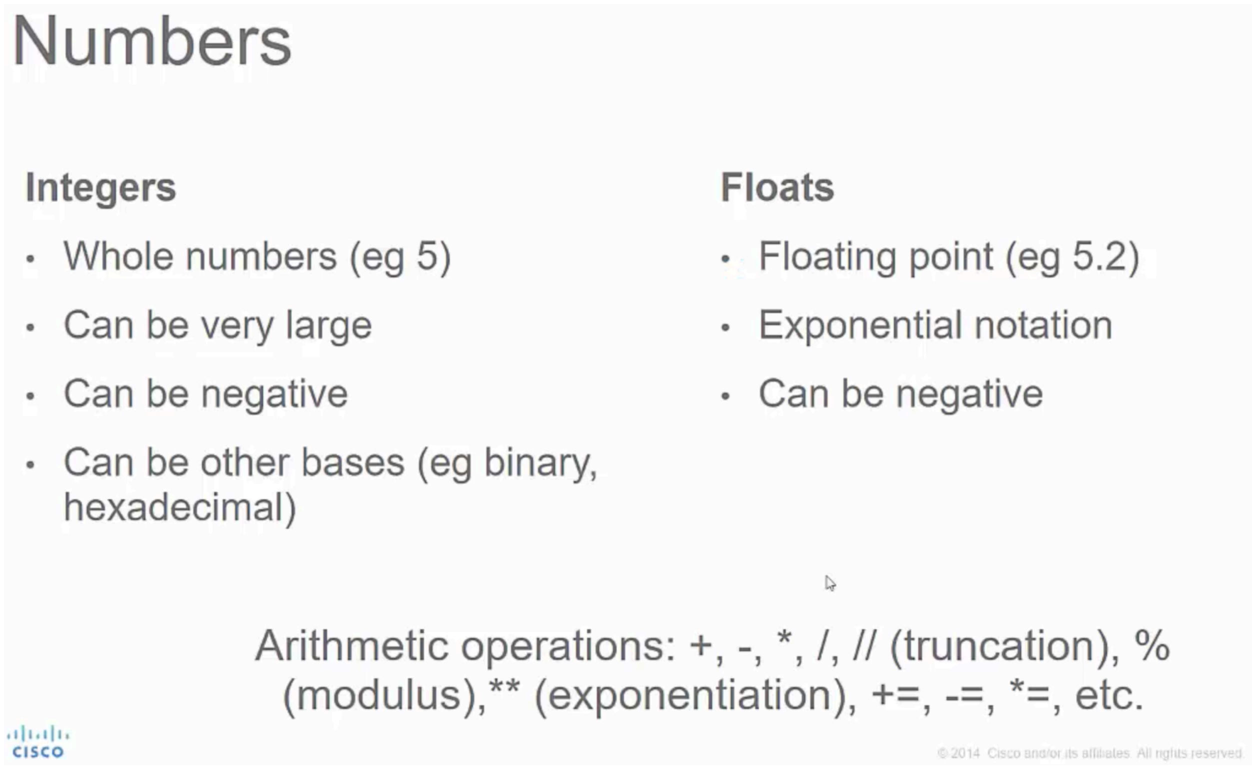
In programming, literals are used to represent constant values. Different types of literals are used to represent different types of values. Variables are names that reference values that may change over time. The three primary classes of literals in Python are boolean literals, numbers, and strings. Boolean literals can only have the values true or false.



Numbers

Integers

- Whole numbers (eg 5)
- Can be very large
- Can be negative
- Can be other bases (eg binary, hexadecimal)

Floats

- Floating point (eg 5.2)
- Exponential notation
- Can be negative

Arithmetic operations: +, -, *, /, // (truncation), % (modulus),** (exponentiation), +=, -=, *=, etc.

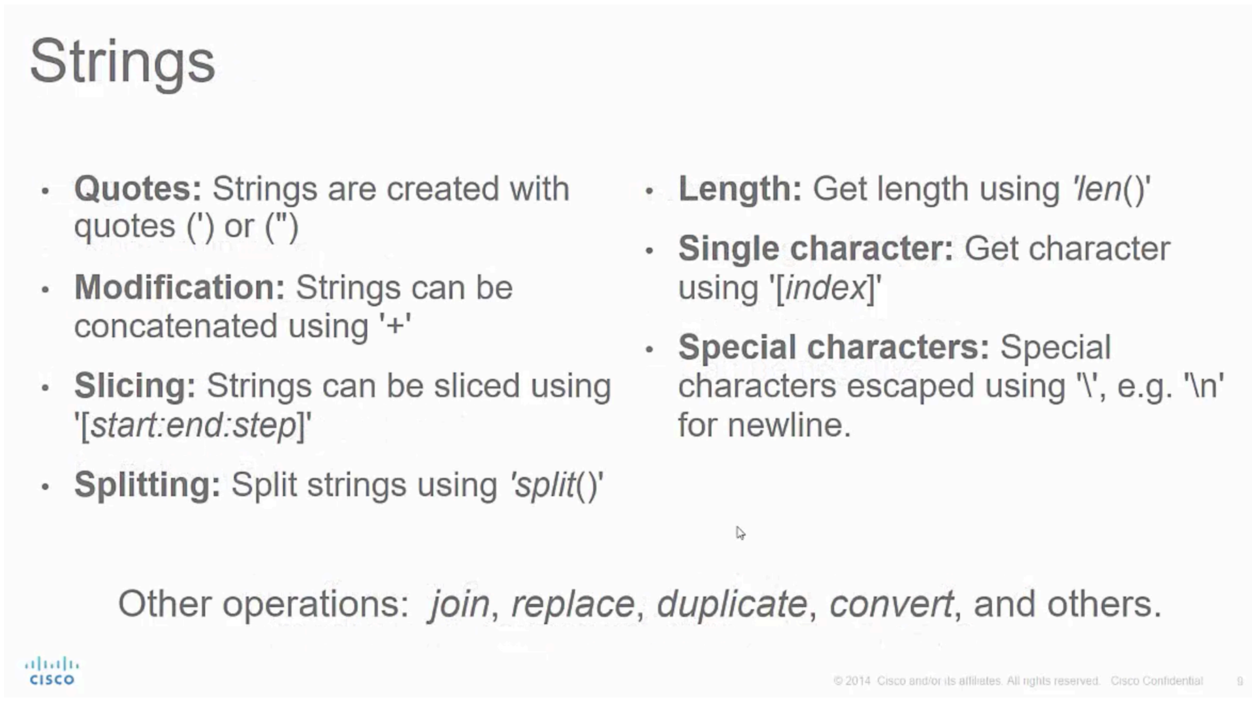
Numbers

In general there are two types of numbers with Python:

Integers. Integers are whole numbers, which can be very large, and which can also be negative. You can also use other numerical bases with integers, such as binary, octal, and hexadecimal.The following examples show 27 represented in other bases.

Binary: 0b00011011
Octal: 0o33
Hexadecimal: 0x1B

Floats. Python provides the floating-point data type for precise numerical values such as 1.23. Floats can also use exponential notation. For example, 1.0e6 which is 1,000,000.


Strings

- **Quotes:** Strings are created with quotes (') or (")
- **Modification:** Strings can be concatenated using '+'
- **Slicing:** Strings can be sliced using '[*start:end:step*]'
- **Splitting:** Split strings using '*split*()'
- **Length:** Get length using '*len*()'
- **Single character:** Get character using '[*index*]'
- **Special characters:** Special characters escaped using '\', e.g. '\n' for newline.

Other operations: *join*, *replace*, *duplicate*, *convert*, and others.

© 2014 Cisco and/or its affiliates. All rights reserved. Cisco Confidential 9

Strings

Strings are a series of characters. The characters in a string can be alphanumeric including many special characters such as punctuation. A few important points about strings:

- **Quotes**. Strings can be denoted by either single (') or double (") quotation marks. For example, `'this is my string'`. However, you must be consistent. If you need to have one type of quote as actual data within a string, you can use the other type of quote to delimit the string. If you wish to extend a single string across multiple lines, you can use triple quotes at the beginning and ending of the string.

```
"""This is an example of a string that spans more than one
line. Denoting the string with three quotation marks allows
your strings to span multiple lines."""
```

- **Escape characters.** If you need to represent something that might cause problems within a string, such as quotes or newline or carriage return, you can use a backslash ('\') character, which is called 'escaping' the character. Two examples are \n and \r, which represent newline and carriage return.

```
>>> my_string = "This is a test of a newline. \nIt inserts
a line when the string is printed."
```

```
>>> print my_string

This is a test of a newline.

It inserts a line when the string is printed

>>>
```

- **Concatenation.** Strings can be concatenated using the addition symbol '+'. For example, 'my' + 'string'. You may need to add a space between the strings to be concatenated, but when you print, a space is added for you automatically.

- **Single character.** You can get a single character from a string by specifying its offset, or index, in square brackets 'my_string[n]'. For example, if my_string = 'Cisco', then my_string[2] would be the character 's'. Remember that in Python, indexes start at 0.

- **Slicing.** You can get a specific slice or subset of a string using offsets inside square brackets. With slices, you specify the start index, the end index, and the step. For example, my_string[0::2] would result in every second character from my_string. Default for the start index is 0, default for the end index is the last character in the string, and default for the step is 1.

```
>>> my_string = "Cisco Systems"

>>> print my_string[0::2]

CsoSses

>>> print my_string[0:5:1]

Cisco

>>> print my_string[6::]

Systems
```

- **Split.** You can split a string into a list of strings using the 'split' method, which will split the string using the split character you provide. For example, for a string with a CSV, you can type my_string.split(',') to split the string into a list of the separated values.

```
>>> my_string = "router, switch, access point"

>>> my_string.split(',')
['router', ' switch', ' access point']
```

You can also combine string operations. For example, you can combine the split method and slicing:

```
>>> device1 = my_string.split(',')[0]
>>> print device1
router
```

- **Strip.** You can use strip() to remove white space from the beginning or end of a string. In the split example, switch and access point both have a leading space. The strip method can be used to strip these leading spaces.

```
>>> my_string = "router, switch, access point"

>>> my_string.split(',')

['router', ' switch', ' access point']

>>> device2 = my_string.split(',')[1]
>>> print device2
     switch

>>> device2 = device2.strip()

>>> print device2
switch
```

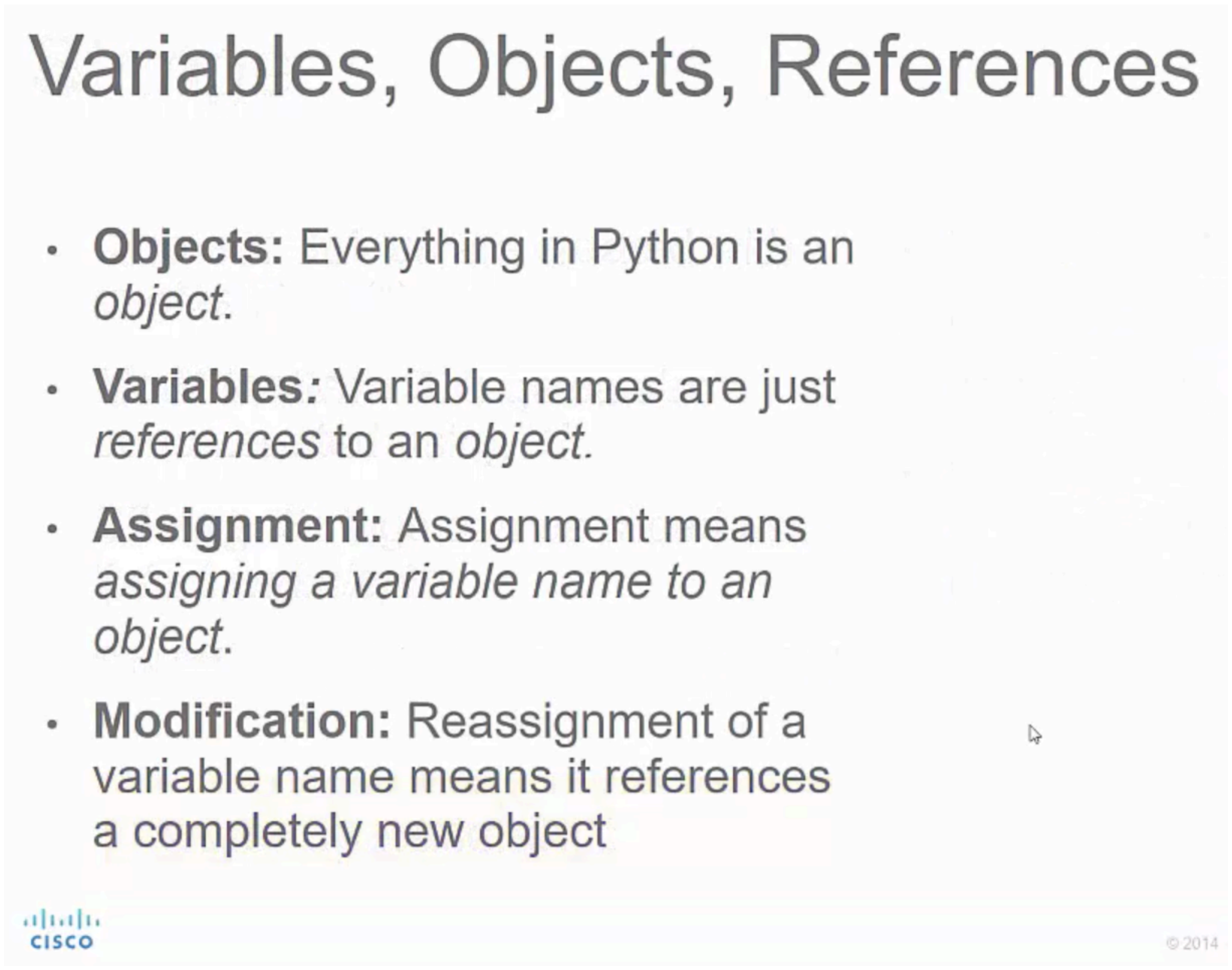
- **Length.** You can get the length of a string using len().

```
>>> my_string = "Cisco Systems"

>>> len(my_string)

13
```

There are several other string operations including join(), replace(), duplicate(), find(), and many others. Consult the Python documentation and other sources for a complete list and examples.



Variables, Objects, References

- **Objects:** Everything in Python is an *object*.
- **Variables*:*** Variable names are just *references* to an *object.*
- **Assignment:** Assignment means *assigning a variable name to an object*.
- **Modification:** Reassignment of a variable name means it references a completely new object

© 2014

Mutable vs. Immutable – Some data types can be changed; mutable, whereas others cannot, immutable.

Variables, Objects, and References

Consider your 'hello device' application.

```
ping = pexpect.spawn('ping –c 5 localhost')
```

The statement is creating a variable that is called 'ping', which is a reference to the object created by the 'pexpect.spawn(...)' method call. The result is that in 'ping' there is a reference to this object, which you can then use to call the 'expect' method.

```
result = ping.expect([pexpect.EOF, pexpect.TIMEOUT])
```

Note also that 'result' is also a variable pointing at the object returned by the 'ping.expect()' call.

It is important to understand that in Python everything is an object and variables are references to objects. When you create a variable in Python, you are creating a reference to the actual object you have created. So when you say:

```
>>> a=5
```

You are creating a variable that is named 'a', which points to the object which contains the value '5'. If you then declare another variable 'b':

```
>>> b=a
```

You are creating another variable and having it reference the same object that 'a' references.

```
>>> a
5
>>> b
5
```

If you later change the value that 'a' points to, Python creates another object, and now the values of 'a' and 'b' are referencing different objects.

```
>>> a=a+2
>>> a
7
>>> b
5
```

Printing

During development and with a finished application, it will be important to print results of your program. There are two methods for printing variables:

- **Print statement.** Using the 'print' function is suitable either during development to print state information, or as the output results of your application. Printing can be constant strings or variables which can be substituted or formatted to your satisfaction. One thing to note is that 'print' changed between Python 2 and 3. With Python 2, print was a keyword which could be invoked without parenthesis '()'. With Python 3, print became a function that you call, thus requiring parenthesis and the passing of parameters. Since the lab environment is using Python 2.7, you can use either method for printing.

- **Automatic.** With Python, it is also possible to have variables or constants be printed automatically, just by listing them as a statement. Note that the output includes quotes for strings:

```
>>>my_string = "Cisco Systems"

>>>my_string

'Cisco Systems'
```