Strings are sequences

- A string is a sequence of characters

- We can access the characters with the “square brackets”

```python
fruit = 'banana'
letter = fruit[0]
```
Strings are sequences

• The expression in the brackets is the “index”

• In Python, the first index is zero

• So the expression 'fruit[0]' is the character 'b'
The 'len' function

- 'len' is a built in function that returns the number of elements in a sequence

```python
>>> len(fruit)
6
```

- The last character is located at len(fruit) - 1
The 'len' function

• To print out the last character we type:

```python
>>> print(fruit[len(fruit) - 1])
```

• Or we can use a negative index:

```python
>>> print(fruit[-1])
```
Traversing a 'for' loop:

- A lot of computations involve processing a string one character at a time.

- With a 'while' loop:

```python
i = 0
while i < len(fruit):
    print(fruit[i])
    i += 1
```
Traversal with a 'for' loop

- Much simpler to use a 'for' loop:

```python
for ch in fruit:
    print(ch)
```

- Each time through the loop the variable 'ch' is assigned the next character in the sequence
String slices

● A string segment is called a slice

```python
>>> s = 'abcdefghijklmnopqrstuvwxyz'
>>> s[5:10]
fgij
```

● A slice begins at the 'start' index and ends before the 'last' index
String slices

• If you omit the 'start' index, the slice starts at the beginning of the string

• If you omit the 'last' index, the slice goes to the end of the string

• If the 'start' index is greater that the 'last' index the result is an empty string
String slices

- A slice can take an optional third argument that specifies a “step size”

- For example, a “step size” of 2 means every other character

```python
>>> s[:2]
'acegikmoqsuwy'
```
Strings are immutable

- You cannot change the value of a string
- Attempting to assign a new character to a string produces an error

```python
>>> greeting = 'Hello, world!

>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```
Strings are immutable

- The best we can do is create a new string that is a variation of the original

```python
>>> greeting1 = 'Hello, world!
>>> greeting2 = 'J' + greeting1[1:]
```

- The original string does not change
String methods

- A method is similar to a function, but the syntax is different

- For example:

  ```python
  >>> word = 'banana'
  >>> word.upper()
  >>> print(word)
  'BANANA'
  ```
The 'find' method

- Usage: S.find(sub)
- Returns the lowest index in S where the substring 'sub' is found
- Returns '-1' on failure
The 'count' method

- Usage: S.count(sub)

- Returns the number of occurrences of substring 'sub' in the string
More string methods

- All of the string methods can be found by typing:

  >>> help(str)

- At the Python shell
The 'in' operator

- The 'in' operator is a boolean operator

- It takes two strings and returns 'True' if the first string is appears as a substring in the second

```python
>>> 'ana' in 'banana'
True
```
String comparison

- All of the relational operators work on strings

- However, strings are ordered lexicographically

```python
>>> 'Pineapple' < 'banana'
True
```
String comparison

- A simple way to overcome this problem is to convert all string to upper (or lower) case

```python
>>> s = 'Pineapple'
>>> t = 'banana'
>>> s.upper() < t.upper()
False
```
Some tricks

• An easy way to reverse a string:

```python
>>> s = 'hello, world!

>>> s[::-1]

'!dlrow ,olleh'
```
More tricks

• Use 'in' to validate DNA/RNA:

```python
def validate_dna(base):
    return base in 'ACGTacgt'
```