
# UNIT 2

# Exception handling, Functions , Strings

Reference textbook :

An introduction to Python programming

Author: Gowrishankar S

# Types of errors

- There are at least two kinds of errors:

1. Syntax Errors (parsing errors)
2. Runtime errors or Exceptions
3. Semantic errors

- **Syntax errors** are produced by Python when it is translating the source code into byte code.
- They usually indicate that there is something wrong with the syntax of the program.
- Example:  Omitting the colon at the end of a def statement , while, for loop
- SyntaxError: invalid syntax.

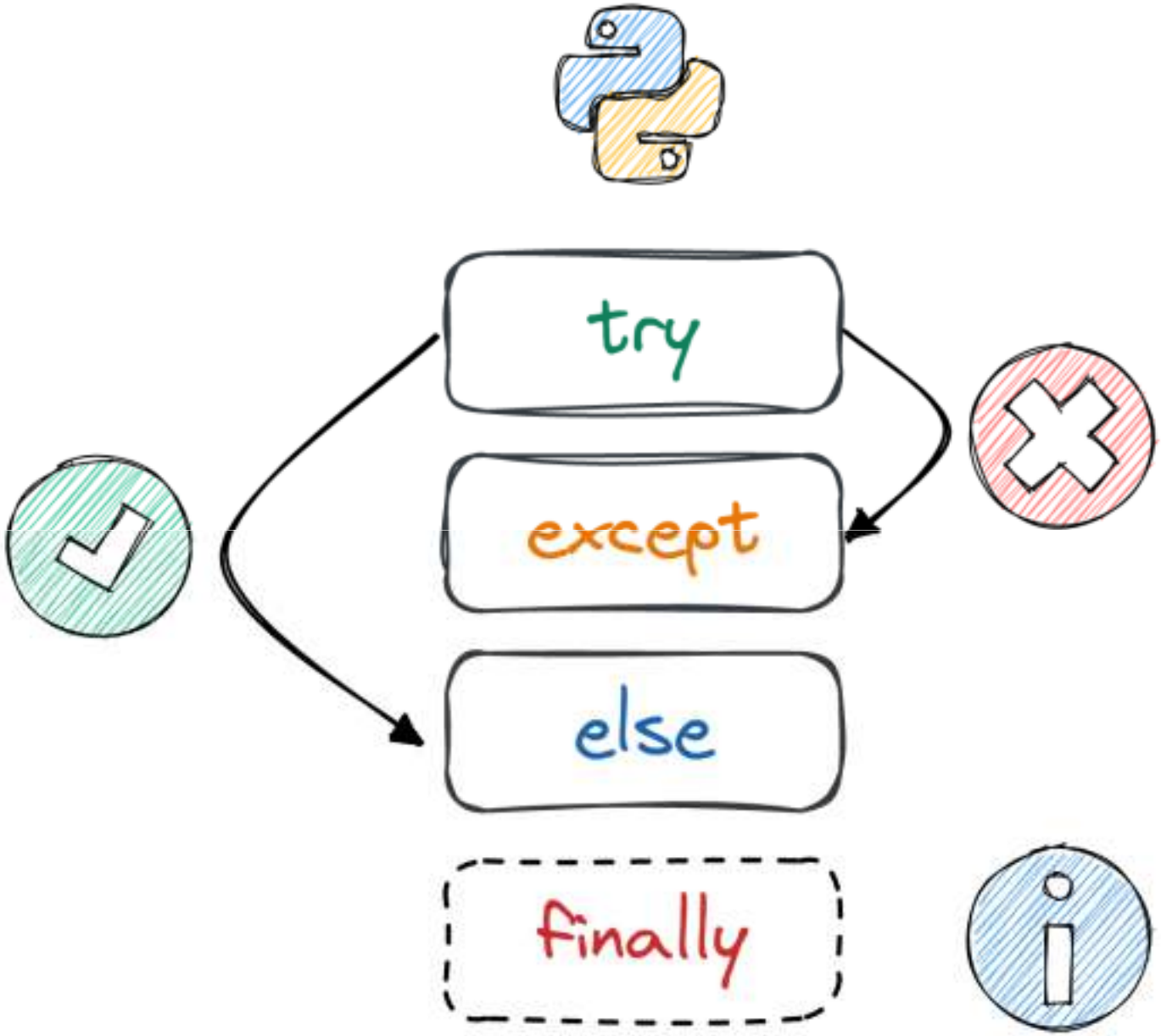
- **Runtime errors or exceptions** are produced by the runtime system if something goes wrong while the program is running.
- Most runtime error messages include information about where the error occurred and what functions were executing.
- Example: infinite loop,
- Example: An infinite recursion eventually causes a runtime error of “maximum recursion depth exceeded.”

- **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing.
- Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

# Exception Handling

- **An exception is an unwanted event that interrupts the normal flow of the program.**
- **Exception handling** is mechanism provided to handle the run-time errors or exceptions which often disrupt the normal flow of execution.
- Most common types of exceptions are: **TypeError, IndexError, NameError and ValueError.**
- When an exception is raised ,it causes program to terminate, we say that an unhandledexception has been raised

- There are 4 keywords used for **exception handling mechanism .** They are try, except, raise and finally.
- **Try block:** If you feel that block of code is going to cause an exception than such code you need to embed in a try block.
- **Except block** : Every try block should be followed by one or more except block(s). It is the place, where we catch the exception.
- **Raise:** In some cases, we need to manually raise an exception, its done through raise keyword followed by the exception name.
- **Finally:** The finally block will include house-keeping task or cleanup job. Its optional block, which always get executed.
- **Try-else block:** if the exception has not occurred or raised then the control will be redirected to else block after try block.


try
except
else
finally

- Exceptions can be either built-in exceptions ( created by interpreter) or user-defined exceptions (created by user)
- When the exceptions are not handled by programs it results in error messages.
- There can be multiple except blocks with different names which can be chained together.

Example of built in or system generated exceptions

1. >>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
**ZeroDivisionError:** division by zero

2. >>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
**NameError:** name 'spam' is not defined

3. >>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
**TypeError:** Can't convert 'int' object to str implicitly

# Example on valueError exception

```
while True:
 try:
    number = int(input("Please enter a number: "))
    print(f"The number you have entered is {number}")
    break
except ValueError:
    print("Oops! That was no valid number. Try again…")
```

# Functions in Python

- **Python Functions** is a structured block of statements that return the specific task.
- The common or repeated tasks are put together to make a function, so that for different inputs, we can call the function again and again to reuse code contained in it .
- Some **Benefits of Using Functions**
  - Increase Code Readability
  - Increase Code Reusability
  - Easy to manage and debug code

# Types of functions

1. **Built-in library function:** These are Standard system defined functions in Python that are available to use.

   ex: max(), min(), type(), len(), int(), float(), sin(), cos(), sqrt(), random() etc

2. **User-defined function:** We can create our own functions based on our requirements.

# Built in functions

- Modules in Python are reusable libraries of code having .py extension, which implements a group of methods and statements.
- Python comes with many built-in modules as part of the standard library.
- To use a module in your program, import the module using import statement.

Example: 1) import math

2) from math import sqrt, abs

- to use a module syntax is :

module_name.function_name()

example: math.sqrt(25)

- **the dir() function** will give a comma separated list of functions supported by a module

  example:  dir ( math)
- **help() function** will provide related help page for the module and its function

  example : help(math.gcd)
- **Third-party modules** or libraries can be installed and managed using Python's package manager pip.

  The syntax for pip is,

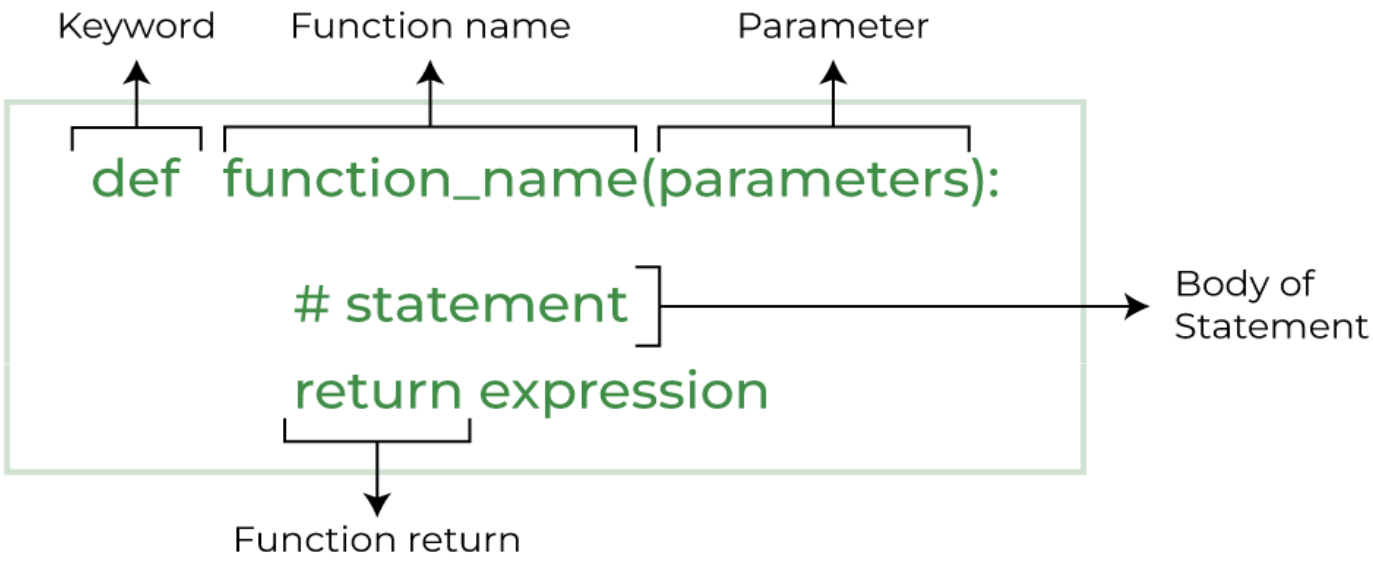
  pip install module_name

  example:  pip install arrow

  pip install  numpy

# User Defined Functions

## Defining a user defined function




Example: def myfun(a,b):
sum=a+b
return sum

# Calling a user defined function

function_name(argument_1,argument_2,…,argument_n)

Example:

```
a = int(input(Enter first number))
 b = int(input(Enter second number))
# below is a function call
c =  myfun(a,b)
print(“sum of two numbers is “, sum)
```

in Python, a function definition consists of the def keyword, followed by

1. The name of the function. The function's name has to adhere to the same naming rules as variables: use letters, numbers, or an underscore, but the name cannot start with a number. Also, you cannot use a keyword as a function name.
2. A list of parameters to the function are enclosed in parentheses and separated by commas.
3. A colon is required at the end of the function header.
4. Block of statements that define the body of the function start at the next line of the function header and they must have the same indentation level.

- Arguments are the actual value that is passed into the calling function. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function.
- A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function
- When you call a function, the control flows from the calling function to the function definition.
- Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement

- Before executing the code in the source program, the Python interpreter automatically defines a few special variables.
- If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in __name__ variable to have a string value of "__main__". ( note: it is double underscore __ )
- After setting up these special variables, the Python interpreter reads the program to execute the code found in it.
- This is the entry point of your program
- All of the code that is at indentation level 0 gets executed. Block of statements in the function definition is not executed unless the function is called.

```
if __name__ == "__main__":
    main()
```

**Fruitful function and void function**

- Fruitful function is something which returns a value.

- Void function is something which doesn't return a value.

```
def minnum(x, y):
    if(x<y):
        print("x is smaller")
    else:
        print("y is smaller")
x= int(input("enter first number"))
y= int(input("enter second number"))
minum(x ,y)
```

```
#example program on user defined functions
def function_definition_with_no_argument():
    print("This is a function definition with NO Argument")
def function_definition_with_one_argument(message):
    print(f"This is a function definition with {message}")
def main():
     function_definition_with_no_argument()
     function_definition_with_one_argument("One
    Argument")
if __name__ == "__main__":
    main()
```

```
#program to find area of a trapezium using function
def area_trapezium(a, b, h):
    area = 0.5 * (a + b) * h
    print(f"Area of a Trapezium is {area}")
def main():
    area_trapezium(10, 15, 20)
if __name__ == "__main__":
    main( )
```

# Unpacking (returning multiple values from a function )

- In some cases, it is necessary to return multiple values from a function if they are related to each other.
- This can be done by returning multiple values separated by a commas which by default is constructed as a tuple by Python.

  example:  return sum, avg
- Further , the calling function receives a tuple from the function definition and it assigns the result to multiple variables by specifying the same number of variables on the left-hand side of the assignment as there were returned from the function definition. This is called **tuple unpacking.**
- example:  sum, average = calculate_sum_avg ( )

```
#returning multiple values from a function
def  IPL():
    IPL_2023_winner = input("Which team won IPL 2023?")
    winning_captain_2023 = input("Who was their captain?")
    return  IPL_2023_winner,winning_captain_2023

def main():
     IPL_2023_winner,winning_captain_2023 = IPL()
     print(f"The IPL 2023 was won by {IPL_2023_winner} and
     their captain was {winning_captain_2023}")

if __name__ == "__main__":
    main()
```

# Scope and lifetime of variables

- Python programs have two scopes: global and local.
- The **lifetime** of a variable refers to the duration of its existence.

**Global variables:**

- A variable is a global if its value is accessible and modifiable throughout the program. They have a global scope.
- A global variable can be accessed inside a function provided it does not have the same name as that of the local variable.

**Local Variable:**

- A variable that is **defined inside a function** definition is a local variable.
- It is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition.
- They have local scope and exist as long as the function is executing.
- A local variable can have the same name as the global variable.

```
#program on scope of variables
test_variable = 5    # global
def outer_function():
    test_variable = 60   #local

def inner_function():
    test_variable = 100  #local
    print(f"Local variable value of {test_variable} having local scope to
    inner function is displayed")

inner_function()  #calling inner function

print(f"Local variable value of {test_variable} having local scope to
    outer function is displayed ")
outer_function()  #calling outer function

print(f"Global variable value of {test_variable} is displayed ")
```

# Nested functions & scope

- We can nest a function definition within another function definition.
- A nested function (inner function definition) can “inherit” the arguments and variables of its outer function definition.
- Hence, the inner function contains the scope of the outer function and can access variables & arguments of outer function.

```
#Calculate and Add the Surface Area of TwoCubes.
#Use Nested Functions

def add_cubes(a, b):
    def cube_surface_area(x):
        return 6 * pow(x, 2)
    return cube_surface_area(a) + cube_surface_area(b)

def main():
    len1=int(input("Enter length of cube 1 : "))
    len2=int(input("Enter length of cube 2 : "))
    result = add_cubes(len1, len2)
    print(f"The surface area after adding two Cubes is {result}")
if __name__ == "__main__":
    main()
```

# Default parameters

- Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value defined in the function definition.
- Any calling function must provide arguments for all required parameters in the function definition and can omit arguments for default parameters.
- Usually, the default parameters are defined at the end of the parameter list, after any required parameters

```
#example on default  parameters
def work_area(prompt, domain="Data Analytics"):
    print(f"{prompt} {domain}")
def main():
    work_area("Sam works in")
    work_area("Alice has interest in", "Internet of
    Things")
if __name__ == "__main__":
    main( )
```

# Example 2 on default parameters

```
def sum(a, b=10) :
        s=a+b
        print(“ sum is = “, s)

a = int(input(“enter a “))
b= int (input(“enter b “))
 sum(a)    #a=5    s =15
 sum(a,b) # a=5, b=6   s=11
```

# Keyword Arguments (kwargs)

- We can also send arguments with the ***key = value*** syntax.
- Hence , the order of the arguments (positional) does not matter in the calling function.
- Example:

```
def  simple_interest(P, T, R=3):
        SI = (P*T*R)/100
        print("simple interest is : ", SI)
simple_interest(T=5, P=10000, R=5)
simple_interest(T=5, P=10000)
Simple_interest(T, R, P)
```

# *args and **kwargs

- Sometimes the user will not know how many arguments will be passed to the calling function.

  **So, *args and **kwargs allows us to pass different number of arguments to the calling function.**
- ***args parameter** in function definition allows us to pass a non-keyworded, variable length **tuple** argument list to the calling function.
- ****kwargs** as parameter in function definition allows you to pass keyworded, variable length **dictionary** argument list to the calling function.
- *args must come after all the positional parameters and **kwargs must come right at the end

# Example on args and kwargs

```
def my_function(*kids, **kname):
  print("His last name is " + kname["lname"])
  print("The youngest child is " + kids[1])

def main():
    my_function("Emil", "Trever", "Linus", fname =
    “Trever", lname = "Refsnes")

if __name__ == "__main__":
    main()
```

# Command line arguments

- A Python program can accept any number of arguments from the command line.
- Command line arguments is a methodology in which user will give inputs to the program through the console using commands.
- We need to import sys module to access command line arguments.
- All the command line arguments in Python can be printed as a list of string by executing sys.argv

```
import sys
def main():
    print(f"the arguments at the command line are
    {sys.argv}")
    for arg in sys.argv:
        print(arg)
    print("No of arguments at command line are
    ",len(sys.argv))
if __name__ == "__main__":
    main()
```

# Recursive functions

- Recursion is a programming technique where-in function is called repeatedly until a condition is met.
- A function that calls itself is called as recursive.
- In general, a recursive definition is made up of two parts:
  - There is at least one base case that directly specifies result for a special case and stops the recursion
  - there is least one recursive case
- Example : factorial of a number ‘ n’

  -- base case is → if n == 1 then fact = 1

  -- recursive case is → n * fact(n-1)

```
#factorial using recursion
def factR(n):
    if n==1:
        return n
    else:
        return  n * factR(n-1)

m = int(input("Enter a number: "))
if m==0:
    print("Factorial of 0 is 1")
else:
    ans = factR(m)
    print("Factorial is ", ans)
```

- **Advantages of Recursion**
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.

- **Disadvantages of Recursion**
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.

```
#fibonacci using recursion
def fib(n):
    if n==0 or n==1:
        return n
    else:
        return fib(n-1) + fib(n-2)

def testFib(n):
    for i in range(n):
        print("fib of ", i, "=", fib(i))

n = int(input("Enter a number "))
testFib(n)
```

# STRINGS

**Creating and storing strings**

- Strings consist of one or more characters surrounded by matching quotation marks.
- example: str1=“ belgaum” or
  str2=‘belgaum’
  str3= “ I’m from belgaum “
  empty_str = “ “
- str() function: returns string version of the object
  example : str(10) returns ‘10’
  str() returns empty string

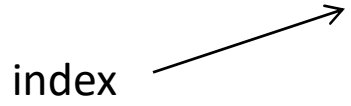
# Accessing String Characters using Index Number

- Each of the string's character corresponds to an index number starting from 0 (zero)
- The length of a string is the number of characters in it.
- We can access each character in a string using a subscript operator [ ]
- Example: str = " Belgaum"
- Hence str[0] = B , str[1] = e … and so on

| | B | e | l | g | a | u | m |
|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

- We can also access a string from backwards by using negative index . Hence we can access characters present at end of the string
- example: str[-1] = m , str[-2]= u … and so on

| B | e | l | g | a | u | m |
|---|---|---|---|---|---|---|
| -7 | -6 | -5 | -4 | -3 | -2 | -1 |



# String concatenation

- strings can also be concatenated using + sign and * operator is used to create a repeated sequence of strings.
- example:

```
string1 =“ face”
string2 = “book “
string3 = string1 + string2
print(string3)          # output is ‘facebook’
string4 = str(50) + “cent “
print(string4)          # output is  ‘50cent’
string5 = “ hi “ * 3
print(string5)          # output is  ‘hihihi’
check1= string1 in string3
print(check1)          #output is   True
check2 = "google" not in string3
print(check2)          # output is True
```

# String Comparison

- Python compares strings using ASCII value of the characters.
- We can use (>, <, <=, >=, ==, !=) to compare two strings which result in either Boolean True

  or False value.
- Example:

```
#string comparison
string1="january"
string2="jane"
print(string1 == string2)     #output is False
print(string1 != string2)     #output is True
print(string1 <= string2)     #output is False
```

# String Slicing and Joining

- With string slicing, we can access a sequence of characters (or part of a string or substring) by specifying a range of index numbers separated by a colon.
- It returns a sequence of characters (substring) beginning at start and extending up to but not including end.
- Syntax for slicing is :

**stringName [start : end [: step] ]**

where start , end → integers

- This step refers to the number of characters that can be skipped after the start index. The default value of step is one.
- we can also specify negative indexing for slice operation.

**#string slicing**

```
string1="belgaum"
#providing start and stop
print(string1[0:3])          # output is 'bel'
print(string1[:5])           # output is 'belga'
print(string1[3:])           # output is 'gaum'
print(string1[:])            # output is 'belgaum'
print(string1[3:10])         # output is 'gaum'
print(string1[-3:-1])        # output is 'au'
print(string1[5:-1])         # output is 'u'

# providing start , stop, step
print(string1[0:10:2])     #step 2 - output is 'blam'
print(string1[1:10:3])     #start 1 ,step 3 - output is 'ea'
print(string1[::4])          #start 0, step 4 - output is 'ba'
```

Determine if a string is palindrome using slicing

```
def main():
    user_string = input("Enter string: ")
    if user_string == user_string[::-1]:
        print(f"User entered string is palindrome")
    else:
        print(f"User entered string is not a
    palindrome")
if __name__ == "__main__":
    main()
```

# Joining Strings

- Strings can be joined with the join() string.
- Syntax : string_name.join(sequence)
  where sequence → string or list
- example:

1) date_of_birth = ["17", "09", "1950"]

**“ : “ . join(date_of_birth)**

output is '17:09:1950‘

2) str1 = [“Python", "is", “interesting”]

**" “ . join(str1)**

output is Python is interesting‘

3) num= 007 str3= “SPY”

**num . join (str3)**

output is S007P007Y007

# Splitting strings

- The split() method returns a list of string items by breaking up the string using the delimiter or seperator.
- Syntax is :

  string_name.split([separator [, maxsplit]])
- If the separator is not specified then whitespace is considered as the separator.
- If maxsplit is given, at most max-split splits are done.

Example on splitting

1) inventors = "edison, tesla, marconi, newton“

inventors.split(",")

#output is ['edison', ' tesla', ' marconi', ' newton']

2) watches = "rolex hublot cartier omega“

watches.split()

#output is ['rolex', 'hublot', 'cartier', 'omega']

# String Traversing

- Since the string is a sequence of characters, each of these characters can be traversed using the for loop.

```
def main():
    alphabet = "google"
    index = 0
    print(f ” the input string is  {alphabet} ")
    for each_character in alphabet:
        print(f “ Character {each_character} has an index value of {index} “ )
        index += 1
if __name__ == "__main__":
    main()
```

| Method | Description | Example |
|---|---|---|
| capitalize() | Converts the first character to upper case | txt = “hello world”<br>txt.capitalize()  # Hello World |
| casefold() | Converts string into lower case | txt = “Hello World”<br>txt.casefold()  # hello world |
| center() | Returns a centered string with length specified | txt = "banana"<br>txt.center(20, "O") |
| count() | Returns the number of times a specified value occurs in a string | txt = "I love apples, apple are my favorite fruit"<br>x = txt.count("apple") |
| endswith() | Returns true if the string ends with the specified value | txt = "Hello, welcome to my world."<br>x = txt.endswith("my world.") |
| index() | Searches the string for a specified value and returns the position of where it was found | txt = "Hello, welcome to my world."<br>x = txt.index("e", 5, 10) |
| isalnum() | Returns True if all characters in the string are alphanumeric | txt = "Company12“<br>x = txt.isalnum() |

| Method | Description | Example |
|---|---|---|
| isalpha() | Returns True if all characters in the string are in the alphabet | txt = "Company10“<br>x = txt.isalpha()  #false |
| isascii() | Returns True if all characters in the string are ascii characters | txt = "Company123“<br>x = txt.isascii() |
| isdecimal() | Returns True if all characters in the string are decimals | txt = "1234“<br>x = txt.isdecimal() |
| isdigit() | Returns True if all characters in the string are digits | txt = "50800“<br>x = txt.isdigit() |
| islower() | Returns True if all characters in the string are lower case | txt = "hello world!“<br>x = txt.islower() |
| isnumeric() | Returns True if all characters in the string are numeric | a = "\u0030" #unicode for 0<br>b= "1.5"<br>print(a.isnumeric())  #true<br>print(b.isnumeric())  #false |

| Method | Description | Example |
|---|---|---|
| isprintable() | Returns True if all characters in the string are printable | txt = "Hello! Are you #1?“<br>x = txt.isprintable()<br>print(x) |
| isspace() | Returns True if all characters in the string are whitespaces | txt = "  “<br>x = txt.isspace()<br>print(x) |
| istitle() | Returns True if the string follows the rules of a title | txt = "Hello, And Welcome “<br>x = txt.istitle()<br>print(x) |
| isupper() | Returns True if all characters in the string are upper case | txt = "THIS IS NOW!“<br>x = txt.isupper()<br>print(x) |
| join() | Converts the elements of an iterable into a string | myTuple = ("John", "Peter", "Vicky")<br>x = "#".join(myTuple)<br>print(x) |

| | |
|---|---|
| partition() | Returns a tuple where the string is parted into three parts |
| replace() | Returns a string where a specified value is replaced with a specified value |
| rfind() | Searches the string for a specified value and returns the last position of where it was found |
| rindex() | Searches the string for a specified value and returns the last position of where it was found |
| rjust() | Returns a right justified version of the string |
| rsplit() | Splits the string at the specified separator, and returns a list |

| | |
|---|---|
| split() | Splits the string at the specified separator, and returns a list |
| splitlines() | Splits the string at line breaks and returns a list |
| startswith() | Returns true if the string starts with the specified value |
| strip() | Returns a trimmed version of the string |
| swapcase() | Swaps cases, lower case becomes upper case and vice versa |
| title() | Converts the first character of each word to upper case |
| translate() | Returns a translated string |
| upper() | Converts a string into upper case |
| zfill() | Fills the string with a specified number of 0 values at the beginning |

# Formatting strings with f-strings

- The f-strings provide a way to embed expressions inside strings literals, using a minimal syntax.
- An f-string is a literal string, prefixed with 'f', which contains expressions within curly braces '{' and '}‘ and the expression evaluated at run time.
- example: print( f “ the results is {result} )

**Format specifiers for f-strings:**

- The syntax is :

  f'string_statements {variable_name [: {width}.{precision}]}’

Example

1) width = 10 , precision = 5 , value = 12.34567

```
print( f “result: {value:{width}.{precision}} “ )
    #output is   'result: 12.346'
print(f “result: {value:{width}}”)
    #output  is  'result: 12.34567'
print(f'result: {value:.{precision}}‘ )
    #output is  'result: 12.346'
```

# Escape sequences or control sequences

- they are used to give an alternate or new interpretation to the characters by escaping the original meaning.
- Example : \n, \r, \t, \\, \” , \b

List of Escape Sequences

| Escape Sequence | Meaning |
|---|---|
| \ | Break a Line into Multiple lines while ensuring the continuation of the line |
| \\ | Inserts a Backslash character in the string |
| \' | Inserts a Single Quote character in the string |
| \" | Inserts a Double Quote character in the string |
| \n | Inserts a New Line in the string |
| \t | Inserts a Tab in the string |
| \r | Inserts a Carriage Return in the string |
| \b | Inserts a Backspace in the string |
| \u | Inserts a Unicode character in the string |
| \0*oo* | Inserts a character in the string based on its Octal value |
| \x*hh* | Inserts a character in the string based on its Hex value |

# Raw Strings

- A raw string is created by prefixing the character r to the string.
- A raw string ignores all types of formatting within a string including the escape characters.
- Hence , by constructing a raw string you can retain quotes, backslashes in the output
- Example:

print ( r “ this is \” a raw string \n example

#output is this is \" a raw string \n example

# Unicode Strings

- The Unicode Standard provides a unique number for every character, no matter what platform, device, application or language.
- Hence this helps data to be transported easily without being corrupted.
- Unicode standards include UTF-8, UTF-16, UCS-2
- Regular Python strings are not Unicode: they are just plain bytes. To create a Unicode string, use the 'u' prefix on the string literal.
- Example :

  unicode_string = u'A unicode \u018e string \xf1‘

  print(unicode_string)