**Module 1:**

**Title: Keywords and Identifiers, variables and Data Types**

**Keywords:**

* Keywords are the words whose meaning is already defined, we cannot change the meaning of the reserved words.
* You cannot use reserved words as constant or variable or any other identifier names.
* All the keywords are represented in lowercase only.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| and | def | for | is | raise |
| as | del | from | lambda | return |
| assert | elif | global | nonlocal | try |
| break | else | if | not | while |
| class | except | import | or | with |
| continue | finally | in | pass | yield |
| True | False | None | async | await |

**Identifiers:**

A python identifier is a name used to identify a variable, function, class, module or other object.

* An identifier name must start with a letter or the underscore character. class name start with an uppercase letter. All other identifiers starts with lowercase.
* An identifier name can’t start with a number.
* An identifier name can only contain alpha-numeric characters and underscores.
* Identifier names are case sensitive(age, Age, AGE are three different types)
* Single leading underscore🡪identifier is private
* Two leading underscores🡪strongly private identifier
* Ends with two trailing underscores🡪identifier is a language-defined special name

**Variables:**

* Unlike other programming languages, python has no command for declaring a variable.
* a variable is created the moment you first assign a value to it.

**Ex:** x=5

Y=”xyz”

Print(x)

Print(y)

* Variables do not need to be declared with any particular type and can even change type after they have been set.

Eg: x=4 # x is of type int

x=”abc” #x is now of type string

print(x)

**OUTPUT :**

the python print statement is often used to display something on the screen.

Ex: print(“hello”)

X=10

Print(x)

**INPUT:**

The python input function is often used to take or accept the input from the user. by default input function returns string type for other type of values we need to perform typecasting explicitly.

Ex: name=input(“enter name of the student”)

Roll no=int(input(“enter marks of a students”)

**Data Types:**

Data types are the some of the keywords of the programming language, which are used to specify what type of data has to be store into the variables.

Without defining the data types we can’t store the data into the variable. Because the memory is not allocated for the variable if the data type is not defined.

Programming languages supports two different varieties of data types they are:

1. Static data types
2. Dynamic data types

**STATIC DATA TYPES:**

In static data types supported languages programmer should define the data type to the variables explicitly at the time of writing the program.

In static data type supported programming languages one variable can store one variety of data only.

Ex: int I=10;

I=20;

I=12.2;

C,c++,java,.NET,…languages supports static data types.

**DYNAMIC DATA TYPES:**

In dynamic data types supported languages data types of the variables will be decided at the time of execution of the program based on the data which is assigned to the variable.

In dynamic data types supported languages one variable can store different varieties of data.

Ex: I=10 #int

I=12.24 #float

I=True #bool

I=”abc” #string

Python, java script,..supports dynamic data types.

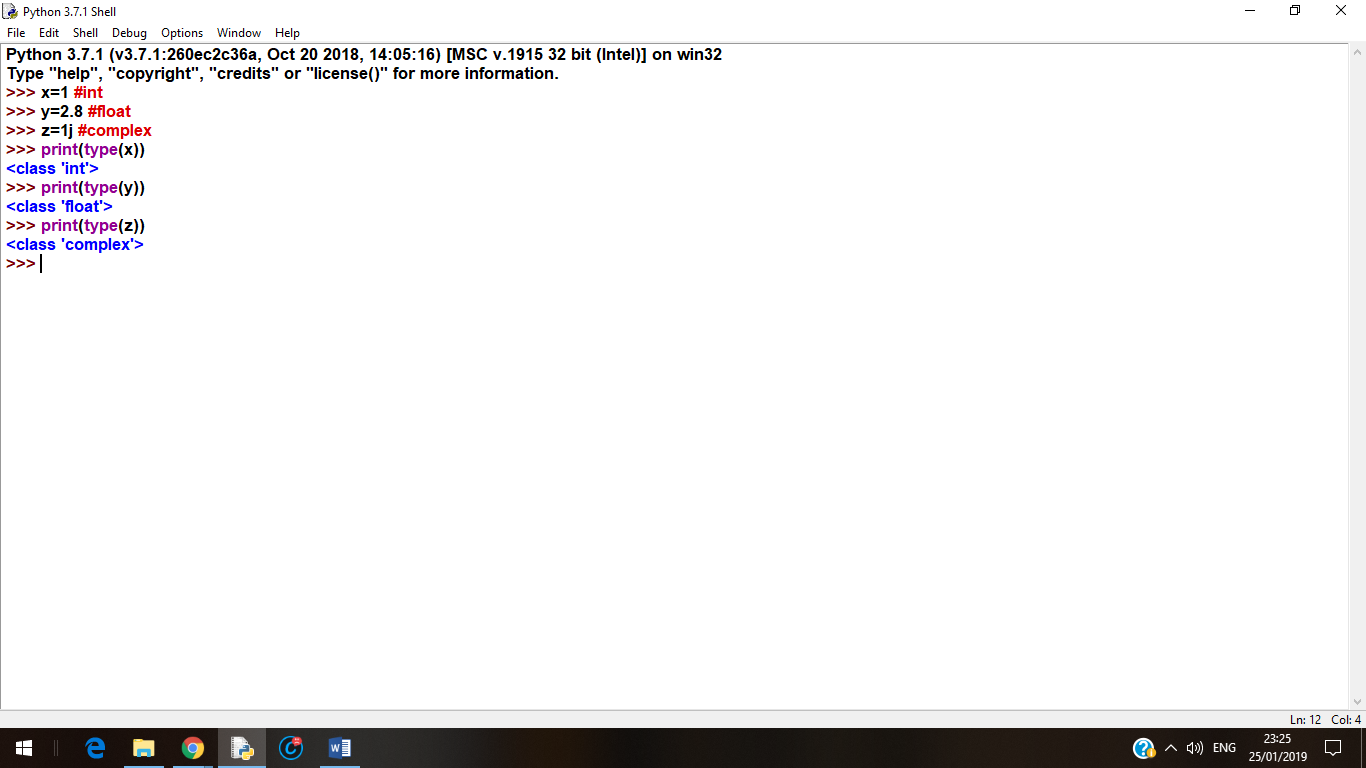
Every data type in python internally implemented as class.

Python data types are categorized into 2 categories they are:

1. Fundamental types:
2. Int
3. Float
4. Complex
5. Bool
6. Str
7. collection types:
8. list
9. tuple
10. set
11. dictionary

**FUNDAMENTAL TYPES:**

* it represents class objects of storing data.
* Variables of numeric types are created when you assign a value to them.
* to verify the type of any object in python, use the type() function.



1. **int**: int or integer is a whole number, positive or negative without decimals of unlimited length.

**Ex:** x=1

Y=1235455667788

Z=-1234567

1. **Float**: float or floating point numbers are positive or negative, containing one or more decimals.

Float can also be scientific numbers with an ‘e’ to indicate the power of 10.

**Ex:** x=1.2 a=35e4

Y=1.0 b=12E4

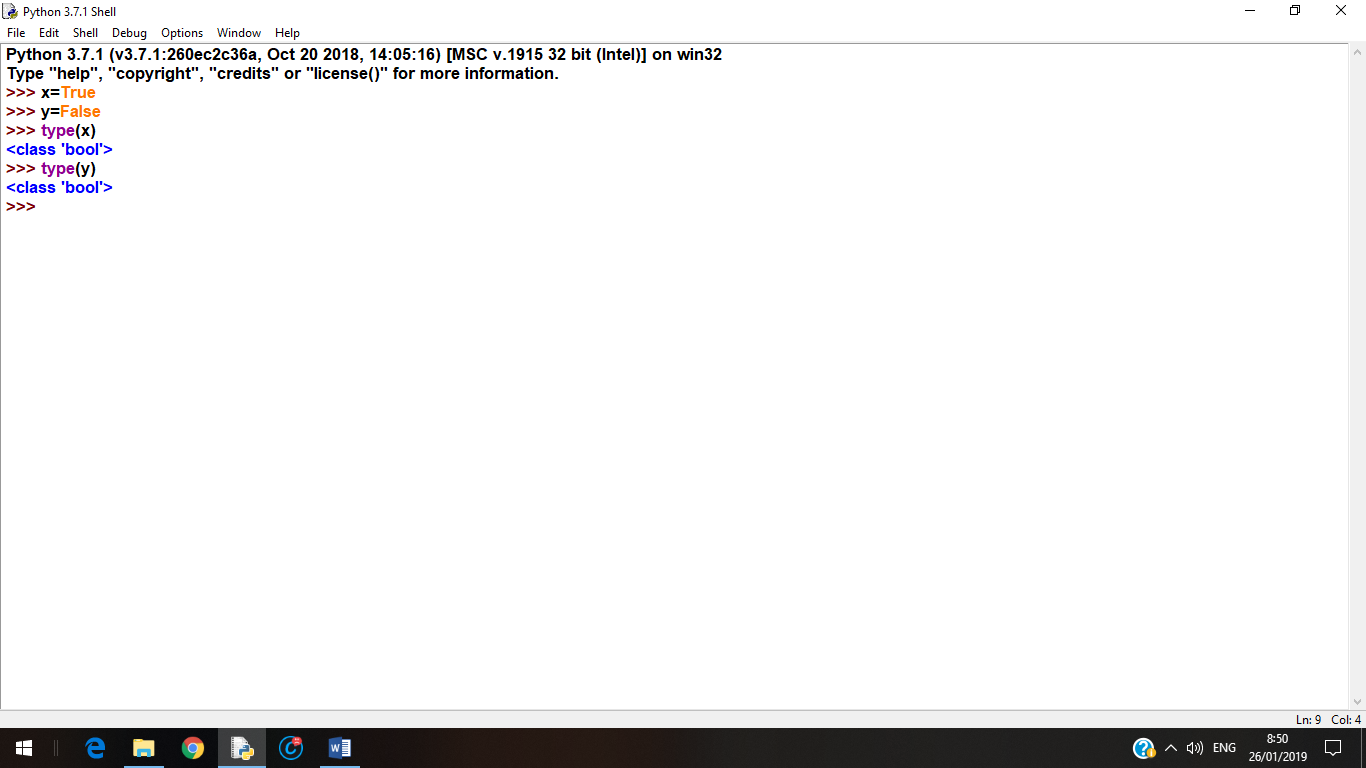
Z=-1.12 c=-87.7e100

1. **Complex**: complex numbers are written with a ‘j’ as the imaginary part.

**Ex:** x=3+5j

Y=5j

1. **Bool:** bool type contains two values either true or false



1. **Str**: we can create strings simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

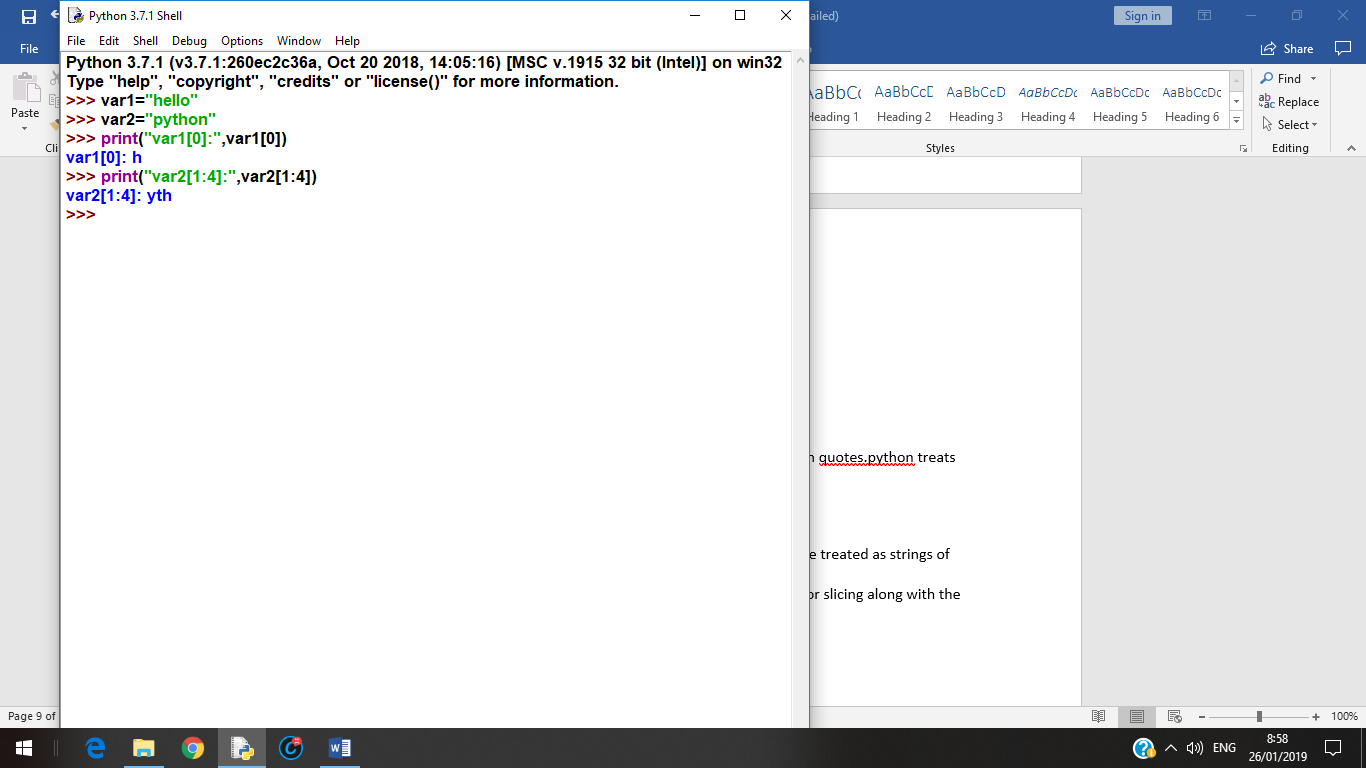
**Ex**: var1=’hello’

Var2=”python”

**ACCESSING VALUES IN STRINGS:**

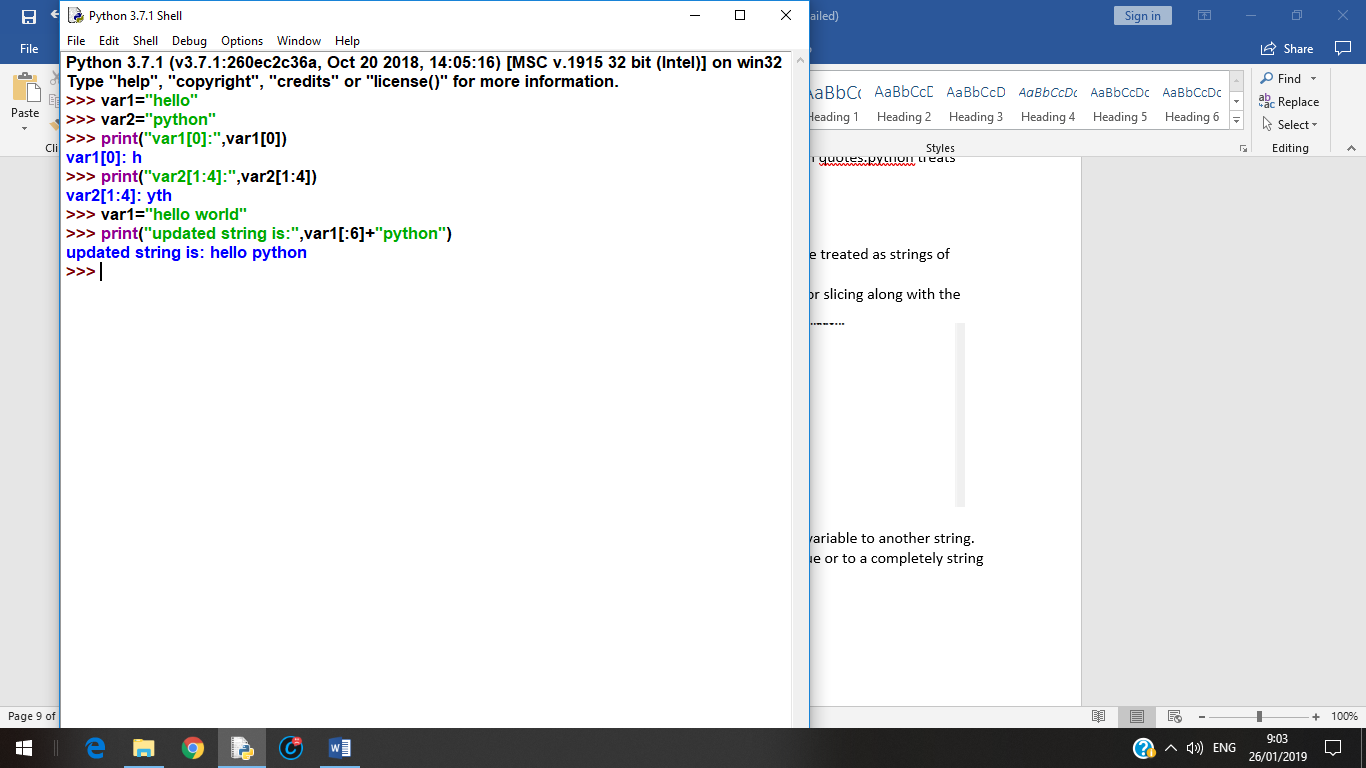
Python does not support a character type these are treated as strings of length one. thus also considered a substring.

To access the substrings, use the square brackets for slicing along with the index or indices to obtain your substring.



**UPDATING STRINGS:**

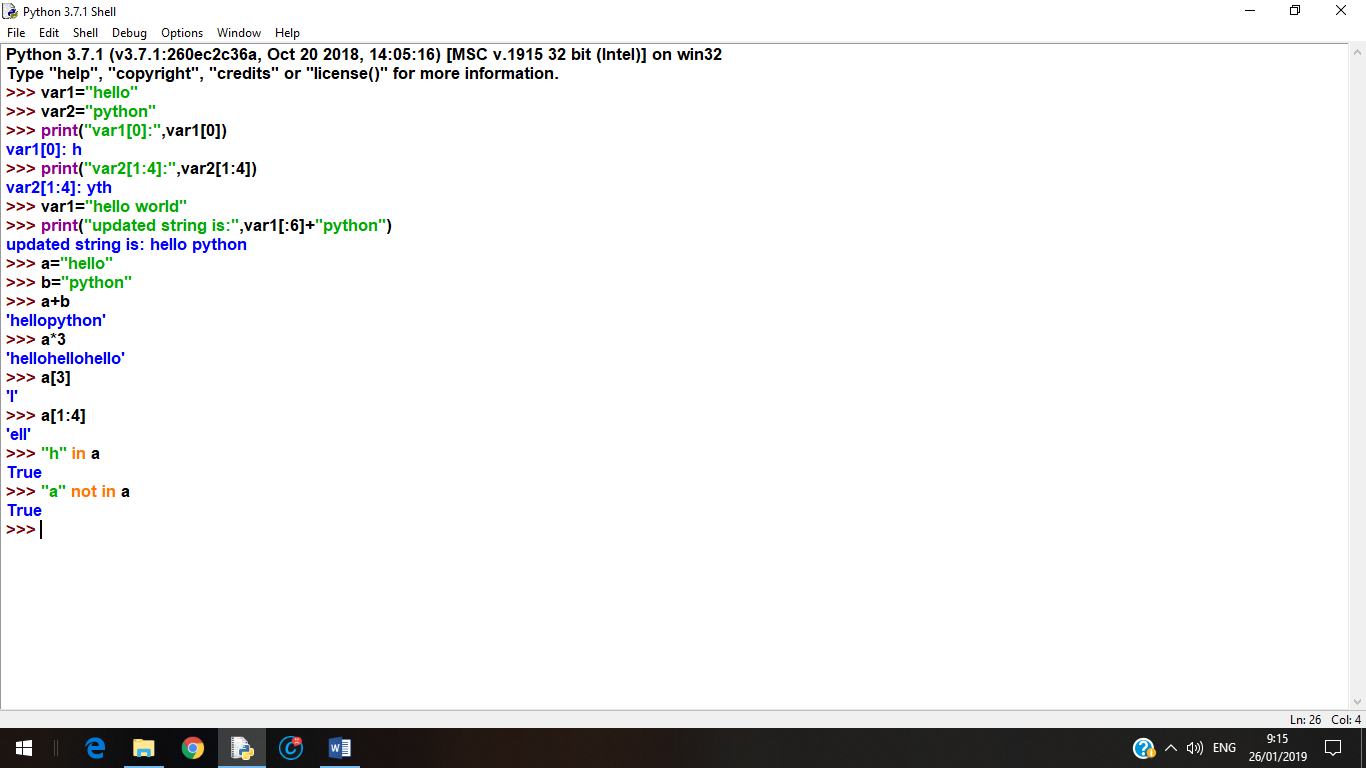
* You can update an existing string by assigning a variable to another string.
* The new value can be treated to its previous value or to a completely string altogether.



**STRING SPECIAL OPERATORS:**

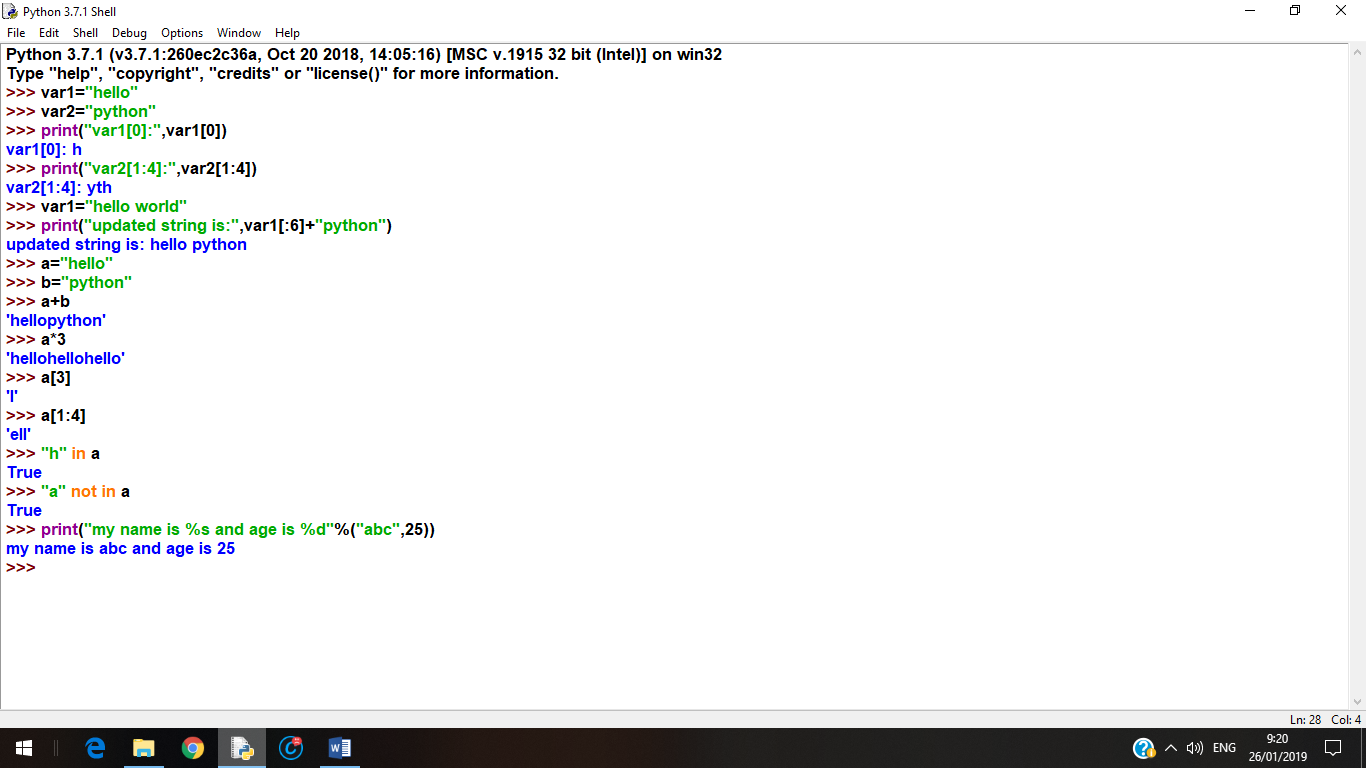
Assume two string variables a=”hello” and b=”python”

1. +🡪concatenation: adds values on either side of the operator.
2. \*🡪repetition: creates new strings ,concatenating multiple copies of the same string.
3. [ ]🡪slice: gives the character for the given index
4. [ : ]🡪range slice: gives the characters for the given range
5. In🡪membership operator: returns true if the character exists in the given string.
6. Not in 🡪 membership operator: returns true if the character does not exists in the given string.



**FORMAT OPERATOR:**

%-->format operator performs strings formatting



**I. STRING BUILT-IN METHODS:**

1. capitalize(self, /) : Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

2.Casefold(self, /) : Return a version of the string suitable for case less comparisons.

3. center(self, width, fill char=' ', /) : Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

4.count(...) : S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of substring

sub in string S[start: end]. Optional arguments start and end are interpreted as in slice notation.

5.encode(self, /, encoding='utf-8', errors='strict')

Encode the string using the codec registered for encoding.

6.endswith(...):S.endswith(suffix[, start[, end]]) -> boolReturn True if S ends with the specified suffix, False otherwise.With optional start, test S beginning at that position.With optional end, stop comparing S at that position.suffix can also be a tuple of strings to try.

7. expand tabs(self, /, tab size=8) : Return a copy where all tab characters are expanded using spaces.If tabsize is not given, a tab size of 8 characters is assumed.

8. find(...) :S.find(sub[, start[, end]]) -> int : Return the lowest index in S 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. -1 on failure.

9. format(...) :S.format(\*args, \*\*kwargs) -> str : Return a formatted version of S, using substitutions from args and kwargs.The substitutions are identified by braces ('{' and '}').

10. format\_map(...) :S.format\_map(mapping) -> str : Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

11. index(...) : S.index(sub[, start[, end]]) -> int : Return the lowest index in S 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.Raises ValueError when the substring is not found.

12. isalnum(self, /) : Return True if the string is an alpha-numeric string, False otherwise.A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

13. isalpha(self, /) : Return True if the string is an alphabetic string, False otherwise.A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

14. isascii(self, /) : Return True if all characters in the string are ASCII, False otherwise.ASCII characters have code points in the range U+0000-U+007F

Empty string is ASCII too.

15. isdecimal(self, /) : Return True if the string is a decimal string, False otherwise.A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

16. isdigit(self, /) : Return True if the string is a digit string, False otherwise. A string is a digit string if all characters in the string are digits and thereis at least one character in the string.

17. isidentifier(self, /) : Return True if the string is a valid Python identifier, False otherwise.Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

18. islower(self, /) : Return True if the string is a lowercase string, False otherwise.A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

19. isnumeric(self, /) : Return True if the string is a numeric string, False otherwise.A string is numeric if all characters in the string are numeric and there is atleast one character in the string.

20. isprintable(self, /) : Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

21. isspace(self, /) : Return True if the string is a whitespace string, False otherwise.A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

22. istitle(self, /) : Return True if the string is a title-cased string, False otherwise.In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

23. isupper(self, /) : Return True if the string is an uppercase string, False otherwise.A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

24. join(self, iterable, /) : Concatenate any number of strings.The string whose method is called is inserted in between each given string.The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

25. ljust(self, width, fillchar=' ', /) : Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

26. lower(self, /) : Return a copy of the string converted to lowercase.

27. lstrip(self, chars=None, /) : Return a copy of the string with leading whitespace removed.If chars is given and not None, remove characters in chars instead.

28. partition(self, sep, /) : Partition the string into three parts using the given separator.This will search for the separator in the string. If the separator is found,

returns a 3-tuple containing the part before the separator, the separator itself,and the part after it.If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

29. replace(self, old, new, count=-1, /) : Return a copy with all occurrences of substring old replaced by new. Count Maximum number of occurrences to replace.-1 (the default value) means replace all occurrences.If the optional argument count is given, only the first count occurrences are replaced.

30. rfind(...) :S.rfind(sub[, start[, end]]) -> int : Return the highest index in S where substring sub is found,such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.Return -1 on failure.

31. rindex(...) : S.rindex(sub[, start[, end]]) -> int : Return the highest index in S 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.Raises Value Error when the substring is not found.

32. rjust(self, width, fillchar=' ', /)🡪 Return a right-justified string of length width.Padding is done using the specified fill character (default is a space).

33. rpartition(self, sep, /)🡪 Partition the string into three parts using the given separator. This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

34 .rsplit(self, /, sep=None, maxsplit=-1) : Return a list of the words in the string, using sep as the delimiter string.

sep : The delimiter according which to split the string.None (the default value) means split according to any whitespace,and discard empty strings from the result.

Maxsplit : Maximum number of splits to do.-1 (the default value) means no limit.

splits are done starting at the end of the string and working to the front.

35. rstrip(self, chars=None, /) : Return a copy of the string with trailing whitespace removed.If chars is given and not None, remove characters in chars instead.

36. split(self, /, sep=None, maxsplit=-1) : Return a list of the words in the string, using sep as the delimiter string.

Sep : The delimiter according which to split the string.None (the default value) means split according to any whitespace,and discard empty strings from the result.

Maxsplit :Maximum number of splits to do. -1 (the default value) means no limit.

37 splitlines(self, /, keepends=False) : Return a list of the lines in the string, breaking at line boundaries.Line breaks are not included in the resulting list unless keep ends is given and true.

38 .startswith(...) : S.startswith(prefix[, start[, end]]) -> bool : Return True if S starts with the specified prefix, False otherwise.With optional start, test S beginning at that position.With optional end, stop comparing S at that position.

prefix can also be a tuple of strings to try.

39 .strip(self, chars=None, /) : Return a copy of the string with leading and trailing whitespace remove.If chars is given and not None, remove characters in chars instead.

40. swapcase(self, /) : Convert uppercase characters to lowercase and lowercase characters to uppercase.

41. title(self, /) : Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining

cased characters have lower case.

42. translate(self, table, /) : Replace each character in the string using the given translation table table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via \_\_getitem\_\_, for instance a

dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

43. upper(self, /) : Return a copy of the string converted to uppercase.

44. zfill(self, width, /) : Pad a numeric string with zeros on the left, to fill a field of the given width.The string is never truncated.

**7.LOOPING STATEMENTS**

Looping statements are used to execute the set of statements repeatedly. Python supports 2 types of looping statements they are

* 1. While
  2. For

While loop: while loop executes set of statements repeatedly until condition becomes false.

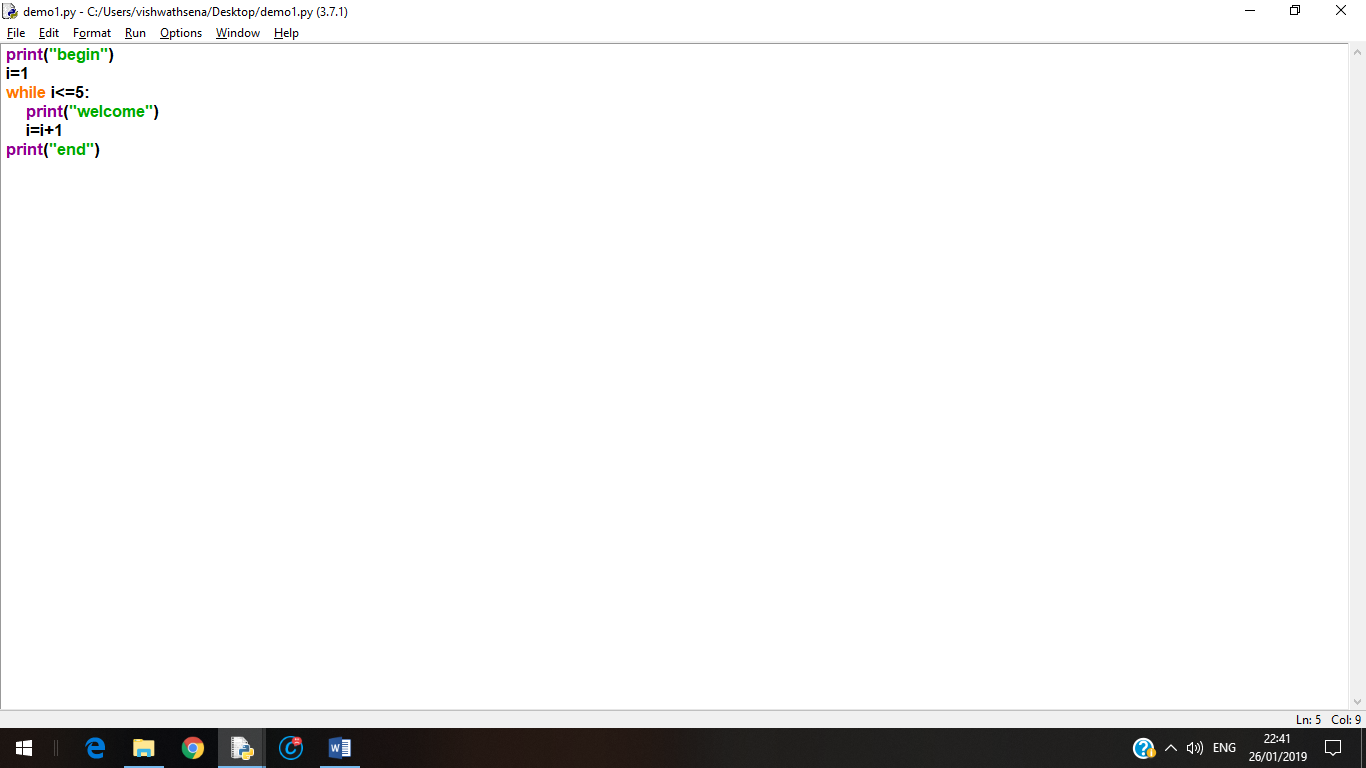
**Syntax**: while condition:

Stmt1

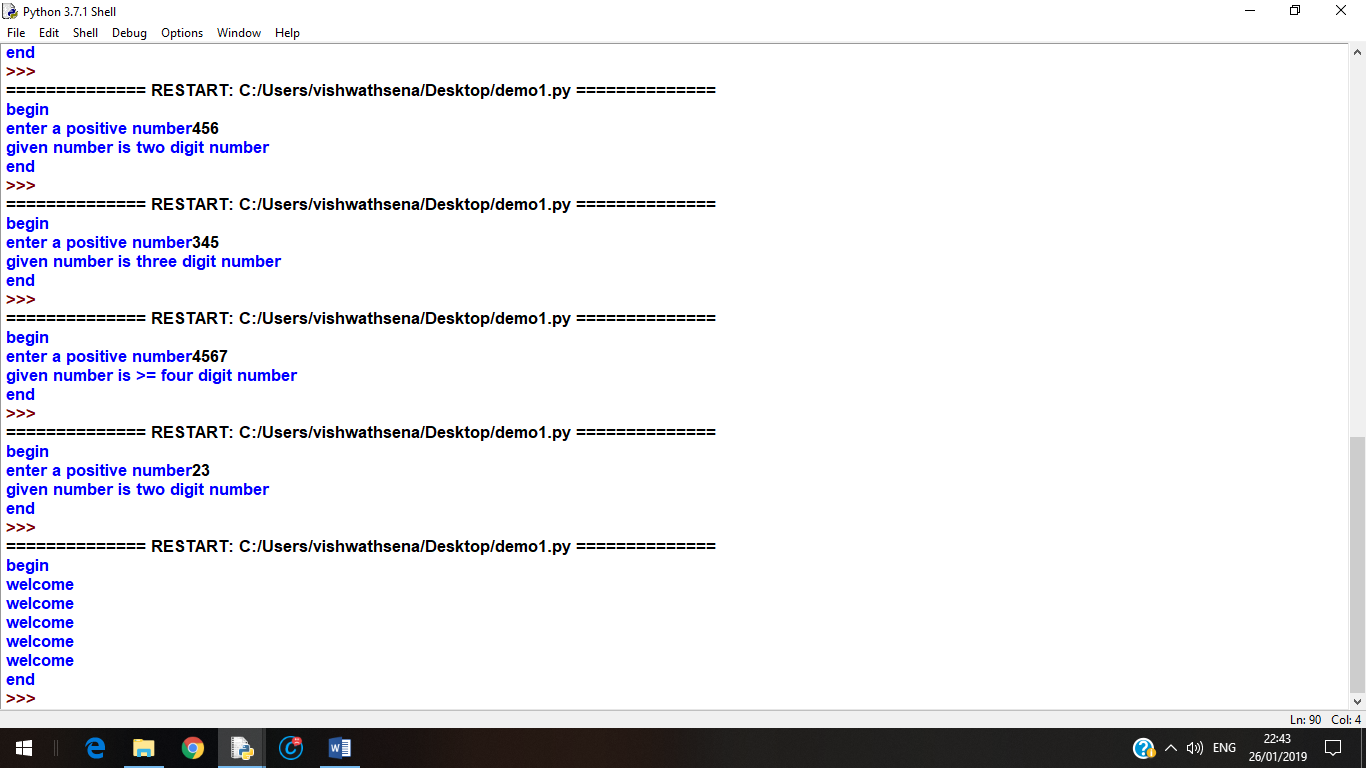
Stmt2

………..

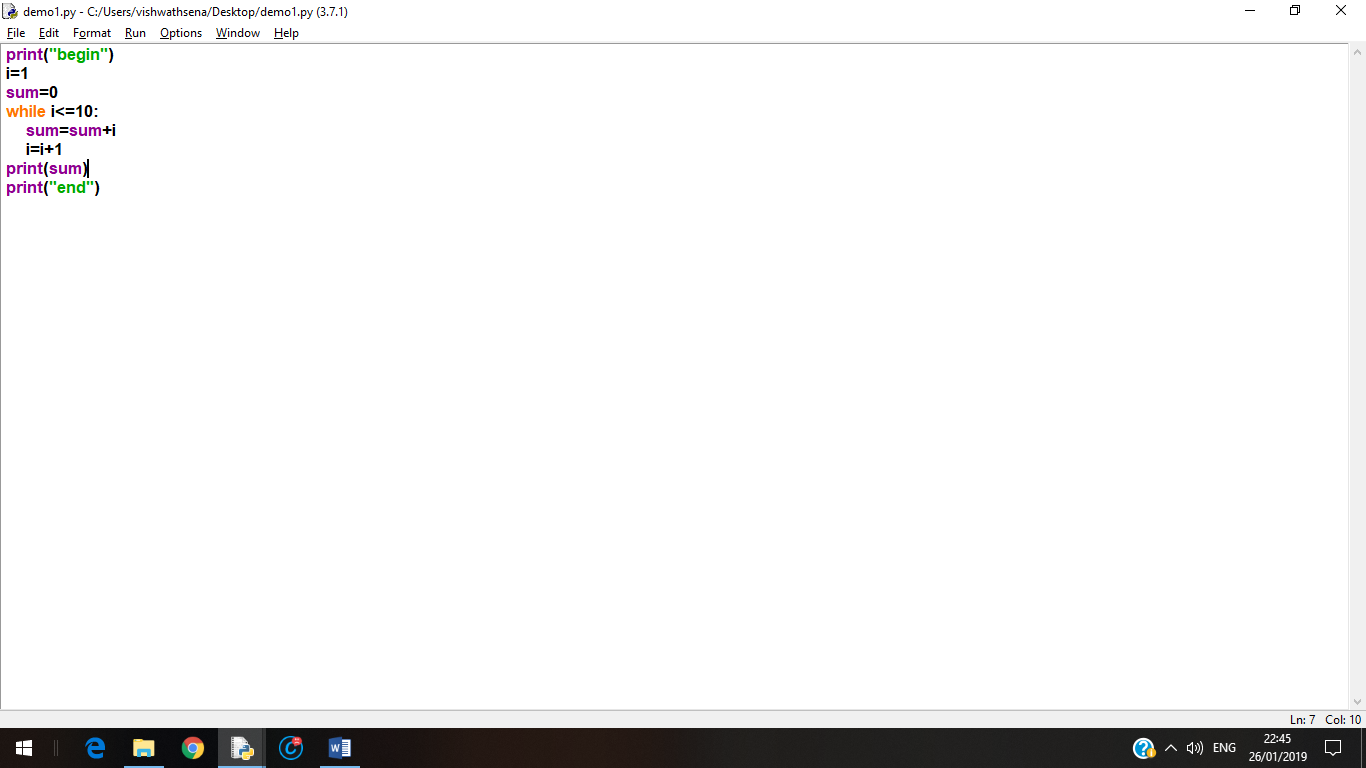
**Example 1**:



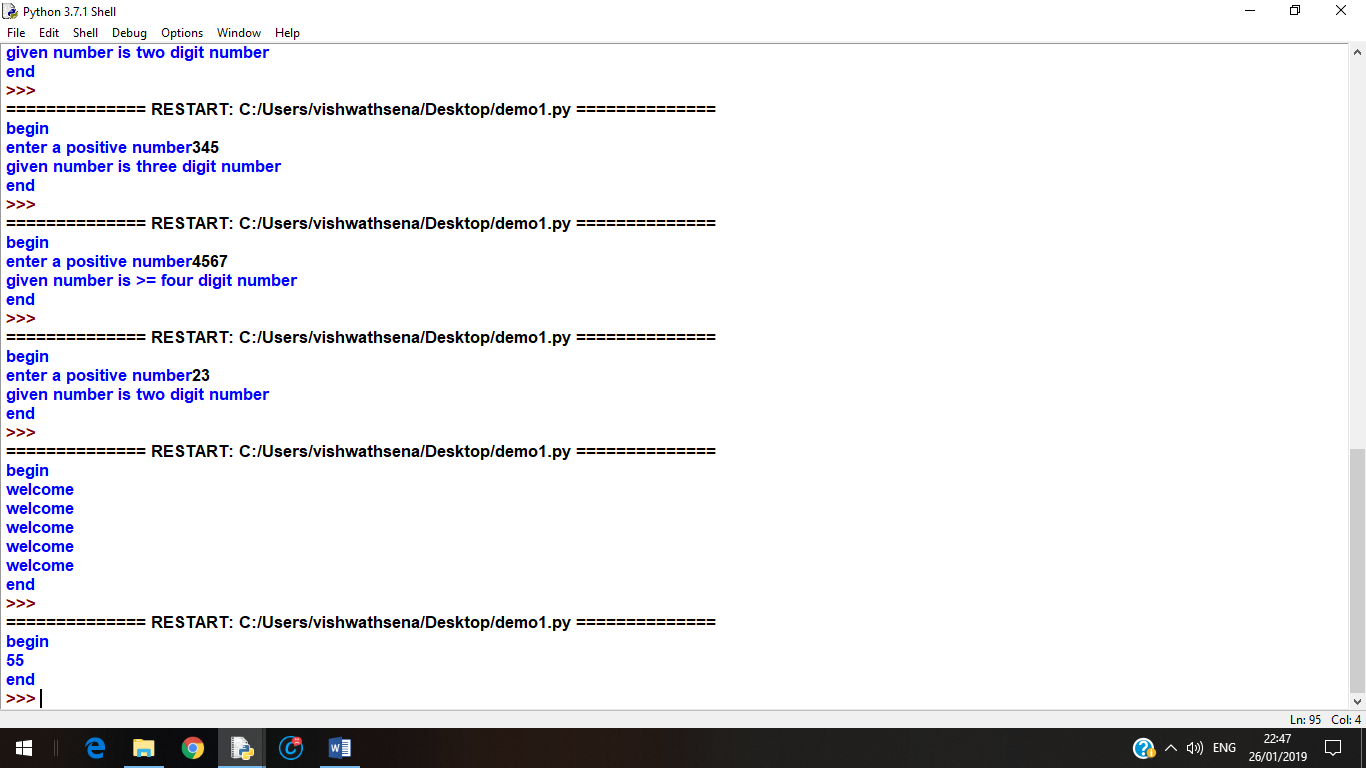
**output**:



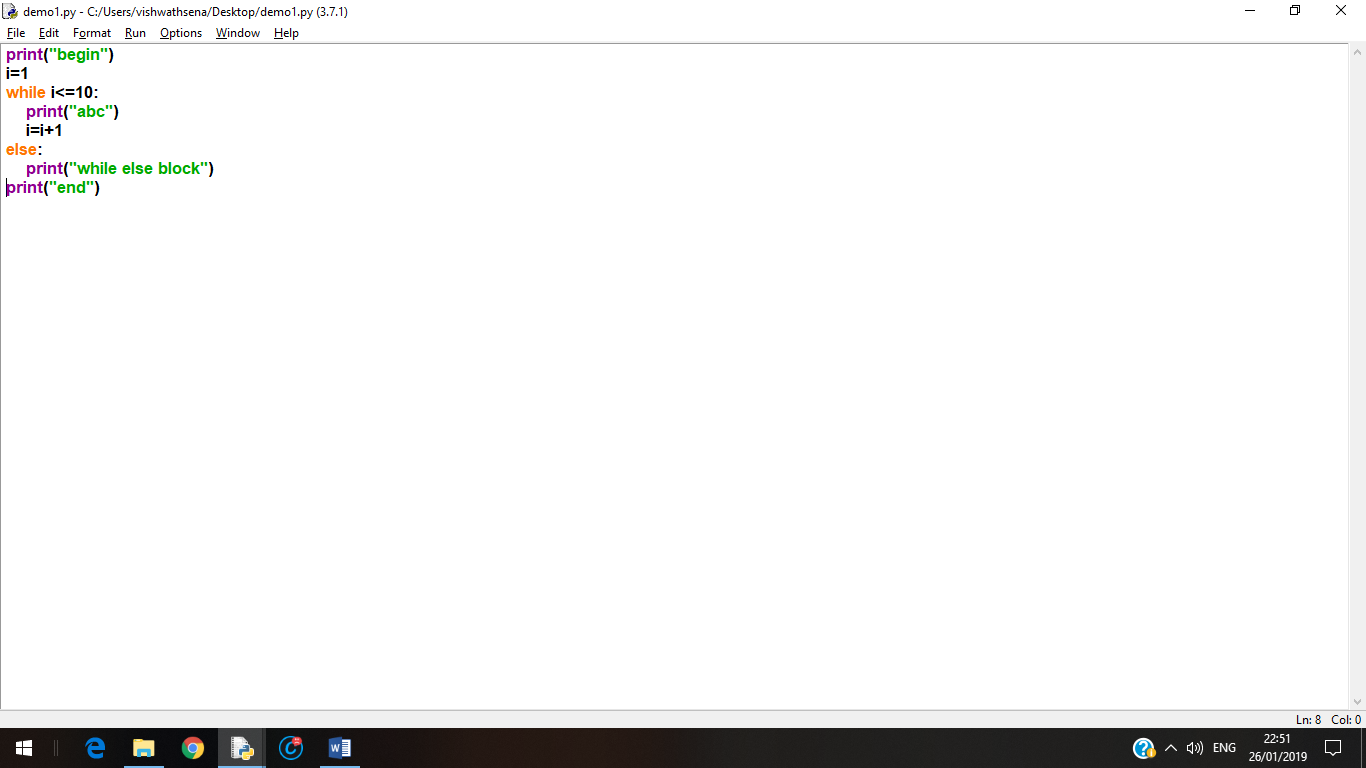
**Example2**:



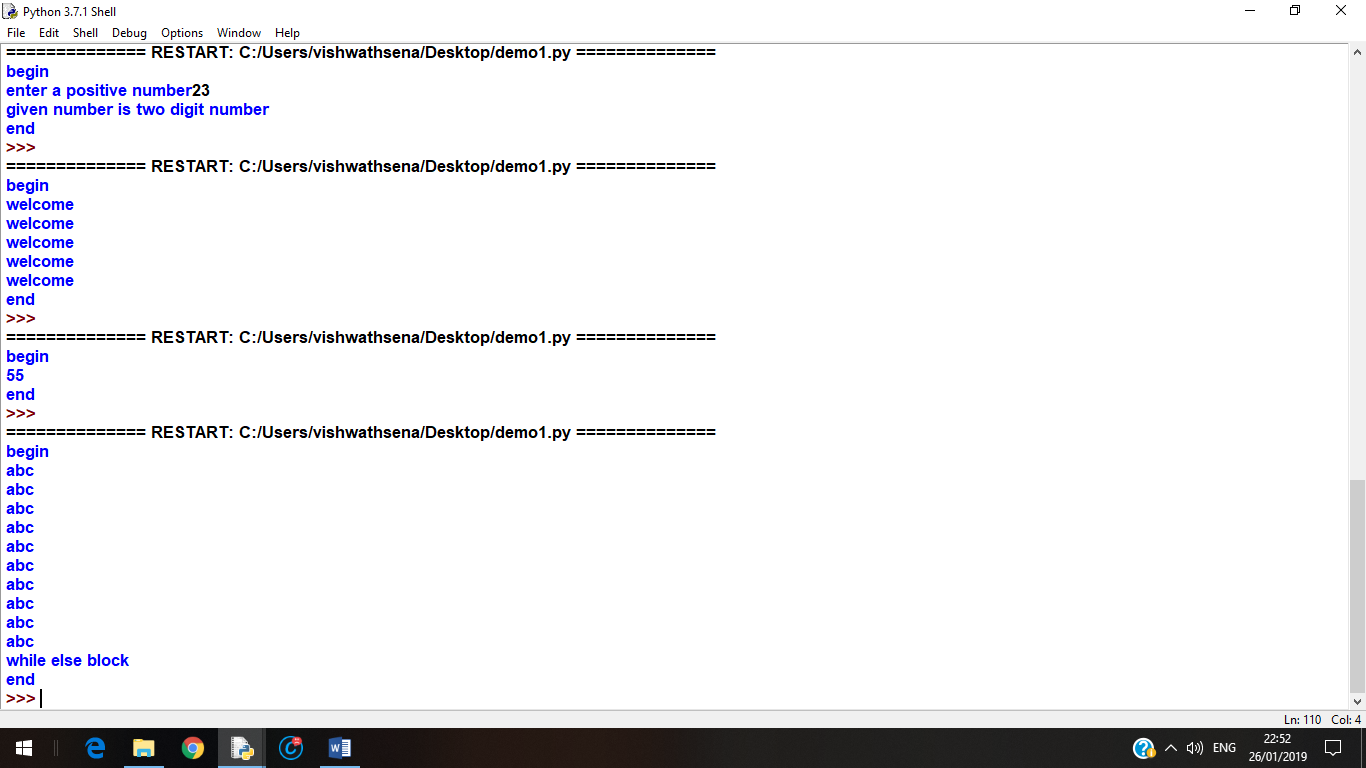
**Output**:



**Example 3**:



**Output**:



**2.for loop:**

For loop executes set of statements with respect to every element of given inerrable object.

**Syntax**: for variable in iterableobj:

…………………………..

**Example 1**: x=”abc” **output**: a

for p in x: b

print(p) c

**Example 2**: a=range(10)

for p in a:

print(p)

**outpu**t: prints 0-9

**Example 3**: b=range(10,20)

for q in b:

print(q)

**output**: prints 10 to 19

**Example 4**: c=range(30,40,2)

for r in c:

print(r)

**output**: 30,32,34,36,38

**Example 5**: d=range(40,30,-1)

for s in d:

print(s)

**output:** prints 40.39,38…..31

**Example 6**: z=”python is a language”

it=z.split()

for p in it:

print(p)

**output**: python

is

a

language

**loop control statements:**

Two loop control statements are there in python. They are:

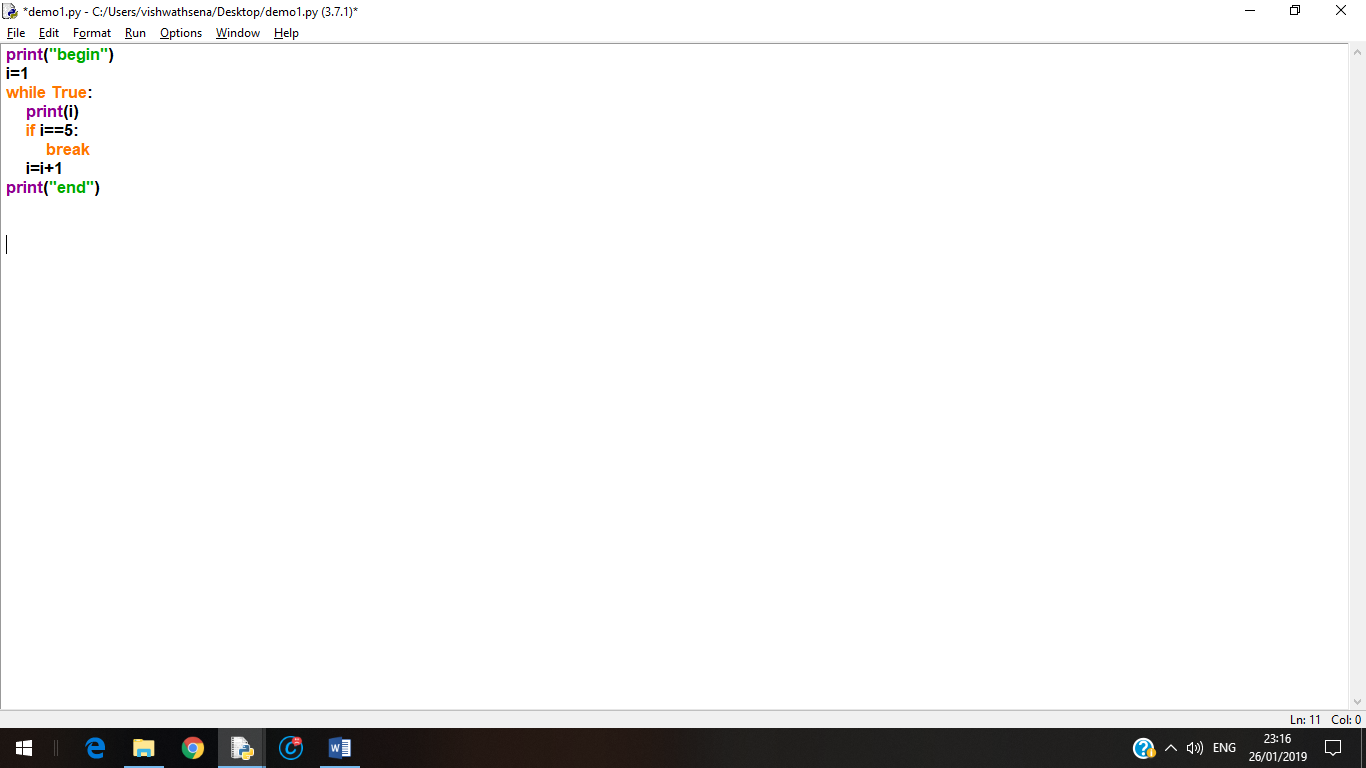
1. Break
2. Continue

**Break:**

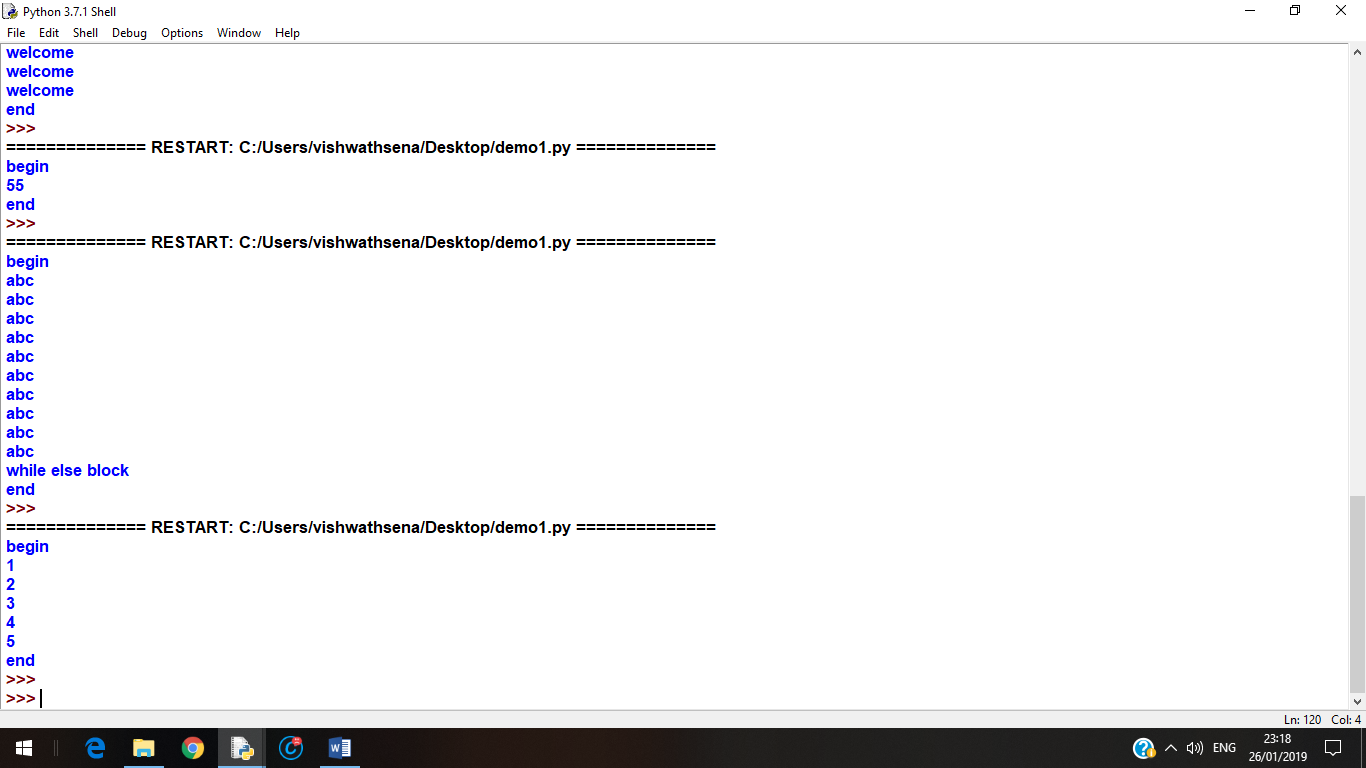
break is a statement ,which can be used in looping statements. Whenever control reached to the break statement of the looping statements then without executing the loop control comes out from the loop.

Generally we use break statement to exit from the infinite loops.

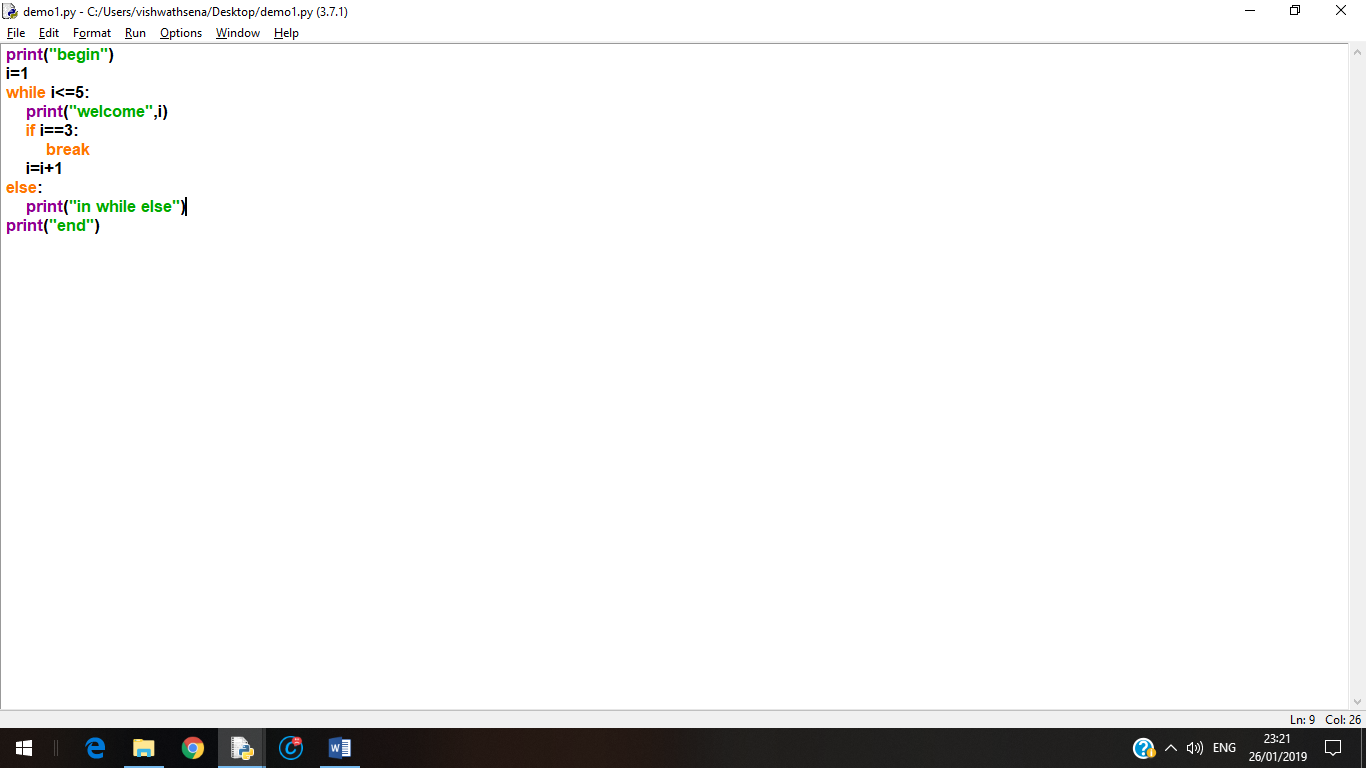
**Example 1**:



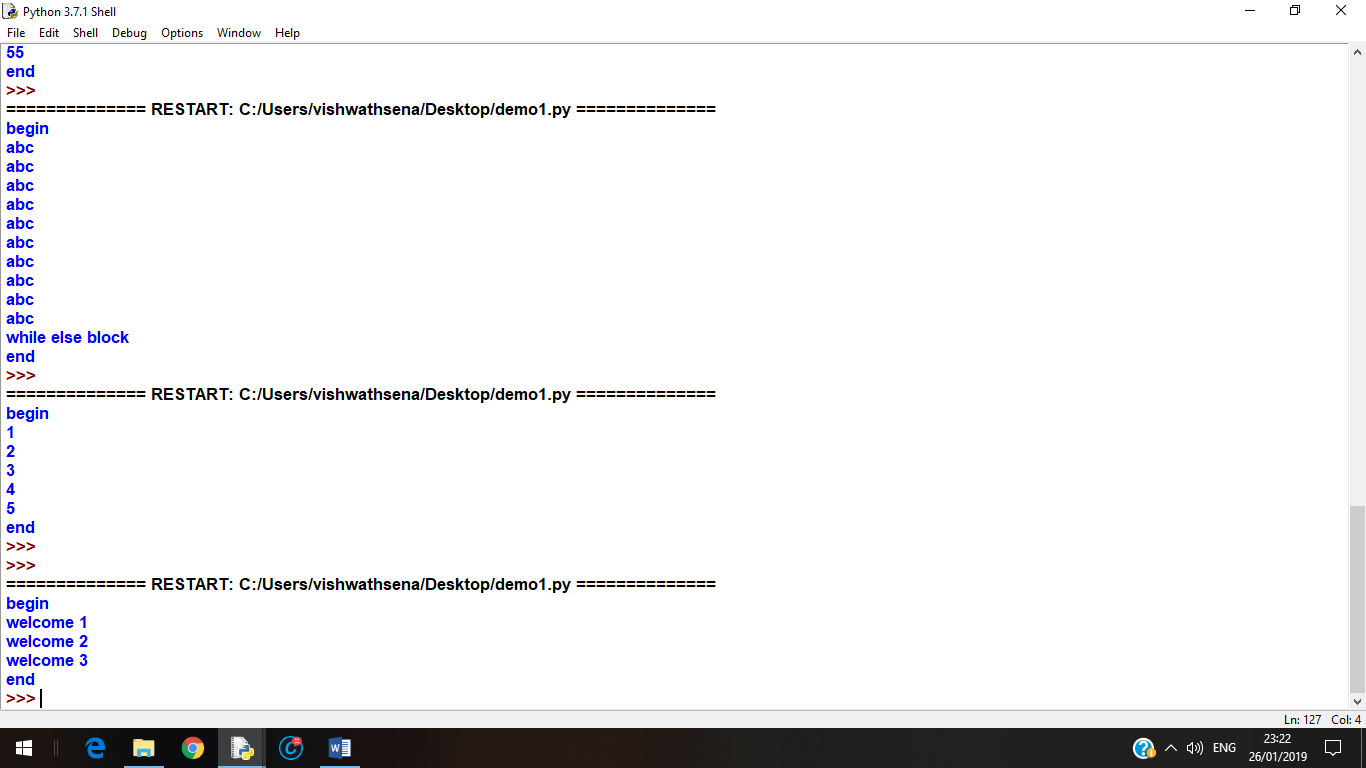
**Output:**



**Example 2:**



**Output:**

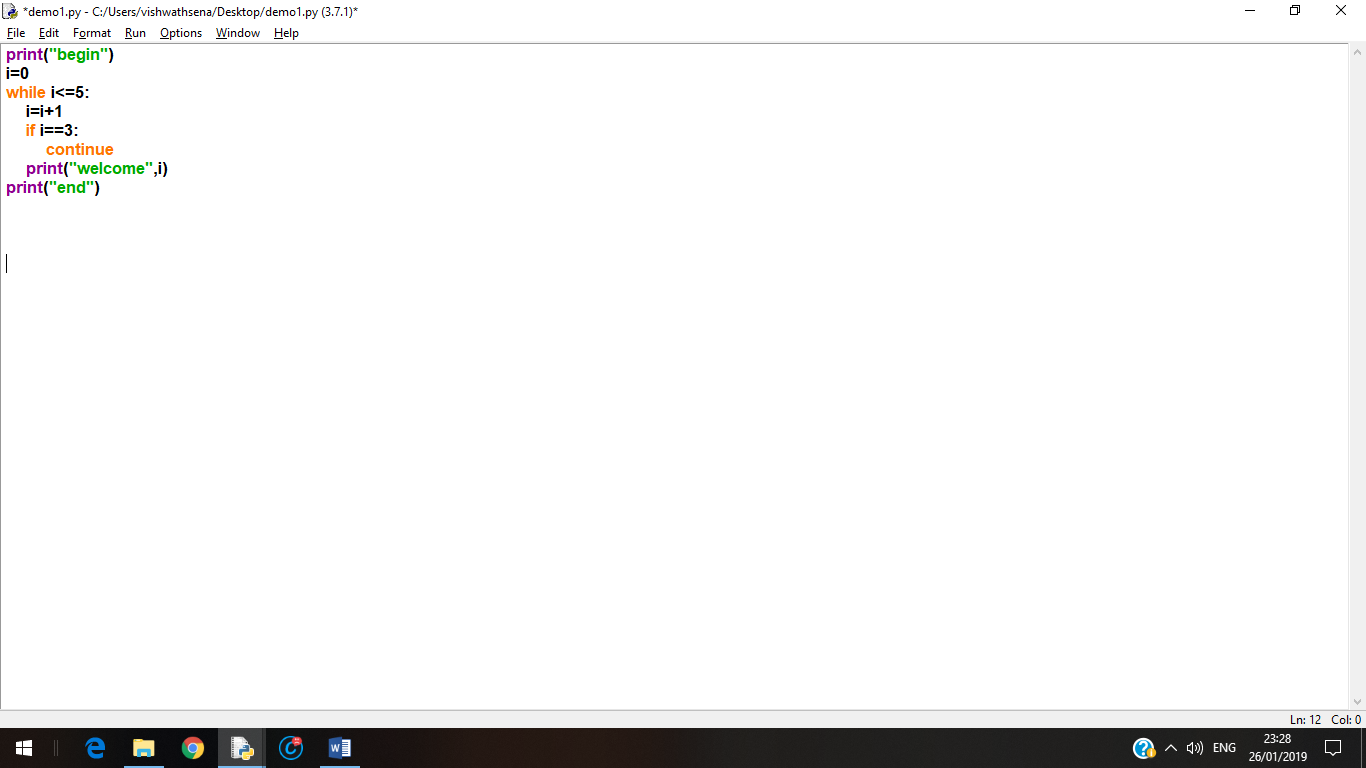


**Continue:**

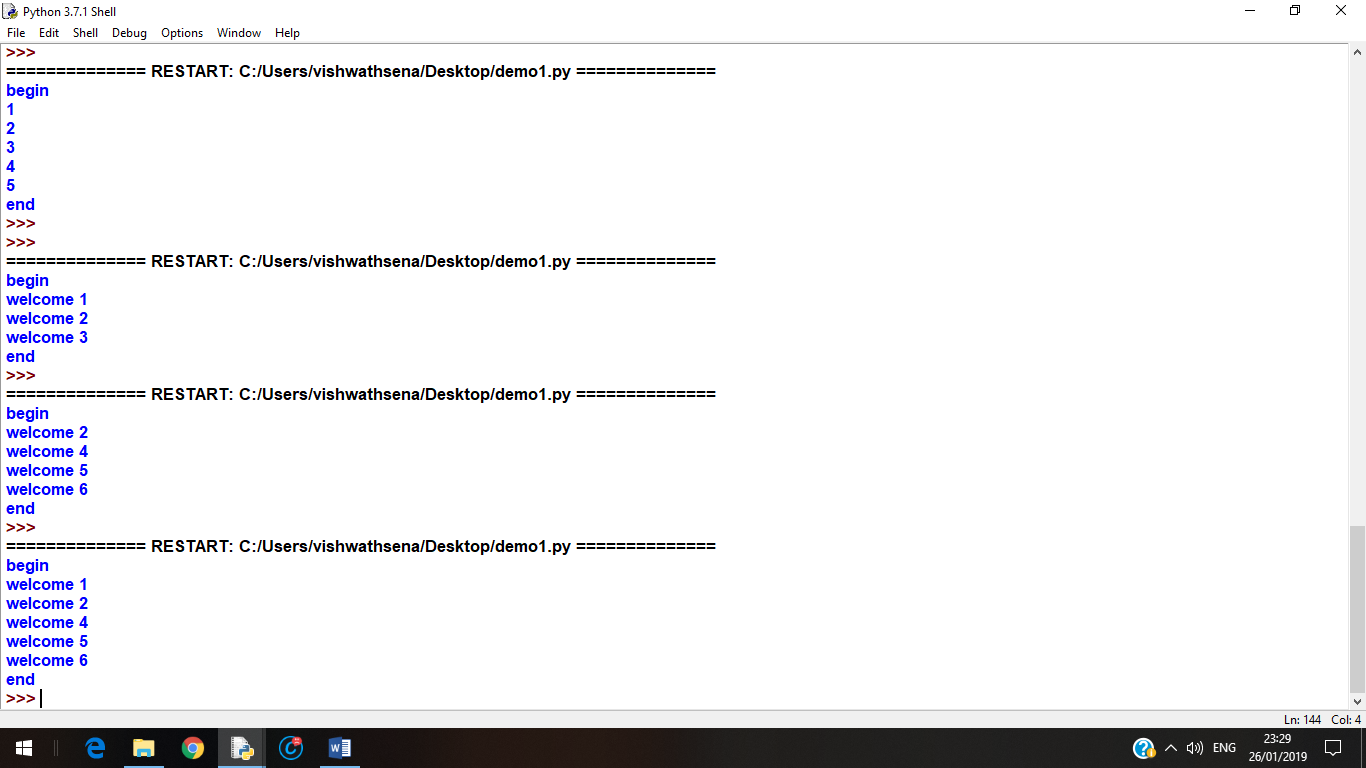
Continue is a statement, which can be used in looping statements.

Whenever control reached to the continue statements of loops. Without executing remaining statements of that iteration control goes to the next iteration.

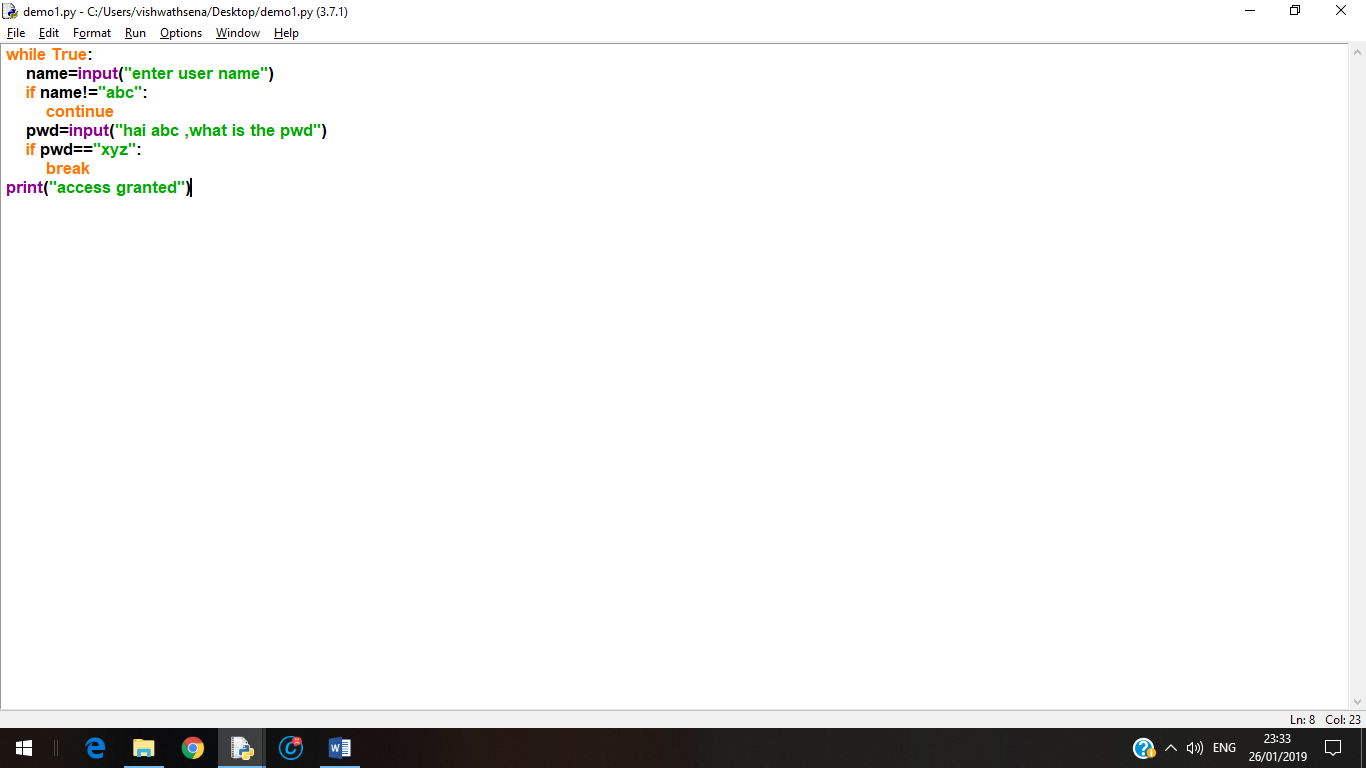
**Example**:



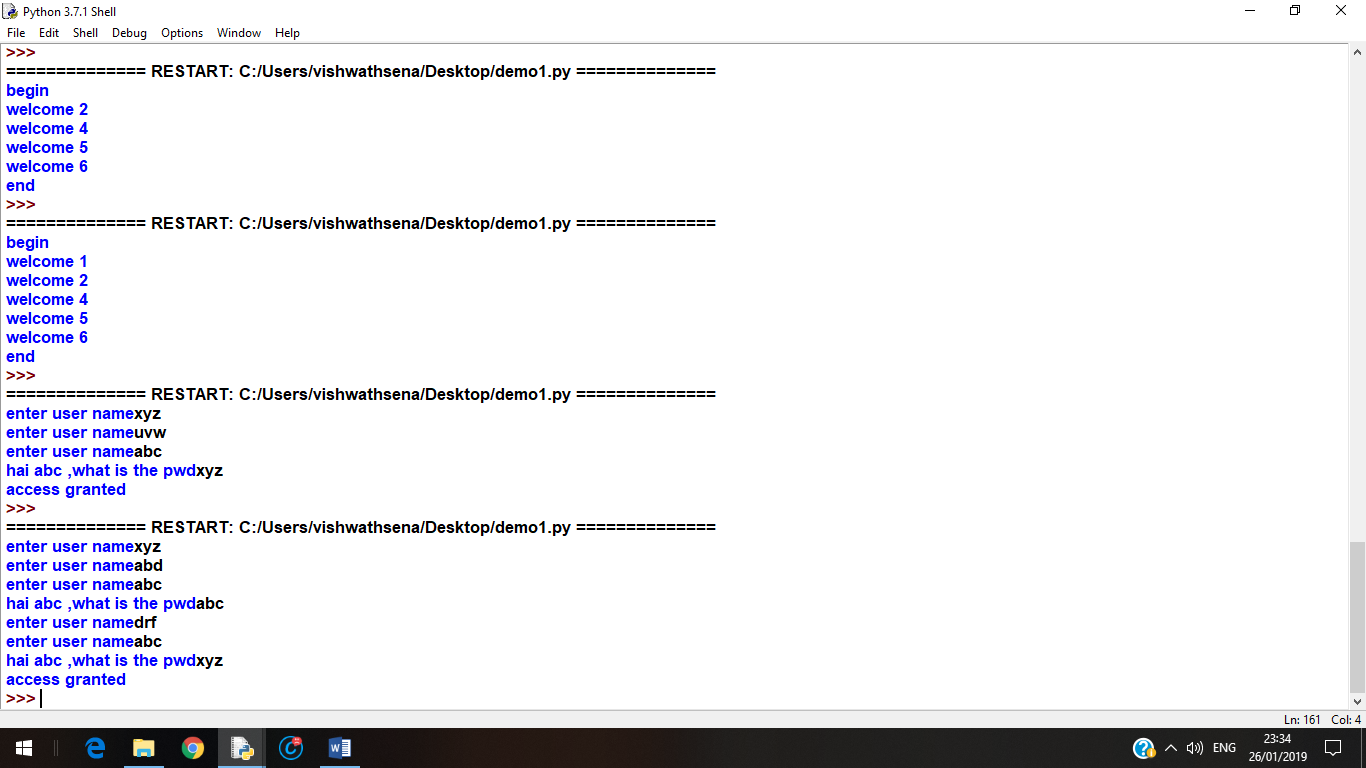
**Output**:



**Example 2:**



Output:



**PASS STATEMENT:**

It is used when a statement required syntactically but you do not want any command or code to execute.

The pass statement is a null operation. Nothing happens when it executes.

**Example**: for I in range(1,11):

if (i%2==0):

pass

else:

print(i)

the pass is also useful in places where your code will eventually go, but has not been written yet.

Example: for letter in ‘python’:

if letter==’h’:

Pass

Print(“this is pass block”)

Print(“current letter:”,letter)

Print(“good bye”)