

(Refer Slide Time: 00:01)

Meeting Topic: LPS Lecture-35
Meeting Number: 803 451 703
Date: Friday, March 07, 2014
Time: 11:15 AM, Local Time (GMT +05:00)
Host: Seer Akademi
Presenter: Seer Akademi
Participant: Seer Akademi, Anand

Table of Contents:

Recording Start	00:00:00
App/Desktop Share (1) Start	00:00:02
App/Desktop Share (1) End	00:59:36
Recording End	00:59:36

Hi everyone again welcome to the next lecture of the LPS class the LPS is essentially the programming languages that we are going to as you know we *ted with the UNIX the covered Perl then we moved on to TCL and TK explaining the some of the advanced concepts TCL in TK, now we are actually going to Python we already finished a couple of lectures. I hope you remember, I just wanted to recap what we did in the last lecture.

(Refer Slide Time: 00:53)

Recap

Chapter 1: Very Basic Stuff
Chapter 2: Conditionals
Chapter 3: Functions
Chapter 4: Iteration
Chapter 5: Strings
Chapter 6: Collection Data Types
Chapter 7: Advanced Functions
Chapter 8: Exception Handling
Chapter 9: Python Modules
Chapter 10: Files
Chapter 11: Documentation
Chapter 12: Classes
Chapter 13: CGI Programming

So we went through the very basic stuff it is essentially how do we initially write a Python program basically for just for printing something and then we talked about so in this one we had covered several topics like , how the variables are wrong what are the different types of variables how do you actually print them how do you actually input every inputs some value into the

variable and what essentially are the variables covered several key things and then we also went through like some operators what are the key words that are reserved in the language. Which you cannot use for variables even like variable name rules that we covered in the last lecture then we also talked about how we form those variables etc. And then we went to some conditionals in essentially for conditional if a less and more else. I hope you remember those things basically and we talked about that then we also talked about functions the functions are essentially the * with the D EF as for keyword and then followed with function name and then we go into the functions one thing we also talked about in that context is scope of a variable, how the scope cannot be cannot go into the function itself in variable and how Python actually and if you put the same name Python actually adds a new variable. And then well science values to that variable and then again the scope of the variable basically like finishes when only the extent of the function, so when it exits out of the function all the variables are gone it is all like going to be and CV we saw some examples as to how long this can be this affects your programming we also saw like how to get around this or by using this keyword called global inside the function with the variable name. So that it you can actually pass the variables that are outside of the function into the function and then manipulated , so I think that was a good introduction essentially like walking covered now I think we will go into today's lectures basically. I will be talking about the iterations so let us go into that so this is the new one that we are *ting today, so without much I do want to *t that chapter on iterations. (Refer Slide Time: 04:17)

Recap

- while loops
 - for loops
 - range function
 - Flow control within loops: break, continue, pass, and the "loop else"
- Chapter 8: Exception Handling
Chapter 9: Python Modules
Chapter 10: Files
Chapter 11: Documentation
Chapter 12: Classes
Chapter 13: CGI Programming

(Refer Slide Time: 04:19)

Chapter 4: Iteration

- while loops
- for loops
- range function
- Flow control within loops: break, continue, pass, and the "loop else"

So let us begin so the iteration consists of two major things basically while loops ,and for loops and then we also have a range function which we will talk about and then we need to also like we can do some controls in the inside the flow, so the flow control within the books such as break continue as and then the loop else, so we will talk about those things today.
(Refer Slide Time: 04:48)

while

- Example while.py

```
i=1
while i < 4:
    print i
    i += 1
```

Output:

```
1
2
3
```

So first of all the while loop essentially this repeats a statement or a group of statements while a given condition is full it tests the condition before executing the loop body, so once it tests it basically * running the loop body and then every time the body finishes it again checks for the condition that continue still valid if the condition is true it goes ahead and then does the loop until the condition becomes false the for loop we on the other hand that is actually it is execute

the sequence of statements multiple times and abbreviate the code that manages the loop variable, so the follow-up is simply like having say like okay.

I want it to be n number of times or m number ten and then basically like a minute the loop variable can be abbreviated to be a some fixed kind of quantity one thing with both these conditions are basically like there are nested loops basically you can use while followed by for all of my while so these kind of things can be like nested so one finishes then this will finish then the whole loop finishes the definition for the this flow control essentially the break statement this terminates the loop statement and transfers the execution to the statement immediately following be so whenever this break condition be very satisfied the loop completely is broken and then in the flow control just goes to the next statement pass to the loop.

The continue is more graceful in the sense that it causes the loop to skip the remainder of its body and immediately retest to the condition back to the retreading so say like I mean you have a wall and then you have XY V then if A=B and then you have break then you have a again CDEF this is a while loop when this A=B happens and then immediately like the loop is broken and then the condition is to be tested and goes back to here.

In the break actually it falls through and then goes to annex a deformable or something it goes to that whereas in a continued statement it breaks and then goes back the top and then it tested thing and then again boost around the wall now the past statement if this is used then a statement is required in systemically but you do not want any command or code to execute, so if you ask put an else then you can say like pass which is basically if A is not B then it just goes to C.

So this is basically like it just for the syntax purposes okay. So now let us look at some of these conditions.

(Refer Slide Time: 09:14)

while

- Example while.py

```
i=1
while i < 4:
    print i
    i += 1
```

Output:

1
2
3

So here example of the while loop we define is one while I less than 4 : we see print I and then I += 1 we know that this is the increment operator that we studied earlier, so the output will be like since I is 1 it becomes 1 and then here I gets implemented 2 and it goes back and again I less than 4 that is still true so it executes so it prints 2 and then it adds one more so I becomes 3 then it goes back and now still this condition is true so it goes down and it prints 3 and then now I equal to 4.

I becomes 4 then it goes to this loop and then it basically I more than false, so it just comes out so it only prints 1 2, 1 2.

(Refer Slide Time: 10:06)

for

- Example for.py

```
for i in range(3):
    print i,
```

output:

0, 1, 2

- range(n) returns a list of integers from 0 to n-1.
range(0,10,2) returns a list 0, 2, 4, 6, 8

Okay in example four for essentially so we specify for I in range three this is basically it is the same as for 0 to 3 and then we say print I and then a comma essentially like these are the two

things that you normally true so basically like in this case I printed as 0 because it is a 0 it *ts with you 1 & 2 and then when it is 3 it is out of the range ,so that it stops printing basically like a minute increments and then it becomes 3 it is the same for it goes outside so one of the key thing is this range of function.

So here is the definition range n opens the list of integers from 0 to n - 1 so here returns actually linkage is 0 to 2 if you specify one additional number for example here it is 0 10 2this returns in an increment of this in fact this so the incremental, incremental is already built in so this will return 0 2 4 6 & 8.

Because it is a * from 0 to 10 in a with an increment of 2 that is how we should played it so if you do not specify then it assume that it is 0 through n with an incremental of 1 increment of 1 so range this can be made into a shortcut as range N for anything else you need to specify all the numbers okay.

(Refer Slide Time: 12:06)

Flow control within loops

- General structure of a loop:
while <statement> (or for <item> in <object>):
 <statements within loop>
 if <test1>: break # exit loop now
 if <test2>: continue # go to top of loop now
 if <test3>: pass # does nothing!
 else:
 <other statements> # if exited loop without
 # hitting a break

So here is so much more complicated loop, so while some statement you can also define like or some item in object now there are some statements and then we do a test one and then cause it to break, so when test 1 is true it basically goes and then goes outside now the test 2-basically it is a continue so now when test 2 fails it basically goes back or test tube passes it goes back to the top of the loop and then we also use a passive which is a test 3 so when test three is true it does nothing and basically control portables.

(Refer Slide Time: 13:55)

Using the "loop else"

- An else statement after a loop is useful for taking care of a case where an item isn't found in a list. Example: search_items.py:

```
for i in range(3):  
    if i == 4:  
        print "I found 4!"  
        break  
    else:  
        print "Don't care about", i  
else:  
    print "I searched but never found 4!"
```

Handwritten notes: (0, 3, 1, 1) above range(3); 0, 1, 2 next to the else branch.

So here it is actually an else condition you so now let us look at the loop else the else statement after the loop is useful for taking care of the case where the item is informed in a list for example here search items .PY so for I in range 3 this 0 3 1 if I =4 then we print I Phone 4 and then break otherwise print do not care about I now obviously like I mean in this case own. I will never before so after the loop after going through the three iterations it is going to exit the loop without anything because here basically it is like it says.

I do not care about and I so do not have care about 0 1 2 all those things get printed out by the statement but now we have to also like print out so basically like an import for this particular loop if there is an else basically then you can also log on the ones never hit the break and we can actually let do something, so here we put that break so if it does not hit the break in the else gets printed and that says basically. I searched but never found for so this is the example for using the loop else. Which is an interesting command or in Python.
(Refer Slide Time: 16:08)

for ... in

- Used with collection data types (see Chapter 6) which can be iterated through ("iterables"):

```
for name in ["Mutasem", "Micah", "Ryan"]:  
    if name[0] == "M":  
        print name, "starts with an M"  
    else:  
        print name, "doesn't start with M"
```

- More about lists and strings later on

In more details so one thing is basically like, I mean you can use the for loop with collection data types the collection data types are explained in the next chapter and this can be iterated through what are known as each of the iterables so for example, here the simple one is for name in this particular collection, so and if the first letter which is denoted as the name 0 is M then we say print *ts with M including the name otherwise the print name does not *t it M so this prints for Mutasem and Micah prints first one and then for Ryan it is infinite form so we will learn about some of these lists and strings from the little part of the this lecture.

(Refer Slide Time: 17:42)

Parallel traversals

- If we want to go through 2 lists (more later) in parallel, can use zip:

```
A = [1, 2, 3]  
B = [4, 5, 6]  
for (a,b) in zip(A,B):  
    print a, "*", b, "=", a*b
```

output:

```
1 * 4 = 4  
2 * 5 = 10  
3 * 6 = 18
```

So now let us see what kind of what are parallel services.

(Refer Slide Time: 18:09)

Chapter 5: Strings

- String basics
- Escape sequences
- Slices
- Block quotes
- Formatting
- String methods

$2 * 5 = 10$

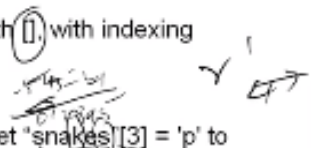
So for this we use another command called the zip for doing the parallel traversal.
(Refer Slide Time: 18:11)

More string basics

- Type conversion:

```
>>> int("42")  
42
```
- An empty string is denoted by having nothing between string delimiters (e.g., "").
- Can access elements of strings with `[]`, with indexing starting from zero:

```
>>> "snakes"[3]  
'k'
```


- Note: can't go other way — can't set "snakes"[3] = 'p' to change a string; strings are *immutable*.
- `a[-1]` gets the last element of string `a` (negative indices work through the string backwards from the end).
- Strings like `a = r'c:\home\temp.dat'` (starting with an `r` character before delimiters) are "raw" strings (interpret literally).

So what is the parallel traversal use things together we say like an A B for A B in zip these two lists A and B we can say like print A then the * B is A times B, so now what Python does is it moves these two together ,these together, these two together, so that is where we can traverse two lists in parallel this is all very useful because, so when we have like coordinate systems there you need to come up quickly with the addition of two ,two coordinates you can actually like do it in just one command other than actually spending multiple commands.

So here the output is going to be like 4 10 and 18 because it is multiplying 1 times 4 then 2 times 5 and then 3 times 6 ,so that is pretty much that covers on the integrators and how we can perform variations now we will go into the strings as a variable type so we will talk about the

string basics the escape sequences slices it is another key concept and then we will use we will do some barcodes some formatting and then the string methods.

So strings are delimited by single or double quotes the both of this we saw in TCL one thing is basically that Python is actually uses Unicode, so strings are not limited to disk as we practice so Unicode has much more taxes than just a character's ASCII characters amounts to probably about 65 of them so here we can use much more than that the empty string is denoted by having nothing in between the dealing.

So we are singly could delimiter nothing between that that is an string we can access the elements in the string with the [] the indexing *ting from zero so if you say like I mean say string and then the third element actually it is 1 0 1 2 3 so that is K a gets printed out ,and then the other thing is even the strings are immutable we thought or the variables are immutable within that we do not change the value of variables here is the same thing strings are immutable so that you cannot assign a snakes 3 to P.

And expect that that work because stinky is always better to access the last element of a string we can have two ways basically you can borrow for example, this snakes gone off with this one that is a 0 1 2 3 4 5 and say this makes 5 or you can also say negative one so the - one gets the last little box and then the negative integers work backwards so with this - 1 - 3 X 4 X 5 so they work this way then positive.

So the strings actually also have the concept of being raw string and usually the raw *ts of the character are before the delimiter, so if you stay like R then the delimiter some string that is a raw string and they are interpreted literally, so there is you cannot change anything that.

(Refer Slide Time: 23:55)

More string basics

- Type conversion:

```
>>> int("42")  
42  
>>> str(20.4)  
20.4
```
- Compare strings with the is-equal operator, ==
(like in C and C++):

```
>>> a = "hello"  
>>> b = "hello"  
>>> a == b  
True
```
- ```
>>> location = "Chattanooga " + "Tennessee"
>>> location
Chattanooga Tennessee
```

Now you can convert any numbers into the string and vice versa for example ,we can say like in of string 42 that is the actual value positive and then string of 20.4 now appears within the codes this you can also compare strings with is - equal operator == this is just like in C and C++ so we do not have this right specific wave command for the strings in Perl we saw directly like this so to write that basically we just say with A=B and then that returns true .  
 Now a good thing in Python is that icon provides some the way to increment the or as two strings & string to another one by just using this addition symbol, so if you do a Chattanooga + Tennessee now and we want to print the location just things like with the Escape.  
 (Refer Slide Time: 25:57)

| Escape sequences       |                                          |
|------------------------|------------------------------------------|
| Escape                 | Meaning                                  |
| <code>\n</code>        | newline                                  |
| <code>\t</code>        | tab                                      |
| <code>\N{d}</code>     | unicode dbase id                         |
| <code>\Uhhhh</code>    | unicode 16-bit hex                       |
| <code>\Uhhhh...</code> | Unicode 32-bit hex                       |
| <code>\xhh</code>      | Hex digits value hh                      |
| <code>\0</code>        | Null byte (unlike C, doesn't end string) |

Now let us look at some of the escape characters and what their meanings are so the escape sequences so the \ is a very common one that is basically for escape and then the meaning is just backlash for \ , \ will be this \ so that is so other thing now the quotation is also letting seen so the escape sequence is the \ and if you say like I mean \ n improbable new line \ t is for tab now \ N and ID is the Unicode database ID for that particular, now we can also represent the 16-bit x by this going like escape under escape and then lower case move followed by the hexadecimal number.

If you specify an uppercase followed by x additional number it treats that as a Unicode 32-bit and then if you specify like \ X and then H, H and that is the H digit some notation or values for H H and then \ zero well good so how does it end this trip and then if you just specify like at NNN this is actually octal notation well that is in this within n has to be from zero to same you not till that is how it is then some other sequences also support.

If we for example \c x is a shorthand for control X and then there is also like a vertical tab that you can instantiate which is \ instead of \ T similarly like the plenty of other characters because it is a uniform it has a rich set of various characters.  
(Refer Slide Time: 29:11)

## Block quotes

- Multi-line strings use triple-quotes:  

```
>>> lincoln = """Four score and seven years
... ago our fathers brought forth on this
... continent, a new nation, conceived in
... Liberty, and dedicated to the proposition
... that all men are created equal"""
```

Now let us talk about the block code some we can use triple codes to represent multiple lines groups so here we have a triple line triple quotes then basically like all the way up here this entire thing is treated as this one string 104 Lincoln but it works because of capitals triple quotes here.

(Refer Slide Time: 29:41)

## String formatting

- Formatting syntax:  
format(%object-to-format)  

```
>>> greeting = "Hello"
>>> "%s, Welcome to python." % greeting
'Hello, Welcome to python.'
```
- Note: formatting creates a new string (because strings are immutable)
- The advanced printf-style formatting from C works. Can format multiple variables into one string by collecting them in a tuple (comma-separated list delimited by parentheses) after the % character:  

```
>>> "The grade for %s is %.1f" % ("Tom", 76.051)
'The grade for Tom is 76.1'
```
- String formats can refer to dictionary keys (later):  

```
>>> "%(name)s got a %(grade)d" % {"name": "Bob", "grade": 82.5}
'Bob got a 82'
```

Now we talked about the concatenation of vapor before I go into the string there is also a repeater operation which is essentially the \* similar to TCL so if you say like my name and 2 then it means we should be my name so now one thing that I also want to monitor this end is we

talked about this things you can actually like specify them as slices so you can have like the small slice of 0 3 which basically is here XL is AK so those kind of slicing is also possible. Now let us go talk about the string formatting you and this is percentage reading so three places this basically below here so it becomes hello welcome to Python the formatting actually creates new string because as you know the screens are immutable so you cannot change it so the polymer actually creates a new string so if print style formatting also works we can format multiple variables into one string by collecting them into a trip so basically a comma separated list maybe parentheses and after the percentage character.

So here we say basically the grade for percentage S is percentage 4 1f so here we can actually see how we will get translated.

(Refer Slide Time: 33:00)

## Stepping through a string

- A string is treated as a collection of characters, and so it has some properties in common with other collections like lists and tuples (see below).
- ```
>>> for c in "snakes": print c,
...snakes
```
- ```
>>> 'a' in "snakes"
True
```

So you are basically in specifies then this number gets converted into 4 + 1 so that is 76.1 so basically ligament it takes and then based on bone format string need or move it so here it is four digits prior to 0 and then 1 5 to 0 so it formatted and 36.1 it rounds it the string formats can also refer to the dictionary keys so again here percentage name the Bob it grade something and then named Bob grade and 82.5 and then again it is a integer so it basically two.

So some more formatting basically percentage these four character percentage s for string conversion we have STR prior to the formatting so we STR sub number and then basically that is formatted but it I is for sign decimal integer posted HD for unsigned decimal integer and then % uppercase X is for the hexadecimal and then the lower case x is for the lowercase lower integer or lower case letters which are used inside the hexadecimal so if you want to you if you want to format this into a or assailing 15.

The first one which is the uppercase X will format this as uppercase F and then the lowercase will form of local.  
(Refer Slide Time: 36:08)

## Stepping through a string

- A string is treated as a collection of characters, and so it has some properties in common with other collections like lists and tuples (see below).
- `>>> for c in "snakes": print c,`  
`s n a k e s`
- `>>> 'a' in "snakes"`  
`True`

So now let us look at how we can step through the string we already saw this so from all those stuff so we have to treat the string as a collection of characters so it has some property in common with the other collections like lists and tuples , so we can step to this for example we can define you will C in the string print C that prints one lip one character at a time because that is what they C takes the value of and we can also do some checking basically if A in snakes then that consists true.  
(Refer Slide Time: 34:14)

## Slices

- Slice: get a substring from position i to j-1:  
`>>> "snakes"[1:3]`  
`'na'`
- Both arguments of a slice are optional; `a[:]` will just provide a copy of string a

Now let us look at the slices in some more details so here we can actually do takes one column three that returns basically the first two actually like this second and the third one that is N and here actually like both are optional PCP if you specify just a and then without anything it just provides a copy of the string.

(Refer Slide Time: 37:50)

## String methods

- Strings are classes with many built-in methods. Those methods that create new strings need to be assigned (since strings are immutable, they cannot be changed in-place).
- S.capitalize()
- S.center(width)
- S.count(substring [, start-idx [, end-idx]])
- S.find(substring [, start [, end]])
- S.isalpha(), S.isdigit(), S.islower(), S.isspace(), S.isupper()
- S.join(sequence)
- And many more!

Okay, So let us talked about the string methods so basically and these are all built-in methods creates new strings that create the new strings that may to be assigned basically need to assigned a string variable to these two methods, basically when you using the method and it goes into the particular variable and then the string are immutable you cannot change basically place that is a reason assigned into new variable.

So let us look at some of these methods manipulate strings so the first one is capitalized this capitalized basically it capitalizes the first letter of the string so we capitalize this first now what about Center the center actually has an argument width we can also provide another argument for fill character.

And what it does is it returns a space padded string Center to the total width cause this is total of the width cause, so whatever that width that be specifying and we can actually specify these filters if you do not specify it is as big otherwise you can it can fill with whatever that is, so it treats the string it expands the string and puts those fill characters in the middle, and it expands it to fill the width ascension.

So this is another nifty command, now you can also say account we give like a substring a star a start index and the end index basically, here it counts how many times this substring occurs in the string, or in a substring of the string if starting index beginning and in the ending index and



arguments, so these are two optional ones, so if you do not specify and basically count substring it just sees how many times the substring occurs in the string.

If not it actually now is basically like it mean it also meaning like if you are giving both the starting books at the end index, it starts from that start index and ends at the end index, and look for the substance, so it takes a substring and then massage a stack for to find the custom specification inside the dot, then we also have fine and the find it also again a substring start and end the these two are optional, the find again if we determine if the this particular substring occurs in a string.

Or in a substring of the string starting from start to the end and it returns the index if found, otherwise it returns - 1, so it returns index where this thing the substring is occurring or - 1, then there are other, other one also we can also like decode and encode strings we can along with this we can also see another one which is ends with which is ends with it has a suffix beginning and ending, this actually determines if the string or the substring of string ends with a suffix.

Returns true if into the suffix or it returns false, false other words if digit again it is the opposite of that basically, where it returns the true if the string contains only digits and false otherwise, if lower is basically if that mean is at least one case character and all taste characters are in the lower case, and then we also have these numeric is another one, so that is and then the join is basically to take multiple sequences and join them so and there are others also.

(Refer Slide Time: 45:12)

### replace method

- Doesn't really replace (strings are immutable) but makes a new string with the replacement performed:

```
>>> a = "abcdefg"
>>> b = a.replace('c', 'C')
>>> b
abCdefg
>>> a
abcdefg
```

So we also have another method on the replays, it really does not replace again go back to our first principle which is strings are immutable, but it makes new strings with the replacement performed, so here is an example we define a variable string variable A with ABCDEFG and



then another variable B which is now we are going to call it as A. replace C with appetizing, so now if you want to print B it is basically gave in appetizing ABCDEF, whereas the A is simply in ABCDEFG okay.

(Refer Slide Time: 46:17)

### More method examples

- methods.py:  

```
a = "He found it boring and he left"
loc = a.find("boring")
a = a[:loc] + "fun"
print a

b = "and" + join(["cars", "trucks", "motorcycles"])
print b

c = b.split()
print c

d = b.split("and")
print d
```

Output:  
He found it fun  
cars and trucks and motorcycles  
['cars', 'and', 'trucks', 'and', 'motorcycles']  
['cars', 'trucks', 'motorcycles']

So and then there are more examples, of the methods, so we go through this one, so here we can actually like say that basically so that we define a variable A this whole sentence and then now we say like I mean find essentially, so we define the location to be a find boring so right here, so whatever the number of characters to this that stored in the location, now what we need to do is you should add that whatever the location we add fun to it.

And then now we print A actually A it should be the replay command and essentially we can be hard to replace this with nothing, and then add convert and then we define another 1, B with this and then we say like join cars, trucks and motorcycles, so the and they supply to join these the different strings, so it becomes cars and trucks and motorcycles, then if we say like I mean so now see is depend as a split of B or B dot split and we print C it prints each one and a certain element in the string list or string collection.

And then if you specify again like I mean A split the string D, but with this particular string so then it takes out that string and only reports cars, trucks and motorcycle, so here we are using like several fine that is also like replace then and then join and then split, so all these are different string methods.

(Refer Slide Time: 49:29)

## Regular Expressions

- Regular expressions are a way to do pattern-matching. The basic concept (and most of the syntax of the actual regular expression) is the same in Java or Perl

Now let us talk about the regular expression, regular expressions are a way to do pattern matching the basic concept is pretty much it is the same as Java or Perl that we already are we are used to this.

(Refer Slide Time: 49:52)

## Regular Expression Syntax

- Common regular expression syntax:
    - . Matches any char but newline (by default)
    - ^ Matches the start of a string
    - \$ Matches the end of a string
    - \* Any number of what comes before this
    - + One or more of what comes before this
    - | Or
    - \w Any alphanumeric character
    - \d Any digit
    - \s Any whitespace character
- (Note: \W matches NON-alphanumeric, \D NON digits, etc)
- [aeiou] matches any of a, e, i, o, u
- junk Matches the string 'junk'

amf  
amf  
x  
xxxxx

So the common regular expression in a syntax the dot matches any character but new line, then the character matches the beginning of the string, and then \$ matches the end of the string, so these are the things that we already know from Perl and we carried over to TCL also for the same symbols and then a star is any number of whatever comes before this so X \* will be like X, X, X, X whatever may be and + is essentially like only one item, so it is A + all this will A and matches only those.

Now there is also like or function basically which means this or it is either the first string or defining string and then when we escape W that is basically again you know characters, and then escape D becomes any digit and escapes becomes the non white space, and then upper cases are just exactly opposite of peace so that we just have to just remember that, proper case W matches non and de-non digits and then upper cases is a non white space. And then if you specify it in the square bracket basically one AEIOU it matches any of AEIOU so any letters you can match, but if you do not specify the junk matches the string junk. (Refer Slide Time: 51:50)

### Simple Regexp Example

```
import re #import the regular expression module
infile = open("test.txt", 'r')
lines = infile.readlines()
infile.close()

replace all PSI3's with PSI4's
matchstr = re.compile(r"PSI3",
re.IGNORECASE)
for line in lines:
 line = matchstr.sub(r'PSI4', line)
 print line
```

So some Regex examples essentially so here this is basically like we have seen more import RE stands for the importing the reviews expression module, so once we export the regular expression module, so we specify basically like in file by opening this text test or text in a read mode, and then we pour the lines essentially in again we start reading the line with this method read lines, so just watch out I will be talking about some of these things basically like the method, but this will be like coming in the future but this is again another method you can see read lines with method that is applicable for file structure. And then that is why we are calling the in file and then infile close, and then once we reread all the lines maybe a say before you focus, now we can actually replace all PCI 3 with PCI 4, so the way that we can do is like you can save match string, the regular expression compile so this is the method for the regular expression and then basically like it is a raw string that is specified as a PSI 3 and then we also specify another method the ignore case in RE, so it ignores it and it composes.

And then for the line in the line so the lines that we read here, we use the match string and then we say like sub R PSI 4 and then the line and then we print that post lines.  
(Refer Slide Time: 53:53)

### Simple Regexp Example

- In the previous example, we open a file and read in all the lines (see upcoming chapter), and we replace all PSI3 instances with PSI4
- The regular expression module needs to be imported
- We need to "compile" the regular expression with re.compile(). The "r" character here and below means treat this as a "raw" string (saves us from having to escape backslash characters with another backslash)
- re.IGNORECASE (re.I) is an optional argument. Another would be re.DOTALL (re.S); dot would match newlines; by default it doesn't). Another is re.MULTILINE (re.M), which makes ^ and \$ match after and before each line break in a string

So basically like I mean this means that we read all the lines and we replaced any anywhere where we can see with PSI 3 with PSI 4, couple of things about the example we needed to compile the first of all the regular expression module needs to be imported that is the using the input command, then we also need to compile the regular expression with the RE compile, and then the our character is means the raw string that we already saw that this is, so we do not have to escape every special character.

And then IGNORECASE is another optional argument, other would be like a dot all basically, which is like it matches all the strings essential, we also have another one called the multi line which is REM, which makes the caret and dollar match after and before each line.

(Refer Slide Time: 55:07)

## More about Regexp

- The `re.compile()` step is optional (more efficient if doing a lot of regexps)
- Can do `re.search(regex, subject)` or `re.match(regex, subject)` as alternative syntax
- `re.match()` only looks for a match at the beginning of a line; does not need to match the whole string, just the beginning
- `re.search()` attempts to match throughout the string until it finds a match
- `re.findall(regex, subject)` returns an array of all non-overlapping matches; alternatively, can do for `m` in `re.finditer(regex, subject)`
- `Match()`, `search()`, `finditer()`, and `findall()` do not support the optional third argument of regex matching flags; can start regex with `(?)`, `(?s)`, `(?m)`, etc, instead

So now let us talk some more about the regular expressions, the regular expression compiled is an optional step it is more efficient if we doing lot of regular expressions, we can also do this regular expression search and then very extend the subject this equal or even regular expression match that is also we can do that is an alternative syntax for without doing the compact, and then the match only looks for match at the beginning of the line, it does not need to match the whole string, there is the beginning, the search attempts to match throughout the string until it finds the match.

The find all regex the subject is the array of all non-overlapping matches, so alternatively we can do basically fine filter, with regex and object and the subject, and then the other thing that to notice also the match, search, find iteration, find all they do not support an optional third argument for the regex matching plans in fact regex with the person question mark one goes majestic.

(Refer Slide Time: 56:50)

## re.split()

- `re.split(regex, subject)` returns an array of strings. The strings are all the parts of the subject besides the parts that match. If two matches are adjacent in the subject, then `split()` will include an empty string.
- Example:  

```
line = 'Upgrade the PSI3 program to PSI4. PSI3 was an
excellent program.'
matchstr = re.split('PSI3', line)
for i in matchstr:
 print i
>>> Upgrade the
>>> program to PSI4.
>>> was an excellent program.
```

So now we come to the split command, so the regex split basically returns an array of the strings, so we saw that basically like how like, the strings are all the parts of the subject besides the parts that one is if two matches are adjacent in subject then the split will include an empty string. So here we go an example like I mean in line for set us upgrade the PSI 3 PSI 4 PSI 3 was an excellent program, so we here we say like match string PSI 3 we get a match and then for each of these match strings we just print I. So, so the result will be this much and then it gets this much, and then it also splits here so this is also gone and then here, so now you can see how is the I has been printed basically upgrade the one then program to PSI 4 is 1 was an excellent program is being third one, so let us actually continue the, the remainder of the strings in the next lecture so there are a few more things that we need power before we can go into the next chapter okay thank you very much and then we will talk more.