Department of Computer Science and Engineering
PES University, Bangalore, India

# Python For Computational Problem Solving (UE22CS151A)

**Mr. Prakash C O**
**Asst. Professor,**
**Dept. of CSE, PESU**
**coprakasha@pes.edu**

---

## Strings - str() : Immutable sequence(or array) of Unicode Characters

Python 3.x uses str objects to store textual data as immutable sequences of Unicode characters. Practically speaking, that means a str is an immutable array of characters.

Oddly enough, it's also a recursive data structure—each character in a string is itself a str object of length 1.

A string is a sequence of characters. Python directly supports a str type; but there is no character type.

Because strings are immutable in Python, modifying a string requires creating a modified copy.

**A string is a data structure with the following attributes.**

- A string has zero or more characters
- Each character of a string can be referred to by a index or a subscript
- An index is an integer
- Access to an element based on index or position takes the same time no matter where the element is in the string – random access
- Strings are immutable. Once created, we cannot change the number of elements – no append, no insert, no remove, no delete.
- Elements of the string cannot be assigned.
- A string can not grow and cannot shrink. Size of the string can be found using the function len.
- String is a sequence
- String is also iterable - is eager and not lazy.
- Strings cannot be nested.
- Strings can be sliced. This creates a new string.

There are 4 types of string literals or constants

### 1) Single quoted strings

In [2]:
```
s1 = 'Work "is" worship'
print(s1)
```

```
Work "is" worship
```

### 2) Double quoted strings

There is no difference between the two. In both these strings, escape sequences like \t, \n are expanded.

We can use double quotes in a single quoted string and single quote in double quoted string without escaping.

**These strings (single quoted and double quoted ) can span just a line – cannot span multiple lines.**

In [3]:
```
s2 = "I'm being respectful to my elders.\n"
print(s2)
s3 = "Respect your elders and the world will respect you."
print(s3)
```

```
I'm being respectful to my elders.

Respect your elders and the world will respect you.
```

## 3 Triple quoted strings:

In [23]:
```
s4 = '''\t Be loyal to those who are loyal to you.And respect everyone,
even your enemies and competition.\n'''
print(s4)
```

```
	 Be loyal to those who are loyal to you.And respect everyone,
even your enemies and competition.

```

In [5]:
```
s5 = """\t I have learned so many things from my parents and grandparents
about the right upbringing, the right values, value for money, value for
elders, for family members. I think these things only a parent can teach
you."""
print(s5)
```

```
	 I have learned so many things from my parents and grandparents
about the right upbringing, the right values, value for money, value for
elders, for family members. I think these things only a parent can teach
you.
```

## Examples for Triple quoted strings : docstring

docstring:

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the **doc** attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

An object's docstring is defined by including a string constant as the first statement in the object's definition. It's specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments the docstring should describe what the function does, not how. All functions should have a docstring.

This allows the program to inspect these comments at run time, for instance as an interactive help system, or as metadata. Docstrings can be accessed by the

**__doc__**

attribute on objects.

### Declaration of docstrings

The following Python file shows the declaration of docstrings within a python source file:

In [ ]:
```
"""
Assuming this is file mymodule.py, then this string,
being the first statement in the file, will become the
"mymodule" module's docstring when the file is imported.
"""
class MyClass(object):
    """The class's docstring"""
     def my_method(self):
        """The method's docstring"""

def my_function():
    """The function's docstring"""
```

### How to access the Docstring

In [ ]:
```
The following is an interactive session showing how the docstrings may be accessed
>>> import mymodule
>>> help(mymodule)
Assuming this is file mymodule.py then this string, being the first statement in the fil
>>> help(mymodule.MyClass)
The class's docstring

>>> help(mymodule.MyClass.my_method)
The method's docstring

>>> help(mymodule.my_function)
The function's docstring
```

In [4]:
```
s4 = """
we love python
very much """
print('type(s4) =', type(s4))
print("document string : ", __doc__)
```

```
type(s4) = <class 'str'>
document string :  Automatically created module for IPython interactive environment
```

## 4) Raw strings

There are cases where the escape sequence should not be expanded – we require such strings as patterns in pattern matching using regular expressions. In such cases, we prefix r to the string literal – it becomes a raw string.

Note:A regular expression is a sequence of characters that specifies a search pattern in text.

In [25]:
```
# raw string, no escaping
s3 = r"this is a \n string"
print(s3)
print(type(s3))
```

```
this is a \n string
<class 'str'>
```

## A string has the following attributes.

- sequence
- indexed

- leftmost index : 0
- immutable
- no character type
- can be sliced
- cannot assign

#### Creating empty strings

In [26]:
```
str1=''
print(str1)
```

In [27]:
```
str2 = str()
print(str2)
```

#### Python has no character data type.

In [7]:
```
ch='A'
print("type('A') =",type(ch))
print("ord('A') =",ord(ch))
```

```
type('A') = <class 'str'>
ord('A') = 65
```

#### String is a sequence(ordered) and can be indexed.

Each character in a string is itself a str object of length 1.

In [9]:
```
str1 = "Good Work"
print(str1, type(str1))
print(str1[0], type(str1[0]))
print(str1[1], type(str1[1]))
print(str1[2], type(str1[2]))
print(str1[3], type(str1[3]))
```

```
Good Work <class 'str'>
G <class 'str'>
o <class 'str'>
o <class 'str'>
d <class 'str'>
```

#### String is Iterable

In [11]:
```
str2 = "python"
for char in str2:
    print(char)  # each char is a single letter string
```

```
p
y
t
h
o
n
```

In [12]:
```
str2 = "python"
i=0
while i<len(str2):
    print(str2[i])
    i=i+1
```

```
p
y
```

```
t
h
o
n
```

**String is immutable**

In [ ]:
```
str3 = 'cat'
str3[0] = 'b' #TypeError: 'str' object does not support item assignment
```

**String is hashable**

In [13]:
```
print(hash('python'))
```

```
5867849622804858424
```

**Build a string in stages**

In [ ]:
```
str4 = ''           # create an empty string
str4 = str4 + 'do'          # create a new string by concatenation
str4 = str4 + 'something'   # create a new string by concatenation
```

## Python String Methods

Python has a set of built-in methods that you can use on strings.

**Note: All string methods returns new values. They do not change the original string.**

| Method | Description |
|---|---|
| capitalize() | Converts the first character to upper case |
| casefold() | Converts string into lower case |
| center() | Returns a centered string |
| count() | Returns the number of times a specified value occurs in a string |
| encode() | Returns an encoded version of the string |
| endswith() | Returns true if the string ends with the specified value |
| expandtabs() | Sets the tab size of the string |
| find() | Searches the string for a specified value and returns the position of where it was found |
| format() | Formats specified values in a string |
| format_map() | Formats specified values in a string |
| index() | Searches the string for a specified value and returns the position of where it was found |
| isalnum() | Returns True if all characters in the string are alphanumeric |
| isalpha() | Returns True if all characters in the string are in the alphabet |
| isdecimal() | Returns True if all characters in the string are decimals |
| isdigit() | Returns True if all characters in the string are digits |
| isidentifier() | Returns True if the string is an identifier |
| islower() | Returns True if all characters in the string are lower case |
| isnumeric() | Returns True if all characters in the string are numeric |
| isprintable() | Returns True if all characters in the string are printable |
| isspace() | Returns True if all characters in the string are whitespaces |
| istitle() | Returns True if the string follows the rules of a title |
| isupper() | Returns True if all characters in the string are upper case |
| join() | Joins the elements of an iterable to the end of the string |
| ljust() | Returns a left justified version of the string |
| lower() | Converts a string into lower case |

---

| | |
|---|---|
| lstrip() | Returns a left trim version of the string |
| maketrans() | Returns a translation table to be used in translations |
| 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 |
| rpartition() | Returns a tuple where the string is parted into three parts |
| rsplit() | Splits the string at the specified separator, and returns a list |
| rstrip() | Returns a right trim version of the string |
| 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 |

**split([separator[, maxsplit]])**

Returns a list of strings after breaking the given string by the specified separator.

Parameter Values:

- separator is optional. This is a delimiter and specifies the separator to use when splitting the string. By default any whitespace is a separator
- maxsplit is optional. It is a number and specifies how many splits to do. Default value is -1, which is "all occurrences"

Note: When maxsplit is specified, the list will contain the specified number of elements plus one.

In [81]:
```
str1 = "Respect your elders and the world will respect you."
words_list = str1.split()
print(words_list)
words_list = str1.split(' ',0)
print(words_list)
words_list = str1.split(' ',1)
print(words_list)
words_list = str1.split(' ',2)
print(words_list)
```

```
['Respect', 'your', 'elders', 'and', 'the', 'world', 'will', 'respect', 'you.']
['Respect your elders and the world will respect you.']
['Respect', 'your elders and the world will respect you.']
['Respect', 'your', 'elders and the world will respect you.']
```

In [13]:
```
#Split the string, using \n, as a separator:
str2 = """All that glitters is not gold.
A picture is worth a thousand words.
Better safe than sorry.
Every cloud has a silver lining.
A journey of a thousand miles begins with a single step.
Necessity is the mother of invention.
The pen is mightier than the sword.
Time waits for no one."""
```

```
words_list = str2.split('\n')
print(words_list)
```

```
['All that glitters is not gold.', 'A picture is worth a thousand words.', 'Better safe than sorry.', 'Every cloud has a silver lining.', 'A journey of a thousand miles begins with a single step.', 'Necessity is the mother of invention.', 'The pen is mightier than the sword.', 'Time waits for no one.']
```

In [14]:
```
#Split the string, using comma, followed by a space, as a separator:
str3 = "hello, my name is Raj, I am 26 years old"
words_list = str3.split(', ')
print(words_list)
```

```
['hello', 'my name is Raj', 'I am 26 years old']
```

In [16]:
```
# Use a hash character as a separator:
str4 = "apple#banana#cherry#orange"
words_list = str4.split('#')
print(words_list)
```

```
['apple', 'banana', 'cherry', 'orange']
```

**rsplit([separator[, maxsplit]])**

The rsplit() method splits a string into a list, starting from the right. If no "max" is specified, this method will return the same as the split() method.

Note: When max is specified, the list will contain the specified number of elements plus one.

In [82]:
```
str4 = "apple, banana, cherry, orange"
words_list = str4.rsplit(', ')
print(words_list)
words_list = str4.rsplit(', ',1)
print(words_list)
words_list = str4.rsplit(', ',2)
print(words_list)
words_list = str4.rsplit(', ',3)
print(words_list)
```

```
['apple', 'banana', 'cherry', 'orange']
['apple, banana, cherry', 'orange']
['apple, banana', 'cherry', 'orange']
['apple', 'banana', 'cherry', 'orange']
```

In [8]:
```
str4 = "apple, banana, cherry, orange"
words_list = str4.rsplit(', ',1)
print(words_list, len(words_list))
words_list = str4.rsplit(', ',2)
print(words_list, len(words_list))
words_list = str4.rsplit(', ',3)
print(words_list, len(words_list))
```

```
['apple, banana, cherry', 'orange'] 2
['apple, banana', 'cherry', 'orange'] 3
['apple', 'banana', 'cherry', 'orange'] 4
```

**upper() - converts alphabetic characters to uppercase.**

s1.upper() returns a new string with all alphabetic characters in s1 converted to uppercase.

In [19]:
```
s1 = 'python 3.11'
s2 = s1.upper()    # .upper() returns new string
print('s2 =',s2)
print('s1 =',s1)   # s1 is not modified, bcoz strings are immutable
```

```
s2 = PYTHON 3.11
s1 = python 3.11
```

In case we want the original string to change, assign the result of the function call back to the same variable – thus recreating the variable.

In [8]:
```
s1 = 'python 3.11'
print('s1 =',s1)
s1 = s1.upper()
print('s1 =',s1)
```

```
s1 = python 3.11
s1 = PYTHON 3.11
```

**lower() - converts alphabetic characters to lowercase.**

s1.lower() returns a new string with all alphabetic characters in s1 converted to lowercase.

In [14]:
```
s1 = 'PYTHON 3.11'
print('s1 =',s1)
s1 = s1.lower()
print('s1 =',s1)
```

```
s1 = PYTHON 3.11
s1 = python 3.11
```

**casefold()**

The casefold() method is similar to the lower() method but it is more aggressive. This means the casefold() method converts more characters into lower case compared to lower() .

For example, the German letter ß is already lowercase so, the lower() method doesn't make the conversion.

But the casefold() method will convert ß to its equivalent character ss in english.

In [32]:
```
# The German alphabet has 26 letters(A to Z), a ligature (ß) and
#   3 umlauts Ä, Ö, Ü.

text = 'ÄÖÜß'
# convert text to lowercase using lower()
print('Using lower():', text.lower())

# convert text to lowercase using casefold()
print('Using casefold():', text.casefold())
```

```
Using lower(): äöüß
Using casefold(): äöüss
```

In [13]:
```
text = 'groß'

# convert text to lowercase using casefold()
print('Using casefold():', text.casefold())

# convert text to lowercase using lower()
print('Using lower():', text.lower())
```

```
Using casefold(): gross
Using lower(): groß
```

**Exercise 01:**

Print 'm K gandhi' from given string 'mohanDas Karamchand gandhi'

In [11]:
```
# Approach 1
name='mohanDas Karamchand gandhi'
```

```
namewords_list=name.split()
print(namewords_list[0][0],namewords_list[1][0],namewords_list[2])
```

```
m K gandhi
```

In [12]:
```
# Approach 2
name='mohanDas Karamchand gandhi'
namewords_list=name.split()
initials_name =''
for word in namewords_list[:-1]:
    initials_name = initials_name + word[0] +' '
initials_name = initials_name + namewords_list[-1]
print(initials_name)
```

```
m K gandhi
```

**title() - converts the target string to “title case”**

s1.title() returns a new string of s1 in which the first letter of each word is converted to uppercase and remaining letters are lowercase.

In [8]:
```
name1 = 'm K gandhi'
tc_name = name1.title()
print(tc_name)
```

```
M K Gandhi
```

In [84]:
```
name2 = 'mohanDAS KaramChand gandhi'
tc_name = name2.title()
print(tc_name)
```

```
Mohandas Karamchand Gandhi
```

In [21]:
```
s1="what's happened to ted's IBM stock?".title()
print(s1)
```

```
What'S Happened To Ted'S Ibm Stock?
```

This method uses a fairly simple algorithm. It does not attempt to distinguish between important and unimportant words, and it does not handle apostrophes, possessives, or acronyms gracefully

**endswith(suffix[, start[, end]])**

Determines whether the target string ends with a given substring.

s1.endswith(suffix) returns True if string s1 ends with the specified suffix and False otherwise.

Parameters:

- suffix: Suffix is nothing but a string that needs to be checked.
- start: Starting position from where suffix is needed to be checked within the string.
- end: Ending position + 1 from where suffix is needed to be checked within the string.

Note: A suffix is a word part added to the end of a word.

In [22]:
```
print('Subhash Chandra Bose'.endswith('Bose'))
print('Chandra Shekhar Azad'.endswith('Singh'))
```

```
True
False
```

The comparison is restricted to the substring indicated by start and end, if they are specified:

In [87]:
```
print('Subhash Chandra Bose'.endswith('ash',0,7))
print('Chandra Shekhar Azad'.endswith('Azad',0,15))
```

```
True
False
```

**Exercise 02:**

Print names endswith patel from the given name list

In [5]:
```
name_list = [
"mahatma gandhi",
"subhash chandra bose",
"sardar patel",
"brijesh patel",
"bhagat singh",
"Chandra Shekhar Azad",
"J H patel" ]

for name in name_list:
    if name.endswith('patel') :
        print(name)
```

```
sardar patel
brijesh patel
J H patel
```

**startswith(prefix[,start[,end]])**

Determines whether the target string starts with a given substring.

When you use the Python .startswith() method, s1.startswith(prefix) returns True if s1 starts with the specified prefix and False otherwise.

Parameters:

- prefix : The string to be searched.
- start : start index of the str from where the search_string is to be searched.
- end : end index of the str, which is to be considered for searching.

In [27]:
```
print('foobar'.startswith('foo'))
print('foobar'.startswith('bar'))
```

```
True
False
```

The comparison is restricted to the substring indicated by start and end, if they are specified:

In [12]:
```
print('foobar'.startswith(''))
print('foobar'.startswith('bar', 3))
print('foobar'.startswith('bar', 3,2))
```

```
True
True
False
```

**find(sub[, start[, end]])**

s.find(sub) returns

- the lowest index in s where substring sub is found.
- -1 if the value is not found.

The find() method is almost the same as the index() method, the only difference is that the index() method raises an exception if the value is not found. (See example below)

Parameter Values:

| Parameter | Description |
|---|---|
| *value* | Required. The value to search for |
| *start* | Optional. Where to start the search. Default is 0 |
| *end* | Optional. Where to end the search. Default is to the end of the string |

In [13]:
```
my_str = "Hello, welcome to my world."
wc_index = my_str.find("welcome")
print(wc_index)

wc_index = my_str.index("welcome")
print(wc_index)
```

```
7
7
```

In [89]:
```
my_str = "Hello, welcome to my world."
wc_index = my_str.find("Hi")
print(wc_index)

#wc_index = my_str.index("Hi") #ValueError: substring not found
#print(wc_index)
```

```
-1
```

In [35]:
```
my_str = "Hello, welcome to my world."
x1 = my_str.find("wel", 5, 10)
x2 = my_str.find("ful", 5, 10)
print(x1)
print(x2)

x2 = my_str.index("wel", 5, 10)
print(x2)
#x3 = my_str.index("ful", 5, 10) # ValueError: substring not found
#print(x3)
```

```
7
-1
7
```

**index(sub[, start[, end]])**

s.index(sub) returns the lowest index in s where substring sub is found. This method is identical to .find(), except that it raises an exception if sub is not found rather than returning -1.

In [ ]:
```
# ValueError: substring not found
print('foo bar foo baz foo qux'.index('grault'))
```

**Exercise 03:**

Find the leftmost index of word bad.

In [21]:
```
str1 = "bad nation bad culture bad people"

print(str1.index('bad')) #searches the string for a specified value and returns the posi
```

```
0
```

Find the second bad word index from left.

In [34]:
```
str1 = "bad nation bad culture bad bad people"
print(str1.index('bad', str1.index('bad') + len('bad')))   #str.index(value,start,end)
```

```
11
```

**replace(oldvalue, newvalue[,count])**

Parameter Values:

- oldvalue is required - the string to search for
- newvalue is required - the string to replace the old value with
- count is optional - A number specifying how many occurrences of the old value you want to replace. Default is all occurrences

In [20]:
```
str1 = "bad nation bad culture bad people"
print(str1.replace('bad', 'good')) # default : all occurrences
print(str1.replace('bad', 'good', 2))
```

```
good nation good culture good people
good nation good culture bad people
```

**Exercise 04:**

str1 = "bad nation bad culture bad people"

1. Extract the substring 'bad culture bad people' from str1 and replace the first occurance of bad by worst.
2. Replace the second occurance of bad by worst in str1
3. Replace the third occurance of bad by worst in str1

In [35]:
```
#1. Extract the substring 'bad culture bad people' from str1 and
#    replace the first occurance of bad by worst.
str1 = "bad nation bad culture bad people"
i = str1.index('bad', str1.index('bad') + len('bad'))
str2 = str1[i:]  # str2 = 'bad culture bad people'
print(str2.replace('bad', 'worst', 1)) #str.replace(newvalue,oldvalue,count)
```

```
worst culture bad people
```

In [23]:
```
#2. Replace the second occurance of bad by worst in str1
str1 = "bad nation bad culture bad people"
i = str1.index('bad', str1.index('bad') + len('bad'))
str2 = str1[:i] # str2 = 'bad nation '
str3 = str1[i:] # str3 = 'bad culture bad people'
print( str2 + str3.replace('bad', 'worst', 1))
```

```
bad nation worst culture bad people
```

In [30]:
```
#3. Replace the third occurance of bad by worst in str1
str1 = "bad nation bad culture bad people"
i = str1.index('bad', str1.index('bad', str1.index('bad')+len('bad')) + len('bad'))
print(i)
str2 = str1[:i] # str2 = 'bad nation bad culture '
str3 = str1[i:] # str3 = 'bad people'
print( str2 + str3.replace('bad', 'worst'))
```

```
23
bad nation bad culture worst people
```

**Exercise 05:**

str1 = "we love python very much"

1. Print the first letter of each word at the end of the word.
2. Print each word in reverse.
3. Print * after each character.

In [36]:
```
#1. Print the first letter of each word at the end of the word.
str1 = "we love python very much"
for word in str1.split():
    print(word[1:]+word[0], end=' ')
```

```
ew ovel ythonp eryv uchm
```

In [46]:
```
#2. Print each word in reverse.
str1 = "we love python very much"
for word in str1.split():
    print(word[::-1], sep='' , end=' ')
```

```
ew evol nohtyp yrev hcum
```

In [91]:
```
#3. Print * after each character.
str1 = "we love python very much"
for ch in str1:
    print(ch, end = "*")

print()
print(str1.replace('','*'))
```

```
w*e* *l*o*v*e* *p*y*t*h*o*n* *v*e*r*y* *m*u*c*h*
*w*e* *l*o*v*e* *p*y*t*h*o*n* *v*e*r*y* *m*u*c*h*
```

**capitalize() Method**

Python String capitalize() method returns a copy of the original string and converts the first character of the string to a capital (uppercase) letter, while making all other characters in the string lowercase letters.

Parameter: The capitalize() function does not takes any parameter.

In [47]:
```
#The capitalize() function returns a string with the first character in the capital.
txt = "hello, and welcome to my world."
cap_txt = txt.capitalize()
print(cap_txt)
```

```
Hello, and welcome to my world.
```

**count(sub[, start[, end]])**

Counts occurrences of a substring in the target string.

Parameter Values:

- sub is required. A substring to search for
- start is optional. An Integer. The position to start the search. Default is 0
- end is optional. An Integer. The position to end the search. Default is the end of the string.

In [49]:
```
# Return the number of times the value "apple" appears in the string:

str1 = "I love apples, apple are my favorite fruit"
apple_count = str1.count("apple")
print(apple_count)
apple_count = str1.count("apple",15,31)
print(apple_count)
apple_count = str1.count("apple",10,24)
print(apple_count)
```

```
2
1
1
```

#### center(length[,character])

The center() method will center align the string, using a specified character (space is default) as the fill character.

Parameter Values:

- length is required. The length of the returned string
- character is optional. The character to fill the missing space on each side. Default is " " (space)

In [93]:
```
str1 = "banana"
print(str1.center(20))
print(str1.center(19, "*"))
```

```
       banana
*******banana******
```

#### strip([string_of_characters])

The strip() method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (space is the default leading or trailing character to remove)

- string_of_characters is optional. A set of characters to remove as leading/trailing characters

In [18]:
```
str1 = '     good work     '
print(str1.strip())
```

```
good work
```

In [95]:
```
str2 = ",,,,,rrttgg.....banana....rrr"
print(str2.strip(",.grt"))
```

```
banana
```

## String Formatting

Methods in this group modify or enhance the format of a string.

#### format() Method

- The format() method formats the specified value(s) and insert them inside the string's placeholder.
- The placeholder is defined using curly brackets: {}.
- The format() method returns the formatted string.

Syntax: string.format(value1, value2...)

Parameter Values:

| Parameter | Description |
|---|---|
| *value1, value2...* | Required. One or more values that should be formatted and inserted in the string.<br><br>The values are either a list of values separated by commas, a key=value list, or a combination of both.<br><br>The values can be of any data type. |

In [40]:
```
# using format option in a simple string
print("{}, is Computer Science Portal".format("CSEStack"))

# using format option for a value stored in a variable
str = "This article is written in {}"
print(str.format("Python"))

# formatting a string using a numeric constant
print("Hello, I am {} years old !".format(40))
#print("Hello, I am {} years old !".format(40,50,60))
```

```
CSEStack, is Computer Science Portal
This article is written in Python
Hello, I am 40 years old !
```

**The Placeholders (The replacement fields)**

The placeholders can be identified using

- named indexes(also called keyword arguments) -> {price}
- numbered indexes -> {0}
- even empty placeholders -> {}

Keyword_argument is essentially a variable storing some value, which is passed as parameter.

In case of multiple empty placeholders, Python will replace the placeholders with values in order.

In [101…
```
txt0 = "My name is {}, I'm {}".format("John",36,67)
print(txt0)
txt1 = "My name is {fname}, I'm {age}".format(fname="John",age=36)
txt2 = "My name is {0}, I'm {1}".format("John",36)
txt3 = "My name is {fname}, I'm earning {salary:b}".format(fname="John",salary=360000)
print(txt1)
print(txt2)
print(txt3)
```

```
My name is John, I'm 36
My name is John, I'm 36
My name is John, I'm 36
My name is John, I'm earning 1010111111001000000
```

In [63]:
```
#Insert the price inside the placeholder,
 # the price should be in fixed point, two-decimal format:
price = 49.6789
print("For only {price:.2f} dollars!".format(price = 49.6789))
print("For only {price:8.2f} dollars!".format(price = 49.6789))
```

```
For only 49.68 dollars!
For only    49.68 dollars!
```

**Specifying Type**

We can include more parameters within the curly braces of our syntax. We'll use the format code syntax

**{field_name:conversion}, where**

- field_name specifies the index number or argument of the str.format() method that we went through in the reordering section, and
- conversion refers to the conversion_specification/formatting_type (and/or width) of the data type that you're using with the formatter.

**conversion specification types**

- :b Binary format
- :c Converts the value into the corresponding unicode character
- :d Decimal format
- :e Scientific format, with a lower case e
- :E Scientific format, with an upper case E
- :f Floating point number format
- :F Floating point number format, in uppercase format (show inf and nan as INF and NAN)
- :g General format
- :G General format (using a upper case E for scientific notations)
- :o Octal format
- :x Hexadecimal format, lower case
- :X Hexadecimal format, upper case
- :n Number format
- :u unsigned decimal integer format
- :% Percentage format

**Formatting types**

Inside the placeholders you can add a formatting type to format the result:

- :< Left aligns the result (within the available space)
- :> Right aligns the result (within the available space)
- :^ Center aligns the result (within the available space)
- := Places the sign to the left most position
- :+ Use a plus sign to indicate if the result is positive or negative
- :- Use a minus sign for negative values only
- : Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
- :, Use a comma as a thousand separator
- :_ Use a underscore as a thousand separator

In [125…
```
txt0 = "My name is {:10}, I'm {:10}".format("John",36,67)
print(txt0)
txt1 = "My name is {fname:^10}, I'm {age:10}".format(fname="John",age=36)
txt2 = "My name is {0:<10}, I'm {1:<10}".format("John",36)
txt3 = "My name is {:>10}, I'm earning {:}".format("John",360000)
print(txt1)
print(txt2)
print(txt3)
```

```
My name is John      , I'm         36
My name is    John   , I'm         36
My name is John      , I'm 36
My name is       John, I'm earning 360000
```

**Errors and Exceptions :**

IndexError :

- Occurs when string has an extra placeholder, and we didn't pass any value for it in the format() method.
- Python usually assigns the placeholders with default index in order like 0, 1, 2, 3.... to access the values passed as parameters. So when it encounters a placeholder whose index doesn't have any value passed inside as parameter, it throws IndexError.

In [107…
```
txt3 = "My name is {}, I'm {}, I'm {}".format("John",36)
```

```
# IndexError: Replacement index 2 out of range for positional args tuple
```

In [56]:
```
#Use "<" to left-align the value:
txt = "We have {:>8} chickens."
print(txt.format(49))
```

```
We have       49 chickens.
```

In [55]:
```
#Use "=" to place the plus/minus sign at the left most position:
txt = "The temperature is {:=8} degrees celsius."
print(txt.format(-5))
```

```
The temperature is -      5 degrees celsius.
```

In [118…
```
#Use "," to add a comma as a thousand separator:
txt = "The universe is {:,} years old."
print(txt.format(13800000000))
```

```
The universe is 13,800,000,000 years old.
```

In [53]:
```
#Use "b" to convert the number into binary format:
print("The decimal version of {0} is {1:d}".format(1010,0b1010))

txt = "The binary version of {0} is {0:b}"
print(txt.format(10))
```

```
The decimal version of 1010 is 10
The binary version of 10 is 1010
```

In [70]:
```
#Use "d" to convert a number, in this case a binary number, into decimal number format:
txt = "We have {:d} chickens."
print(txt.format(0b101))
```

```
We have 5 chickens.
```

In [15]:
```
#Use "%" to convert the number into a percentage format:
txt = "You scored {:%}"
print(txt.format(0.25))
#Or, without any decimals:
txt = "You scored {:.0%}"
print(txt.format(0.25))
```

```
You scored 25.000000%
You scored 25%
```

## Character Classification

Methods in this group classify a string based on the characters it contains.

#### isalnum()

Determines whether the target string consists of alphanumeric characters.

s1.isalnum() returns True if s1 is nonempty and all its characters are alphanumeric (either a letter or a number), and False otherwise:

In [49]:
```
# isalnum() Method
# Check if all the characters in the text are alphanumeric:
txt1 = "PESU2021"
x1 = txt1.isalnum()
print(x1)
txt2 = "PESU 2021"
x2 = txt2.isalnum()
print(x2)
```

```
True
False
```

#### isalpha()

Determines whether the target string consists of alphabetic characters.

s1.isalpha() returns True if s1 is nonempty and all its characters are alphabetic, and False otherwise.

In [8]:
```
# isalpha() Method
# Check if all the characters in the text are letters:
txt1 = "PESUniversity"
print(txt1.isalpha())
txt2 = "Pesu2021"   # not alphabet letters:!#%&?, space character,....
print(txt2.isalpha())
```

```
True
False
```

#### isdecimal()

s1.isdecimal() returns true if all characters in a string s1 are decimal. If all characters are not decimal then it returns false.

In [6]:
```
s = "12345"
print(s,'.isdecimal() -> ',s.isdecimal(),sep='')

# contains alphabets
s = "12sHello34"
print(s,'.isdecimal() -> ',s.isdecimal(),sep='')

# contains numbers and spaces
s = "12 34"
print(s,'.isdecimal() -> ',s.isdecimal(),sep='')

# contains .
s = "12.34"
print(s,'.isdecimal() -> ',s.isdecimal(),sep='')
```

```
12345.isdecimal() -> True
12sHello34.isdecimal() -> False
12 34.isdecimal() -> False
12.34.isdecimal() -> False
```

In [5]:
```
s1 = "\u0030" #unicode for 0 in base 16
s2 = "\u0047" #unicode for G in base 16
# Unicode is the superset of ASCII because it encodes more characters.
print(s1,'.isdecimal() -> ',s1.isdecimal(),sep='')
print(s2,'.isdecimal() -> ',s2.isdecimal(),sep='')
```

```
0.isdecimal() -> True
G.isdecimal() -> False
```

#### isdigit()

Determines whether the target string consists of digit characters.

You can use the isdigit() Python method to check if your string is made of only digits.

s1.isdigit() returns True if s1 is nonempty and all its characters are numeric digits, and False otherwise:

In [6]:
```
# isdigit() Method
# Returns True if all characters in the string are digits
txt = "508"
print(txt,'.isdigit() -> ',txt.isdigit(), sep='')
```

```
txt = '2gether'
print(txt,'.isdigit() -> ',txt.isdigit(), sep='')

txt = '22,000'
print(txt,'.isdigit() -> ',txt.isdigit(), sep='')
```

```
508.isdigit() -> True
2gether.isdigit() -> False
22,000.isdigit() -> False
```

In [4]:
```
x1=chr(48) # x1='0'
print(x1,".isdigit -> ",x1.isdigit(), sep='')

x2=chr(65) # x2='A'
print(x2,".isdigit -> ",x2.isdigit(), sep='')

x3=chr(97) # x3='a'
print(x3,".isdigit -> ",x3.isdigit(), sep='')
```

```
0.isdigit -> True
A.isdigit -> False
a.isdigit -> False
```

#### isnumeric() Method

In Python, decimal characters (like: 0, 1, 2..), digits (like: subscript, superscript), and characters having Unicode numeric value property (like: fraction (1/2, 3/4, ..), roman numerals, currency numerators) are all considered numeric characters.

List of Unicode Characters with Decomposition Mapping “fraction form”

- U+00BC. ¼ Fraction One Quarter.
- U+00BD. ½ Fraction One Half.
- U+00BE. ¾ Fraction Three Quarters.
- U+2150. ⅐ Fraction One Seventh.
- U+2151. ⅑ Fraction One Ninth.
- U+2152. ⅒ Fraction One Tenth.
- U+2153. ⅓ Fraction One Third.
- U+2154. ⅔ Fraction Two Thirds.

Unicode Block “Superscripts and Subscripts”

- U+2070. ⁰ Superscript Zero.
- U+2071. ⁱ Superscript Latin Small Letter I.
- U+2074. ⁴ Superscript Four.
- U+2075. ⁵ Superscript Five.

In [9]:
```
# The isnumeric() method returns True if all the characters are numeric (0-9), otherwise
s1 = "\u0030" #unicode for 0  (0030 is in base 16)
s2 = "\u0061" #unicode for a  (0061 is in base 16)
s3 = "10km2"
s4 = "-1"     # there is no unicode for -1
s5 = "1.5"    # there is no unicode for 1.5
s6 = "999"
print(s1.isnumeric())
print(s2.isnumeric())
print(s3.isnumeric())
print(s4.isnumeric()) #"-1" and "1.5" are NOT considered numeric values
```

```
print(s5.isnumeric())
print(s6.isnumeric())
```

```
True
False
False
False
False
True
```

In [24]:
```
#s = '²'
s = '\u00B2'
print(s.isnumeric())

# s = '½'
s = '\u00BD'
print(s.isnumeric())

s='python12'
print(s.isnumeric())
```

```
True
True
False
```

## Converting Between Strings

Methods in this group convert between a string and some composite data type by either pasting objects together to make a string, or by breaking a string up into pieces.

### join(iterable)

Concatenates strings from an iterable.

s1.join(iterable) returns the string that results from concatenating the objects in iterable separated by s1.

Note that .join() is invoked on 1s, the separator string. iterable must be a sequence of string objects as well.

In [55]:
```
# Below example join all items in a tuple into a string, using a specified character as
myTuple = ("Sri", "Krishna", "BhagavadGeeta")
x = "".join(myTuple)
print(x)
x = " ".join(myTuple)
print(x)
x = "*".join(myTuple)
print(x)
```

```
SriKrishnaBhagavadGeeta
Sri Krishna BhagavadGeeta
Sri*Krishna*BhagavadGeeta
```

In [119…
```
# Join all items in a dictionary into a string, using a the word "TEST" as separator:
myDict = {"name": "Rama's", "country": "Ayodhya"}
separator1 = " and "
separator2 = " birthplace is "
x = separator1.join(myDict) # Note: When using a dictionary as an iterable, the returned
print(x)
x = separator2.join(myDict.values())
print(x)
```

```
name and country
Rama's birthplace is Ayodhya
```

In [120…
```
list1 = ["hi", "hello", "how are you?"]
```

```
S1 ="__"
S2 = S1.join(list1)
print(S2)
```

```
hi__hello__how are you?
```

In [57]:
```
str1 = "hi hello how are you?"
S1 =""
S2 = S1.join(str1)
print(S2)
```

```
hi hello how are you?
```

In [11]:
```
str1 = "hi hello how are you?"
S1 ="_"
S2 = S1.join(str1)
print(S2)
```

```
h_i_ _h_e_l_l_o_ _h_o_w_ _a_r_e_ _y_o_u_?
```

**partition(sep) - divides a string based on a separator.**

s1.partition(sep) splits s1 at the first occurrence of string sep.

The return value is a three-part tuple consisting of:

1. The portion of s1 preceding sep
2. sep itself
3. The portion of s1 following sep

Here are a couple examples of .partition() in action:

In [122…
```
print('foo$$bar$$baz'.partition('$$'))
```

```
('foo', '$$', 'bar$$baz')
```

In [57]:
```
print('foo.bar'.partition('.'))
```

```
('foo', '.', 'bar')
```

## Formatted string literals (f-strings) in Python

PEP 498 introduced a new string formatting mechanism known as Literal String Interpolation or more commonly as F-strings (because of the leading f character preceding the string literal).

The idea behind f-strings is to make string interpolation simpler.

**To create an f-string, prefix the string with the letter " f ". The string itself can be formatted in much the same way that you would with str.format().**

F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.

In [10]:
```
# Python3 program introducing f-string
val = 'Geeks'
print(f"{val}for{val} is a portal for {val}.")
```

```
GeeksforGeeks is a portal for Geeks.
```

In [19]:
```
name = 'Pratham'
age = 12
print(f"Hello, My name is {name} and I'm {age} years old.")
print(f"Hello, My name is {name:10} and I'm {age:^10} years old.")
```

```
Hello, My name is Pratham and I'm 12 years old.
Hello, My name is Pratham    and I'm     12     years old.
```

In [23]:
```
# Prints today's date with the help of datetime library

import datetime

today = datetime.datetime.today()
print(f"Today's date : {today}")
print(f"Today's date : {today:%B %d, %Y}")
```

```
Today's date : 2022-11-05 12:10:13.905275
Today's date : November 05, 2022
```

Note : F-strings are faster than the two most commonly used string formatting mechanisms, which are % formatting and str.format().

## Exercise problems

Write a program to reverse a given string.

In [3]:
```
str1='python'
str2=str1[::-1]
print(str2)
```

```
nohtyp
```

What is the output?

In [1]:
```
s='       Hello           '
print(s.strip())
s='*********Hello*********'
print(s.strip("*"))
```

```
Hello
Hello
```

In [8]:
```
S1='PES University'
S3=S2=S1   # One object three references
print(S1,S2,S3)
```

```
PES University PES University PES University
```

In [25]:
```
s = 'How ' "are " '''you?'''
print(s)
#we can juxtapose string literals, with or without whitespace (True/False)
```

```
How are you?
```

In [2]:
```
print('good ' * 3 + "day", (5+6)*2, (5+6,)*2)
```

```
good good good day 22 (11, 11)
```

In [2]:
```
s=set("abcc".replace('b','z').upper())
print(s)
```

```
{'Z', 'C', 'A'}
```

In [ ]:
```
s="PES UNIVERSITY ROCKS"
s.lower() # returns new string in lowercase, but not modifies the existing string
print(s)
#print(s.lower())
```

References:

1. str_tuple.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python
3. https://docs.python.org/