'Python', 'Programming')*2 # fourth method of string multiplication
print(str5)
```

Output:

```
INST0002 Python ProgrammingINST0002 Python Programming
INST0002 Python ProgrammingINST0002 Python ProgrammingINST0002 Python ProgrammingINST0002 Python Programming
('INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming',
'INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming',
'INST0002', 'Python', 'Programming')
('INST0002', 'Python', 'Programming', 'INST0002', 'Python', 'Programming')
```

# Immutable Strings

An immutable string objects are **strings whose values cannot be change after they are assigned to string object**. This simply means , an immutable string are strings whose values **cannot be updated**. Once, we assign the string's value, it **cannot be reassigned**. When you try to reassign this value in your program, **the program will produce an error**.

Let's look at an example to help us understand this concept better.

```
#!/usr/bin/env python

"""
A program to test string immutability
"""

# define a string
name = "Danny"
print(name)
name[0] = "D" # Cannot reassign
```

Output:

```
name[0] = "D" # Cannot reassign
TypeError: 'str' object does not support item assignment
```

# String Formatting Operator

String formatting operator uses the `%` operator for the operations. Various symbols can be used along with the `%` for formatting strings. Let's look at some string formatting symbols and their description.

## *String formatting symbols and description*

Symbol	Description
<code>%f</code>	It is used for a floating-point real number
<code>%d</code>	It is used for a signed decimal integer
<code>%u</code>	It is used for an unsigned decimal integer
<code>%o</code>	It is used for an octal integer
<code>%X</code>	It is used for a hexadecimal integer (Uppercase letters)
<code>%e</code>	It is used for an exponential notation (with lowercase 'e')
<code>%E</code>	It is used for an exponential notation (with UPPER-case 'E')

# String Formatting Operator

String formatting operator uses the `%` operator for the operations. Various symbols can be used along with the `%` for formatting strings. Let's look at some string formatting symbols and their description.

## *String formatting symbols and description*

Symbol	Description
<code>%g</code>	It is used for the shorter of <code>%f</code> and <code>%e</code>
<code>%G</code>	It is used for the shorter of <code>%f</code> and <code>%E</code>
<code>%c</code>	It is used for a character
<code>%s</code>	It is used for a string conversion via <code>str()</code> before formatting
<code>%i</code>	It is used for a signed decimal integer
<code>%x</code>	It is used for a hexadecimal integer (lowercase letters)
<code>%r</code>	It is used for display string as a raw data

# String Formatting Operator: Example

```
#!/usr/bin/env python
"""
A program to test string formatting operations
"""

# initialising a variable as a string
var = '15'
string = "Displaying variable as string = %s"%(var)
print(string)

# Displaying variable as a raw data
print("Displaying variable as a raw data = %r"%(var))

# Convert the variable to integer
var = int(var)
string = "Displaying variable as integer = %d"%(var)
print(string)
```



# String Formatting Operator: Example

```
print("Displaying variable as float = %f"%(var))
```

```
print("Displaying variable with special character = %c Ram"%(var))
```

```
print("Displaying variable as hexadecimal = %x"%(var))
```

```
print("Displaying variable as octal = %o"%(var))
```

## Output:

Displaying variable as string = 15

Displaying variable as a raw data = '15'

Displaying variable as integer = 15

Displaying variable as float = 15.000000

Displaying variable with special character = Ram

Displaying variable as hexadecimal = f

Displaying variable as octal = 17

# Built-in String Functions

There are several types of built-in string functions. We use these functions to get the desired results. *One important point about these functions is that they return new strings.* **The functions cannot change the original string.** Here are a few essential built-in string functions.

Built-in	Description
<code>count()</code>	It returns the number of times a specified value occurs in a string
<code>islower()</code>	It returns <b>True</b> if all characters in the string are lower case
<code>split()</code>	It splits the string at the specified separator and returns a list
<code>endswith()</code>	It returns True if the string ends with the specified value
<code>center()</code>	It returns a centered string
<code>isalnum()</code>	It returns True if all characters in the string are alphanumeric
<code>isnumeric()</code>	It returns True if all characters in the string are numeric
<code>title()</code>	It converts the first character of each word to upper case

# Built-in String Functions

Let's look at an example to understand the above built-in string functions.

```
#!/usr/bin/env python

"""
A program to test multiple built-in functions
"""

# built-in string functions
text = "Python is a good programming language., Python is very easy to learn, Python is easy to understand."

counter = text.count("Python")
print("The word Python appears {}".format(counter) + "times.")

lowerCase = text.islower() # check if all characters in the text are in lower case
print(lowerCase)

splitText = text.split()# Split a string into a list where each word is a list item
print(splitText)
```

# Built-in String Functions

Let's look at an example to understand the above built-in string functions.

```
endWith = text.endswith(".") # check if the string ends with a (.)  
print(endWith)  
  
centerText = text.center(10) # space of 10 characters  
print(centerText)  
  
alphaNumeric = text.isalnum() # check if all the characters in the text are alphanumeric  
print(alphaNumeric)  
  
numericText = text.isnumeric() # check if all the characters in the text are numeric  
print(numericText)  
  
titleText = text.title() # first letter in each word upper case  
print(titleText)
```

# Built-in String Functions

Let's look at an example to understand the above built-in string functions.

## Output:

```
The word Python appears 3 times.
```

```
False
```

```
['Python', 'is', 'a', 'good', 'programming', 'language.', 'Python', 'is', 'very', 'easy', 'to', 'learn,', 'Python', 'is', 'easy', 'to', 'understand.']
```

```
True
```

```
Python is a good programming language., Python is very easy to learn, Python is easy to understand.
```

```
False
```

```
False
```

```
Python Is A Good Programming Language., Python Is Very Easy To Learn, Python Is Easy To Understand.
```

# String Functions

# String Library

- Python has a number of string **functions** which are in the **string library**
- These **functions** are already **built into** every string - we invoke them by appending the function to the string variable
- These **functions** do not modify the original string, instead they return a new string that has been altered

```
>>> greeting = 'Hello Danny'
>>> greet = greeting.lower()
>>> print(greet)
hello danny
>>> print(greeting)
Hello Danny
>>> print('Hi There'.lower())
hi there
>>>
```

# String Library

```
>>> greeting = 'Hello world'
>>> type(greeting)
<class 'str'>
>>> dir(greeting)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

<https://docs.python.org/3/library/stdtypes.html#string-methods>



# String Methods

**`str.replace(old, new[, count])`**

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

**`str.rfind(sub[, start[, end]])`**

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

**`str.rindex(sub[, start[, end]])`**

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

**`str.rjust(width[, fillchar])`**

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

**`str.rpartition(sep)`**

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

**`str.rsplit(sep=None, maxsplit=-1)`**

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

# String Library

```
str.capitalize()  
str.center(width[, fillchar])  
str.endswith(suffix[, start[, end]])  
str.find(sub[, start[, end]])  
str.lstrip([chars])
```

```
str.replace(old, new[, count])  
str.lower()  
str.rstrip([chars])  
str.strip([chars])  
str.upper()
```

# Slice() Function Operator

A `slice()` is a string function that returns a slice object that can be used to slice `strings`, `lists`, `tuple`, etc.

Syntax of a `slice()` function with parameters:

```
slice(stop)
slice(start, stop, step)
```

**Start:** A start parameter specifies the index at which an object's slicing begins.

**Stop:** A stop parameter sets the index at which an object's slicing comes to an end (to a halt).

**Step:** The increment between each index for slicing is determined by the step parameter, an optional statement.

# Important Facts

The following are important points about the `slice()` function:

- The `slice()` function returns a sliced object that only contains elements from the defined set.
- If only one parameter is passed, then the `start` and `step` are considered to be `None`.
- `slice()` is different from `index[]` in the sense that `slice` does not consider length of element unlike `index`.

Let's look at an example to help us understand the concept of `slice()` function:

# Slice() Example

```
#!/usr/bin/env python
"""
A program to demonstrate slice() function
"""

# It start the slice object at position 2, and slice to position 4 , and return the result
name = ("D", "a", "n", "n", "y")
slice1 = slice(2, 4)
print(name[slice1])

# It use the step parameter to return every second item in step of 2
name = ("D", "a", "n", "n", "y", "O", "n", "a", "h")
slice1 = slice(0, 9, 2) # slice is different from index[]. slice does not consider length of element
print(name[slice1])
```

**Output:**

```
('n', 'n')
('D', 'n', 'y', 'n', 'h')
```

# String Slicing

```
#!/usr/bin/env python

"""
A program to demonstrate string slicing
"""

# string slicing
str = "Danny"
slice1 = slice(2) # slice the first two characters
slice2 = slice(1, 4, 3) # slice the first 4 char (stop at 4), starting from 1 in step of 3
print("String slicing...")
print(str[slice1])
print(str[slice2])
```

**Output:**  
String slicing...  
Da  
a

# List Slicing

```
#!/usr/bin/env python

"""
A program to demonstrate list slicing
"""

# List slicing
numberList = [5, 6, 7, 8, 9, 10]
slice1 = slice(2) # slice the first two integers
slice2 = slice(1, 4, 2) # slice the list starting from index-1 in step of two and stop at the fourth integer
(exclusive)
print("\nList slicing...")
print(numberList[slice1])
print(numberList[slice2])
```

**Output:**  
List slicing...  
[5, 6]  
[6, 8]

# Tuple Slicing

```
#!/usr/bin/env python
```

```
"""
```

```
A program demonstrating a slice tuple
```

```
"""
```

```
# Tuple slicing
```

```
numberTuple = (4, 5, 6, 7, 8, 9)
```

```
slice1 = slice(3) # slice the first three integers
```

```
slice2 = slice(1, 5, 2) # slice the tuple from first index-1 in step of two and stop at fifth integer
```

```
print("\nTuple slicing...")
```

```
print(numberTuple[slice1])
```

```
print(numberTuple[slice2])
```

**Output:**  
Tuple slicing...  
(4, 5, 6)  
(5, 7)



# String Slicing

## Slicing Strings

D	a	n	n	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

- We can also look at any continuous section of a string using a **colon operator**
- The second number is one beyond the end of the slice - “up to but not including”
- If the second number is beyond the end of the string, it stops at the end

```
>>> s = 'Danny Python'
>>> print(s[0:4])
Dann
>>> print(s[6:7])
P
>>> print(s[6:20])
Python
```

# String Slicing

## Slicing Strings

D	a	n	n	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

If we **leave off** the **first number** or the **last number** of the slice, it is assumed to be the beginning or end of the string respectively

```
>>> s = 'Danny Python'
>>> print(s[:2])
Da
>>> print(s[8:])
thon
>>> print(s[:])
Danny Python
```

# The *ord()* Function

The *ord()* function is a built-in function in Python which features **returns the single Unicode character's integer** as an integer value. The return integer represents the *Unicode code point* for the character. Furthermore, the *ord()* function returns an integer representing the character's *Unicode code point* when passed a string of length 1. A **TypeError** will be raised if the *string length is greater than one*.

Let's look at an example to help us understand the concept of *ord()* functions.

# The *ord()* Function

```
#!/usr/bin/env python

"""
A program to demonstrate ord() function
"""

letter1 = ord('A')
letter2 = ord('a')
# displaying the unicode value
print("The Unicode value for 'A' is {} and for 'a' is {}".format(letter1, letter2))
```

**Output:**

The Unicode value for 'A' is 65 and for 'a' is 97

# The *chr()* Function

In Python, the *chr()* function returns a character or string from an integer. The integer represents the character's Unicode code point which is then passed into the *chr()* function as an argument. This *chr()* function does the inverse of the *ord()* function.

Let's look at an example to help us understand the concept of *chr()* function better.

# The *chr()* Function: Example

```
#!/usr/bin/env python

"""
A program to demonstrate the chr() function
"""

letter1 = chr(65) # return a character of Unicode value 65
letter2 = chr(97) # return a character of Unicode value 97
print("The character of the Unicode value '65' is: {} and for '97' is: {}".format(letter1, letter2))
```

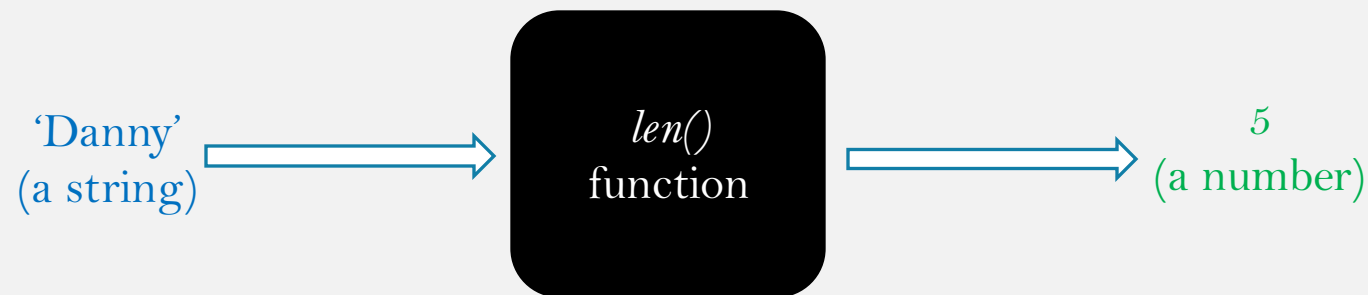
**Output:**

The character of the Unicode value '65' is: A and for '97' is: a

# *len()* Function

If you recall from previous lecture on Functions. We discussed that a function is a **block or sequence of stored code** *that we use to perform specific task or action*. A function takes some **input** and produces an **output**.

The built-in function *len()* gives us the length of a string.



# Concepts in String: *len()*

Concepts of strings such as *len()* function is used for getting the **length** of a string. Keywords such as *in*, *not*, *in* are used for checking the substring or whether a **character is present or not in a string**. It can also be used in an *if* statement.

```
#!/usr/bin/env python

"""
A program to find the length of a string
"""
# print the length of a string
name = "Danny"
print(len(name)) # string length
```

**Output:**

5



# Keyword *in*

We can use the keyword *in* to find a word or character in a string.  
For example:

```
#!/usr/bin/env python

"""
A program to use 'in' keyword to check if a word is in a string
"""

# print "simple" in a string

statement = "Python Programming is so simple"
print("simple" in statement)
```

**Output:**  
True

# Using *in* as a Logical Operator

- The **in** keyword can also be used to check to see if one string is “in” another string
- The **in** expression is a logical expression that returns **True** or **False** and can be used in an **if** statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print('Found it!')
...
Found it!
>>>
```

# Keyword *in* and *if*

We can use the keyword *in* to find a word or character in a string.  
For example:

```
#!/usr/bin/env python

"""
A program to use 'in' keyword to check if a word is in a string
"""
# print only if "simple" is present in a string
statement = "Python programming is so simple"
if("simple" in statement):
    print("Yes, 'simple' is present in the statement.")
```

**Output:**  
Yes, 'simple' is present  
in the statement.

# Keyword *not* and *in*

We can use the keyword *not* and *in* to find a word or character in a string. For example:

```
#!/usr/bin/env python
#print "programming" is not in a string

statement = "Python is so simple"
print("programming" not in statement)
```

**Output:**  
True

# Keyword *not* and *in*

We can use the keyword *not* and *in* to find a word or character in a string. For example:

```
#!/usr/bin/env python

#print "simple" is not in a string
statement = "Python is so simple"

print("simple" not in statement)
```

**Output:**  
False

# Comparing and Searching Strings

# Comparing Strings

Comparing strings means identifying whether the *two strings being compared are equivalent to each other or not*. There are three ways to compare strings in Python.

- Using *relational* operators
- Using *is* and *is not* constructs
- *User-defined function*

*Note: In Python, all strings are represented as **Unicode**.*

# Using Relational Operators

In Python **relational operators** compare the **Unicode values of the characters** of the strings. It returns a **Boolean value** according to the operator used. The following are the relational operators available in Python.

Operator	Description
<code>==</code>	This operator checks whether two strings are equal
<code>!=</code>	This operator checks if two strings are not equal
<code>&lt;</code>	This operator checks if the string on its left is smaller than the one on its right
<code>&lt;=</code>	This operator checks if the string on its left is smaller than or equal to that on its right
<code>&gt;</code>	This operator checks if the string on its left is greater than that on its right
<code>&gt;=</code>	This operator checks if the string on its left is greater than or equal to that on its right



# Using Relational Operators

Let's look at an example to help us understand the concept of relational operators in string.

```
#!/usr/bin/env python
```

```
"""
```

```
A program to check the concept of relational operators in string
```

```
"""
```

```
print("Danny" == "Danny")
```

```
print("Danny" < "danny")
```

```
print("Danny" > "danny")
```

```
print("Danny" <= "danny")
```

```
print("Danny" >= "danny")
```

```
print("Danny" != "Danny")
```

**Output:**

True

True

False

True

False

False

# Using *is* and *is not*

In Python, this type of operator **checks whether both the operands refer to the same object or not**. Let's look an example to help us understand the concept better.

```
#!/usr/bin/env python

"""
A program to check strings using is and is not
"""

# define strings
name1 = "Danny"
name2 = "Danny"
name3 = name1

# use the 'is not' operand
print(name1 is not name1)
print(name1 is not name2)
print(name1 is not name3)

#use the 'is' operand
print(name1 is name1)
print(name1 is name2)
print(name1 is name3)
```

**Output:**

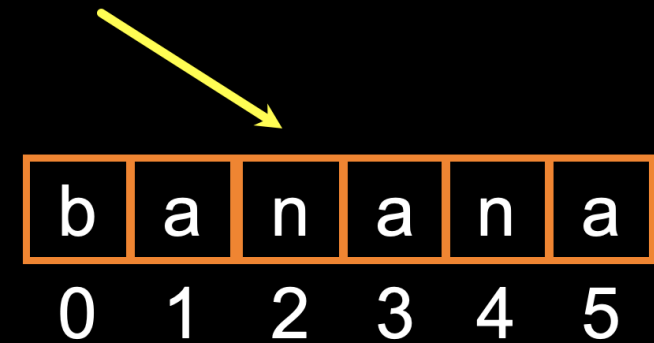
False  
False  
False  
True  
True  
True

# String Comparison – *Conditional Statements*

```
if word == 'banana':  
    print('All right, bananas.')  
if word < 'banana':  
    print('Your word,' + word + ', comes before banana.')elif word > 'banana':  
    print('Your word,' + word + ', comes after banana.')else:  
    print('All right, bananas.')
```

# Search a String: *find()*

- We use the `find()` function to search for a substring within another string
- `find()` finds the first occurrence of the substring
- If the substring is not found, `find()` returns `-1`
- Remember that string position starts at zero



```
>>> fruit = 'banana'
>>> indexPos = fruit.find('na')
>>> print(indexPos)
2
>>> letter = fruit.find('z')
>>> print(letter)
-1
```

# *Upper-* and *Lower*-Case String

- You can make a copy of a string in **lower case** or **upper case**
- Often when we are searching for a string using **find()** we first convert the string to lower case so we can search a string regardless of case

```
>>> greeting = 'Hello Danny'
>>> greet = greeting.upper()
>>> print(greet)
HELLO DANNY
>>> greet = greeting.lower()
>>> print(greet)
hello danny
>>>
```

# Search and *Replace()*

- The `replace()` function is like a “search and replace” operation in a word processor
- It replaces **all occurrences** of the **search string** with the **replacement string**

```
>>> greeting = 'Hello Daniel'
>>> greet =
greeting.replace('Daniel', 'Elaine')
>>> print(greet)
Hello Elaine
>>> greet = greeting.replace('e', 'x')
>>> print(greet)
HXllo DaniXl
>>>
```

# Stripping Whitespace: *lstrip()*, *rstrip()*, *strip()*

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- *lstrip()* and *rstrip()* remove whitespace at the left or right
- *strip()* removes both beginning and ending whitespace

```
>>> greet = '    Hello Danny    '  
>>> greet.lstrip()  
'Hello Danny '  
>>> greet.rstrip()  
'    Hello Danny'  
>>> greet.strip()  
'Hello Danny'  
>>>
```

# String Prefixes: *startswith()*

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

*Notice that it is False because the 'p' in the sentence 'Please have a nice day' is capital and the 'p' in the construct "(line.startswith('p'))" is smaller case 'p'. **Python is case sensitive.***



# Parsing and Extracting Data

11                      21  
↓                      ↓  
From d.onah@ucl.ac.uk Wed Feb 1 20:45:10 2023

```
>>> data = 'From d.onah@ucl.ac.uk Wed Feb 1 20:45:10 2023'
>>> indexPos = data.find('@')
>>> print(indexPos)
11
>>> spaceIndexPos = data.find(' ', indexPos)
>>> print(spaceIndexPos)
21
>>> hostDomain = data[indexPos+1 : spaceIndexPos]
>>> print(hostDomain) # print from index 12 through to 21 (exclusive)
ucl.ac.uk
```

# Function and Iterating Strings

# User-defined Function

We can create user-defined functions for string comparison. A user-defined function will compare strings based upon the number of digits.

Let's look at an example to help us understand this better.

# User-defined Function: Example

```
#!/usr/bin/env python

"""
A program to design a user-defined function for string comparison
"""

def comparison(str1, str2): # comparison of string based on the number of digits
    count1 = 0
    count2 = 0

    for i in range(len(str1)):
        if str1[i] >= "0" and str1[i] <= "9":
            count1 += 1

    for i in range(len(str2)):
        if str2[i] >= "0" and str2[i] <= "9":
            count2 += 1

    return count1 == count2

print(comparison("234", "5672"))
print(comparison("1246", "Danny"))
print(comparison("76Danny", "Danny76"))
```

**Output:**  
False  
False  
True

# Iterating Strings

In Python, **iterating over a string simply** means **accessing each of its character one at a time**. There are various ways we could iterate over the characters of a string in Python. Let's look at an example to see how this is done and to help us understand this better.

```
#!/usr/bin/env python

"""
A program to perform string iteration
"""

# using simple iteration and range() function
name1 = "Danny"

for var in name1:
    print(var, end=' ') # display horizontally
print("\n")

name2 = "Onah"
for i in range(0, len(name2)): # Iterate over index
    print(name2[i]) # display vertically
```

**Output:**

D a n n y

O  
n  
a  
h

# Iterating Strings

In Python, **iterating over a string simply** means **accessing each of its character one at a time**. There are various ways we could iterate over the characters of a string in Python. Let's look at an example to see how this is done and to help us understand this better.

```
#!/usr/bin/env python

"""
A program to perform string iteration
"""

# Using enumerate() function
name = "Danny"

# Iterate over the string
for i, n in enumerate(name):# 'i' display the string index value
    print(n)
```

**Output:**

D  
a  
n  
n  
y

# Iterate Using *split()* Function

In Python, we can also iterate over the words of a string. For example, a string of several words separated by spaces can be iterated over. In Python, there are many methods for iterating over words in a string.

One way to do this is to split the string into a list of words, but this method will fail if the string contains punctuation marks and if this is not handled properly.

Let's look at an example to help us understand this concept better.

# Iterate Using *split()* Function: Example

```
#!/usr/bin/env python
```

```
"""
```

```
A program to perform string iteration using split() function
```

```
"""
```

```
# define a string
```

```
statement = "Python is a good programming language., Python is easy to use, Python is easy to learn and understand."
```

```
print("The original statement is: " + statement)
```

```
words = statement.split() # using the split to extract words from string
```

```
print("\nThe words of the string are:")
```

```
for w in words:
```

```
    print(w)
```



# Iterate Using *split()* Function: Output

The original statement is: Python is a good programming language., Python is easy to use, Python is easy to learn and understand.

**The words of the string are:**

Python  
is  
a  
good  
programming  
language.,  
Python  
is  
easy  
to  
use,  
Python  
is  
easy  
to  
learn  
and  
understand.

# Using *re.findall()*

This approach necessitates the use of **regular expressions (re)** to compute the string. This will allow us to handle the issues of the punctuation marks. After the preprocessing or filtering of the string and extracting words, the punctuation marks are ignored in that process. The *findall()* function returns the list of words in the original text (ignoring the punctuation marks e.g. “.” , “,” etc.).

Let's look at an example to help us understand how this is done.

# Using *re.findall()* : Example

```
#!/usr/bin/env python

"""
A program to filter and find words in a string using regular expression approach
"""

# findall() example

import re
statement = "Python is a good programming language., Python is easy to use, Python is easy to learn and understand."
print("The original statement is: " + statement)

words = re.findall(r'\w+', statement) # filtering & processing words

print("\nThe words of the string are: ")
for w in words: # searching or iterating through the word
    print(w)
```

# Using *re.findall()* : Output

The original statement is: Python is a good programming language., Python is easy to use, Python is easy to learn and understand.

**The words of the string are:**

Python

is

a

good

programming

language

Python

is

easy

to

use

Python

is

easy

to

learn

and

understand

# Looping Through Strings

Using a **while** statement, an **iteration variable**, and the **len** function, we can construct a loop to look at each of the letters in a string individually

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

0	b
1	a
2	n
3	a
4	n
5	a

# Looping Through Strings

- A definite loop using a **for** statement is much more elegant
- The **iteration variable** is completely taken care of by the **for** loop

```
fruit = 'banana'  
for letter in fruit:  
    print(letter)
```

b  
a  
n  
a  
n  
a

# Looping Through Strings

- A definite loop using a **for** statement is much more **elegant**
- The **iteration variable** is completely taken care of by the **for** loop

```
fruit = 'banana'
for letter in fruit:
    print(letter)
```

```
index = 0
while index < len(fruit) :
    letter = fruit[index]
    print(letter)
    index = index + 1
```

b  
a  
n  
a  
n  
a

# Looping and Counting

This is a simple loop that **loops through each letter** in a string and counts the number of times the loop encounters the 'a' character

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print(count)
```



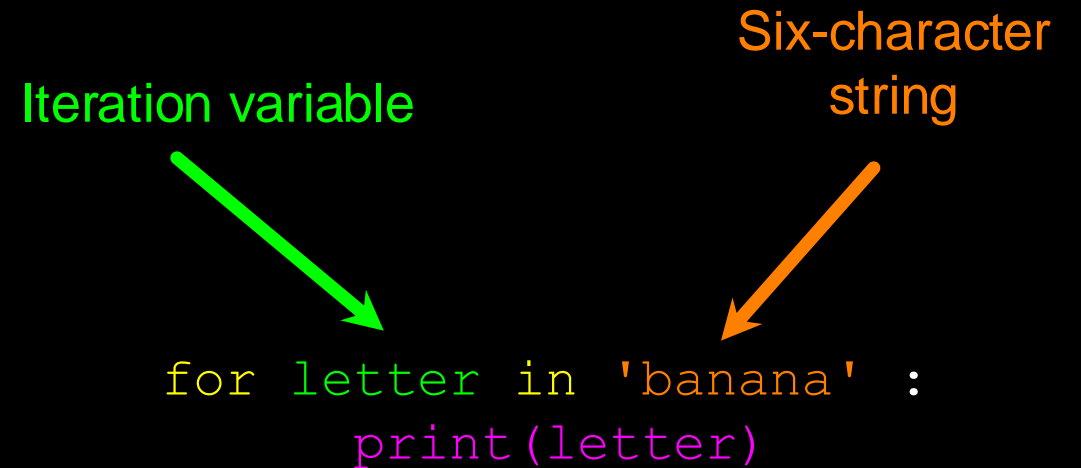
# Looking Deeper into *in*

- The **iteration variable** “iterates” through the **sequence** (ordered set)
- The **block (body)** of code is executed once for each value **in** the **sequence**
- The **iteration variable** moves through all of the values **in** the **sequence**

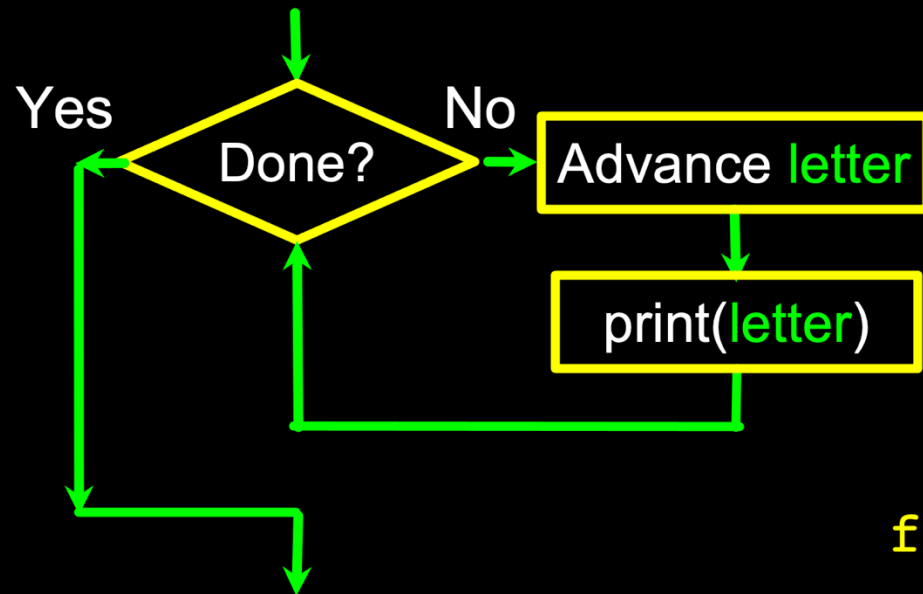
Iteration variable

Six-character string

```
for letter in 'banana' :  
    print(letter)
```

A diagram illustrating the components of a Python for loop. A green arrow points from the text 'Iteration variable' to the variable 'letter' in the code snippet. An orange arrow points from the text 'Six-character string' to the string 'banana' in the code snippet. The code snippet is: 'for letter in 'banana' : print(letter)'. The word 'letter' is green, 'in' is yellow, 'banana' is orange, and 'print(letter)' is purple.

# Flow Diagram and Looping



b	a	n	a	n	a
---	---	---	---	---	---

```
for letter in 'banana' :  
    print(letter)
```

The **iteration variable** “iterates” through the **string** and the **block (body)** of code is executed once for each value **in** the **sequence**

# String Class Module

# The String Module

In Python, `string` module is a built-in module for using classes and constants. Before we use this module, *we must first import it into our program* so it can be used successfully. There are various types of string constants defined in this `string` module. Let's see a few example of these string constants.

- **`string.ascii_letters`**: The following constants are concatenated: **`ascii_lowercase`** and **`ascii_uppercase`**.
- **`string.ascii_lowercase`**: The lowercase letters **`'abcdefghijklmnopqrstuvwxyz'`**. This value is not locale-dependent and will not change.

# The String Module

- **string.ascii\_uppercase:** The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.
- **string.digits:** The string `'0123456789'`
- **string.hexdigits:** The string `'0123456789abcdefABCDEF'`
- **string.punctuation:** String of ASCII characters which are considered punctuation characters in the C locale: `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.`

# The String Module: Example

```
#!/usr/bin/env python

"""
A program to use a string module for string computation
"""

# import the string library function

import string

# storing the value in variables
value1 = string.ascii_letters
value2 = string.ascii_lowercase
value3 = string.ascii_uppercase
value4 = string.digits
value5 = string.hexdigits
value6 = string.punctuation

# display the value
print(value1)
print(value2)
print(value3)
print(value4)
print(value5)
print(value6)
```

# The String Module: Output

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  
abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
0123456789  
0123456789abcdefABCDEF  
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

# String *capwords()* Function

In Python, the *capwords()* function splits the string into words and capitalizes each word in it . Let's look at an example to see how this done.

```
#!/usr/bin/env python

"""
A program to capitalize each word in a string
"""

import string # importing a string module

# defining the string
sentence = "My name is Danny Onah"
result = string.capwords(sentence)# capitalizing first letter of each word in the string
print(result)
```

Output: My Name Is Danny Onah



# String Module Classes

There are two types of string module classes in Python. These are :

- *Formatter*
- *Template*

We will discuss them in detail.

# Formatter

The `formatter` module class is very useful if we want to create a subclass and define the `format` string syntax. This class module is the same as the `str.format()` function. Let's look at an example to help us understand the concept.

```
#!/usr/bin/env python

"""
A program to format string using string formatter module
"""

from string import Formatter

formatter = Formatter()
print(formatter.format('{Name}', Name = 'Danny Onah'))
print(formatter.format('{} {Name}', 'Hello', Name='Danny Onah'))

# It is the same as format() function
print('{} {Name}'.format('Hello', Name='Danny Onah'))
```

**Output:**

Danny Onah  
Hello Danny Onah  
Hello Danny Onah

# Template Module Class

The **Template** *class enables us to construct output specification syntax that is easier to understand.* The format combines **\$ symbol** with valid Python identifiers, such as **alphanumeric characters** and **underscores**, to create **placeholder names**.

Let's look at an example to help us understand this better.

# Template Module Class

```
#!/usr/bin/env python

"""
A program using the template module to compute a string
"""

# import template module
from string import Template

# Creating a template

template = Template('P is $P')

# Substitute value
print(template.substitute({'P': 1}))

# List of names and ID
Name = [('Danny Onah', 158935), ('Daniel Onah', 256783), ('Dan Onah', 450973)]
template = Template('Hello $name, $id is your employee id.')

for n in Name:
    print(template.substitute(name = n[0], id = n[1]))
```

# Template Module Class: Output

P is 1

Hello Danny Onah, 158935 is your employee id.

Hello Daniel Onah, 256783 is your employee id.

Hello Dan Onah, 450973 is your employee id.

# Regular Expression

*Regular expression is a special sequence of characters that uses a specialized syntax to help us process or locate other strings or sets of strings in a text.* It is also used to define a set of strings (pattern) that corresponds to it. Here are a few metacharacters which are available to use with functions.

Metacharacters	Description
(\)	It is used to drop the special meaning of a character
([ ])	It represents a character class
(^)	It matches the beginning
(\$)	It matches the end
(.)	It matches any character except newline
(?)	It matches zero or one occurrence
( )	It matches any of the characters separated by it
(*)	It shows any number of occurrences (including 0 occurrences)

# Regular Expression: *re*

*Regular expression is a special sequence of characters that uses a specialized syntax to help us process or locate other strings or sets of strings in a text.* It is also used to define a set of strings (pattern) that corresponds to it. Here are a few metacharacters which are available to use with functions.

Metacharacters	Description
(+)	It is for one or more occurrences
({})	It indicates several occurrences of a preceding regular expression to match
(( ))	It encloses a group of regular expressions

Let's look at an example to help us understand how this works.

# Regular Expression

```
#!/usr/bin/env python

"""
A program to show the use of regular expression
"""

import re

p = re.compile('[a-z]') # here we indicated smaller letters from 'a-z'
print(p.findall("My name is Danny"))

p = re.compile('[A-Z]') # here we indicated capital letters from 'A-Z'
print(p.findall("My name is Danny"))
```

**Output:**

```
['y', 'n', 'a', 'm', 'e', 'i', 's', 'a', 'n', 'n', 'y']
['M', 'D']
```



# Regular Expression: *re*

It is very important to mention that the ‘\’ metacharacter plays a crucial role in *signaling different sequences in regular expression*. Below are examples on how it is used.

## *Metacharacters used along with backslash(\)*

Symbol	Description
\d	It's equivalent to the set class [0-9] and matches every decimal digit
\D	It can be used to match any non-digit character
\s	It is compatible with every whitespace character
\S	It can be used to match any non-whitespace character
\w	It is equivalent to the class [a-zA-Z0-9_], which matches every alphanumeric character
\W	It can be used to match any non-alphanumeric character



# Regular Expression: Output

```
['2', '9', '1', '9', '9', '4']
```

```
['29', '1994']
```

```
['W', 'e', 'l', 'c', 'o', 'm', 'e', 't', 'o', 'l', 'N', 'S', 'T', 'O', 'O', 'O', '2', 'P', 'y', 't', 'h', 'o', 'n', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm',  
'i', 'n', 'g', 'l']
```

```
['My', 'current', 'age', 'is', '29', 'and', 'my', 'year', 'of', 'birth', 'is', '1994']
```

```
['', '', '', '', '', '', '', '*']
```

```
['da', 'da', 'da', 'da', 'da', 'da', 'da', 'da', 'd', 'd', 'd', 'd', 'daaaaaa']
```

# Regular Expression: Example

```
#!/usr/bin/env python

"""
A program to compute regular expression with substitution
"""

import re

# upon matching, 'y' is replaced by '!!' and CASE has been ignored
print(re.sub('y', '~*', 'My current age is 29 and my year of Birth is 1994', flags = re.IGNORECASE))

# Consider the Case Sensitivity, 'y' in "year", will not be replaced
print(re.sub('y', '!!', 'My current age is 29 year and my year of Birth is 1994.'))

# As count has been given value 1, the maximum time replacement occurs is 1
print(re.sub('y', '!!', 'My current age is 29 year and my year of Birth is 1994', count=1, flags=re.IGNORECASE))

# 'r' before the pattern denotes RE, \s is for start and end of a string
print(re.sub(r'\sAND\s', ' & ', 'TextBook and note book', flags=re.IGNORECASE))
```

# Regular Expression: Output

M~\* current age is 29 and m~\* ~\*ear of Birth is 1994  
M!! current age is 29 !!ear and m!! !!ear of Birth is 1994.  
M!! current age is 29 year and my year of Birth is 1994  
TextBook & note book

# Summary

# Summary

- We discussed strings and their various operations such as:
  - *concatenating*
  - *appending*
  - *multiplying etc.*
- We also discussed about *string formatting operator*, `format()` function, *formatter module* and their working procedures
- We discussed about *built-in string functions* such as *len()* and other operations were performed using functions such as *slice()*, *ord()* and *chr()*
- We discussed *comparing* and *iterating* a string
- We then looked at *regular expression* with some examples

# Important: Further Reading

- String type
- Read/Convert
- Indexing strings [ ]
- Slicing strings [2:4]
- String library
- String comparisons
- Searching in strings
- Concatenating strings with + etc.
- String operations
- Replacing text
- Stripping with space, etc
- Looping through strings with for and while loops



# Further Reading

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : By **Cay Horstmann & Rance Necaise**
  - Chapter 2 Programming with Numbers and Strings
- Learning Python (5<sup>th</sup> Edition): By **Mark Lutz**
  - Chapter 7 Strings Fundamentals
- Basic Core Python Programming: By **Meenu Kohli**
  - Chapter 4 Strings

# Next Lecture 6

Lecture 6 - Lists