Just Enough Python
Introduction

Chapter 1
Course Chapters

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

Introduction

- About this Course
- About Cloudera
- Course Logistics
- Introductions
Course Objectives

During this course, you will learn

- “Just Enough” Python Programming
  - “Just Enough” means to enable a solid foundation for Hands-On Exercises in Cloudera Developer classes
  - Not “proficient as a Python programmer” (Pythonista, Pythoneer)
Audience Background

- **Students should possess the following prerequisites for this course**
  - Interest in one of Cloudera’s developer-oriented courses
  - *Some* programming experience
    - No specific language or level of experience
    - Familiarity with object-oriented programming concepts
    - Basic skills and vocabulary

- **The following are *not* required (and are not covered in this course)**
  - Experience with Cloudera products
  - Experience with Hadoop or Spark
  - Experience with data analytics
Chapter Topics

Introduction

- About this Course
- About Cloudera
- Course Logistics
- Introductions
About Cloudera (1)

- The leader in Apache Hadoop-based software and services
- Founded by Hadoop experts from Facebook, Yahoo, Google, and Oracle
- Provides support, consulting, training, and certification for Hadoop users
- Staff includes committers to virtually all Hadoop projects
- Many authors of industry standard books on Apache Hadoop projects
About Cloudera (2)

- Our customers include many key users of Hadoop
- We offer several public training courses, such as
  - Cloudera Developer Training for Apache Spark and Hadoop
  - Cloudera Administrator Training for Apache Hadoop
  - Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop
  - Designing and Building Big Data Applications
  - Data Science at Scale Using Spark and Hadoop
  - Cloudera Training for Apache HBase
- On-site and customized training is also available
**CDH (Cloudera’s Distribution including Apache Hadoop)**

- **100% open source, enterprise-ready distribution of Hadoop and related projects**
- **The most complete, tested, and widely-deployed distribution of Hadoop**
- **Integrates all the key Hadoop ecosystem projects**
- **Available as RPMs and Ubuntu, Debian, or SuSE packages, or as a tarball**

<table>
<thead>
<tr>
<th>Engines</th>
<th>SECURITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>BATCH PROCESSING</td>
<td>Filesystem HDFS</td>
</tr>
<tr>
<td>ANALYTIC SQL</td>
<td>Online NoSQL HBase</td>
</tr>
<tr>
<td>SEARCH ENGINE</td>
<td>STORAGE FOR ANY TYPE OF DATA</td>
</tr>
<tr>
<td>MACHINE LEARNING</td>
<td></td>
</tr>
<tr>
<td>STREAM PROCESSING</td>
<td></td>
</tr>
<tr>
<td>3rd PARTY APPS</td>
<td></td>
</tr>
</tbody>
</table>

**METADATA**

**WORKLOAD MANAGEMENT (YARN)**

**UNIFIED, ELASTIC, RESILIENT, SECURE (Sentry)**

**DATA INTEGRATION (Sqoop, Flume, NFS)**
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
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
Chapter Topics

Introduction

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

- Class start and finish times
- Lunch
- Breaks
- Restrooms
- Wi-Fi access
- Virtual machines

Your instructor will give you details on how to access the course materials and exercise instructions for the class
Chapter Topics

Introduction

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

- About your instructor
- About you
  - Where do you work? What do you do there?
  - Do you have experience with UNIX or Linux?
  - What programming languages have you used?
  - How much programming experience do you have?
  - What do you expect to gain from this course?
Course Chapters

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

Introduction to Python

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

- **General purpose high-level programming language**
  - Multi-paradigm (functional, object-oriented, procedural, logical)

- **Started by Guido van Rossum in 1989**
  - Version 1.0 released in 1994
  - Transferred to the Python Software Foundation in 2001

- **Design philosophy**
  - Readability
    - Explicit is better than implicit
    - Simple is better than complex, complex better than complicated
  - Fewer lines of code, easier coding
    - Dynamic typing
    - Automatic memory management
  - Comprehensive library
Chapter Topics

Introduction to Python

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

- **Python 2.6 / 2.7** *(not 3.x)*

- **Variables**
  - integer, float, boolean, string

- **Collections**
  - list[], tuple(), set{}, frozenset, dictionary

- **Flow Control**
  - Code blocks, if, if else, if elif, else
  - while, for in range, for in list, try

- **Program Structure**
  - Functions, anonymous functions, generators

- **Working with Libraries**
  - import, from import, sys, math
Out of Scope

- Random numbers
- Statistical or date/time functions
- Advanced string manipulation
- Serialization, JSON, or XML
- Time functions or timers
- GUIs or Web applications
- Database I/O or network I/O
- Python concurrency and process management
- Code debugging or prototyping facilities
- Application packaging and deployment
Chapter Topics

Introduction to Python

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

Example of the look of the IPython shell

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

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

Optional return variable
Formatting Conventions of Code Examples

- Code examples are shown on a pale blue background
- The code you enter is displayed without any prompt in black type
- The system response is shown with a > prompt and in blue type

```python
s = "Titanic 4000"
print s

> Titanic 4000
```

```python
       11: 'NOV', 12: 'DEC'}
```

Python has a 79-character line limit.
Continue lines explicitly with a single backslash.
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
Basic File I/O

- Use the built-in file object and its methods
  - Methods: `open()`, `read()`, `readline()`, and `close()`
  - File modes: `r`=read, `w`=write, `a`=append
  - File format: `b`=binary, `t`=text (default)

```python
line = ''
file = open('loudacre.log','rt')
while True:
    line = file.readline()
    if not line:
        break
    print line
file.close()
```
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`

```python
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

```python
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
Course Chapters

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

- **Basic I/O**
  - `raw_input()`
  - `print()`

- **Variable Names**
  - Initial letter, letters and numbers, case sensitive, underscore but no other special characters
  - Reserved words of the language, keywords, prohibited
  - `__` (double underscore) reserved for some built-in object methods
  - Convention:
    - Initial caps for classes, initial lowercase for instances

```python
yourVar = raw_input('yourPromptHere')
print(yourVar)
```
Preview of Variables

- **Basic Variables**
  - Integers
  - Floats
  - Strings
  - Booleans

- **Compound Variables**
  - Lists
  - Tuples
  - Sets
  - Frozensets
  - Dictionaries

Almost everything in Python is some kind of object or variable.

There are a few additional types available in Python such as a “bytearray” that are outside the scope of this course.
Properties of Python Variables

- **Dynamically Typed**
  - In other languages you need to declare variable type
  - In Python type is established on first use

- **Mutable versus Immutable**
  - Immutable – value cannot be changed after initial assignment
  - Only two collections (Tuple and Frozenset) are immutable

- **Re-assignable**
  - Python re-evaluates type with every assignment
    - In some other languages attempts to use the same variable name with a different type will cause an error

```
abc = 10
abc = 'hello'
```

Now `abc` is an integer
Now `abc` is a string

*Python doesn't care!*
Dynamic Typing in Python

- **Type is established by the initial value assigned to a variable name**
  - The type is inferred by context clues
  - You **must** give a variable an initial value or Python will not recognize the variable
  - No explicit type declaration
  - Type can be changed whenever an assignment is made

- **Python supports multiple assignment syntax**

  - **Transitive**
    - `ab = bc = 7`
      - `ab` and `bc` are both assigned 7
  - **Distributive**
    - `ab, bc, cd = 8, 10, 43`
      - `ab` is 8, `bc` is 10, and `cd` is 43
Automated Memory Management

- **Python provides automated memory management**
  - You don't have to “allocate” and remember to “free” memory to prevent a memory leak

- **Garbage Collection**
  - When a variable is no longer needed, the memory is released
  - Scope and reference counters determine whether a variable is needed
General Points about Python Variables

- Basic I/O is simple, in Python you don't need a lot of boilerplate code
- There are basic variables and collections built from the basic variables
- Understanding the variable types (especially collections) makes Python make sense and is crucial to writing “Pythonic” code
- You don't declare a variable, you initialize it
  - Python infers type by context
- Python re-evaluates and potentially changes the variable type on each assignment.
  - `type(var)` tells you a variable's current type vs. `print(var)` which tells you it's value
- Python automatically recycles memory when it is no longer being used
  - You don't have to `free()` variables
Chapter Topics

Variables

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

- **Precedence and Operations**
  - Follows mathematical operations
  - Uses ** instead of ^ for exponentiation
  - Supports reflexive operators
  - Does **not** support increment/decrement (ab++, ab--)
  - Provides floor division, a companion to the modulus operator

```
ab = 1 + 2 * 3
print(ab) > 7
```
```
ab += bc \rightarrow ab = ab + bc
```

<table>
<thead>
<tr>
<th>floor division operator</th>
<th>modulus</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 (\div) 5 = 1</td>
<td>7 %(5) = 2</td>
</tr>
</tbody>
</table>

The number of whole times 5 will “go into” 7

The partial remainder (i.e. 2/7ths remains)
Division Operations

- How should division be implemented in a dynamic typing language?
  - Two integer operands *could* return a floating point result (unlike in C)

- True Division *(The way division is handled in mathematics)*
  - Example: $3/2 = 1.5$ and $1/4 = 0.25$

- Classical Division *(The way C language implements division)*
  - If both operands are integers: $3/2 = 1$ and $1/4 = 0$ (truncation)
  - Floating example: $3.0/2.0 = 1.5$ and $1/4 = 0.25$

- Python 2.2 → Classical Division
  - If both operands are integers, results are truncated
  - Floor Division Operator ‘//’ — Example: $3.0//2.0 = 1$ and $1//4 = 0$
Numerical Interactions between Int and Float

- **Numbers**
  
  - Implicit transformation: int to float and float to int
  
  - Python handles mixing of int and float variables ... as expected

```python
# int
Dev_temp_celsius = 45
Fahrenheit = 9.0/5.0 * Dev_temp_celsius + 32
type(Fahrenheit)
> type <'float'>

# mixed int and float
Dev_temp = 37
fTemp = float(Dev_temp)
print fTemp
> 37.0

# int
Latitude = 33.19135811
iLat = int(Latitude)
print iLat
> 33
```

- Explicit transformation (casting):
  
  - `int(float_var), float(int_var)`
Numerical Interactions with Strings

- **Strings**
  - There is no `bin`, `oct`, or `hex` type in Python
  - `hex(num), oct(num), bin(num)` all return type `str` strings

```
Unique_Device_ID = "ff375011-34f0-4758-bade-e68cea787115"
print(hex(13552))
> "0x34f0"
```

- Strings are **not** implicitly converted to numeric variables
- Explicit transformation functions provided
  - `str(), hex(), oct(), bin()`
  - `int(), float() ← works appropriately for string type`

```
ab = "123.45"
bc = float(ab)
print bc
> 123.45

ab = "123.45"
bc = int(ab)
> <Error>
```

The string must be in the proper format...
Numbers to String Transformation Examples

```
a = bin(13552)
type(a)
> <type 'str'>
print a
> '0b11010011110000'
a = oct(13552)
type(a)
> <type 'str'>
print a
> '032360'
a = hex(13552)
type(a)
> <type 'str'>
print a
> '0x34f0'
```

```
print(0x79 + 1)
> 122
```

Python recognizes the notation and converts it to a decimal integer before performing the math operations.
Math Library

- **Floating Point Math Accuracy**
  - Accuracy is adaptive to the platform on which Python is running
  - Warning! Don't code with reliance on floating point accuracy

- *If you need better controls, take a look at the math module*

- **The Math Library**
  - The math library is not built-in as it is in other languages, it must be loaded
  - Calling semantics

```
import math

yourVar = math.function(yourParameters)
```

Usually returns a floating point

Function of your choice
Chapter Topics

Variables

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

- **bool** variables are used to control program flow
  - branching, conditional execution, looping
  ... more in the chapter on flow control...

- **bool** variables are created by assigning a **True** or **False** value
  - **True** and **False** are *case sensitive* and *are not delimited*

```
GPS_status = True
Creates a bool with the value True

GPS_status = 'True'
Creates a string of characters T-r-u-e

GPS_status = true
 Causes an error because variable named true has not been initialized
 Same for **TRUE**
```
Comparatives

- **Inequalities**
  - `<`, `>`, `<=`, `>=`
  - In Python, `=>` and `=<` are syntax errors

- **Equalities**
  - `==`, `!=`
  - Both `<>` and `!=` are valid
  - You can mix types, so `1 == 1.0` results in `True`

- **Logic**
  - `and`, `or`, `not`

- **Identities**
  - Compares the identity `id(var)` to see if symbolic names refer to the same exact object in memory
  - `is`, `is not`

```python
a = b = 7
print id(a) > 4298189096
print id(b) > 4298189096
print a is b > True

a and b are two different names for the same object in memory
```
Chapter Topics

Variables

- Python Variables
- Numerical
- Boolean

String

- Essential Points
- Hands-On Exercise: Variables
Strings

- **Delimiters**
  - Single-quote or double-quote
  - The start and end delimiter must be the same character
  - Curly quotes and slanted quotes are not valid delimiters

- **Composition**
  - Use the escape sequence (backslash) for literal characters
  - \t = tab, \n = newline, \b = backspace, \r = carriage return

```
ab = "Python\text"  →  Python  ext
```
- Use the r (raw) prefix for a literal string

```
ab = r'Python\text'  →  Python\text
```
- Use double-escape to indicate a single backslash

```
ab = 'Python\\text'  →  Python\text
```
String Manipulation: Concatenation

- **Concatenation**
  - Use the + plus operator
  - `join(sequence)`

```python
Style, Model = 'Sorrento', 'F10L'
Device_Name_and_Number = Style + " " + Model
print Device_Name_and_Number
> Sorrento F10L

StyMod = Style.join(Model)
print StyMod
> FSorrento1Sorrento0SorrentoL
```
String Manipulation: Slices

- **Slice**
  - A substring consisting of an individual character is called a “slice”
  - \texttt{string[offset]}
  - The offset is zero-based, so the first character is \texttt{str[0]}

- **Slice Range**
  - A substring consisting of multiple characters
  - \texttt{string[start:end]}
  - \textbf{Warning! upper bound is non-inclusive}
    - The “end” character is \textbf{not} included in the substring

```
device = 'MeeToo 4.1'
print device[1:5]  # eeTo
```
### String Transformation

#### Case Manipulation

- `string.title()`
- `string.capitalize()`
- `string.lower()`
- `string.upper()`
- `string.swapcase()`

#### Space Manipulation

- `string.strip(chars)`
- `string.rstrip(chars)`
- `string.lstrip(chars)`
- `string.expandtabs(tabsize)`

#### Justification

- `string.ljust(width, fillchar)`
- `string.rjust(width, fillchar)`
- `string.center(width, fillchar)`

#### Editing

- `string.split('delimiter')`
- `string.partition('char')`
- `string.rpartition('char')`
- `string.splitlines(integer)`
- `string.replace(str1, str2)`
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

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

Format

```
string.isalpha()
string.isdigit()
string.islower()
string.isspace()
string.istitle()
string.isupper()
```

Constitution

```
string1.startswith(string2)
string1.endswith(string2)
string1.count(string2)
string1.find(string2)
len(string)
```
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
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
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
## Collections

- Collection types are powerful and can influence program design.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Mutable</th>
<th>Ordered</th>
<th>Unique</th>
<th>Paired</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>A serial collection of objects.</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>list.sort()</code> → ordered collection</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuple</td>
<td>An immutable collection of objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>An unordered collection of unique objects</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sorted(set)</code> → ordered and unique</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Frozenset</td>
<td>An immutable ordered collection of unique objects</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>An ordered collection of unique key-value object pairs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Collections Syntax

Collections

list[element1, element2, element3]
tuple(element1, element2, element3)
set{element1, element2, element3}
dictionary{key1:value1, key2:value2}

The colon between key and value literals differentiates the initialization of a dictionary from initialization of a set.
Lists

- A list is a serial collection of elements
  - Elements do not have to be of the same type
  - Elements can be other collections (nested)
  - Default qualities – lists are unordered and non-unique
  - A list can be created by initializing a variable with comma separated elements surrounded by square brackets []

```
record['iFruit 3A',27,42,52,17,'enabled','connected',37.02549]
```
List Operations

- **element in list**
  - Returns `True` if the element is in the list

- **element not in list**
  - Returns `True` if the element is not in the list

- **n * list**
  - Copies the list $n$ times and concatenates the copies

```python
devices = ['iFruit','Sorrento']
devices = 2 * devices
print devices
> ['iFruit','Sorrento','iFruit','Sorrento']
```
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

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

- **`list.pop()`**
  - Returns the last element in the list and removes it
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 $list1$
append() Versus extend() for Collections

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

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

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

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

The last item is a list. This is a common mistake!
Slices of a List

- The slice notation works the same with collections as it did with strings
- Notice that the first element is 0, so \( q[0] \rightarrow 1 \), and \( q[5] \rightarrow 6 \)

- Slice range
  - The lower bound is included in the slice, but the upper bound is not.

- Range increment
  - Selects a subset of elements in a list, skipping some

- Slices applies to all collection types
Python `del` keyword

- **`del list`**
  - Deletes the entire list

- **`del list[offset]`**
  - Deletes the element at the given offset

- **`del list[start:end]`**
  - Deletes the range of elements from start to end-1
  - end is not included in the elements deleted

- **`del list[start:end:increment]`**
  - Deletes the elements within the range

```python
q = [1,2,3,4,5,6,7,8,9]
del q
print q
> < Error>

q = [1,2,3,4,5,6,7,8,9]
del q[3]
> q = [1,2,3,5,6,7,8,9]
```

```python
q = [1,2,3,4,5,6,7,8,9]
del q[2:6]
> q = [1,2,7,8,9]
```

```python
q = [1,2,3,4,5,6,7,8,9]
del q[1:9:2]
> q = [1,3,4,7,9]
```
List Interrogation

- **len(list)**
  - The number of elements in the list

- **list.count(element)**
  - Counts the number of identical elements in a list

- **list.min()**
  - Returns the minimum value element in the list

- **list.max()**
  - Returns the maximum value element in the list

- **list.index(element)**
  - Returns the offset of the first occurrence of element in the list

```python
def q1():
    q = ['a','b','c','b','c','c']
    print(len(q))
    print(q.count('c'))
    print(q.count('b'))
    print(q.index('c'))
    print(list(min(q)))
    print(list(max(q)))
```

```
q1()
```

```python
q = ['a','b','c','b','c','c']
q.count('c') > 3
```

```python
print(len(q)) > 6
```
List Transformation

- **list.sort()**
  - Sorts the list of elements in ascending order in place
  - Changes the order of the list itself
  - *The original order is lost*

- **sorted(list)**
  - Returns a sorted copy

- **list.reverse()**
  - Reverses the current order of elements in the list in place
  - Changes the order of the list itself
  - *The prior order is lost*

```python
q = [4,5,3,2,1,7,9,8]
q.sort()
print q
> [1,2,3,4,5,7,8,9]
```

```
q = [4,5,3,2,1,7,9,8]
q.reverse()
print q
> [8,9,7,1,2,3,5,4]
```

Common error!

```python
a = list.sort()
will set a to an empty list
```

Conversion between Strings and Lists

- `list = string.split(delimiter)`
  - Create a list from a delimited string
  - The `split()` method is used frequently in Cloudera classes

- `mystring = delimiter.join(list)`
  - Method for converting a list to a string
  - Note that this is a method of the delimiter... *looks awkward*
  - works with any delimiter string, including the empty string (" ")

```python
ab = '1,2,3,4,5'
print ab
> 1,2,3,4,5

mylist = ab.split(',')
print mylist
> ['1','2','3','4','5']

cd = "-".join(mylist)
print cd
> 1-2-3-4-5
```
**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

```python
lines = [ ]

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

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

```python
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'
```
Creating a List by Splitting a String

- This code uses `split()` to store each field as a string into an element of a list.

```python
onerecord = devlog[75].split(',', '')
print onerecord
> ['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\r\n']
```
Chapter Topics

Collections

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

- Originated from the abstraction of the common counting sequence... triple, quadruple, pentuple, septuple, octuple, ... n-tuple → tuple

- The mathematical concept of a Tuple is “a sequence of elements”
  - In math (1,2,3,4,5) is a pentuple – five elements

- In Python, a tuple is an immutable collection. It has all the same operation and interrogation methods as a list, but none of the manipulation or transformation methods

- The values are assigned when the tuple is created and cannot be changed other than through re-assignment

- Trying to call a method that changes the constitution of a tuple will result in an error

- Create a tuple with the parenthesis () notation.
Python Mutable Versus Immutable – Lists

- **Mutable**
  - Performing a function on an object may change the content of the object
  - The identity of the object is unchanged

- **Immutable**
  - Performing a function cannot change the content of the object
  - If change does occur, the resulting object has a new identity

```python
wifiA = ['25', '34', 'enabled']
id(wifiA)
> 4347030920
wifiB = ['connected']
id(wifiB)
> 4347029048
wifiA += wifiB
print wifiA
> ['25', '34', 'enabled', 'connected']

a = 25
id(a)
> 4298188664
a += 1
id(a)
> 4298188640
```

Python lists are considered mutable – the object ID remains the same. Python integers are considered immutable – the object id changes for the variable named `a`.
Immutable Tuple – No Change Methods

- You can't modify the order or membership of a tuple once it is assigned

```python
# Immutable Tuple – No Change Methods

t = (1, 2, 3, 4)
type(t)
> <type 'tuple'>

t.append(5)
> AttributeError: 'tuple' object has no attribute 'append'
t.extend(t)
> AttributeError: 'tuple' object has no attribute 'extend'
t.sort()
> AttributeError: 'tuple' object has no attribute 'sort'
del t[2]
> TypeError: 'tuple' object doesn't support element deletion
```
You can re-assign the value of an entire tuple, but it creates a new object.

**Tuple**

```python
my_tuple = (1,2,3,4)
p1 = print(my_tuple)
> (1,2,3,4)
p2 = print(id(my_tuple))
> 4300627208

my_tuple = my_tuple + my_tuple
print(my_tuple)
> (1,2,3,4,1,2,3,4)
p3 = print(id(my_tuple))
> 4297732776
```

**List**

```python
my_list = [1,2,3,4]
p1 = print(my_list)
> [1,2,3,4]
p2 = print(id(my_list))
> 4300881792

my_list.extend(my_list)
p1 = print(my_list)
> [1,2,3,4,1,2,3,4]
p2 = print(id(my_list))
> 4300881792
```
Casting Between Tuples and Lists

- Lists commonly contain elements of the same type, although this is not a rule.
- Tuples commonly contain elements of different types, to represent a record.
- A tuple can be reassigned. Casting a tuple into a list, modifying it, and then casting it back into a tuple is called *unfreezing* and *freezing*.

```
t = (39, 18, 29, 1)
type(t)
> <type 'tuple'>
print t
> (39, 18, 29, 1)

mylist = list(t)
print mylist
> [39, 18, 29, 1]

list.append(5)
t = tuple(mylist)
print t
> (39, 18, 29, 1, 5)
```
Chapter Topics

Collections

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

- Sets are unordered and unique

```python
ifruit = set(['1','5','5','3A','3','4','2','3A','4A','5'])
print ifruit
> set(['1', '3', '2', '5', '4', '3A', '4A'])
```

- A set is represented by comma separated elements
  - In 2.6, required syntax is `set([ ... ])`
  - In 2.7, a set can be specified using curly braces `{ ... }`

- You can create a new sorted set with the `sorted()` function

```python
ifruit = sorted(ifruit)
print ifruit
> set(['1', '2', '3', '3A', '4', '4A', '5'])
```
Casting Sets

- You can create a set from a tuple or a list using `set()`
- Casting with sets works the same as between lists and tuples

```
mylist = [5,3,4,1,2,5,3,5]
s1 = set(mylist)
print s1
> set([1,2,3,4,5])

mytuple = (5,3,4,1,2,5,3,5)
s2 = set(mytuple)
print s2
> set([1,2,3,4,5])
```
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])
```
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])
```
Set Interrogation Methods

- `set1.isdisjoint(set2)`
  - Returns `True` if both sets share no elements

- `set1.issubset(set2)`
  - Returns `True` if every element in `set1` is in `set2`
  - Symbolic notation: `set1 <= set2`

- `set1.issuperset(set2)`
  - Returns `True` if every element in `set2` is in `set1`
  - Symbolic notation: `set1 >= set2`

- `set2 > set1`
  - Proper subset
  - Returns `True` if `set2` contains all elements of `set1`, and one or more additional elements
Set Manipulation Methods

- Sets have their own manipulation methods
  - `set.add()`
    - Adds an element to the set
  - `set.update()`
  - `set.copy()`
  - `set.pop()`
  - `set.discard()`
You can create a frozenset from a set using `frozenset()`

A frozenset is immutable
- Tuple is to list as frozenset is to set

```python
s1 = {1, 2, 3, 4, 5}
f1 = frozenset(s1)
type(f1)
> < type 'frozenset' >

print(f1)
> frozenset([1,2,3,4,5])

f1.add(6)
> AttributeError: 'frozenset' object has no attribute 'add'
```
Chapter Topics

Collections

- Lists
- Tuples
- Sets

- Dictionaries

- Essential Points
- Hands-On Exercise: Collections
Dictionaries

- A dictionary is a set composed of key-value pairs
- Define using set notation – curly braces with pairs separated by commas
- Pairs are defined using a colon – `key: value`
  - Keys and values can be of any basic type

```python
diction = {'F40L': 'Sorrento', 'F41L': 'Sorrento', '2500': 'Titanic', '3000': 'Titanic', '3A': 'iFruit', '4': 'iFruit'}
type(diction) > < type 'dict' >

print diction['3A'] > 'iFruit'

print diction['2500'] > 'Titanic'
```
Dictionaries and Lists

- When you explicitly cast a dictionary to any other collection, you get the keys, not the values
- Use the `values()` method to get the values

```python
modellist = list(modelnumbers)
print modellist
> ['F41L', 'F40L', '4', '3A', '2500', '3000']

print modelnumbers.values()
> ['Sorrento', 'Sorrento', 'iFruit', 'iFruit', 'Titanic', 'Titanic']
```
Dictionary Methods

- `dict[key]`
  - Returns the value associated with the key

- `dict[key] = newvalue`
  - Overwrites the old value with the new value for that key

- `dict1.update(dict2)`
  - Adds or updates key-value pairs in `dict1` using contents of `dict2`

- `len(dict)`
  - Provides the number of key-value pairs in the dictionary

- `del dict[key]`
  - Removes the entry identified by the specified key from the dictionary

- `dict.pop(key)`
  - Returns the value and removes the key-value pair from the dictionary
Chapter Topics

**Collections**

- Lists
- Tuples
- Sets
- Dictionaries

**Essential Points**

- Hands-On Exercise: Collections
Part 1: What's your Prediction?

```python
a = [1, 3, 5, 4, 7, 4]
b = ('jan', 'feb', 'mar', 'apr', 'may')
c = {3, 5, 7, 5, 7, 24, 3, 'big'}
d = {1: 'iFruit', 2: 'Ronin', 3: 'Sorrento', 'T': 'Titanic'}
e = [a, b, c, d]
f = (a, b, c, d)

print e
print f

type(e)
type(f)
```

What type of collection will e and f be?
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'>
```
Part 2: What's your Prediction?

```
# Old values
#
a = [1, 3, 5, 4, 7, 4]
b = ('jan', 'feb', 'mar', 'apr', 'may')

# New values
#
a = {5, 4, 3, 2, 1}
b = ('jun', 'jul', 'aug', 'sep', 'oct')

print e
print f
```

Will the new values or the old values be displayed when `e` and `f` are printed?
Part 2: Results

```python
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'}]
```

```python
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'})
```

The list e and the tuple f still contain the original versions of a and b.

**Take-away:** Collections are constructed by duplication, not by reference.
Part 3: What's your Prediction?

```python
a = [1, 2, 3, 4, 5]
b = [a, a, a, a, a]
c = [b, b, b, b, b]
d = [c, c, c, c, c]
print d[3][3][3][3]
```
Part 3: Results

```python
a = [1, 2, 3, 4, 5]
b = [a, a, a, a, a]
c = [b, b, b, b, b]
d = [c, c, c, c, c]

print d[3][3][3][3]
> 4
```

Part of the Python philosophy is “complex” without being “complicated”

Take-away: You can create very complex nested objects to organize data.
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

- Lists
- Tuples
- Sets
- Dictionaries
- Essential Points

- Hands-On Exercise: Collections
Hands-On Exercises: Collections

- **Part 1**
  - Process a file line-by-line into a nested list

- **Part 2**
  - Read an entire file into a list
Course Chapters

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

Flow Control

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

- Code blocks in Python are represented by indentation
  - **Not** curly braces
  - End of line sequence (EOLS), not semicolon

- Statements in a block *must* be indented identically
  - If your indentation is off – even by a space – then it is an error

```python
(a, b) = 7, 12
if 1 < 9:
    print a
else:
    print b:
> 7
```

```python
(a, b) = 7, 12
if 1 < 9:
    print a
else:
    print b:
> IndentationError: expected an indented block
```
In the example on the right, `else:` is no longer associated with a corresponding `if` because the line `print 'hi'` ended the scope of the `if` statement.

Below, one space too much is also an error.

```python
a, b = 7, 12
if 1 < 9:
    print a
    print 'hi'
else:
    print b
```

> Error line 4
> `IndentationError: unexpected indent`

```python
a, b = 7, 12
if 1 < 9:
    print a
    print 'hi'
else:
    print b
```

> Error line 5
> `SyntaxError: invalid syntax`
Chapter Topics

Flow Control

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

- **while test:**
  - Identifies a code block to be executed repeatedly while test is True
  - If test is never False, it is an infinite loop
    - If you accidentally enter an infinite loop in the Python shell, break with CTRL-D or CTRL-C

Note: variable used in test must be initialized before use

```python
a = 1
while a < 10:
    print(a)
    a = a + 1
print("complete")
```

When test is False … continue

Code to execute repeatedly while test is True
Example Code

- This code prints each record of mobile phone data

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

for var in range(start,end,inc):
    code-block

Count from 0 to N-1

for c in range(5):
    print c
> 0
> 1
> 2
> 3
> 4

for c in range(3,8):
    print c
> 3
> 4
> 5
> 6
> 7

for c in range(0,12,3):
    print c
> 0
> 3
> 6
> 9

Count from start to end-1 with increment

start to end-1

Note: counting variable did not need to be separately initialized before use.
Modifying Loop Operations

- **break**
  - Ends a loop

```python
for i in range(100):
    if (i > 4):
        break
    print i
> 0
> 1
> 2
> 3
> 4
```

- **continue**
  - Skips one iteration of a loop

```python
for i in range(10):
    if (i == 5):
        continue
    print i
> 0
> 1
> 2
> 3
> 4
> 6
> 7
> 8
> 9
```
Chapter Topics

Flow Control

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

- **for each_element in list:**
  - Execute a block of code for each element in a list, tuple, set, frozenset, or dictionary

- Returns variable of type of the element

```
mylist = [1,2,3,4,5,6]
for each in mylist:
    print each
> 1
> 2
> 3
> 4
> 5
> 6

type(each)
> <type 'int'>
```
Key and Value Iteration

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

```python
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

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

type(each)
> <type 'tuple'>

len(each)
> 2
```
Corresponding Iteration with Zip

- for each_element in zip(list1, list2):
  - Returns a two-value tuple where the 0 element comes from the first list, and the 1 element is the corresponding element from the second list

```python
mylist1 = ['Ronin', 'MeToo', 'iFruit', 'Sorrento']
mylist2 = ['s1', '4.0', '3A', 'F41L']
for each in zip(mylist1, mylist2):
    print each
> ('Ronin', 's1')
> ('MeToo', '4.0')
> ('iFruit', '3A')
> ('Sorrento', 'F41L')

type(each)
> <type 'tuple'>

len(each)
> 2
```
Multiple List Iteration with \texttt{zip() \textcolor{red}{and}} List Comprehension

\begin{verbatim}
x = ['x1','x2','x3']
y = ['y1','y2']
for a,b in zip(x,y):
    print a,b
> x1 y2
> x2 y2

x = ['x1','x2','x3']
y = ['y1','y2']
for a,b in[(a,b) for a in x for b in y]:
    print a,b
> x1 y1
> x1 y2
> x2 y1
> x2 y2
> x3 y1
> x3 y2
\end{verbatim}

Multiple list iteration without the need for explicit nested for loops or counters
Chapter Topics

Flow Control

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

- **`if test:`**
  - Make a block of code conditional
  - If False, continue on next line of code following the block

- **`test`**
  - Boolean
    - `True` or `False`
  - Comparator
    - Operators discussed in chapter on variables
      - `==`, `!=`, `<`, `>`
      - `<=`, `>=`
      - `is`, `is not`
  - Ends with a colon

---

```python
# Device not to exceed 120 F
#   120 F = 48.889 C
#
devicetemp=[30,33,54,34,29,49]
maxtemp = 48.889

for eachtemp in devicetemp:
    if eachtemp > maxtemp:
        print eachtemp

> 54
> 49
```
Alternative Conditions with \texttt{Else}

- Adds an alternate block of code to execute if test is False
- \texttt{else}: identifies alternate block of code
  - \texttt{else}: keyword is aligned with \texttt{if}
  - requires indented block beneath
- \textit{Either} code block 1 or 2 will be executed, then processing will continue

```python
device = "Titanic 2000"
if device[8:13] == "2000":
    print "Model recognized"
else:
    print "Model unknown"
> Model recognized
```
Elif

- Creates alternate tests
- `elif test2:` identifies an alternate test to perform if `test1` is False
  - If `True` executes following code block

```python
if test1:
    code-block-1
elif test2:
    code-block-2
else:
    code-block-n
```
Stacking **Elif**'s

- Python does **not** have a `switch()` case construct
- Use multiple `elif`'s
  - Else...If...
- If `test1` is False, try `test2`, if `test2` is False, try `test3`, if `test3` is False try `test4`...
  .. <else:>.. if all the previous tests were False... then *do this*

```python
if test1:  
    code-block-1
elif test2: 
    code-block-2 
elif test3: 
    code-block-3
elif test4: 
    code-block-4
elif test5: 
    code-block-5
elif test6: 
    code-block-6
elif test7: 
    code-block-7
else:  
    code-block-n
```

*Code to execute if test1 is True*

*Code to execute if all tests are False*
Example of `Elifs`

- 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

```python
# 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
```
Single-line If

- Python supports a single-line conditional statement
  - also called an “inline function”

```
return-true if(test) else return-false
```

```
a = 5
'low' if a <= 5 else 'high'
> 'low'
```

Python does not support the ternary operator (the conditional operator) that is common in many other languages

```
# test ? return-true : return-false
a = 5
a <= 5 ? 'low' : 'high'
> Syntax Error
```
Chapter Topics

**Flow Control**

- Code Blocks
- Repetitive Execution
- Iterative Execution
- Conditional Execution

**Tentative Execution (Exception Handling)**

- Essential Points
- Hands-On Exercise: Flow Control
Tentative Execution

- **Exception Handling**
  - The opportunity to execute code in the event of an error rather than quitting

- **Errors**
  - NameError
  - ValueError
  - IndexError
  - SyntaxError

```python
try:
    code-block-1
except: error
    code-block-2
finally:
    code-block-3
```
Error Handling

```python
mylist = ['Ronin', 'MeToo', 'iFruit', 'Sorrento']
print len(mylist)
toofar = len(mylist)+1
index = 0
for index in range(0, toofar):
    print mylist[index]

> Ronin
> MeToo
> iFruit
> Sorrento
> IndexError: List out of range
```

Index out of bounds
Error Handling

```python
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**
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
Program Structure

Chapter 6
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
Named Functions

- Defines a function and associates it with a name

- `return` must be indented so it is ‘within’ the function
- `return` is optional

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

```python
return(1, 2, "data")
```

Q: Will this work? If so, what type of variable will it return?

Function gets a local copy of the passed parameters
Optional return variable
Example of Named Functions

```python
# 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.
Variable Scope

- Variables may be defined within the function (i.e., local scope)
- Local variables defined in the code that calls a function are not visible to the function that is being called
- Parameters are passed into functions by copying, not by reference
- Modifying the value of a parameter only modifies the local copy
- When a function returns, the local variables defined within the function are lost
- It is possible to define global variables by prefixing their definition with the `global` keyword, but this should be avoided whenever possible
Definitions
- \( a \) is a passed parameter
- \( b \) is a global variable
- \( c \) is a local variable

Changes
- In the call to \texttt{fun1()} all three variables are incremented

Results
- What will happen in the main program after the call to \texttt{fun1()}?
- \textit{What will the result of the print statements be?}
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
# -- 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
Single-Line Notation Functions

- Occurs on a single line
- `return` keyword precedes single line of code

```python
def CtoF(tempC): return (tempC*9/5) + 32
print CtoF(25)
> 77
```
Pass by Reference

- When a function name appears in parentheses, meaning it is passed as a parameter, Python executes it.
- When a function name appears without parentheses, Python treats it as an object and passes it by reference (*called first class functions*).

```python
def square(number):
    return number*number

def halve(number):
    return number/2

def doit(somefunction,input):
    return somefunction(input)

doit(halve,7)  # > 3

doit(square,7)  # > 49
```
Chapter Topics

Program Structure

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

- 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

\[ \text{lambda parameters: code-line} \]
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

```python
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()`

```python
# -- 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

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

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

- Anonymous function in call to `map()`

```python
# -- 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]
```
Second Example of `map()`

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

```python
# -- 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

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

```python
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()`

```python
# -- 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
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-\text{eee}-\text{ssss} = 3 + 1 + 3 + 1 + 4 = $ Each phone is a 12 character string
- Skip zeroes in the exchange field
- $\text{aaa}-123-4567 = 10,000,000 = \text{about } 10$ million numbers
- $12 \times 10^m = 120 \text{ MB of data}$
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

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
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()`?
Generators

- **A Python generator is an iterable function**
  - Also known as a *cofunction* in computer science

- **yield**
  - Preserves the functional context (local variables and state)
  - Returns control to the caller
  - Passes an iterator object back
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

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

# --- Generate all phone numbers in the target area codes
def phonegen():
```
def phonegen():
    global phonenum
    d1=d2=d3=d4=d5=d6=d7 = 0
    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)
                                phonenum = phone
                                yield phonenum

it = 0
for phonenum in phonegen():
    if it > 50:
        break
    print phonenum
    it += 1
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
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

- **Generator Functions**
  - Write a number series generator using a standard function
Course Chapters

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

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

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

def userinput(prompt):
    keybuffer = raw_input(prompt)
    return keybuffer
```
Examples of Accessing the Library

- The `import` syntax loads the functions as methods of the library object.

  ```python
  import library
  mylist = library.loadfile('loudacre.log')
  print mylist
  print "\n\n"
  myline = library.userinput('Greetings: ')  
  print myline
  ```

- The `from` syntax loads the functions directly into the main program symbol table.

  ```python
  from library import loadfile, userinput
  mylist = loadfile('loudacre.log')
  print mylist
  print "\n\n"
  myline = userinput('Greetings: ')  
  print myline
  ```
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/pythonversion`

- **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

```python
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 . . .

```python
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

```python
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
```
Where Do Libraries Come From?

- **Standard (ships with Python)**
  - Not loaded by default, must be imported
  - Examples: `sys, math, re`

- **Separately installed**
  - `*.egg` files
  - A Python “egg” (`*.egg`) is a distribution of a Python project that may contain code, metadata, and resources
  - Separately installed using `pip` or `easy_install`

```
/Library/Python/2.7/site-packages

bigquery-2.0.17-py2.7.egg
oauth2client-1.2-py2.7.egg
pip-1.5.2-py2.7.egg
google_api_python_client-1.2-py2.7.egg
python_gflags-2.0-py2.7.egg
```
What's in a Library?

- `import libraryname`
- `dir(libraryname)`

```python
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

```python
print math.__doc__
> 'This module is always available. It provides access to the mathematical 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'
```
Example of `__name__`

A program can test `__name__` to determine if it is running after import in a main program or if it is running standalone.

- One use is to test code that only runs if the library is executed standalone.

```python
if __name__ == '__main__':
    # only prints if in main program
    print "running library test"

def loadfile(filename):
    lines=[  ]

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

$ python library.py
> running library 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

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

```python
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
**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) 
      [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
Built-in `math`

- `import math`
  - Imports the math module
- `math.ceil()`
  - Rounds up to a whole number, returns a `float`
- `math.floor()`
  - Rounds down to whole number, returns a `float`
- `math.sqrt()`
  - Square root
- `math.pi()`
  - Constant
- `math.e()`
  - Constant

More information:
https://docs.python.org/2/library/math.html
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()`
Overview of Regular Expressions

- RegEx redefines elements within the pattern string that already have a different meaning to Python outside of the search string
  - `[ ]` = delimits a group of characters, any of which are a match
  - `[0–9]` = matches characters between zero and nine
  - `[^0–9]` = matches any characters except zero through nine
  - `[0–9]*` = matches number of times (zero or more repetitions)
  - `.` = matches any character except a newline
  - `\w` = matches any whitespace character

- Python wrinkle... if you wanted to match `[`
  - you'd have to escape it like this in the search string: \\[
  - but... Python will process the escape sequence in the string...
  - so... you'd need to escape the escape and the bracket: \\\\[
  - or... use Python raw: r"\\[
  - or... triple quotes: """match[this] string exactly"""
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`
Chapter Topics

Working with Libraries

- Storing and Retrieving Functions
- Module Control
- Common Standard Libraries
- Essential Points

- Hands-On Exercise: Libraries
Hands-On Exercise: Libraries

- **Demonstrate basic skills with Python Modules**
  - Create a library file and call it from a main program
  - Move the library to a different location
  - Use a common standard library to interface with the OS
  - Use the RegEx library to perform a sophisticated search
Conclusion

Chapter 8
Course Chapters

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

During this course, you will learn

- “Just Enough” Python Programming
  - "Just Enough" means to enable a solid foundation for Hands-On Exercises in Cloudera training classes
  - Not "proficient as a Python programmer" (Pythonista, Pythoneer)
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*