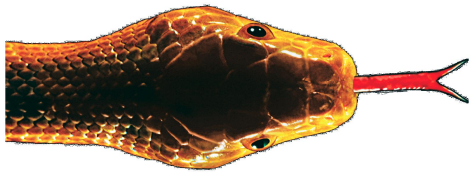
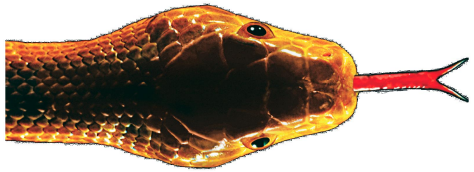


Python's **str** class

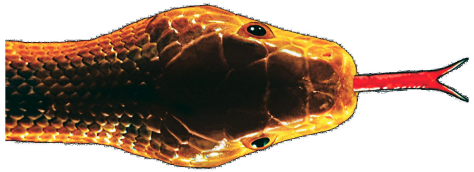
- A list can represent any sequence of objects
- A very common need in computing is for a sequence of text characters.
- There is a specialized class, named **str**, devoted to manipulating character strings.



String literals

- Can enclose in single quotes: `'bread'`
- Can enclose in double quotes: `"bread"`
- This choice helps when you want to use a single or double quote as a character within the string: `"Who's there?"`
- Can embed a newline character using an escape character `\n` as in:

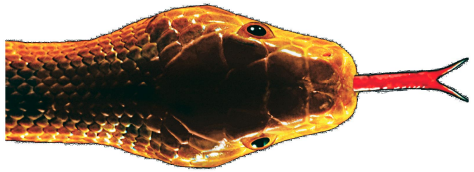
```
"Knock Knock\nWho's there?"
```



Common behaviors

greeting = 'How do you do?'

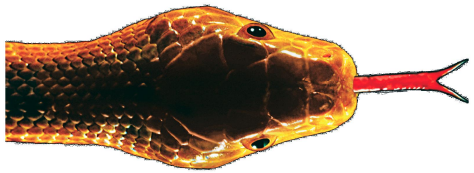
- len(greeting) returns 14
- `'yo'` in greeting returns **True**
- greeting.count(`'do'`) returns 2
- greeting.index(`'do'`) returns 4
- greeting[2] returns `'w'`



Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                01234567890123456789012345
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

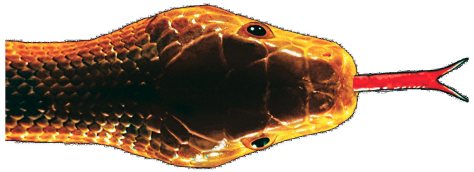


Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                0123456789012345678901234 5
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[4] returns `'e'`



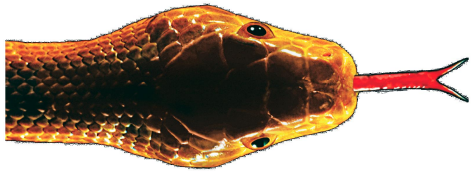
Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111112222222
                0123456789012345678901234567
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[4:13] returns `'efghijklm'`

(starting at 4, going up to but not including 13)



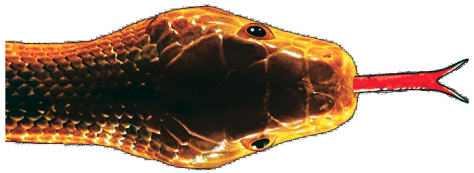
Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                012345678901234567890123456
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[:6] returns **'abcdef'**

(starting at beginning going up to but not including 6)



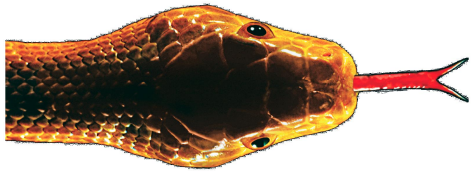
Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                0123456789012345678901234 5
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[23:] returns `'xyz'`

(starting at 23 going all the way to the end)



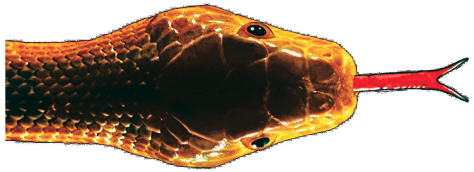
Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                01234567890123456789012345
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[9:20:3] returns `'jmps'`

(starting at 9, stopping before 20, stepping by 3)



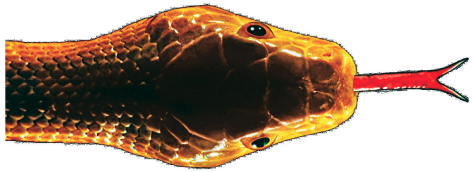
Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                01234567890123456789012345
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[17:5:-3] returns `'roli'`

(starting at 17, toward but not with 5, stepping by -3)

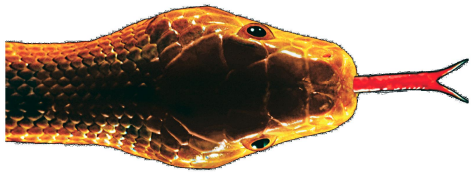


Slicing

Slicing is a generalization of indexing that is supported by strings (and lists too).

```
                          1111111111222222
                0123456789012345678901234 5
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

alphabet[: :-1] **'zyxwvutsrqponmlkjihgfedcba'**
(everything, but in reverse order)



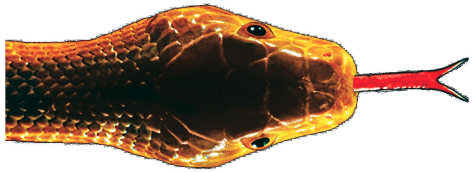
Summary of Slicing

Notice that convention for slicing

alphabet[start:stop:step]

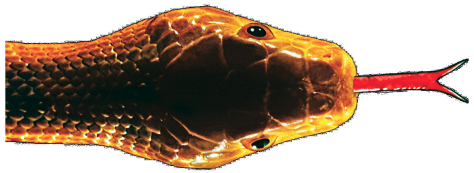
uses indices akin to that of

range(start, stop, step)



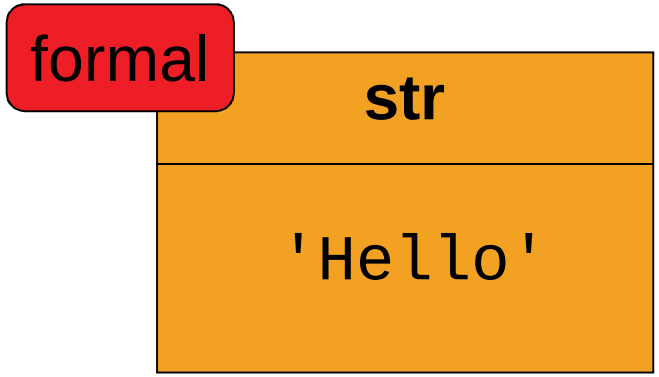
Differences: **list** and **str**

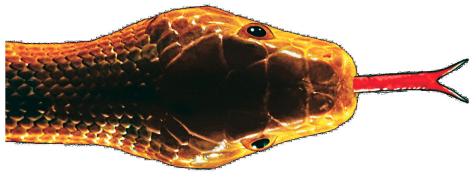
- List are ***mutable***; strings are ***immutable*** (allows Python to optimize the internals)
- We cannot change an existing string.
- However, we can create new strings based upon existing ones.



Example: lower( )

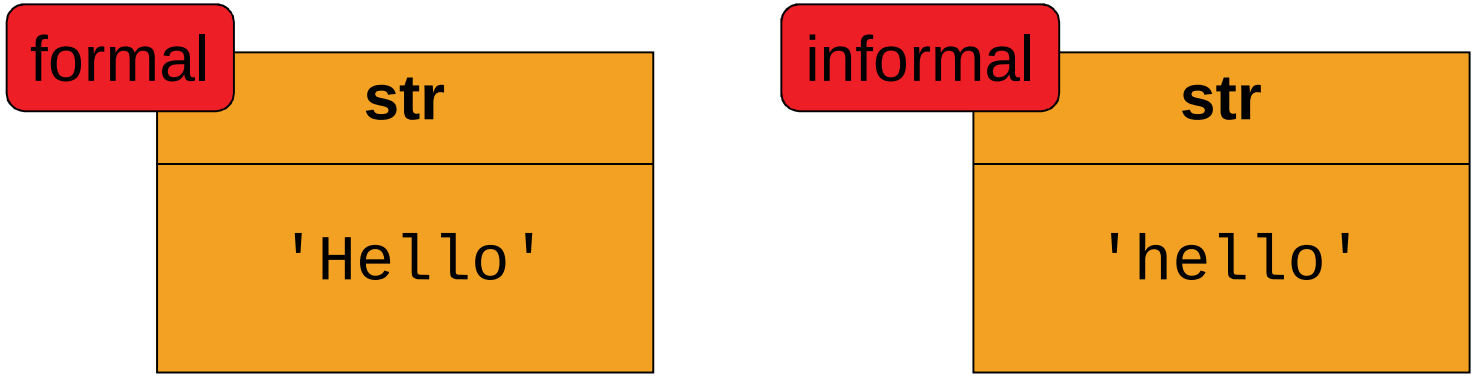
```
>>> formal = 'Hello'
>>>
```



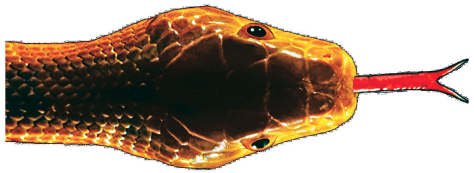


Example: lower( )

```
>>> formal = 'Hello'
>>> informal = formal.lower()
>>>
```

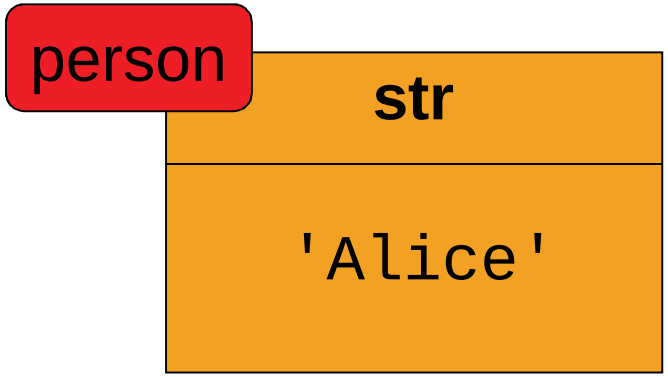


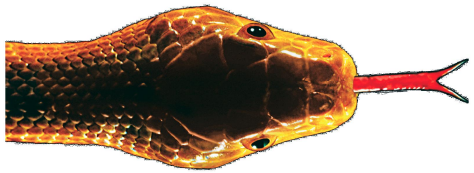
Note that formal is unchanged



Reassigning an Identifier

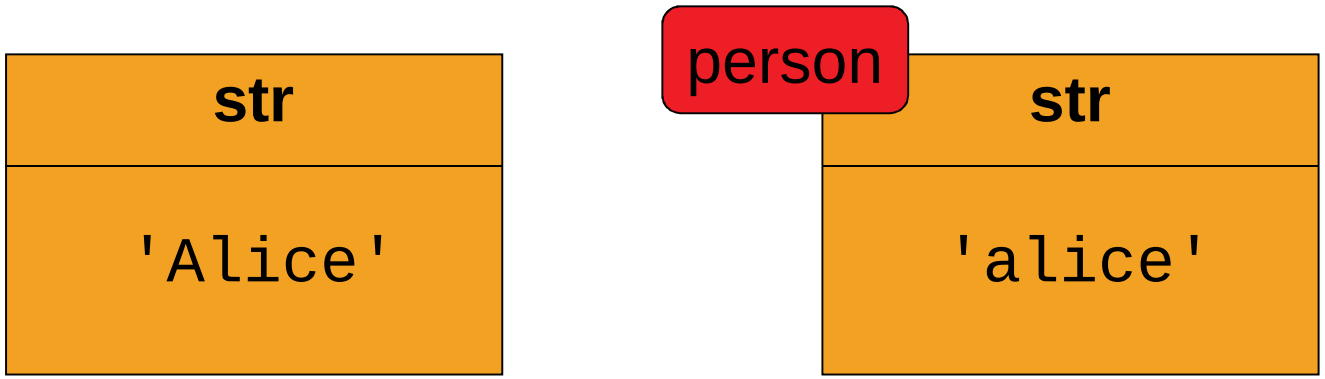
```
>>> person = 'Alice'
>>>
```

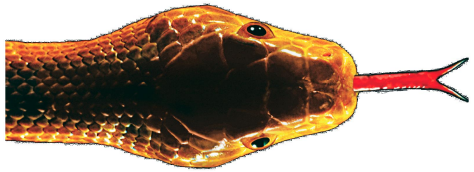




Reassigning an Identifier

```
>>> person = 'Alice'
>>> person = person.lower()
>>>
```

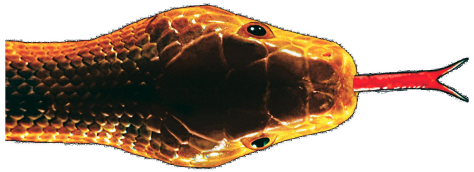




Creating New Strings

Each of the following leaves the original string unchanged, returning a new string as a result.

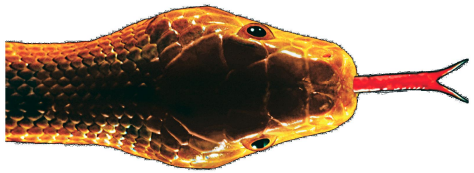
- greeting.lower( )
- greeting.upper( )
- greeting.capitalize( )
- greeting.strip( )
- greeting.center(30)
- greeting.replace('hi','hello')



Additional string methods

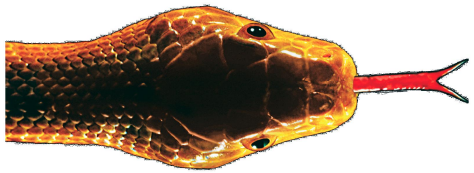
Strings support other methods that are specific to the context of textual information

- greeting.islower( ) not to be confused with lower( )
- greeting.isupper( )
- greeting.isalpha( )
- greeting.isdigit( )
- greeting.startswith(pattern)
- greeting.endswith(pattern)



Converting between strings and lists

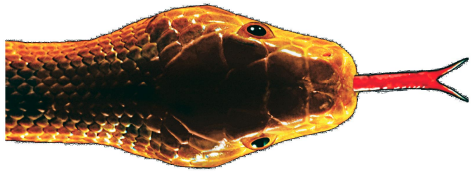
- To support text processing, the **str** class has methods to split and rejoin strings.
- split is used to divide a string into a list of pieces based upon a given separator.
- join is used to assemble a list of strings and a separator into a composite string.



The split method

By default, the pieces are based on dividing the original around any form of whitespace (e.g., spaces, tabs, newlines)

```
>>> request = 'eggs and milk and apples'
>>> request.split( )
['eggs', 'and', 'milk', 'and', 'apples']
```

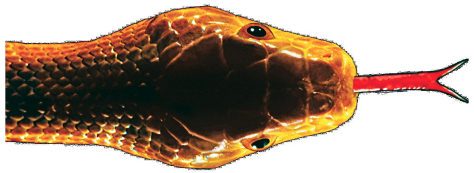


The split method

Some other separator can be specified as an optional parameter to split. That string will be used verbatim.

```
>>> request = 'eggs and milk and apples'
>>> request.split('and')
['eggs ', ' milk ', ' apples']
```

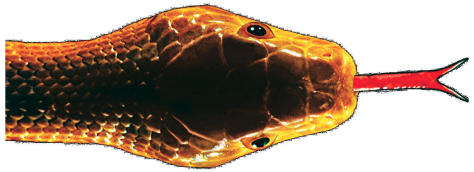
(note well the spaces that remain)



The split method

Here is the same example, but with spaces embedded within the separator string.

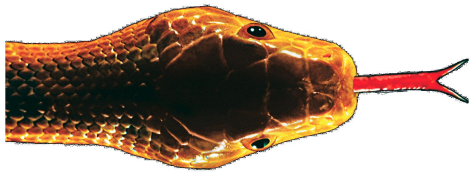
```
>>> request = 'eggs and milk and apples'
>>> request.split(' and ')
['eggs', 'milk', 'apples']
```



The join method

The join method takes a sequence of strings and combines them using a given string separator between each pair. Formally, this method is invoked upon the separator.

```
>>> guests = ['John', 'Mary', 'Amy']
>>> conjunction = ' and '
>>> conjunction.join(guests)
'John and Mary and Amy'
```



The join method

The separator is often expressed as a literal.

```
>>> guests = ['John', 'Mary', 'Amy']
>>> ' and '.join(guests)
'John and Mary and Amy'
```

The sequence could be a string of characters.

```
>>> '-'.join('respect')
'r-e-s-p-e-c-t'
```