Strings

A Python ***string*** is an *ordered*, *immutable* sequence of characters used to represent and store text-based information.
Unlike C, Python provides a powerful set of tools for manipulating strings

```
>>> s1 = 'mercury'
>>> s2 = "venus"
>>> s3 = '9' * 3
>>> s3
'999'
>>> s1 == s2
False
>>> s2 == 'venus':
True
>>> s4 = s1 + ' and ' + s2
>>> s4
'mercury and venus'
```

Either single (') or double (") quotes may be used in assigning a string

string comparisons

string concatenation

Strings

```
>>> s4
'mercury and venus'
>>> s4.replace('ercury', 'ars')
'mars and venus'
>>> s4
'mercury and venus'

>>> s4[3]
'c'

>>> home = 'earth'
>>> len(home)
5
>>> home[1:4]
'art'
```

`s4` hasn't been changed: the `replace()` method returned a new (modified) string

string *indexing*

note that Python indexes start at 0

string *slicing*: `[n:m]` gets characters `n, n+1, ... m-1`

String Methods

```
>>> s4
'mercury and venus'
>>> s4.capitalize()
'Mars and venus'
>>> s4.title()
'Mercury And Venus'
>>> s4.startswith('merc')
True
>>> s4.endswith('us')
True
>>> 'cury an' in s4
True
>>> s4.swapcase()
'MERCURY AND VENUS'
```

the keyword `in` compares for 'membership': here, for a substring of a string

And there's more: http://docs.python.org/library/stdtypes.html#string-methods

String Escaping

Certain special characters are introduced with a backslash:

`\t`	tab
`\n`	newline (linefeed)
`\b`	backspace
`\'`	single quote
`\"`	double quote
`\\`	the backslash character itself

String Escaping

```
>>> label1 = "Christian's string"
>>> label2 = 'Christian\'s string'

>>> heading = 'one\ttwo\tthree\tfour'
>>> heading
'one\ttwo\tthree\tfour'
>>> print(heading)
one     two     three   four

>>> shopping_list = 'apples\nbananas\nbread\nmilk'
>>> print(shopping_list)
apples
bananas
bread
milk
```

the print command evaluates the special characters

Lists

A Python ***list*** is an *ordered*, *mutable* sequence of objects
Lists can contain a mixture of any sort of object: numbers, boolean values, strings ... and even other lists
Lists can grow or shrink *in place*

```
>>> list1 = [4, 5, 6]
>>> len(list1)
3
>>> list1[0]
4
>>> list1[2]
6
>>> list[-1]
6
>>> list[-2]
5
```

note that Python indexes start at 0

negative indices count backwards from the end of the list

List Methods

```
>>> list1
[4, 5, 6]
```

```
>>> list2 = ['foo']*3
>>> list2
['foo', 'foo', 'foo']
```

repetition

```
>>> list1.extend([2, -1, 0])
>>> list1
[4, 5, 6, 2, -1, 0]
```

concatenation

```
>>> list1.append(5)
>>> list1
[4, 5, 6, 2, -1, 0, 5]
```

grow a list

```
>>> 7 in list1
False
```

membership

List Methods

```
>>> list1
[4, 5, 6, 2, -1, 0, 5]
```

```
>>> list1[2] = 1
>>> list1
[4, 5, 1, 2, -1, 0, 5]
```

we can change lists in place

```
>>> list1.index(5)
1
```

returns the index of the *first* occurrence of 5

```
>>> list1.index(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list
```

oops

List Methods

```
>>> list1
[4, 5, 1, 2, -1, 0, 5]
```

```
>>> list1[1:5]
[5, 1, 2, -1]
```

list slicing

```
>>> list1[1:5:2]
[5, 2]
```

list striding

```
>>> del list1[5]
>>> list1
[4, 5, 1, 2, -1, 5]
```

shrink list by removing the item at index 5

```
>>> list1.sort()
>>> list1
[-1, 1, 2, 4, 5, 5]
```

list sorting (in place)

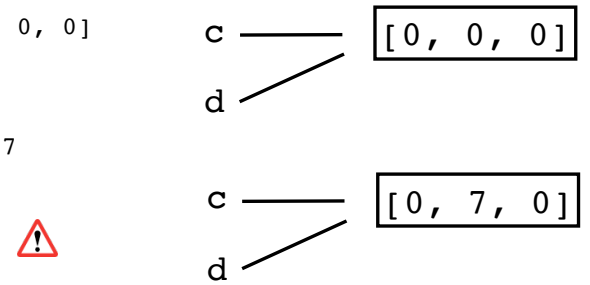
```
>>> list1.reverse()
>>> list1
[5, 5, 4, 2, 1, -1]
```

list reverse (also in place)

Lists – beware

A variable, c, assigned to a list references that list object
Another variable, d, set equal to c references *the same object*
So if you change d, you change c!

```
>>> c = [0, 0, 0]
>>> d = c
>>> d
[0, 0, 0]
>>> d[1] = 7
>>> d
[0, 7, 0]
>>> c
[0, 7, 0]
```

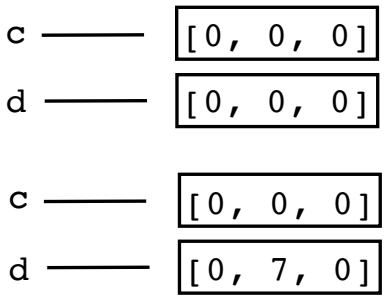


Making a copy of a list

Use the list "constructor": `d = list(c)`
or slice the whole list: `c[:] == c[0:]`
d is then an entirely independent object

```
>>> c = [0, 0, 0]
>>> d = list(c)
```

```
>>> d[1] = 7
>>> d
[0, 7, 0]
>>> c
[0, 0, 0]
```



Tuples

A Python ***tuple*** is an *ordered*, *immutable* sequence of objects
They work like lists but cannot be altered ...
... and don't have methods such as `sort()`, `reverse()`, etc.

```
>>> t1 = ('X', 'Y', 'Z')
>>> t1[2]
'Z'
```

tuples are indicated by round brackets

```
>>> t2 = (1.4, 'hello', [0, 0, 0])
>>> t2[1:]
('hello', [0, 0, 0])
```

indexing and slicing work as for lists

```
>>> t3 = t1 + t2
>>> t4 = t3 * 7
```

tuple concatenation and repetition also work

Tuples

```
>>> t2
(1.4, 'hello', [0, 0, 0])
>>> 0 in t2
False
>>> [0, 0, 0] in t2
True
```

note that the membership test is *by value*, not *by reference*

```
>>> t2[1] = 'goodbye'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

oops

```
>>> t3 = tuple([0, 0, 0])
>>> t3
(0, 0, 0)
```

you can make a tuple from a list

for Loops

To loop over an iterable object (such as a string, list or tuple):

fruitlist.py

```
fruit = ['apple', 'banana', 'cherry', 'durian']
for f in fruit:
    print('Here\'s a fruit:', f)
```

statements to be executed for each iteration of the loop are inside an *indented block* (use 4 spaces)

```
$ python fruitlist.py
Here's a fruit: apple
Here's a fruit: banana
Here's a fruit: cherry
Here's a fruit: durian
```

for Loops

Another example: calculate the first 10 square numbers

squares.py

```
for i in range(10):
    print(i+1, 'squared is', (i+1)**2)
```

`range(n)` generates the n-item sequence:
`0, 1, 2, ..., n-1`

```
$ python squares.py
1 squared is 1
2 squared is 4
...
...
9 squared is 81
10 squared is 100
```

while Loops

squares2.py

```
i = 1
while i <= 10:
    print(i, 'squared is', i**2)
    i += 1
```

```
$ python squares2.py
1 squared is 1
2 squared is 4
...
...
9 squared is 81
10 squared is 100
```

Conditionals: the `if` keyword

Basic syntax:

```
if <test1>:
    <statements-1>
elif <test2>:
    <statements-2>
...
else:
    <statements-n>
```

The statements in the indented block `<statements-1>` are executed if `<test1>` is `True`.
If it is `False`, `<test2>` is evaluated; if `<test2>` is `True`, then `<statements-2>` are executed, and so on.
If none of the tests is `True`, the statements in the (optional) `else` block are executed.

Conditionals: the `if` keyword

Example:

divisible.py

```
for i in range(1,17):
    if (i % 5) == 0:
        print(i, 'is divisible by 5')
    elif (i % 7) == 0:
        print(i, 'is divisible by 7')
```

`range(1,17)` generates: 1,2, ..., 16

```
$ python divisible.py
5 is divisible by 5
7 is divisible by 7
10 is divisible by 5
14 is divisible by 7
15 is divisible by 5
```

More flow control

More keywords:

`pass`	do nothing (placeholder statement)
`continue`	go back to the top of the loop and resume without completing the execution of the loop block on this iteration
`break`	break out of the loop immediately, without completing its iterations
`else`	a block executed after a `while` loop, only if it completed without encountering a `break`

More flow control

Example

elements.py

```
symbols = ['H', 'He', 'Li', 'Be', 'B', 'C', 'N',
'O', 'F', 'Ne', 'Na', 'Mg', 'Al', 'Si', 'P', 'S']
for symbol in symbols:
    if symbol.startswith('B'):
        continue  # ignore symbols starting with B
    if symbol == 'Mg':
        break     # stop the loop if we see Mg
    print(symbol, end='')
```

the `end=''` here suppresses the newline

```
$ python elements.py
H He Li C N O F Ne Na
```

Program comments

Everything after a '#' sign (except inside string literals) is ignored by Python - and so is a 'comment'
For multiline comments, start each line with a '#'
Comment your code - the person you help is likely you
Keep comments up-to-date: comments that are inconsistent with the code are worse than no comments at all
Don't over-comment your code!

Bad

```
i = i + 1      # Increment i
```

Good

```
x = x + 10.    # Compensate for border
```