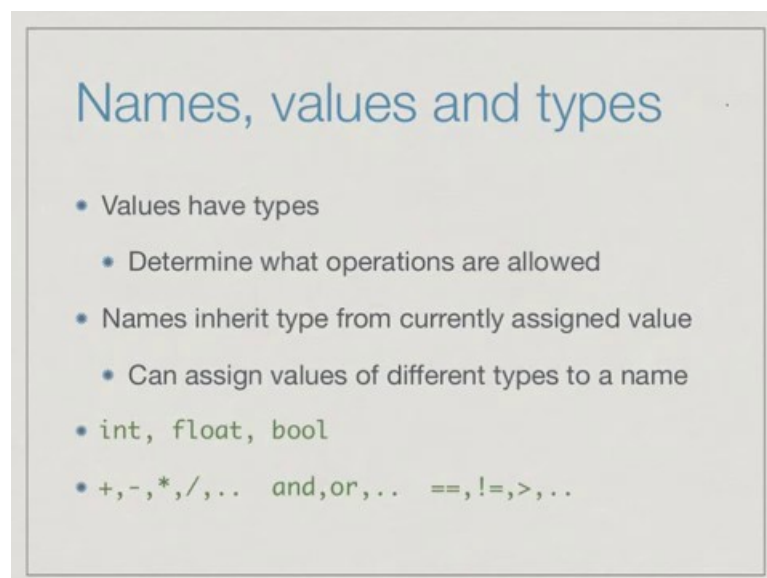


Programming, Data Structure and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture - 02
Strings

We have seen now that python uses names to remember values. Values are the actual quantities that we manipulate in our program, these are stored in names. Values have types, and essentially the type of a value determines what operations are allowed.

(Refer Slide Time: 00:02)



The slide is titled "Names, values and types" in a blue font. It contains a bulleted list of concepts in Python:

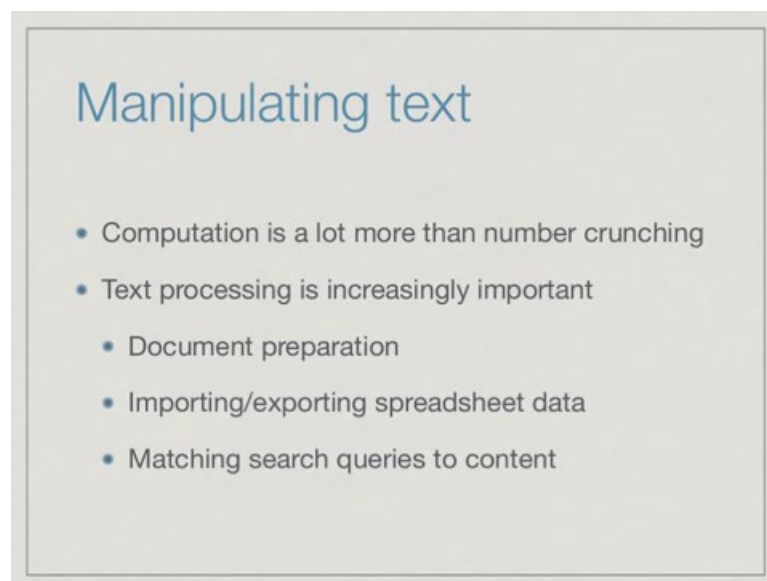
- Values have types
 - Determine what operations are allowed
- Names inherit type from currently assigned value
 - Can assign values of different types to a name
- `int`, `float`, `bool`
- `+`, `-`, `*`, `/`, `..` and, `or`, `..` `==`, `!=`, `>`, `..`

The types we have seen are the basic numeric types - `int` and `float`, and the logical type `bool` which takes values `true` or `false`. So, for the numeric types, we have arithmetic operations, we also have other operations which are more complicated. For the Boolean types we have `and`, `or`, `not`, which allows us to manipulate `true` and `false` values. And then we have these comparison operators `equal to`, `greater than` and so on, which allows us to check the relative values of two different quantities, and decide whether they are in some order with each other.

The important thing that we said was that in python the names themselves do not have a

fixed type. So, we cannot say that `i` is of type `int` or `x` is of type `float`, rather it depends on what values assigned and in particular, if a name is used for the first time without assigning a value then python will **complain**. We do not have to announce names in advance like other programming languages, but whenever we first use a new name; its first use must be in an assignment statement on the left hand side. So, before we use a name in an expression on the right hand side it must be assigned a valid value.

(Refer Slide Time: 01:36)

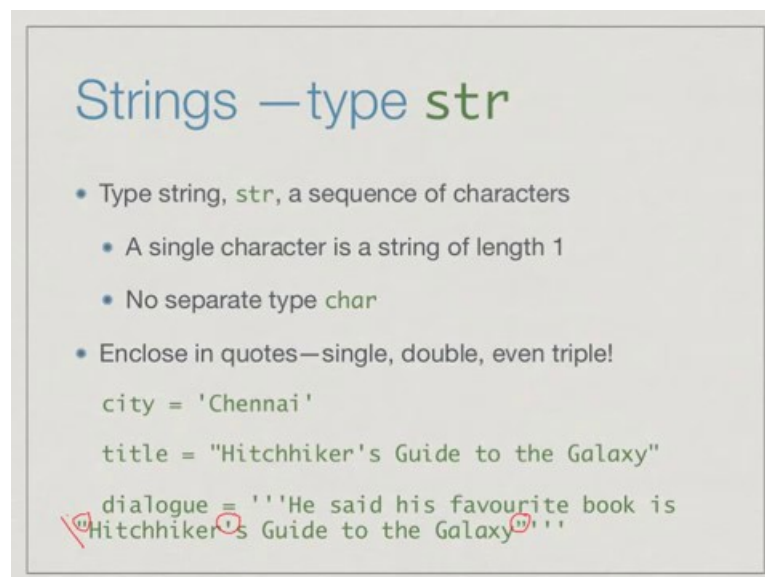


Numeric types by no means the only things that are of interest these days in computation. A lot of the computation we do is actually dealing with text. So, whenever we prepare a document, for example, using a word processor or some other things for presentation, then we are actually manipulating text; so **we are** moving text around, searching for something to replace and so on. Also when we are manipulating data itself, very often data comes from multiple sources.

We might have tables of values which are typed in by somebody or generated by a device and we have to import them in a spreadsheet. And then if we want to manipulate them by using another program, we might **want to** export them from a spreadsheet this is typically done using text files in which the columns of the spreadsheet are stored in a systematic way separated by say commas. So, this also involves text processing.

And finally, most of us spend a **time using** a computer actually working with the internet. One of the most common things we do when we use the internet is to type queries and look for matching documents or other resources on the internet. So, most of this search query processing currently is done using text. It matches the text **in** the queries that we give with some information about the documents also implicitly in text and decides which documents are most relevant to our query. So, text processing is an important part of computation in general. And the ease in which you can manipulate text in python is one of the reasons why it has become a very popular language to program many things including internet applications.

(Refer Slide Time: 03:18)



Strings —type `str`

- Type string, `str`, a sequence of characters
 - A single character is a string of length 1
 - No separate type `char`
- Enclose in quotes—single, double, even triple!

```
city = 'Chennai'
```

```
title = "Hitchhiker's Guide to the Galaxy"
```

```
dialogue = '''He said his favourite book is  
Hitchhiker's Guide to the Galaxy'''
```

Python uses the type string for text, which internally is called `str`. So, we will use the word string instead of `str`, because it is easier to say. So, a string is basically sequence of characters. Unlike other programming languages, python does not have a specific character type to distinguish a single character from a string of length 1. So, there is only one type for text in python, which is string, and a single character is indistinguishable from a string of length 1. So, there are not two types of things; it is not that we have single characters and then string is a sequence of characters, a string is sequence of symbols and one symbol is just a sequence of length 1.

The values of this type are written as we would normally do in English using quotes. We use quotation marks to **demarcate** the beginning and at the end of a string when we want to write down an explicit value. So, we can use any type of quote, so a single quote would denote in this case the name city is assigned the string 'Chennai'. Note that when we write symbols like this capital C is different from small c and so on. So, we have seen exactly seen the symbols within **these** two quotes as the value assigned to the string to the name city.

Now we can also use double quotes; and one reason to use double quotes is if you actually need to use a single quote as part of the string. This is one way to do it; and the other way to do it is actually to write a back slash. If you write a back slash and a quote in the middle of the string, it means that this quote is to be taken as a symbol and not at the end of the string, but a much simpler way to include special things like quotes inside **other** quotes is to change the quotation. So, a single quote can include double quotes, and the double quote can include single quote without any confusion. So, this says that the name title is assigned a **value** "Hitchhiker's Guide to the Galaxy".

Now, what if you wanted to combine both double quotes and single quotes in a string? So, python has a very convenient thing called a triple quote. So, you can open three single quotes, and then you can write whatever you want with multiple double quotes and single quotes. So, if you want to say "“He said his favorite book is within quotes “Hitchhiker's Guide to the Galaxy” ”". Then this value string has both double quotes inside it and it also has a single quote inside it. So, we cannot enclose it in double quotes and we cannot enclose it in single quotes, because either **of them** will be ambiguous unless we use this back slash as I said before. So, if we do not want to use back slash, you can use a triple quote.

(Refer Slide Time: 06:06)

```
madhavan@dolphinair:~$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> s = 'Chennai'
>>> s
'Chennai'
>>> type(s)
<class 'str'>
>>> t = 'X'
>>> type(t)
<class 'str'>
>>> title = "Hitchhiker's"
>>> title
'Hitchhiker's'
>>> type(title)
<class 'str'>
>>> myquote = '''Hitchhiker's'''
>>> myquote
'Hitchhiker\'s'
>>> myquote = '''First line
... Second line
... Third line'''
>>> myquote
'First line
Second line
Third line'
>>>
```

Let see how this works in python interpreter. So, we can say s equal to ‘Chennai’ and now been asked the value of this and we see that it is reported with single quote. If we ask for the type of s, it says that s is class of str. So, this tells us that internally python realizes that s is a string. If we say t is equal to say just the letter x, then the type of t is also a string. So, there is no distinction between single character and multiple characters. Now if we say let us just shorten it say title is equal to “Hitchhiker’s” then if you ask for the value of title, it shows it to you with double quotes outside and a single quote outside. So, this indicates that **this** is a single string and again the type of title is str.

And finally, if I say **myquote** is equal to and I use three quotes and I use ““Hitchhiker’s” ””. So, I have “Hitchhiker’s” in double quotes and Hitchhiker’s itself contains a single quote. And I use triple quotes around it then my quote is correctly shown. Now notice that when it displays my quote, it does not show triple quotes. It includes puts another single quote outside and it shows this internal single quote has been highlighted with the back slash. So, back slash single quote is python's way and many programming languages’ way of saying that the next character should not be treated as what it stands for, but as it is. So, just take the next single quote as a single quote, do not treat as the end of the quotation.

The other thing that **you** can do with single quote is to actually write multiple lines. So, I do this first line, and then second line, and then third line, and then close the quote then my quote is shown as first line with back slash n. So, back slash again is a special character which indicates a new line; then second line, then new line, and then third line. We said before that python is very useful for manipulating text and one other thing that you would like to do is actually read and say a paragraph of text or multiple lines of a document and not have to worry about the fact that these are multiple lines just store it as a text value as a string. This is very much possible **in** python you can embed multiple lines of text into a single value.

(Refer Slide Time: 09:00)

The slide is titled "Strings as sequences" in blue text. It contains a bulleted list of points about string indexing in Python. The first point is "String: sequence or list of characters". The second point is "Positions 0,1,2,...,n-1 for a string of length n". The third point is "Positions -1,-2,... count backwards from end". To the right of the text, there is a diagram for the string "hello". The characters 'h', 'e', 'l', 'l', 'o' are each in a small box. Above the boxes are indices 0, 1, 2, 3, 4. Below the boxes are indices -5, -4, -3, -2, -1. A red arrow points to the index 0 above the 'h'.

- String: sequence or list of characters
- Positions 0,1,2,...,n-1 for a string of length n
- `s = "hello"`
- Positions -1,-2,... count backwards from end

As we said the string is a sequence or a list of characters. So, how do we get to individual characters in this list? Well, these characters have positions and in python positions in a string start with 0. So, if I have n characters in a string, the positions are named 0 to n minus 1. So, supposing we have a string hello, it has 5 characters. So, the positions in the string will be called 0, 1, 2, 3 and 4; so this is how we label positions.

And another convenience in python is that we can actually label it backwards. We can say that this is position minus 1; very often you want to say take the last character of a string and do something. So, instead of having to remember the length and then go to the

end, it is convenient to say take the last character. So, take the minus 1th character. So, we actually saw this and we did the gcd, we talked about the last element of the list say the list of common characters, and we said the minus 1th element n the list is the last element.

This numbering scheme that we use for list informally in the gcd example without formally explaining, it is actually the same numbering scheme that is used for positions in the string. We **have** minus 1, minus 2, minus 3, minus 4, minus 5, so the important thing to remember is that going forward, you start at 0, and coming backward you start at minus 1, **because** obviously, minus 0 is same as 0. So, if we use minus 0 for the right most thing there **would** be terrible confusion as to whether we are talking about the first value or the last value. So, the forward position start from 0 from the beginning and the reverse position start from minus 1 from the last element.

(Refer Slide Time: 10:37)

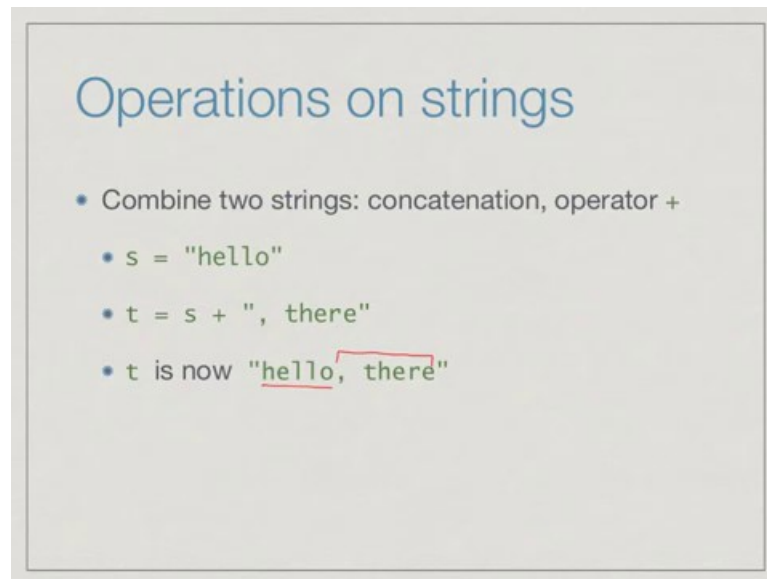
Strings as sequences

- String: sequence or list of characters
- Positions 0,1,2,...,n-1 for a string of length n
- `s = "hello"`

0	1	2	3	4
h	e	l	l	o
-5	-4	-3	-2	-1
- Positions -1,-2,... count backwards from end
- `s[1] == "e", s[-2] == "l"`

Once we have this then we can see that we use this square bracket notation to extract individual positions. So, `s[1]`, so that is the character at position 1 is an e and if I walk backwards then `s[-2]` is an l.

(Refer Slide Time: 10:59)



The slide is titled "Operations on strings" in a blue font. It contains a bulleted list with three items. The first item is "Combine two strings: concatenation, operator +". The second item is a code snippet `s = "hello"`. The third item is a code snippet `t = s + ", there"`. Below the third item, it says "t is now" followed by the string `"hello, there"`. In the original image, the word "hello" in the final string is underlined with a red line, and a red bracket connects the plus sign in the code to the word "hello".

Operations on strings

- Combine two strings: concatenation, operator +
 - `s = "hello"`
 - `t = s + ", there"`
 - t is now `"hello, there"`

One of the most basic things one can do with strings is to put them together; to combine two values into a larger string and this is called concatenation; putting them one after the other. And the operator that is used for this is plus. So, plus, we saw for numeric values **add** them; for strings the same symbol plus does not **add** strings; obviously, **it** does not make **sense** to **add** strings, but it **juxtaposes** them, puts them one after the other.

So, if we have a string hello as we did before, and we take this, and we take a new string and we add it to s. Then we get a string t, whose value is the part that was **in** hello plus the part that was added. So, plus is just the simple operator which takes two strings and sticks them side by side.

(Refer Slide Time: 11:52)

```
>>> s = "hello"
>>> t = "there"
>>> s+t
'hellothere'
>>> t = " there"
>>> s+t
'hello there'
>>>
```

Let us look at an example in the interpreter. Just to emphasize one point; supposing I said s was hello and t was there, then s plus t would be the value hello there. Now notice that there is no space. So, plus literally puts s followed by t, it does not introduce punctuation, any separation, any space and this is as you would like it. If you want to put a comma or a space you must do that, so if you say t instead of that was space there t is the string consisting of blank space followed by there, now if I say s plus t, I get a space between hello and there.

This is important to note that plus directly puts things together it does not add any punctuation or any separation between the two values. So, it is as though you have one new string which is composed of many old strings whose boundaries disappear completely.

(Refer Slide Time: 12:47)

Operations on strings


- Combine two strings: concatenation, operator +
 - `s = "hello"`
 - `t = s + ", there"`
 - `t` is now `"hello, there"`
- `len(s)` returns length of `s`
- Will see other functions to manipulate strings later

We can get length of the string using the function `len`. So, `len(s)` returns the length of `s`. So, this is the number of characters. So, remember that if the number of characters is `n` then the positions are 0 to `n` minus 1. So, the length of the string `s` here would be 5, the length of the string `t` here would be 5 plus 7 – 12. There are many other interesting functions that one can use to manipulate strings, you can search and replace things, you can find the first occurrence of something and so on, and we will see some of these later on, **when** we get into strings and text processing and reading data from files in more details.

(Refer Slide Time: 13:26)

Extracting substrings

A **slice** is a “segment” of a string




The diagram shows a horizontal rectangle representing a string. Two vertical red lines are drawn inside the rectangle, and a double-headed red arrow is positioned above them, indicating the segment between the two lines.


A very common thing that we want to do with strings is to extract the part of a string. We might want to extract the beginning, the first word and things like that. The **most simple** way to do this in python is to take what is called a slice. Slice is a segment, a segment means I take a long string which I can think of as a list of character and I want the portion from some starting point to some ending point.

(Refer Slide Time: 13:55)

Extracting substrings

A **slice** is a “segment” of a string

- `s = "hello"`


The string "hello" is shown with indices 0, 1, 2, 3, 4 written above each character. A red bracket is drawn under the characters 'e', 'l', and 'l'.
- `s[1:4]` is "ell"


The characters '1' and '4' in the slice notation are underlined with red lines.

`range(1, m+1)`

This is what python calls a slice. So, if we say s is hello as before, then for a slice we give this starting point and the ending point separated by colon. So, we use this square bracket notation exactly as though we were extracting part of a string, but the part that we are extracting is not the single position, but a range of positions from 1 to 4.

Now in python, we saw that we had this range function which we wrote last time, it said things like, if I want the numbers from 1 to m, I must write 1 to m plus 1, because the range function in python stops one position short of the last element of the range. So, in the same way, a slice stops one position short of the last index in the slice. So, if I do this then remember that hello has position 0, 1, 2, 3, 4, so the slice from 1 to 4 starts at 1 goes to 2, goes to 3, but does not go to 4, so it is only from e to l - the second l.

(Refer Slide Time: 14:59)

Extracting substrings

A **slice** is a "segment" of a string *range(1, n+1)*

- `s = "hello"`
- `s[1:4]` is "ell"
- `s[i:j]` starts at `s[i]` and ends at `s[j-1]`
- `s[:j]` starts at `s[0]`, so `s[0:j]`
- `s[i:]` ends at `s[len(s)-1]`, so `s[i:len(s)]`

In general, if I write s i colon j then it starts at s i and ends at s j minus 1. There are some shortcuts which are easy to remember and use; very often you want to take the first n characters in the string, then you could omit the 0, and just say start implicitly from 0, so just leave it out, so just start say colon and j. So this will give us all position 0 1 up to j minus 1. So, if I leave out first position, it is implicitly starting from 0.

Similarly, if I leave out the last position it runs **to** the end of the string. So, if I want

everything from i onwards then I can say s i colon and this will go up to the position length of s minus 1, but if I write explicitly as a slice, I will only write length of s. So, essentially **this is the** main reason that python has this convention that whenever I write something like a range of 1 to m plus 1 then I have this extra plus 1 here. So, the main reason for this plus 1 here is to avoid having to write minus 1.

If I had to include the last character and if I start numbering at 0, then every time I wanted to go to the end of the string I would have to say length of s minus 1. It is much more convenient to just say length of s, and implicitly assume that it knows that it should not go to length of s, but **length of s** minus 1. So, this whole confusion **if you** would like to call **that** in python about that fact that all **ranges** end one short of the right hand side of the range, **stems from** the fact that **you very often** want to run from something to the length of it in a list or a sequence or a string and when you say that you do not want have keep remembering to say minus 1.

(Refer Slide Time: 16:45)

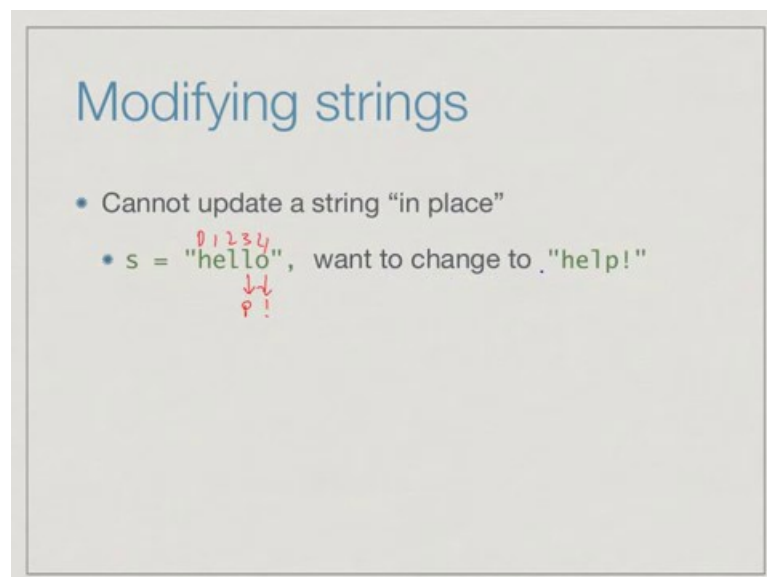
```
>>> s = "hello"
>>> s[1:4]
'ell'
>>> s[:3]
'hel'
>>> s[2:]
'llo'
>>> s[3:1]
''
>>> s[0:7]
'hello'
>>> []
```

Let us play with the second in the python interpreter. So, if I say s is equal to hello then we saw that if I do 1 to 4, I get 'ell'. If I say colon 3 then I get 'hel' that is 0 1 2. If I say 2 colon, I get 'llo' that is 2 3 4. What if I say 3 2 1, so **this says: start at** position 3 and go up to position 1 minus 1 which is 0. So, python does not give you an error, it takes all these

invalid ranges, anything where for example, the starting point to the ending point does not define a valid range, and it says this is the empty string.

On the other hand, if I say something like go from 0 to 7, where there is no 7th position in the string, here python will not give an error instead, it will just go up to the last position which actually exists in the string below 7. So, in general these range values are treated in a sensible way, if you give values which do not make sense. As far as possible python tries to do something sensible with the slice definition.

(Refer Slide Time: 17:57)



Modifying strings

- Cannot update a string “in place”
- `s = "hello"`, want to change to `"help!"`

Annotations for the code example:

- Indices 0, 1, 2, 3, 4 are written above the characters 'h', 'e', 'l', 'l', 'o' respectively.
- Red arrows point from index 2 to 'p' and from index 3 to '!'.
- A red 'p' and a red '!' are written below the arrows.

Though we have access to individual positions or individual slices or sequences within a string, we cannot take a part of a string and change **it as it** stands. So, we cannot update a string in place. Suppose, we want to take our string “hello” and change it to the string “help!” it would be nice if we could take the third and the fourth position. So, remember 0, 1, 2, 3, 4, 5, so 0, 1, 2, 3, 4, so it would be nice if we could say make this into a p and make this into an exclamation mark, so that I could get help instead of hello.

(Refer Slide Time: 18:36)

Modifying strings

- Cannot update a string “in place”
 - `s = "hello"`, want to change to `"help!"`
 - `s[3] = "p"` — error!

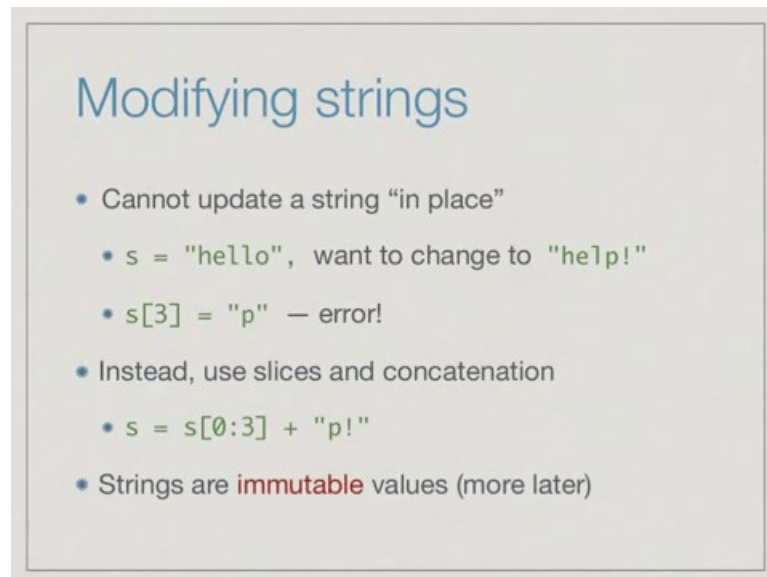
We would want to write something like change s 3, assign the value s 3 to be the string p. Now, unfortunately python does not allow this. So, you cannot update a string in place by changing its part. In fact, if you try this, you will actually get an error message, let us see.

(Refer Slide Time: 18:54)

```
>>> s = "hello"
>>> s[1:4]
'ell'
>>> s[:3]
'hel'
>>> s[2:]
'llo'
>>> s[3:1]
''
>>> s[0:7]
'hello'
>>> s[3] = 'p'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> 
```

Here we have the string hello defined in four, and if I now try to say s[3] is equal to p, then it says this does not support item assignment, which is what we are trying to say you cannot change parts of a string as it stands.

(Refer Slide Time: 19:12)



Modifying strings

- Cannot update a string “in place”
 - `s = "hello"`, want to change to `"he!p!"`
 - `s[3] = "p"` — error!
- Instead, use slices and concatenation
 - `s = s[0:3] + "p!"`
- Strings are **immutable** values (more later)

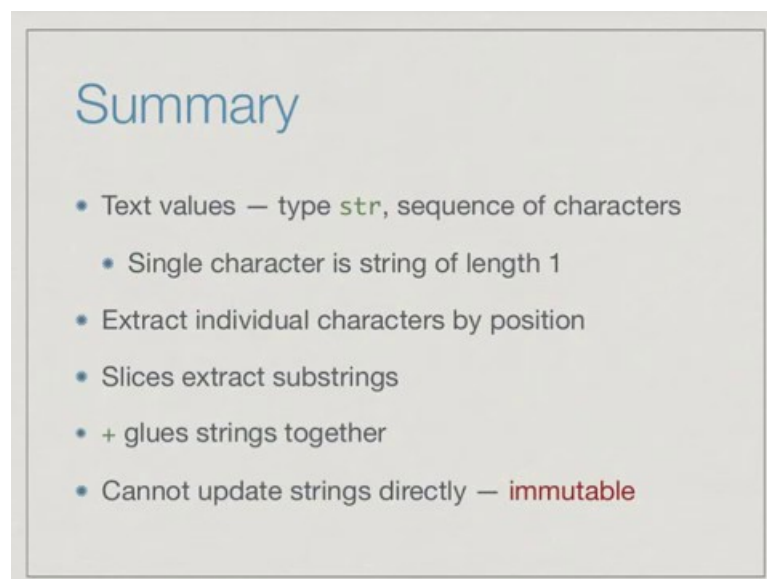
Instead of doing this, instead of trying to take a string and change the part of it as it stands what you need to do is actually construct a new string effectively using the notion of slices and concatenation. Here what we want to do is we want to take the first part of the string as it is. These are the first three characters, and then we want to change this to p exclamation mark. So, what we can say is update s by taking 0, 1, 2 which is slice 0 to 3 and concatenating it with the new string p exclamation mark. So, this is how you modify strings in python, but important thing is this is a new s we are not claiming that this s is same as old s.

There we build a new string from the old string and perhaps **store it** back in the same name, **it is partly** like when we say j is equal to j plus 5, we are actually saying that we have created a new value for j and stored it back in j.

Here again we are creating a new string and putting it back, but we are not modifying it. Now this distinction between modifying **and creating** a new value may not seem very

important at this moment, but it will become important as we go along. So, strings are what are called immutable values, you cannot change them without actually creating a fresh value; whereas, lists as we will see which are more general type of sequence can be changed in place you can take one part of a list and then replace it by something else. So, we will see more about this later, this is a fairly important concept. Remember for now that strings cannot be changed in place.

(Refer Slide Time: 20:42)



The slide is titled "Summary" in a blue font. It contains a bulleted list of text values in Python. The word "str" is highlighted in green, and the word "immutable" is highlighted in red.

- Text values — type `str`, sequence of characters
 - Single character is string of length 1
- Extract individual characters by position
- Slices extract substrings
- `+` glues strings together
- Cannot update strings directly — `immutable`

To summarize what we have seen is that text values are important for computation, and python has the types - string or str, which is a sequence of characters to denote text values. And there is no distinction for a separate type for a single character; there is no single character type in python, a single character is just a string of a length 1.

We can extract individual characters by index positions, we can use slices to extract sub strings, and we can glue strings together using the concatenation operator plus, but strings are immutable. We cannot take a value assigned to a string name and update it in place. We can create a new value by manipulating it using slices and concatenation, but we cannot directly update it, because strings are immutable.