2 PYTHON STRINGS, LIST AND TUPLE

Table of Content

UNIT OBJECTIVE

After going through this unit, you will be able to:

- Handle strings with built in methods;
- Understand Python special data types like List and Tuple;
- Distinguish List and Tuple;
- Learn List and Tuple Operations;
- Use List and Tuple built in functions;
- Classify Mutable Vs Immutable Data Type;
- Write simple python Programs on String, List and Tuple;

Written and prepared by:

Mr. Bhushan A. Nikam

Dr. D. Y. Patil ACS College, Pimpri-Pune-18

Mob. : 9890657299

01

INTRODUCTION

This unit Introduces built-ins data structures like strings, lists, tuples that mostly used in python. Lists, strings and tuples are ordered sequences of objects. Strings that contain only characters, whereas list and tuples can contain any type of objects. Lists and tuples are like arrays. Lists are mutables so they can be extended, reduced or altered at drive, but Tuples and strings are immutables .

LEARNING OUTCOME

The content and assessments of this unit has been developed to achieve the following learning outcomes:

- Use string special and formatting operations in your python program
- Understand python built in string functions and other string features
- Write simple python programs to perform various operations on List and Tuple elements
- Run programs using built in List, tuple and string functions

2.1 String Special Operations

1. Concatenation operator '+'

Concatenation means combining two string. when + operator is used with string, the string on right side of the operator is concatenated to the string on the left side of the operator.

For example:

```
var1 = 'Welcome '
var2 = 'Learners'
var3 = var1 + var2


print ("'welcome ' + 'Learners' = " + var3)
```

Output: 'welcome '+'John' = Welcome Learners

2. Repetition of string using * operator

To duplicte multiple copies of same string '*' is used. If you see at the code in cost.py given below, var1 has the value = 'Welcome' and var2 = var1*6 = var1 + var1 + var1 + var1 + var1 + var1.

```
var1 = 'Welcome '
var2 = var1*6
print ("'welcome ' * 6 = " + var2)
```

Output:

'welcome ' * 6 = Welcome Welcome Welcome Welcome Welcome Welcome

3) Retrieving character from a given index of string

If you want to know which is the character at a particular location in string then that can be done easily. The first character in the string is at position 0, the second character is at position 1, the third character is at index 2 and so on.

In the following piece of code, var1 has the value of 'Welcome'. Here we try to find out which character is present at second position which means index 1.

```
var1 = 'Welcome '
print ("The second character in string 'welcome ' is " + var1[1])
```

Output: The second character in string 'welcome ' is e

3. Range of slice
Range of slice is used when you want to retrieve a certain section of the string. In the code shown below we try to retrieve four characters from the string var1 starting from second position i.e. index 1.

```
var1 = 'Welcome '
print ("The second to fourth characters in string 'welcome ' are " + var1[1:5])
```

Output: The second to fourth characters in string 'welcome ' are elco

4. String operator 'in'
String operator 'in' returns a Boolean value of either true or false. If the pattern of characters that you are searching for exists in the string then it will return the value of 'true' or else it will return 'false'

```
var1 = 'Welcome'
if 'e' in var1: print ("'e' exists in the word 'welcome'")
else: print ("'e' does not exist in the word 'welcome'")
```
Output: 'e' exists in the word 'welcome'

String operator 'not in'
String operator 'not in' returns a Boolean value of either true or false. If the pattern of characters that you are searching for does not exist in the string then it will return the value of 'true' or else it will return 'false'.

```
var1 = 'Welcome'
if 'e' not in var1: print ("'e' does not exist in the word 'welcome'")
else: print ("'e' exists in the word 'welcome'")
```
Output: 'e' exists in the word 'welcome'

2.2 String formatting operator

String Formatting
Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

Let's say you have a variable called "name" with your user name in it, and you would then like to print(out a greeting to that user.)

```
# This prints out "Hello, DPU Learning!"
name = "DPU Learning"
print("Hello, %s!" % name)
```
Output: Hello, DPU Learning!

To use two or more argument specifiers, use a tuple (parentheses):

```
# This prints out "John is 15 years old."
name = "Rahul"
age = 15
print("%s is %d years old." % (name, age))
```
Output: Rahul is 15 years old.

Any object which is not a string can be formatted using the %s operator as well.

For example:
```
# This prints out: A list: [1, 2, 3]
mylist = [1,2,3]
print("A list: %s" % mylist)
```
Output: A list: [1, 2, 3]

Here are some basic argument specifiers you should know:
%s - String (or any object with a string representation, like numbers)
%d - Integers
%f - Floating point numbers
%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.
%x/%X - Integers in hex representation (lowercase/uppercase)

2.3 Raw String, Unicode strings

Python raw string is created by prefixing a string literal with ‘r’ or ‘R’. Python raw string treats backslash (\) as a literal character. This is useful when we want to have a string that contains backslash and don’t want it to be treated as an escape character.

Python Raw String
Let’s say we want to create a string Hi\nHello in python. If we try to assign it to a normal string, the \n will be treated as a new line.

```
s = 'Hi\nHello'
print(s)
```

Output:
Hi
Hello

Let’s see how raw string helps us in treating backslash as a normal character.

```
raw_s = r'Hi\nHello'
print(raw_s)
```
Output: Hi\nHello

Let's see another example where the character followed by backslash doesn't have any special meaning.

```
>>> s = 'Hi\xHello'
```
Output: File "<input>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \xXX escape

We got the error because python doesn't know how to decode '\x' as it doesn't have any special meaning. Let's see how we can create the same string using raw strings.

```
>>> s = r'Hi\xHello'
>>> print(s)
Hi\xHello
```
If you are on Python console and create a raw-string like below.

```
>>> r'Hi\xHello'
'Hi\\xHello'
```

Don't get confused with the output having two backslashes. It's just to show it as a normal python string where backslash is being escaped.

Python Raw String and Quotes

When a backslash is followed by a quote in a raw string, it's escaped. However, the backslash also remains in the result. Because of this feature, we can't create a raw string of single backslash. Also, a raw string can't have an odd number of backslashes at the end.

Some of the invalid raw strings are:

```
r'\'       # missing end quote because the end quote is being escaped
r'ab\\\'   # first two backslashes will escape each other, the third one will try to escape the
           end quote.
```
Let's look at some of the valid raw string examples with quotes.

```
>>>raw_s = r'\''
>>>print(raw_s)
```
Output: \'

```
>>>raw_s = r'ab\\'
>>>print(raw_s)
```
Output: ab\\

```
raw_s = R'\\\"'  # prefix can be 'R' or 'r'
print(raw_s)
```
Output: \\\"

Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all different possible characters. There are many different types

of character encodings floating around at present, but the ones we deal most frequently with are **ASCII** , 8-bit encodings, and **Unicode-based** encodings. The Unicode Standard provides a unique number for every character, no matter what platform, device, application or language.
In Python 3, all strings are sequences of **Unicode characters** . You have two options to create Unicode string in Python. Either use **decode()** , or create a new Unicode string with **UTF-8 encoding** by unicode().
Using string's encode() method, you can convert unicoded strings into any encodings supported by Python. By default, Python uses **utf-8** encoding.

The syntax of encode() method is:

string.encode(encoding='UTF-8',errors='strict')

String encode() Parameters
By default, encode() method doesn't require any parameters.

It returns utf-8 encoded version of the string. In case of failure, it raises a UnicodeDecodeError exception.
However, it takes two parameters:

- **encoding** - the encoding type a string has to be encoded to
- **errors** - response when encoding fails. There are six types of error response
    - strict - default response which raises a UnicodeDecodeError exception on failure
    - ignore - ignores the unencodable unicode from the result
    - replace - replaces the unencodable unicode to a question mark **?**
    - xmlcharrefreplace - inserts XML character reference instead of unencodable unicode
    - backslashreplace - inserts a \uNNNN espace sequence instead of unencodable unicode
    - namereplace - inserts a \N{...} escape sequence instead of unencodable unicode

Example 1: Encode to Default Utf-8 Encoding

```
# unicode string
string = 'pythön!'
# print string
print('The string is:', string)
# default encoding to utf-8
string_utf = string.encode()
# print result
print('The encoded version is:', string_utf)
```

When you run the above program code , the output will be:

```
The string is: pythön!
The encoded version is: b'pyth\xc3\xb6n!'
```

Example 2: Encoding with error parameter

```
# unicode string
string = 'pythön!'
```

```
# print string
print('The string is:', string)
# ignore error
print('The encoded version (with ignore) is:', string.encode("ascii", "ignore"))
# replace error
print('The encoded version (with replace) is:', string.encode("ascii", "replace"))
```

When you run the program, the output will be:

```
The string is: pythön!
The encoded version (with ignore) is: b'pythn!'
The encoded version (with replace) is: b'pyth?n!'
```

2.4 Built-in String methods with example code

Sr. No.	Method	Description	Examples
1	capitalize()	Returns a copy of the string with its first character capitalized and the rest lowercased. Use title() if you want the first character of all words capitalized (i.e. title case).	a = "bee sting" print(a.capitalize()) **Output:** Bee sting
2	casefold()	Returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.	a = "BEE" print(a.casefold()) **Output:** bee
3	center(width[,fillchar])	Returns the string centered in a string of length *width*. Padding can be done using the specified *fillchar* (the default padding uses an ASCII space). The original string is returned if *width* is less than or equal to len(s)	a = "bee" b = a.center(12, "-") print(b) **Output:** ----bee-----
4	count(sub[,start[,end]])	Returns the number of non-overlapping occurrences of substring (sub) in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Non-overlapping occurrences means that Python won't double up on characters that have already been counted. For example, using a substring of xxx against xxxx returns 1.	a = "Mushroooom soup" print(a.count("O")) print(a.count("o")) print(a.count("oo")) print(a.count("ooo")) print(a.count("Homer")) print(a.count("o", 4, 7)) print(a.count("o", 7)) **Output:** 0 5 2 1 0 2 3

5	encode(encoding="utf-8",errors="strict")	Returns an encoded version of the string as a bytes object. The default encoding is utf-8. errors may be given to set a different error handling scheme. The possible value for errors are: strict (encoding errors raise a UnicodeError) ignore replace xmlcharrefreplace backslashreplace any other name registered via codecs.register_error()	from base64 import b64encode a = "Banana" print(a) a = b64encode(a.encode()) print(a) **Output:** Banana b'QmFuYW5h'
6	endswith(suffix[, start[, end]])	Returns True if the string ends with the specified suffix, otherwise it returns False. suffix can also be a tuple of suffixes. When the (optional) start argument is provided, the test begins at that position. With optional end, the test stops comparing at that position.	a = "Banana" print(a.endswith("a")) print(a.endswith("nana")) print(a.endswith("z")) print(a.endswith("an",1,3)) **Output:** True True False True
7	expandtabs(tabsize=8)	Returns a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every tabsize characters (the default is 8, giving tab positions at columns 0, 8, 16 and so on).	a="12\t3" print(a.expandtabs()) 12 3
8	find(sub[, start[, end]])	Returns the lowest index in the string where substring sub is found within the slice s[start:end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 if sub is not found. The find() method should only be used if you need to know the position of the substring. If you don't need to know its position (i.e. you only need to know if the substring exists in the string), use the in operator.	a = "Fitness" print(a.find("F")) print(a.find("f")) print(a.find("n")) print(a.find("ness")) print(a.find("ess")) print(a.find("z")) print(a.find("Homer")) RESULT 0 -1 3 3 4 -1 -1
9	format(*args, **kwargs)	Performs a string formatting operation. The string on which this	# Example 1 print("{} and

		method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.	{}".format("Tea", "Coffee"))
			Output: Tea and Coffee
			
			# Example 2
			print("{1} and {0}".format("Tea", "Coffee"))
			Output:
			Coffee and Tea
			
			# Example 3
			print("{lunch} and {dinner}".format(lunch="Peas", dinner="Beans"))
			Output:
			Peas and Beans
			
			# Example 4
			print("{0},{1},{2}".format(*"123"))
			Output:
			1, 2, 3
			
			# Example 5
			lunch = {"food": "Pizza", "drink": "Wine"}
			print("Lunch: {food}, {drink}".format(**lunch))
			Output:
			Lunch: Pizza, Wine

0110	10	format_map(mapping)	Similar to format(**mapping), except that mapping is used directly and not copied to a dictionary. This is useful if for example mapping is a dict subclass	# Example 1
				lunch = {"Food": "Pizza", "Drink": "Wine"}
				print("Lunch: {Food}, {Drink}".format_map(lunch))
				Output:
				Lunch: Pizza, Wine
				
				# Example 2
				class Default(dict):
				def__missing__(self, key):
				return key
				lunch = {"Food": "Pizza"}
				print("Lunch: {Food},{Drink}".format

			_map(Default(lunch))) lunch = {"Drink": "Wine"} print("Lunch: {Food}, {Drink}".format_map(Default(lunch))) **Output:** Lunch: Pizza, Drink Lunch: Food, Wine
11	index(*sub*[, *start*[, *end*]])	Like find() (above), but raises a ValueError when the substring is not found (find() returns -1 when the substring isn't found).	a = "Fitness" print(a.index("F")) print(a.index("n")) print(a.index("ness")) print(a.index("ess")) print(a.index("z")) #Error **Output:** 0 3 3 4 ValueError: substring not found
12	isalnum()	Returns True if all characters in the string are alphanumeric and there is at least one character. Returns False otherwise. A character *c* is deemed to be alphanumeric if one of the following returns True: • c.isalpha() • c.isdecimal() • c.isdigit() • c.isnumeric()	c = "Fitness" print(c.isalnum()) **Output:**True c = "123" print(c.isalnum()) **Output:**True c = "1.23" print(c.isalnum()) **Output:**False c = "$*%!!!" print(c.isalnum()) **Output:**False c = "0.34j" print(c.isalnum()) **Output:**False
13	isalpha()	Returns True if all characters in the string are alphabetic and there is at least one character. Returns False otherwise. Note that "alphabetic" in this case are those characters defined in the Unicode character database as "Letter". These are the characters with the general category property	c = "Fitness" print(c.isalpha()) **Output:** True c = "123" print(c.isalpha()) **Output:** False c = "$*%!!!"

		being one of "Lm", "Lt", "Lu", "Ll", or "Lo". This is different from the "Alphabetic" property defined in the Unicode Standard.	print(c.isalpha()) **Output:** False
14	`isdecimal()`	Returns True if all characters in the string are decimal characters and there is at least one character. Returns False otherwise. Decimal characters are those that can be used to form numbers in base 10.	c = "123" print(c.isdecimal()) **Output:** True
15	`isdigit()`	Returns True if all characters in the string are digits and there is at least one character. Returns False otherwise. The isdigit() method is often used when working with various unicode characters, such as for superscripts (eg, ²). A digit is a character that has the property value Numeric_Type=Digit or Numeri c_Type=Decimal.	c = "123" print(c.isdigit()) **Output:** True c = u"\u00B2" print(c.isdigit()) **Output:** True
16	`isidentifier()`	Returns true if the string is a valid identifier according to the language definition, section Identifiers and keywords from the Python docs. Use keyword.iskeyword() to test for reserved identifiers such as def, for, and class.	a = "123" print(a.isidentifier()) **Output:**False a = "_user_123" print(a.isidentifier()) **Output:**False

17	islower()	Returns True if all cased characters in the string are lowercase and there is at least one cased character. Returns False otherwise	a = "homer"
			print(a.islower())
			Output: True
			
			a = "HOMER"
			a = a.casefold() #Force lowercase
			print(a.islower())
			Output: True

18	isnumeric()	Returns True if all characters in the string are numeric characters, and there is at least one character. Returns False otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property. Numeric characters are those with the property value. Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric	c = "123" print(c.isnumeric()) **Output:** True c = u"\u00B2" print(c.isnumeric()) **Output**:True c = "1.23" print(c.isnumeric()) **Output**:False
19	isprintable()	Returns True if all characters in the	a = ""

		string are printable or the string is empty. Returns False otherwise. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", except for the ASCII space (0x20) which is considered printable. Printable characters in this context are those which should not be escaped when repr() is invoked on a string. It has no bearing on the handling of strings written to sys.stdout or sys.stderr.	print(a.isprintable()) **Output:**True a = " " print(a.isprintable()) **Output:**True a = u"\u00B2" print(a.isprintable()) **Output:**True a = "Bart" print(a.isprintable()) **Output:**True **Output:**True a = "\t" print(a.isprintable()) **Output:**False a = "\r\n" print(a.isprintable()) **Output:**False a = "Bart \r" print(a.isprintable()) **Output:**False
20	isspace()	Returns True if there are only whitespace characters in the string and there is at least one character. Returns False otherwise	a = " " print(a.isspace()) **Output**: True a = "\t" print(a.isspace()) **Output**: True
21	istitle()	Returns True if the string is a titlecased string and there is at least one character (for example uppercase characters may only follow uncased characters and lowercase characters only cased ones). Returns False otherwise.	a = " t" print(a.istitle()) **Output:** False a = " T" print(a.istitle()) **Output:** True a = "Tea" print(a.istitle()) **Output:** True a = "Tea and Coffee" print(a.istitle()) **Output:** False a = "Tea And Coffee" print(a.istitle()) **Output:** True

			a = "1. Tea & Coffee \r" print(a.istitle()) **Output:** True
22	isupper()	Returns True if all cased characters in the string are uppercase and there is at least one cased character. Returns False otherwise.	a = "_USER_123" print(a.isupper()) **Output:** True a = "Homer" print(a.isupper()) **Output:**False a = "HOMER" print(a.isupper()) **Output:** True
23	join(iterable)	Returns a string which is the concatenation of the strings in iterable. A TypeError will be raised if there are any non-string values in iterable, including bytes objects. The separator between elements is the string providing this method	a = "-" print(a.join("123")) **Output:1-2-3** a = "." print(a.join("USA")) **Output:U.S.A** a = ". " print(a.join(("Dr", "Who"))) **Output:**Dr.Who
24	isnumeric()	Returns the string left justified in a string of length *width*. Padding can be done using the specified *fillchar* (the default padding uses an ASCII space). The original string is returned if *width* is less than or equal to len(s)	a = "bee" b = a.ljust(12, "-") print(b) **Output:** bee---------
25	lower()	Returns a copy of the string with all the cased characters converted to lowercase.	a = "BEE" print(a.lower()) **Output: bee**
26	lstrip([*chars*])	Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or set to None, the chars argument defaults to removing whitespace.	a = " Bee " print(a.lstrip(), "!") **Output:** Bee ! a = "-----Bee-----" print(a.lstrip("-")) **Output:** Bee----
27	maketrans(*x*[, *y*[, *z*]])	This is a static method that returns a translation table usable for str.translate(). If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or set to None. Character keys will then be converted to ordinals.	frm = "SecrtCod" to = "12345678" trans_table = str.maketrans(frm, to) secret_code = "Secret Code".translate(trans_table) print(secret_code) **Output:** 123425 6782

		If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.	
28	partition(*sep*)	Splits the string at the first occurrence of *sep*, and returns a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, it returns a 3-tuple containing the string itself, followed by two empty strings	a = "Python-program" print(a.partition("-")) print(a.partition(".")) **Output:** ('Python', '-', 'program') print(a.partition(".")) a = "Python-program" print(a.partition("-")) **Output:** ('Python-program', '', '')
29	replace(*old*, *new*[, *count*])	Returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is provided, only the first count occurrences are replaced. For example, if count is 3, only the first 3 occurrences are replaced.	a = "Tea bag. Tea cup. Tea leaves." print(a.replace("Tea", "Coffee")) **Output:**Coffee bag. Coffee cup. Coffee leaves. print(a.replace("Tea", "Coffee", 2)) **Output:**Coffee bag. Coffee cup. Tea leaves.
30	rfind(*sub*[, *start*[, *end*]])	Returns 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. This method returns -1 on failure.	a = "Yes Fitness" print(a.rfind("Y")) **Output:** 0 print(a.rfind("e")) **Output:** 8 print(a.rfind("s")) **Output:** 10 print(a.rfind("ss")) **Output:** 9 print(a.rfind("y")) **Output:-1**
31	rindex(*sub*[, *start*[, *end]*)	Like rfind() but raises ValueError when the substring *sub* is not found.	print(a.rfind("y")) **Output:** ValueError: substring not found
32	rjust(*width*[, *fillchar*])	Returns the string right justified in a string of length *width*. Padding can be done using the specified *fillchar* (the default padding uses an ASCII space). The original string is returned if *width* is less than or equal to len(s)	a = "bee" b = a.rjust(12, "-") print(b) **Output**:---------bee
33	rpartition(*sep*)	Splits the string at the last	a = "Homer-Jay-

		occurrence of *sep*, and returns a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, it returns a 3-tuple containing the string itself, followed by two empty strings.	Simpson" print(a.rpartition("-")) **Output:**('Homer-Jay', '-', 'Simpson') print(a.rpartition(".")) **Output**:('', '', 'Homer-Jay-Simpson')
34	rsplit(sep=None, maxsplit=-1)	Returns 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 is set to None, any whitespace string is a separator. Except for splitting from the right, rsplit() behaves like split()	a = "Homer Jay Simpson" print(a.rsplit()) **Output:** ['Homer', 'Jay', 'Simpson'] a = "Homer-Jay-Simpson" print(a.rsplit(sep="-",maxsplit=1)) **Output**:['Homer-Jay', 'Simpson']
35	rstrip([chars])	Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or set to None, the chars argument defaults to removing whitespace.	a = " Bee " print(a.lstrip(), "!") **Output:** Bee ! a = "-----Bee-----" print(a.lstrip("-")) **Output:** Bee----
36	split(sep=None, maxsplit=-1)	Returns a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If maxsplit is not specified or -1, then there is no limit on the number of splits. If sep is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']). The sep argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2', '3']). Splitting an empty string with a specified separator returns [''] If sep is not specified or is set to None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting	a = "Homer Jay Simpson" print(a.split()) **Output:** ['Homer', 'Jay', 'Simpson'] a = "Homer-Jay-Simpson" print(a.split(sep="-",maxsplit=1)) **Output:** ['Homer', 'Jay-Simpson'] a = "Homer,,Bart," print(a.split(",")) **Output:** ['Homer', '', 'Bart', ''] a = "Homer,,Bart" print(a.split(",", maxsplit=1)) **Output:**['Homer', ',Bart'] a = "Homer<>Bart<>Marge" print(a.split("<>"))

		of just whitespace with a None separator returns [].	**Output:** ['Homer', 'Bart', 'Marge']
37	splitlines([keepends])	Returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and its value is True. This method splits on the following line boundaries. **Representation Description** \n Line Feed \r Carriage Return \n\r Carriage Return + Line Feed \v or \x0b Line Tabulation \f or \x0c Form feed \x1c File separator \x1d Group separator \x1e Record separator \x85 Next Line (C1 Control Code) \u2028 Line separator \u2029 Paragraph separator	a = "Tea\n\nand coffee\rcups\r\n" print(a.splitlines()) **Output:** ['Tea', '', 'and coffee', 'cups'] print(a.splitlines(keepends=True)) **Output:** ['Tea\n', '\n', 'and coffee\r', 'cups\r\n']
38	startswith(*prefix*[, *start*[, *end*]])	Returns True if the string starts with the specified prefix, otherwise it returns False. prefix can also be a tuple of prefixes. When the (optional) start argument is provided, the test begins at that position. With optional end, the test stops comparing at that position.	a = "Homer" print(a.startswith("H")) **Output:**True print(a.startswith("Homer")) **Output:** True print(a.startswith("om", 1, 3)) **Output:** True
39	strip([*chars*])	Returns a copy of the string with leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or set to None, the chars argument defaults to removing whitespace.	a = " Bee " print(a.strip(), "!") **Output:** Bee ! a = "-----Bee-----" print(a.strip("-")) **Output:**Bee
40	swapcase()	Returns a copy of the string with uppercase characters converted to lowercase and vice versa.	a = "Homer Simpson" print(a.swapcase()) **Output:** hOMER sIMPSON

41	title()	Returns a title-cased version of the string. Title case is where	a = "tea and coffee" print(a.title())

		words start with an uppercase character and the remaining characters are lowercase.	**Output:** Tea And Coffee a = "TEA AND COFFEE" print(a.title()) **Output:** Tea And Coffee
42	translate(*table*)	Returns a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via __getitem__(), typically a mapping or sequence. You can use maketrans() to create a translation map from character-to-character mappings in different formats.	frm = "SecrtCod" to = "12345678" trans_table = str.maketrans(frm, to) secret_code = "Secret Code".translate(trans_table) print(secret_code) **Output:** 123425 6782
43	upper()	Returns a copy of the string with all the cased characters converted to uppercase.	a = "bee" print(a.upper()) **Output:** BEE
44	zfill(width)	Returns a copy of the string left filled with ASCII 0 digits to make a string of length width. A leading sign prefix (+/-) is handled by inserting the padding after the sign character rather than before. The original string is returned if width is less than or equal to len(s).	a = "36" print(a.zfill(5)) **Output:** 00036 a = "-36" print(a.zfill(5)) **Output:** -0036

Activity 1:

1) What will be the output of the following Python code snippet?

print('The sum of {0:b} and {1:x} is {2:o}'.format(2, 10, 12))

a) The sum of 2 and 10 is 12 b) The sum of 10 and a is 14

c) The sum of 10 and a is c d) Error

2) What will be the output of the following Python code snippet?

print('for'.isidentifier())

a) True b) False c) None d) Error

3. What will be the output of the following Python code?

print('''
\tfoo'''.lstrip())

a) \tfoo b) foo c) foo d) none of the mentioned

4. What is “Hello”.replace(“l”, “e”)?

a) Heeeo b) Heelo c) Heleo d) None

5) What will be the output of the following Python code?

print("abc DEF".capitalize())

a) abc def b) ABC DEF c) Abc def d) Abc Def

2.5 Python Lists

Concept, creating and accessing elements

List is compound, very versatile and most frequently used datatype in Python, often referred to as sequences. A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

For Example:

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

Trying to access an element other that this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

Nested list are accessed using nested indexing.

```
my_list = ['p','r','o','b','e']
print(my_list[0])
```

Output: p

```
# Nested List
n_list = ["Happy", [2,0,1,5]]

# Nested indexing
print(n_list[0][1])
```

Output: a

```
print(n_list[1][3])
```

Output: 5

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_list = ['p','r','o','b','e']
print(my_list[-1])
```

Output: e

```
print(my_list[-5])
```

Output: p

List Slicing:

We can access a range of items in a list by using the slicing operator (colon). The syntax for this construction is list[x:y:z], with z referring to stride which refers to how many items to move forward after the first item is retrieved from the list. So far, we have omitted the stride parameter, and Python defaults to the stride of 1, so that every item between two index numbers is retrieved.

Consider, for example:

```
my_list = ['l','e','a','r','n','i','n','g','s']
# elements 3rd to 5th
print(my_list[2:5])
```

Output: ['a', 'r', 'n']

```
print(my_list[:-5])
```

Output: ['l', 'e', 'a', 'r']

```
# elements 6th to end
print(my_list[5:])
```

Output: ['i', 'n', 'g', 's']

```
# elements beginning to end
print(my_list[:])
```

Output: ['l', 'e', 'a', 'r', 'n', 'i', 'n', 'g', 's']

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

L	E	A	R	N	I	N	G	S
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

How to change or add elements to a list?

List are mutable, meaning, their elements can be changed unlike string or tuple.

We can use assignment operator (=) to change an item or a range of items.

For Example:

```
# mistake values
odd = [2, 4, 6, 8]

# change the 1st item
odd[0] = 1
```

Output: [1, 4, 6, 8]

```
# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
print(odd)
```

Output: [1, 3, 5, 7]

We can add one item to a list using append() method or add several items using extend() method.

For Example:

```
odd = [1, 3, 5]

odd.append(7)

print(odd)
```

Output: [1, 3, 5, 7]

```
odd.extend([9, 11, 13])
print(odd)
```

Output: [1, 3, 5, 7, 9, 11, 13]

We can also use + operator to combine two lists. This is also called concatenation.
The * operator repeats a list for the given number of times.

For Example:

```
odd = [1, 3, 5]
print(odd + [9, 7, 5])
```

Output: [1, 3, 5, 9, 7, 5]

```
print(["re"] * 3)
```

Output: ["re", "re", "re"]

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

For Example:

```
odd = [1, 9]


odd.insert(1,3)
print(odd)
```

Output: [1, 3, 9]

```
odd[2:2] = [5, 7]
print(odd)
```

Output: [1, 3, 5, 7, 9]

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

For Example:

```
my_list = ['p','r','o','b','l','e','m']

# delete one item
del my_list[2]

print(my_list)
```

Output: ['p', 'r', 'b', 'l', 'e', 'm']

```
# delete multiple items
del my_list[1:5]

print(my_list)
```
Output: ['p', 'm']

```
# delete entire list
del my_list
print(my_list)
```
Output: Error: List not defined

We can use remove() method to remove the given item or pop() method to remove an item at the given index.
The pop() method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).
We can also use the clear() method to empty a list.

For Example:

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')



print(my_list)
```
Output: ['r', 'o', 'b', 'l', 'e', 'm']

```
print(my_list.pop(1))
```
Output: 'o'

```
print(my_list)
```
Output: ['r', 'b', 'l', 'e', 'm']

```
print(my_list.pop())
```
Output: 'm'

```
print(my_list)
```
Output: ['r', 'b', 'l', 'e']

```
my_list.clear()
print(my_list)
```
Output: []

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>>          DPU   my_list[2:5] = []
>>> my_list
```

['p', 'r', 'm']

Python Matrix

Python doesn't have a built-in type for matrices. However, we can treat list of a list as a matrix. For example:

A = [[1, 4, 5], [-5, 8, 9]]

We can treat this list of a list as a matrix having 2 rows and 3 columns.

Python List Methods

Methods that are available with list object in Python programming are tabulated below. They are accessed as list.method(). Some of the methods have already been used Some examples of Python list methods:

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list

copy() - Returns a shallow copy of the list

For Example:

```
my_list = [3, 8, 1, 6, 0, 8, 4]
print(my_list.index(8))
```

Output: 1

```
print(my_list.count(8))
```

Output: 2

```
my_list.sort()
print(my_list)
```

Output: [0, 1, 3, 4, 6, 8, 8]

```
my_list.reverse()
print(my_list)
```

Output: [8, 8, 6, 4, 3, 1, 0]

2.6 Using Lists as stacks and Queues, List comprehensions

Stacks and Queues using Lists

Python's built-in List data structure comes bundled with methods to simulate both stack and queue operations.

Consider following python program on stack

```
letters = []

# Let's push some letters into our list
letters.append('c')
letters.append('a')
letters.append('t')
letters.append('g')

# Now let's pop letters, we should get 'g'
last_item = letters.pop()
print(last_item)

# If we pop again we'll get 't'
last_item = letters.pop()
print(last_item)

# 'c' and 'a' remain
print(letters) # ['c', 'a']
```

We can use the same functions to implement a Queue. The pop function optionally takes the index of the item we want to retrieve as an argument.

So we can use pop with the first index of the list i.e. 0, to get queue-like behavior.

Consider a "queue" of fruits:

```
fruits = []

# Let's enqueue some fruits into our list
fruits.append('banana')
fruits.append('grapes')
fruits.append('mango')
fruits.append('orange')

# Now let's dequeue our fruits, we should get 'banana'
first_item = fruits.pop(0)
print(first_item)

# If we dequeue again we'll get 'grapes'
first_item = fruits.pop(0)
```

```
print(first_item)

# 'mango' and 'orange' remain
print(fruits) # ['mango', 'orange']
```

Again, here we use the append and pop operations of the list to simulate the core operations of a queue.

Stacks and Queues with the Deque Library

Python has a deque (pronounced 'deck') library that provides a sequence with efficient methods to work as a stack or a queue.

deque is short for *Double Ended Queue* - a generalized queue that can get the first or last element that's stored: Consider following python program with commented output:

```
from collections import deque

# you can initialize a deque with a list
numbers = deque()

# Use append like before to add elements
numbers.append(99)
numbers.append(15)


numbers.append(82)
numbers.append(50)
numbers.append(47)



# You can pop like a stack
last_item = numbers.pop()
print(last_item) # 47
print(numbers) # deque([99, 15, 82, 50])

# You can dequeue like a queue
first_item = numbers.popleft()
print(first_item) # 99
print(numbers) # deque([15, 82, 50])
```

If you'd like to learn more about the deque library and other types of collections, you can refer links provided in further readings section.

List Comprehensions

List comprehensions offer a concise way to create lists based on existing lists. When using list comprehensions, lists can be built by leveraging any iterable, including strings and tuples.

Syntactically, list comprehensions consist of an iterable containing an expression followed by a for clause. This can be followed by additional for or if clauses. Familiarity with for loops and conditional statements will help you understand list comprehensions better.

List comprehensions provide an alternative syntax to creating lists and other sequential data types. While other methods of iteration, such as for loops, can also be used to create lists, list

comprehensions may be preferred because they can limit the number of lines used in your program.

In Python, list comprehensions are constructed like so:

list_variable = [x for x in iterable]

A list, or other iterable, is assigned to a variable. Additional variables that stand for items within the iterable are constructed around a for clause. The **in** keyword is used as it is in for loops, to iterate over the iterable.

Let's look at an example that creates a list based on a string:

```
shark_letters = [letter for letter in 'shark']
print(shark_letters)
```
Output:
```
['s', 'h', 'a', 'r', 'k']
```

List comprehensions can be rewritten as for loops, though not every for loop is able to be rewritten as a list comprehension.

Using our list comprehension that created the shark_letters list above, let's rewrite it as a for loop. This may help us better understand how the list comprehension works.

```
shark_letters = []
for letter in 'shark':
   shark_letters.append(letter)
print(shark_letters)
```

When creating a list with a for loop, the variable assigned to the list needs to be initialized with an empty list, as it is in the first line of our code block. The for loop then iterates over the item, using the variable letter in the iterable string 'shark'. Within the for loop, each item within the string is added to the list with the list.append(x) method.

Rewriting the list comprehension as a for loop provides us with the same output:

Output:
```
['s', 'h', 'a', 'r', 'k']
```

Using Conditionals with List Comprehensions

List comprehensions can utilize conditional statements to modify existing lists or other sequential data types when creating new lists.

Let's look at an example of an if statement used in a list comprehension:

```
fish_tuple = ('blowfish', 'clownfish', 'catfish', 'octopus')
fish_list = [fish for fish in fish_tuple if fish != 'octopus']
print(fish_list)
```

The list comprehension uses the tuple fish_tuple as the basis for the new list called fish_list.

When we run this, we'll see that fish_list contains the same string items as fish_tuple except for the fact that the string 'octopus' has been omitted:

Output:
['blowfish', 'clownfish', 'catfish']

Our new list therefore has every item of the original tuple except for the string that is excluded by the conditional statement.

We'll create another example that uses mathematical operators, integers, and the range() sequence type.

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

The list that is being created, number_list, will be populated with the squared values of each item in the range from 0-9 if the item's value is divisible by 2. The output is as follows:

Output:
[0, 4, 16, 36, 64]

To break down what the list comprehension is doing a little more, let's think about what would be printed out if we were just calling x for x in range(10). Our small program and output would then look like this:

```
number_list = [x for x in range(10)]
print(number_list)
```

Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Now, let's add the conditional statement:

```
number_list = [x for x in range(10) if x % 2 == 0]
print(number_list)
```

Output:
[0, 2, 4, 6, 8]
The if statement has limited the items in the final list to only include those items that are divisible by 2, omitting all of the odd numbers.

Finally, we can add the operator to have each x squared:

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

So each of the numbers in the previous list of [0, 2, 4, 6, 8] are now squared:

Output:
[0, 4, 16, 36, 64]
You can also replicate nested if statements with a list comprehension:

```
number_list = [x for x in range(100) if x % 3 == 0 if x % 5 == 0]
print(number_list)
```

Here, the list comprehension will first check to see if the number x is divisible by 3, and then check to see if x is divisible by 5. If x satisfies both requirements it will print, and the output is:

Output:
[0, 15, 30, 45, 60, 75, 90]
Conditional if statements can be used to control which items from an existing sequence are included in the creation of a new list.

Nested Loops in a List Comprehension
Nested loops can be used to perform multiple iterations in our programs.

This time, we'll look at an existing nested for loop construction and work our way towards a list comprehension.

Our code will create a new list that iterates over 2 lists and performs mathematical operations based on them. Here is our nested for loop code block:

```
my_list = []

for x in [20, 40, 60]:
   for y in [2, 4, 6]:
      my_list.append(x * y)

print(my_list)
```

When we run this code, we receive the following output:

Output:
[40, 80, 120, 80, 160, 240, 120, 240, 360]

This code is multiplying the items in the first list by the items in the second list over each iteration.

To transform this into a list comprehension, we will summarize each of the lines of code into one line, beginning with the x * y operation. This will be followed by the outer for loop, then the inner for loop. We'll add a print() statement below our list comprehension to confirm that the new list matches the list we created with our nested for loop block above:

```
my_list = [x * y for x in [20, 40, 60] for y in [2, 4, 6]]
print(my_list)
```

Output:
[40, 80, 120, 80, 160, 240, 120, 240, 360]

2.7 Functional programming tools - filter(), map(), and reduce():

map, filter, and reduce
Python provides several functions which enable a **functional approach** to programming. These functions are all convenience features in that they can be written in Python fairly easily.

Functional programming is all about **expressions**. We may say that the Functional programming is an **expression-oriented programming**.
Expression oriented functions of Python provides are:

1. map(aFunction, aSequence)
2. filter(aFunction, aSequence)
3. reduce(aFunction, aSequence)
4. lambda
5. list comprehension

map

One of the common things we do with list and other sequences is applying an operation to each item and collect the result.

For example, updating all the items in a list can be done easily with a for loop:

```
>>> items = [1, 2, 3, 4, 5]
>>> squared = []
>>> for x in items:
        squared.append(x ** 2)
>>> squared
```

Output:[1, 4, 9, 16, 25]

Since this is such a common operation, actually, we have a built-in feature that does most of the work for us.

The **map(Function, Sequence)** function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

```
>>> items = [1, 2, 3, 4, 5]
```

```
>>> def sqr(x): return x ** 2
```

```
>>> print(list(map(sqr, items)))
```

Output: [1, 4, 9, 16, 25]

We passed in a user-defined function applied to each item in the list. **map** calls **sqr** on each list item and collects all the return values into a new list.

Because **map** expects a function to be passed in, it also happens to be one of the places where **lambda** routinely appears:

```
>>>print(list(map((lambda x: x **2), items)))
```

Output: [1, 4, 9, 16, 25]

In the short example above, the **lambda(**anonymous) function squares each item in the items list.

As shown earlier, map is defined like this:

map(Function, Sequence)

While we still use **lamda** as a **Function**, we can have a **list of functions** as **Sequence**:

For Example:

```
def square(x):
    return (x**2)
def cube(x):
```

```
    return (x**3)

funcs = [square, cube]
for r in range(5):
   value = map(lambda x: x(r), funcs)
   print(list(value))
```

Output:
[0, 0]
[1, 1]
[4, 8]
[9, 27]
[16, 64]

Because using **map** is equivalent to **for** loops, with an extra code we can always write a general mapping utility:

For example:

```
def sqr(x): return x ** 2
def mymap(aFunc, aSeq):
    result = []
    for x in aSeq:
      result.append(aFunc(x))
    return result
```

```
>>> print(list(mymap(sqr, [1, 2, 3])))
```
Output:[1, 4, 9]
```
>>> print(mymap(sqr, [1, 2, 3]))
```
Output:[1, 4, 9]

Since it's a built-in, **map** is always available and always works the same way.

ForExample:

```
>>> pow(3,5)
```
Output:243

```
>>> pow(2,10)
```
Output:1024

```
>>> pow(3,11)
```
Output:177147

```
>>> pow(4,12)
```
Output:16777216

```
>>> list(map(pow, [2, 3, 4], [10, 11, 12]))
```
Output:[1024, 177147, 16777216]

As in the example above, with multiple sequences, **map()** expects an N-argument function for N sequences. In the example, **pow** function takes two arguments on each call.
Here is DPU another example of map() doing element-wise addition with two lists:

```
x = [1,2,3]
y = [4,5,6]

from operator import add
print(list(map(add, x, y)))
```

output: [5, 7, 9]

The **map** call is similar to the **list comprehension expression**. But **map** applies a **function call** to each item instead of an **arbitrary expression**. Because of this limitation, it is somewhat less general tool. In some cases, however, **map** may be faster to run than a list comprehension such as when mapping a built-in function. And **map** requires less coding.

If **function** is **None**, the **identity** function is assumed; if there are multiple arguments, **map()** returns a list consisting of **tuples** containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list:

```
>>> m = [1,2,3]
>>> n = [1,4,9]
>>> new_tuple = map(None, m, n)
>>> print(list(new_tuple))
```

Output: TypeError: 'NoneType' object is not callable

For Python 3, we may want to use itertools.zip_longest instead:

```
m = [1,2,3]
n = [1,4,9]
from itertools import zip_longest
for i,j in zip_longest(m,n):
    print(i,j)
```

Output:

```
1 1
2 4
3 9
```

The **zip_longest()** makes an **iterator** that aggregates elements from the two **iterables (m & n)**.

We can do typecasting using map. In the following example, we construct 4x3 matrix from the user input:
For Example:

```
arr=[]
for _ in range(4):
   arr.append(list(map(int, input().rstrip().split())))
print(arr)
```

With an input from the user:

1 2 3
4 5 6
7 8 9
10 11 12

We get a 4x3 integer array as **Output**:
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

filter and reduce
As the name suggests **filter** extracts each element in the sequence for which the function returns **True**. The **reduce** function is a little less obvious in its intent. This function reduces a list to a single value by combining elements via a supplied function. The **map** function is the simplest one among Python built-ins used for **functional programming**.
These tools apply functions to sequences and other iterables. The **filter** filters out items based on a test function which is a **filter** and apply functions to pairs of item and running result which is **reduce**.
Because they return iterables, **range** and **filter** both require **list** calls to display all their results in Python 3.0.
As an example, the following **filter** call picks out items in a sequence that are less than zero:
>>>list(range(-5,5))
Output: [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter((lambda x: x < 0), range(-5,5)))
Output:[-5, -4, -3, -2, -1]

Items in the sequence or iterable for which the function returns a true, the result are added to the result list. Like **map**, this function is roughly equivalent to a **for** loop, but it is built-in and fast:

```
result = []
for x in range(-5, 5):
    if x < 0:
        result.append(x)
```

>>>result
Output: [-5, -4, -3, -2, -1]

Here is another use case for **filter()**: finding intersection of two lists:
a = [1,2,3,5,7,9]
b = [2,3,5,6,7,8]
print (list(filter(lambda x: x in a, b)))
Output: [2, 3, 5, 7]

Note that we can do the same with list comprehension:

```
a = [1,2,3,5,7,9]
b = [2,3,5,6,7,8]
print ([x for x in a if x in b])
```
Output: [2, 3, 5, 7]

The **reduce** is in the **functools** in Python 3.0. It is more complex. It accepts an iterator to process, DPU but it's not an iterator itself. It returns a single result:

```
>>> from functools import reduce
>>> reduce( (lambda x, y: x * y), [1, 2, 3, 4])
```

Output:24

```
>>> reduce( (lambda x, y: x / y), [1, 2, 3, 4] )
```
Output:0.041666666666666664

At each step, **reduce** passes the current product or division, along with the next item from the list, to the passed-in **lambda** function. By default, the first item in the sequence initialized the starting value.

Here's the **for** loop version of the first of these calls, with the multiplication

hardcoded inside the loop:

```
>>> L = [1, 2, 3, 4]
>>> result = L[0]
>>> for x in L[1:]:
        result = result * x

>>> result
24
>>>
```

Let's make our own version of **reduce**.

```
>>> def myreduce(fnc, seq):
tally = seq[0]



for next in seq[1:]:
        tally = fnc(tally, next)
return tally

>>> myreduce( (lambda x, y: x * y), [1, 2, 3, 4])
24
>>> myreduce( (lambda x, y: x / y), [1, 2, 3, 4])
0.041666666666666664
```

We can concatenate a list of strings to make a sentence.

```
>>>import functools
>>> L = ['Testing ', 'shows ', 'the ', 'presence', ', ','not ', 'the ', 'absence ', 'of ', 'bugs']
>>> functools.reduce( (lambda x,y:x+y), L)
'Testing shows the presence, not the absence of bugs'
```

We can get the same result by using **join** :

```
>>> ''.join(L)
'Testing shows the presence, not the absence of bugs'
```

We can also use **operator** to produce the same result:

```
>>> import functools, operator
>>> functools.reduce(operator.add, L)
'Testing shows the presence, not the absence of bugs'
```

The built-in **reduce** also allows an optional third argument placed before the items in the sequence to serve as a default result when the sequence is empty.

Activity 2:

1) What Will Be The Output Of The Following Code Snippet?

```
a=[1,2,3,4,5]
print(a[3:0:-1])
```

a) Syntax error b) [4, 3, 2] c) [4, 3] d)[4, 3, 2, 1]

2) What Is The Correct Command To Shuffle The Following List?

```
fruit=['apple', 'banana', 'papaya', 'cherry']
```

a) fruit.shuffle() b) shuffle(fruit)
c) random.shuffle(fruit) d) random.shuffleList(fruit)

3) What Will Be The Output Of The Following Code Snippet?

```
arr = [[1, 2, 3, 4],
      [4, 5, 6, 7],
      [8, 9, 10, 11],
      [12, 13, 14, 15]]
for i in range(0, 4):
   print(arr[i].pop())
```

a) 1 2 3 4 b) 1 4 8 12 c) 12,13,14,15 d) 4 7 11 15

4) What Will Be The Output Of The Following Code Snippet?

```
arr = [1, 2, 3, 4, 5, 6]
for i in range(1, 6):
   arr[i - 1] = arr[i]
for i in range(0, 6):
   print(arr[i], end = " ")
```

a) 1 2 3 4 5 b) 2 3 4 5 6 1 c) 1 1 2 3 4 5 d) 2 3 4 5 6 6

5) What Will Be The Output Of The Following Code Snippet?

```
fruit_list1 = ['Apple', 'Berry', 'Cherry', 'Papaya']
fruit_list2 = fruit_list1
fruit_list3 = fruit_list1[:]

fruit_list2[0] = 'Guava'
fruit_list3[1] = 'Kiwi'

sum = 0
for ls in (fruit_list1, fruit_list2, fruit_list3):
  if ls[0] == 'Guava':
    sum += 1
  if ls[1] == 'Kiwi':
        sum += 20
```

print (sum)

a) 22 b) 21 c) 0 d) 43

2.8 Python Tuples:

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ()
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,)
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples
To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
```

When the above code is executed, it produces the following result −

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples
Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print (tup3);
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
To explicitly remove an entire tuple, just use the **del** statement. For example −

```
tup = ('physics', 'chemistry', 1997, 2000)
print (tup)
del tup
print ("After deleting tup : ")
print (tup)
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print (tup)
NameError: name 'tup' is not defined
```

Basic Tuples Operations
Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.
In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print (x)	1 2 3	Iteration

Indexing, Slicing, and Matrixes
Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings.
For Example:
Assuming following input −

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description

L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
print ('abc', -4.24e93, 18+6.6j, 'xyz')
x, y = 1, 2;
print ("Value of x , y : ", x,y)
```

When the above code is executed, it produces the following result −

```
abc -4.24e+93 (18+6.6j) xyz
```

```
Value of x , y : 1 2
```

2.9 Built-in Tuple Functions:

Python includes the following tuple functions –

Sr.No.	Function	Description
1	len(tuple)	Gives the total length of the tuple.
2	max(tuple)	Returns item from the tuple with max value.
3	min(tuple)	Returns item from the tuple with min value.
4	tuple(seq)	Converts a list into tuple.

Activity 3)

1) What Will Be The Output Of The Following Code Snippet?

```
a = 1, 2
b = (3, 4)
[print(sum(x)) for x in [a + b]]
```

2) Which Of The Following Statements Given Below Is/Are True?
   a) Tuples have structure, lists have an order.
   b) Tuples are homogeneous, lists are heterogeneous.

c) Tuples are immutable, lists are mutable.
d) Both a and c.

3) What Will Be The Output Of The Following Code Snippet?

```
init_tuple = ('Python') * 3
print(type(init_tuple))
```

a) <class 'tuple'> b) <class 'str'> c) <class 'list'> d) <class 'function'>

4) What Will Be The Output Of The Following Code Snippet?

```
init_tuple = (1,) * 3
init_tuple[0] = 2
print(init_tuple)
```

a) (1, 1, 1) b) (2, 2, 2)
b) c) (2, 1, 1) d) TypeError: 'tuple' object does not support item assignment

5) What will be the output of the following Python code?

```
>>>t=(1,2,4,3)
>>>t[1:3]
```

a) (1, 2) b) (1, 2, 4) c) (2, 4) d) (2, 4, 3)

2.10 SUMMARY

In this unit, we studied frequently used built in data types including, string, its formatting, lists and tuples along with associated inbuilt functions and features. All this information plays very important role to perform complex task in less efforts that we will study in next units.

2.11 CASE STUDY

1) Write python simple programs that includes varieties of string, list and tuple functions specified in this unit and try to understand the purpose of each function.

2) Write Python programs for addition and multiplication of two matrices.

3) Practice more programs on formatting operators, list comprehension, map, filter and reduce functions.

2.12 FURTHER READINGS

1) https://stackabuse.com/introduction-to-pythons-collections-module/
2) http://www.openbookproject.net/books/bpp4awd/ch03.html
3) https://www.journaldev.com/22960/python-map-function
4) https://www.python-course.eu/python3_lambda.php

DPU

2.13 SELF-ASSESMENT QUESTIONS

1) Write a Python program to count the number of characters (character frequency) in a string.

2) Write a Python program to get a string made of the first 2 and the last 2 chars from a given a string. If the string length is less than 2, return instead of the empty string.

3) Write a Python script that takes input from the user and displays that input back in upper and lower cases.

4)Write a Python program to remove duplicates from a list.

5) Write a Python program to unzip a list of tuples into individual lists.

2.14 ANSWER KEYS

Activity 1

Questions No.	Answers
1	b **Explanation:** 2 is converted to binary, 10 to hexadecimal and 12 to octal.
2	a **Explanation:** Even keywords are considered as valid identifiers
3	B **Explanation**: All leading whitespace is removed.
4	a
5	c

Activity 2:

Questions No.	Answers
1	c
2	c
3	d
4	d
5	a

Activity 3:

Questions No.	Answers
1	c
2	d
3	b
4	d
5	C **Explanation:** Slicing in tuples takes

	place just as it does in strings.

SELF-ASSESMENT ANSWERS:

1)

```
def char_frequency(str1):
    dict = {}
    for n in str1:
        keys = dict.keys()
        if n in keys:
            dict[n] += 1
        else:
            dict[n] = 1
    return dict
print(char_frequency('dpu.learning'))
```

2)

```
def string_both_ends(str):
  if len(str) < 2:
    return ''
  return str[0:2] + str[-2:]
print(string_both_ends('dpulearning'))
print(string_both_ends('dp'))
print(string_both_ends('d'))
```

3)

```
user_input = input("What's your favourite language? ")
print("My favourite language is ", user_input.upper())
print("My favourite language is ", user_input.lower())
```

4)

```
a = [10,20,30,20,10,50,60,40,80,50,40]
uniq_items = []
for x in a:
    if x not in uniq_items:
      uniq_items.append(x)
print(uniq_items)
```

5)

```
l = [(1,2), (3,4), (8,9)]
print(list((zip(*l))))
```