Just Enough Python: Instructor Guide

CONFIDENTIAL

This guide is confidential, and contains Cloudera proprietary information. It must not be made available to anyone other than Cloudera instructors and approved Partner instructors.

Version	Release Date	Description
201511a	02/02/2016	Corrected name of exercise on slide 06-32 and in course timings of instructor guide.
201511	01/29/2016	Initial Release

NOTE: The Exercise Instructions follow the course slides.

Suggested Course Timings

Day 1 **[Total classroom time: 8 hours, 15 minutes]**

Arrivals and registration [15 minutes total]

1. **Introduction** [20 minutes total]
   - 20 minutes lecture

2. **Introduction to Python** [1 hour total]
   - 25 minutes lecture
   - 20 minutes exercise: Using IPython
   - 15 minutes exercise: Loudacre Mobile

3. **Variables** [1 hour, 10 minutes total]
   - 40 minutes lecture
   - 30 minutes exercise: Variables

4. **Collections** [1 hour, 25 minutes total]
   - 55 minutes lecture
   - 30 minutes exercise: Collections

5. **Flow Control** [1 hour, 15 minutes total]
   - 45 minutes lecture
   - 30 minutes exercise: Flow Control

6. **Program Structure** [1 hour, 25 minutes total]
   - 45 minutes lecture
   - 40 minutes exercise: Program Structure

7. **Libraries** [1 hour, 20 minutes total]
   - 35 minutes lecture
   - 45 minutes exercise: Libraries

8. **Conclusion** [5 minutes total]
   - 5 minutes lecture

Issues Fixed in this Release

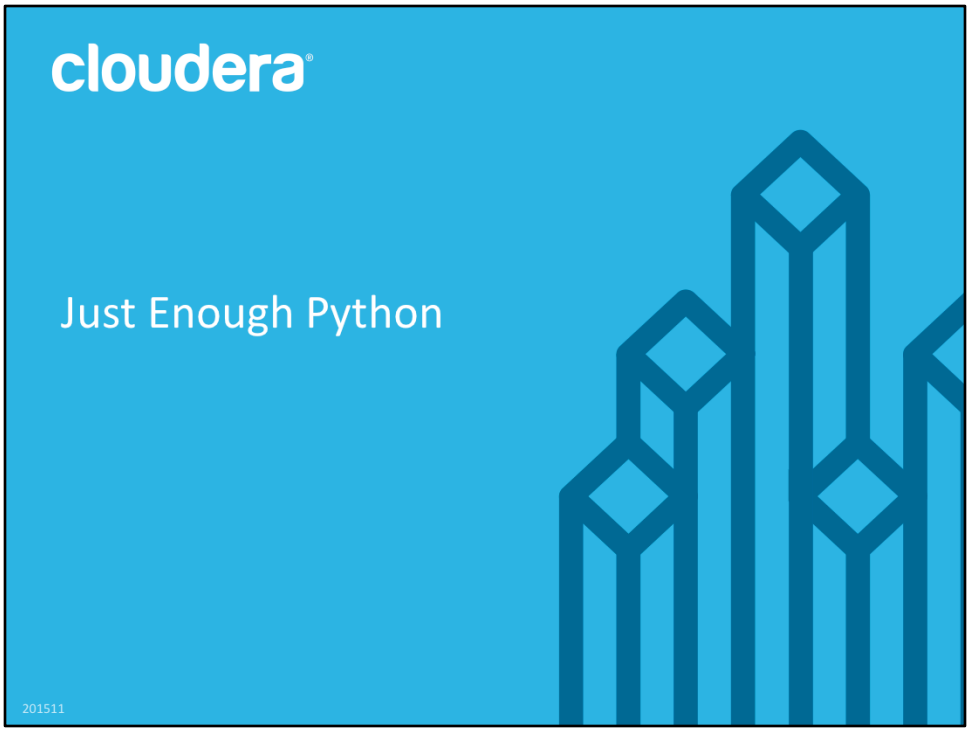
This is the first release of this course

Troubleshooting Advice

This section explains advice for identifying and resolving known issues.

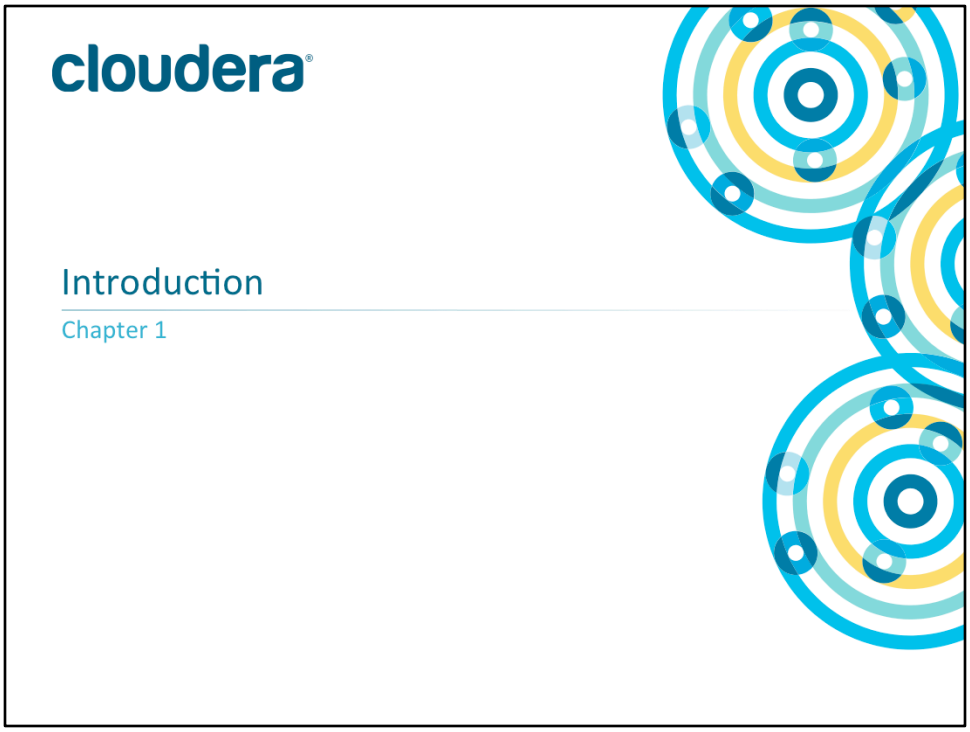
The Just Enough VM for JEP 201511 does not contain any Cloudera software. The class offers Python using IPython.

The class has not yet been delivered. When it has, check back here for issues and advice.

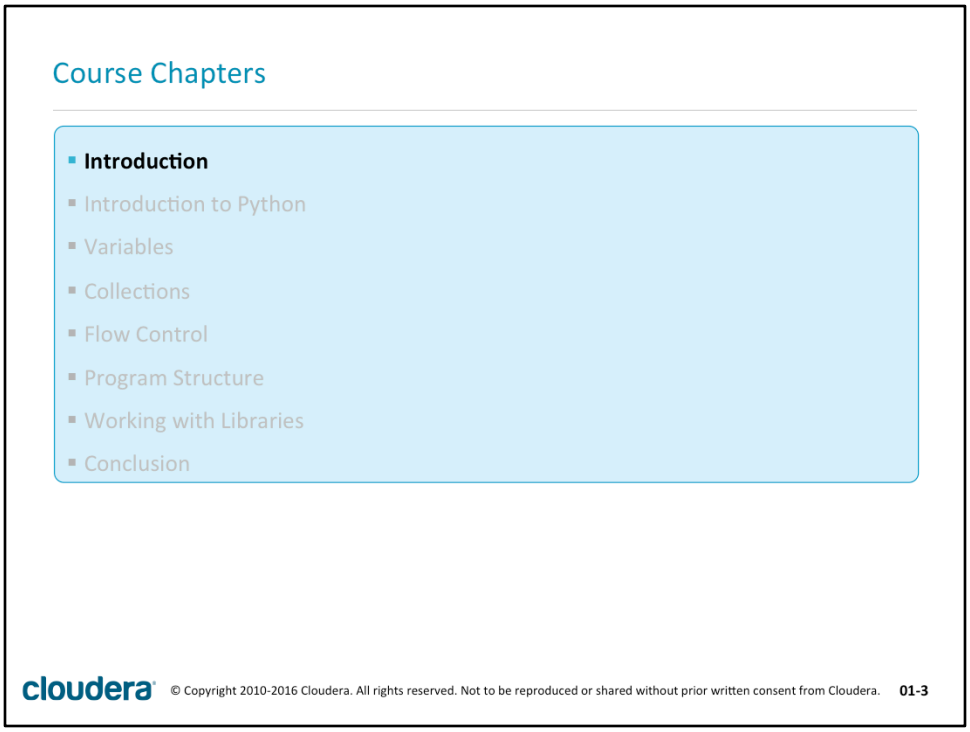


Welcome to "Just Enough Python"

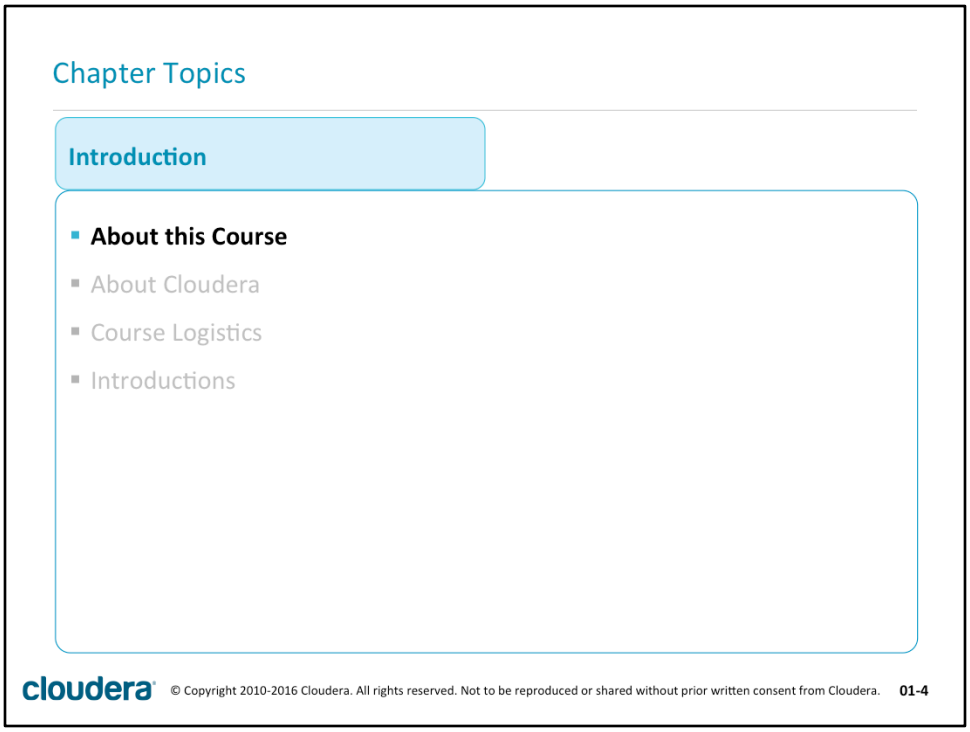
This is a crash course in Python that is intended to teach you "just enough" to be well prepared for Cloudera Developer courses that use Python.



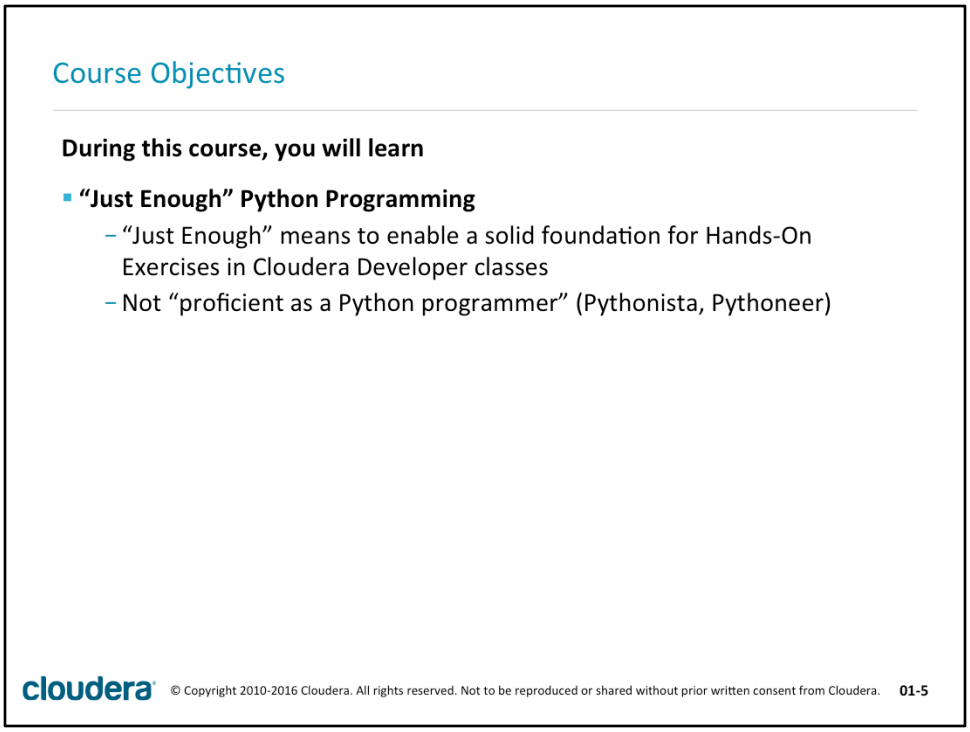
Goal: This chapter is intended to inform students what to expect from the course and for the instructor to learn about the students' level of expertise as well as how they plan to apply what they'll learn.



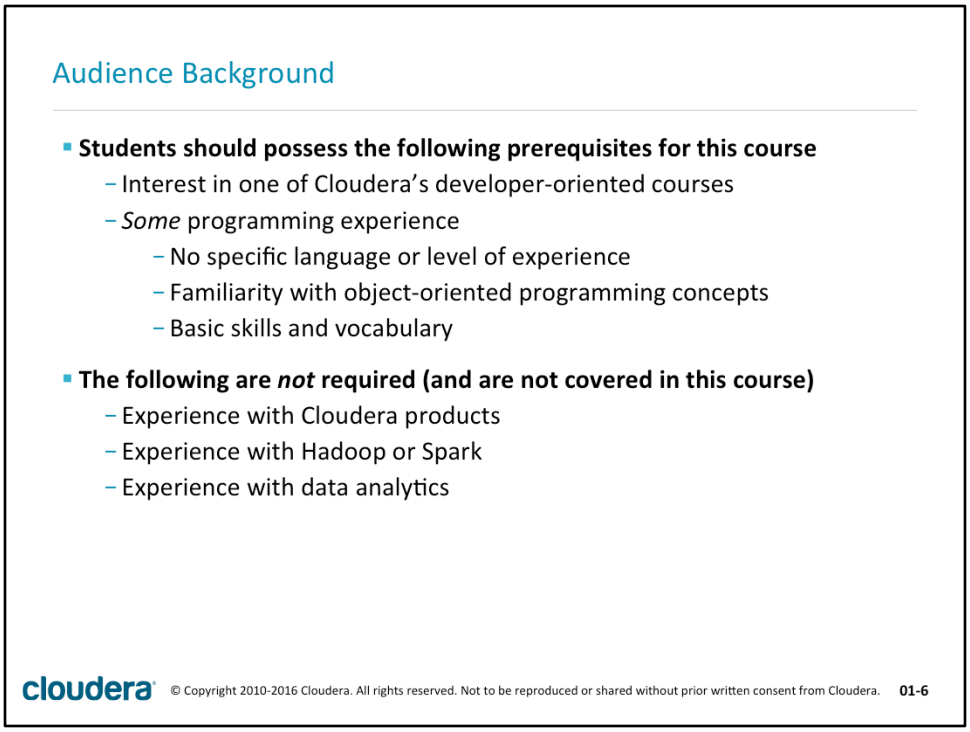
There are eight chapters in the course. We'll look at them in detail in the Introduction to Python (Chapter 2).



This slide shows the main topics covered in the current chapter. The upcoming topic is highlighted to illustrate what is about to be covered.



This class very specifically prepares you for Cloudera classes as rapidly and directly as possible. The class is not trying to cram everything into a crash course, and it is not a goal for you to become some kind of expert Python Programmer. The goal is to learn "Just Enough" Python so that learning the language is not a distraction from learning to develop with Hadoop or Spark in later classes.



You are assumed to have programmed in some language before – that 'print' and 'variable' will not be new concepts. Also, you are expected to know Object Oriented programming and what a 'class', 'object', 'instance', and 'method' are. The class does not assume any specific language. The class assumes that Big Data and Hadoop classes are ahead for you and that you will not know about Big Data technologies.

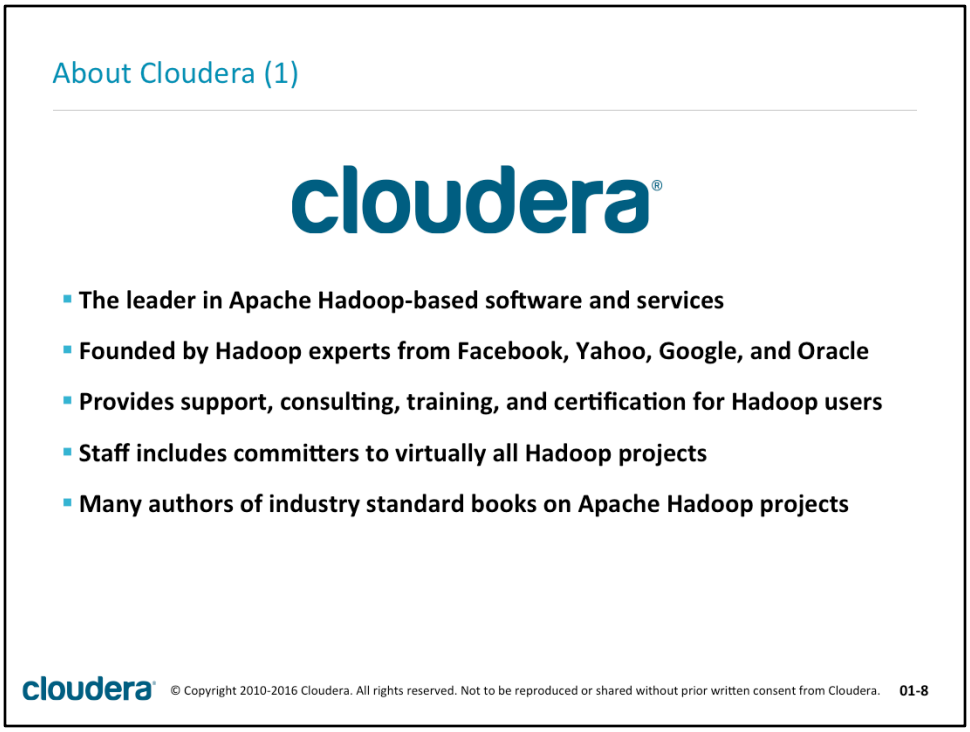
The class does not cover any Big Data concepts or techniques and does not cover any Cloudera software.

Note:
In general, people with a strong Java background will prefer the Just Enough Scala (JES) class to this class. Scala is a superset of Java. People with any other programming background will likely prefer this Just Enough Python (JEP) class because Python is a more automated and forgiving language and much easier to learn than Scala.

Chapter Topics

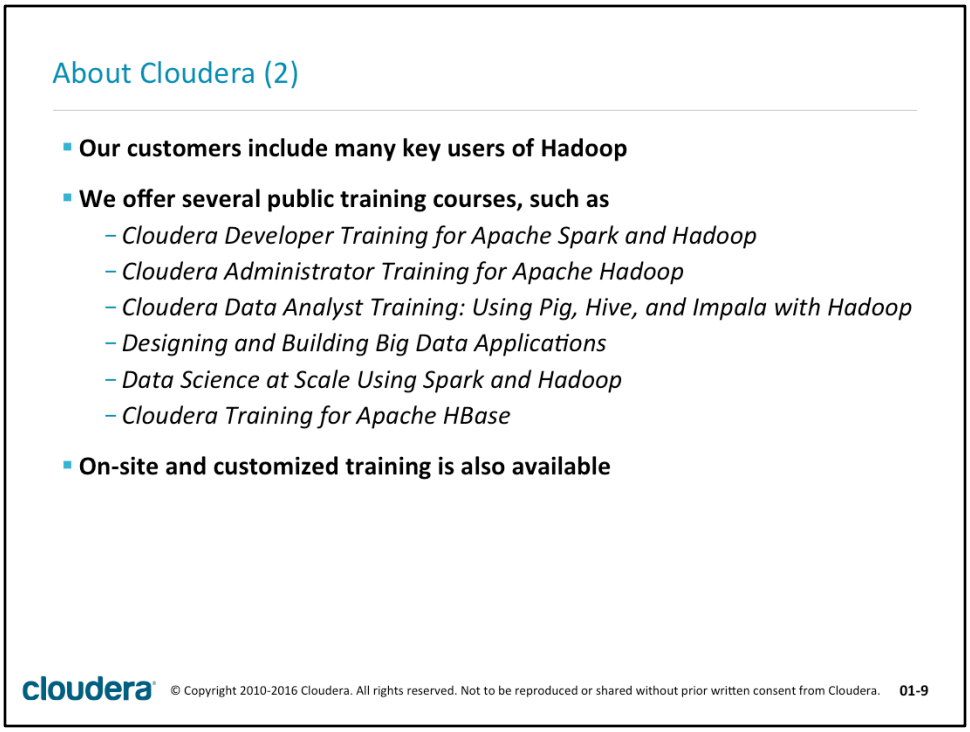
Introduction

- About this Course
- **About Cloudera**
- Course Logistics
- Introductions

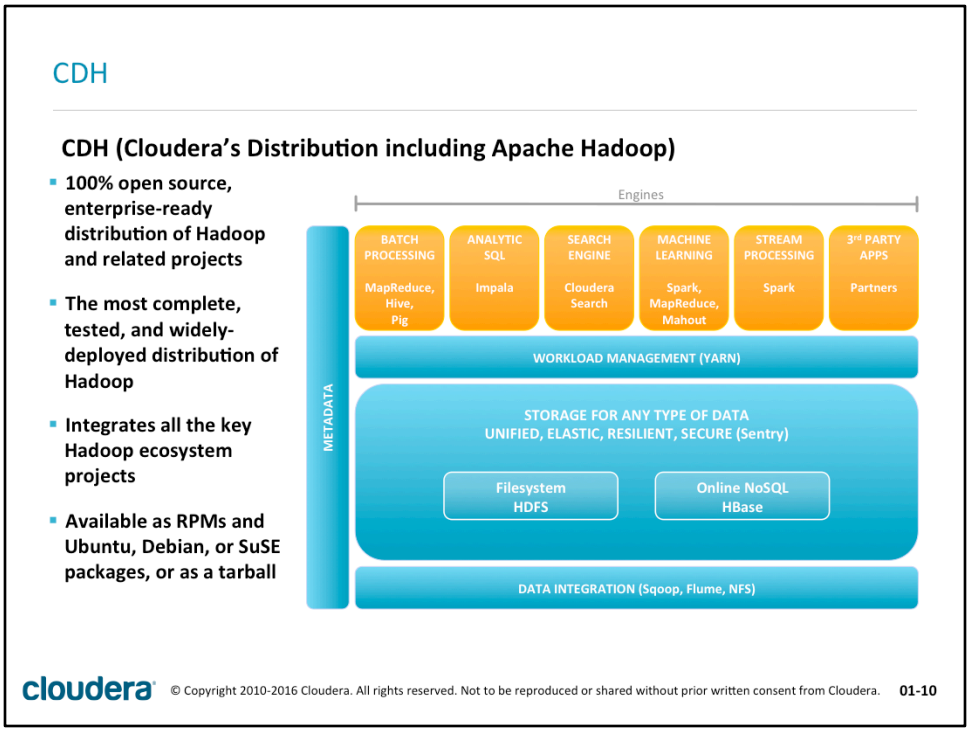


Cloudera was founded in 2008. Our staff also includes the ASF chairperson and creator of Hadoop, Doug Cutting, as well as many involved in the project management committee (PMC) of various Hadoop-related projects. The person who literally wrote the book on Hadoop, Tom White, works for Cloudera.

There are many Cloudera employees who have written or co-authored books on Hadoop-related topics, and you can find an up-to-date list here [`http://www.cloudera.com/content/cloudera/en/developers/home/hadoop-ecosystem-books.html`]. Instructors are encouraged to point out that many of these books are available at a substantial discount to students in our classes [`http://shop.oreilly.com/category/deals/cloudera.do`].



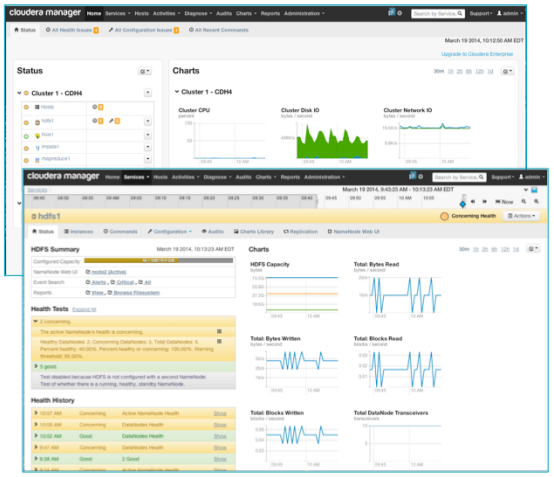
You can see a list of customers that we can reference on our Web site [`http://cloudera.com/content/cloudera/en/our-customers.html`]. Note that Cloudera also has many customers who do not wish to be referenced, and it is essential that we honor this. The only exception to this important rule is that you may reference something that was intentionally made available to the public in which Cloudera or that customer has disclosed that they are a Cloudera customer. For example, it is permissible to mention an article in a reputable trade publication in which Cloudera's CEO mentions a specific customer or the keynote address that the customer's CTO gave at the Strata conference talking about the benefits they've experienced as a Cloudera customer.



You can think of CDH as analogous to what RedHat does with Linux: although you could download the "vanilla" kernel from kernel.org, in practice, nobody really does this. If you're in this class, it is probably because you're thinking about using Hadoop in production and for that you'll want something that's undergone greater testing and is known to work at scale in real production systems. That's what CDH is: a distribution which includes Apache Hadoop and all the complementary tools you'll be learning about in the next few days, all tested to ensure the different products work well together and with patches that help make it even more useful and reliable. And all of this is completely open source, available under the Apache license from our Web site.

Cloudera Express

- **Cloudera Express**
  - Completely free to download and use
- **The best way to get started with Hadoop**
- **Includes CDH**
- **Includes Cloudera Manager**
  - End-to-end administration for Hadoop
  - Deploy, manage, and monitor your cluster

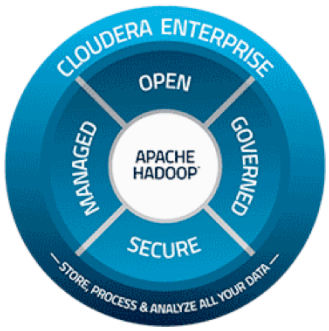


Main point: Cloudera Express is free, and adds Cloudera-specific features on top of CDH, in particular Cloudera Manager (CM).

Note in case anyone asks: the free version of Cloudera Manager (i.e., the product now known as Cloudera Express) once had a 50-node limit, but no longer does.

Cloudera Enterprise

- **Cloudera Enterprise**
  - Subscription product including CDH and Cloudera Manager
- **Includes support**
- **Includes extra Cloudera Manager features**
  - Configuration history and rollbacks
  - Rolling updates
  - LDAP integration
  - SNMP support
  - Automated disaster recovery
- **Extend capabilities with Cloudera Navigator subscription**
  - Event auditing, metadata tagging capabilities, lineage exploration
  - Available in both the Cloudera Enterprise Flex and Data Hub editions

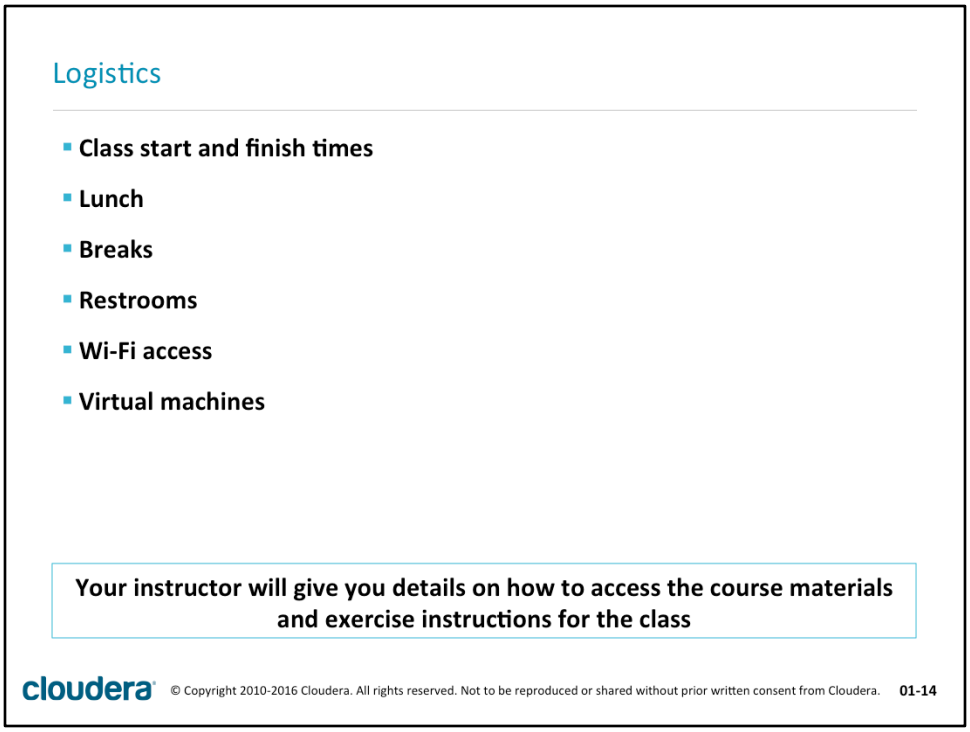


LDAP = Lightweight Directory Access Protocol
SNMP = Simple Network Management Protocol

Chapter Topics

Introduction

- About this Course
- About Cloudera
- **Course Logistics**
- Introductions



This is a good time to start the VM, if they're not already running.

The registration process for students is:

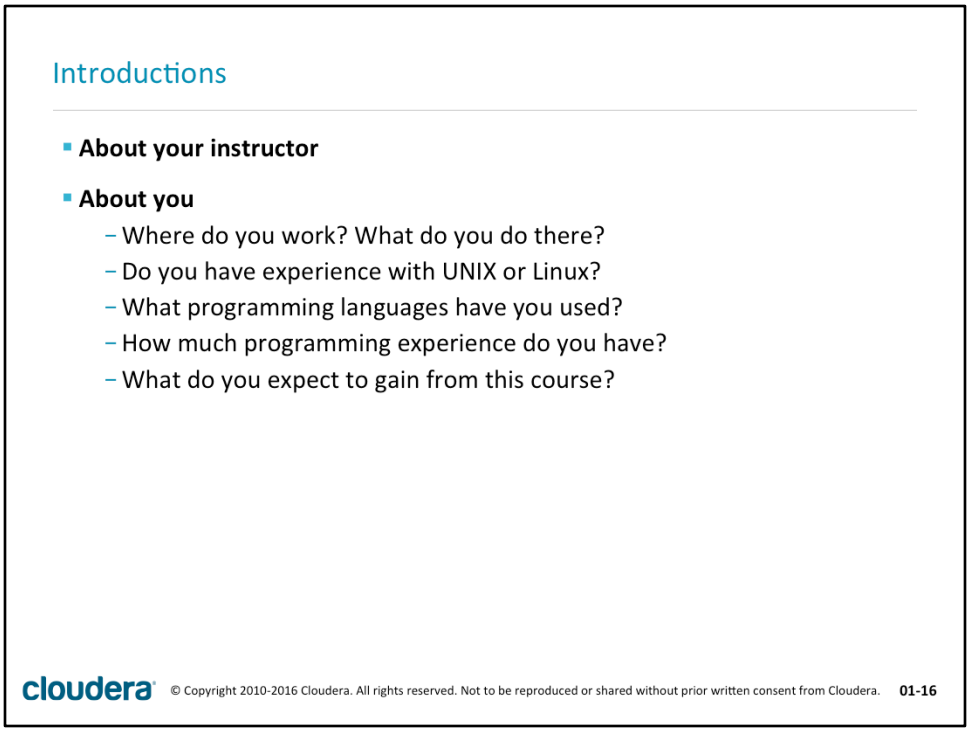
1. Visit [`http://training.cloudera.com/`]
2. Register as a new user (ensure that you give an e-mail address which you can check from here, since the system will send a confirmation e-mail to you)
3. Confirm registration by clicking on the link in the e-mail
4. Log in (if the system has not already logged you in)
5. Enter the course ID and enrollment key (which the instructor will have been sent the week before the class starts)
6. Once this is done, the student will be taken to the course page, from which they can download the slides and exercise instructions. Emphasize that they must, at the very least, download the exercise instructions. Unless this is an onsite course they should not download the VM – it is already on the classroom machines, and downloading it here will consume too much bandwidth.

Note: The VM for this course provides the standard hands-on training environment and experience, but does not contain CDH (nor any Cloudera software, for that matter). Also, note that the VM currently provides Python 2.6.6, which works with CDH.

Chapter Topics

Introduction

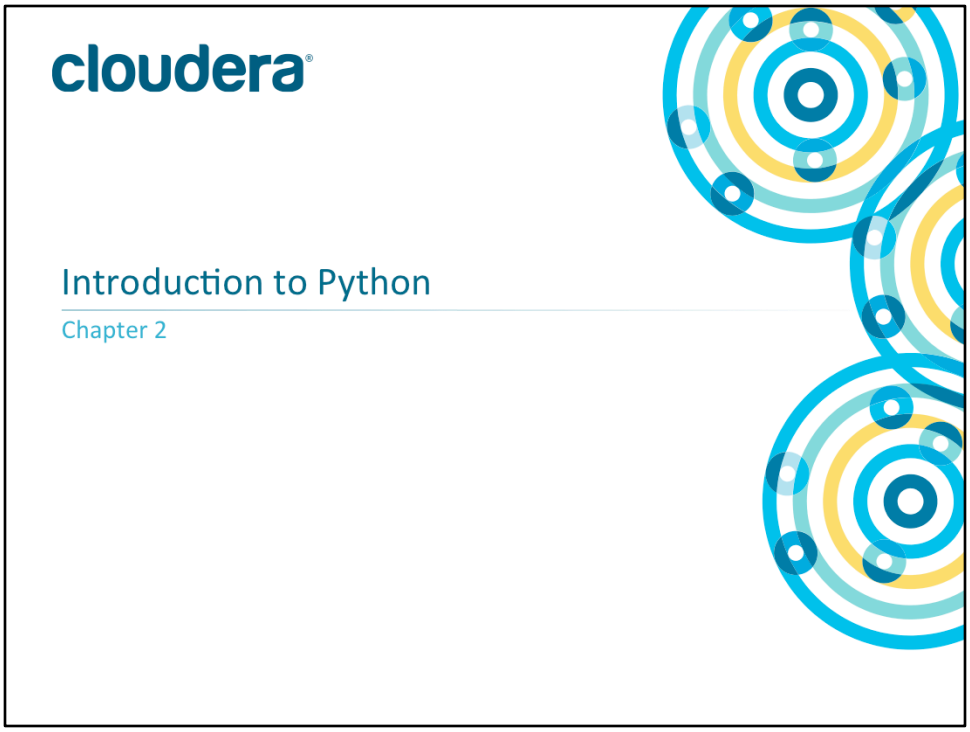
- About this Course
- About Cloudera
- Course Logistics
- **Introductions**



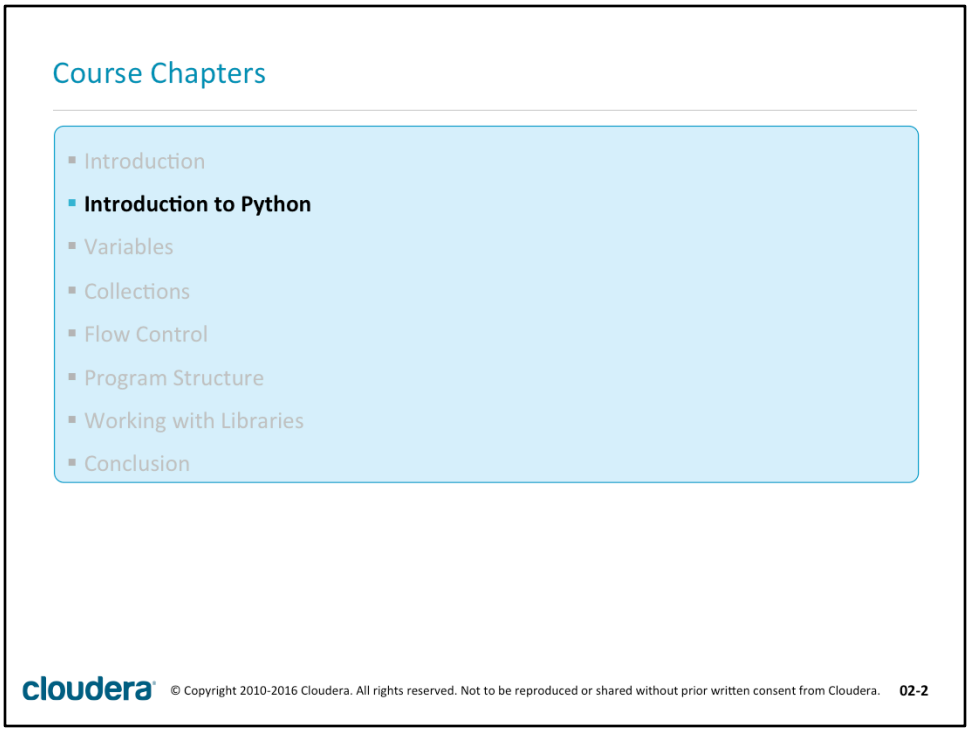
Establish your credibility and enthusiasm here. You'll likely want to mention your experience as an instructor, plus any relevant experience as a developer, system administrator, DBA or business analyst. If you can relate this to the audience (because you're from the area or have worked in the same industry), all the better.

This is also an opportunity to get to know the students, so you can tailor your explanations and analogies to the experience that they already have. It is a good idea to draw out a grid corresponding to the seat layout and write students' names down as they introduce themselves, allowing you to remember someone's name a few days later based on where they're sitting.

The outline for all our courses are available online [`http://cloudera.com/content/cloudera/en/training/courses.html`], so you should be familiar with them and will know whether a student's expectations from the course are reasonable.



We have a number of general items to cover before starting the technical content and learning to program in Python.

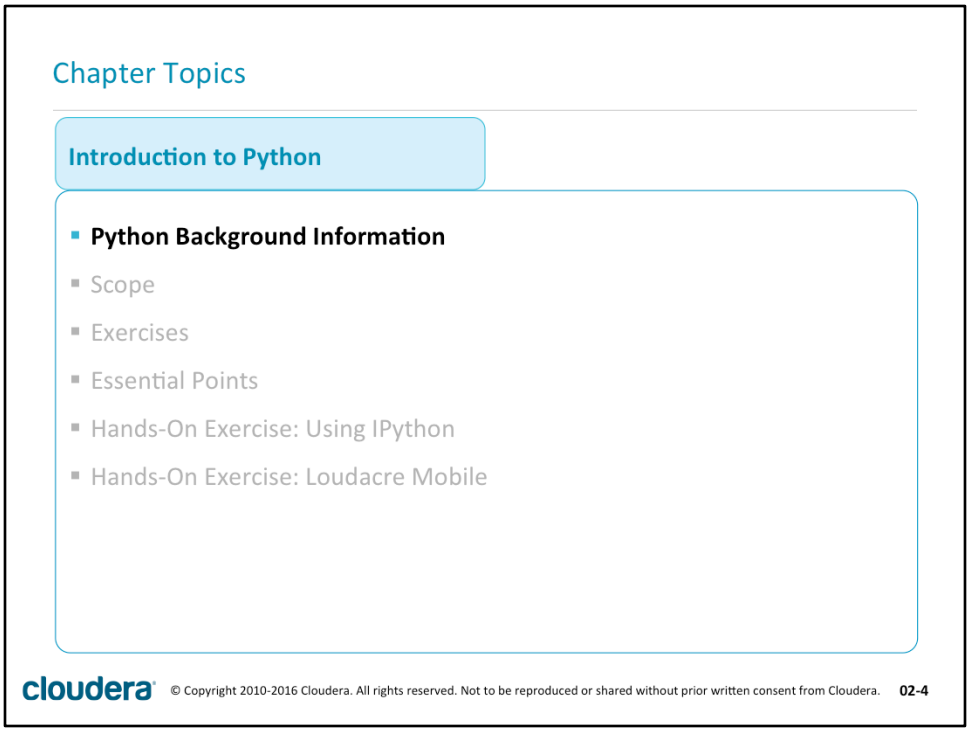


There are eight chapters in the course.

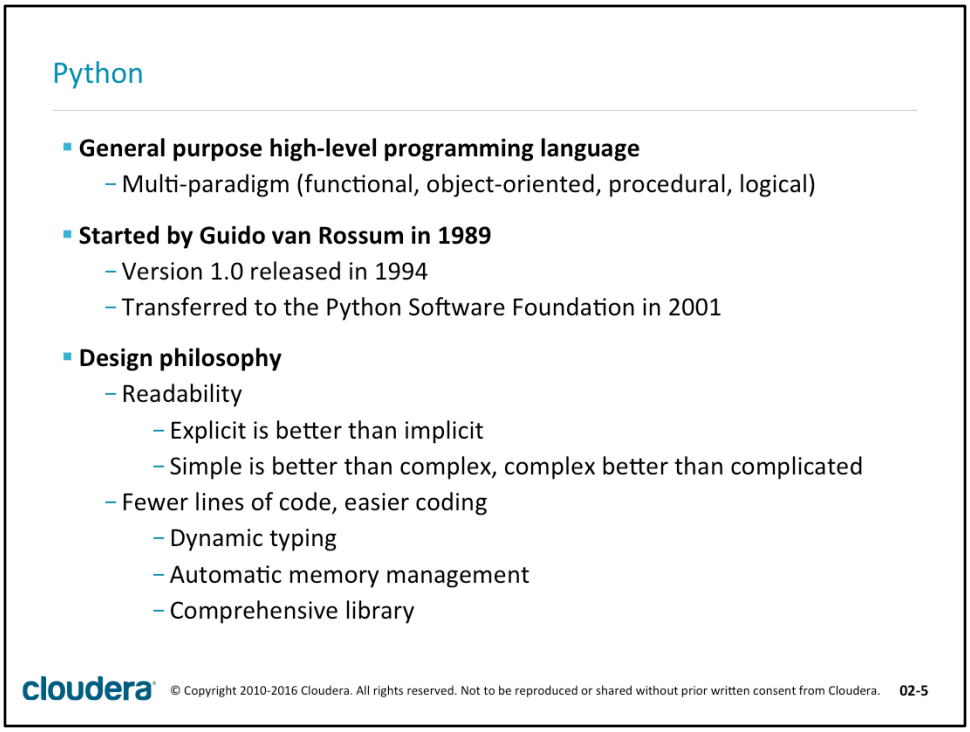
Introduction to Python

In this chapter you will learn

- **The history of Python and what makes Python special among languages**
- **The purpose and scope of the class, and what parts of Python will be discussed and what topics are beyond the aims of this class**
- **Explanation of how Python syntax is represented in the class slides**
- **Introduction to the Hands-On Exercise environment including IPython**

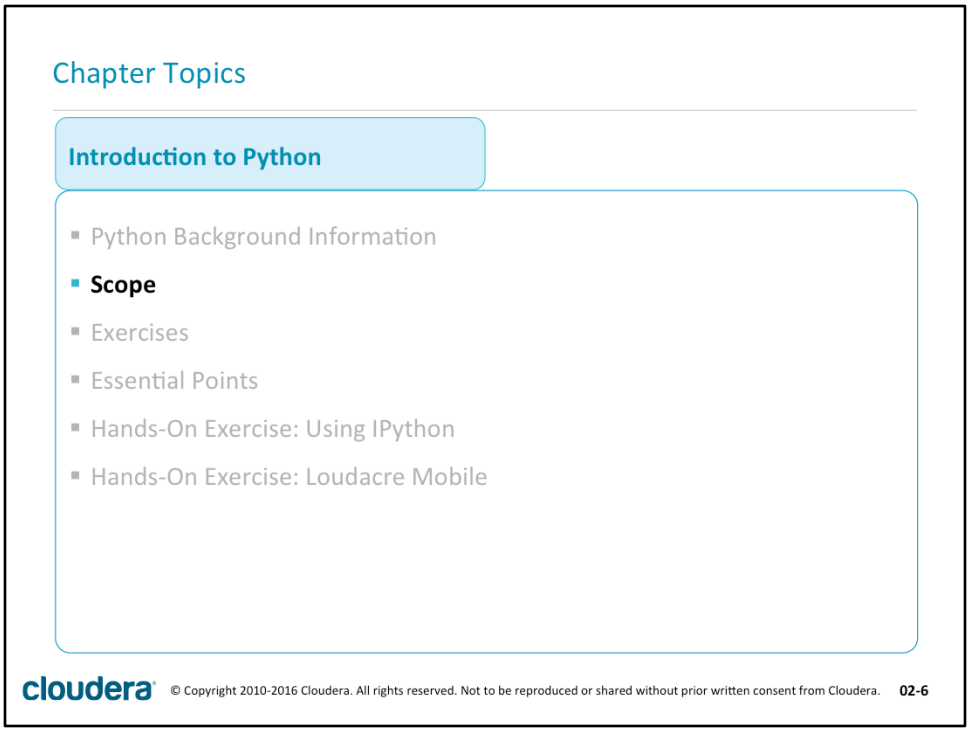


Let's discuss a little about Python, and why it has become a popular language in the Big Data era.

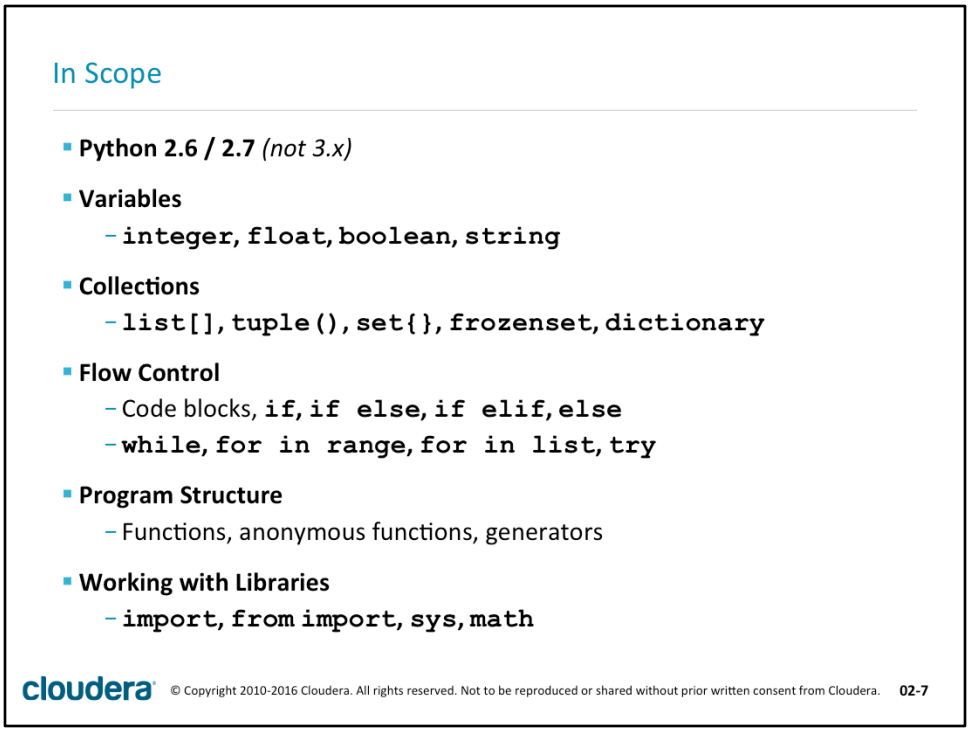


Python was created in 1989 by Guido van Rossum in the Netherlands. He named it after "Monty Python's Flying Circus". It was intended to be a language that was more readable and less laborious to code than other languages of the time. Version 0.9.0 was first made public in 2/1991. Version 1.0 was published in 1994. Guido passed control to the Python Software Foundation in 2001 and continues to steer development of the language. His PSF title is "BDFL" – Benevolent Dictator For Life.

One of the key design ideas that's embodied in Python (if you read between the lines here…) is "Let the Computer Do The Work". There are a lot of things the computer can figure out for you. And when the computer does that, you don't have to. So a lot of the coding that is performed by ritual in other languages like Java and C++ is just absent in Python, making it much easier to get results quickly. Python has become a very popular language because the TCO (total cost of ownership – cost to develop and maintain Python code) is overall lower than the TCO for other languages such as Java, Ruby, and so forth.

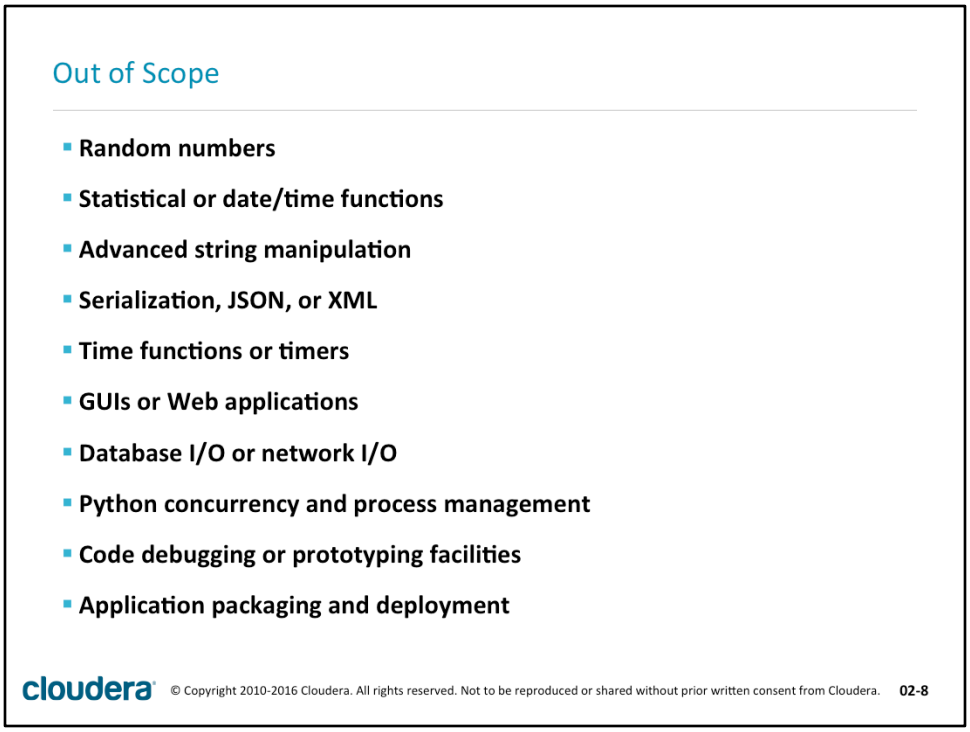


Python programming can be a semester-long course – or two of them. It is powerful, but deep. So… we had to be very selective about what we chose to be part of this course and what we left out. Let's explore the reasoning behind those choices.



Subjects covered in this course.
Notice that we've come up with this sorting for you. By narrowing the scope, we can spend a decent amount of time and focus on each core subject so that you'll have a solid foundation in the basics of Python as used in our classes. In some cases we cover slightly more Python than is used in our classes to make sure your base of skills is solid and complete.

About 130+ concepts covered in the course. The training method we use is to introduce the concepts in the course and then jump into hands on exercises to anchor the concepts as practical skills.



Subjects that are specifically NOT covered in this course. And this might serve as a follow-on checklist of items to learn if you DID want to become a Pythonista later.

Chapter Topics

Introduction to Python

- Python Background Information
- Scope
- **Exercises**
- Essential Points
- Hands-On Exercise: Using IPython
- Hands-On Exercise: Loudacre Mobile

Python Shell – REPL

- **The Python shell is entered by typing `python`**
  - The default shell
  - Does not interface with the host file system, no path completion
  - Does not provide command completion
  - Is REPL: Read – Evaluate – Print – Loop
    - When you enter an expression, Python immediately evaluates it, assigns the value to an implicit variable, and prints it to the console
    - REPL is excellent for interactive code exploration

```
$ python

>>> a = 123
>>> 123
>>> type(a)
>>> type <'int'>
```


More info on REPLs can be found on Wikipedia: [`http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`].

IPython Shell

- **The IPython shell is entered by typing `ipython`**
  - IPython is a separate project from Python
  - Provides programming features over the default shell
  - File system interaction, **`ls`**, pathname completion *(tab)*
  - Command completion and suggestions *(tab-tab)*
  - Integrated Python help / documentation with **`help()`**
  - Run files from within the shell with **`run filename`**
  - Interrogate objects with **`?object`**
  - Edit files with **`%ed`**
  - After editing the program, if you exit using **`[Esc]wq`**, the program will be run

This is the shell we will use in class

```
$ ipython

In [1]: a = 123
In [2]: print a
123
In [3]: type(a)
Out[3]: int
```

Example of the look of the IPython shell

Note:
Some people report that exiting the editor using shift-z-z (ZZ), the program **may** not run.
The course author tested it on the 201511 Just Enough VM and found that it ran every time.

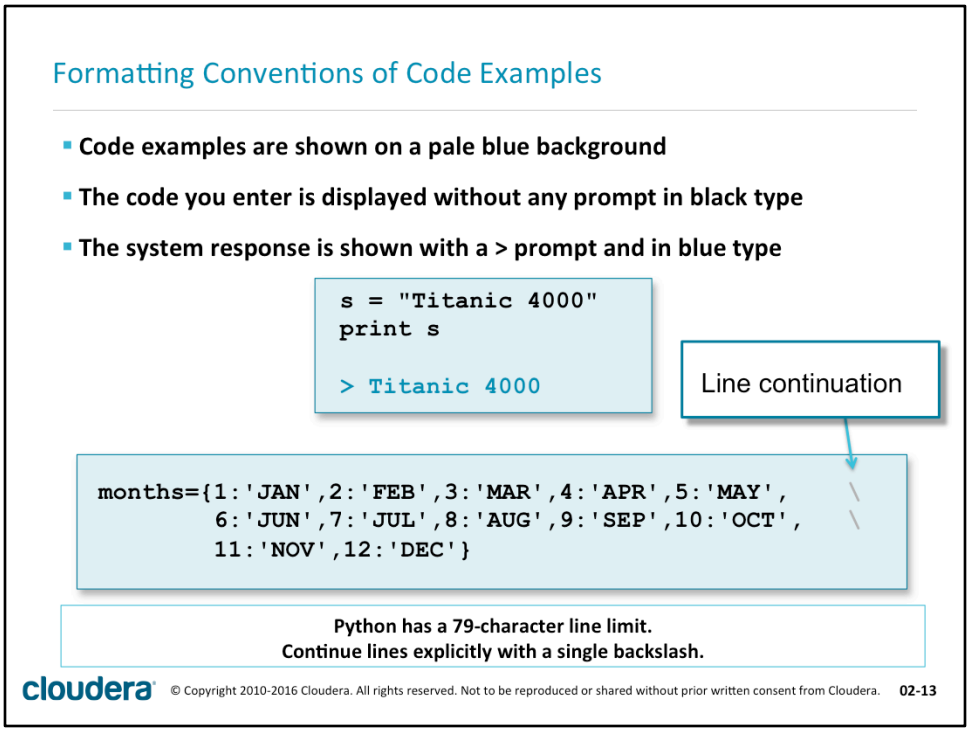
Some people prefer `:wq`
The program is run consistently when exiting using `:wq`

Formatting Conventions of Documentation

- **Keywords and syntax in bold**
- **Developer-supplied names and variables in italics**
- **Developer-provided code in gray**
- **Call-outs in italic blue text**

```
def name(parameters):
    code-block
    return variable
```

Optional return variable



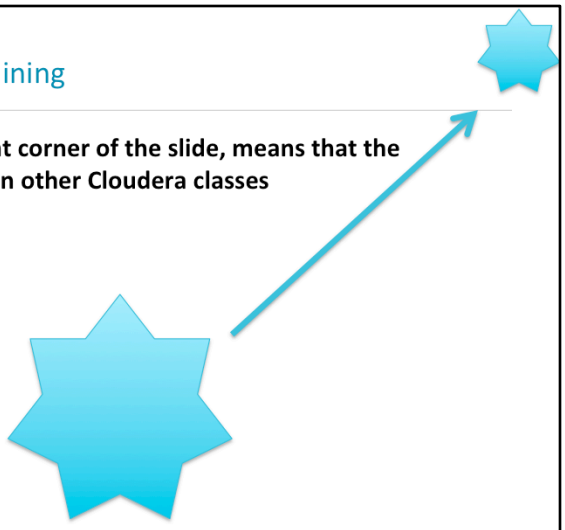
Python Line Limit is 79 characters as of release PEP-8 (2008).

The preferred way of wrapping long lines is by using Python's **implicit** line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using Python's **explicit** continuation, which is a single backslash.

Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is after the operator, not before it.

Used In Other Cloudera Training

- **This symbol, in the upper right corner of the slide, means that the technique is frequently used in other Cloudera classes**

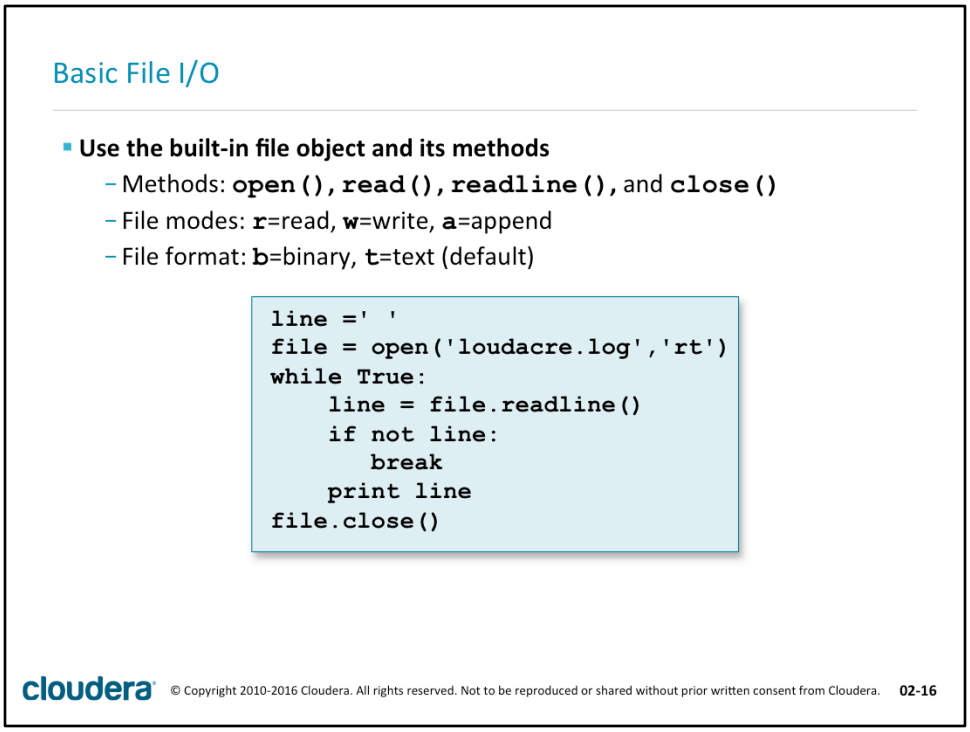


Loudacre Mobile

- **Loudacre is a mobile phone carrier**
  - They have provided us with realistic device log data from their mobile phones which we will be using in our exercises
  - Loudacre is a fictional company
- **About the log data**
  - Every time a phone has a catastrophic error requiring a soft or warm restart, the phone reports device status information and sends it to the central system where it is collected for later analysis

Loudacre mobile

Note: Review the next several slides on basic programming before going to the first preparatory exercise.



`readline()` reads to the EOL character at the end of the line.

Note: This code is a direct example of what is needed in the exercise. Please review this code with the class carefully.
Some of the Flow Control elements are covered in detail in a later chapter. However, if you cover some of them here it is sufficient for the exercise at the end of this chapter.

Basic Printing and Keyboard Input

- **Use the built-in `print()` function**
  - Basic printing of variables: **`print`** ***`var1, var2, var3`***
  - Print formatted string:
    - **`print`** ***`formatstring`*** **`%(`*****`var1, var2, var3`*****`)`**
    - Formatting: **`%d`**=integer, **`%r`**=real (float), **`%s`**=string

```
s = "Sorrento F41L"
print s

formatstring = "Device temperature %d to %r celsius"
print formatstring %(24, 31.24)

> Sorrento F41L
> Device temperature 24 to 31.24 Celsius
```


Basic Keyboard Input

- Use the `raw_input()` function

```
s = raw_input("Enter: ")
print s

> Enter:
> Enter: Titanic 4000
> Titanic 4000
```


Chapter Topics

Introduction to Python

- Python Background Information
- Scope
- Exercises
- **Essential Points**
- Hands-On Exercise: Using IPython
- Hands-On Exercise: Loudacre Mobile

Essential Points

- Background information about Python
- Scope and version coverage
- Symbolism and syntax used in slides
- Introduction to exercises, IPython shell, and Loudacre

Chapter Topics

Introduction to Python

- Python Background Information
- Scope
- Exercises
- Essential Points
- **Hands-On Exercise: Using IPython**
- Hands-On Exercise: Loudacre Mobile

Hands-On Exercise: Using IPython

- **Start using IPython shell**
  - Use the shell interactively (REPL)
  - Try some simple Python
  - Use the IPython shell to execute OS commands, **`pwd`** and **`ls`**
  - Try the integrated **`help()`** system
  - Edit and run a program using **`%ed`**

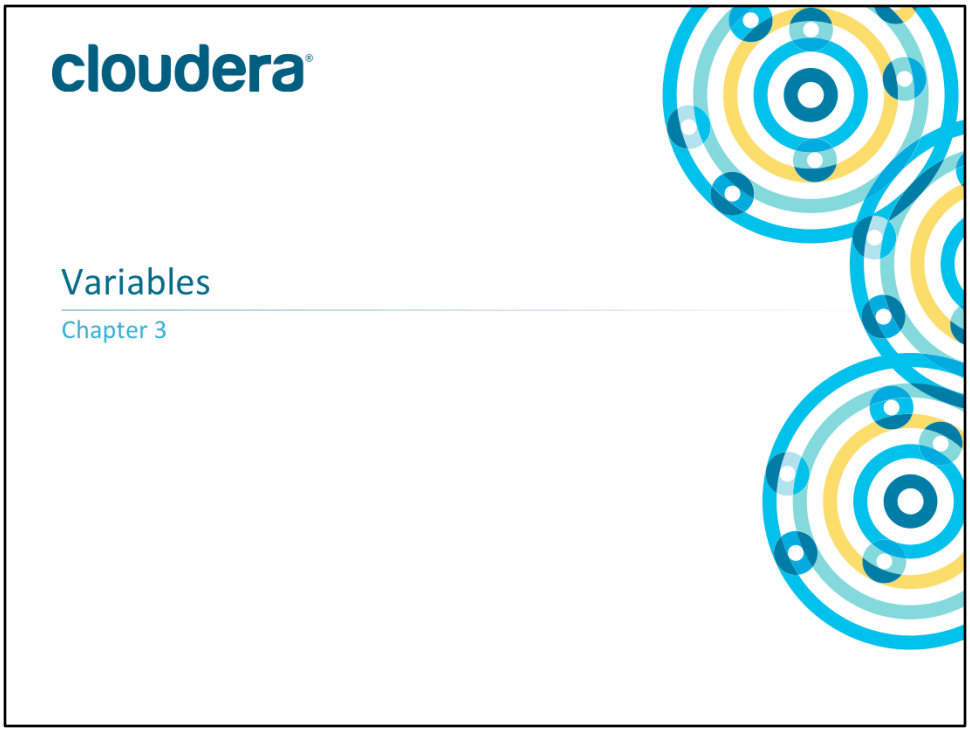
Chapter Topics

Introduction to Python

- Python Background Information
- Scope
- Exercises
- Essential Points
- Hands-On Exercise: Using IPython
- **Hands-On Exercise: Loudacre Mobile**

Hands-On Exercise: Loudacre Mobile

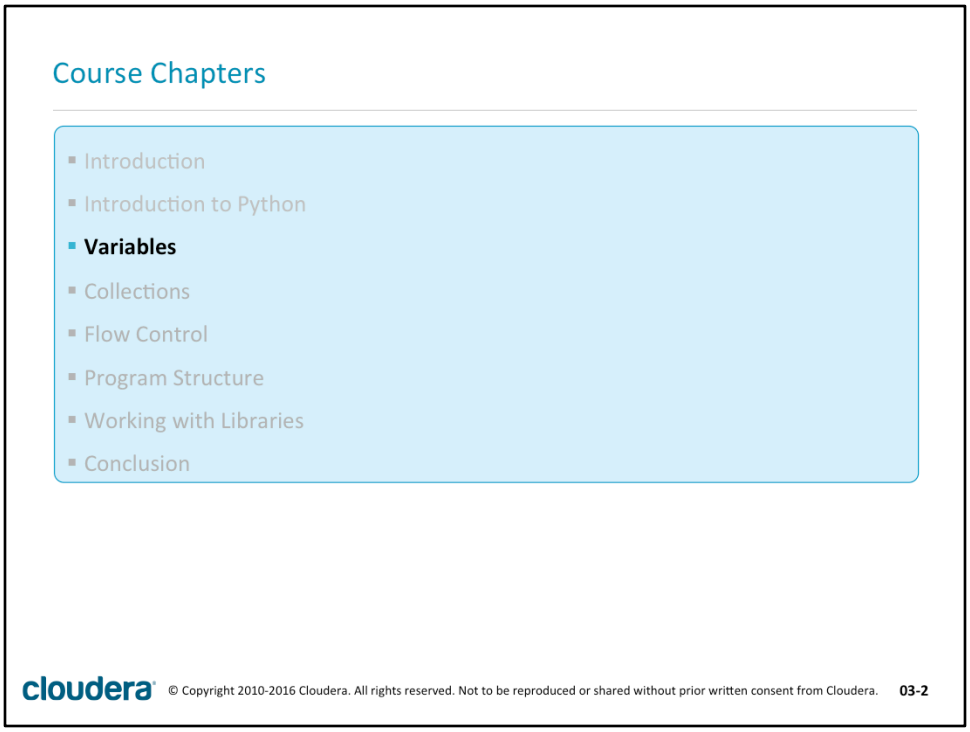
- **Introduction to the `loudacre.log` data**
  - Edit and run a file I/O program
  - Read each line of **`loudacre.log`** and print it



In this chapter we will explore basic variables.

The data structures supported in a language have a profound influence on the general approach to problem-solving and the way in which programs are written.

Note: important points are identified with '[*]'



There are eight Chapters in the course. We'll look at them in detail in just a moment.

Variables

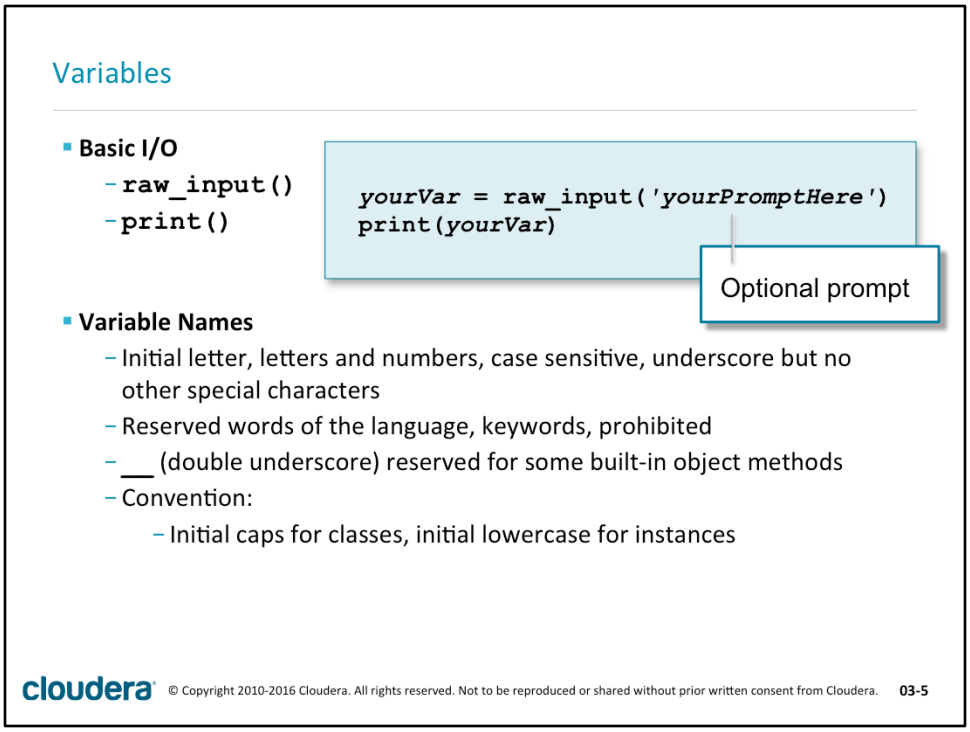
In this chapter you will learn

- **How to create new variables without explicitly declaring them, using Python's dynamic typing**
- **How to distinguish between Mutable and Immutable properties of variables**
- **What is the basic usage and manipulation of numerical variables, booleans, and strings?**
- **How to build sophisticated string transformations using chaining, slices, and concatenation techniques**

Chapter Topics

Variables

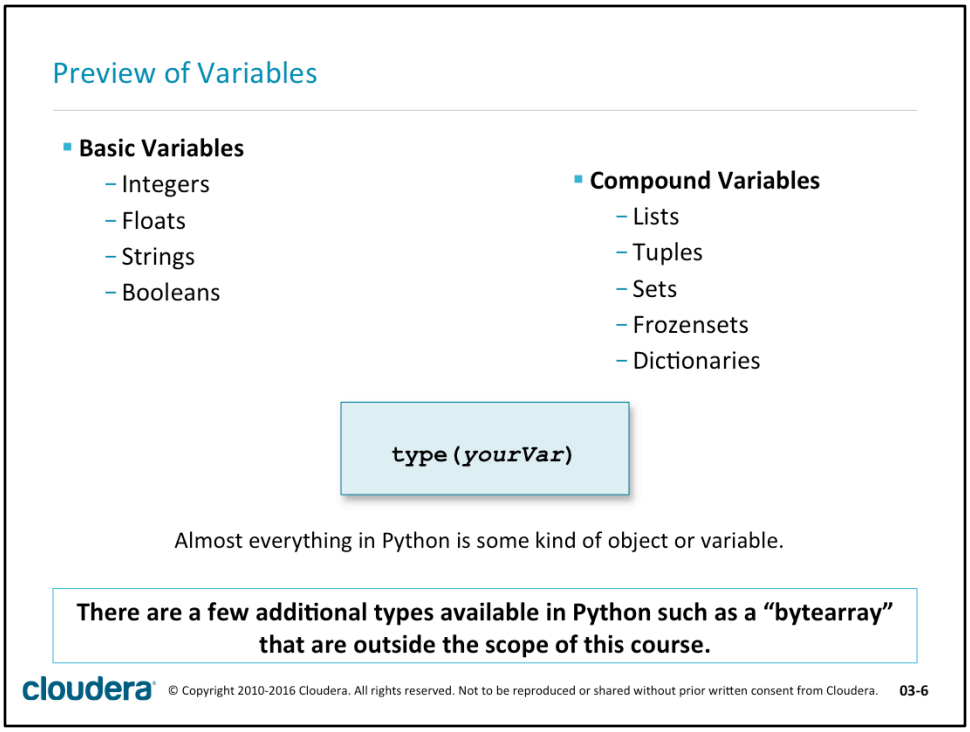
- **Python Variables**
- Numerical
- Boolean
- String
- Essential Points
- Hands-On Exercise: Variables



Basic I/O – Python supports Web I/O and GUI I/O through optional modules, but this course will only use console I/O (keyboard and display). We don't need "fancy" formatted I/O, since we are mainly just using the print() function to explore the results of applying methods to variables. Variable names are just as you'd expect.

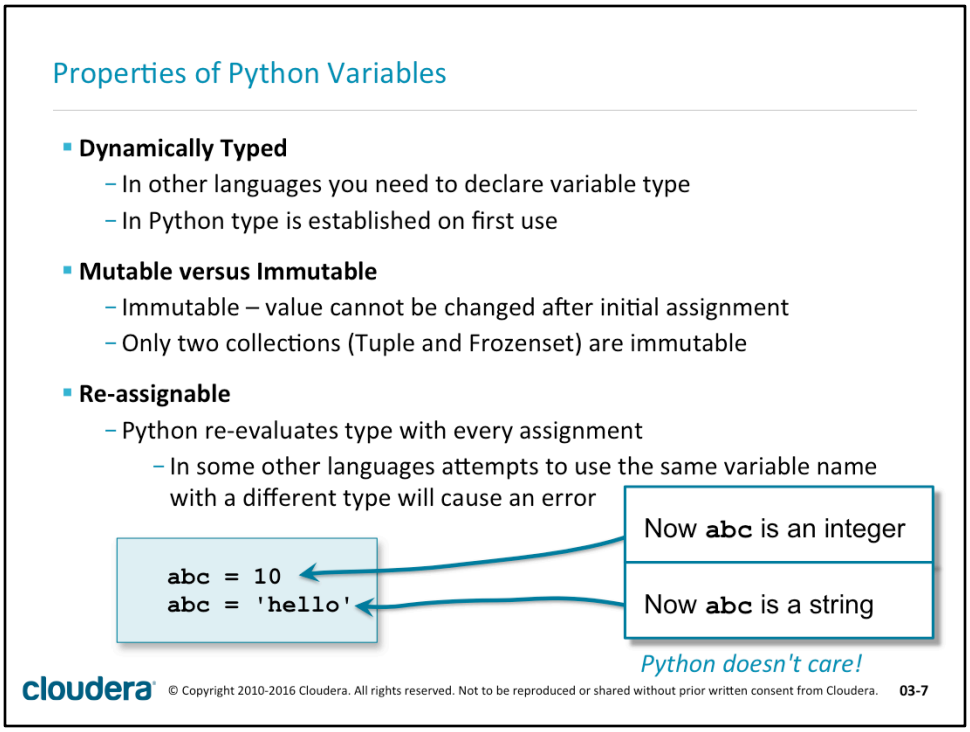
Familiarity with OO is a pre-requisite for this course, however, because these items were not previously defined, they are defined here:

- **Built-in Object Methods** – Methods of objects that are defined in the base language (not in an optional or external library) and do not need to be imported are prefixed with a double underscore. One example is __init__ You don't have to define it, Python will add it automatically.
- **Classes** – From Wikipedia Programming Glossary: "An object-oriented programming , a class is a template definition of the method s and variable s in a particular kind of object . Thus, an object is a specific instance of a class; it contains real values instead of variables."
- **Instances** – From Wikipedia: "In object-oriented programming (OOP), an instance is a specific realization of any object. Formally, "instance" is synonymous with "object" as they are each a particular value (realization), and these may be called an instance object; "instance" emphasizes the distinct identity of the object. The creation of an instance is called instantiation."
- **Object** – From Wikipedia: "In computer science, an object is a location in memory having a value and possibly referenced by an identifier. An object can be a variable, a data structure, or a function. In the class-based object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures."



These are the variables we will be discussing in the remainder of the class.

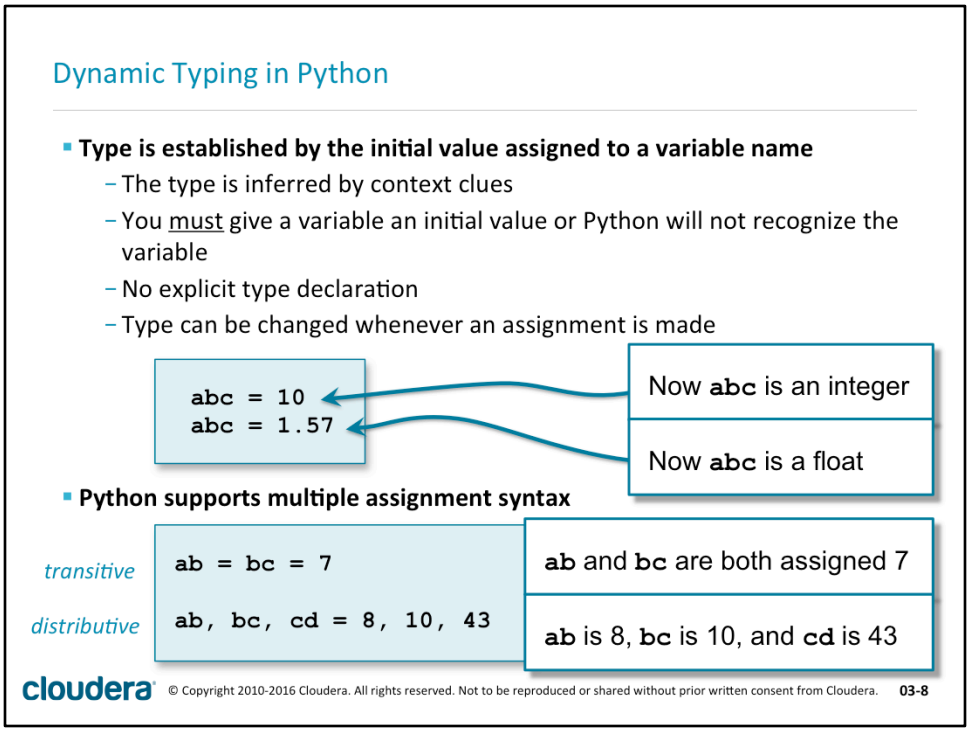
[*] If you are ever confused by the Dynamic Typing in Python, you can always use the type() function to find out the current type of a variable.



We've covered Dynamically Typed. However, there are a few other properties that apply to variables and will help the students relate Python to whatever language they already know. These are 'mutable' versus immutable.

[*] Mutable. All of the Basic variables in Python are mutable. And most of the Compound variables in Python are mutable. Only a couple of compound variables (collections) are immutable. The number one cause of bugs in multi-threaded code are data items that can change value while being used concurrently. This creates bugs that are really hard to track down and maybe be subject to race conditions. Using immutable data means that the value of data can't be changed after it is initialized. So, basically, if a concurrent thread needs to change a value, it has to make its own copy of the variable rather than potentially changing a variable that other code depends on.

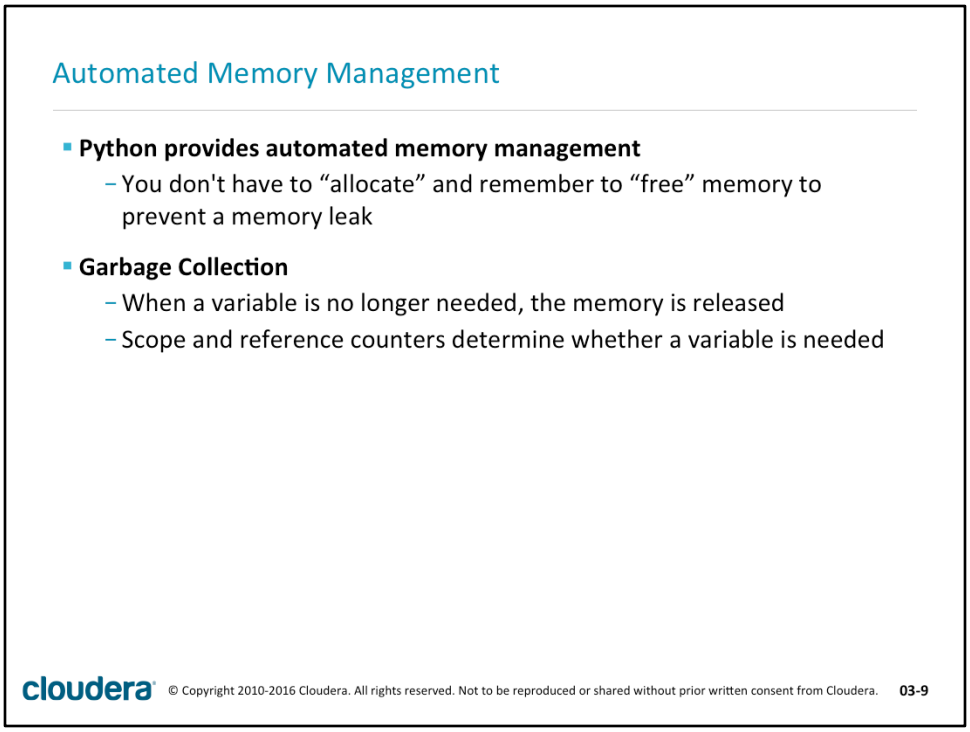
[*] Re-assignable. Another challenge for people coming from strongly typed languages is that a variable can be reassigned to a different type within a single program. "With great power comes great responsibility" (Spider Man, the Movie). Programmers who rely on the compiler to tell them to error when re-using a variable name as a different type need to be more careful. On the other hand, they no longer need to declare the type of a variable before it is used – so a lot of typing is saved and the code is a lot easier to read.



[*] Dynamic typing. Python provides Dynamic Typing. Part of the philosophy of Python is that human beings shouldn't have to do work that computers can do. And this applies to determining the type of variable that should be used. Python can determine the type using inference from the context. For example, if the right side of an assignment has all numerals and a decimal, then the variable should be of the 'float' type.

Type of a variable can change. One thing that can be confusing for people coming from a 'strongly typed' language is that a variable can change type while it is being used. Every time an assignment is made to a variable, Python re-evaluates the type and updates it. That can be confusing because 'abc' might be an integer in one part of the code, and then change type to a float or a string in another part of the code.

[*] Multiple assignments. Python enables multiple assignments on a single line, either through transitive (all variables are given the same initial value) or distributive (each variable on the left side of the equals sign is associated with the value in the same position on the right side).

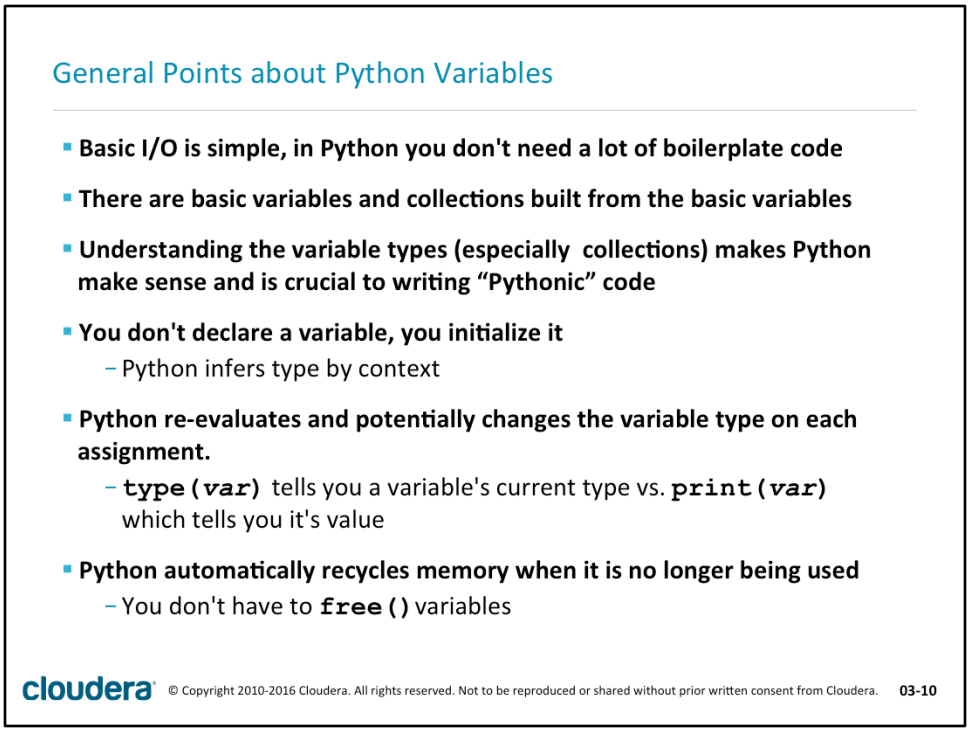


[*] Built-in memory management. Another difference between Python and other languages (especially lower-level languages like C) that do not provide memory management. You have to 'free' any memory that has been allocated. One feature that low-level coders often have to implement is a memory management system to guard against consuming memory. They need to free memory when it is not being used. When this doesn't work it can cause a 'memory leak' which is another painful bug to troubleshoot.

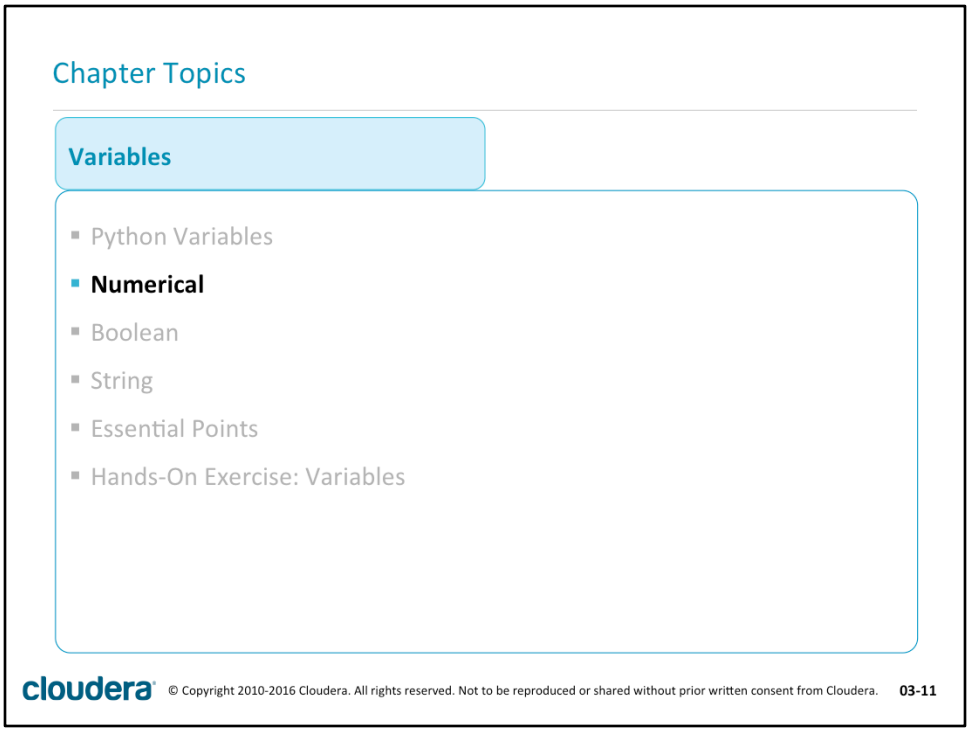
[*] Memory management is based on scope and reference count. Python tracks variable use by scope, and when execution passes out of the scope for a variable, its reference count is decremented. When the reference count reaches zero, the variable is placed on a queue for recycling at a convenient time.

Instructor value-add. Additional detail if needed.

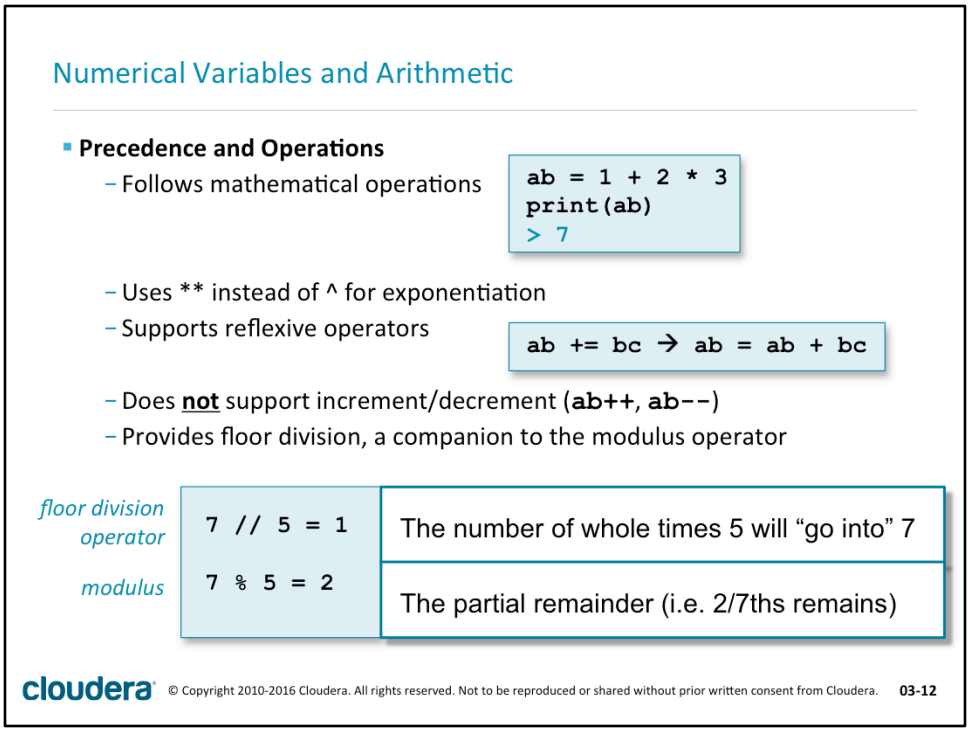
- When a variable is initiated, Python creates a Reference Counter for it
- When a variable is used, the Reference Counter is incremented
- When the variable is no longer being used (falls out of scope) the Reference Counter is decremented
- When it reaches zero, the variable becomes eligible for recycling
- The space is not immediately recovered, but is placed in queue and Python recycles the space on its own schedule



Note: Just reviewing key points of the overview before diving into the details with each variable type.



First are numbers… integers and floats.



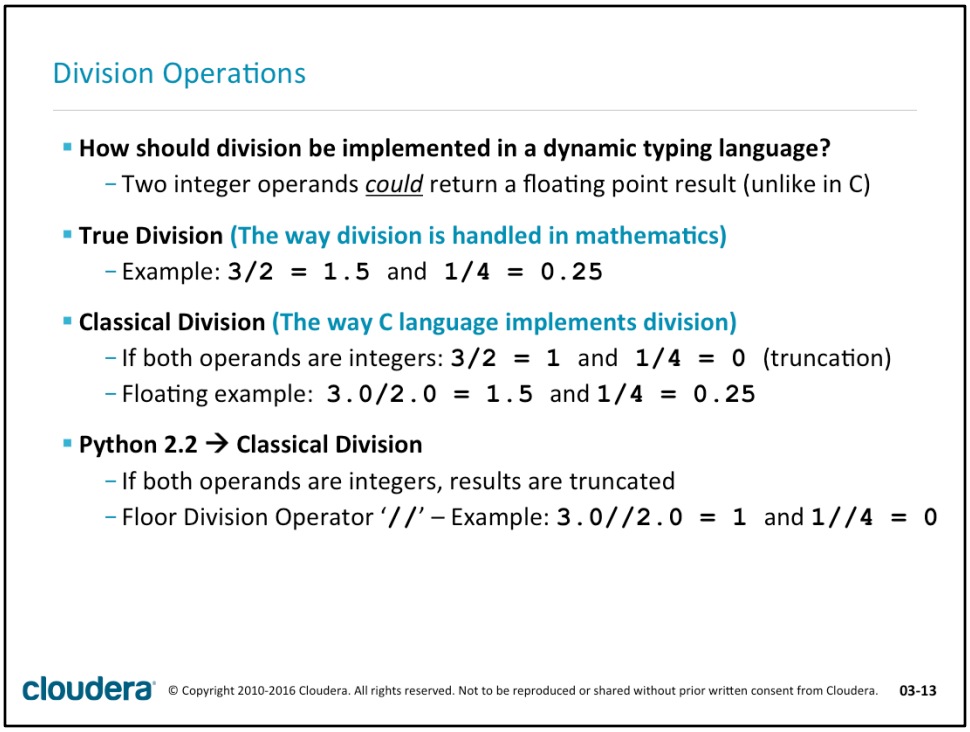
Pretty much as you'd expect…

There are a few differences from other languages.

[*] Python uses ** instead of ^ for exponentiation. The ^ is a logical operator between Set objects.

[*] Also, for programmers accustomed to increment and decrement (x++ and x--), Python doesn't support this. Instead, use the reflexive x += 1

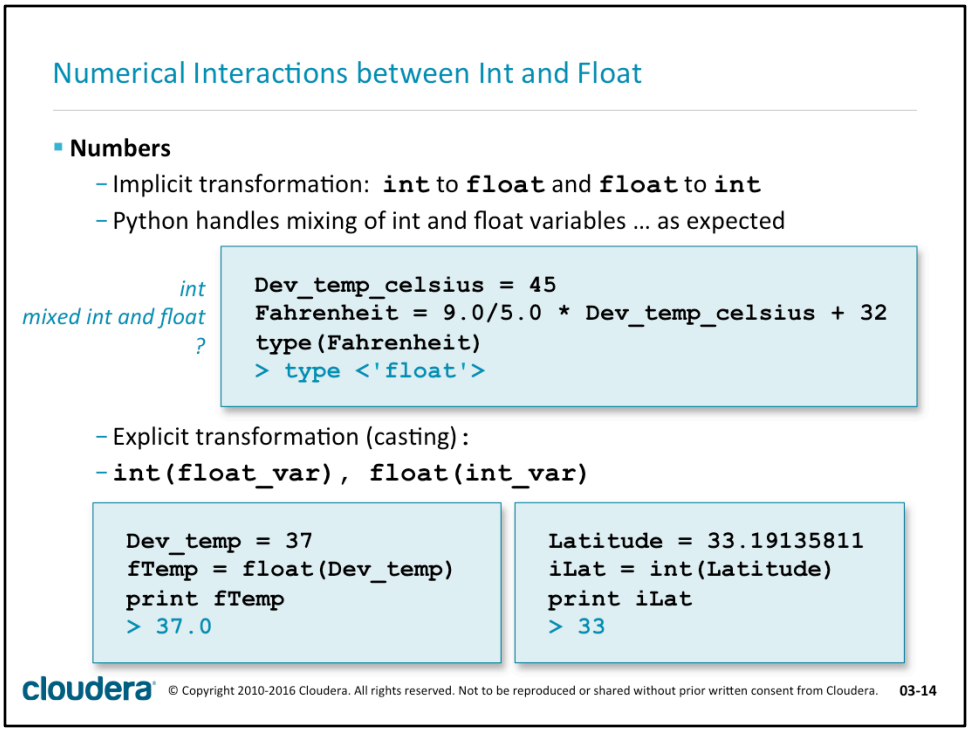
[*] Python provides a floor division operator that is the companion of modulus. Details about floor division on the next slide.



Yes, the floor division operator looks like a C-style comment

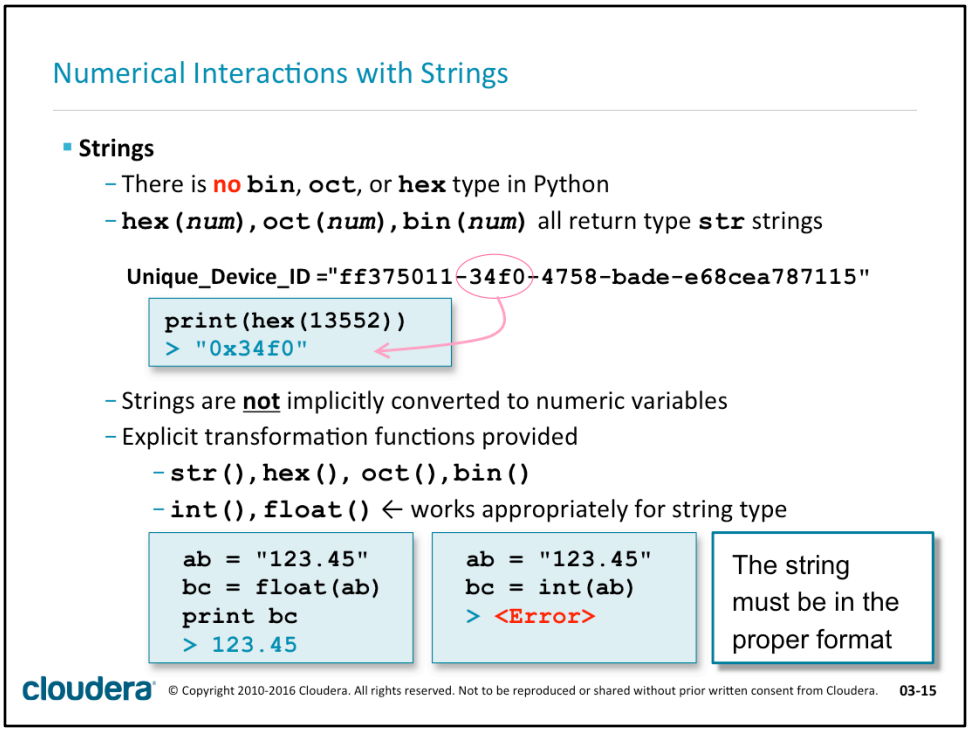
The issue is that in a dynamic typing language like Python, the results of two integer operands can be a float rather than an integer. By breaking with classical division (as implemented in 'C') which preserves the integrity of the type in the result, Python breaks code-level compatibility with legacy code in favor of being more "math-like". The change is introduced in phases. 2.2 introduced the "Floor Division" operator and warned developers who expected the integer/truncation behavior to start migrating their code to use '//'. In 3.0 the transition was completed by changing the default behavior of division, so that if both operands are integers, and the result is not whole, the result will be passed back as a floating point variable rather than truncating it to fit in an integer format.

This debate has led to much consternation. However, as we know from Scala's design and uses with Big Data, fidelity to math will win out.



[*] Dynamic typing – context interpretation.
Basically, Python figures out if there is a float involved and makes the resulting variable assignment into a float.

Dev_temp_celsius → the mobile device temperature in the Loudacre data.
Overheating may be one reason that certain models restart.
If the GPS is enabled, the device reports Latitude and Longitude.



[*] There are no low-level binary, octal, or hexadecimal variable types. These functions convert decimal to their representation as strings. So it is basically for formatting output. You CAN specify hex and binary values as input such as using 0x… and Python will parse this input and convert it to a decimal and store it as an integer.

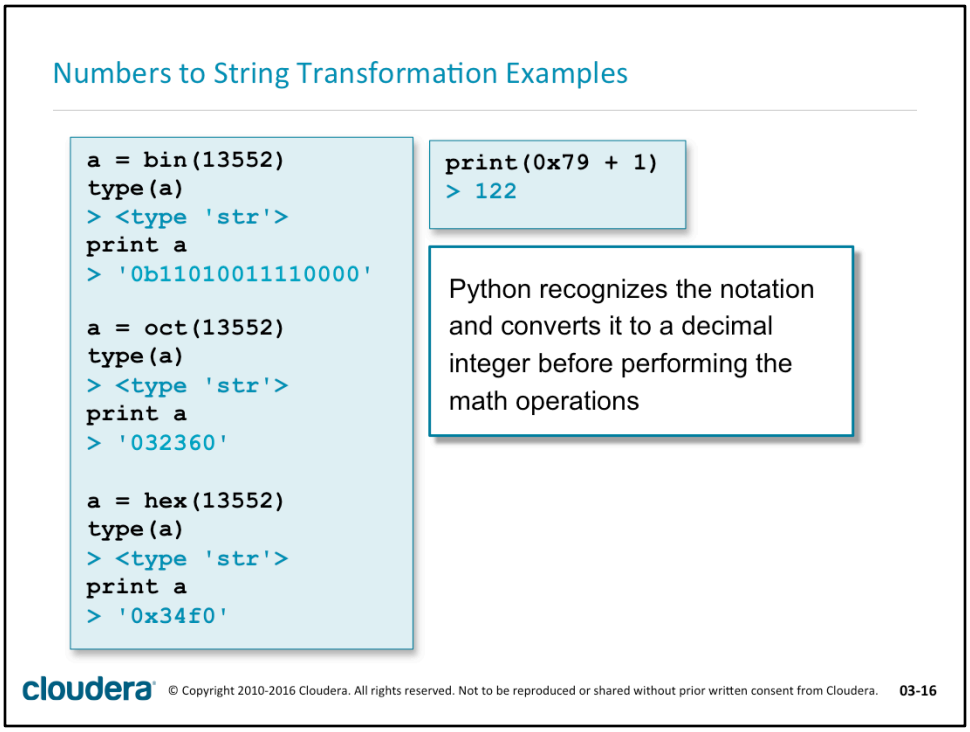
The Unique_Device_ID is a string composed of hexadecimal sequences in the Loudacre data.

[*] Also, a string containing an integer or floating point number can be converted into an integer or float variable. The casting functions int() and float() use polymorphism to properly process strings.

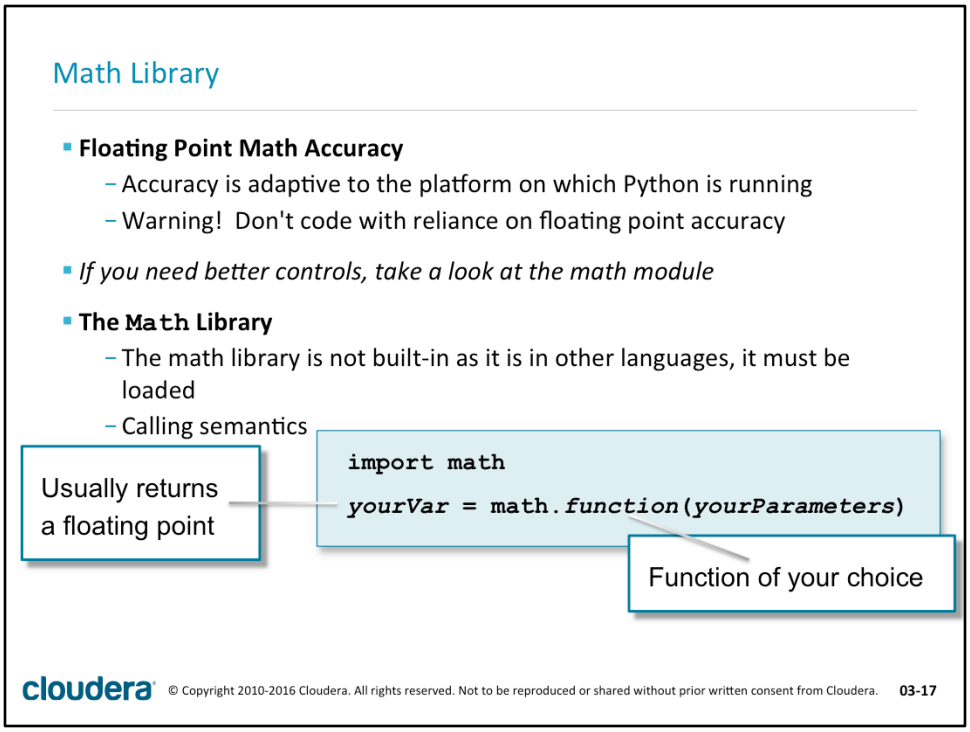
Note that in the code that generates an error in the slide, int(ab) could optionally be casted to a float first, and then there would be no problem.

This is a good point to discuss "Duck Typing" as an instructor value-add. We don't specifically describe Duck Typing in the slides. But you can explain that Python knows whether it is looking at a string or an integer, and calls the appropriate version of the function to cast the source.

This concept is covered in the final chapter on object oriented programming as polymorphism.



[*] The assumption in Python is that decimals are your 'calculating' system, and that binary, octal, and hexadecimal are primarily methods to display the results of calculations.



[*] Math functions that are commonly 'built in' to other languages are a separate module in Python. You have to 'import math' to enable those functions. We are not going into detail here, but making the learner aware, as they are likely going to need math for Data Analytics.

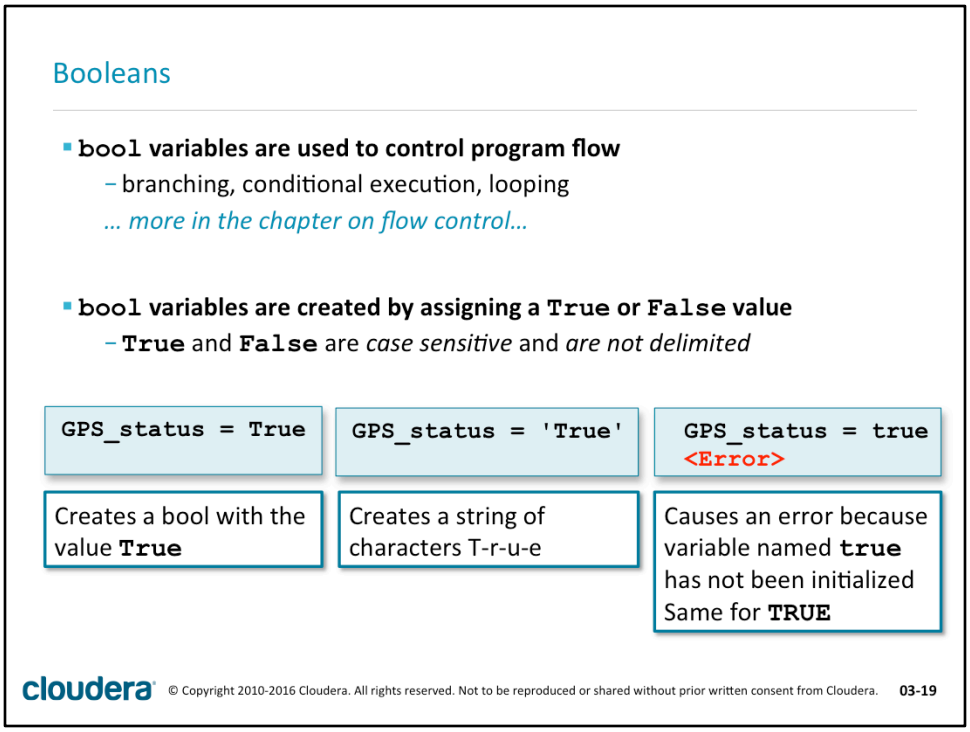
Note: One reason we are mentioning this at all is that math functions are native in many other programming languages, and we don't want students 'hitting the wall' if they expect sin() or sqrt() to 'just work'.

There is more about math and importing of modules later in the class.

Chapter Topics

Variables

- Python Variables
- Numerical
- **Boolean**
- String
- Essential Points
- Hands-On Exercise: Variables

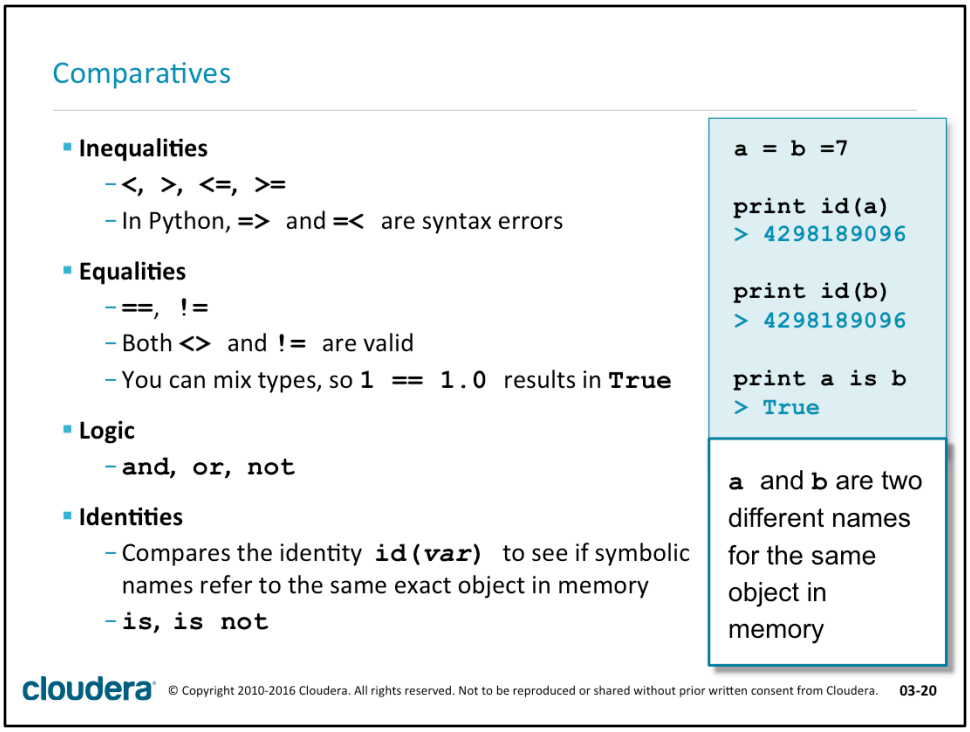


[*] Main point here are that Boolean values in Python are **`True`** and **`False`** (with initial caps) and that '**`true`**' and '**`TRUE`**' are not boolean values, but undefined variable names in Python.

GPS_status is a boolean value set in the Loudacre data, indicating whether GPS is enabled (**`True`**) or disabled (**`False`**) on the mobile device.

Note: Students coming from other languages may be habituated to pre-processor directives in which textual variations have been set to expand to the logical value – meaning that even if the language is case sensitive, the programmer may be habituated that it doesn't make a difference.

Booleans are used for Flow Control in Python. We have a whole chapter on that later in the class. So we are just mentioning it here for foreshadowing.



The Comparatives are pretty much as you'd expect.

A few peculiarities…
[*] Python allows '<>' instead of '!=' and also the English phrase 'is not'

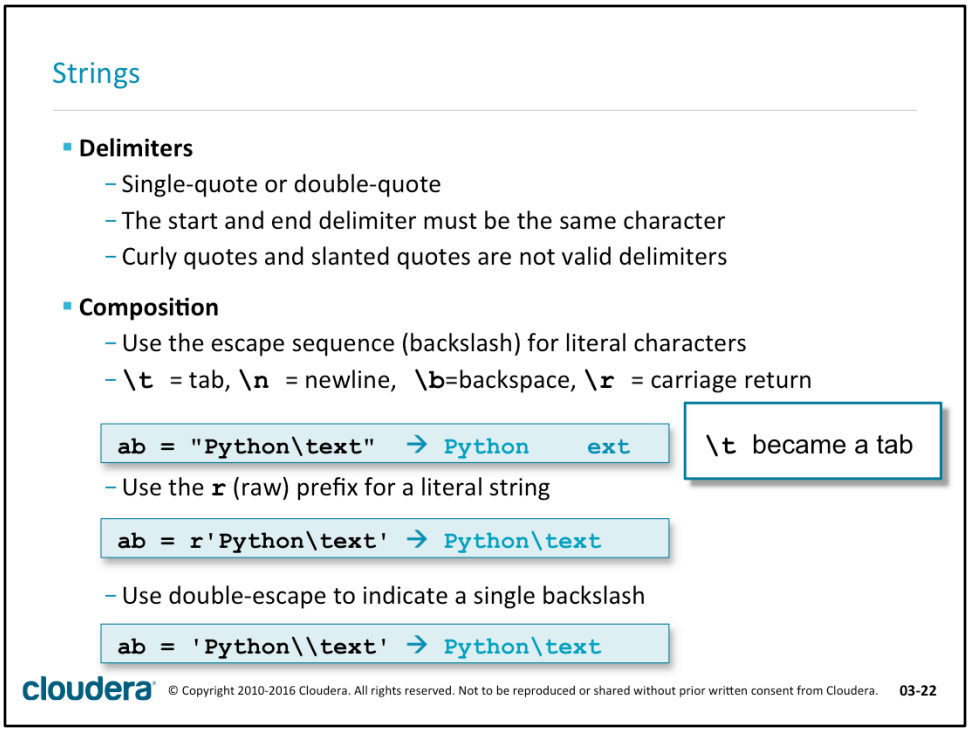
Note: You should mention that the syntax shown is all that's supported. Students with other language experience might expect to see text alternatives such as _IS_EQUAL_TO_ , or .GT. or' greater_than' that are used in some contexts to make code more readable.

These operators are overloaded to support Duck Typing for compound variables and mixed type operations. For example, you can test if int < string
(Integers, no matter how large a number, are always less than any string).

Chapter Topics

Variables

- Python Variables
- Numerical
- Boolean
- **String**
- Essential Points
- Hands-On Exercise: Variables



[*] You can use single quotes or double quotes to denote a string.

Python doesn't have a char type, so there is no need to distinguish between a character literal ' ' and a string literal " ".

[*] Escape sequences in strings.
The escape sequence (\...) within a string is – as you'd expect. it is the usual standard escape sequences from formatted strings.

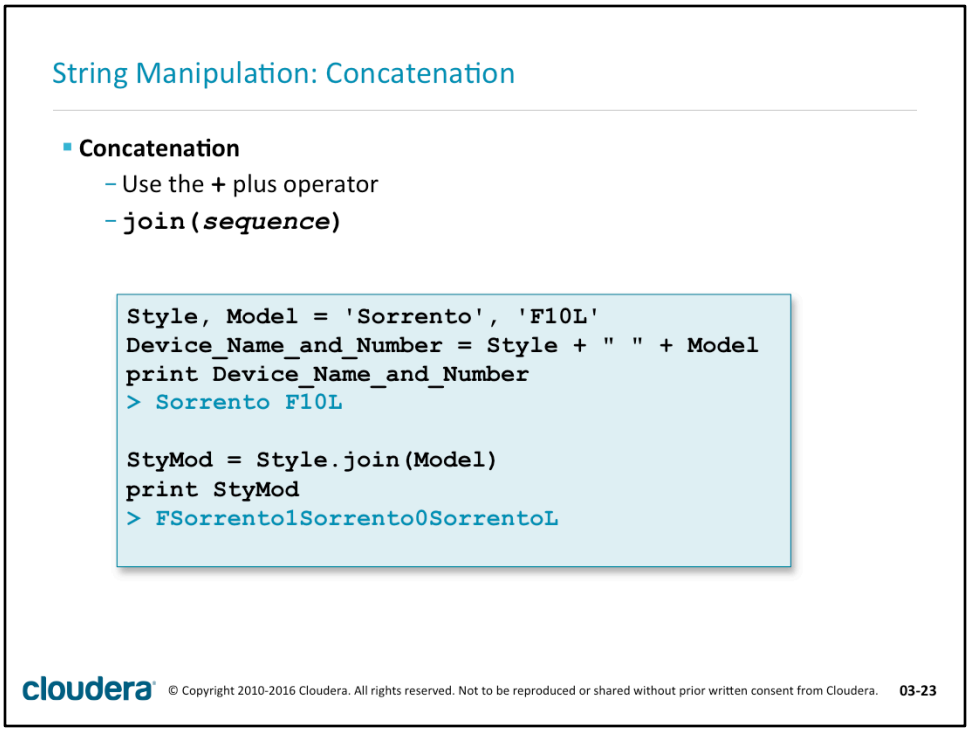
You can disable a single escape sequence with another slash, so '\\' → '\'

[*] One peculiarity of Python is that embedded single or double quotes must be escaped.

Example: "He said \'stop\'!" → He said 'stop'!

Note: Students might be familiar with triple delimiter such as "He said '''stop'''! which won't work in Python.

Another Python feature, is you can disable escape sequence processing by prefixing the string with the letter 'r'. (raw) This is a lot easier than trying to maintain double-escapes when generating HTML or other scripts where the backslash is a native character.



The '+' is used for both arithmetic and for string concatenation.

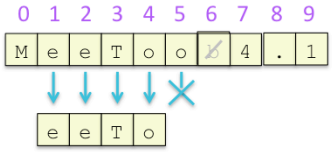
Device Make and Model number is a single field in the Loudacre data. We've identified the separate parts here to illustrate concatenation.

Join performs an interleaving or repetitive concatenation.

String Manipulation: Slices

- **Slice**
  - A substring consisting of an individual character is called a "slice"
  - ***string*[*offset*]**
  - The offset is zero-based, so the first character is **str[0]**
- **Slice Range**
  - A substring consisting of multiple characters
  - ***string*[*start*:*end*]**
  - *Warning! upper bound is non-inclusive*
    - The "end" character is **not** included in the substring

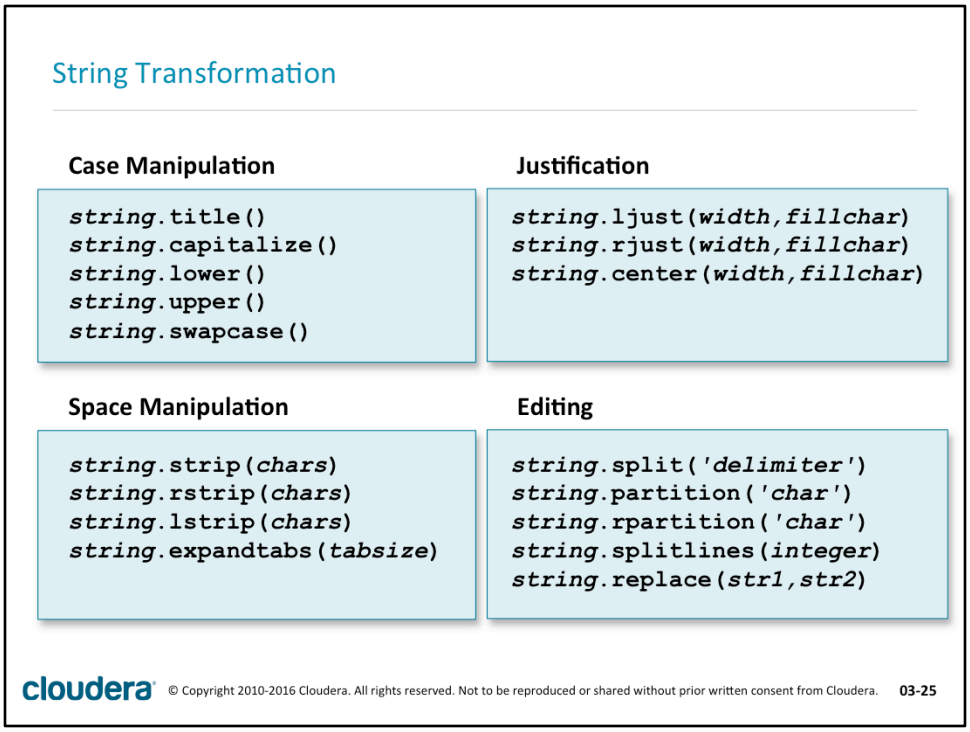
```
device = 'MeeToo 4.1'
print device[1:5]
> eeTo
```



[*] In Python the syntax name[x] is called a 'slice' and name[x:y] a 'slice range'. In other languages this is called an array or a subscripted element or just a range.

[*] The lower bound is inclusive in the substring, but the upper bound is not. The lack of symmetry can be confusing to those who have used other languages where the upper bound is inclusive.
The reason for non-inclusion of the upper bound makes more sense if you consider it as a mathematical expression. For an entity i; if you were to start numbering it is parts with '1', you'd mathematically describe the set of parts of i as '1' <= i < N+1. A natural consequence of simplifying this expression to start with zero would be: 0 <= i < N. And that means the 'nth' element is 'greater than' i and not included in the parts of i.

[*] One key awareness is that Python elevates the 'slice' syntax to the level of an operator. It can be used on any collection. We introduce it here in the context of a String. But it applies to all collection sequences such as lists and tuples. You're going to see 'slice' return later in this chapter, so make sure you introduce and enforce proper Python terminology to set the students up for understanding later.



This is just a light overview of the kinds of string methods that are supported in Python.

You don't need to cover them in detail here.

[*] We are going to use split() in the hands on exercises in this course. Split is used to convert delimited strings to lists. And this method is used frequently in our developer courses.

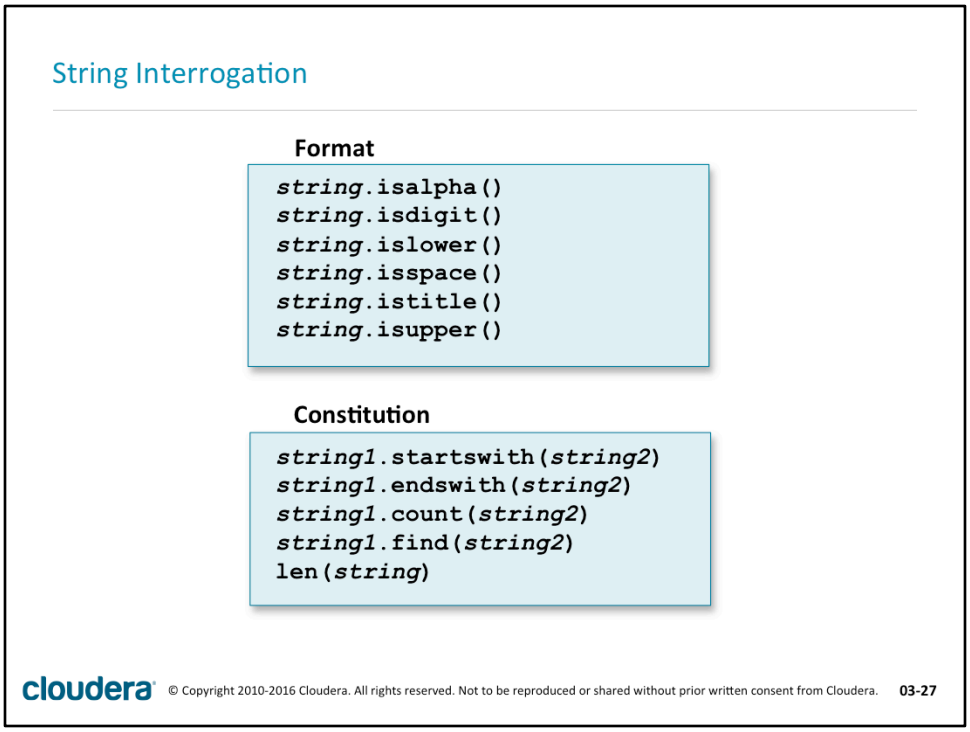
Note:
We need collections (lists) to demonstrate how 'split' parses a delimited string into separate elements. So we will cover this in detail in the next chapter, not here.

Chaining Example

- Methods are called from the object using the dot operator, as in most object oriented programming languages
- You can *chain* methods in Python. In the example shown, the results of the `title()` method are passed to the `swapcase()` method

```
device = "titanic 2300"
> titanic 2300
device.title()
> 'Titanic 2300'
device.swapcase()
> 'TITANIC 2300'
device.swapcase().title()
> 'Titanic 2300'
```


[*] You can endlessly chain methods in Python. The methods are processed from right to left and the results of each method become the source for the subsequent method.



Ways to find information about strings and the contents of strings.

Chapter Topics

Variables

- Python Variables
- Numerical
- Boolean
- String
- **Essential Points**
- Hands-On Exercise: Variables

Essential Points

- **Python does a lot of things automatically for you with variables**
  - That makes the code cleaner and easier to read
  - But you need to *know what it is doing for you*, or it is easy to get confused – *one reason we are spending so much time on variables*
  - The basic types are integer, float, boolean, and string
- **Int and Float behave as you'd expect**
  - math is an "add in," so math functions don't work by default
- **Boolean**
  - **`True`** and **`False`**
- **String**
  - Slice ***`str[x], str[start:end]`*** ← "end" is not inclusive

Just reviewing key points.

Chapter Topics

Variables

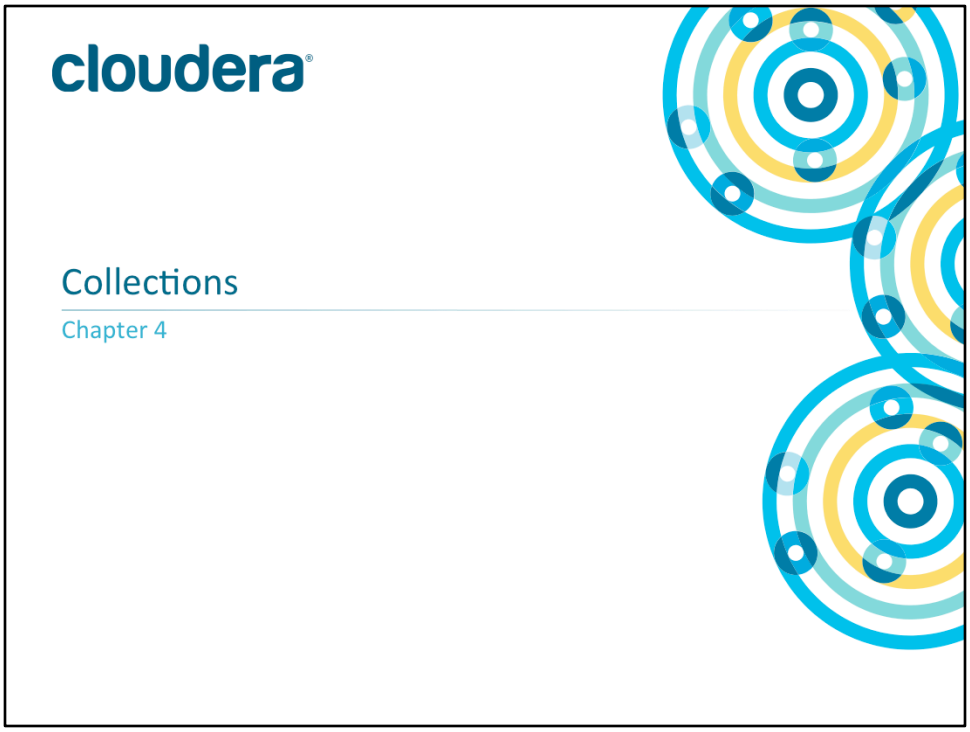
- Python Variables
- Numerical
- Boolean
- String
- Essential Points
- **Hands-On Exercise: Variables**

Hands-On Exercise: Variables

- **Interactive Exploration**
  - Explore Numerical variables
  - Operators, Operators, Casting, Math Functions, Booleans
- **Program**
  - Write a program to extract fields from a single data record

Interactive Exploration
- Numerical variables
- Operators and reflexive operators
- Casting
  - Integer and Float
  - Binary, Octal, and Hexadecimal
- Advanced math functions
- Booleans and comparators

Program
- Write a program to extract fields from a single data record
- Sample record string is in **`sample_one_record.txt`**
- String methods, **`find()`**, **`upper()`**, Slices
- Boolean



In this chapter we will explore compound Data Structures called 'Collections'.

The data structures supported in a language have a profound influence on the general approach to problem-solving and the way in which programs are written.

Course Chapters

- Introduction
- Introduction to Python
- Variables
- **Collections**
- Flow Control
- Program Structure
- Working with Libraries
- Conclusion

Collections

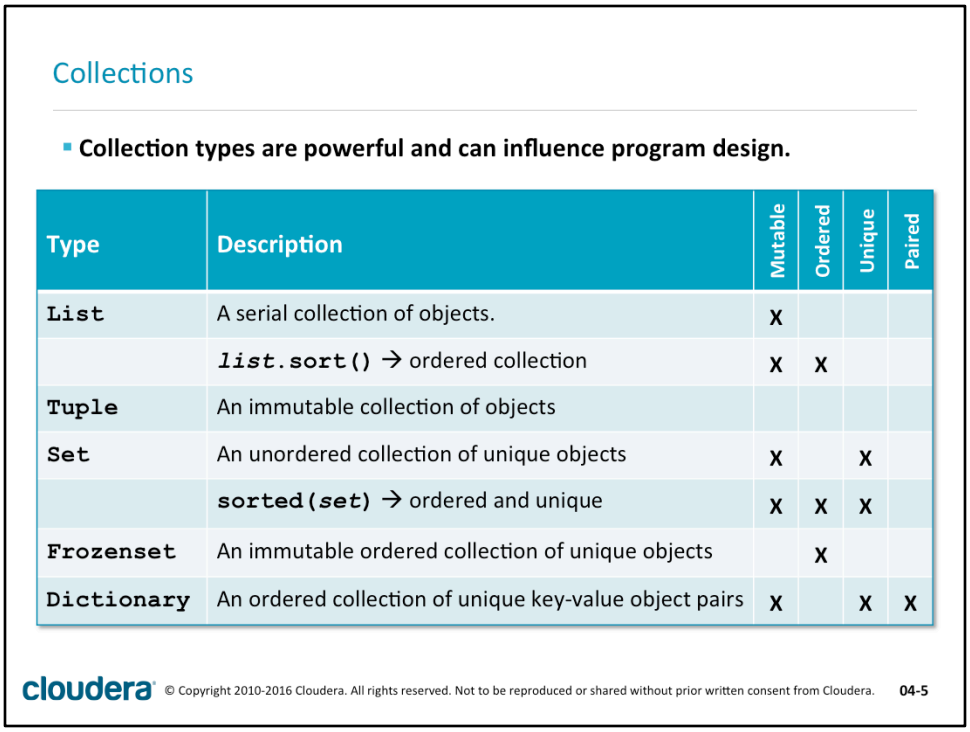
In this chapter you will learn

- **Essential information about Python's powerful collection types**
- **What are the different properties of each collection type, and suggestions about when to use one type or another**
- **How to create, manage, interrogate, and update collections**

Chapter Topics

Collections

- **Lists**
- Tuples
- Sets
- Dictionaries
- Essential Points
- Hands-On Exercise: Collections



Different sources use the terms "Collections", "Sequences" and "Sequence Variables" interchangeably. Sometimes they are referred to as "Lists" after the first and most common collection type.

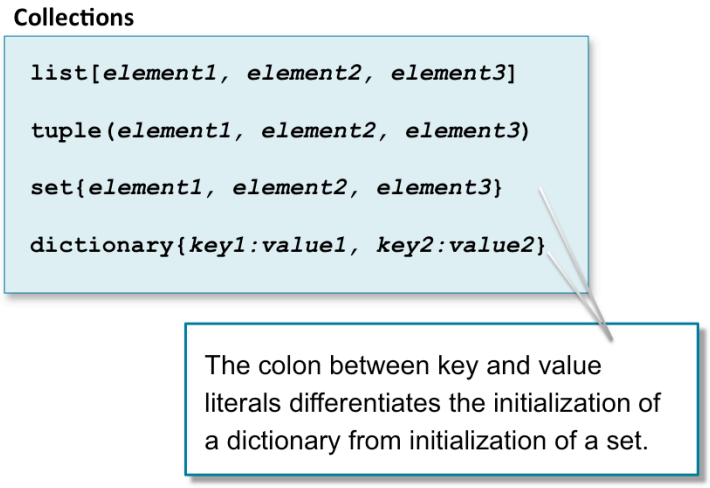
A lot of the power of Python is due to these extremely flexible and powerful collections.

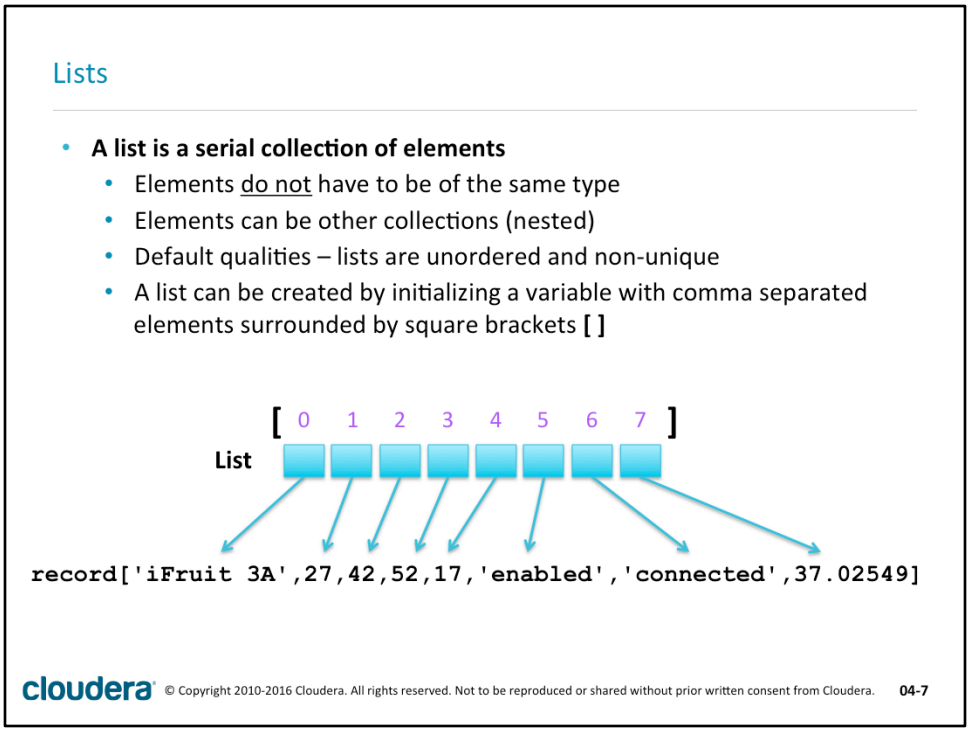
A useful way to categorize variables is by (1) Mutable or Immutable, (2) Ordered or 'as is', (3) Unique or Repeatable elements, and (4) single values or pairs of values (key-value pairs).

[*] A list is in natural 'as is' order, but can be sorted to create order within it. A list can also be ordered without being unique, by using the list.sort() method.

[*] A Tuple is basically an immutable list.
[*] A Set is a list that is unique. It can be ordered (ascending order) with the sort method.
[*] A Frozenset is basically an immutable Set.
[*] And a Dictionary is a Set with Key Value pairs rather than individual values.

Collections Syntax

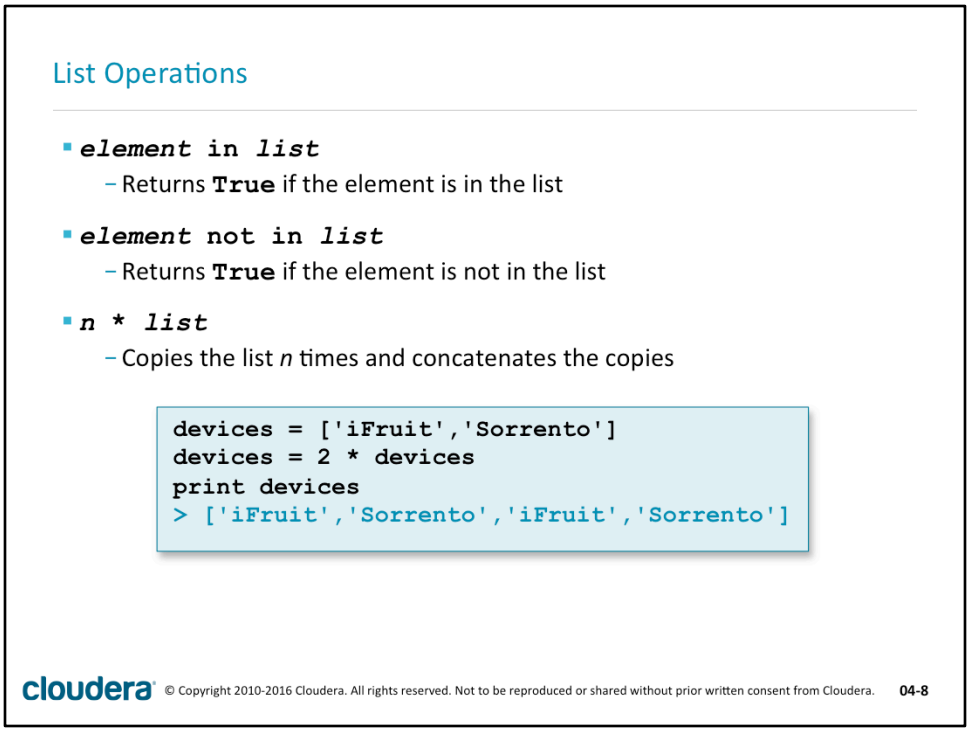




[*] A list is a sequence of values separated by commas and surrounded by square brackets.

[*] elements in lists don't have to be of the same type. But it is more common to see lists of a single type and Tuples of multiple types.

Note:
The illustrations only show a list as being composed of basic variables, because this is all we've covered so far in the course. In Python, the collections are collections of objects. So you can actually use a list as an element of a list (nested structure) as well as other collections and even other types of objects, such as functions.



[*] Operators are overloaded and do what you'd expect.

Comparators (equalities and inequalities) are overloaded for lists. They will compare each element in the two lists, left-to-right, to evaluate.

Whiteboard:

```
list1 = [1,1,1]
list2 = [1,1,2]
list1 < list2   → True
```

List Modification

- ***list*.append(*element*)**
  - Appends an element after the final element in a list
- ***list*.insert(*offset, element*)**
  - Inserts an element into a list after the offset
- ***list*.remove(*element*)**
  - Removes the first occurrence of element from the list

```
models = ['Sorrento','iFruit','Titanic']
models.remove('iFruit')
print models
> ['Sorrento', 'Titanic']
```

- ***list*.pop()**
  - Returns the last element in the list and removes it

[*] list.append(element) vs. list.extend(list2)

list.append(element) adds one element to the end of the list.

Working with Multiple Lists

- ***list3 = list1 + list2***
  - Returns a new list containing the elements from **list1** and **list2**
  - Does not change the original lists
- ***list1*.extend(*list2*)**
- ***list1 += list2***
- ***list1 = list1 + list2***
  - Concatenates the elements from **list2** onto **list1**
  - Copies all elements in **list2** to the end of **list1**
  - Overwrites the contents of **list1**

[*] list.append(element) vs. list.extend(list2)

list.append(element) adds one element to the end of the list.

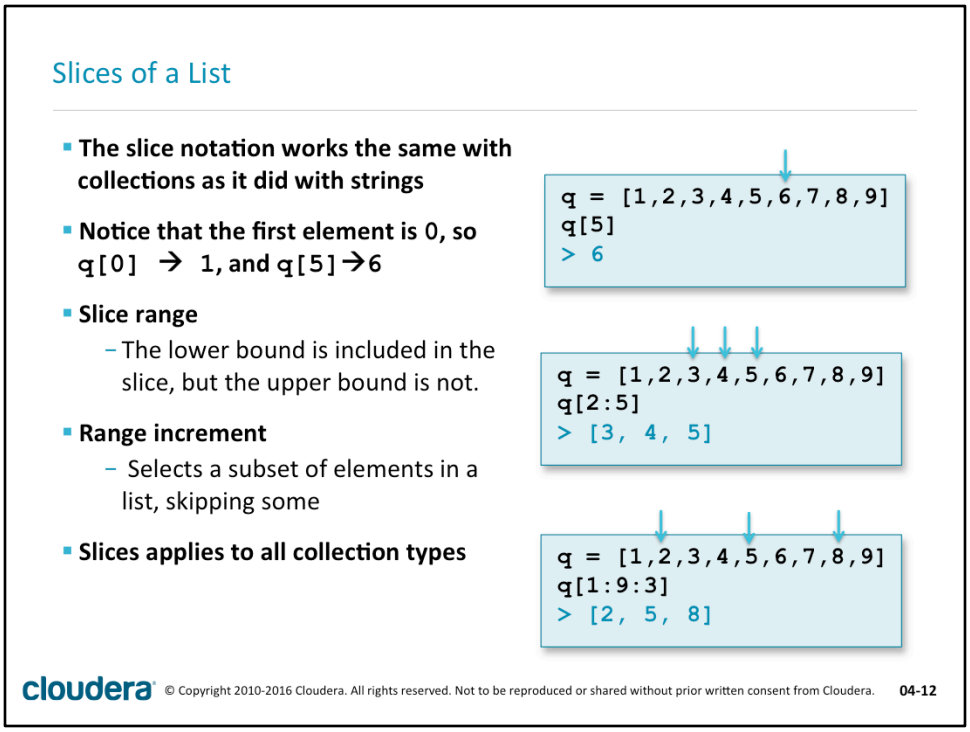
list.extend (list2) adds ALL of the elements in list2 to the end of list1. Note that it does this by copying. So if you later change the value of an element in list2, list1's copy is not affected.

`append()` Versus `extend()` for Collections

```
mods1 = ['Sorrento','iFruit','Titanic']
mods2 = ['Ronin','MeToo']

mods1.append(mods2)
print mods1
> ['Sorrento','iFruit','Titanic',['Ronin','MeToo']]
```

The last item is a list. This is a common mistake!

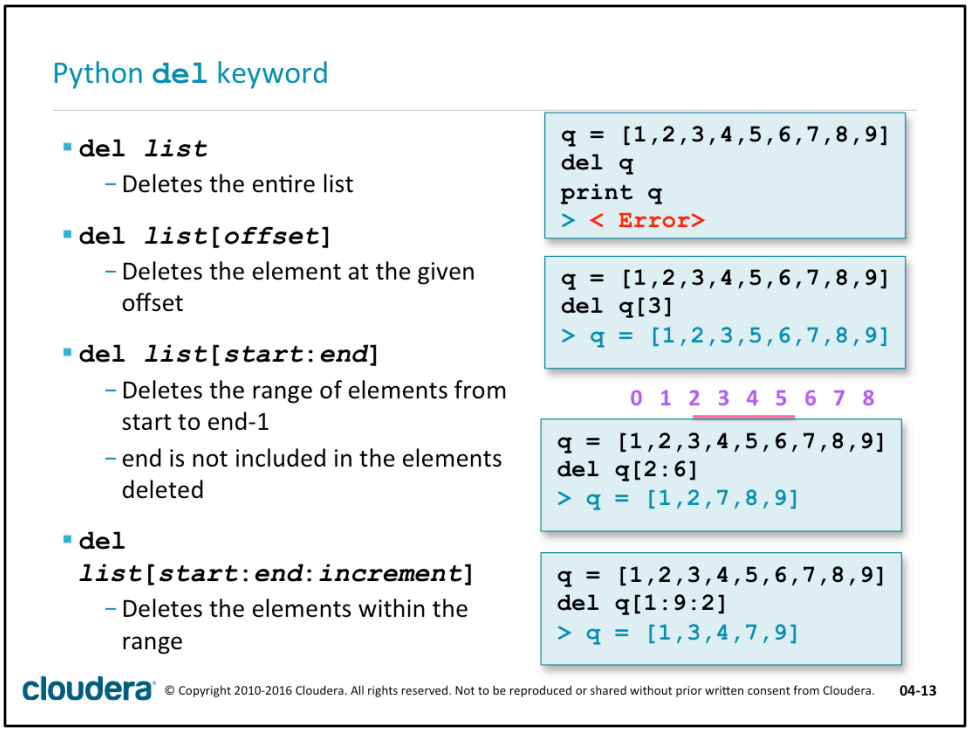
```
mods1 = ['Sorrento','iFruit','Titanic']
mods2 = ['Ronin','MeToo']

mods1.extend(mods2)
print mods1
> ['Sorrento','iFruit','Titanic','Ronin','MeToo']
```

[*] Slices work the same way in lists as we discussed in strings, effecting each element rather than each character.

[*] Slices can have a third element. This is the 'iterator'.

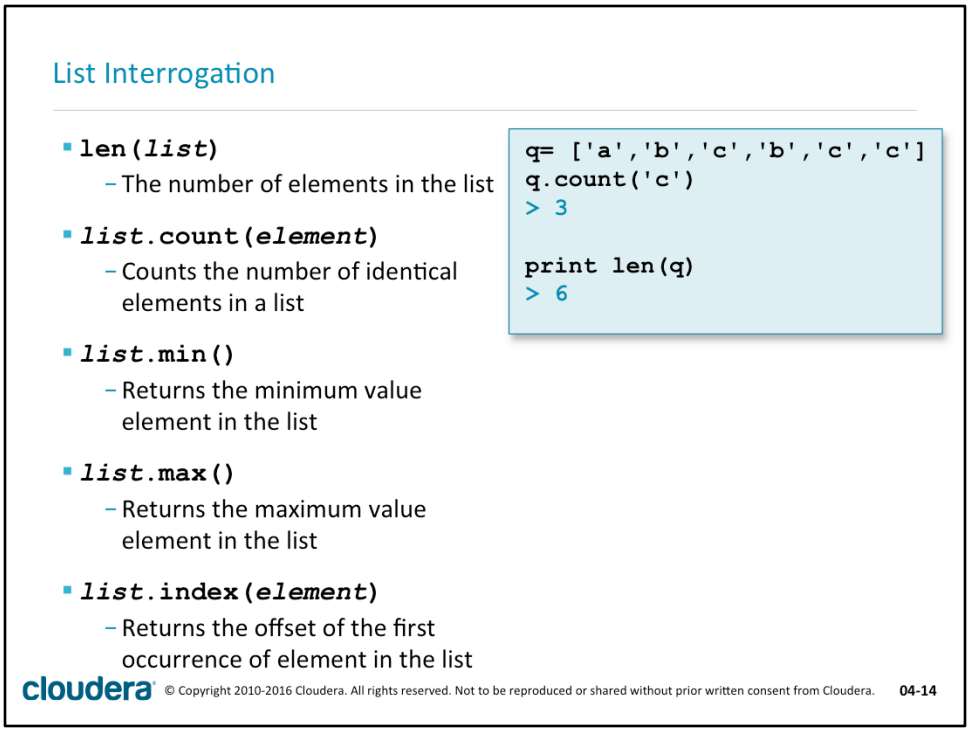
Note:
The iterator will be important later because it is often used with a collection to process only certain elements in a collection. In other languages you would use a variation of the "for" loop to iterate in this way. In Python it is more common and elegant to use "slices" to imply iteration instead of explicit loops.



[*] Note that 'del' is a Python keyword, not a method of the object or a function.

When you delete the entire list, it *undefines* the variable.

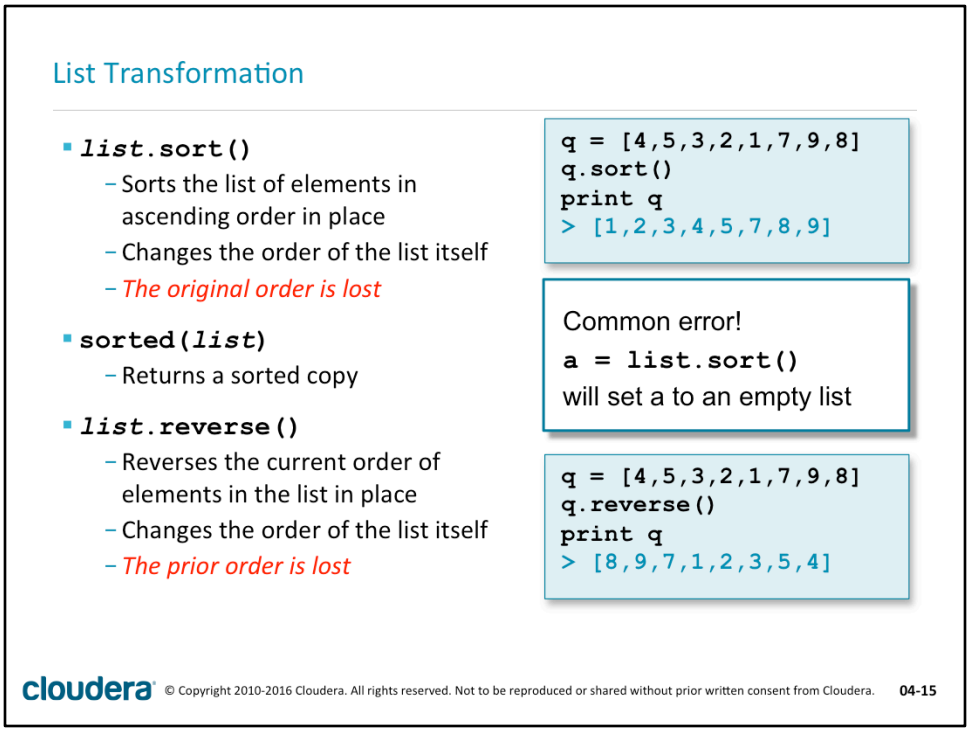
Note:
The del keyword queues an object to memory management for recycling. Some functions such as list.remove(element) are the same as del list.element



[*] len(*list*) versus *list*.count(element)

len() *length* returns the number of elements in the list.

list.count(element) searches the list for identical instances of 'element' and returns a count of the number of them in the list.

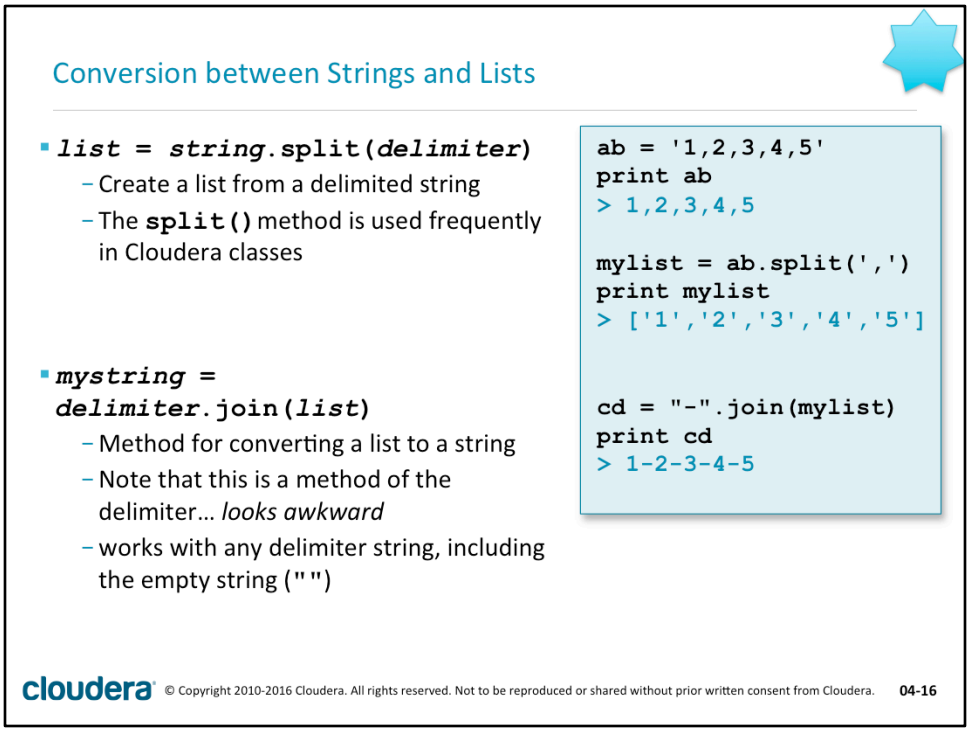


[*] list.sort() vs. list.reverse()

list.sort orders the list in ascending order. This is one of the few functions in Python that is destructive, in that the original order of the list is lost. It does not create a new list that is ordered and return that, it actually changes the order of the list in place.

list.reverse() changes the order of elements in the list, first to last. Note that it does not sort the list. So if you called reverse on the original order it will reverse the original order.

If you want to sort in descending order, you'd call list.sort() then list.reverse() or… list.reverse().sort()



[*] split

Shown – comma delimited. You can use anything in the parameter string as delimiter.

Note: Split is very common in our Developer classes because it is used to process delimited data and pass it to an RDD (Resilient Distributed Dataset).

[*] .join

The .join method is very awkward, because you actually have to call the delimiter as an object and then .join as the method. The intuitive structure seems like it should be…

list.join("delimiter") ← but this won't work
instead you need
"delimiter".join(list)

List File I/O

- **Use the `readlines()` method**
  - Reads all the lines in the file to end-of-file (EOF)
  - Returns each line as a separate element in a list

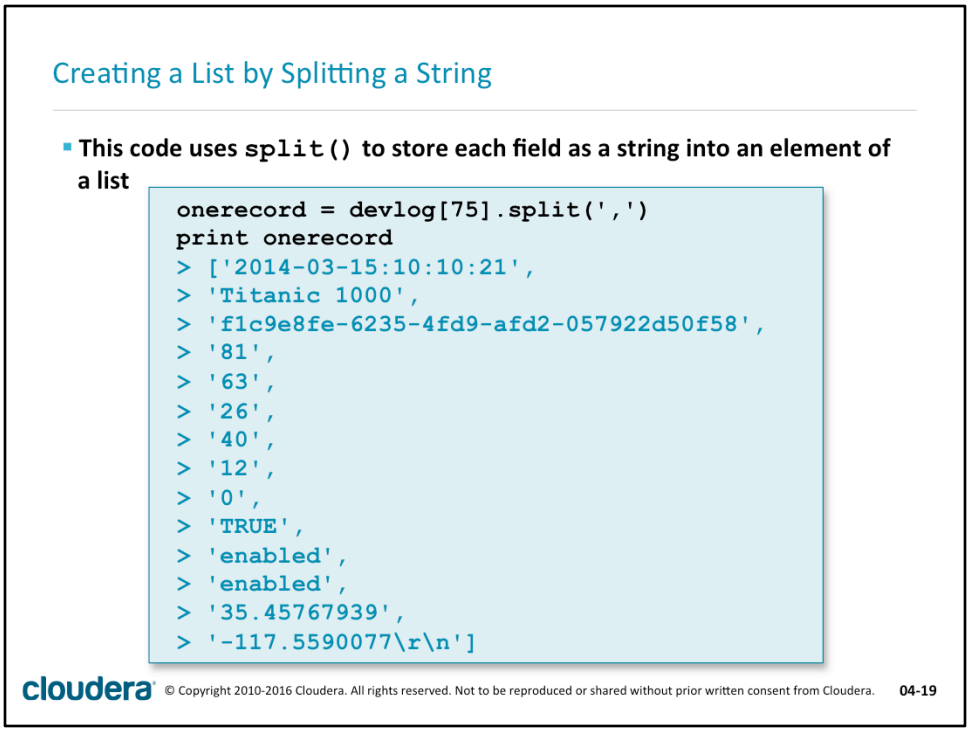
```
lines = [ ]

file = open('example.txt','r')
lines = file.readlines()
file.close()
print lines
```


readline() reads to the EOL character at the end of the line.
readlines() –with and 's'– reads the entire file until EOF and places each line into an element in a list.

Example Code

- This code reads the entire `loudacre.log` file
- Each line is stored in one element of a list

```
devlog = [ ]
file = open('loudacre.log','r')
devlog = file.readlines()
file.close()
print devlog[75]

> '2014-03-15:10:10:21, Titanic 1000,
f1c9e8fe-6235-4fd9-afd2-057922d50f58, 81, 63, 26, 40,
12, 0, TRUE, enabled, enabled, 35.45767939,
-117.5590077'
```

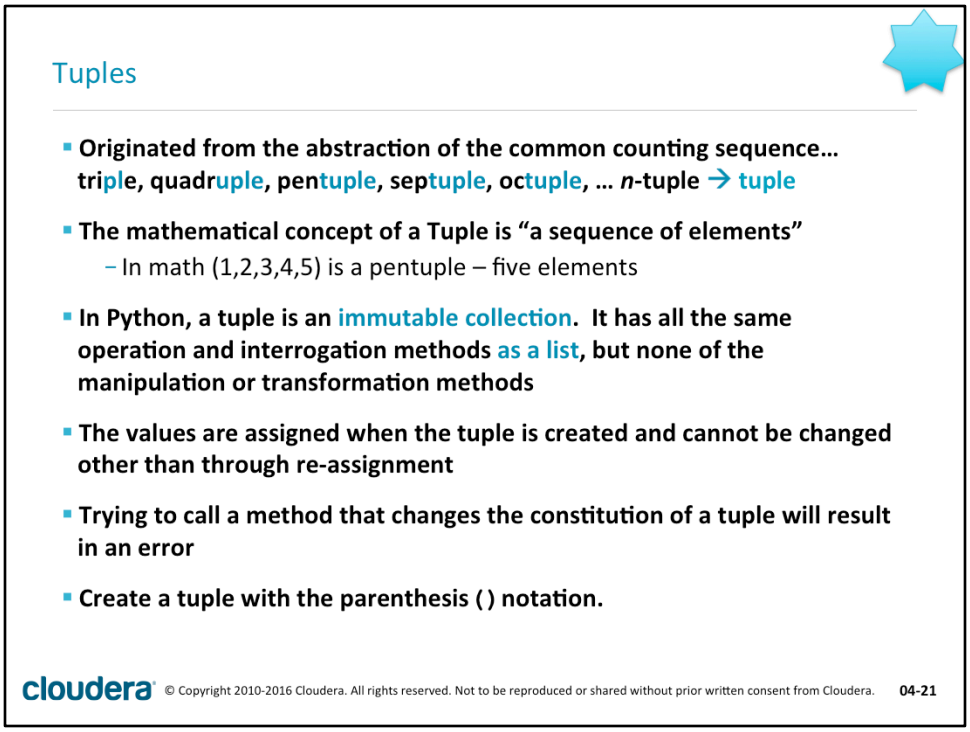



This example uses a record (at element index 75) read from the code on the previous slide, and calls the split method on that string to split it using a comma as a delimiter. This has the effect of producing a new list, where each field in that record is an element.

Chapter Topics

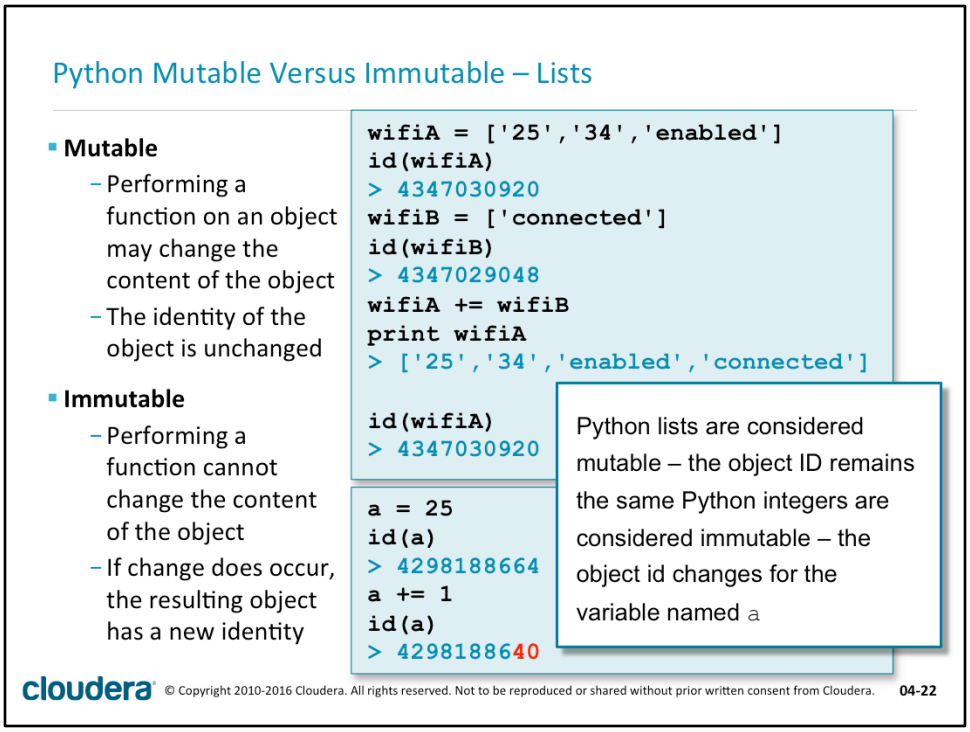
Collections

- Lists
- **Tuples**
- Sets
- Dictionaries
- Essential Points
- Hands-On Exercise: Collections



Tuple
[*] A tuple in Python is an immutable list. This is accomplished by not defining any methods for tuples that would change or add information. The tuple is basically fixed after it is initialized.
[*] You create a tuple the same as you would a list, but use parenthesis instead of square brackets. Initialization is the only time you can assign a value to a tuple. All of the methods that change, add to, remove from, or order a list are missing for the tuple.

Note:
The concept of a tuple is one specific thing in mathematics, but it has been interpreted and implemented differently in programming languages and systems. People may have different expectations of the tuple depending on where they learned the concept. For some students, a tuple is a pair of values, and when you give the first value in the pair, the second one is returned. This key-value pair in Python is called a Dictionary. So if they ask about that functionality, tell them it is covered a bit later. For other students, a tuple is not a key-value, but still a pair of values. This is easily implemented in Python as the even and odd elements in the tuple. So tuple[0] and tuple[1] would be the first pair, and tuple[2] and tuple[3] would be the second pair. Then you iterate over first and second elements with a slice using an increment value such as tuple[0:n:2] or tuple[1:n:2].

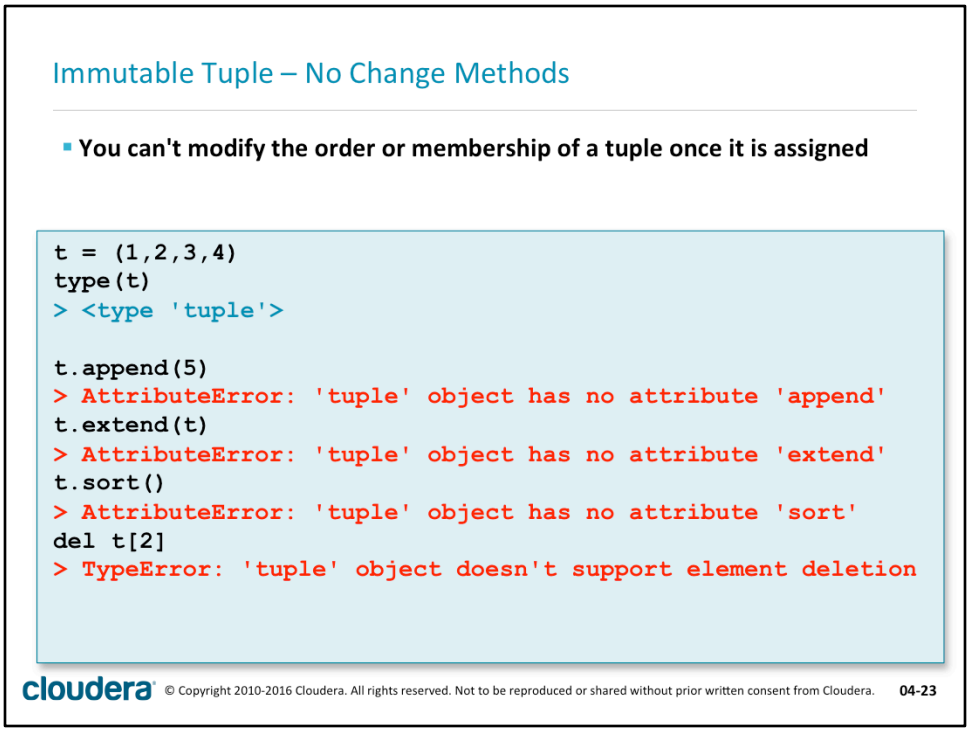


Python has a peculiar notion and implementation of mutable versus immutable.

In the above examples, a list is concatenated using the reflexive "+=" operator, and an integer it incremented using the same operator.
Using the Python id() function to display the object id reveals what's going on under the covers. In the case of the list, the change happens in place and the original object is changed.
By displaying the id before and after, we can see that it is still the same object, ending in '20'. Changing an object in place means that it is 'mutable' according to Python.

In the second example, a very similar operation is performed on an integer. However, the original object (id ending in '64') is lost and a new object ending in id '40' is created.
For this reason, Python considers the integer to be immutable, even though many other interpretations and implementation in other languages would consider it mutable since the result is that variable 'a' has a new value.

Note: the Python id() function prints out the memory address associated with a symbol. In the example in the second block, the id(a) reveals that the a += 1 resulted in allocation of a new variable named 'a' at a different location in memory.

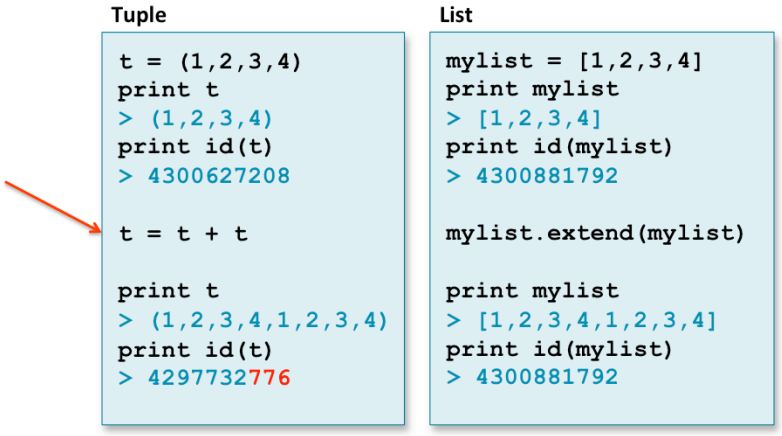


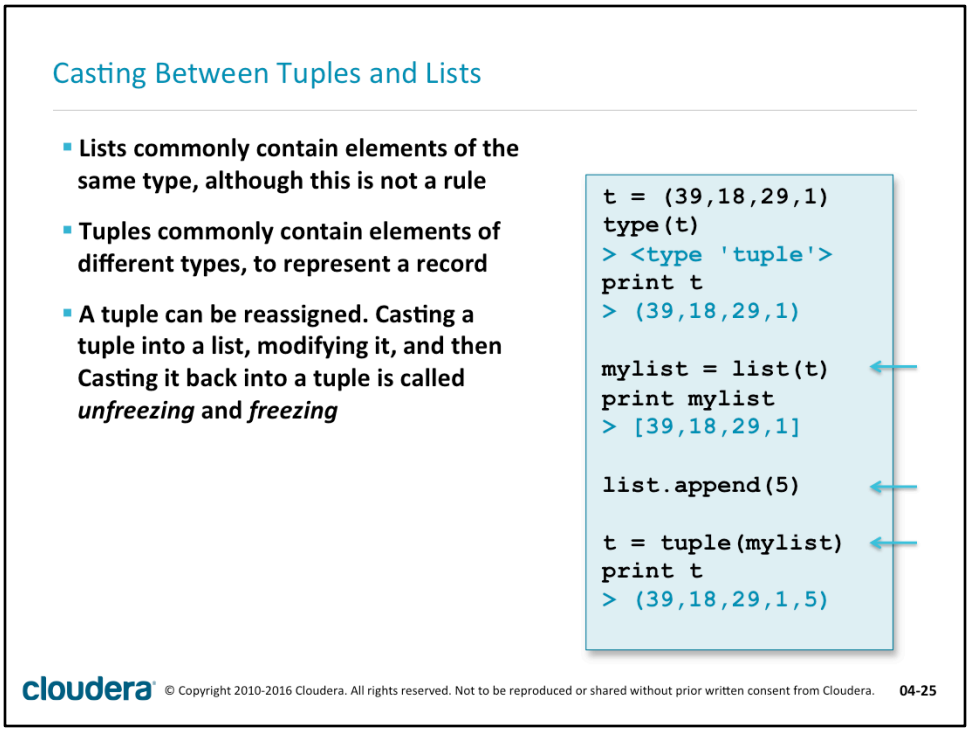
[*] Tuples are immutable after initialization.

Note:
The error message is shown above as it appears in the Python shell. Note that it says 'attribute' is not defined. it is quite clear from the calling semantics with the parenthesis for passing parameters, that these are not attributes, but methods. The error is in error.

Immutable Tuple

- You can re-assign the value of an entire tuple, but it creates a new object





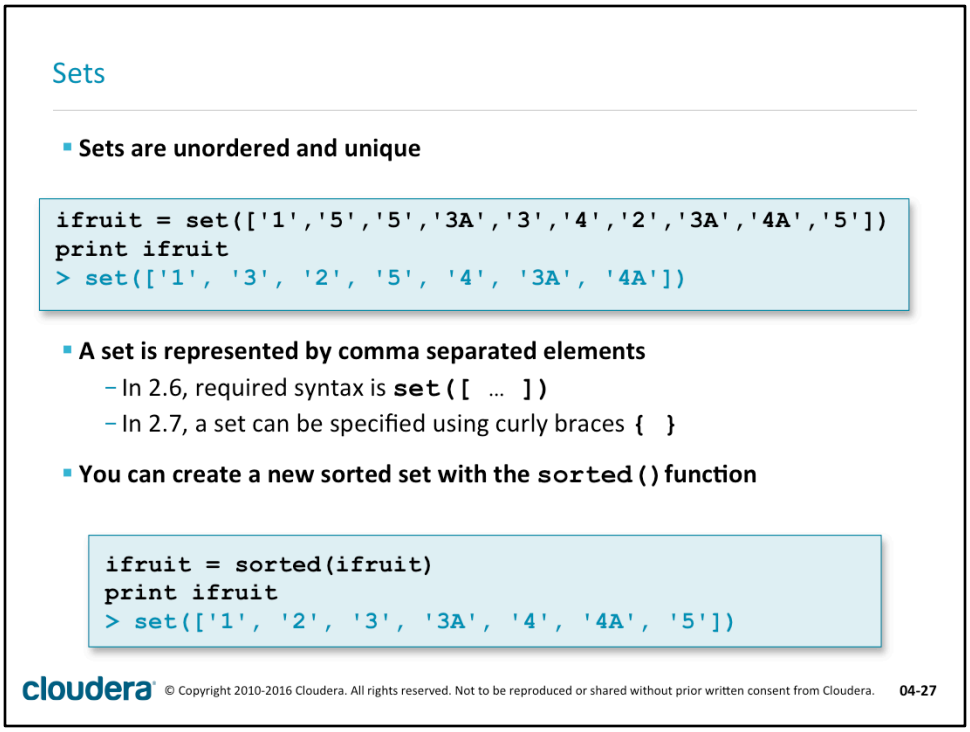
[*] You can 'cast' a tuple to a list and a list to a tuple to 'unfreeze' and 'refreeze'.

Note:
Note that when you cast the list back to the tuple, it is actually a new location in memory. So in the example above, the first 't' is a tuple. Then it is converted to list lst, then converted to another 't' tuple. The new tuple replaces the old one, and the old one is queued for memory recycling.

Chapter Topics

Collections

- Lists
- Tuples
- **Sets**
- Dictionaries
- Essential Points
- Hands-On Exercise: Collections



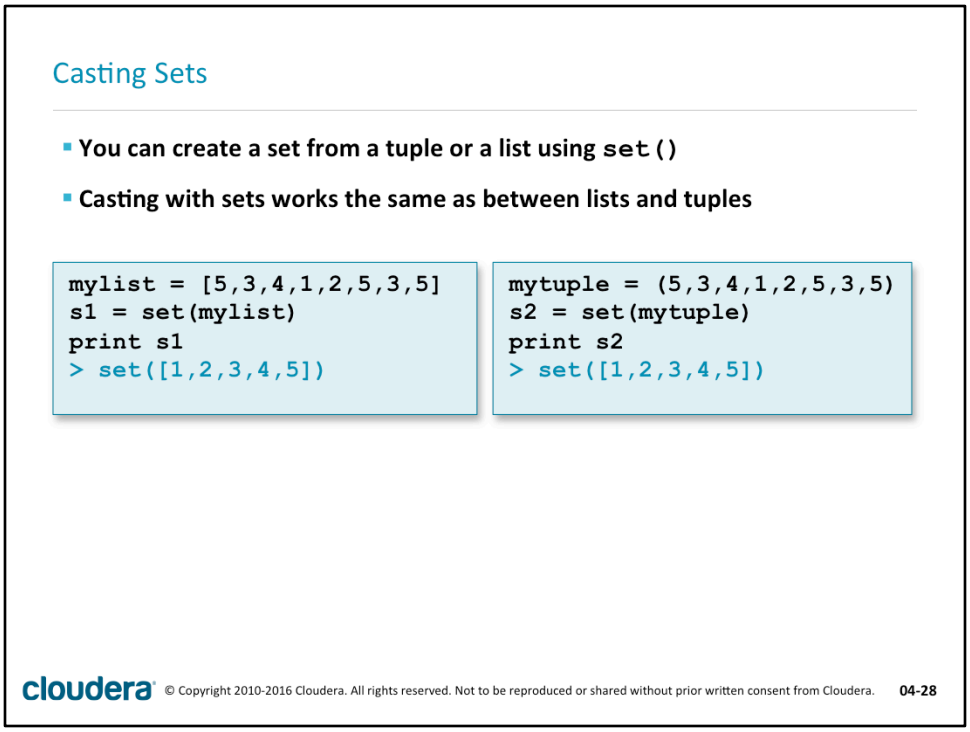
Sets

ifruit models of mobile phone.

[*] A set is a defined by a comma delimited series of elements bounded by curly braces.

[*]The main properties of a set are that it is unordered and unique. You can see in the example that sorting of the long series of values result in the very orderly series from lowest to highest, and with no repeated elements.

Note: Actually, it IS ordered – it is in order of entry.



Sets

[*] A set is a defined by a comma delimited series of elements bounded by curly braces.

[*]The main properties of a set is that it is ordered and unique. You can see in the example, that initialization of the long series of values result in the very orderly series from lowest to highest, and with no repeated elements.

Set Theory Methods and Operators

- ***set1*.union(*set2*)**
  - All the elements in both sets
  - Symbolic notation also works: ***set1* | *set2***
- ***set1*.intersection(*set2*)**
  - Only elements that are in both sets, and not in only one
  - Symbolic notation also works: ***set1* & *set2***

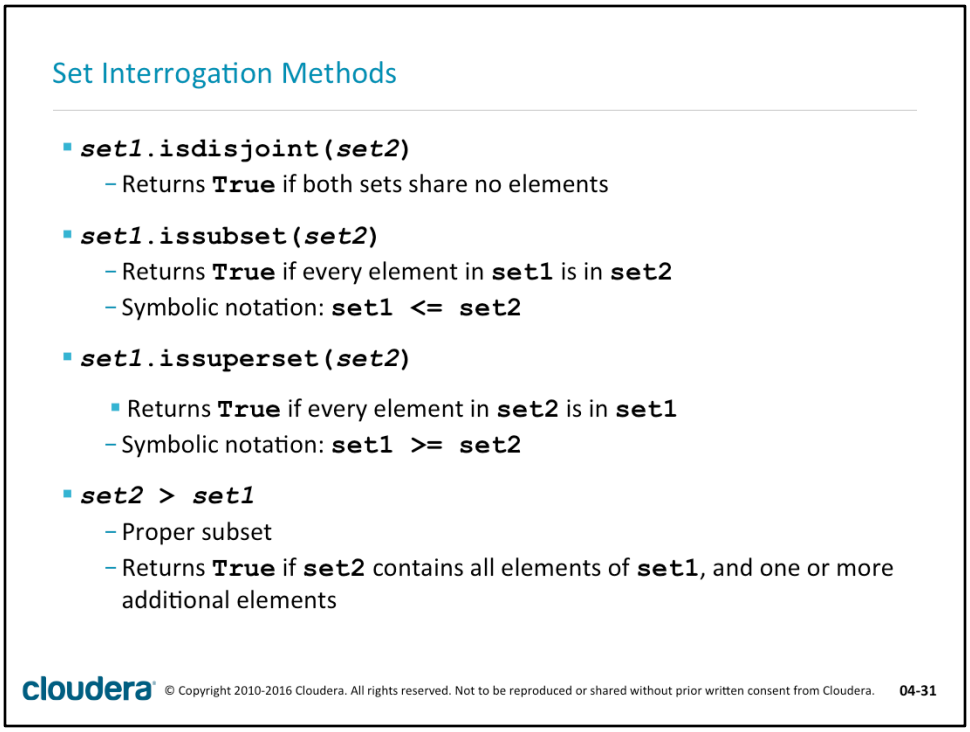
```
s1 = {1,2,3,4,5}
s2 = {3,4,5,6,7}

s1.union(s2)
> set([1, 2, 3, 4, 5, 6, 7])
s1.intersection(s2)
> set([3, 4, 5])
print s1 & s2
> set([3, 4, 5])
```


[*] The methods on sets provide 'set theory' operators, such as union, intersection (difference, and symmetric difference are on the next slide).

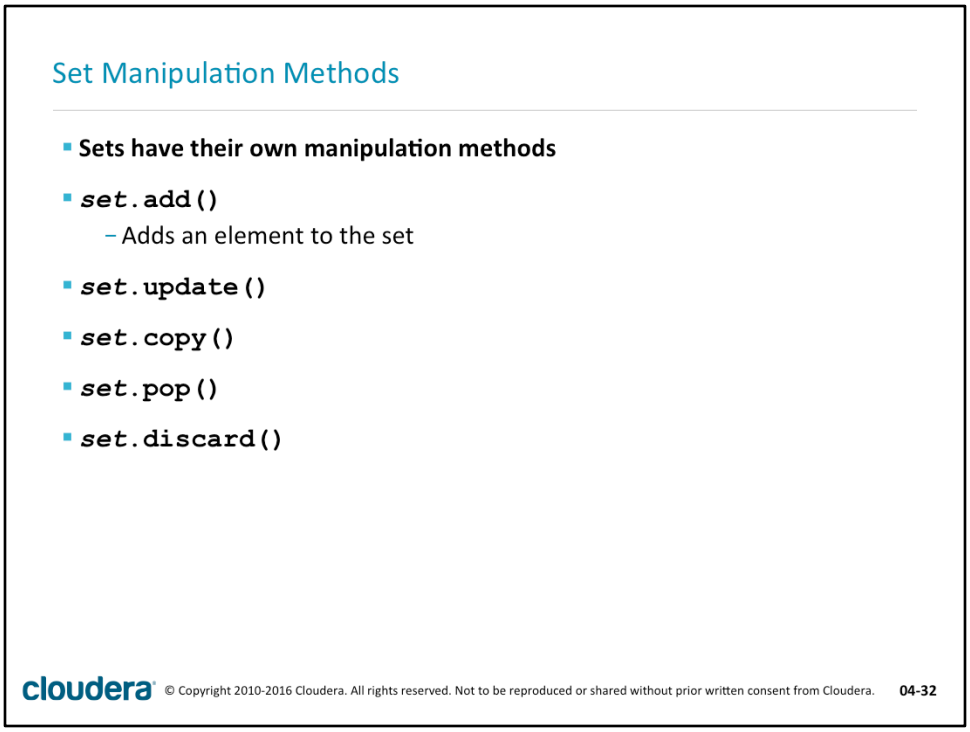
Set Theory Methods and Operators

- ***set1*.difference(*set2*)**
  - Provides the difference between the sets
  - The elements that are in either one or the other set, but are not in the intersection
  - Symbolic notation: ***set1 - set2***
- ***set1*.symmetric_difference(*set2*)**
  - Symbolic notation: ***set1 ^ set2***
  - Returns the set of all elements that are not in the intersection

```
s1 = {1,2,3,4,5}
s2 = {3,4,5,6,7}

s1.difference(s2)
> set([1, 2])

s2.difference(s1)
> set([6, 7])

s1 ^ s2
> set([1, 2, 6, 7])
```

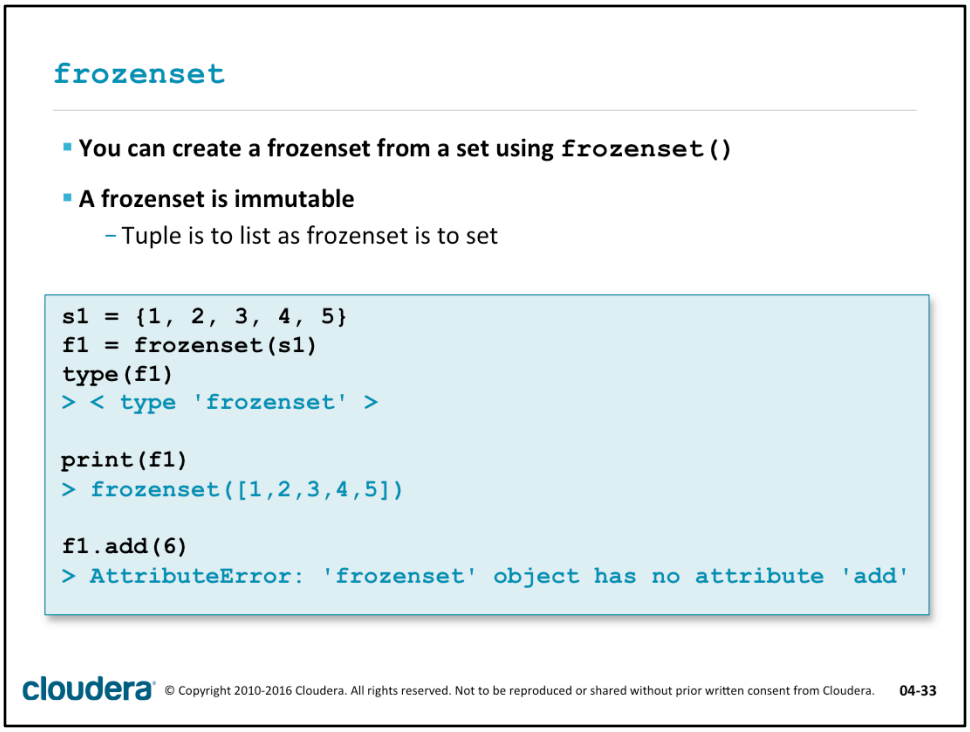
[*] The sets can be compared and interrogated about their relationship.



[*] In a list, you would append elements, but in a set, you add them. This makes sense because when an element is added to a list it will not necessarily appear at the end, but will appear in order. And if the element is redundant with the same element already in the set, it will result in no change to the set.

Example: .pop takes no arguments in sets. It destructively removes the first element from the set.
.pop in the list collection takes an argument. This is an offset into the list and that element is destructively removed from the lists.

```
mySet = set([1,2,3,4])
set.pop()
> 1
mySet
set([2,3,4])

myList = [1,2,3,4]
myList.pop(2)
myList
List[1,2,4]
```



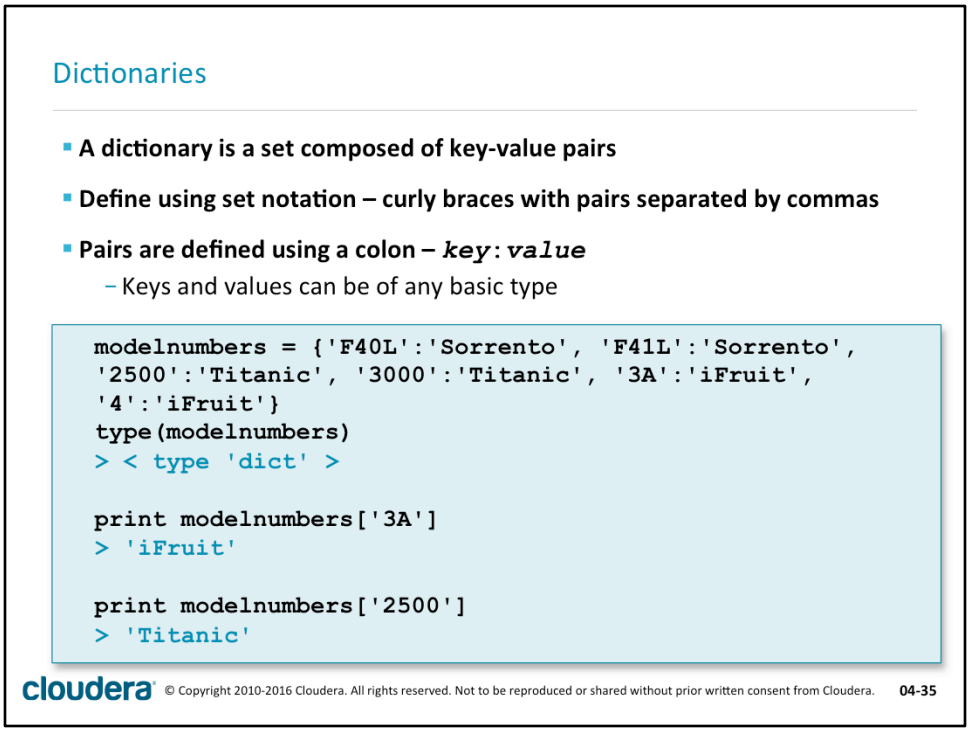
[*] Frozenset is to set as tuple is to list

A frozenset is an immutable set.

Chapter Topics

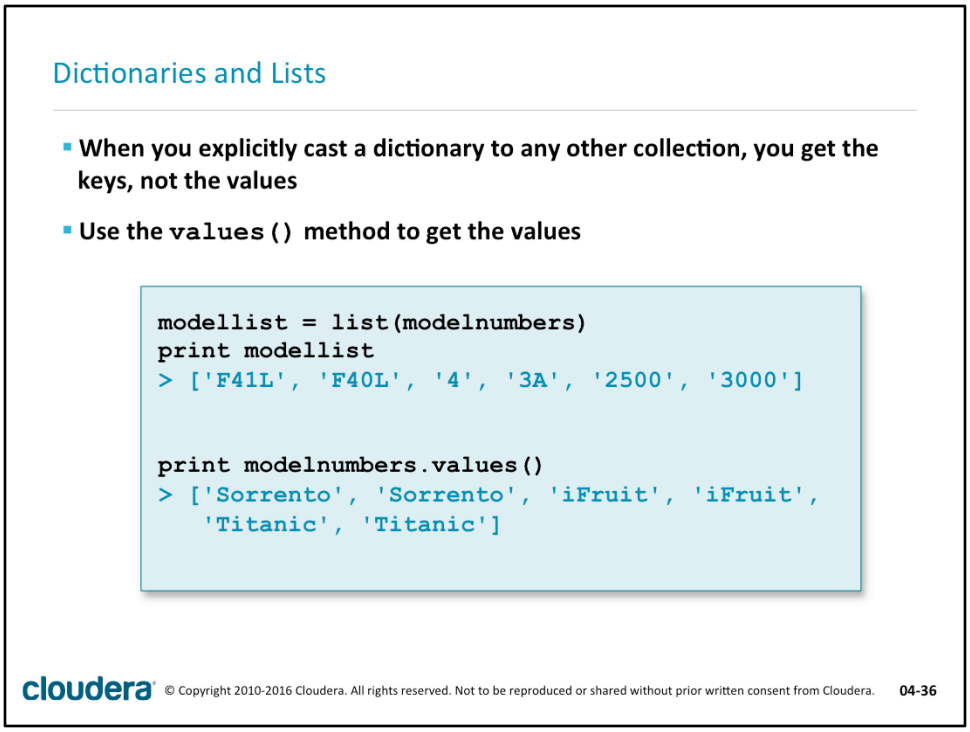
Collections

- Lists
- Tuples
- Sets
- **Dictionaries**
- Essential Points
- Hands-On Exercise: Collections



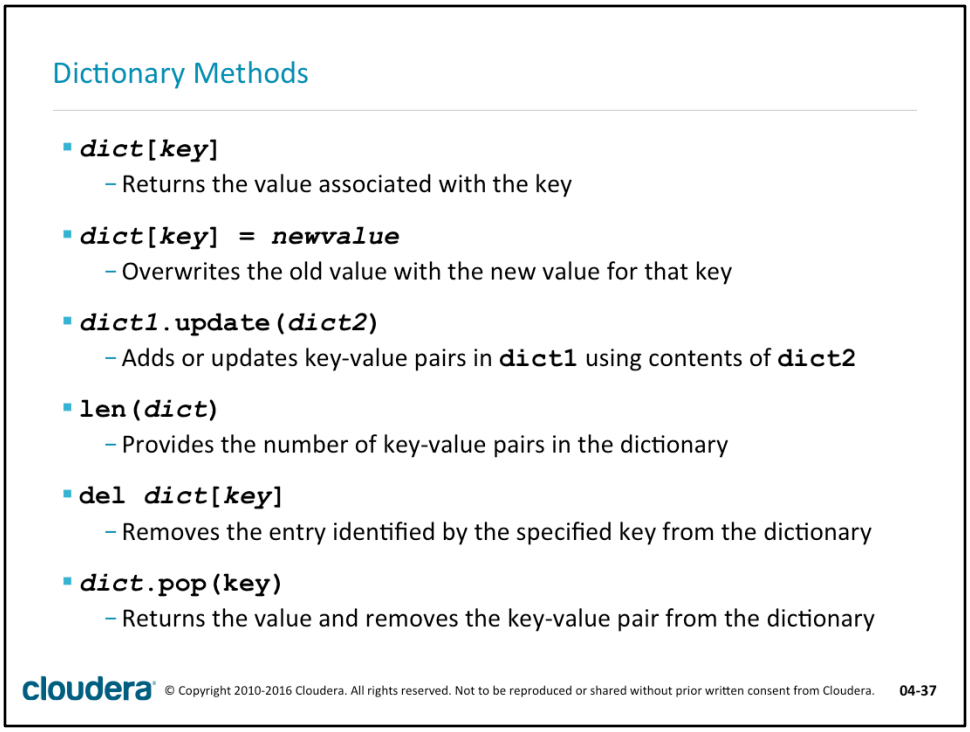
[*] A Dictionary is a Set made up of Key-Value pairs.
[*] You create a Dictionary by defining the elements in the set with the ':' between the Key and Value and each key:value pair is separated by commas.
[*] Dictionary[key] returns the corresponding value.
[*] The keys in Python are objects and do not have to be numerical.
[*] Keys and values can be of ANY type

```
modelnumbers =
{ 'F40L':'Sorrento','F41L':'Sorrento','2500':'Titanic',
'3000':'Titanic', '3A':'iFruit','4':'iFruit'}
```



[*] Casting a Dictionary to another collection results in the list of keys, not the values.

Instructor value-add
This is probably too detailed for the scope of this course. But the information is listed here in case it is useful.
A Tuple of Frozenset can be a key in a key:value pair, but a list and a set cannot be a key because they are mutable, meaning that the id of the objects could change.



[*] You can update the key in a key:value pair.

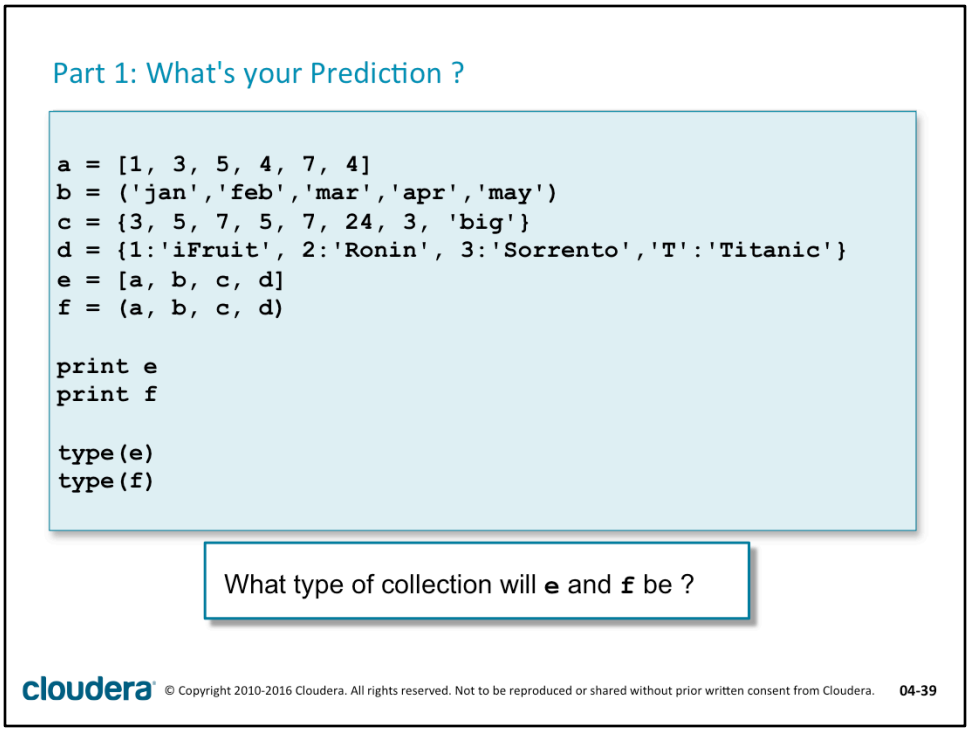
dict[key] = newkey
updates one key

dict1.update(dict2)
Uses the elements in dict2 to update/change the elements in dict1
Key:Value pairs are updated or added if necessary

Chapter Topics

Collections

- Lists
- Tuples
- Sets
- Dictionaries
- **Essential Points**
- Hands-On Exercise: Collections



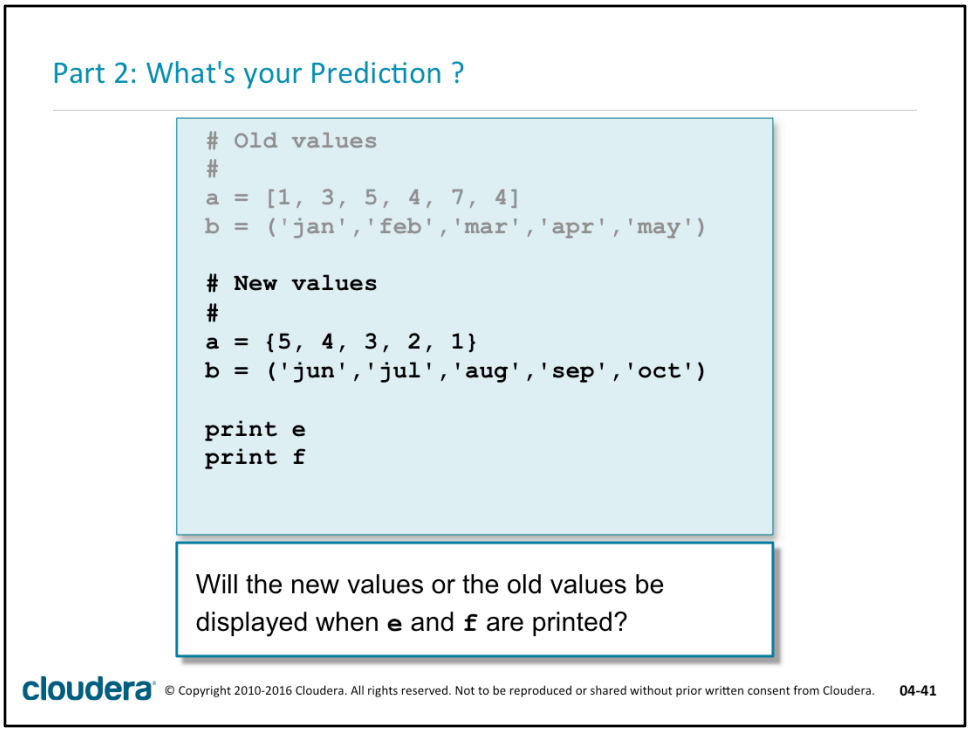
Let's have a little interactive quiz and see how everyone does.

What will be printed for e and for f and what type are they?

Note: answers on next slide
e is a list (square brackets), and f is a tuple(parenthesis)

Part 1: Results

```
print e
> [[1, 3, 5, 4, 7, 4],
   ('jan', 'feb', 'mar', 'apr', 'may'),
   set([24, 'big', 3, 5, 7]),
   {1: 'iFruit', 2: 'Ronin', 3: 'Sorrento', 'T': 'Titanic'}]

print f
> ([1, 3, 5, 4, 7, 4],
   ('jan', 'feb', 'mar', 'apr', 'may'),
   set([24, 'big', 3, 5, 7]),
   {1: 'iFruit', 2: 'Ronin', 3: 'Sorrento', 'T': 'Titanic'})

type(e)
> <type 'list'>

type(f)
> <type 'tuple'>
```

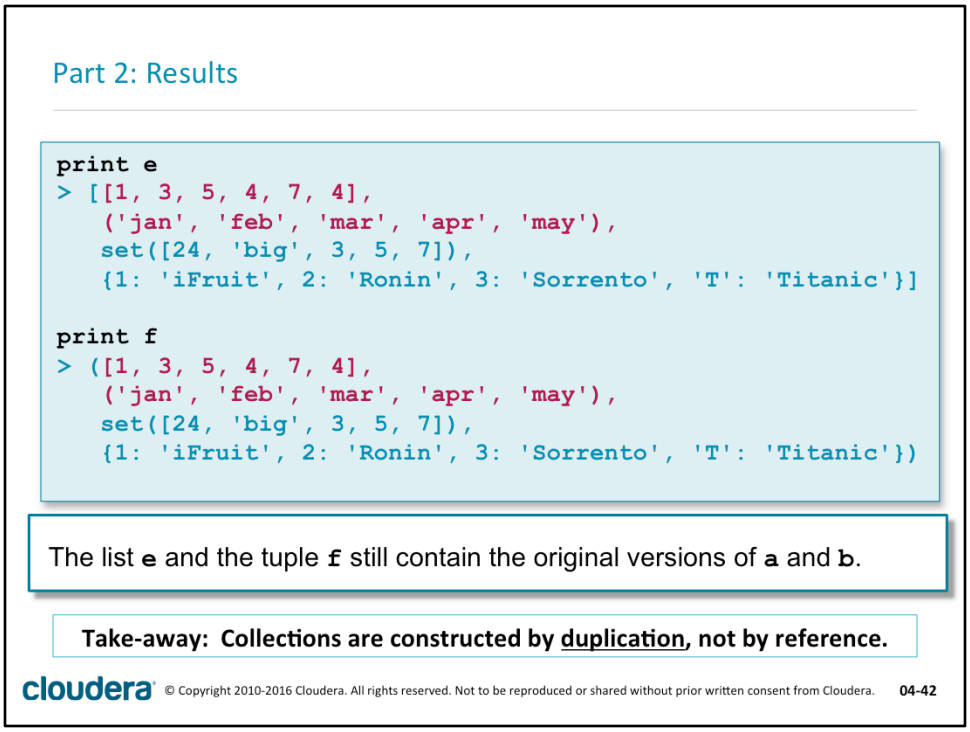
Now for the tricky part…

Originally, 'b' was 'jan','feb','mar'…
Now we are updating it to 'jun', 'jul', 'aug'.

Q: How will this affect 'e' and 'f'.

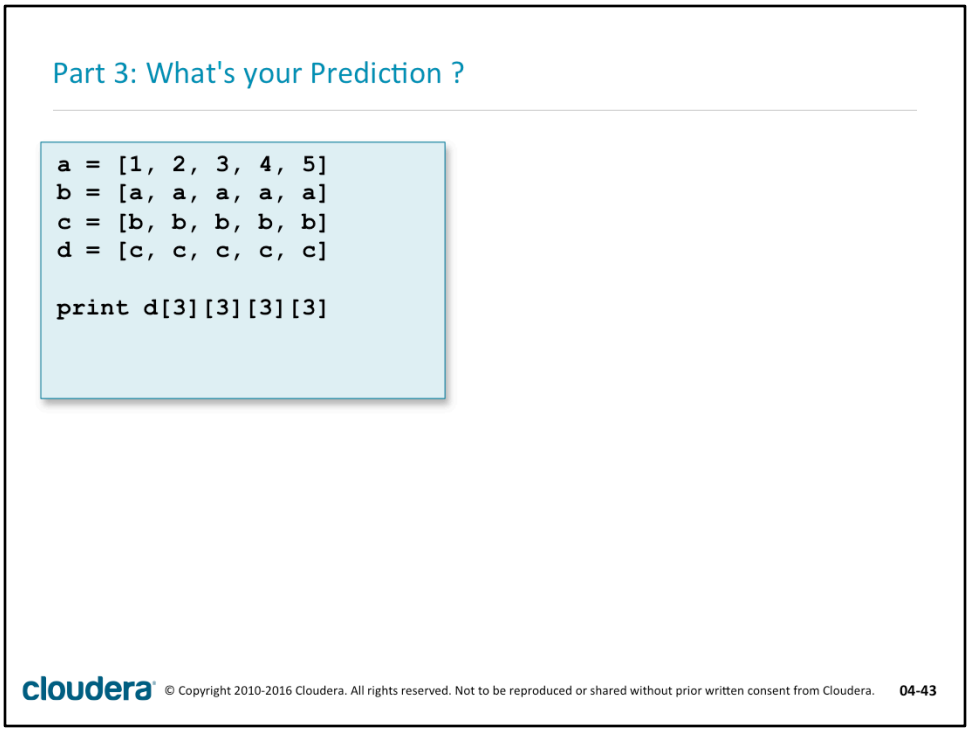
Remember, 'e' and 'f' were made from 'b'. Both of them included 'b'. Now that we've changed the definition of 'b' will 'e' and 'f' contain the old months ('jan'…) or the new months ('jun'…)

The student's intuition should lead them to guess that when list 'e' and tuple 'f' were created, that the then current value of 'b' was copied into the new data structures, not imported by reference.

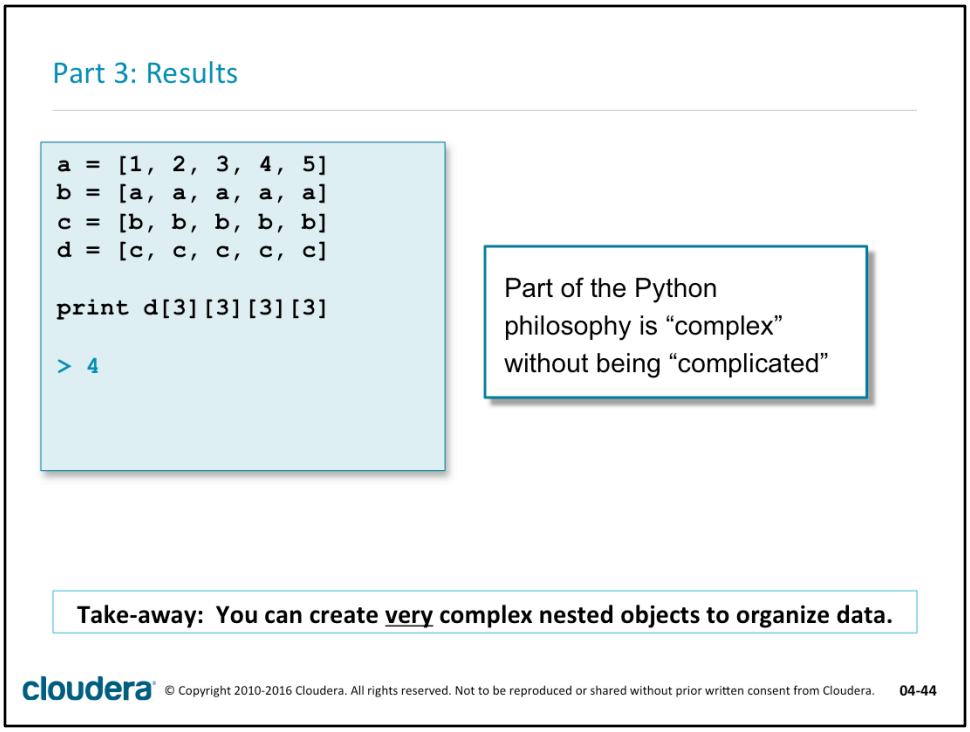


The original relationship between 'e' and 'f' and 'b' was not preserved.

[*] Collections are assigned and constructed by copying data elements, not by reference. So the logical association is lost after the assignment has been executed.



This illustrates how complex collections can be built up.



[*] Collections can be nested.

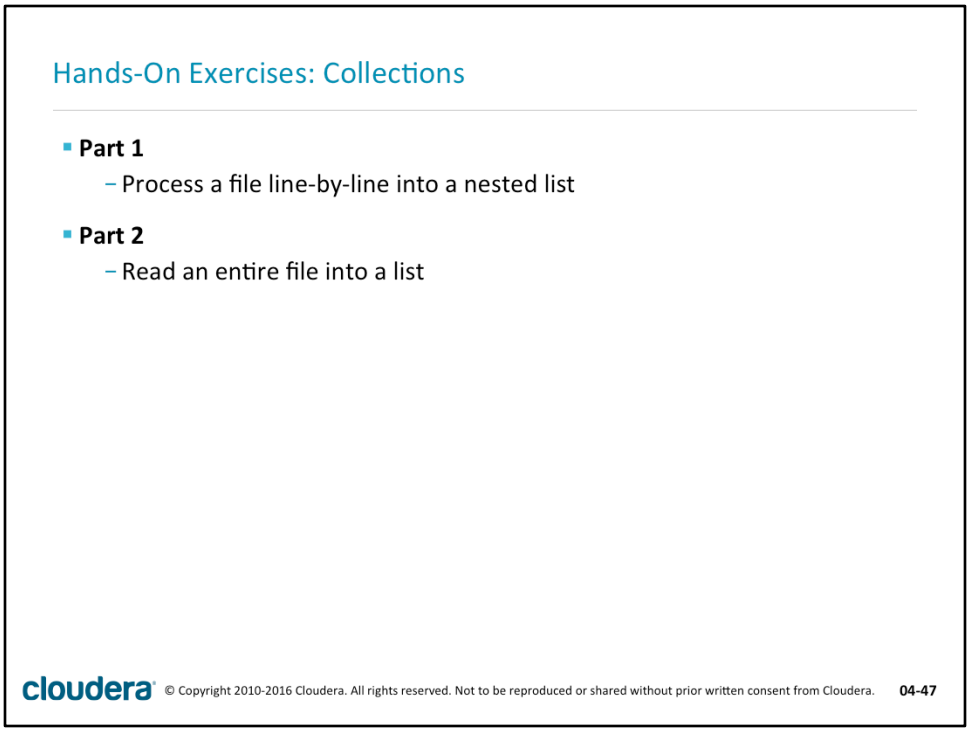
Essential Points

- **Collections**
  - May consist of elements of different types
  - Can contain other collections as elements (nested)
- **`Lists []`**
- **`Tuples ()`**
- **`Sets {}`**
- **`Frozensets` – immutable sets**
- **`Dictionaries {key:value}`**

Chapter Topics

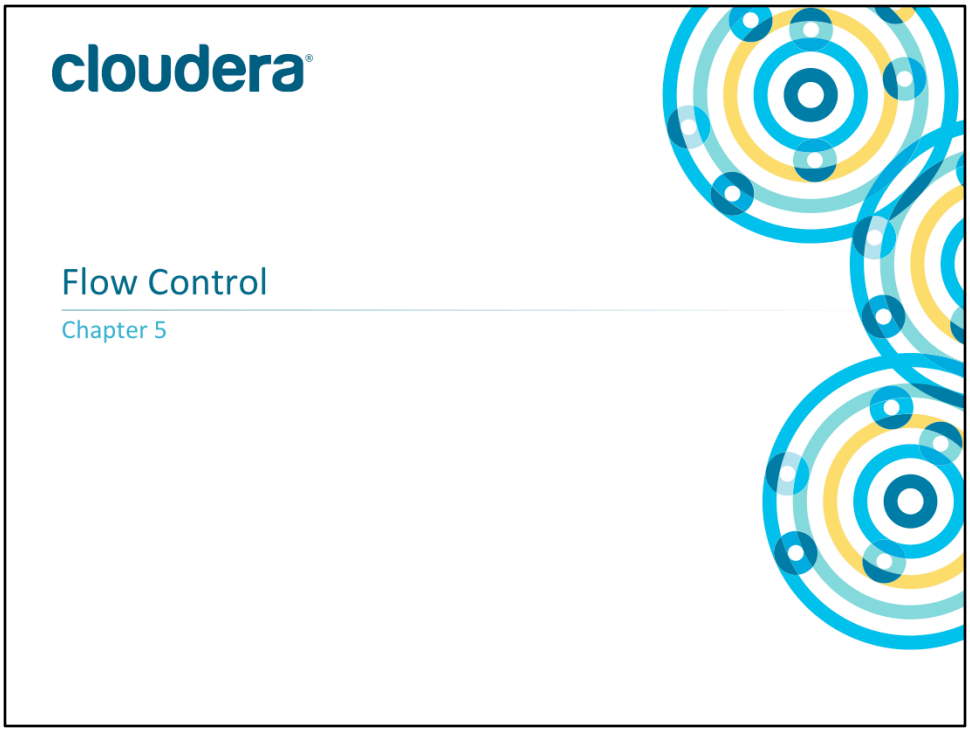
Collections

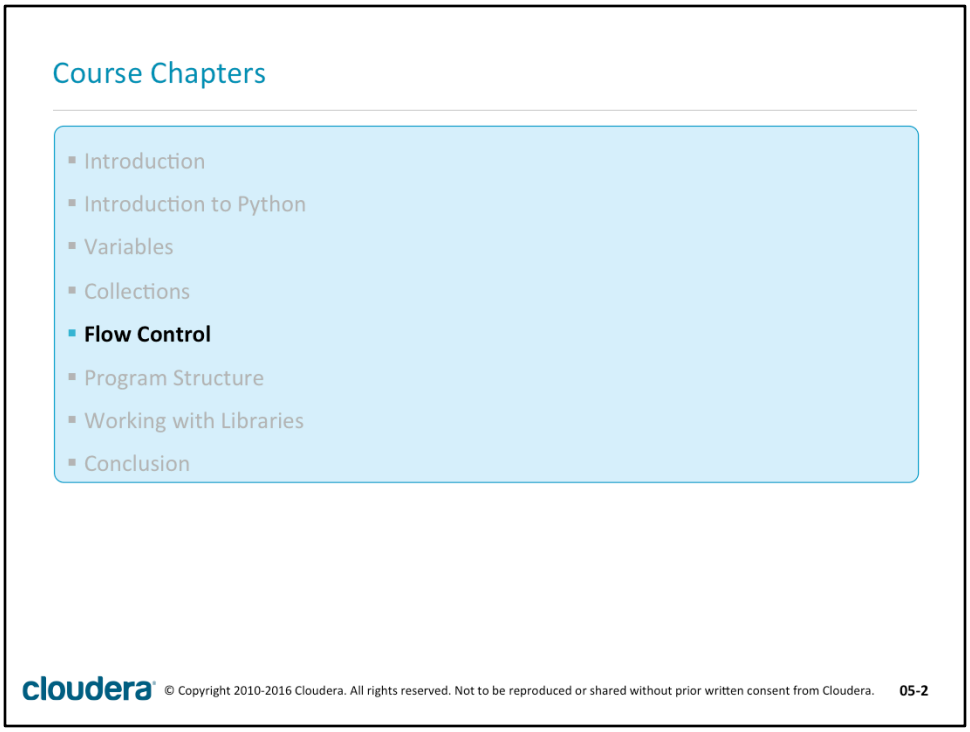
- Lists
- Tuples
- Sets
- Dictionaries
- Essential Points
- **Hands-On Exercise: Collections**



Note – they don't know flow control yet, so the more sophisticated exercise comes next.
This exercise is mainly for familiarity with the collections and their methods.

This exercise is just for familiarity with collections and file I/O. Iterators and conditionals (for and if) are needed to address individual elements within a collection. More sophisticated exercises with lists, sets, and tuples are in the next chapter.



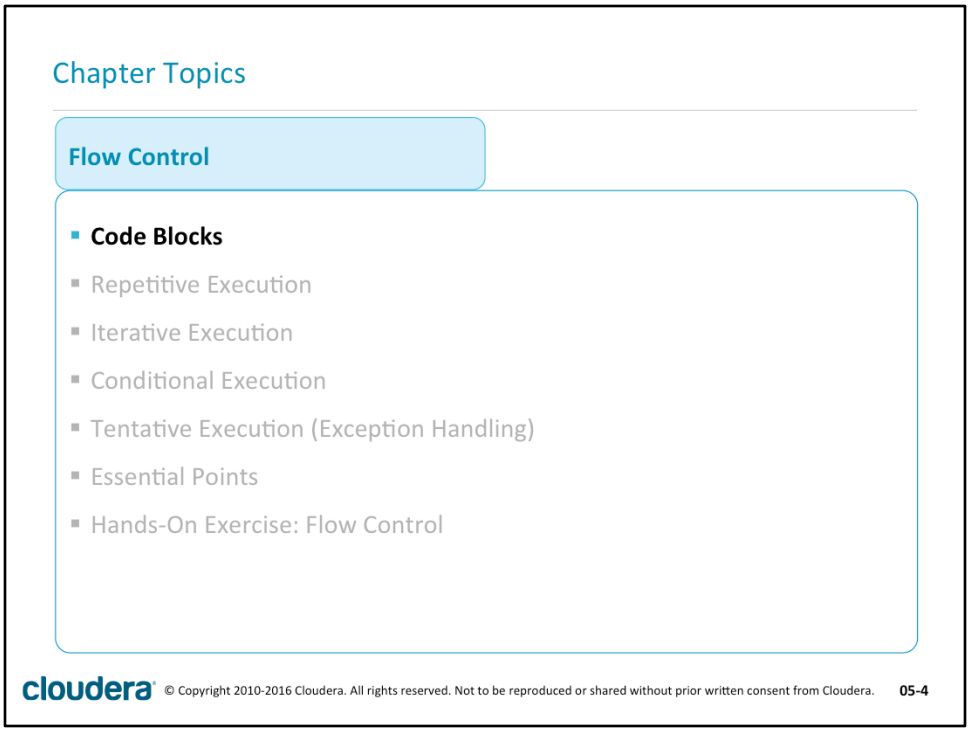


Flow control – how control is passed from one section of code to another.

Flow Control

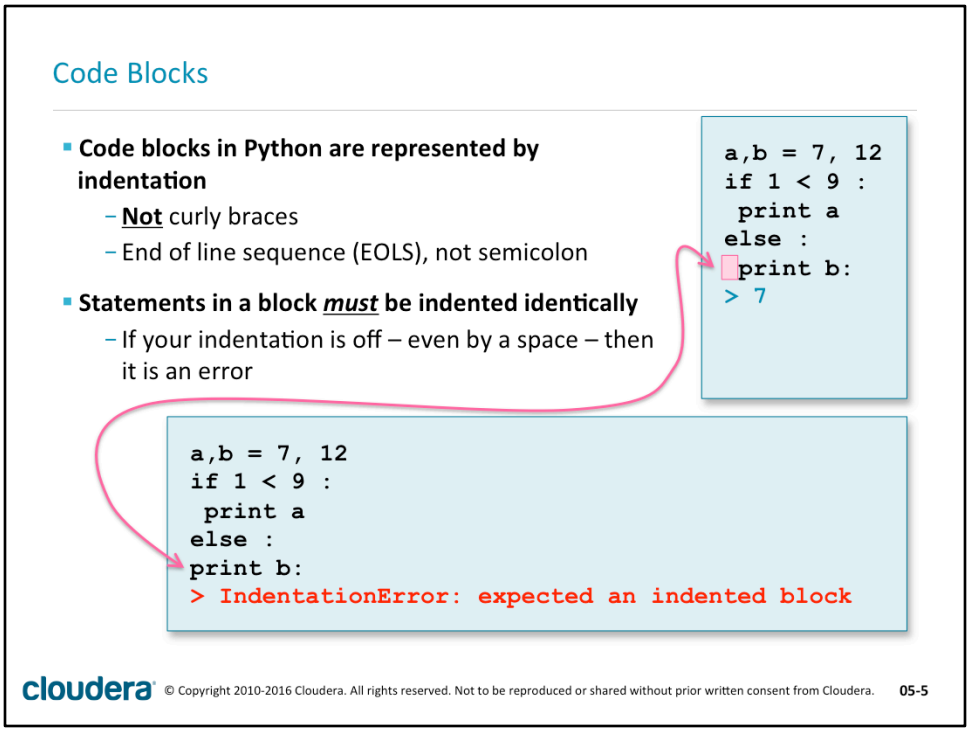
In this chapter you will learn

- **How to loop and perform repetitive operations**
- **How to iterate over individual items in a Collection**
- **How to make a block of code conditionally execute**
- **How to identify and handle exceptions**



[*] First – how we create sections of code, called 'code blocks'

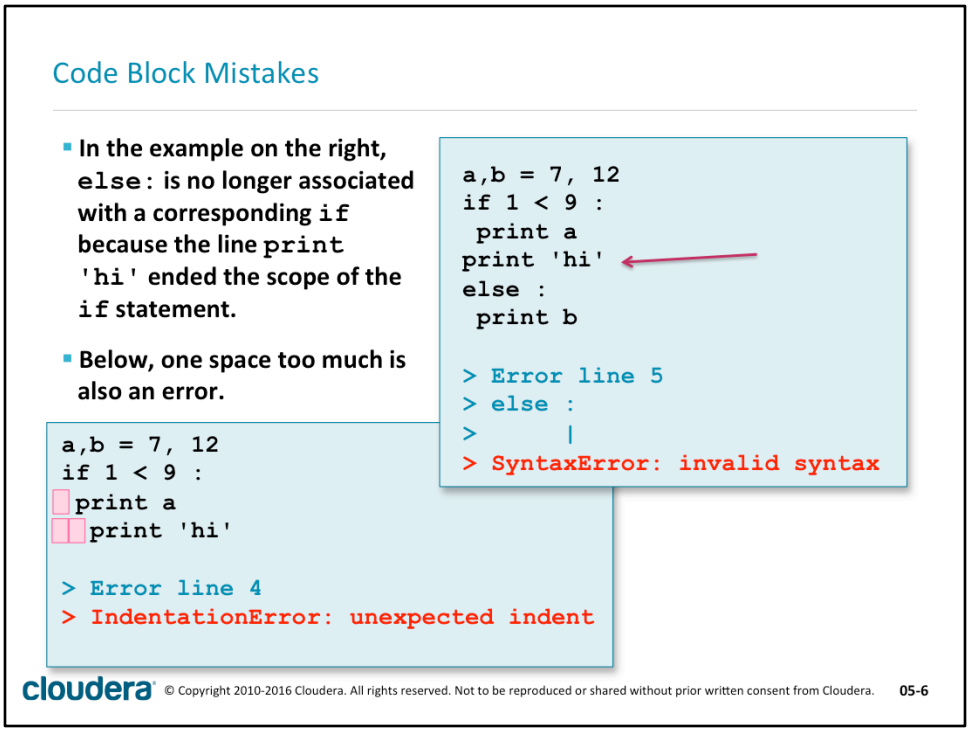
[*] Next – how to change flow of control. Python uses keywords that operate on code blocks to change the execution of the code.



[*] Code blocks in Python are indicated by indentation.

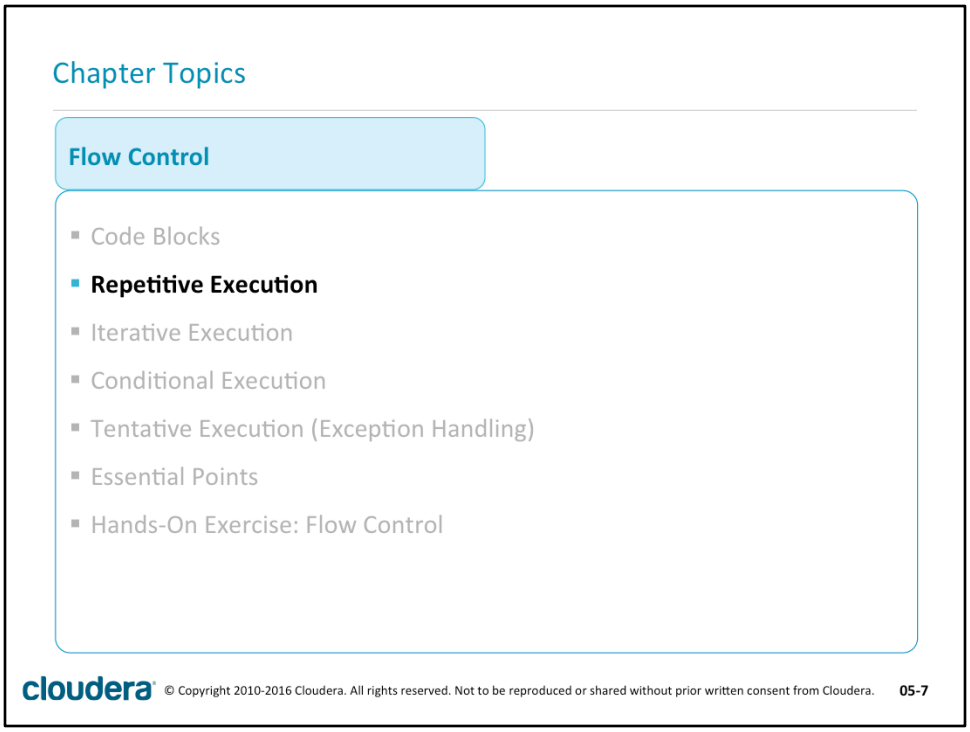
[*] No alternatives – such as curly braces

[*] You have to indent a code block by at least one space. You can indent by more than one space. However, once the initial indent is started, all other lines in the block must be indented the same number of spaces. If you indent more, then it is a new code block – and potentially a syntax error.

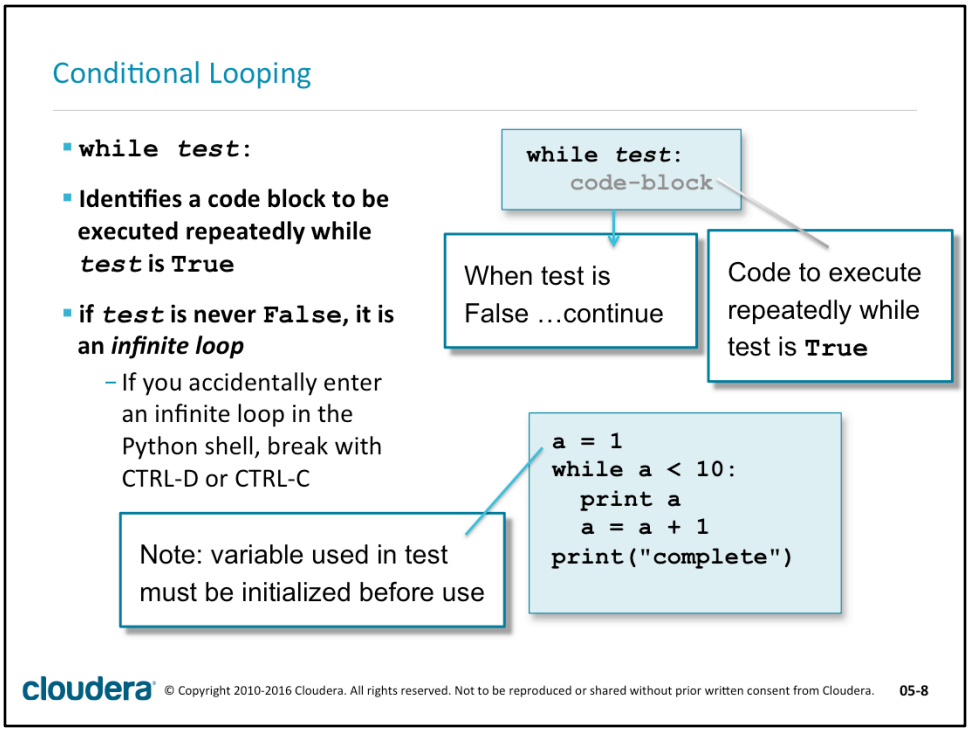


[*] It is easy to create code block error. Just indent for readability or forget a space and you've got a syntax error.

The indentation keeps Python code easily readable, but can be very annoying to those who have not had to be concerned with consistent indentation before.



Repetitive execution is also called "looping".



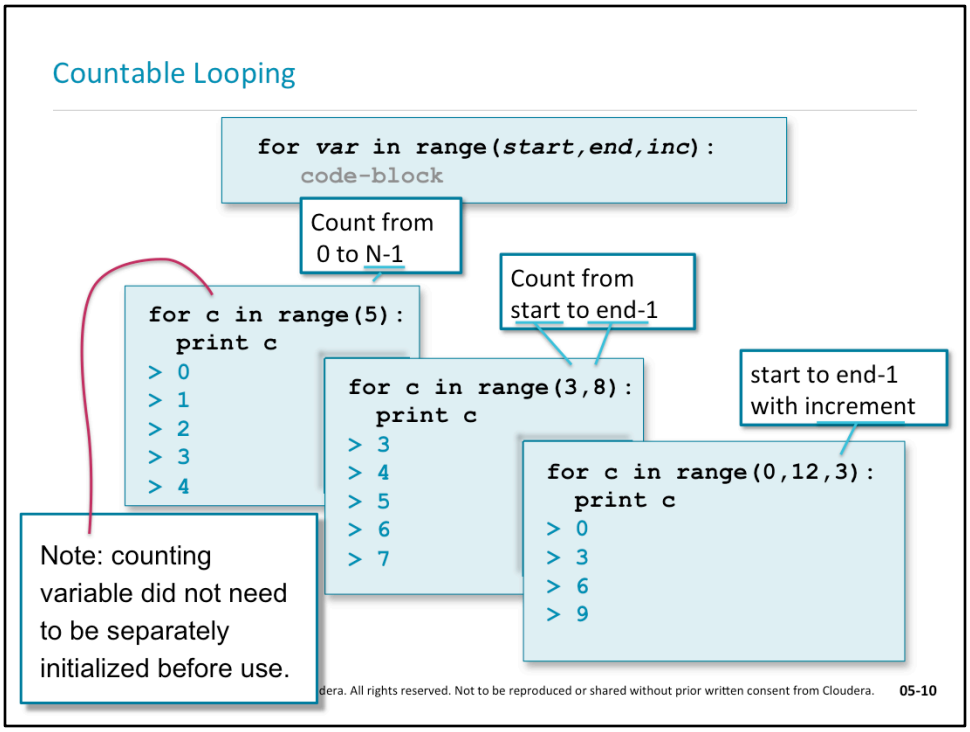
[*] While executes a code block repeatedly if test is **`True`**.

[*] You must modify some variable involved in the test within the code block or it will continue to loop until interrupted by a 'break' or program interrupt sequence. Such behavior is called an *infinite loop*.

Example Code

- This code prints each record of mobile phone data

```
count = 0
while count < len(devlog):
   print count, len(devlog[count])
   print devlog[count]
   count += 1
```

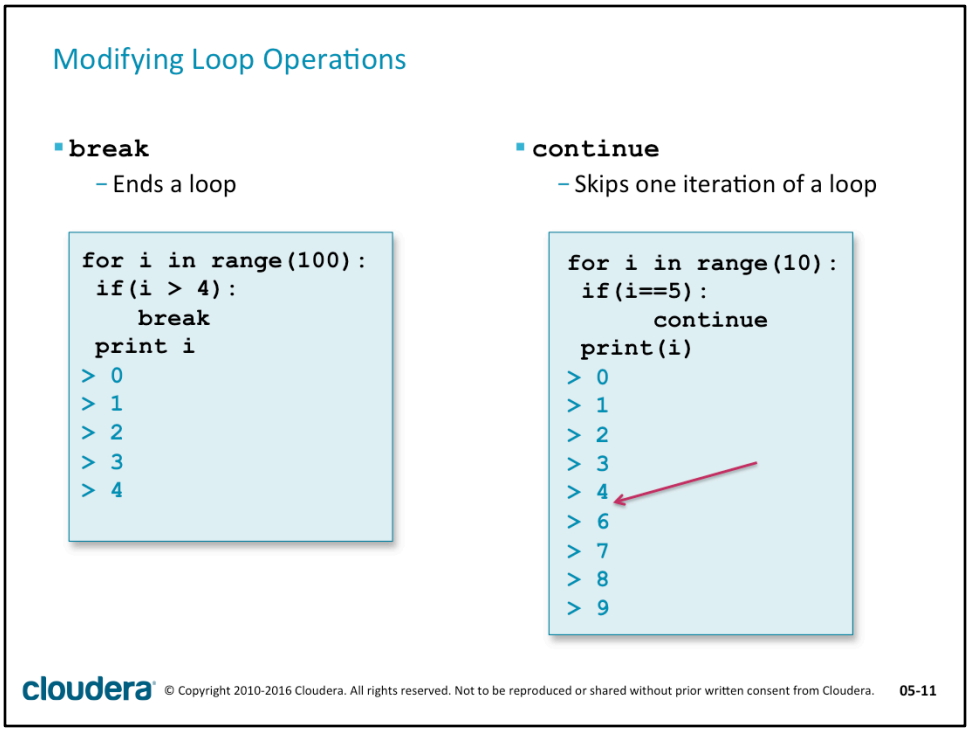



Countable looping is implemented with a "for" construct, and is handy when you need an integer that increments in a specific pattern. In other languages, countable loops are critical because they are used to index collections. Python provides iterative looping for collections, so countable looping is less significant.

[*] Upper end of range is **not** inclusive.
[*] Bottom end of range IS inclusive.
[*] Increment is optional. If omitted, '1' is used.

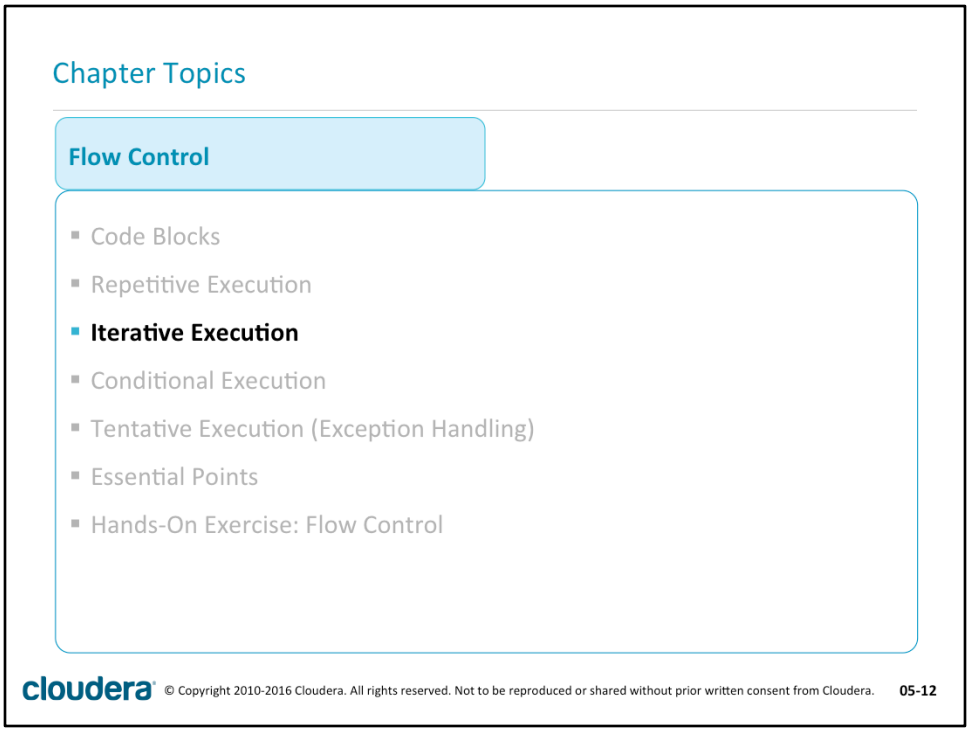
Note:
for…in…range() is really a subset of for….in...collection. However, we are introducing it first because students can find it confusing to discuss iteration first if they are more familiar with a countable looping language. Presenting the topics in this order ensures that the counting questions are *already answered* before you discuss iteration.

Note: 'c' in the for loop is initialized automatically by the interpretation of the for statement, which is why it did not need to be initialized in a separate statement.

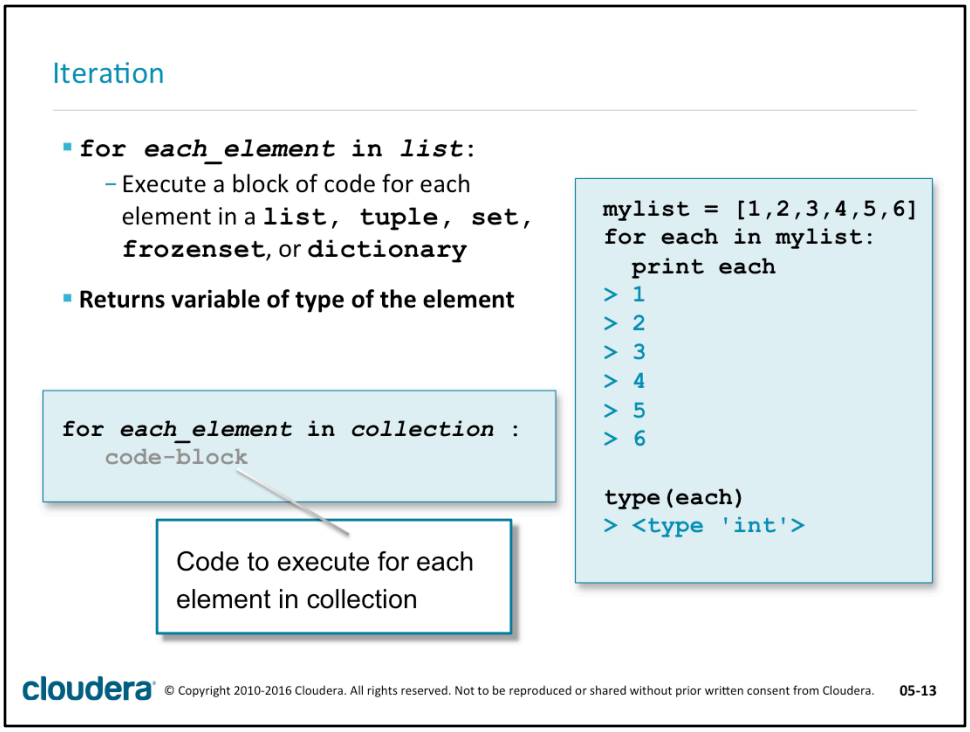


[*] **break** unconditionally ends the loop, forcing execution to resume at the code following the looping code block.

[*] **continue** forces execution to resume at the top of the loop, skipping the remainder of the code in the current code block. In the example above, when the looping counter, i, is equal to 5, the print line is skipped.



Iteration of collections is separate from countable looping in Python.



[*] Executes a code block for each individual element in a collection (list, tuple, set, frozenset, or dictionary). The element is returned to the variable (each_element in the example in the slide).

Note that because collection can consist of many types of elements, the variable type for 'each_element' could change as each element is fetched.

`Key` and `Value` Iteration

- Example of iterating over keys and values simultaneously in a Dictionary

```
modelnumbers = {'F40L':'Sorrento', 'F41L':'Sorrento',
'2500':'Titanic', '3000':'Titanic', '3A':'iFruit',
'4':'iFruit'}

for key,value in modelnumbers.items():
 print key,value
> F41L Sorrento
> F40L Sorrento
> 4 iFruit
> 3A iFruit
> 2500 Titanic
> 3000 Titanic
```

List Comprehension

- **Python provides a technique to create a new list from an existing collection by placing an iterator within brackets**
  - Each element is processed to create the new list

```
newlist = [code for each_element in collection]
```

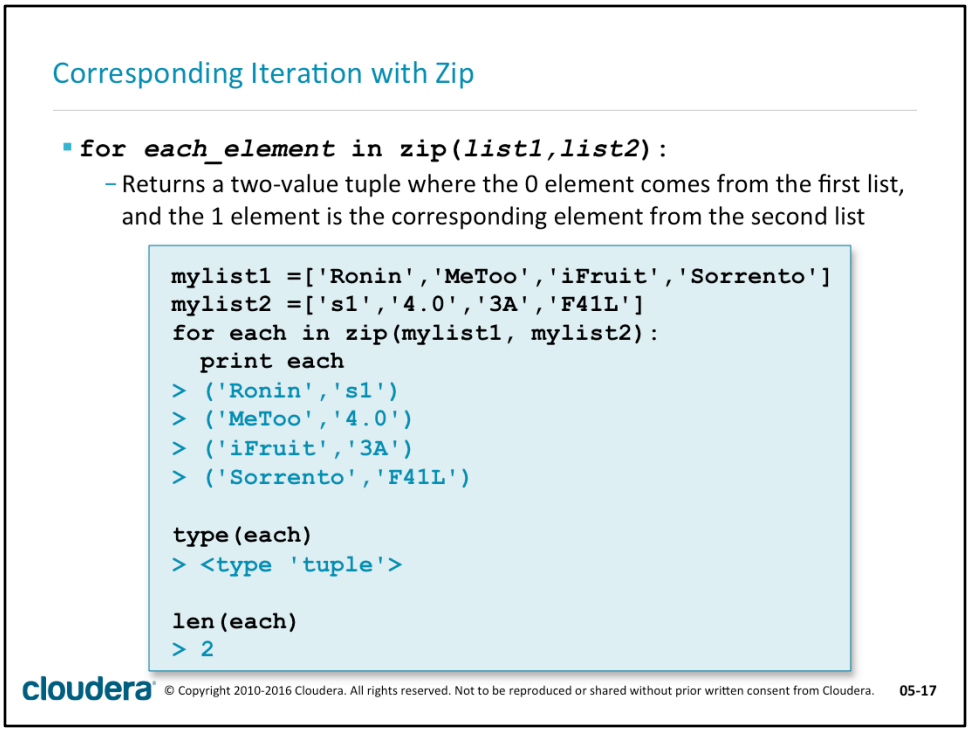
```
devicetemp=[30,33,54,34,29,49]
Ftemp = [elem*9/5+32 for elem in devicetemp]
print Ftemp

> [86, 91, 129, 93, 84, 120]
```


Iteration with Enumeration

- **`for each_element in enumerate(list):`**
  - Returns a two-value tuple where the element at index 0 is the enumeration, and the element at position 1 is the item content

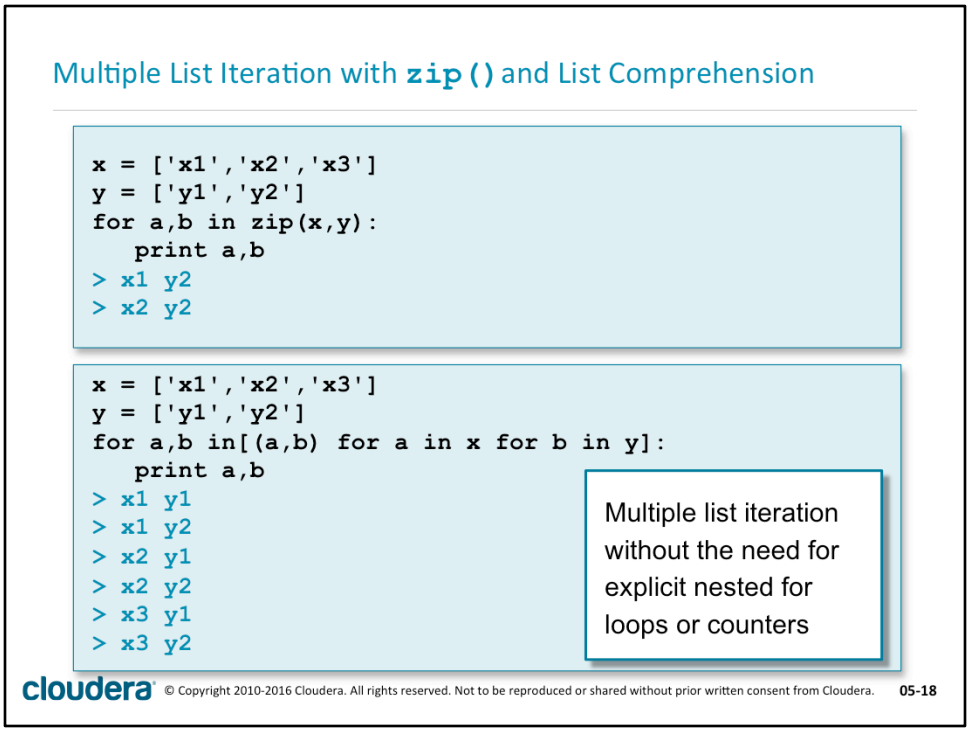
```
mylist = ['Ronin','MeToo','iFruit']
for each in enumerate(mylist):
  print each
> (0, 'Ronin')
> (1, 'MeToo')
> (2, 'iFruit')

type(each)
> <type 'tuple'>

len(each)
> 2
```


[*] enumerate passes back an enumeration (counting) of the ordinal number of the element in the collection (0, 1, 2, …) paired with the element itself. The variable returned is a two-element tuple with the number in the zero position followed by the element itself in the one position.



[*] zip ordinal matches and pairs corresponding elements in two collections. The first element in list1 is paired with the first element in list2, and a two-element tuple is returned with the pair.

Many students may think of file compression when they hear “zip” in a computer class. We recommend explaining that this method is unrelated to compression, but instead the name should convey that it works like a zipper (items from the two independent sides come together piece by piece).



Note: The :for comprehension" isn't just a "for statement" as exist in other languages. This isn't just an alternative syntax to create nested for loops. There is no need to define a counting variable (as would appear in the for construct in other languages) just to serially access items in a group. And Python knows that the for comprehensions are defined together, so it has the discretion to implement the generation. For example, it can construct a single loop with multiple pointers to address contents in x and in y, looping for x*y. It is not forced to implement two concentric loops.

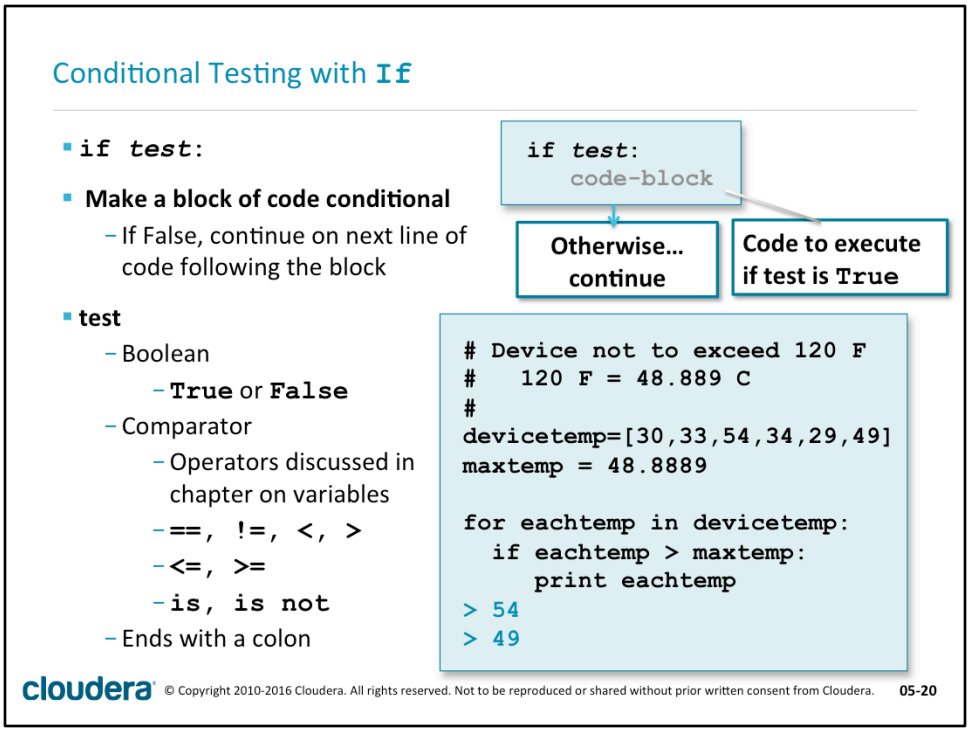
Note: map() example below can be used in context with zip() to illustrate the flexibility of the :for comprehension. map() is covered elsewhere in the course in more detail.

```
x = ['x1','x2','x3']
y = ['y1','y2']
for a,b in map(None,x,y):
   print a,b
> x1 y1
> x2 y2
> x3 None
```

Chapter Topics

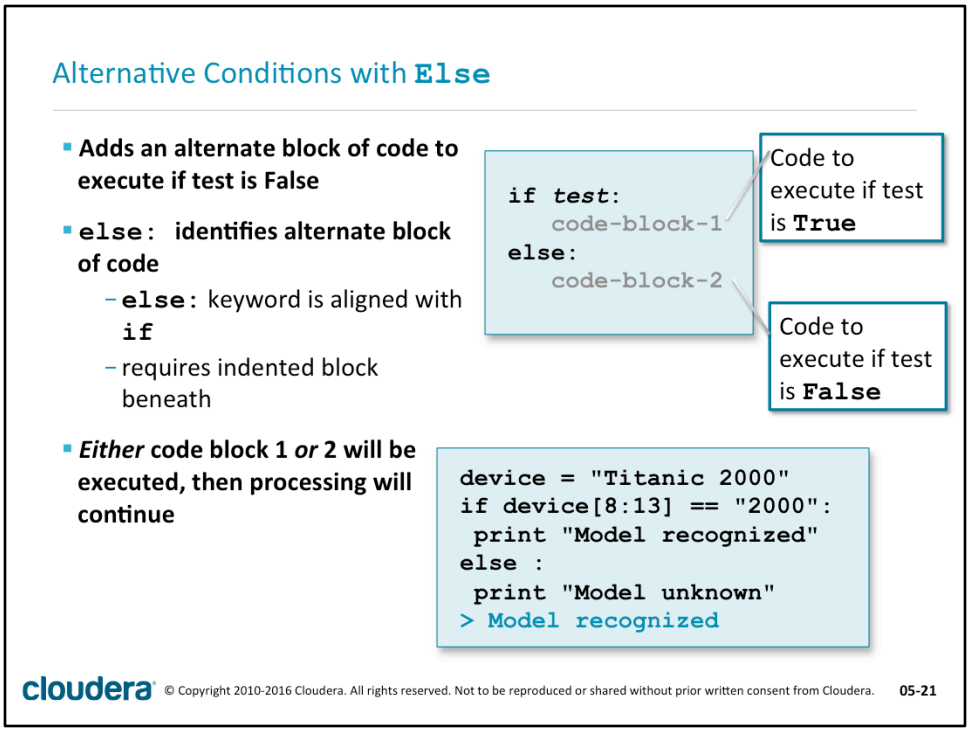
Flow Control

- Code Blocks
- Repetitive Execution
- Iterative Execution
- **Conditional Execution**
- Tentative Execution (Exception Handling)
- Essential Points
- Hands-On Exercise: Flow Control

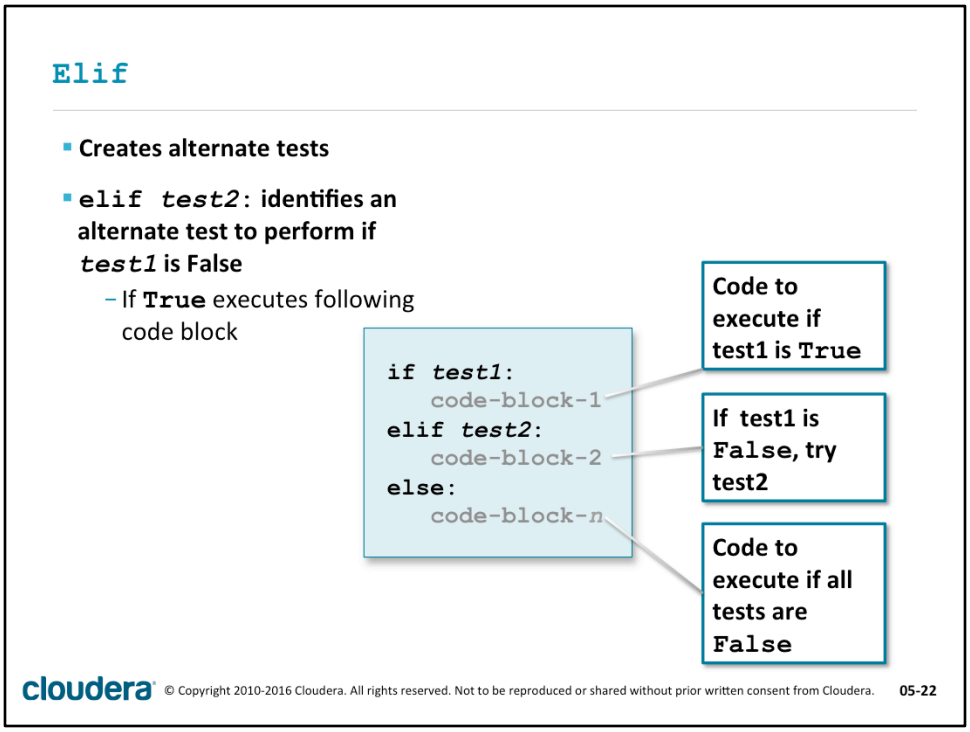


[*] Use If (test): to cause code to be executed only if 'test' is **`true`**.

[*] test is any of the operators or comparators resulting in a boolean.

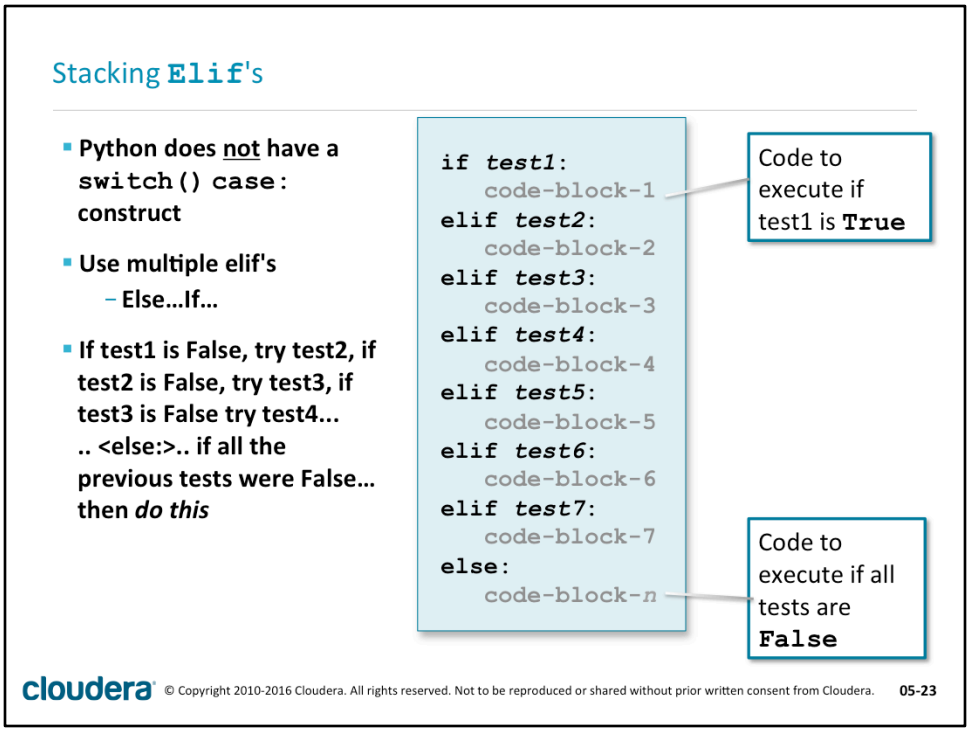


[*] Else
Adds a code block to be executed when test is false.



[*] Elif adds an additional test and code block to be executed if this alternative test is **`true`**.

[*] Python evaluates test1. If it is **`true`**, then its associated code block is executed. If not, then test2 is evaluated. If test 2 is **`true`**, then its associated code block is executed. This continues through test N. If none of the tests was **`true`**, then Python executes the code block identified by "Else".

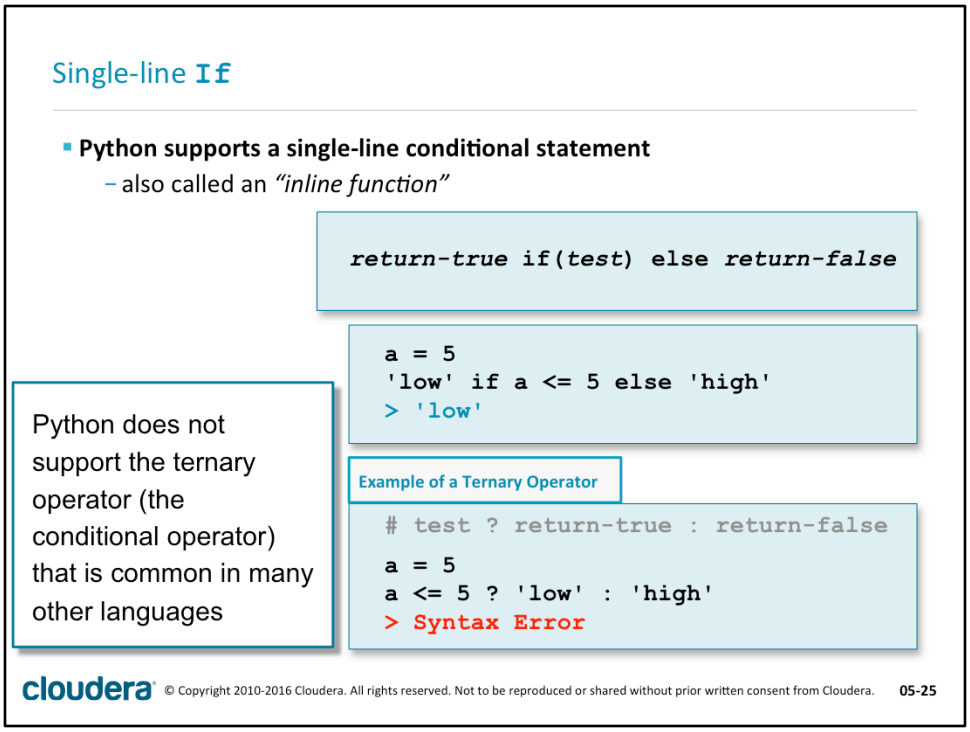


[*] You can stack up "Elif's" to simulate a "switch() and case:" statement which is a common construct in many languages, but not available in Python.

Example of `Elif`s

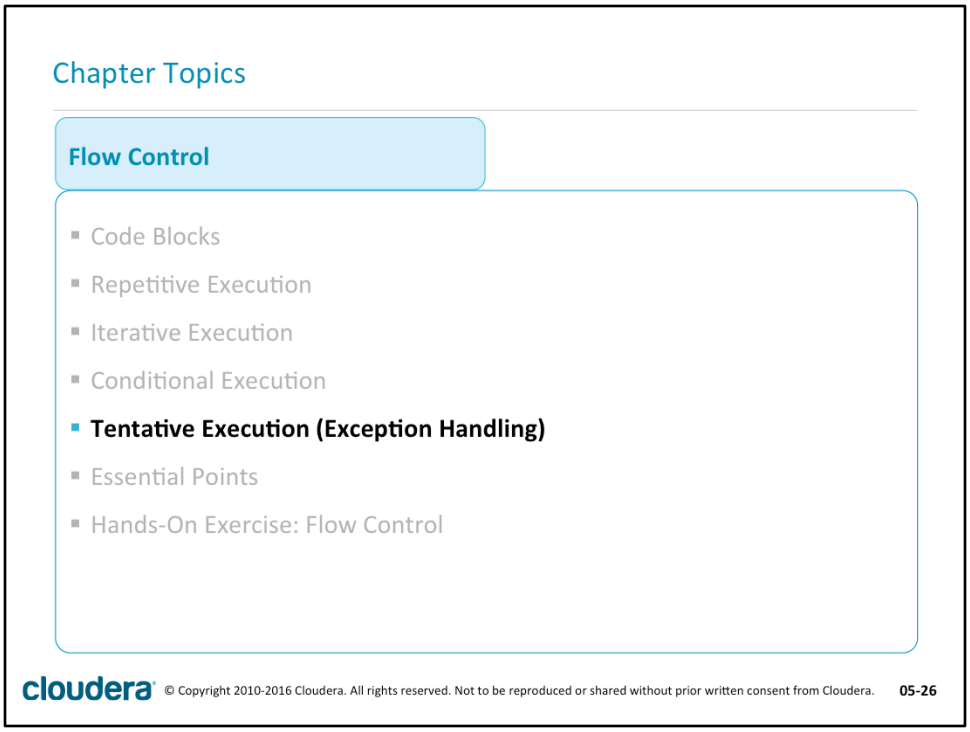
- Titanic phones are advertised by brand name rather than model number
- Advertising names are based on types of sailing ships
- Example shows use of `if…elif` to recognize model number and print corresponding market name

```
# Get make and model from keyboard
device = raw_input('Make Model :')

# Recognize model number
if device[8:13] == "1000":
 mod = 'Clipper'
elif device[8:13]== "1100":
 mod = 'Schooner'
elif device[8:13]== "2000":
 mod = 'Sloop'
elif device[8:13]== "2100":
 mod = 'Caravel'
elif device[8:13]== "2200":
 mod = 'Cutter'
else :
 mod = device[8:13]
# Prepare output string
mod = device[0:8]+mod
print mod
```

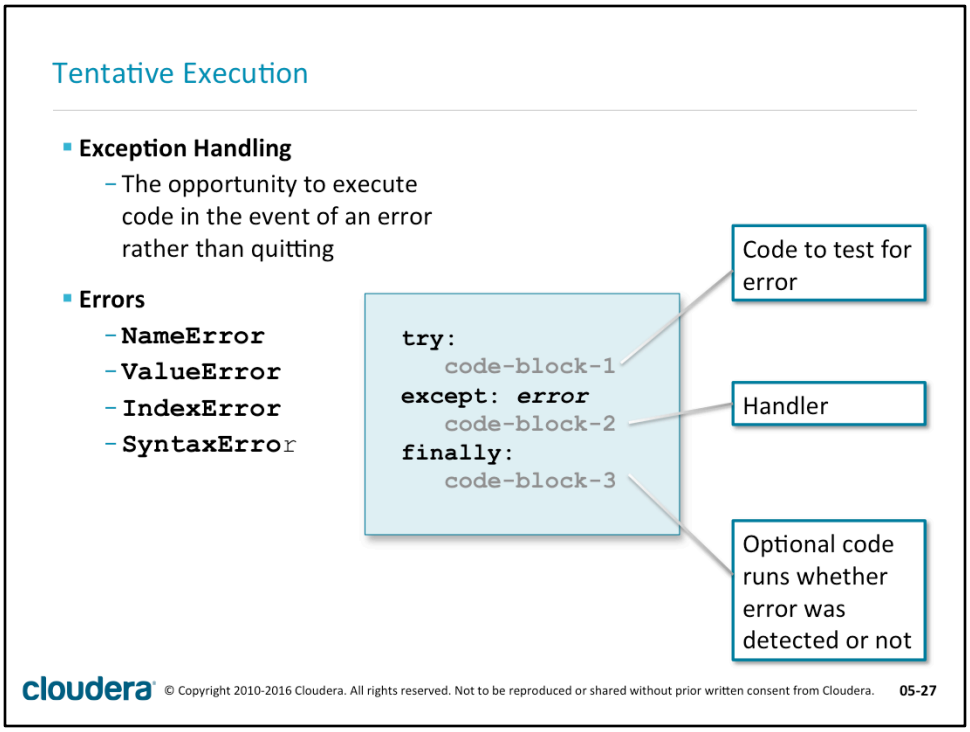


Instructor value add:

Python does not support unconditional branching, such as GOTO or JUMP to Label, that would continue execution at a labeled location in code.



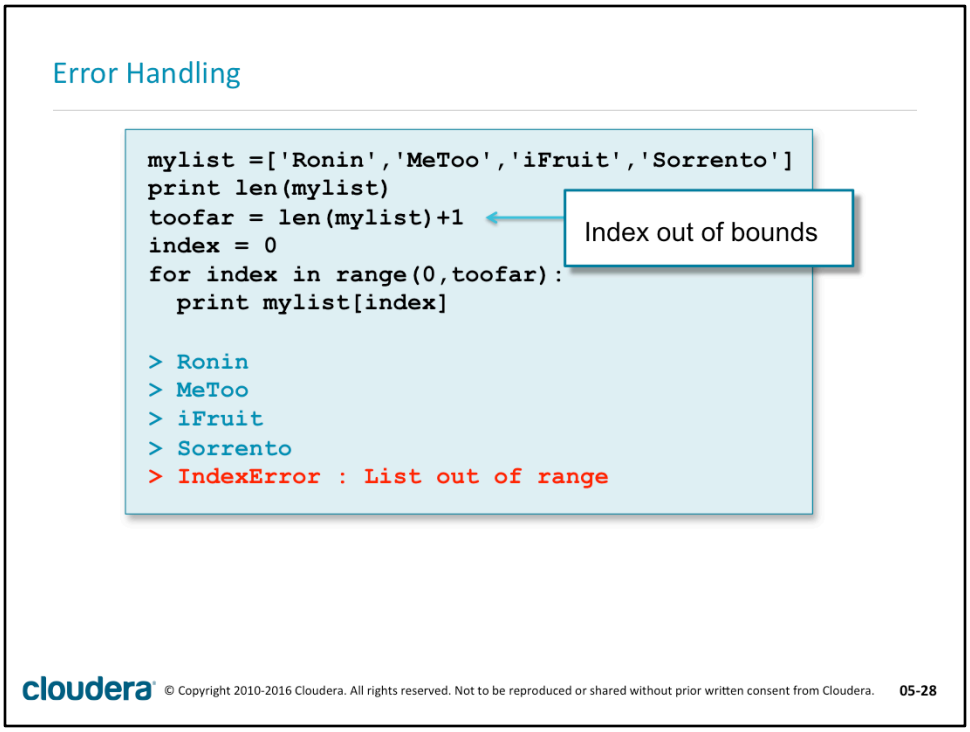
Tentative execution is also called error handling or exception handling.



[*] When an error is detected by Python, the default behavior is to halt execution and terminate the program. Instead, you can provide a code block to be executed if the error occurs.

This can be handy if you expect an error to occur.

[*] Use **try** to identify the code block for which an exception handler is to be defined. Then use **except** to define the type of error that causes the associated code block to be executed. A code blocked tagged with **finally** will run following the try: code block whether or not an error was detected.



How can we *trap* the IndexError so that the program doesn't quit?

Error Handling

```
mylist =['Ronin','MeToo','iFruit','Sorrento']
print len(mylist)
toofar = len(mylist)+1
 index = 0
try:
 for index in range(0,toofar):
   print mylist[index]
except IndexError:
  print ">> tragedy averted"
finally:
  print ">> continuing"

> Ronin
> MeToo
> iFruit
> Sorrento
> >> tragedy averted
> >> continuing
```

Code block must be indented

Chapter Topics

Flow Control

- Code Blocks
- Repetitive Execution
- Iterative Execution
- Conditional Execution
- Tentative Execution (Exception Handling)
- **Essential Points**
- Hands-On Exercise: Flow Control

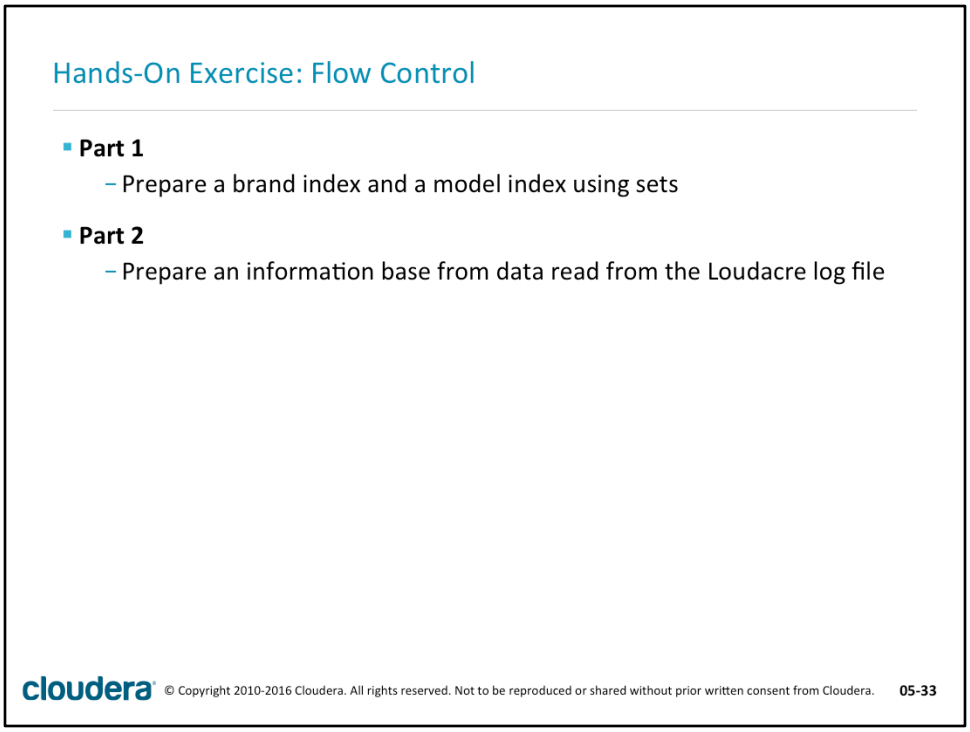
Essential Points

- **Code blocks**
  - Code flow in Python is dependent on indented code blocks
- **Conditional (`True`/`False`) execution using `if`, `elif`, and `else`**
  - Stack **`elif`'s** to simulate **`switch()`**…**`case`**:
- **`while` is used in conditional (`True`/`False`) looping**
- **`for` is used for repetition**
  - **`for in range()`**
  - **`for` *`element`* `in` *`collection`***
  - **`for` *`element`* `in enumerate(`*`collection`*`)`**
  - **`for` *`element`* `in zip(`*`collection1, collection2`*`)`**
- **`try` is used for exception handing**

Chapter Topics

Flow Control

- Code Blocks
- Repetitive Execution
- Iterative Execution
- Conditional Execution
- Tentative Execution (Exception Handling)
- Essential Points
- **Hands-On Exercise: Flow Control**

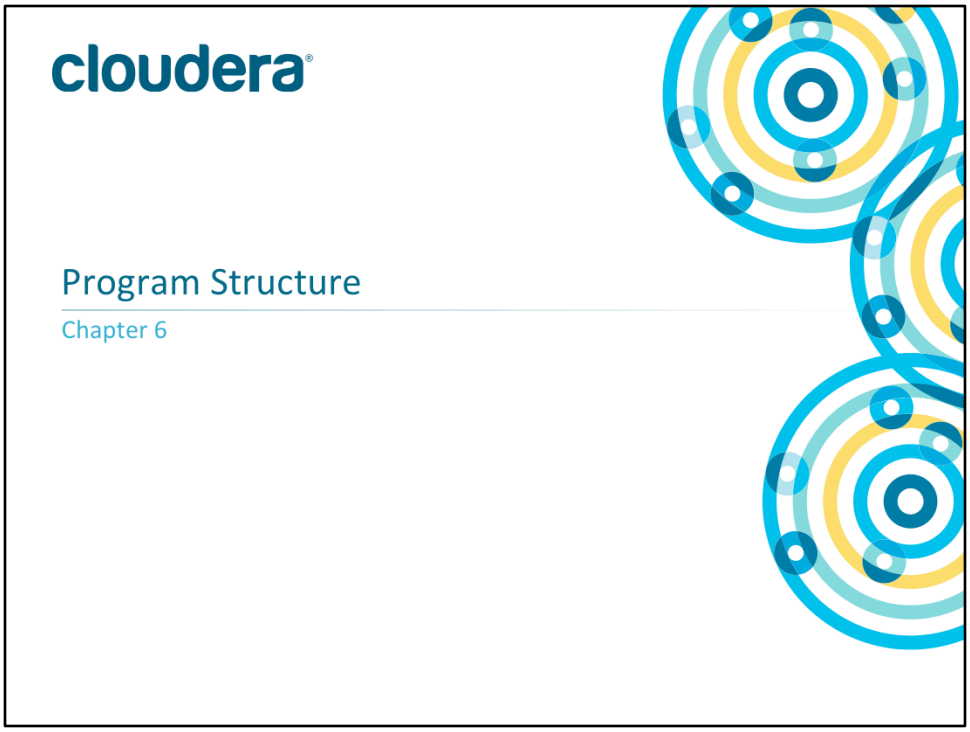


Part 1

Prepare a brand index and a model index using sets

Part 2

Prepare an information base from data read from the loudacre log file
Use **`split()`**, **`sorted()`**, and arithmetic, casting, and conditional flow to prepare and process the data
Cast the final information base to a tuple



Course Chapters

- Introduction
- Introduction to Python
- Variables
- Collections
- Flow Control
- **Program Structure**
- Working with Libraries
- Conclusion

Program Structure

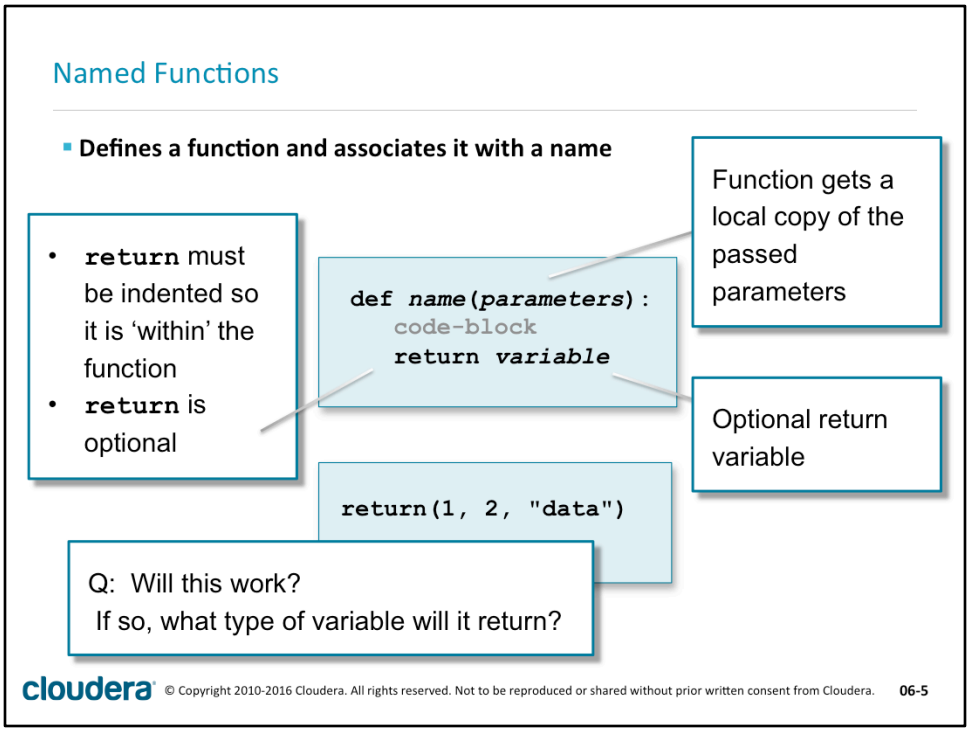
In this chapter you will learn

- **How to create and work with named functions including a basic understanding of function context and scoping**
- **How to create and work with anonymous functions (also known as Lambda functions)**
  - Note: Lambda functions are used extensively to interface with Big Data services and software
- **How to pass one function to another function by reference**
- **The purpose and general use of Python generators**

Chapter Topics

Program Structure

- **Named Functions**
- Anonymous Functions (Lambda)
- Generator Functions
- Essential Points
- Hands-On Exercise: Program Structure



[01] You create a function in Python using the keyword **def**. Parameters are passed to the function by copying, so if you modify the value of a parameter, only the local copy is altered, not the source in the calling code.
[02] The function is delimited by the indented code block. You do not need a **return,** and it does not terminate the function.
[03] A single optional variable can be returned.

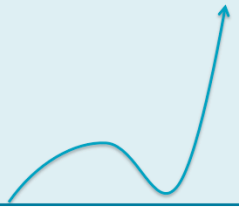
Note: *if not specified, functions return <type 'NoneType'>*

[04] Will it work? Yes. That is correct syntax, even without a space between 'return' and the variable. But how does this work if only a single variable is returned?
The answer is that the parenthesis define a tuple. So the syntax shown will return a Tuple with elements 1,2, and "data".
So although Python only returns one variable, that can be a collection, so you can pass back multiple results.

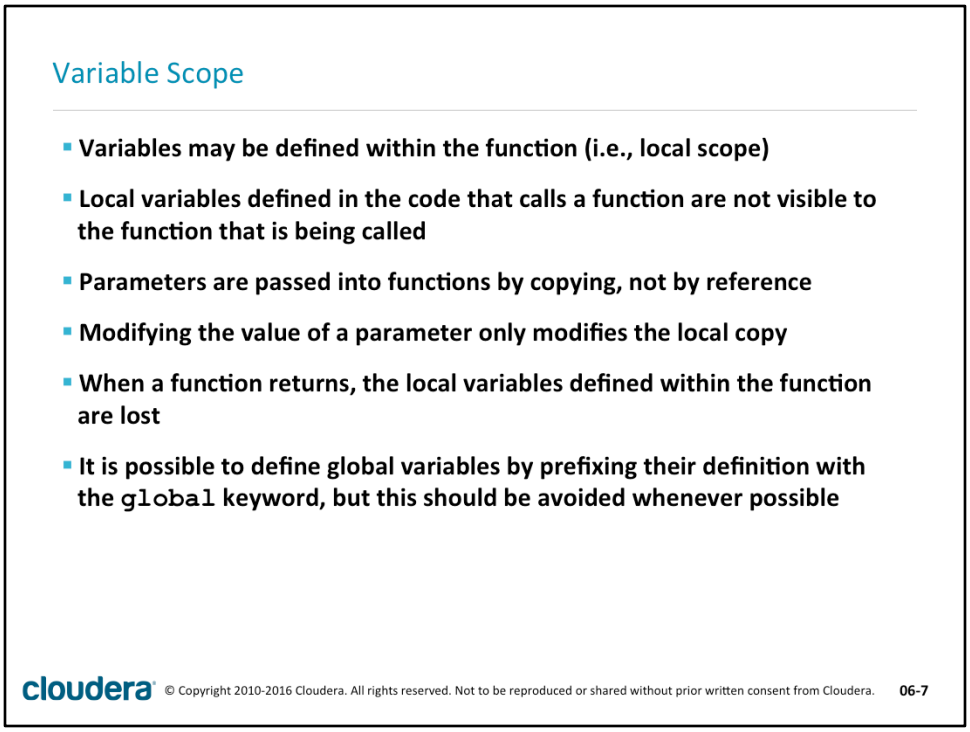
Note:
Most languages treat return values as a stack and pass back a single value.
Students familiar with other languages might expect to pass back a memory address (pointer) to a base location in memory (a structure) containing multiple variables.

Example of Named Functions

```
# Define function to return month from date time stamp
#
def getmonth(dts):
   months={1:'JAN',2:'FEB',3:'MAR',4:'APR',5:'MAY',    \
        6:'JUN',7:'JUL',8:'AUG',9:'SEP',10:'OCT',      \
        11:'NOV',12:'DEC'}
   month = dts[5:7]
   nummonth = int(month)
   return months[nummonth]

dts="2014-03-15:10:10:20"
print getmonth(dts)

> MAR
```



Continuation marks used for formatting of slide. The marked lines can be entered as a single line of code or use continuation marks for readability.



[05] All variables within the function are local and do not change anything outside of the function.

[06] The variables defined within the function are called the 'function context'. A new function context is created every time a function is called. By default, this 'function context' is queued for recycling when a function completes execution.

[07] You can override the default behavior of an individual variable using the **global** keyword. This will cause Python to use the variable from the calling context. And changes to the value of that variable will survive the end of the function execution and the recycling of the function context. The use of **global** is generally regarded as poor coding practice among Python developers.

Note:
The following two slides are a code walk-through showing execution of code and the resulting scoping behavior. We have included this code walk-through because scoping can be confusing to students who have not encountered it before.

Scope

- **Definitions**
  - **a** is a passed parameter
  - **b** is a global variable
  - **c** is a local variable
- **Changes**
  - In the call to **fun1()** all three variables are incremented
- **Results**
  - What will happen in the main program after the call to **fun1()**
  - *What will the result of the print statements be?*

```
# -- function definition
def fun1(a):
    global b
    c = 1
    # -- change of values
    a += 1   # passed parameter
    b += 1   # global variable
    c += 1   # local variable
    return a

# -- main
a = 7
b = 3
d = fun1(a)

print a  # What will this do?
print b  # What will this do?
print c  # What will this do?
```


Answer to "a"

```
# -- function definition
def fun1(a):
    global b
    c = 1
    # -- change of values
    a += 1    # passed parameter
    b += 1    # global variable
    c += 1    # local variable

# -- main
a = 7
b = 3
fun1(a)

print a
print b
print c
```

- a is 7
- When a is passed into `fun1()`, `fun1` gets a local copy of a
- Local-a has the same value as the a in main, 7
- Local-a is incremented to 8
- When `fun1()` ends, and control returns to main, local-a is deleted
- The `print a` statement prints the value of global-a, 7, and not local-a, which no longer exists

Answer to "b"

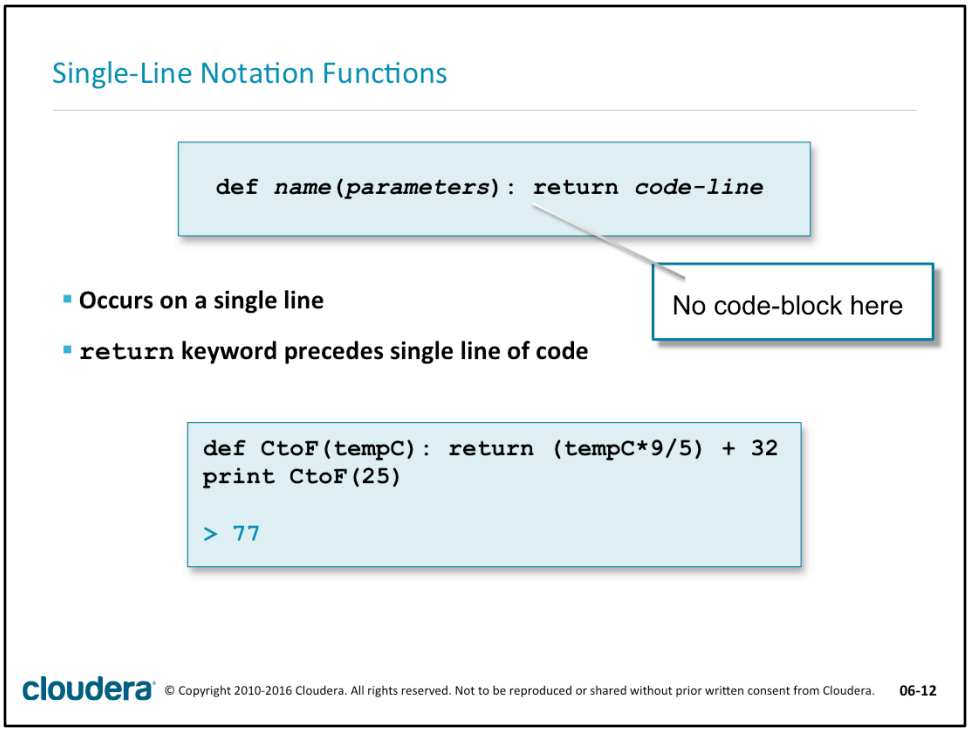
```
# -- function definition
def fun1(a):
    global b
    c = 1
    # -- change of values
    a += 1    # passed parameter
    b += 1    # global variable
    c += 1    # local variable

# -- main
a = 7
b = 3
fun1(a)

print a
print b
print c
```

- b is 3
- The statement `global b` in `fun1()` makes global-b accessible from within the function
- When b is incremented in `fun1()`, it is global-b
- The `print b` statement prints the value of global-b which is now 4

Answer to "c"

```
# -- function definition
def fun1(a):
    global b
    c = 1
    # -- change of values
    a += 1    # passed parameter
    b += 1    # global variable
    c += 1    # local variable

# -- main
a = 7
b = 3
fun1(a)

print a
print b
print c
```

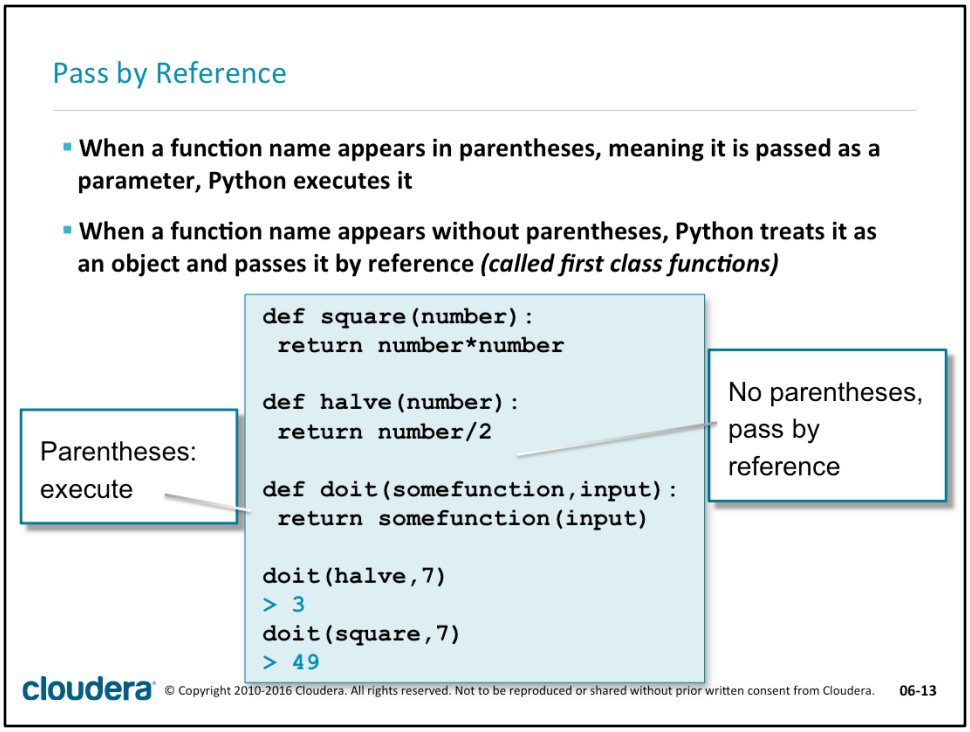
- c does not exist in main
- In `fun1()`, a local variable c is created and given a value of 1
- Local-c is incremented to 2
- When `fun1()` ends, and control returns to main, local-c is deleted
- The `print c` statement causes an `<Error>`, because in the global context c never existed



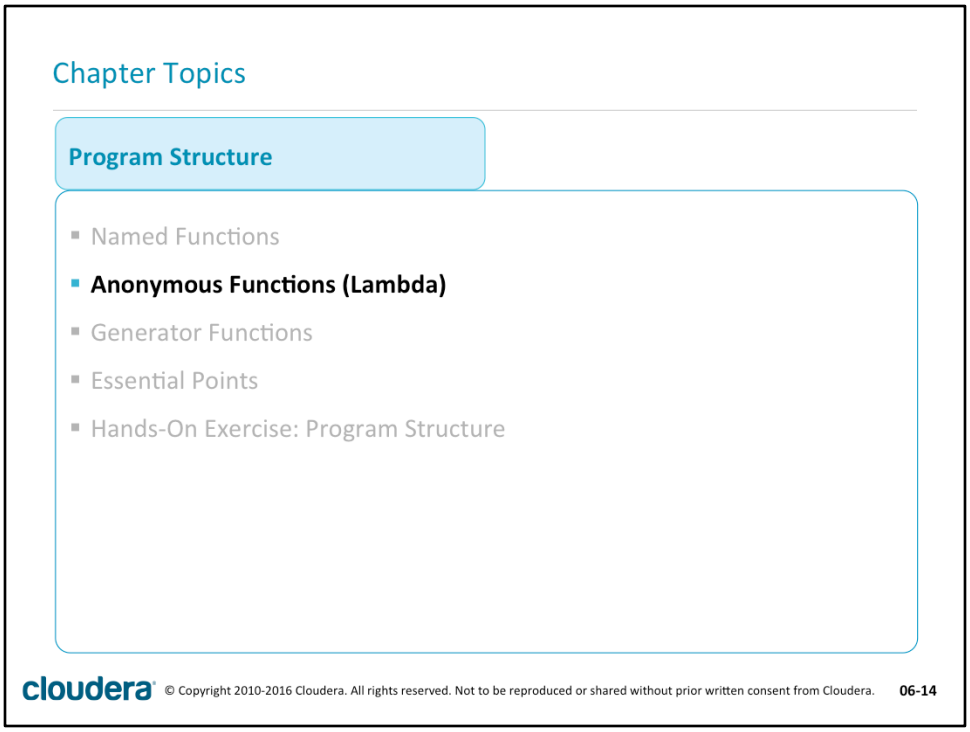
[08] The single line notation for a function places the main code inside the domain of the **return** keyword (instead of preceding it).

These two definitions result in identical functions:

```
def myfun1 (ab):
    temp = ab** .5
    return temp

def myfun1 (ab) : return ab** .5
```



Note: In version 2.6.6 I will return '3', but in version 2.7.x it will return 3.5



Note:

Lambda functions, that are covered next, are critical to our Spark Developer class. In Spark you create lambda functions and assign them to the spark context (sc), which is how the functions are exported to be executed on the servers in the cluster.

Anonymous Functions

```
lambda parameters: code-line
```

- **Enables a function to be defined inline without a symbolic name**
  - Defined within the call to another function
  - Useful for one-time functions where reuse is not an objective
  - Limited to a single line – tiny functions
  - Code must implicitly return a value

filter()

- **Iterates over each element in the collection and returns only those elements that match the criteria determined by the function**
  - The code below simulates **filter**'s basic operation
  - An example using **filter** is on the next slide

```
filter(test_function, collection)
```

```
def myfilter(function, collection):
   newlist=[]
   for each in collection[1:]:
       if function(each):
          newlist.append(each)
   return newlist
```


Example `filter()`

- **Anonymous function in call to `filter()`**

```
# -- display only names fewer than 6 characters
#

devices = ['iFruit','Sorrento','Titanic','Ronin','MeToo']

filtered_devices = filter(lambda x: len(x)<6, devices)
print filtered_devices

> ['Ronin', 'MeToo']
```

map()

- **Iterates over each element in the collection and returns 1-for-1 elements that are transformed by the function**
  - The code below simulates **map**'s basic operation
  - An example using **map** is on the next slide

```
map(transform_function, collection)
```

```
def mymap(function, collection):
   newlist=[]
   for each in collection[1:]:
       newlist.append(function(each))
   return newlist
```


Example `map()`

- **Anonymous function in call to `map()`**

```
# -- CPU utilization for devices was read as strings
#    Convert to numeric, and turn them into percentages
#

cpu_util= ['32','46','18','76','23','49','91','87']

cpu_percent = map(lambda c: float(c)/100.0, cpu_util)
print cpu_percent

> [0.32000000000000001, 0.46000000000000002,
0.17999999999999999, 0.76000000000000001,
0.23000000000000001, 0.48999999999999999,
0.91000000000000003, 0.87]
```


Note: The above output is from Python 2.6.6 loaded on our VM.
The above output in Python 2.7 would look like this:

```
> [0.32, 0.46, 0.18, 0.76, 0.23, 0.49, 0.91, 0.87]
```

Second Example of `map()`

- You can pass `map()` multiple collections of the same size

```
# -- Find the absolute difference between ambient
#    temperature and device temperature
#

ambient = [21,26,54,87,69,77,61,47,81,49,37]
device  = [86,97,53,98,31,61,71,70,92,98,58]

variation = map(lambda x,y: abs(x-y), ambient,device)
print variation

> [65, 71, 1, 11, 38, 16, 10, 23, 11, 49, 21]
```

reduce()

- **Iterates over a sequence of elements in the collection and performs a function on the sequence**
  - The code below simulates **reduce's** basic operation
  - An example using **reduce** is on the next slide

```
reduce(reduce_function, collection)
```

```
def myreduce(function, collection):
  tally = collection[0]
  for next in collection[1:]:
      tally = function(tally, next)
  return tally
```


Example `reduce()`

- **Anonymous function in call to `reduce()`**

```
# -- restarts is a dictionary the contains number of
#    reported device restarts by brand name
#    This code uses reduce() to calculate the total
#    number of restarts on all devices
#

restarts = {'MeToo':7,'Ronin':7,'Sorrento':15,      \
'Titanic':11, 'iFruit':7}

total = reduce(lambda x,y: x+y, restarts.values())
print total

> 47
```


Chapter Topics

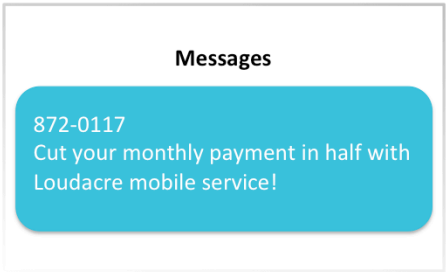
Program Structure

- Named Functions
- Anonymous Functions (Lambda)
- **Generator Functions**
- Essential Points
- Hands-On Exercise: Program Structure

Problem #1

Loudacre mobile

- **Loudacre marketing wants to send an advertising text message to every telephone in the 555 area code.**
- **First challenge is to write a program to generate all the phone numbers**
  - 555-*eee*-*ssss* = 3 + 1 + 3 + 1 + 4 = Each phone is a 12 character string
  - Skip zeroes in the exchange field
  - *aaa*-123-4567 = 10,000,000 = *about* 10 million numbers
  - 12 x 10m = 120 MB of data



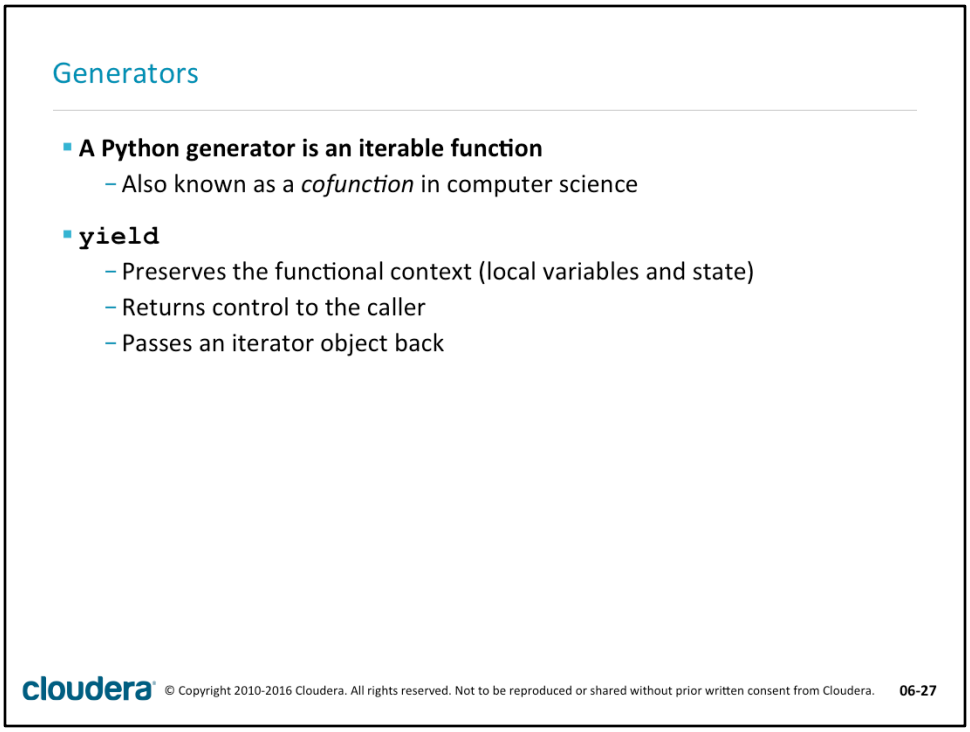
phonegen()

- Generates the phone numbers in the 555 area code in sequence skipping leading zeroes in the exchange digits
- Returns a list of the numbers generated
- This code takes about 30 seconds to run on the target platform

```
def phonegen():
 d1=d2=d3=d4=d5=d6=d7 = 0
 phonelist = []
 for d1 in range(1,10):
  for d2 in range(1,10):
   for d3 in range(1,10):
    for d4 in range(0,10):
     for d5 in range(0,10):
      for d6 in range(0,10):
       for d7 in range(0,10):
        phone = '555'+'-'+str(d1)+ \
str(d2)+str(d3)
        phone = phone+'-'+str(d4)+ \
str(d5)+str(d6)+str(d7)
        phonelist.append(phone)
 return phonelist

# -- test program
list_of_phones = phonegen()
for each_phone in list_of_phones:
#    send_message(each_phone)
     print each_phone
```


Problem #2

- **The advertising program was a success! Marketing wants to expand to cover 250 area codes in the United States.**
- **Concerns**
  - 250 area codes x 120 MB each = 30 GB of data
    - *My computer only has 16 GB of RAM!*
  - The previous program took about 30 seconds to generate the list
    - *The new program will take about two hours to generate the list!*
- **What's needed is a way to iterate over each phone number in sequence**
- **But how do you iterate with a function?**
  - Local variables are lost when the function returns
  - How would you preserve the current state so you could continue at the next number in sequence each time you call `phonegen()`?



`next(functional_context)`

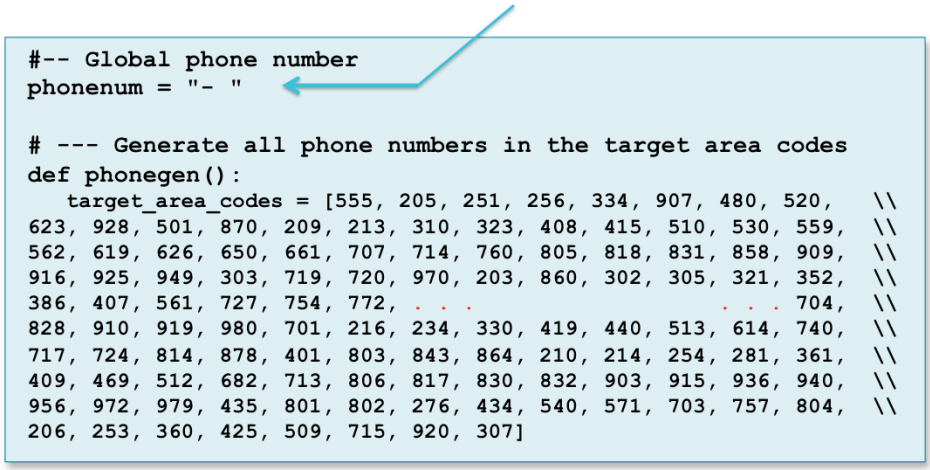
Resumes processing the function at the point where it was suspended

Note:

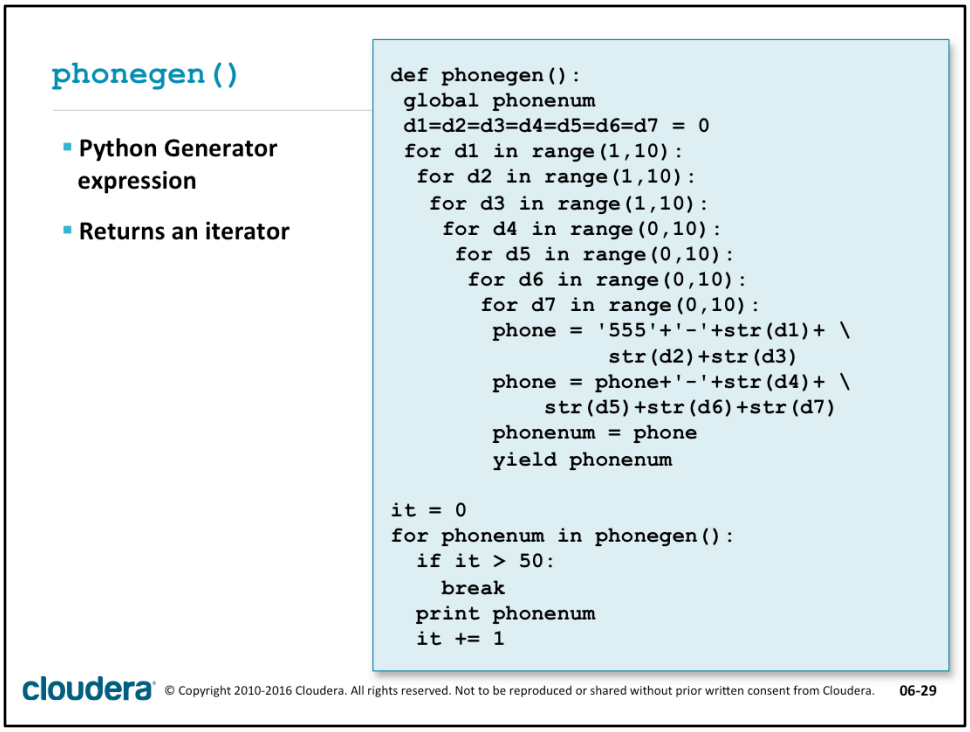
Note that the yield in Python works differently than in Scala. In Python, "yield" returns a generator (similar to Scala's iterator). In Scala, it returns a materialized collection (which could be used to create an iterator, as covered in the JES class.)

Area Codes for `phonegen()`

- Yield will pass back functional context. So a new way to pass the phone number back to the calling code is needed



```
#-- Global phone number
phonenum = "- "

# --- Generate all phone numbers in the target area codes
def phonegen():
   target_area_codes = [555, 205, 251, 256, 334, 907, 480, 520,   \\
623, 928, 501, 870, 209, 213, 310, 323, 408, 415, 510, 530, 559,  \\
562, 619, 626, 650, 661, 707, 714, 760, 805, 818, 831, 858, 909,  \\
916, 925, 949, 303, 719, 720, 970, 203, 860, 302, 305, 321, 352,  \\
386, 407, 561, 727, 754, 772, . . .                    . . . 704,  \\
828, 910, 919, 980, 701, 216, 234, 330, 419, 440, 513, 614, 740,  \\
717, 724, 814, 878, 401, 803, 843, 864, 210, 214, 254, 281, 361,  \\
409, 469, 512, 682, 713, 806, 817, 830, 832, 903, 915, 936, 940,  \\
956, 972, 979, 435, 801, 802, 276, 434, 540, 571, 703, 757, 804,  \\
206, 253, 360, 425, 509, 715, 920, 307]
```



Python Generator passes back the function as an iterator.

Note: This covers the common use of a Python Generator and is considered 'Pythonic' code.

Chapter Topics

Program Structure

- Named Functions
- Anonymous Functions (Lambda)
- Generator Functions
- **Essential Points**
- Hands-On Exercise: Program Structure

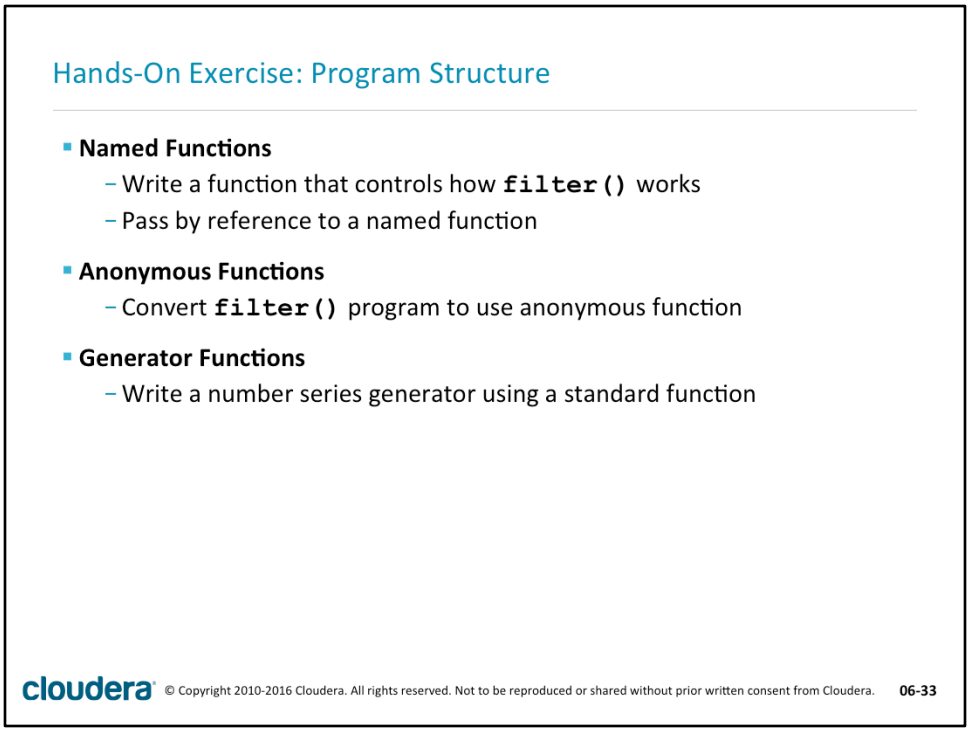
Essential Points

- **Named functions**
  - Call by reference
- **Variable scoping and the function context**
  - Global
- **Anonymous (lambda) functions**
  - Commonly used in: `filter()`, `map()`, `reduce()`
- **Generator functions**
  - `yield`

Chapter Topics

Program Structure

- Named Functions
- Anonymous Functions (Lambda)
- Generator Functions
- Essential Points
- **Hands-On Exercise: Program Structure**



Named Functions

Write a function that controls how **`filter()`** works

Pass by Reference to a named function

Anonymous Functions

Convert **`filter()`** program to use anonymous function

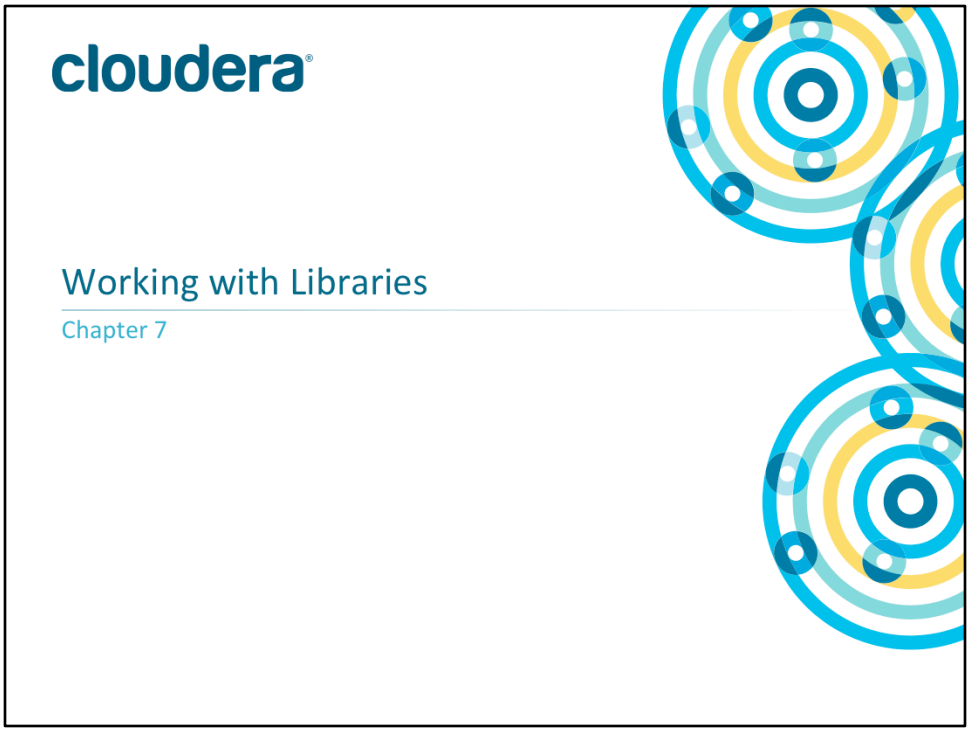
Use Python **`map()`** to prepare temperature data (**`string`** to **`float`**, degrees Celsius to degrees Fahrenheit)

Use Python **`reduce()`** to find the hottest device

Generator Functions

Write a number series generator using a standard function

Convert the series generator for scalability using **`yield`** and **`next()`**



Course Chapters

- Introduction
- Introduction to Python
- Variables
- Collections
- Flow Control
- Program Structure
- **Working with Libraries**
- Conclusion

Working with Libraries

In this chapter you will learn

- **How to organize code into separate modules**
- **How to gain and control access to modular code**
- **What is offered in common standard libraries**

Chapter Topics

Working with Libraries

- **Storing and Retrieving Functions**
- Module Control
- Common Standard Libraries
- Essential Points
- Hands-On Exercise: Libraries

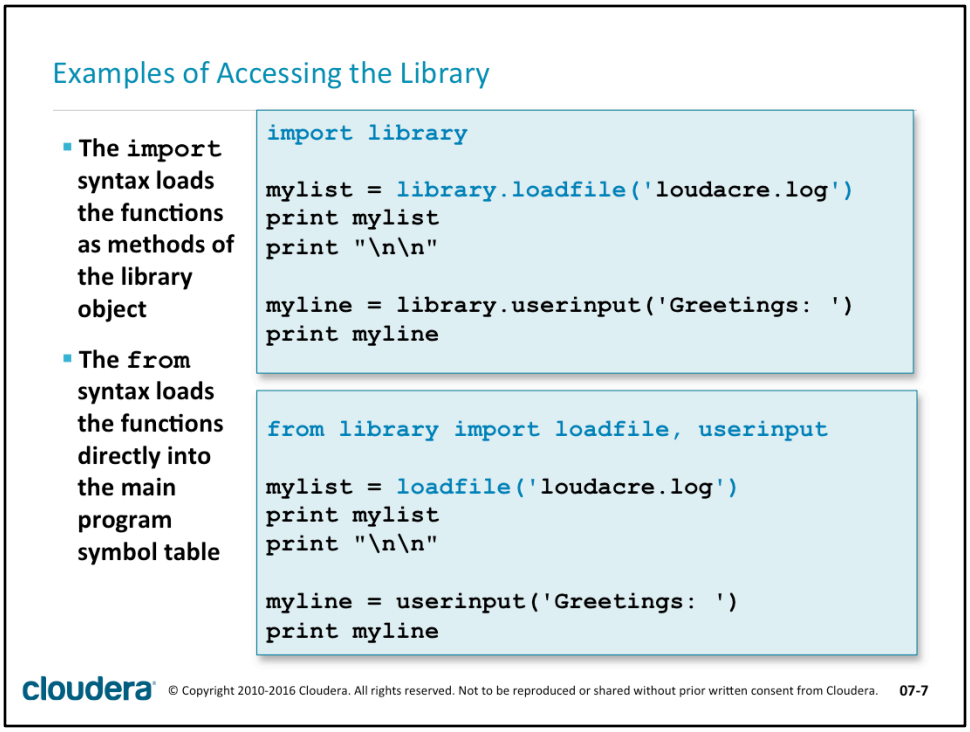
Python Modules

- Also called a "library"
- The program that is loaded into Python from the command line is the main program
  - `$ python program.py`
- Load code from other `*.py` files into the main program
  - `import libraryname`
    - Searches the path for `libraryname.py`
    - Loads the library as an object, and all functions as methods
    - To access `fun1()` in *`libraryname`*, use *`libraryname`*`.fun1()`
  - `from libraryname import fun1, fun2, fun3`
    - Loads the functions directly into the main program
    - To access `fun1()`, use `fun1()`

Example Library

File: `library.py`

```
def loadfile(filename):
  lines=[ ]

  file = open(filename,'rt')
  lines = file.readlines();
  file.close()
  return lines

def userinput(prompt):
   keybuffer = raw_input(prompt)
   return keybuffer
```



The "import" syntax is considered easier to debug and maintain, but it loads everything. If you only need a few functions and don't want to load a large library of code that won't be run, use the "from" syntax.

Note:

`from library import *`

This works but is considered bad practice.

Chapter Topics

Working with Libraries

- Storing and Retrieving Functions
- **Module Control**
- Common Standard Libraries
- Essential Points
- Hands-On Exercise: Libraries

How Does Python Find Libraries?

- **At Python startup**
  - Current directory where the program is located
  - **`PYTHONPATH`**
    - OS-dependent path
    - A list of directory names with the same syntax as **`PATH`** in the OS
    - Path to default libraries such as ***`prefix`*`/lib/python`*`version`***
- **After program is running**
  - The path is available and can be changed by a running program
  - It is located in **`sys.path`**
  - Add to path with **`sys.path.append(`*`newpath`*`)`**

Example of Adding to the Path

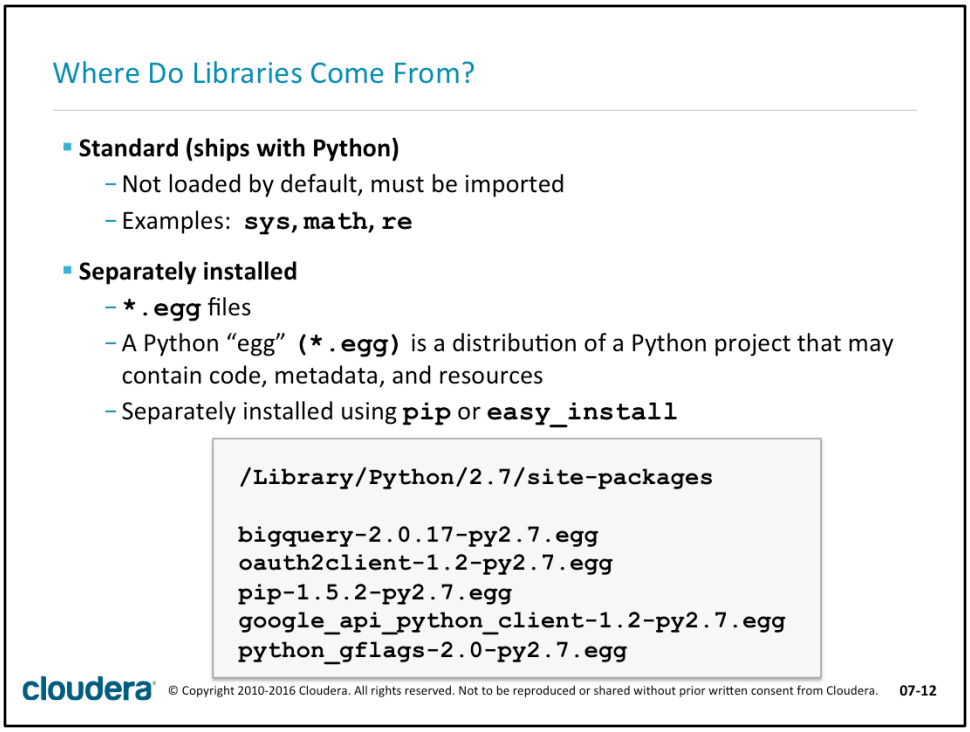
- **Adding a user-defined directory of libraries**

```
import sys
print sys.path

> ['',
 '/Library/Frameworks/Python.framework/Versions/2.7/bin',
 '/Library/Python/2.7/site-packages/bigquery-2.0.17-py2.7.egg',
 '/Library/Python/2.7/site-packages/httplib2-0.8-py2.7.egg',
 '/Library/Python/2.7/site-packages/oauth2client-1.2-py2.7.egg',
. . .and so forth  . . .

sys.path.append('/home/user/python-libs')
```

Site-Specific Paths

- **Provides a single location for maintenance of site-specific paths**
- **On import, `site` extends `sys.path` with `sys.prefix` and `sys.exec_prefix`**
  - The exact files/paths are OS-dependent
    - Example Unix: **`lib/python$version/site-packages`** and **`lib/site-python`**
  - You can globally set/change site-specific paths

```
import sys
print sys.path
print sys.prefix
print sys.exec_prefix

import site
print site.PREFIXES
print site.USER_BASE
print site.USER_SITE
```



sys is operating system
math is advanced math
re is RegEx – regular expressions

*.egg files are used to distribute program, similar to .jar files in Java
For example, a developer might create *.egg files to distribute programs to a cluster of machines.
Creating *.egg files is beyond the scope of this course.

What's in a Library?

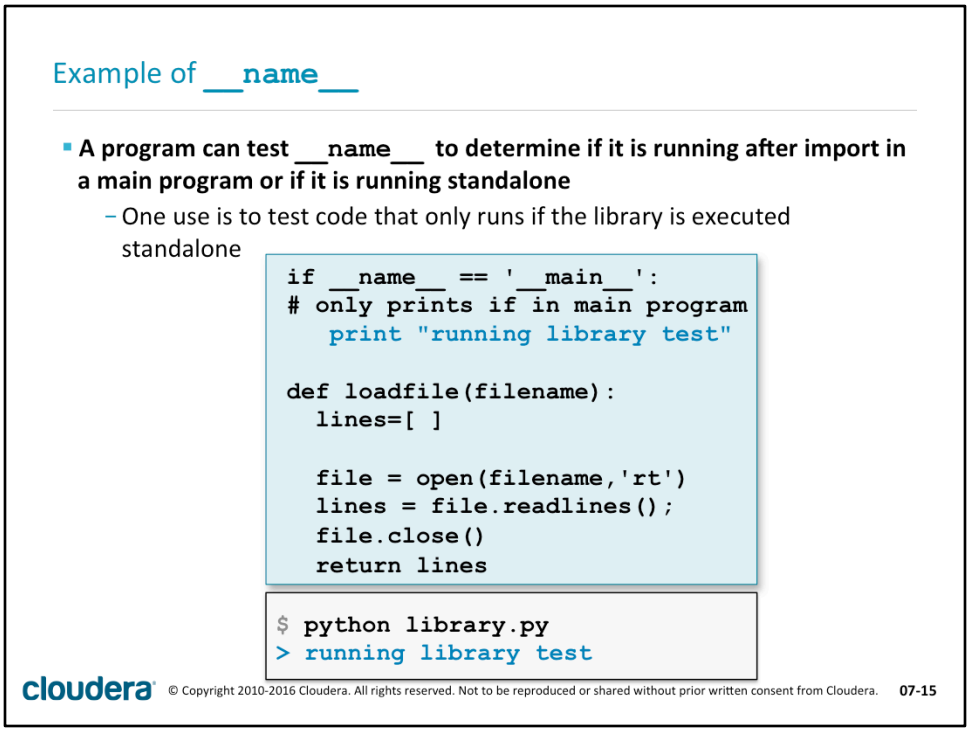
- **import *libraryname***
- **dir(*libraryname*)**

```
import math
dir(math)
> ['__doc__', '__file__', '__name__', '__package__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

Special Strings

- **__doc__ Contains a documentation string**
- **__name__**
  - Contains a string with the context in which the code is running, name of the library if imported, or **__main__**
- **__file__ Contains the path from which the library originated**

```
print math.__doc__
> 'This module is always available.  It provides access
to the\nmathematical functions defined by the C
standard.'
print math.__name__
> 'math'
print math.__file__
> '/Library/Frameworks/Python.framework/Versions/2.7/lib/
python2.7/lib-dynload/math.so'
```

__name__ can be used to determine whether code is being executed as a separate program or if it has been imported (as a library) into another program. Sometimes programs are written to be used both as independent 'standalone' programs and also to be imported as libraries. One common method is to use a condition as shown in this example around library code. When the code is run standalone, it executes a test program that verifies that the library is working properly and drives it's critical functions with sample or testing values. When the code is imported, the condition bypasses the test.

Passing Command Line Arguments

- **Arguments are passed from the command line using the sys module**
  - **`sys`** is a built-in; not enabled by default, you have to import it
  - **`import sys`**
- **Arguments on the command line are passed as space-delimited string**

```
$ python args.py 1 2 3
    → sys.argv[0]='1', sys.argv[1]='2', sys.argv[2]='3'

$ python args.py 1,2,3
    → sys.argv[0] ='1,2,3'
```

```
import sys

arguments = len(sys.argv)
for i in range(0,arguments):
  print "argv[",i,"] ", sys.argv[i]
```


Chapter Topics

Working with Libraries

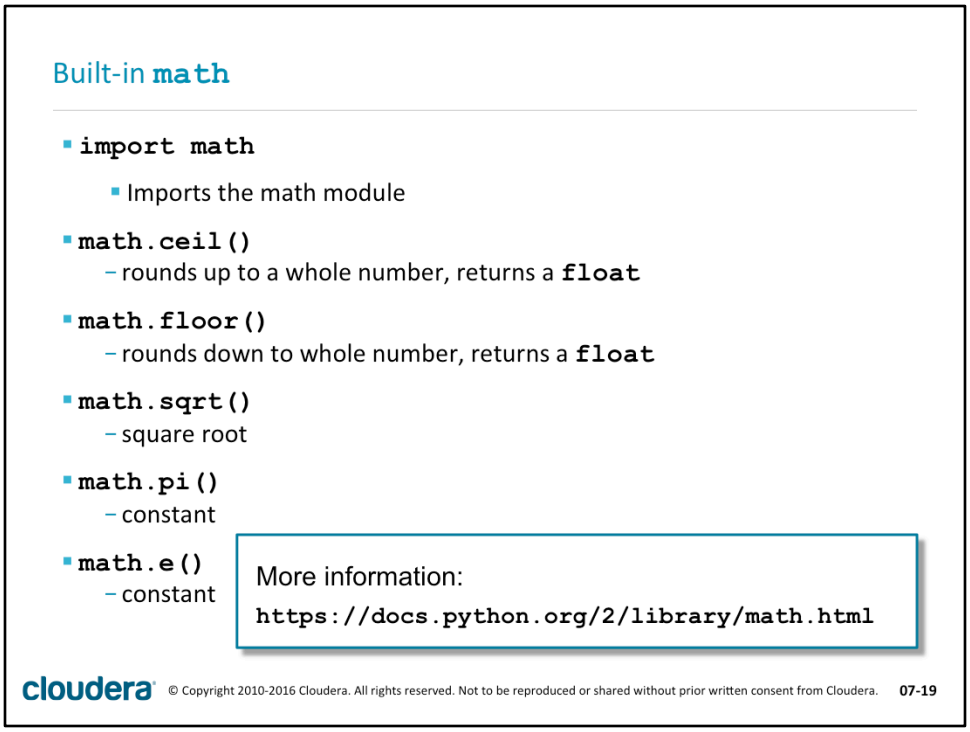
- Storing and Retrieving Functions
- Module Control
- **Common Standard Libraries**
- Essential Points
- Hands-On Exercise: Libraries

Built-in `sys`

- **`import sys`**
  - Imports system environment object
  - **`help(sys)`** → details on available values and methods
    - **`sys.version`**
      - Gives the version of Python

```
'2.7.6 (v2.7.6:3a1db0d2747e, Nov 10
2013, 00:42:54) \n[GCC 4.2.1 (Apple Inc.
build 5666) (dot 3)]'
```

    - **`sys.executable`**
      - Gives the path to the Python executable
    - **`sys.getrefcount(var)`**
      - Gives the refcount of a variable



This is a very high level overview of some functions and constants.

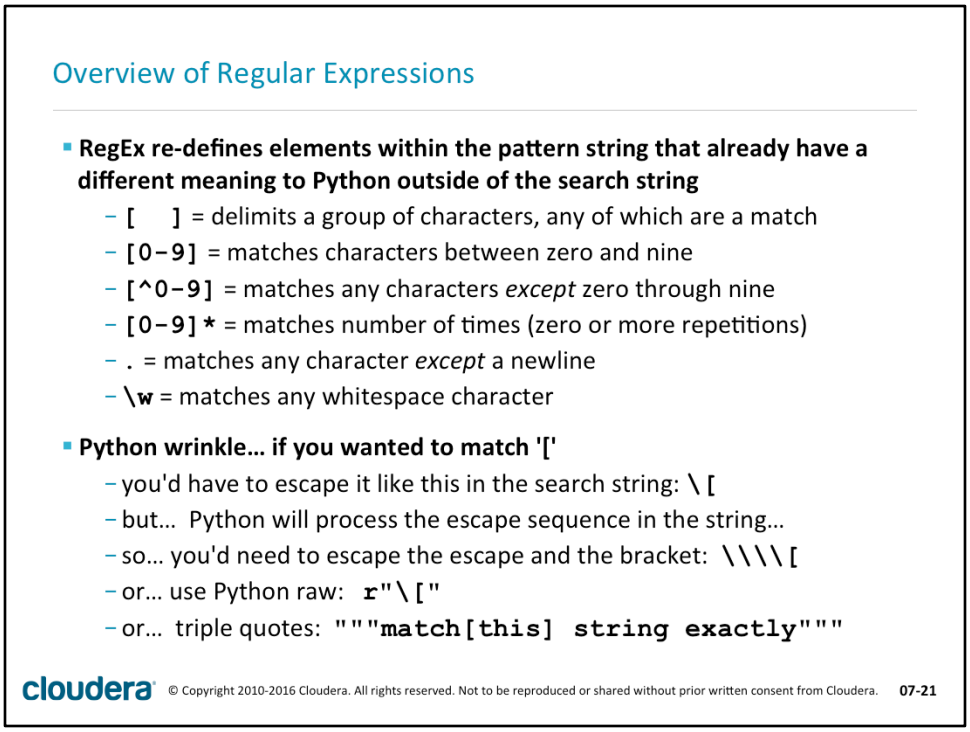
Note:
This is to serve as general awareness concerning math… 'more to learn here'. Refer interested students to the link at the bottom of the page.

Demonstration:

```
sqrt(16)        → Error
math.sqrt(16)   → Error
import math
math.sqrt(16)   → 4.0
```

Built-in `re`

- **`import re`**
  - Imports regular expression module for sophisticated pattern matching
  - **`re.compile(`*`pattern`*`)`**
    - Creates the **`pattern`** object
    - Loads and parses the regular expression search string
  - **`pattern.match(`*`content`*`)`**
    - creates the **`Match`** object
    - Executes the regular expression code and stores the results
    - You can then call methods on the pattern object to investigate
      - **`pattern.start(),pattern.end()`**
      - **`pattern.group(),pattern.span()`**



In Python, because strings have to be escaped with a backslash, and the same trick is used within Regular Expression strings, the result can be cumbersome multi-escape sequences. You can bypass Python's escape sequence processing using the r"…" (raw) modifier immediately before a string.

RegEx Example

```
import re

regobj = re.compile("[a-z]*")
print(type(regobj))
mat = regobj.match("abcdefghijk")

print(mat)
print(type(mat))
print
print(mat.start(), mat.end(), mat.group(), mat.span())

> <type '_sre.SRE_Pattern'>
> <_sre.SRE_Match object at 0x1006ae030>
> <type '_sre.SRE_Match'>

> (0, 11, 'abcdefghijk', (0, 11))
```


Chapter Topics

Working with Libraries

- Storing and Retrieving Functions
- Module Control
- Common Standard Libraries
- **Essential Points**
- Hands-On Exercise: Libraries

Essential Points

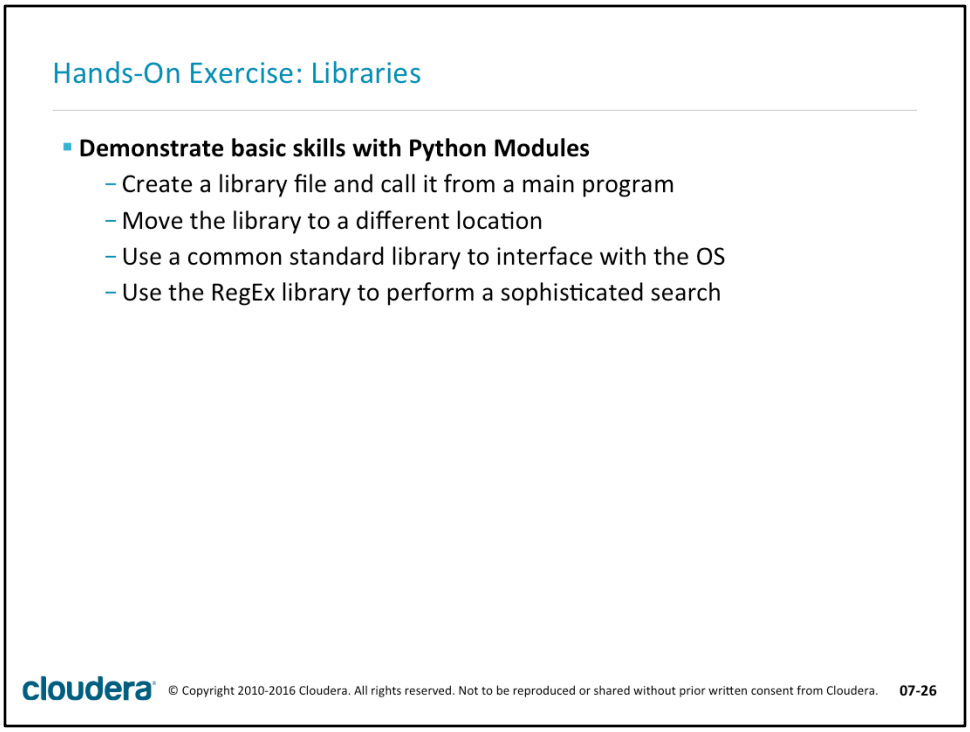
- **Creating and Using Libraries**
  - Method-calling semantics
    - `import library`
  - Function-calling semantics
    - `from library import functions`
- **Control**
  - Paths
  - Special strings
  - Pip and `*.egg`
- **Standard Libraries**
  - `sys`, `math`, `site`, `re`

Note: library 're' is Regular Expressions

Chapter Topics

Working with Libraries

- Storing and Retrieving Functions
- Module Control
- Common Standard Libraries
- Essential Points
- **Hands-On Exercise: Libraries**



Demonstrate basic skills with Python Modules

Create a library file and call it from a main program

`import` semantics

`from` semantics

Move the library to a different location

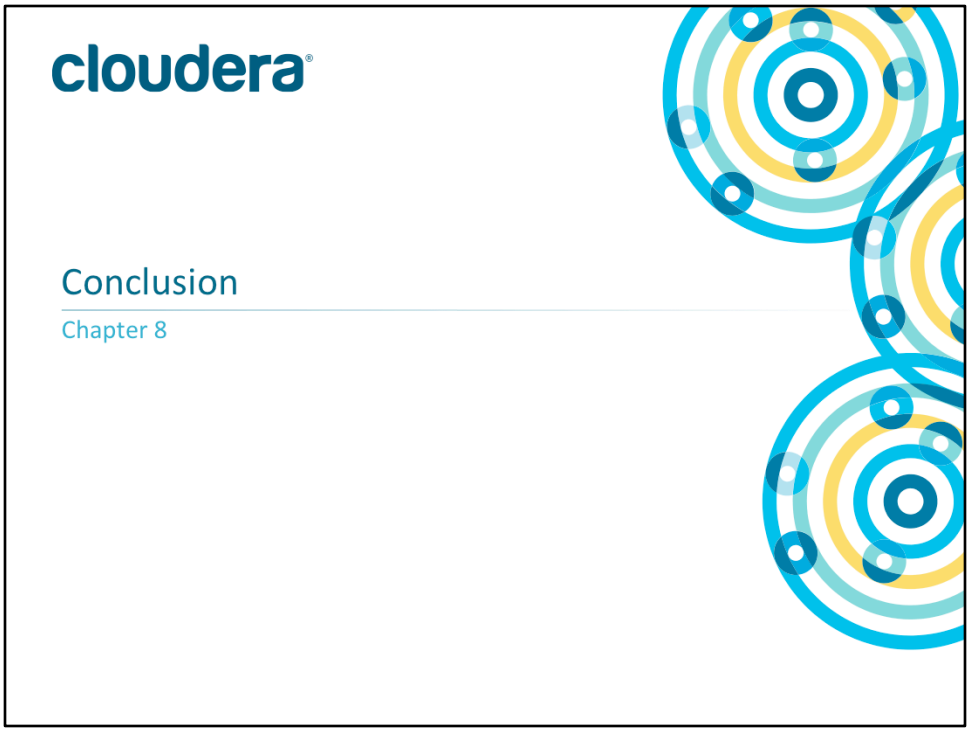
Update the Python search path so the library can be found

Use a common standard library to interface with the OS

Identify the number of arguments passed to the command line

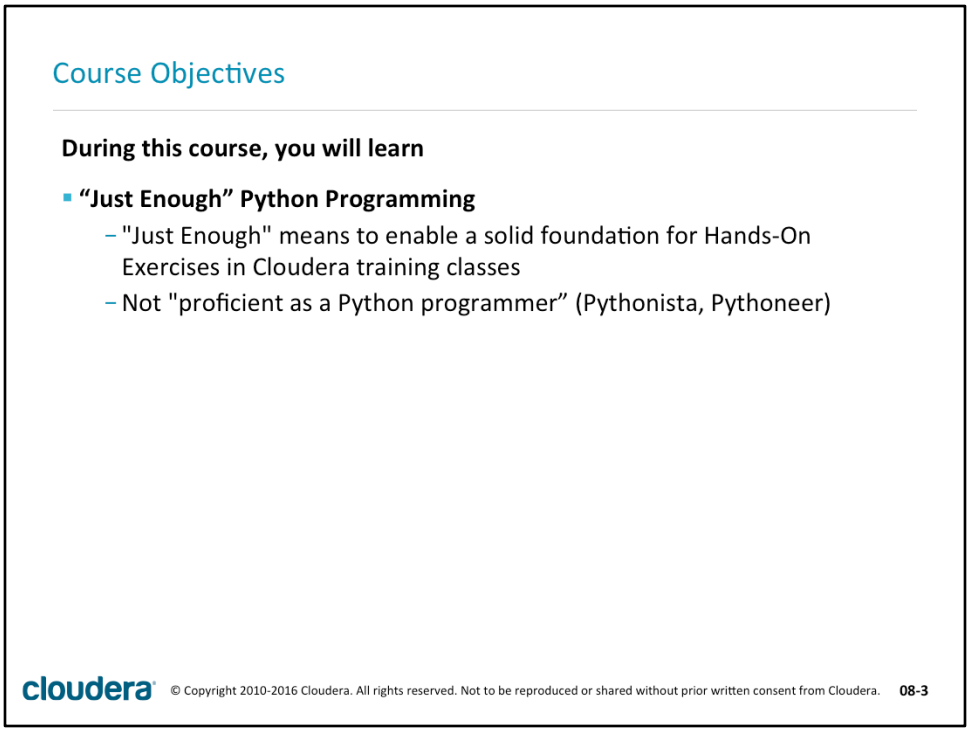
Read and display each argument

Use the RegEx library to perform a sophisticated search



Course Chapters

- Introduction
- Introduction to Python
- Variables
- Collections
- Flow Control
- Program Structure
- Working with Libraries
- **Conclusion**



This class very specifically prepares you for Cloudera classes as rapidly and directly as possible. The class is not trying to cram everything into a crash course, and it is not a goal for you to become some kind of expert Python Programmer. The goal is to learn "Just Enough" Python so that learning the language is not a distraction from learning to develop with Hadoop or Spark in later classes.

Which Course to Take Next?

Cloudera offers a range of training courses for you and your team

- **For developers**
  - *Cloudera Developer Training for Apache Spark and Hadoop*
  - *Designing and Building Big Data Applications*
- **For system administrators**
  - *Cloudera Administrator Training for Apache Hadoop*
- **For data analysts and data scientists**
  - *Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop*
  - *Data Science at Scale using Spark and Hadoop*
- **For architects, managers, CIOs, and CTOs**
  - *Cloudera Essentials for Apache Hadoop*

Just Enough Python: Hands-On Exercises

Instructor Guide

Please see the Hands-On Exercise Manual for exercise instructions. There are currently no additional instructor notes for the exercises in this course.