Course 1905

Python Programming Introduction

Introduction and Overview

Learning Tree International

Training You Can Trust

Course Objectives

Upon completion of this course, you will be able to

- **Create, edit, and execute Python programs in Eclipse**
- **Use Python simple data types and collections of these types**
- **Control execution flow: conditional testing, loops, and exception handling**
- **Encapsulate code into reusable units with functions and modules**
- **Employ classes, inheritance, and polymorphism for an object-oriented approach**
- **Read and write data from multiple file formats**
- **Query relational databases using SQL statements within a Python program**
- **Display and manage GUI components, including labels, buttons, entry, and menus**
- **Create a web application with the Django framework**

GUI = graphical user interface

SQL = structured query language

Course Contents

Chapter 1

Python Overview

Chapter Objectives

After completing this chapter, you will be able to

- **Describe the uses and benefits of Python**
- **Enter statements into the Python console**
- **Create, edit, and execute Python programs in Eclipse**
- **Identify sources of documentation**

Chapter Contents

- ➢ **Python Background**
- **Executing Python**
- **Documentation Resources**

A Definition of Python

- **An object-oriented, open-source programming language**
  - Inheritance, encapsulation, and polymorphism supported
  - Freedom to use Python and its applications for no charge
- **A general-purpose language used for a wide variety of application types**
  - Text and numeric processing
  - Operating system utilities and networking through the standard library
  - GUI and web applications through third-party libraries
- **Python Software Foundation (PSF) controls the copyright and development of the language after version 2.1**
  - An independent nonprofit group
  - Guido Van Rossum started creating the language in 1989
    - Known as the Benevolent Dictator For Life (BDFL)
    - Leads language development and selection of new features

Python Philosophy

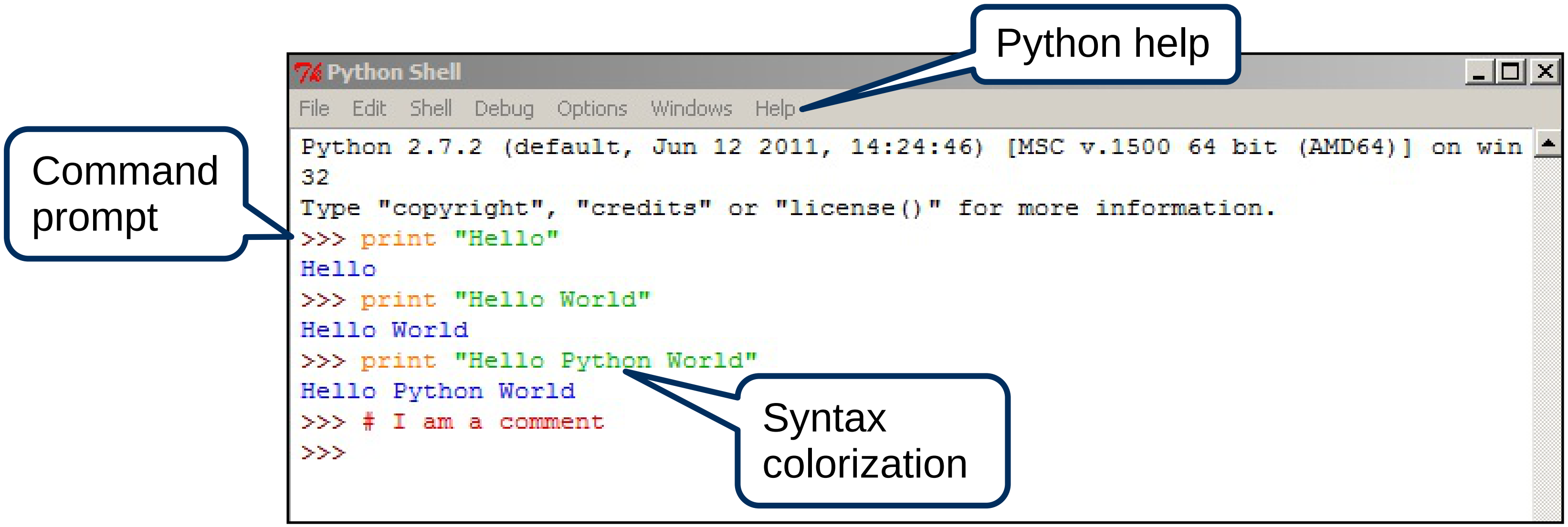
- **Simple solutions**
  - Code should clearly express an idea or task
- **Readability**
  - White space and indentation to define blocks of code
  - Less coding required as compared to the equivalent .NET, C++, or Java
    - Decreases development and maintenance time requirements
- **Dynamic typing and polymorphism**
  - No data type declarations
  - Operator and method overloading
    - Operators are evaluated at runtime based on the expression type

- Python Background
- ➢ **Executing Python**
- Documentation Resources

Accessing the Interpreter

- **Console provides a command-line interface to the interpreter**
  - Executes commands interactively
  - Has a built-in help system
- **Statements are executed in the same manner as when run from a file**
  - Enables testing as you write
  - Copy code from an editor and paste into the console
  - History mechanism retrieves previous statements
    - Modify before executing
- **In this course, there are two interfaces to the command interpreter**
  - Launch the IDLE application from a taskbar button 
  - Launch the console from a taskbar button 

1. **Access the application using the button**
   - Provides a command-line interpreter
     - Also an editor and help browser



Do Now

2. Enter the following statements:

```
>>> print "Hello"
Hello
```

Press <Enter>

3. Use `<Alt><P>` to access the previous `print`, then `<Left arrow>` to move within the string; append the text `World`

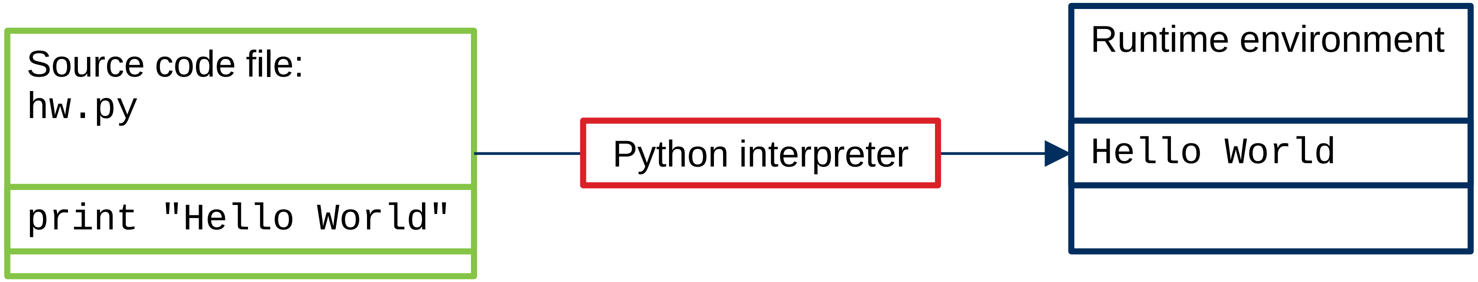
```
>>> print "Hello World"
Hello World
```

4. Use `<Alt><P>` to access the previous `print`, then `<Left arrow>` to move to the middle of the string; insert the text `Python`

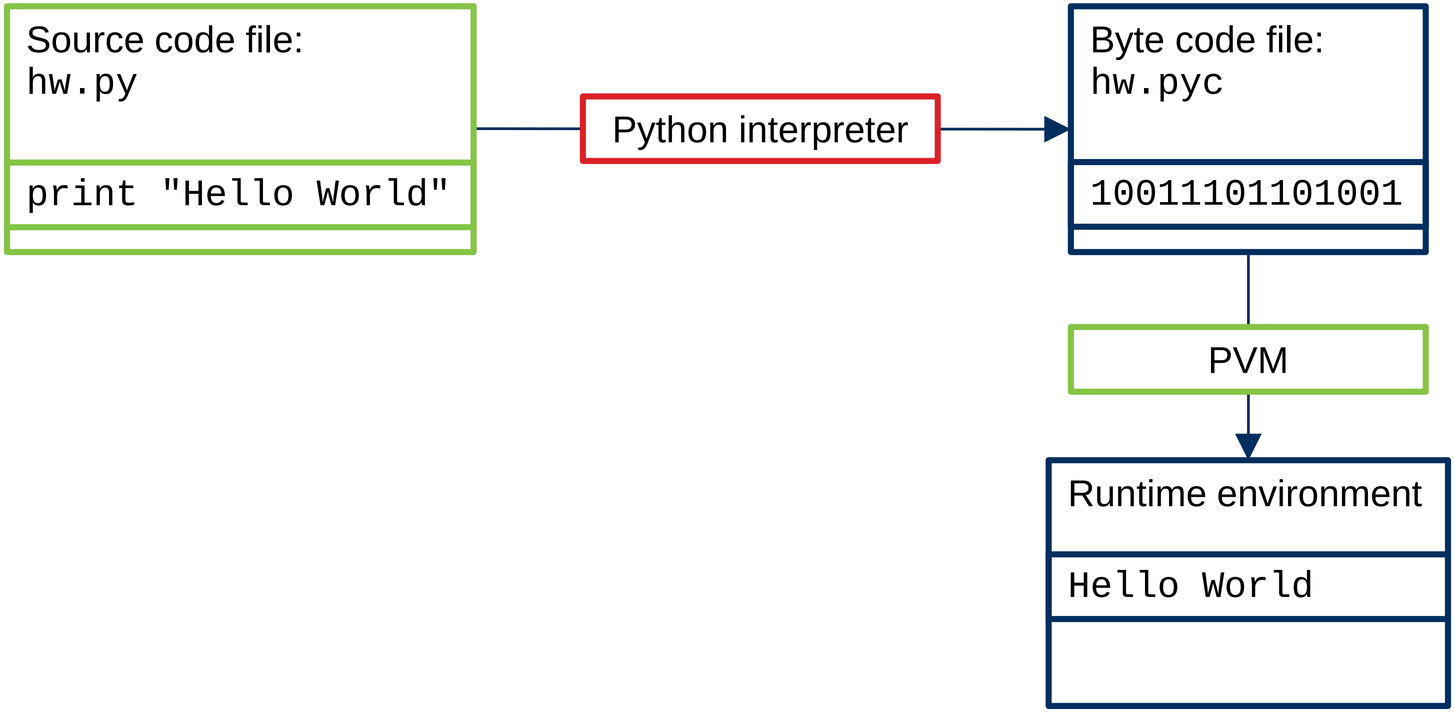
```
>>> print "Hello Python World"
Hello Python World
```

The Python Interpreter

- **Executes source code statements**
  - Entered interactively from a console
  - Read from a text file
    - A Python program



- **Can optionally create byte code**
  - Byte code is machine architecture independent
  - Byte code files have `.pyc` extension
- **Byte code is executed by the Python Virtual Machine (PVM)**
  - Operating system dependent



Do Now

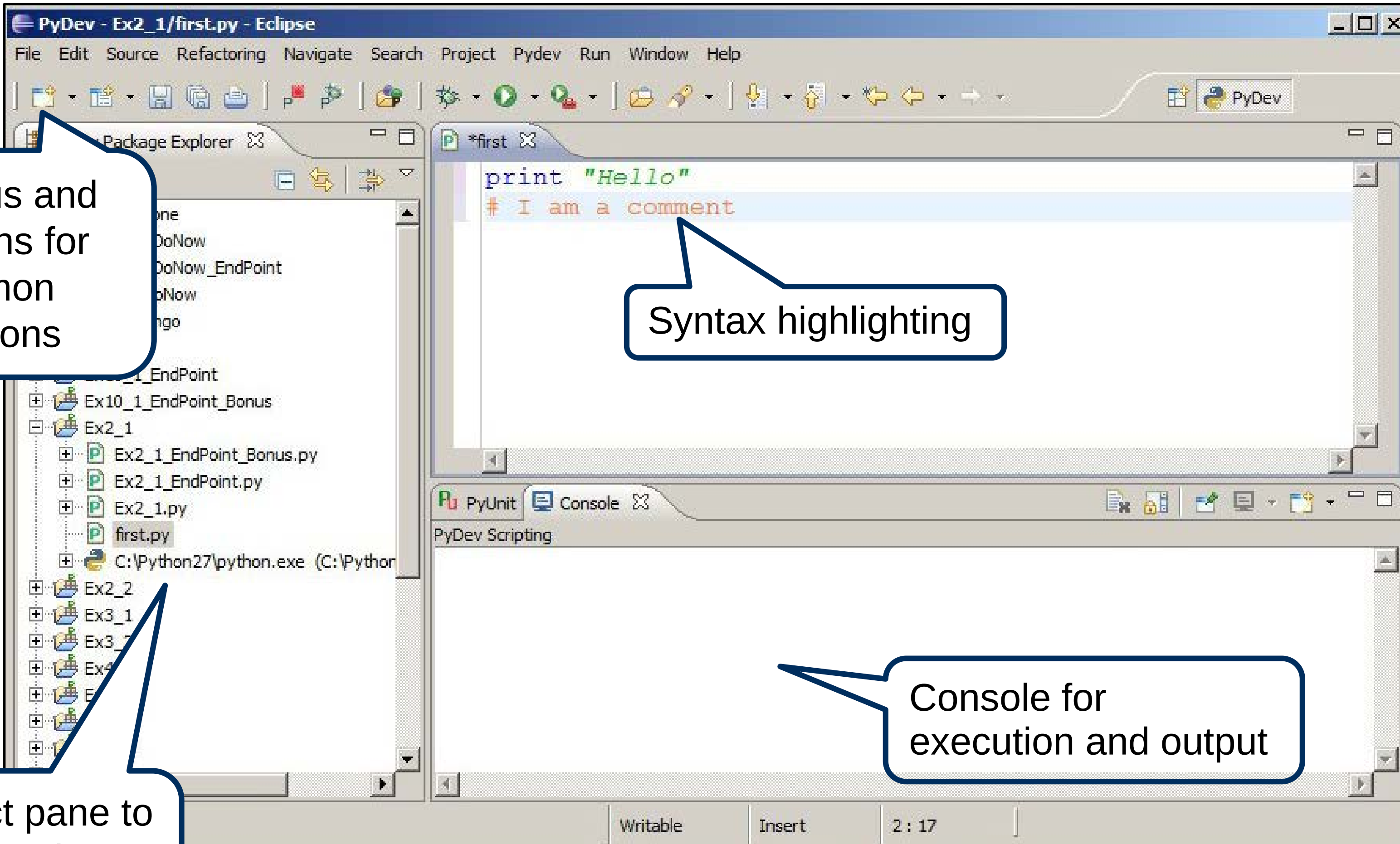
1. Access Eclipse using the button
2. Create a new file by following the menu path File | New | File
3. Enter `Ex2_1` as the parent folder and `first.py` as the file name
4. Input the following lines into the editor pane:

```
print "Hello"
# I am a comment
```

5. Save the editor contents by following the menu path File | Save
6. Execute your program by following the menu path Run | Run and selecting Python Run from the Run As pop-up
   - Output appears in the console under the editor pane

```
Hello
```

Do Now



Menus and buttons for common functions

Syntax highlighting

Console for execution and output

Project pane to manage the source files

- Python Background
- Executing Python
- ➢ **Documentation Resources**

Documentation

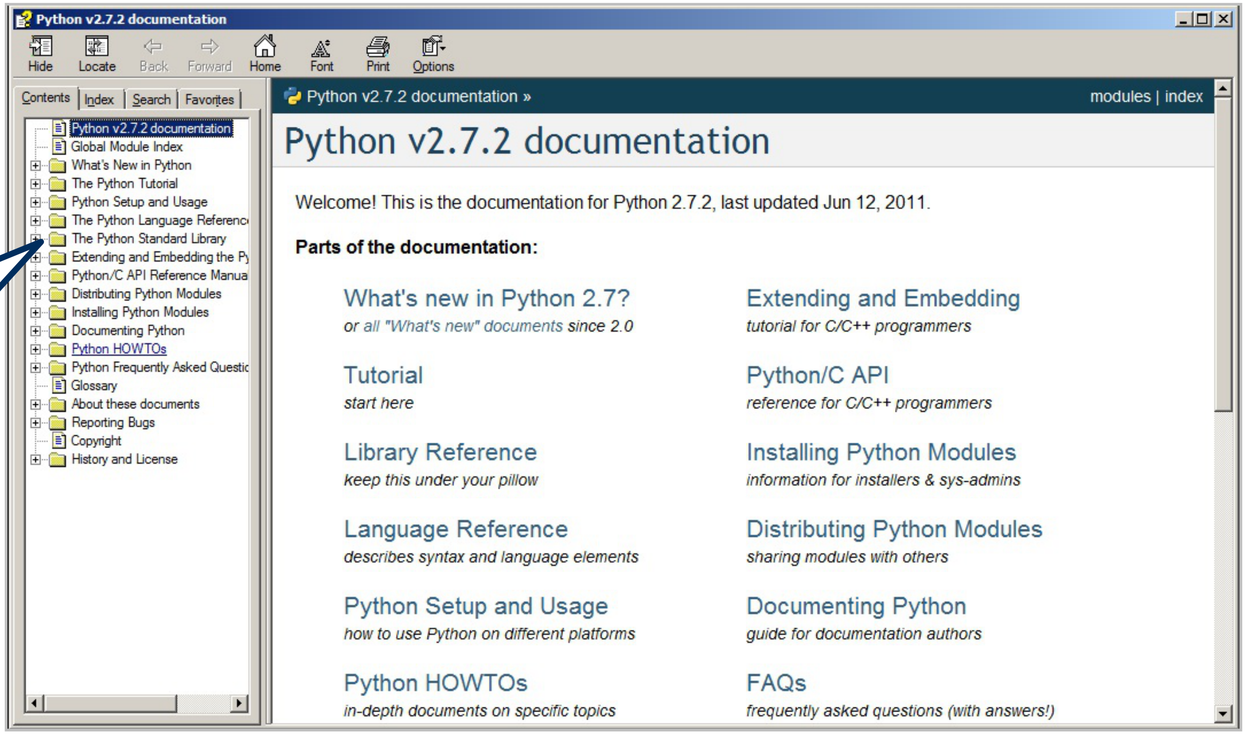
- **Comments in source code follow the # character**
  - Remainder of the line is ignored
  - No block format
- **Doc strings**
  - Describe larger code sections
    - Written at the top of modules, functions, classes
  - Enclosed in a set of triple quotation marks
  - Available as the `__doc__` attribute of an object
- **Built-in `help()` function**
  - Console interface to PyDoc

PyDoc

PyDoc

- Accesses docstrings and presents them with a GUI or HTML interface

From the Help menu of IDLE



HTML = hypertext markup language

Chapter Summary

You are now able to

- Describe the uses and benefits of Python
- Enter statements into the Python console
- Create, edit, and execute Python programs in Eclipse
- Identify sources of documentation

Chapter 2

Working With Numbers and Strings

Chapter Objectives

After completing this chapter, you will be able to

- **Write simple Python programs**
  - Create variables
  - Manipulate numeric values
  - Manipulate string values
  - Make decisions

➢ **Objects and Variables**

- **Numeric Types and Operations**
- **String Types and Operations**
- **Conditionals**

Python Objects

- **An object is an instance of a data value stored in a memory location**
  - Memory is allocated when the object is created
    - Built-in function `id()` shows memory address
  - Memory is reclaimed when the object is no longer referenced
    - *Garbage collection*
  - `1`, `2.5`, and `'Welcome'` are all objects
- **An object has a *type***
  - Built-in function `type()` shows type
  - `1` is an integer type
  - `2.5` is a floating point type
  - `'Welcome'` is a string type
- **An object's type constrains the operations on that object**
  - Arithmetic on integer or floating point types
  - Concatenation on a string type

Objects Illustrated

```
>>> 1
1
>>> id(1)
14539386
>>> type(1)
<type 'int'>
```

14539386
int
1

Memory address
Type
Value

```
>>> 1 + 2
3
```

14539386
int
1

14539634
int
2

14539740
int
3

Variables

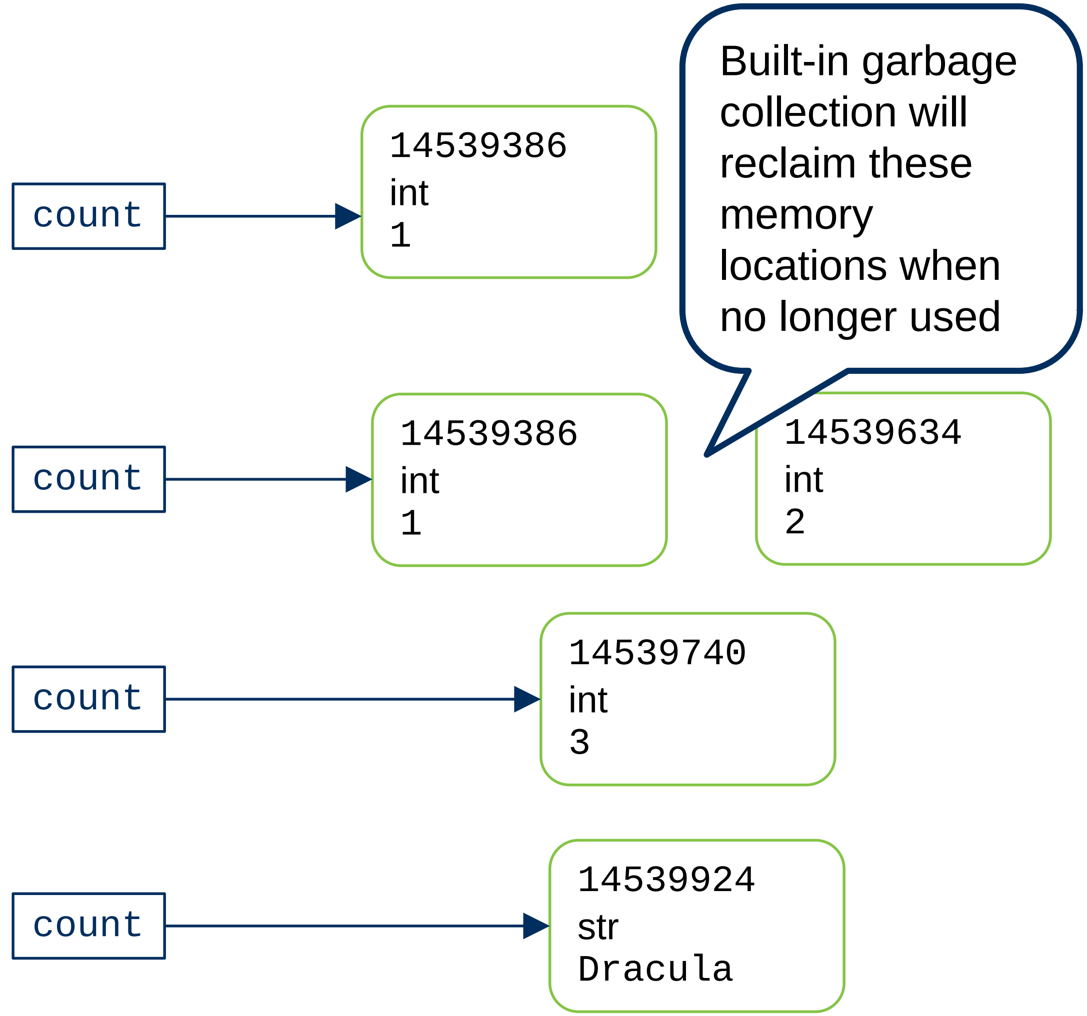
- **A variable is a named reference to an object**
  - Operations on variables use the object referenced
  - Operations are constrained by the type of the object
  - Variable instance is created when an object is assigned
    - An *identifier*
- **A variable is modified by assigning a different object**
- **May reference any type of object**

Variables Illustrated

```
>>> count = 1
>>> count
1
```

```
>>> count = count + 2
>>> count
3
```

```
>>> count = "Dracula"
>>> count
Dracula
```



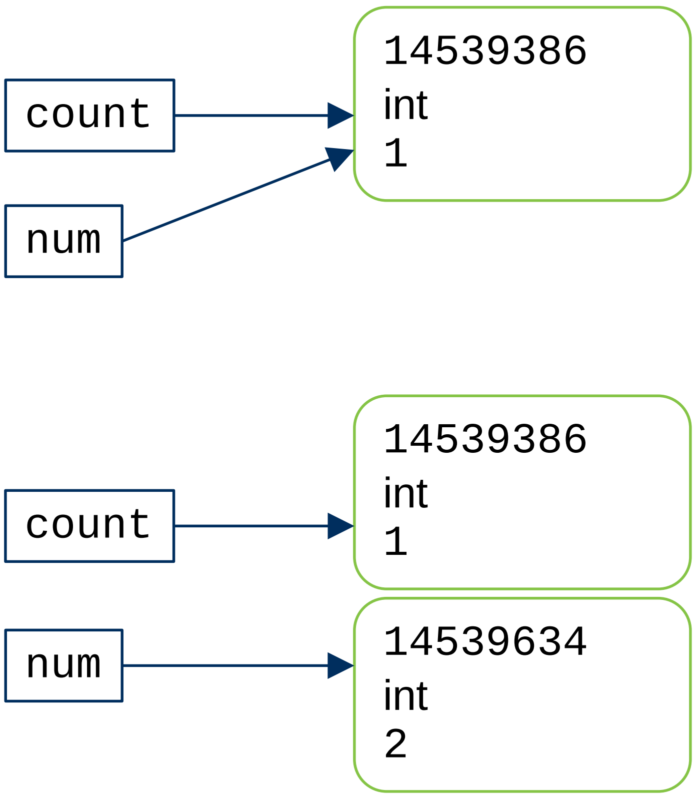
Shared Reference

- **Multiple variables reference the same object**
  - Created by assigning one variable to another
    - Or the same object to both
  - Confirmed through object identity test operator `is`

```
>>> count = 1
>>> count
1
>>> num = count
>>> num
1
>>> num is count
True
```

Boolean result

```
>>> num = 2
>>> num
2
>>> num is count
False
```



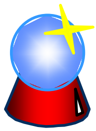
Naming Rules

- **Start with letter or underscore**
  - Followed by any number of letters, digits, or underscores
    - Case sensitive
- **May not be a keyword**
  - Should not be a built-in name
- **PEP 8, the style guide for Python, recommends lowercase with underscores as necessary**
  - `firstname`
  - `first_name`

- Objects and Variables
- ➢ **Numeric Types and Operations**
- String Types and Operations
- Conditionals

Python Built-In Types

- **Immutable types**
  - String literals
  - Arithmetic literals
    - Integer, floating point



- **Collection types**
  - Lists, dictionaries, and sets are mutable
  - Tuples are immutable

Numeric Objects

- **Numbers are a core Python type**
- **Integers can be represented exactly in memory and have no fractional part**
  - Examples:

    `4        123        0`
  - May be specified in octal, hexadecimal, or binary representation
    - Example: `014`, `0xC`, and `0b1100` are equivalent to decimal `12`
- **Floating point objects have an integer portion and a fractional portion**
  - Examples:

    `4.0    123.56    0.000001      1.5e10      6.9E-6`
  - Floating point objects are represented as approximations in memory
- **Complex number objects are stored as two floating point values**
  - For the real and imaginary parts

Numeric Operators

Precedence

<table>
<tr><td><code>( )</code></td><td>Raise precedence level</td></tr>
<tr><td><code>a ** b</code></td><td>Exponentiation</td></tr>
<tr><td><code>~a</code>
<code>-a</code>
<code>+a</code></td><td>Bitwise NOT
Negation
Identity</td></tr>
<tr><td><code>a * b</code>
<code>a / b</code>
<code>a // b</code>
<code>a % b</code></td><td>Multiplication
True division
Floor division
Modulus</td></tr>
<tr><td><code>a + b</code>
<code>a - b</code></td><td>Addition
Subtraction</td></tr>
<tr><td><code>a << b, a >> b</code></td><td>Bit shift</td></tr>
<tr><td><code>a & b</code></td><td>Bitwise AND</td></tr>
<tr><td><code>a ^ b</code></td><td>Bitwise exclusive OR</td></tr>
<tr><td><code>a | b</code></td><td>Bitwise OR</td></tr>
<tr><td><code>a < b, a <= b, a > b, a >= b</code>
<code>a != b, a == b</code></td><td>Relational
Equality</td></tr>
<tr><td><code>=, +=, -=, *=, /=, %=</code></td><td>Assignment</td></tr>
</table>

Numeric Operators Example

```
>>> count = 1
>>> count = count + 1
>>> count
2
>>> count += 1
>>> count
3
>>> num = 2
>>> num < count
True
>>> count == 3
True
```

Multiple precedence levels

Compound assignment

Comparisons yield Boolean results

Numeric Operation Type

- Results of operations on objects of the same type yield results of the same type
- Results of operations on objects of mixed types are converted to the bigger type

```
>>> 5 / 3
1
>>> 5 % 3
2
>>> 5 // 3
1
>>> 5.0 / 3.0
1.6666666666666667
>>> 5.0 // -3.0
-2.0
```

All integer

Floor division, //, always rounds down

```
>>> 5 / 3.0
1.6666666666666667
>>> 5 % 3.0
2.0
>>> 5 // 3.0
1.0
```

Result is floating point

Python 3 integer division always yields a floating point result

Arithmetic Typing Functions

➤ **The `float()` and `int()` functions return the argument in the specified type**

- Argument may be the string representation of a numeric value
- Argument may be an expression

```
>>> num = '100'
>>> float(num)
100.0
>>> int(num)
100
>>> num
'100'
>>> int(9.0 / 5.0)
1
>>> int('9 / 5')
```

String representation of a numeric value

Raises `ValueError` exception

Arithmetic Base Functions

- The `oct()`, `hex()`, and `bin()` functions return the argument as a string in the specified base

```
>>> num1 = 12
>>> oct(num1)
'014'
>>> hex(num1)
'0xc'
>>> bin(num1)
'0b1100'
```

Base-10 value

- The `int()` function converts a string representation of a base into an integer

```
>>> int('10')
10
>>> int('10', base=8)
8
>>> int('10', base=16)
16
>>> int('10', base=2)
2
```

`print` Statement

- **Accepts a comma-delimited series of expressions**
- **Converts the expressions into strings and writes them to standard output**
  - Output is terminated by a newline
    - Newline is suppressed if argument list ends with a comma

```
>>> emp_id = 45733
>>> sal = 150000.00
>>> print emp_id, sal
45733 150000.00
>>> print emp_id, ',', sal
45733 , 150000.00
```

`print` is a function in Python 3

```
>>> emp_id = 45733
>>> sal = 150000.00
>>> print(emp_id, sal, sep=',')
45733,150000.00
```

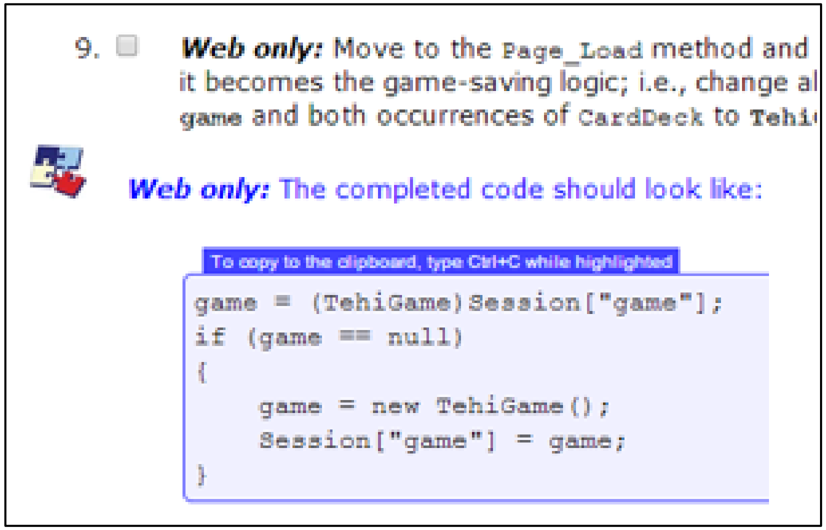
AdaptaLearn™ Enabled

- **Electronic, interactive exercise manual**
- **Offers an enhanced learning experience**
  - Some courses provide folded steps that adapt to your skill level
  - Code is easily copied from the manual
  - After class, the manual can be accessed remotely for continued reference and practice

- **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**

Do Now

1. **Launch AdaptaLearn by double-clicking its icon on the desktop**
   - Move the AdaptaLearn window to the side of your screen or shrink it to leave room for a work area for your development tools
2. **Select an exercise from the exercise menu**
   - Zoom in and out of the AdaptaLearn window
   - Toggle between the AdaptaLearn window and your other windows
3. **Look for a folded area introduced with blue text (not available in all courses)**
   - Click the text to see how folds work
4. **Try to copy and paste text from the manual**
   - Some courses have code boxes that make it easy to copy areas of text while highlighted (as shown)

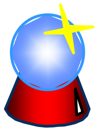


Hands-On Exercise 2.1

In your Exercise Manual, please refer to
Hands-On Exercise 2.1: Arithmetic and Numeric Types

The `math` Module

- **Provides many additional arithmetic capabilities**
- **Is part of Python's standard library**
  - `import` to make the functions available
    - References to objects within the module's namespace require a qualified name



```
>>> import math
>>> math.pow(2,3)
8.0
>>> math.sqrt(4)
2.0
>>> math.factorial(4)
24
>>> math.pi
3.141592653589793
```

Functions from the module

Constant from the module

Chapter Contents

- Objects and Variables
- Numeric Types and Operations
- ➢ **String Types and Operations**
- Conditionals

String Objects

- **String values are defined between a pair of quotation marks**
  - Single and double quotes are equivalent
  - Triple quotes of either type are allowed

```
>>> name = 'Guido'
>>> question = "Don't you love Python?"
>>> question = ''' Don't you love Python?'''
```

- **Strings are the core Python type `str`**
  - Sequence type
    - Series of single characters ordered left to right by position
    - Individual elements can be referenced
  - Immutable type
    - Object cannot be changed

String Slicing

➢ A slice is a portion of a sequence

- Described by its offset
- Slice boundary is a range specified in `[`*`start`*`:`*`end`*`]`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	**t**	**l**	**a**	**n**	**t**	**i**	**c**		**O**	**c**	**e**	**a**	**n**

```
>>> sea = 'Atlantic Ocean'
>>> sea[0]
'A'
>>> sea[0:8]
'Atlantic'
>>> sea[:8]
'Atlantic'
>>> sea[9:]
'Ocean'
>>> sea[:]
'Atlantic Ocean'
```

Bounded slice

Unbounded slices extend to an end

Entire sequence

String Slicing

- **An offset may be described from either end of the string**
  - Use a negative offset

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	t	l	a	n	t	i	c		O	c	e	a	n
–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

```
>>> sea[-1]
'n'
>>> sea[-5:]
'Ocean'
>>> sea[-1:-5]
''
```

Always the last reference from a sequence

Undefined slice yields an empty string

- **A step or stride may access nonsequential values**
  - Use [*start*:*end*:*step*]

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	**t**	**l**	**a**	**n**	**t**	**i**	**c**		**O**	**c**	**e**	**a**	**n**
–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

```
>>> sea[1:12:2]
'tatcOe'
>>> sea[-1:-5:-1]
'naecO'
>>> sea[::-1]
'naecO citnaltA'
>>> sea[9] = 'o'

Traceback (most recent call ...
TypeError: 'str' object does not
support item assignment
```

Every second element from 1 to 12

String reversal

String is immutable

- **Operators create and return new string objects**
  - \+ concatenation
  - * repetition
- **The `str()` function returns its argument as a string**

Concatenation

Repetition

Type conversion

```
>>> name = 'Guido'
>>> name + name + name
'GuidoGuidoGuido'
>>> name * 3
'GuidoGuidoGuido'
>>> name[0] * 3
'GGG'
>>> name[0] * 3 + name + name
'GGGGuidoGuido'
>>> str(-12.5)
'-12.5'
>>> str( 7 / 3.0 )
'2.3333333333'
```

String Methods

- **Functions that operate on string type objects**
  - Syntax: *`string`*`.`*`method`*`()`
- ***`string`*`.upper()` and *`string`*`.lower()` return a new string**
- ***`string`*`.isupper()`, *`string`*`.islower()`, and *`string`*`.isdigit()` return a Boolean**

New string returned

Boolean returned

```
>>> sea = 'Atlantic Ocean'
>>> bigsea = sea.upper()
>>> bigsea
'ATLANTIC OCEAN'
>>> smallsea = sea.lower()
>>> smallsea
'atlantic ocean'
>>> smallsea.isupper()
False
>>> smallsea.islower()
True
```

String Methods

- ***string*.find()** and ***string*.rfind()** return the offset of the search string
  - Or -1 if the string is not found

Offset

```
>>> sea = 'Atlantic Ocean'
>>> sea.find('a')
3
>>> sea.rfind('a')
12
>>> sea[sea.find('a'):sea.rfind('a') + 1]
'antic Ocea'
```

Use returned values for slicing

String Methods

- ***string*.replace(*old*, *new*)** returns a new string after replacing all occurrences of ***old*** with ***new***
- ***string*.split()** returns a *list* of strings based on a delimiter
- ***string*.join()** returns a delimited string from a sequence

```
>>> newsea = sea.replace('Atlantic','Pacific')
>>> newsea
'Pacific Ocean'
>>> words = newsea.split(' ')
>>> words
['Pacific', 'Ocean']
>>> csvwords = ','.join(words)
>>> csvwords
'Pacific,Ocean'
```

New string returned

List returned

Delimiter in the returned string

String returned

String Formatting

- `string.format(args)` returns a new string after formatting `args`
- `string` contains a series of `{position:spec}`
  - Mapped to `args` by position

```
>>> price = 350
>>> tax = 0.07
>>> cost = price + price * tax

>>> 'price {0} = tax {1} * cost {2}'.format(price, tax, cost)
'price 374.5 = tax 0.07 * cost 350'
```

Argument 2

Returned string

Argument 0

String Formatting

- ***spec*** **may specify**
  - Width
  - Formatting type code

n	Width
d	Integer
f	Floating point with precision

```
>>> print '|{0:d}|{1:f}|{2:f}|'.format(price, tax, cost)
|350|0.070000|374.500000|


>>> print '|{0:9d}|{1:9f}|{2:9f}|'.format(price, tax, cost)
|      350| 0.070000|374.500000|
```

Less than 9, pad with spaces

More than 9, no truncation

```
>>> print '|{0:9d}|{1:.2f}|{2:9.2f}|'.format(price, tax, cost)
|      350|0.07|   374.50|
```

2 places after decimal

String Formatting

- *spec* may specify
  - Width
  - Formatting type code

n	Width
d	Integer
f	Floating point with precision

```
>>> '{0:5d} and tax {1:5f} = {2:7f}'.format(price, tax, cost)
'  350 and tax 0.070000 = 374.500000'

>>> '{0:5d} and tax {1:5f} = {2:7.2f}'.format(price, tax, cost)
'  350 and tax 0.070000 =  374.50'

>>> '{0:f} and tax {1:.2f} = {2:.2f}'.format(price, tax, cost)
'350.000000 and tax 0.07 = 374.50'

>>> print 'Final cost is {0:.2f}'.format(cost)
Final cost is 374.50
```

The `string` Module

- Standard library module providing string functions and constants

```
>>> import string
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
>>> string.octdigits
'01234567'
>>> string.capwords('now is the time')
'Now Is The Time'
>>> string.capwords('now_is_the_time','_')
'Now_Is_The_Time'
```

Constants from the module

Function from the module

Special Strings

- **Strings objects may contain escape sequences**
  - Special byte encodings that are described following a backslash, \

`\'`, `\"`, `\\`	Literal single quote, double quote, backslash
`\r`, `\n`	Carriage return, newline
`\t`	Tab
`\0`*num*, `\x`*num*	Character value represented in octal or hexadecimal

- **Raw strings ignore the special meaning of the backslash, **
  - Specified with `r` before the opening quotation mark

```
>>> print '\t is tab'
        is tab
>>> print r'\t is tab'
\t is tab
```

All strings in Python 3 are Unicode

```
print('A\u00f1o')
```

```
print u'A\u00f1o'
```

- Objects and Variables
- Numeric Types and Operations
- String Types and Operations
- **Conditionals**

Simple Comparisons

- **Yield a Boolean `True` or `False` value**
  - The strings `'True'` and `'False'` both evaluate to Boolean `True`
- **Types of conditional expressions**
  - Object identity, `is`
  - Arithmetic relational; e.g., > or ==
  - Strings use the same equality and inequality operators as numeric objects

```
>>> sea = 'Atlantic'
>>> ocean = sea
>>> ocean is sea
True
>>> ocean == sea
True
>>> 7 == 3
False
```

Larger Comparisons

- **Several simple conditions may be chained together to yield an overall Boolean value**
  - Evaluated from left to right
  - All individual conditions must yield `True` for overall truth

All tests yield `True`

```
>>> first = 1
>>> second = 2
>>> third = 3
>>> first < second < third
True
>>> first < second == third
False
```

Second test is `False`

- **Explicit Boolean operators may be more readable**

Compound Comparisons

- **Several simple conditions joined by Boolean operators**
  - `and` yields `True` if both operands are `True`
  - `or` yields `True` if either is `True`
  - `not` reverses the Boolean value

```
>>> first < second and second == third
False
>>> first < second or second == third
True
>>> second is third
False
>>> second is not third
True
```

Both must be `True`

Either yields `True`

Compound Statements

- **Begin with a header statement that is terminated by a colon, :**
- **Followed by a group of statements that are syntactically treated as a unit—a *suite***
  - A code block
  - For example: a loop body
- **Following statements are tied to the header based on the same indentation**
  - One of Python's readability features
  - Python Enhancement Proposal (PEP) 8 recommends indentation of four spaces
- **End of code block detected by lack of indentation**
  - Or an empty line if entering statements into the interpreter

The `if` Statement

- **Evaluates an expression's Boolean value and executes the associated block**
  - `False` is `0`, empty string, empty collection, and `None`; anything else is considered `True`
- **Syntax:**

```
if conditionA:
    blockA
elif conditionB:
    blockB
elif conditonC:
    blockC
else:
    blockD
restOfCode
```

Indentation defines the block

Any number of optional `elif`s may follow

The single `else` is optional

- **The block associated with first condition that yields `True` is executed**
  - The `else:` block is executed if no condition yields `True`

- **Empty and nonempty strings**

Yields a Boolean

Empty terminates block

Condition was False

```
>>> sea = 'atlantic'
>>> if sea:
...     print sea.upper()
...
ATLANTIC
>>> sea = None
>>> if sea:
...     print sea.upper()
... else:
...     print 'does not exist'
...
does not exist
```

Testing Alternatives Using `elif`

- **A series of tests may be combined using `elif`**

```
>>> sea = 'baltic'
>>> if sea == None:
...     print 'sea is empty'
... elif sea == 'atlantic':
...     print sea, 'ocean is green'
... elif sea == 'pacific':
...     print sea, 'ocean is blue'
... elif sea == 'red':
...     print sea, 'sea is red'
... else:
...     print sea, 'sea is unknown'
...
baltic sea is unknown
```

The pass Statement

- **Explicitly does nothing**
  - Null statement
- **Serves as a placeholder where a statement is required**

```
>>> if not sea:
...     pass
... else:
...     print 'I see the', sea
...
I see the baltic
```

Placeholder between `if` and `else`

Hands-On Exercise 2.2

In your Exercise Manual, please refer to
Hands-On Exercise 2.2: Strings and **`if`**

Chapter Summary

You are now able to

- **Write simple Python programs**
  - Create variables
  - Manipulate numeric values
  - Manipulate string values
  - Make decisions

Chapter 3

Collections

Chapter Objectives

After completing this chapter, you will be able to

- **Create and manage collections**
  - Lists, tuples, sets, and dictionaries
- **Perform iteration**

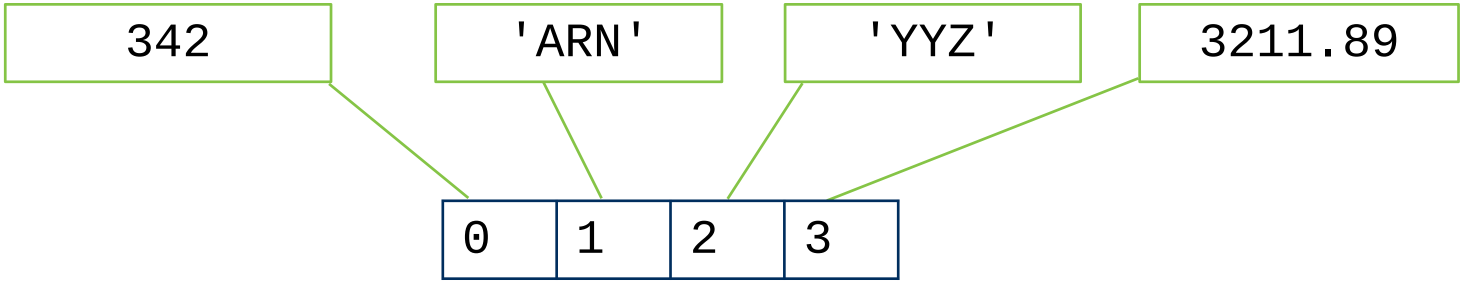
Chapter Contents

Collections

- **Python provides several types of collections**
  - Compound data types or data structures
    - Composed of elements of various types
- **Collections are categorized as**
  1. Sequential
     - Access individual values by a numeric offset
       - Strings, lists, and tuples
  2. Mapped or associative
     - Access individual values by a key
       - Dictionaries
  3. Unordered
     - Sets
- **May be mutable or immutable**
  - Lists, sets, and dictionary values are mutable
  - Strings, tuples, and dictionary keys are immutable

List

- **Core Python type**
  - Similar to an array in other languages
  - No maximum size
- **Contents may be a combination of different types**
  - Numeric literals, string literals, Booleans, and any other type
- **Represented as a comma-delimited series of values within brackets [ ]**
  - `[342, 'ARN', 'YYZ', 3211.89]`



List Indexing

➢ Lists can be assigned

- A sequence of values within [ ]
- Or simply [] to represent an empty list

➢ Contents are accessed with syntax similar to strings

- Numeric offset range described in [ ]

```
>>> flight = [342, 'ARN', 'YYZ', 3211.89]
>>> flight
[342, 'ARN', 'YYZ', 3211.89]

>>> flight[1]
'ARN'
>>> if flight[3] > 3000:
...     print 'Cost exceeds the max'
Cost exceeds the max

>>> tax = 1.10
>>> flight[3] = flight[3] * tax
```

List assignment

Access a single element

Use list elements like any other object

Lists are mutable

List Slicing

- **Consecutive elements can be referenced as a slice**
  - `[`*`start`*`:`*`end`*`]` syntax as with strings
    - `[`*`start`*`:`*`end`*`:`*`step`*`]` syntax references every *`step`*th element in the slice

- **Slice of a list is itself a list**

```
>>> airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
>>> airports[1:3]
['HNL', 'YYZ']
>>> airports[3:]
['NRT', 'CDG']
>>> airports[:]
['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
>>> airports[::2]
['LAX', 'YYZ', 'CDG']
```

Result is a list

List Operators

- The + operator concatenates lists
- The * operator repeats lists

```
>>> north_airports = ['YYZ', 'ARN', 'LHS']
>>> south_airports = ['SYD', 'RIO', 'CPT']

>>> north_airports + south_airports
['YYZ', 'ARN', 'LHS', 'SYD', 'RIO', 'CPT']
>>> north_airports * 2
['YYZ', 'ARN', 'LHS', 'YYZ', 'ARN', 'LHS']
```

List Operations

- List content can be modified by assignment
- The `len()` function returns the number of elements in the list
- The `list(arg)` function returns its argument as a list

```
>>> airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
>>> airports[0] = 'SFO'
>>> airports[1:2] = ['LNY', 'YHZ']
>>> airports
['SFO', 'LNY', 'YHZ', 'YYZ', 'NRT', 'CDG']
>>> destinations = airports
>>> destinations is airports
True
>>> destinations = list(airports)
>>> destinations is airports
False
>>> destinations == airports
True
```

A one-element slice is replaced by a two-element list

Assignment creates shared reference

List is copied

List Methods

- **Method functions allow in-place modification of list contents**
  - *list*.append(*value*)—Add *value* to the end
  - *list*.pop(*n*)—Remove element *n* and return it
  - *list*.insert(*posit*,*value*)—Add *value* at position *posit*
  - *list*.sort() and *list*.reverse()—Change contents sequence
  - *list*.remove(*value*)—Remove first element containing *value*

```
>>> airports = ['SFO', 'LNY', 'YHZ', 'YYZ', 'NRT', 'CDG']
>>> airports[6] = 'LGA'

Traceback (most recent call last):
  File "<pyshell#240>", line 1, in <module>
    airports[6] = 'LGA'
IndexError: list assignment index out of range

>>> airports.append('LGA')
>>> airports
['SFO', 'LNY', 'YHZ', 'YYZ', 'NRT', 'CDG', 'LGA']
```

Cannot extend the list by assigning a new element

List Methods Example

```
>>> airports.pop()
'LGA'
>>> airports
['SFO', 'LNY', 'YHZ', 'YYZ', 'NRT', 'CDG']
>>> airports.insert(0,'AMS')
>>> airports
['AMS', 'SFO', 'LNY', 'YHZ', 'YYZ', 'NRT', 'CDG']
>>> airports.sort()
>>> airports
['AMS', 'CDG', 'LNY', 'NRT', 'SFO', 'YHZ', 'YYZ']
>>> airports.remove('NRT')
>>> airports
['AMS', 'CDG', 'LNY', 'SFO', 'YHZ', 'YYZ']
```

List is mutable

Remove by value

Tuple

- **A sequenced type**
  - Contents are accessed by an offset like a list
- **An immutable type**
  - No change in size or content after creation
- **Normally represented by a comma-delimited list of values within ( )**

```
>>> airports = ('LAX', 'HNL', 'YYZ', 'NRT', 'CDG')
>>> airports[1]
'HNL'
>>> airports[1] = 'LNY'

Traceback (most recent call last):
TypeError: 'tuple' object does not support item assignment
```

Immutable

Tuple

- Parentheses are optional on assignment
- Single element tuple requires a comma on assignment

```
>>> planes = 'A350', 'A380', 'B747', 'B737'
>>> planes
('A350', 'A380', 'B747', 'B737')
>>> biggest_plane = ('A380',)
>>> biggest_plane
('A380',)
>>> oldest_plane = 'B747',
>>> oldest_plane
('B747',)
```

What type of object would `start = (1)` create?

Tuple Operations

- **Consecutive elements are accessed with standard slice notation**
  - Slice of a tuple is itself a tuple
- **+ and * operators concatenate or repeat tuples**
- **The `tuple()` function returns its argument as a tuple**

```
>>> airports = ('LAX', 'HNL', 'YYZ', 'NRT', 'CDG')
>>> airports[1:3]
('HNL', 'YYZ')
>>> airports[1:3] * 2
('HNL', 'YYZ', 'HNL', 'YYZ')
>>> codes = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
>>> destinations = tuple(codes)
>>> destinations
('LAX', 'HNL', 'YYZ', 'HNL', 'YYZ')
>>> airports == destinations
True
>>> airports is destinations
False
```

- **Lists and tuples may contain any type of object**
  - Including lists and tuples
- **A list of lists is similar to a two-dimensional array in some other languages**

```
>>> twocodes = [['AMS', 'SFO'], ['NRT', 'CDG']]

>>> twocodes[0]
['AMS', 'SFO']
>>> twocodes[0][1]
'SFO'
```

Inner lists

Right bracket]
terminates the outer list

What would `tuple(twocodes)` return?

Sequence Unpacking

- **Multiple values from a collection are assigned**
  - Collection slice is allowed

- **Correct number of variables needed to hold all unpacked values**

```
>>> airports = ['LAX', 'HNL', 'YYZ', 'NRT']
>>> depart, layover1, layover2, arrive = airports
>>> layover2
'YYZ'
>>> layover2, arrive = airports[2:]
>>> arrive
'NRT'
```

- **Python 3 allows unpacking using a wildcard variable**
  - References a list of the remaining values

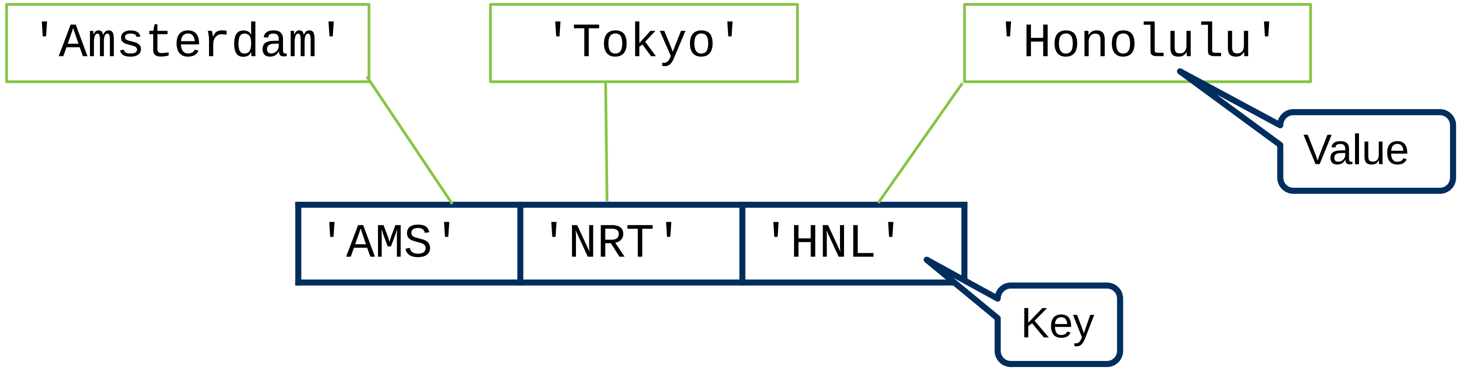
```
>>> depart, *layovers, arrive = airports
```

Hands-On Exercise 3.1

In your Exercise Manual, please refer to
Hands-On Exercise 3.1: Collections and Slicing

Dictionary

- **Associative type**
  - Contents accessed through symbolic keys instead of numeric indices
    - No maximum size
    - Similar to an associative array or hash in other languages
- **Contents may be composed of any combination of types**
  - Numeric literals, string literals, Booleans, and any other type
- **Is represented within curly brackets `{ }` by a comma-delimited series of `key:value` pairs**
  - `{'AMS': 'Amsterdam', 'NRT': 'Tokyo', 'HNL': 'Honolulu'}`



Dictionary Operations

- **Any individual element can be referenced through its key**
  - Value for that key may be retrieved or updated
- **Only the keys are immutable**
  - Change in key implies a new entry for the dictionary
  - Keys may be numeric literals, strings, or tuple elements

```
>>> cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo/Narita'}
>>> cities['NRT']
'Tokyo/Narita'
>>> cities['NRT'] = 'Tokyo'
>>> cities['HNL'] = 'Honolulu'
>>> cities
{'NRT': 'Tokyo', 'HNL': 'Honolulu', 'YYZ': 'Toronto'}
```

Values are mutable

New entry

Dictionary Methods and Functions

- **Method functions allow modification of contents or data retrieval**
  - *`dict`*`.keys()`—Returns a list of keys
  - *`dict`*`.values()`—Returns a list of values
  - *`dict`*`.update(`*`newdict`*`)`—Merges contents of *`newdict`* into *`dict`*
  - *`dict`*`.get(`*`key`*`)`—Returns value at *`key`*, else `None` for undefined *`key`*
  - *`len(dict)`*—Returns the number of items in the dictionary

```
>>> cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}
>>> cities.keys()
['NRT', 'YYZ']
>>> cities.values()
['Tokyo', 'Toronto']
```

Lists

In Python 3, `keys()`, and `values()` return iterable objects

Dictionary Methods Example

```
>>> cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}
>>> asia_cities = {'HKG': 'Hong Kong', 'NRT': 'Narita'}
>>> cities.update(asia_cities)
>>> cities
{'NRT': 'Narita', 'HKG': 'Hong Kong', 'YYZ': 'Toronto'}
>>> cities.get('MSY')
>>> cities['MSY']
Traceback (most recent call last):
cities['MSY']
KeyError: 'MSY'
```

Duplicate keys

Returned None

`KeyError` exception is raised

Creating a Dictionary

- ***`dict`*`.items()`—Returns a list of key–value pairs as tuples**

- **A dictionary can be created from a sequence of key–value pairs**
  - `dict(`*`arg`*`)` returns a dictionary
  - *`arg`* is a sequence containing the key–value pairs

- ***`dict`*` = {}`—Creates an empty dictionary**

```
>>> cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}
>>> cities.items()
[('NRT', 'Tokyo'), ('YYZ', 'Toronto')]

>>> dict([('HNL', 'Honolulu'), ('ARN', 'Stockholm')])
{'HNL': 'Honolulu', 'ARN': 'Stockholm'}
>>> dict((('HNL', 'Honolulu'), ('ARN', 'Stockholm')))
{'HNL': 'Honolulu', 'ARN': 'Stockholm'}

>>> old_cities = dict(cities.items())
```

List of tuples

Tuple of tuples

Duplicate the dictionary

`zip()` Function

- **Combines two collections in parallel and returns a new list**
  - Composed of two element tuples based on position
  - Returned list is the length of the shorter argument

```
>>> countries = ['FR', 'GB', 'CA', 'JP', 'US']
>>> prefixes = [33, 44, 1, 81]
>>> zip(countries, prefixes)
[('FR', 33), ('GB', 44), ('CA', 1), ('JP', 81)]

>>> look_by_country = dict(zip(countries, prefixes))
>>> look_by_country
{'JP': 81, 'FR': 33, 'CA': 1, 'GB': 44}

>>> look_by_prefix = dict(zip(prefixes, countries))
>>> look_by_prefix
{33: 'FR', 44: 'GB', 81: 'JP', 1: 'CA'}
```

List of tuples

Convert to a dictionary

Sets

- **Unsequenced mutable collections of unique, immutable objects**
  - Created with the `set()` function
  - Or by assignment with `{ }`
- **The *`set`*`.add()` and *`set`*`.remove()` methods can add or remove members**

```
>>> hawaii_airports = set(['HNL', 'ITO'])
>>> pacific_airports = {'HNL', 'NRT'}
>>> hawaii_airports.add('LNY')
>>> pacific_airports.add('SYD')
>>> hawaii_airports
set(['ITO', 'LNY', 'HNL'])
>>> pacific_airports
set(['SYD', 'HNL', 'NRT'])
```

- **Support arithmetic-style operators**

`hawaii_airports` **`pacific_airports`**

-	Difference
|	Union
&	Intersection
>	Superset
<	Subset
==	Equality
!=	Inequality

```
>>> hawaii_airports - pacific_airports
set(['ITO', 'LNY])
>>> pacific_airports - hawaii_airports
set(['NRT', 'SYD'])
>>> hawaii_airports | pacific_airports
set(['ITO', 'LNY', 'SYD', 'HNL', 'NRT'])
>>> hawaii_airports & pacific_airports
set(['HNL'])
```

Membership Quiz

- Given the following:

```
>>> codes = {'France': 33, 'Japan': 81,
             'GreatBritain': 44, 'USA': 1}
>>> caps = {'France': 'Paris', 'Cuba': 'Havana',
            'Japan': 'Tokyo'}
```

- How could you determine which keys from codes are also keys in caps using sets?

____

- How could you determine which keys from codes are not keys in caps using sets?

____

- How could these results be assigned to a list?

____

Collection Membership Testing: `in`

- **Syntax: `value in collection`**
  - Returns `True` if the value is a member
  - Works for all collection types and strings
    - Limited to testing keys for dictionaries

```
>>> truth = ('Always', 'test', 'your', 'data')
>>> 'test' in truth
True
>>> advice = {'Always', 'test', 'your', 'coding'}
>>> 'test' in advice
True
>>> list2 = ['This', 'is', 'a', ['good', 'test', 'example']]
>>> 'test' in list2[3]
True
>>> facts = {'test': 'Good idea', 'no test': 'Bad idea'}
>>> 'test' in facts
True
```

Chapter Contents

- Lists, Dictionaries, and Tuples
- `for` Loops and Iterators
- `while` Loops

Flow Control With Loops

- Fixed number of iterations with `for`
- Conditional iterations with `while`
- Loop body is defined by its indentation

```
Loop statement:
     loopStatement1
     loopStatement2
     ...
restOfCode
```

The `for` Loop

- **Steps through a sequence of objects**
  - *`var`* is assigned each object in turn
  - When the sequence is exhausted, exit the loop
    - *`var`* has the final value processed

- **Syntax:**

```
for var in sequence:
    loopBlock
restOfCode
```

Loop Through a Sequence

- **The *sequence* is a series of values**
  - Strings, lists, and tuples
    - Or their slices

```
>>> airports = ['LAX', 'HNL', 'YYZ', 'NRT']
>>> for airport in airports:
...     print airport
...
LAX
HNL
YYZ
NRT
>>> airport
'NRT'
```

Final value processed in the loop

Nested Looping

```
>>> prices = [200, 400, 500]
>>> fees = [20, 50]
>>> totals = []
>>> for fee in fees:
...     for price in prices:
...         totals.append(price - fee)
...
>>> print totals
[180, 380, 480, 150, 350, 450]
```

Loop Through a Dictionary

➢ **Dictionary methods `keys()`, `values()`, and `items()` can provide an iterable sequence**
  - Dictionary name alone provides the keys

```
>>> airports = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}

>>> for code in airports.keys():
...     print code

>>> for code in airports:
...     print code

>>> for value in airports.values():
...     print value

>>> for key, value in airports.items():
...     print key, value
```

Both print
NRT
YYZ

```
Tokyo
Toronto
```

```
NRT Tokyo
YYZ Toronto
```

Membership Quiz With a Loop

- Create a list of keys from codes that are also keys in caps

```
>>> codes = {'France': 33, 'Japan': 81,
             'GreatBritain': 44, 'USA': 1}
>>> caps = {'France': 'Paris', 'Cuba': 'Havana',
            'Japan': 'Tokyo'}
>>> countries = []
>>> for code in codes:
...     if code in caps:
...         countries.append(code)
...
>>> countries
['Japan', 'France']
```

Optional Flow Control Within Loops

- `break` terminates the loop
- `continue` returns flow control to the top of the loop
- `else:` defines a block of code executed after the loop terminates normally
  - Without a `break`

```
for var in sequence:
     loopBlock
     if condition1:
          break
     if condition2:
          continue
else:
     elseblock
restOfCode
```

Optional test to terminate the loop

Optional test to return to the top of the loop

Optional block if `break` was not executed

Using break, continue, and else in a Loop

```
>>> airports = ['LAX', 'HNL', 'YYZ']
>>> for airport in airports:
...     if airport == 'HNL':
...         break
...     print airport
... else:
...     print 'The end', airport
...
LAX
>>> for airport in airports:
...     if airport == 'HNL':
...         continue
...     print airport
... else:
...     print 'The end', airport
...
LAX
YYZ
The end YYZ
```

Terminate with `'HNL'`

Skip with `'HNL'`

Executed if there was no `break`

- **Provides a list of sequential integers**
- **Syntax:**
  - `range(m,n,s)`
    - A list of every *s* value from *m* to *n* `- 1`
    - Both *m* and *s* are optional

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> range(10,5,-1)
[10, 9, 8, 7, 6]
>>> airports = ['LAX', 'HNL', 'YYZ']
>>> for index in range(len(airports))
...     print index, airports[index]
...
0 LAX
1 HNL
2 YYZ
```

Negative step used for decreasing

The `xrange` Function

- **Similar to `range()` but**
  - Does not return a list
  - Returns an object that generates the values on demand
    - More efficient when looping
- **Syntax:**
  - `xrange(`*m*`,`*n*`,`*s*`)`

```
>>> airports = ['LAX', 'HNL', 'YYZ']
>>> for index in xrange(len(airports))
...     print index, airports[index]
...
0 LAX
1 HNL
2 YYZ
```

`range()` is `xrange()`

Iterable Object

- **The `iter()` built-in function returns an iterable object**
- **The `next()` method provides each element**
  - Raises the `StopIteration` exception when sequence is exhausted

```
>>> airports = ['LAX', 'HNL', 'YYZ']
>>> airport_iter = iter(airports)
>>> airport_iter.next()
'LAX'
>>> airport_iter.next()
'HNL'
>>> airport_iter.next()
'YYZ'
>>> airport_iter.next()
Traceback (most recent call last):
 airport_iter.next()
StopIteration
```

Create an iterable object from the list

Method to deliver each element

Exception raised when iteration is complete

Iterating Through a Sequence

- The `for` loop uses the iteration protocol internally

```
>>> for airport in airports:
...     print airport
LAX
HNL
YYZ
```

Uses `iter()` and `next()` internally:

```
>>> airport = iter(airports)
>>> airport.next()
LAX
>>> airport.next()
HNL
```

List Comprehension

- **An *operation* is applied to each element with a `for` loop**
  - Result is a list

- **Syntax: `[`*`operation`*` for `*`var`*` in `*`iterable`*`]`**

```
>>> prices = [200, 400, 500]
>>> fee = 20
>>> totals = [price - fee for price in prices]
>>> print totals[0]
180
>>> for total in totals:
...     print total
...
180
380
480
>>> fees = [20, 50]
>>> [price - fee for fee in fees for price in prices]
[180, 380, 480, 150, 350, 450]
```

`for` loop applies the operation

Result is a list

Nested

List Comprehension With Conditional

- **The *operation* may be executed conditionally**
  - With an embedded `if`

[*operation* for *var* in *iterable* if *condition*]

```
>>> fee = 30
>>> min = 200
>>> [price - fee for price in prices if price > min]
[370, 470]

>>> codes = {'France': 33, 'Japan': 81,
             'GreatBritain': 44, 'USA': 1}
>>> caps = {'France': 'Paris', 'Cuba': 'Havana',
            'Japan': 'Tokyo'}

>>> countries = [code for code in codes if code in caps]
>>> countries
['Japan', 'France']
```

Keys from codes that are also keys in caps

Generator Comprehension

- **Creates an iterable object that supports the `next()` method**
- **The *`operation`* is applied to each element with a `for` loop when requested using `next()`**
  - No list of all results created
- **Syntax: (*`operation`* `for` *`var`* `in` *`iterable`*)**

```
>>> prices = [200, 400, 500]
>>> fee = 20
>>> totals = (price - fee for price in prices)
>>> totals.next()
180
>>> for total in totals:
...     print total
...
380
480
```

Result is an iterable object

Additional Comprehensions Backported from Python 3

- **Set comprehensions are available in Python 2.7**

  {*operation* for *var* in *set* if *condition*}

```
airports = {'LAX', 'HNL', 'YYZ'}
hawaiiairports = {airport for airport in airports
                  if airport in ['HNL', 'ITO']}
print hawaiiairports
```

```
{'HNL'}
```

Result is a set

- **Dictionary comprehensions are also available in Python 2.7**

  {*key*: *value* for *key*, *value* in *sequence* if *condition*}

```
airports = {'LAX': 'Los Angeles', 'HNL': 'Honolulu',
            'YYZ': 'Toronto'}
hawaiidict = {code: city for code, city in airports.items()
              if code in ['HNL', 'ITO']}
print hawaiidict
```

```
{'HNL': 'Honolulu'}
```

Result is a dictionary

- Lists, Dictionaries, and Tuples
- `for` Loops and Iterators
- ➢ **`while` Loops**

The `while` Loop

- **Evaluates the Boolean value of an expression**
  - Executes the loop body so long as the expression is `True`
    - Terminates on `False` or execution of a `break`

- **Syntax:**

```
while expression:
    loopBlock
    if condition1:
        break
    if condition2:
        continue
else:
    elseblock
restOfCode
```

Optional test to terminate the loop

Optional test to skip to next iteration

Optional block if `break` was not executed

while Loop Example

```
>>> count = 0
>>> while count <= 5:
...     if count == 2:
...         count += 1
...         continue
...     print count
...     count += 1
... else:
...     print 'Made it past 5', count
...
0
1
3
4
5
Made it past 5 6
```

Condition is evaluated before each pass through the loop body

Hands-On Exercise 3.2

In your Exercise Manual, please refer to
Hands-On Exercise 3.2: Dictionaries, Sets, and Looping

Chapter Summary

You are now able to

- **Create and manage collections**
  - Lists, tuples, sets, and dictionaries
- **Perform iteration**

Chapter 4

Functions

Chapter Objectives

After completing this chapter, you will be able to

- **Create and call a simple function**
- **Use anonymous lambda functions**
- **Apply generator functions**

- **Defining and Calling**
- **Lambda Functions**
- **Generators**

Function Overview

- **A block of statements that execute as a unit when called and may have**
  - A name
  - An argument list
  - A `return` statement
- **A generic, reusable unit that simplifies program design**
  - Breaks a solution into smaller pieces
  - Hides internal details
  - Provides a single place for modification
- **Created by a `def` statement**
  - Assigns a name to a function
  - Compound statement
  - Indentation defines the function body

Simple Function Example

```
def print_one():
    num = 1
    print 'the value of num is', num

def print_two():
    num = 2
    print 'the value of num is', num
```

Function is defined and named

```
>>> print_one()
the value of num is 1
>>> print_two()
the value of num is 2
>>> print_one
<function print_one at 0x011BFF70>
```

Function is called by its name

Function is an object at a memory location

Passing Data Into a Function

- **Arguments are passed to the function when called**
- **Function receives arguments from the parameters specified on the `def` statement when the function is executed**
- ***Positional* parameters are mapped to the argument list based on their position when the function is called**

```
def printposit(depart, arrive):
    print 'depart and arrive by position:', depart, arrive
```

Arguments

```
>>> printposit('NRT', 'HNL')
depart and arrive by position: NRT HNL
```

Keyword Parameters

- **Are mapped to the argument list based on their names**
  - Function call determines keyword or positional style
  - Optional default values may be assigned

```
def printkey(depart, arrive):
    print 'depart and arrive by keyword:', depart, arrive


def printdef(depart='LAX', arrive='HNL'):
     print 'depart and arrive defaults:', depart, arrive
```

Specify default parameter values

```
>>> printkey(arrive='HNL', depart='NRT')
depart and arrive by keyword: NRT HNL

>>> printdef(depart='AMS')
depart and arrive defaults: AMS HNL
```

Keyword arguments may be passed in any order

Default is applied for `arrive` within the function

Variable-Length Parameter Lists

- Functions may be written to accept any number of arguments
- Parameter name preceded by `*` will hold all remaining positional arguments in a tuple
- Parameter name preceded by `**` will hold all remaining keyword arguments in a dictionary
- Function header syntax:
  - Positional and keyword without defaults must be the leftmost
  - Keywords with defaults follow
  - A single `*parameter` follows
  - A single `**parameter` is rightmost

Parameter list

- Function call syntax:
  - Positional arguments must be the leftmost
  - Keyword arguments follow

Argument list

Keyword parameters may follow after the *parameter or **parameter in Python 3

Variable-Length Parameter List Example

```
def printargs(*args, **kwargs):
    print 'Positional', args
    print 'Keyword', kwargs
```

Positional arguments are in a tuple

```
>>> printargs('Jean', 35, 97.85)
Positional ('Jean', 35, 97.85)
Keyword {}
>>> printargs(name='Jean', age=35, rate=97.85)
Positional ()
Keyword {'age': 35, 'name': 'Jean', 'rate': 97.85}
>>> printargs('Employee', name='Jean', age=35, rate=97.85)
Positional ('Employee',)
Keyword {'age': 35, 'name': 'Jean', 'rate': 97.85}
>>> printargs( name='Jean', age=35, rate=97.85, 'Employee')
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Keyword arguments are in a dictionary

Variable-Length Argument Lists

- **Functions may be called with a sequence or dictionary argument**
- **Argument name preceded by * will pass a collection as a sequence of positional parameters**
- **Argument name preceded by ** will pass a dictionary as keyword parameters**

```
>>> employee1 = ['Jean', 35, 97.85]
>>> employee2 = {'name': 'Jules', 'age': 29, 'rate': 89.99}
>>> printargs(*employee1)
Positional ('Jean', 35, 97.85)
Keyword {}
>>> printargs(**employee2)
Positional ()
Keyword {'age': 29, 'name': 'Jules', 'rate': 89.99}
>>> printargs(employee2)
Positional ({'age': 29, 'name': 'Jules', 'rate': 89.99}, )
Keyword {}
```

`employee2` without ** is a single positional

Mutable and Immutable Arguments

➢ **An argument that references a mutable object may have its referenced object changed**

```
def addone(num, handle):
    handle.append('ONE')
    num += 1
    print handle, num
```

num

14539024
int
1

handle

14539434
list
['Sam', 'ONE']

```
>>> name = ['Sam']
>>> count = 0
>>> addone(count, name)
['Sam', 'ONE'] 1
>>> print name, count
['Sam', 'ONE'] 0
```

count

14539010
int
0

name

Object reference by name is mutable

Object reference by `count` is immutable

Enclosed Functions

➢ **A function definition may be within another function**

```
def logdata ():
    def print_header():
        print 'Report starting'
    def print_footer():
        print 'End of report'
    print_header()
    print 'Log data'
    print_footer()
```

Enclosed functions

```
>>> logdata()
Report starting
Log data
End of report
```

Scope

- **Namespace where an object is known**
- **Based on the location of the assignment**
  1. Within an enclosed function
  2. Within an enclosing function
  3. Outside any function

```
var = 'global'
def fun1():
    var = 'enclosing'
    print var
    def fun2():
        var = 'local'
        print var
    fun2()

print var
fun1()
```

Enclosing function

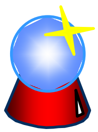
Enclosed function

```
global
enclosing
local
```

LEGB Rule

- **Describes attribute resolution order**

1. **Local: within a function**
2. **Enclosing: within an enclosing function**
3. **Global: within the module or file**
4. **Built-in: within the Python `builtin` module**



Covered in
later sections

- **A new reference is created for each argument**
  - Created when the function is called
  - Removed when the function completes

- **Parameters are local to the function**

`increment()` scope

```
def increment(number):
    number += 1
    print 'function number is', number
```

Global scope

```
number = 5
increment(number)
print 'global number is', number
```

```
function number is 6

global number is 5
```

`global` Statement

- **Declares a variable as a reference to a global object**
  - For duration of the code block

`area()` scope

```
def area(radius):
    global pi
    print pi * radius ** 2
    pi = pi + 1
    print pi
```

Global scope

```
pi = 3.14
area(1)
print 'global pi is', pi
```

```
3.14
4.14

global pi is 4.14
```

14539132
float
3.14
4.14

Global object was modified by reference in the function

`return` Statement

- **Terminates the function execution**
  - Control returns to the point of the call
- **Optionally includes values sent back to the caller**
  - Or `None` if no value is explicitly returned
- **Is optional**
  - Function terminates at the end of the indented block
  - `None` is returned

Function `return` Example

```
def addtwice(num):
    return num + num

def double_vals(arg):
    return arg, arg * 2
```

Return a reference to a single object

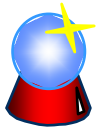
Return a sequence

```
>>> ans = addtwice(3)
>>> ans
6

>>> first, second = double_vals('a')
>>> first
'a'
>>> second
'aa'
```

Assigned the returned value

Unpack a sequence

- **A single function can work with many types**
  - An example of *polymorphism*
  - Any type of objects may be passed as arguments
  - Any type of object may be returned
- **Only operations within the function are type-dependent**
  - Otherwise, Python raises an exception



```
def twice(parm):
    return parm + parm
```

```
>>> twice(5.5)
11.0
>>> twice(['a', 'list'])
['a', 'list', 'a', 'list']
>>> twice({'firstname': 'Robert', 'lastname': 'Johnson'})
TypeError: unsupported operand type(s) for +: 'dict' and
'dict'
```

Functions as Arguments

➢ A function is an object

- Name is a reference to that object
- Can be used as an argument

```
def print_german():
    print 'Guten Morgen!'

def print_italian():
    print 'Buon Giorno!'

def print_greeting(lang, printer):
    print 'Good Morning in', lang, 'is',
    printer()
```

Reference to the function object used as an argument

Call the function

```
>>> print_greeting('German', print_german)
Good Morning in German is Guten Morgen!
>>> print_greeting('Italian', print_italian)
Good Morning in Italian is Buon Giorno!
```

Hands-On Exercise 4.1

In your Exercise Manual, please refer to
Hands-On Exercise 4.1: Creating and Calling Functions

Chapter Contents

- Defining and Calling
- ➢ **Lambda Functions**
- Generators

The Lambda Expression

- **Creates an anonymous function that is an expression**
  - Reference may be assigned
- **Syntax:**
  - `lambda` *`args`*`:` *`expression`*
- **Used in the same way as regular functions**
  - Arguments may be passed in
  - *`expression`* result is returned

Reference to the `lambda` function object

```
>>> addfirst = lambda num: (num + num) * 2
>>> addfirst
<function <lambda> at 0x18BA73F90C12>
>>> addfirst(3)
12
>>> addfirst(4)
16
```

Argument to the `lambda` function

The Lambda Expression

- May be used where statements are not syntactically allowed

```
>>> applydisc = {
...      'cruise': lambda price: price - 5 ,
         'flight': lambda price: price - 10,
         'train' : lambda price: price - 1 }

>>> applydisc['cruise'](100)
95
>>> applydisc['flight'](100)
90
>>> applydisc['rocket'] = lambda price: price ** 2
>>> applydisc['rocket'](100)
10000
```

Each anonymous function is a dictionary value

Dictionary keys

Argument to the lambda function

Hiding Function Calls in Lambda Expressions

- ***`function(args)`*** **can be hidden within a `lambda` expression**
  - Executed when the `lambda` is executed
    - Not when the `lambda` is created

```
def print_german(name):
    print 'Guten Morgen!', name

def print_italian(name):
    print 'Buon Giorno!', name

def print_greeting(lang, printer):
    print 'Good Morning in', lang, 'is',
    printer()
```

Reference to the `lambda` object used as an argument

Executes the `lambda`

Function call with argument is the `lambda` body

```
>>> print_greeting('German', lambda: print_german('Hans'))
Good Morning in German is Guten Morgen! Hans
>>> print_greeting('Italian', lambda: print_italian('Gina'))
Good Morning in Italian is Buon Giorno! Gina
```

- Defining and Calling
- Lambda Functions
- ➢ **Generators**

Generator Function

- **Returns a series of results**
  - One with each call
- **Maintains its state between calls**
  - Subsequent calls continue where previous call stopped
  - Without using any persistent object
- **Created with typical `def` function header**
- **Contains a `yield` statement**
  - Delivers the series of values to the caller
  - Pauses execution
- **Supports the iteration protocol**
  - The `next()` method retrieves a value

Generator Function Example

```
def gen_next_day(today):
    wk = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
    while True:
        yield wk[today]
        if today == 6:
            today = 0
        else:
            today += 1
```

`yield` defines the function as a generator function

days is the iterable object

```
>>> days = gen_next_day(5)
>>> days.next()
'Fri'
>>> days.next()
'Sat'
```

First execution from top of function to `yield`

Subsequent executions proceed to next `yield`

`next()` method retrieves each item

Stopping Iteration

- **A `return` statement will terminate the generator function**
  - Raises a `StopIteration` exception
  - Only None may be returned

```
def gen_next_day(today):
    wk = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
    while True:
        yield wk[today]
        if today == 6:
            return
        else:
            today += 1
```

```
>>> days = gen_next_day(5)
>>> for day in days:
...     print day
...
'Fri'
'Sat'
```

Calls `next` and handles `StopIteration` internally

Hands-On Exercise 4.2

In your Exercise Manual, please refer to
Hands-On Exercise 4.2: Lambda and Generator Functions

You are now able to

- Create and call a simple function
- Use anonymous lambda functions
- Apply generator functions

Chapter 5

Object-Oriented Programming

Chapter Objectives

After completing this chapter, you will be able to

- **Define a class**
- **Create subclasses through inheritance**
- **Attach methods to classes**
- **Overload operators**

- ➢ **Classes and Instances**
- **Inheritance**
- **Additional Classes and Methods**

- **A generic description of an object—a *type***
  - A template for objects
- **A container**
  - Attributes that describe the object’s state
  - Methods that describe the object's behavior
- **Main building block of an Object-Oriented Programming (OOP) solution**

```
class Cruise has:

        ship
        cabin
        balance

        dine ()
        dance ()
        swim ()
```

Attributes

Methods

The `class` Statement

- **Creates a new object template**
- **Assigns a name to the class**
  - PEP 8 recommends *`CapWords`* style

A docstring is customary to describe the class

```
>>> class Cruise(object):
...     ''' This class describes a cruise.'''
...
>>> Cruise
<class '__main__.Cruise'>
```

A class is an object

The `__init__()` Method

- **Called automatically when an instance is created**
  - A *constructor*
  - Python calls *`class`*`.__init__(`*`instance, args`*`)`
- **Used to assign initial attribute values**
  - Based on its argument list
    - Defaults may be provided

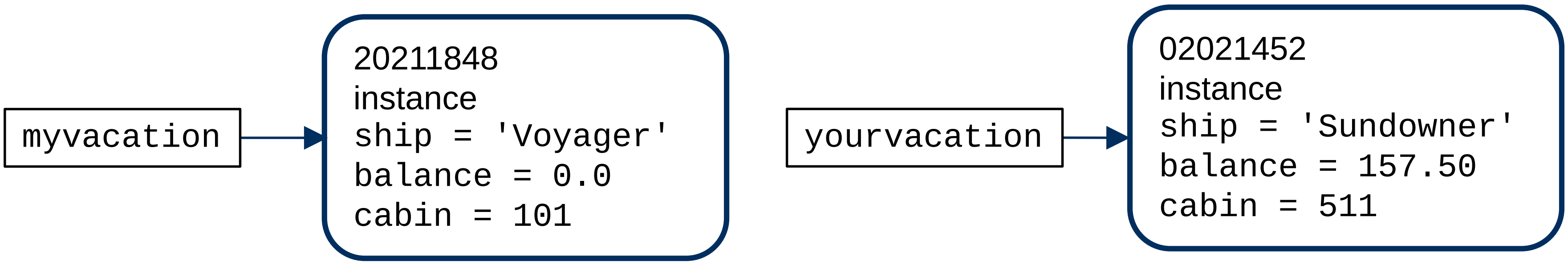
```
class Cruise(object):
    ''' This class describes a cruise.'''
    def __init__(self, ship=None, balance=0.0, cabin=0):
        self.ship = ship
        self.balance = balance
        self.cabin = cabin
```

Keyword parameters with defaults

The `self` Argument

- **References the particular instance making the call**
  - First argument of a method

```
myvacation = Cruise(ship='Voyager', cabin=101)

yourvacation = Cruise(ship='Sundowner',
                      balance=157.50, cabin=511)
```



`__init__()` Parameter Styles

- **Keyword or positional parameters may be used**

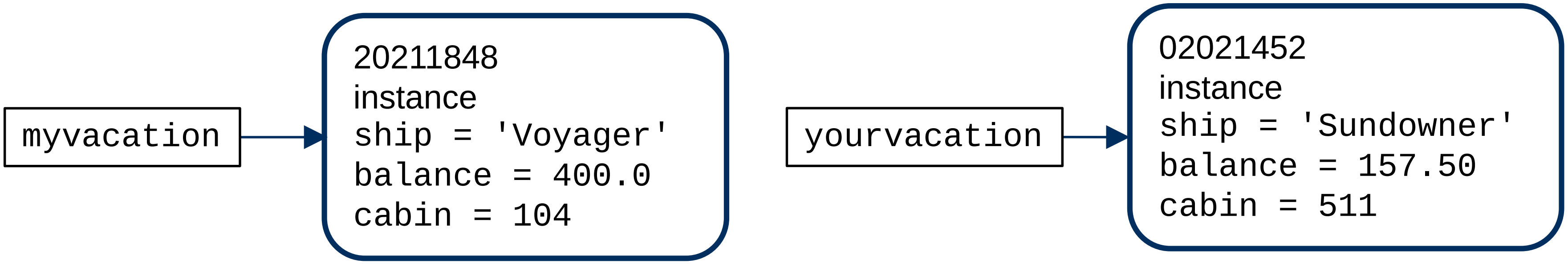
```
class Cruise(object):
    ''' This class describes a cruise.'''
    def __init__(self, shipname, bal, room):
        self.ship = shipname
        self.balance = bal
        self.cabin = room

myvacation = Cruise(shipname='Voyager', bal=0, room=101)

yourvacation = Cruise('Sundowner', 157.50, 511)
```

Attribute names

Modifying Instance Attributes

- **Assigned into the instance namespace**
  - Affects only that instance

```
myvacation.balance = 400.0

myvacation.cabin = 104
```



➢ Functions bound to a class

- Created by a `def` statement within the `class` statement
- Provide the interface for the class
- Are available for any instance

```
class Cruise(object):
    ''' This class describes a cruise.'''
    def __init__(self, ship=None, balance=0.0, cabin=0):
        self.ship = ship
        self.balance = balance
        self.cabin = cabin

    def dine(self, amount):
        self.balance += amount
```

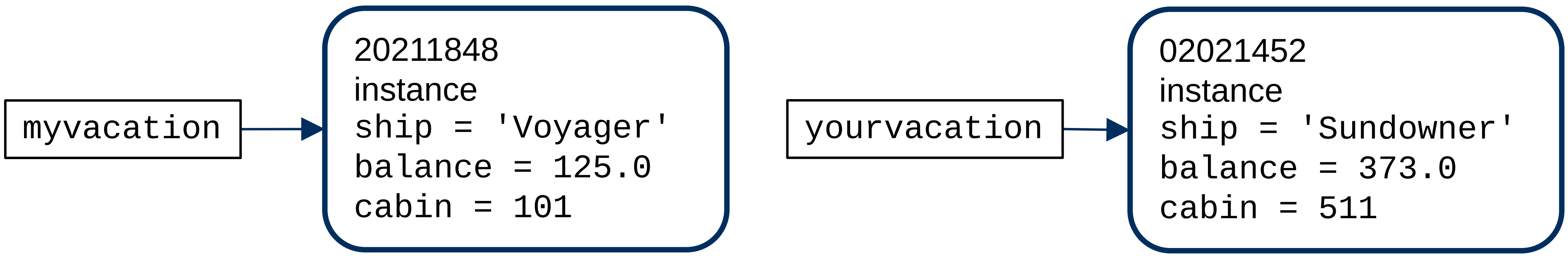
Attribute names

Modifies the instance attribute

Methods Illustrated

- **Called as *instance.method(args)***
  - Python changes to *class.method(instance, args)*

```
myvacation = Cruise(ship='Voyager', cabin=101)
yourvacation = Cruise(ship='Sundowner',
                      balance=157.50, cabin=511)

myvacation.dine(125.0)
yourvacation.dine(215.50)
```



Class Variables

- **Variables encapsulated within a class**
- **Accessed through class name qualification**

```
class Cruise(object):
   def __init__(self, ship=None, balance=0.0, cabin=0):
       self.ship = ship
       self.balance = balance
       self.cabin = cabin
   discountcabins = (101, 102, 105, 106, 109, 110)
   def discount(self):
       if self.cabin in Cruise.discountcabins:
           self.balance -= 50.0
```

Class variable

Check instance attribute for membership in a class variable

Hands-On Exercise 5.1

In your Exercise Manual, please refer to
Hands-On Exercise 5.1: Classes and Initialization

- Classes and Instances
- ➢ **Inheritance**
- Additional Classes and Methods

Describe the *Is A* relationship

- Derived/subclass is an extension of the parent/base class
  - Subclass performs class/type specific operations

Syntax:

```
class BaseClass(object):
    ...
class SubClass1(BaseClass):
```

```
class Trip(object):
    ...
class Cruise(Trip):
    ...
class Flight(Trip):
    ...
```

Both inherit from `Trip` class

Class Inheritance

➢ **Methods and attributes from the parent class are available in subclasses**

```
class Trip(object):
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday
    def print_departure(self):
        print 'Trip leaves on', self.departday

class Cruise(Trip):
    def print_schedule(self):
        print 'Cruise', self.departday, 'to', self.arriveday

class Flight(Trip):
    def print_arrival(self):
        print 'Flight arrives on', self.arriveday
```

Inheritance Hierarchy

- **Subclasses *without* `__init__()` call the parent class `__init__()`**

```
voyage = Cruise(departday='Friday', arriveday='Monday')
voyage.print_departure()
voyage.print_schedule()
```

Method within `Cruise`

Method inherited from `Trip`

```
Trip leaves on Friday
Cruise Friday to Monday
```

```
flthome = Flight(departday='Monday', arriveday='Monday')
flthome.print_departure()
flthome.print_arrival()
```

Method within `Flight`

```
Trip leaves on Monday
Flight arrives on Monday
```

Subclass Instance Initialization

```
class Trip(object):
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

class Cruise(Trip):
    def __init__(self, ship=None, departday=None,
                 arriveday=None):
        self.ship = ship
        Trip.__init__(self, departday=departday,
                      arriveday=arriveday)

class Flight(Trip):
    def __init__(self, plane=None, departday=None,
                 arriveday=None):
        self.plane = plane
        self.departday = departday
        self.arriveday = arriveday
```

Assign ship attribute only

Assign all attributes in `Flight` constructor

Subclass Extension

- **Subclasses may add additional attributes**
- **Subclasses *with* `__init__()` may call the parent class `__init__()`**
  - Not called automatically

```
voyage = Cruise(departday='Friday', arriveday='Monday',
                ship='Sea Breeze')

flthome = Flight(departday='Monday', arriveday='Monday',
                 plane='CRJ')

print voyage.departday
print voyage.ship
print flthome.departday, flthome.plane
```

From `Trip` class

From `Cruise` class

From `Flight` class

```
Friday
Sea Breeze
CRJ Monday
```

The super() Function

- **Returns an object that delegates methods to a parent class**
  - Without explicitly naming the parent class
- **`super(`*`class, object`*`)`**
  - *`class`* is the subclass name
  - *`object`* is an instance of that subclass

```
class Parent(object):
    def __init__(self, ...

class Subclass(Parent):
    def __init__(self, ..
        super(Subclass, self).__init__( ...
```

Calls the constructor from its parent class

Subclass Instance Initialization Using super()

```
class Trip(object):
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

class Cruise(Trip):
 def __init__(self, ship=None, departday=None,
             arriveday=None):
        self.ship = ship
        super(Cruise, self).__init__(departday=departday,
                                     arriveday=arriveday)

class Flight(Trip):
    def __init__(self, plane=None, departday=None,
                arriveday=None):
        self.plane = plane
        self.departday = departday
        self.arriveday = arriveday
```

Assigned in parent class

Assigned in the subclass

Subclass Attributes

- **Hide the same named attributes of the parent class**

```
voyage = Cruise(departday='Friday', arriveday='Monday',
                ship='Sea Breeze')
print voyage.departday, voyage.ship
```

```
Friday Sea Breeze
```

Inherited from `Trip`

```
flthome = Flight(departday='Monday', arriveday='Monday',
                 plane='CRJ')
print flthome.departday, flthome.plane
```

```
Monday CRJ
```

Hide same named attribute in `Trip`

Overriding Methods

- **Single operation name may replace the same named operation from a parent class**
- **Attribute lookup order determines which is found first**

```
class Trip(object):
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

    def print_departure(self):
        print 'Trip leaves on', self.departday

class Cruise(Trip):
    def print_departure(self):
        print 'Cruise', self.departday, 'to', self.arriveday
```

Trip objects call this method

Cruise objects call this method

Overriding Methods

- Instances find the `departure()` method from their class hierarchy

```
vacation = Trip(departday='Sunday', arriveday='Monday')
vacation.print_departure()
```

```
Trip leaves on Sunday
```

```
day1 = Cruise(departday='Monday', arriveday='Monday')
day1.print_departure()
```

```
Cruise Monday to Monday
```

```
day2 = Cruise(departday='Tuesday', arriveday='Tuesday')
day2.print_departure()
```

```
Cruise Tuesday to Tuesday
```

```
plans = [vacation, day1, day2]
for plan in plans:
        plan.print_departure()
```

```
Trip leaves on Sunday
Cruise Monday to Monday
Cruise Tuesday to Tuesday
```

Extending Methods

- **Methods in subclass perform type-specific operations**
  - Parent class provides common operations
  - `super()` may be used to access the parent's methods

Trailing comma suppresses `print`'s newline

```
class Trip(object):
    def __init__(self, departday=None, arriveday=None):
        ...
    def print_trip(self):
        print 'Schedule is', self.departday, self.arriveday,

class Cruise(Trip):
    def __init__(self, ship=None, departday=None,
                 arriveday=None):
        ...
    def print_trip(self):
        super(Cruise, self).print_trip()
        print 'Ship is', self.ship
```

Call method in parent class

Handle class specific task

Extending Methods

```
class Flight(Trip):
    def __init__(self, plane=None, departday=None,
                 arriveday=None):
    ...
    def print_trip(self):
        super(Flight, self).print_trip()
        print 'Plane is', self.plane

travels = [Cruise(departday='Friday', arriveday='Saturday',
           ship='Moonbeam'),
           Flight(departday='Wednesday', arriveday='Friday',
           plane='CRJ')]

for travel in travels:
    travel.print_trip()
```

Call method in subclass

```
Schedule is Friday Saturday Ship is Moonbeam
Schedule is Wednesday Friday Plane is CRJ
```

Multiple Inheritance

- **Class inherits from more than one parent class**
  - Precedence specified in the `class` statement in left-to-right order

```
class Person(object):
   name = 'Bob'
   age = 27

class City(object):
   name = 'New York'
   zip = 10002

class Meeting(Person, City):
   day = 'Monday'
```

```
interview = Meeting()
print interview.age
27


print interview.zip
10002


print interview.day
Monday

print interview.name
Bob
```

Overloaded Operators

- **Operator implementation is based on the type of its arguments**
- **Implemented with special methods named as `__method__`( )**
- **Class method `__add__` will be called if + is used by an instance**
  - `num1 + num2` is implemented as `num1.__add__(num2)`

Overloaded Operators Example

- **Intercept the overloaded + operator and assign a new meaning**
  - Add a value to each reference in the `Listmgr` object
  - Return a new list

```
class Listmgr(object):
    def __init__(self, initial_list):
        self.initial_list = initial_list

    def __add__(self, value):
        retlist = []
        for element in self.initial_list:
            retlist.append(element + value)
        return retlist


nums = Listmgr([100, 50, 250])
ans = nums + 5
```

Internal __add__( ) intercepts + operator

Returns a list

```
[105, 55, 255]
```

Hands-On Exercise 5.2

In your Exercise Manual, please refer to Hands-On Exercise 5.2: Inheritance

- Classes and Instances
- Inheritance
- **Additional Classes and Methods**

Decorator Function

- **Able to perform pre- and post-function processing**
- **Wrapper function**
  - Receives a function as an argument
  - Returns a function
- ***@decorator_name***

```
def logit(original_fun):
    def new_fun():
        print 'Start logging'
        original_fun()
        print 'Stop logging'
    return new_fun

@logit
def print_status():
    print 'Processing'

print_status()
```

```
Start logging
Processing
Stop logging
```

Equivalent to:
`print_status = logit(print_status)`

Class Methods

- **Functions that operate on the class itself**
  - Use `cls` as first parameter as a reference to the class

```
class Cruise(object):
    discount = 0.5
    @classmethod
    def adjust_discount(cls, num):
        cls.discount = num
class Sunsetsail(Cruise):
    pass

print 'Cruise', Cruise.discount
print 'Sunsetsail', Sunsetsail.discount
Cruise.adjust_discount(.10)
Sunsetsail.adjust_discount(.25)
print 'Cruise', Cruise.discount
print 'Sunsetsail', Sunsetsail.discount
```

Decorator identifying the class method

```
Cruise 0.5
Sunsetsail 0.5


Cruise 0.1
Sunsetsail 0.25
```

Static Methods

Reference

- **Functions contained within a class**
  - Do not operate on an instance
    - No `self` parameter

```
class Cruise(object):
    discount = 0.5
    @staticmethod
    def adjust_discount(num):
        Cruise.discount = num
class Sunsetsail(Cruise):
    pass

print 'Cruise', Cruise.discount
print 'Sunsetsail', Sunsetsail.discount
Cruise.adjust_discount(.10)
Sunsetsail.adjust_discount(.25)
print 'Cruise', Cruise.discount
print 'Sunsetsail', Sunsetsail.discount
```

Decorator identifying the static method

```
Cruise 0.5
Sunsetsail 0.5


Cruise 0.25
Sunsetsail 0.25
```

Abstract Class

Reference

- **Class that cannot be instantiated**
  - Must be inherited by a concrete subclass
- **May contain abstract methods**
  - Must be implemented by the subclass

- **Provides tools to implement Abstract Base Classes (ABCs)**
  - ABCMeta—metaclass for ABCs
    - Metaclasses set up classes
  - abstractmethod—decorator function that requires abstract methods are overridden

```
from abc import ABCMeta, abstractmethod
class Trip(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def show_msg(self):
        pass

class Cruise(Trip):
    def show_msg(self):
        print 'Anchors aweigh'

night_trip = Cruise()
night_trip.show_msg()
```

Abstract method

Subclass must provide this method

```
Anchors aweigh
```

Chapter Summary

You are now able to

- **Define a class**
- **Create subclasses through inheritance**
- **Attach methods to classes**
- **Overload operators**

Chapter 6

Modules

Chapter Objectives

After completing this chapter, you will be able to

- **Create new modules**
- **Access additional modules**
- **Use the standard library**

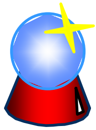
- **Module Overview**
- `import` **and Namespace**
- `from` **and Namespace**
- **The Standard Library**

Module

- **Highest-level programming unit**
  - Modules have classes and functions
  - Functions have statements
  - Statements have expressions
- **Library providing a set of services**
  - Included functions provide each service
- **Single container of reusable code**
  - One place to manage changes
  - May be shared with other modules
    - Reduces repetition

Module Files

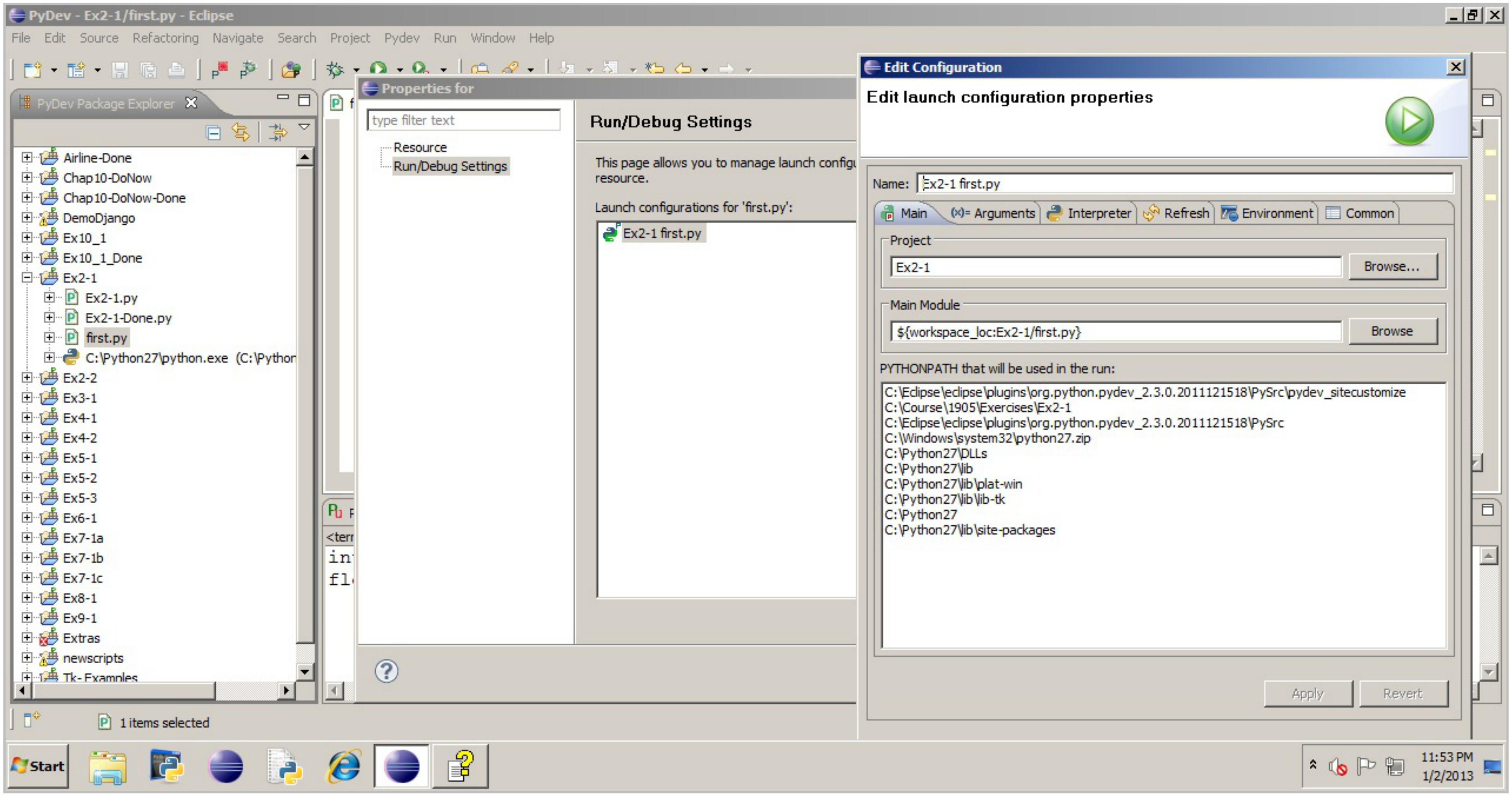
- **Could be**
  - Source code file, *mod*`.py`, or byte code file, *mod*`.pyc`
  - Dynamically linked library, *mod*`.dll` or *mod*`.so`
    - Extension modules



- **Located by a Python search in**
  - The current directory
  - One of the directories contained in `PYTHONPATH`
  - The standard library
  - Directories specified in `.pth` files
    - Contain pathnames to module files

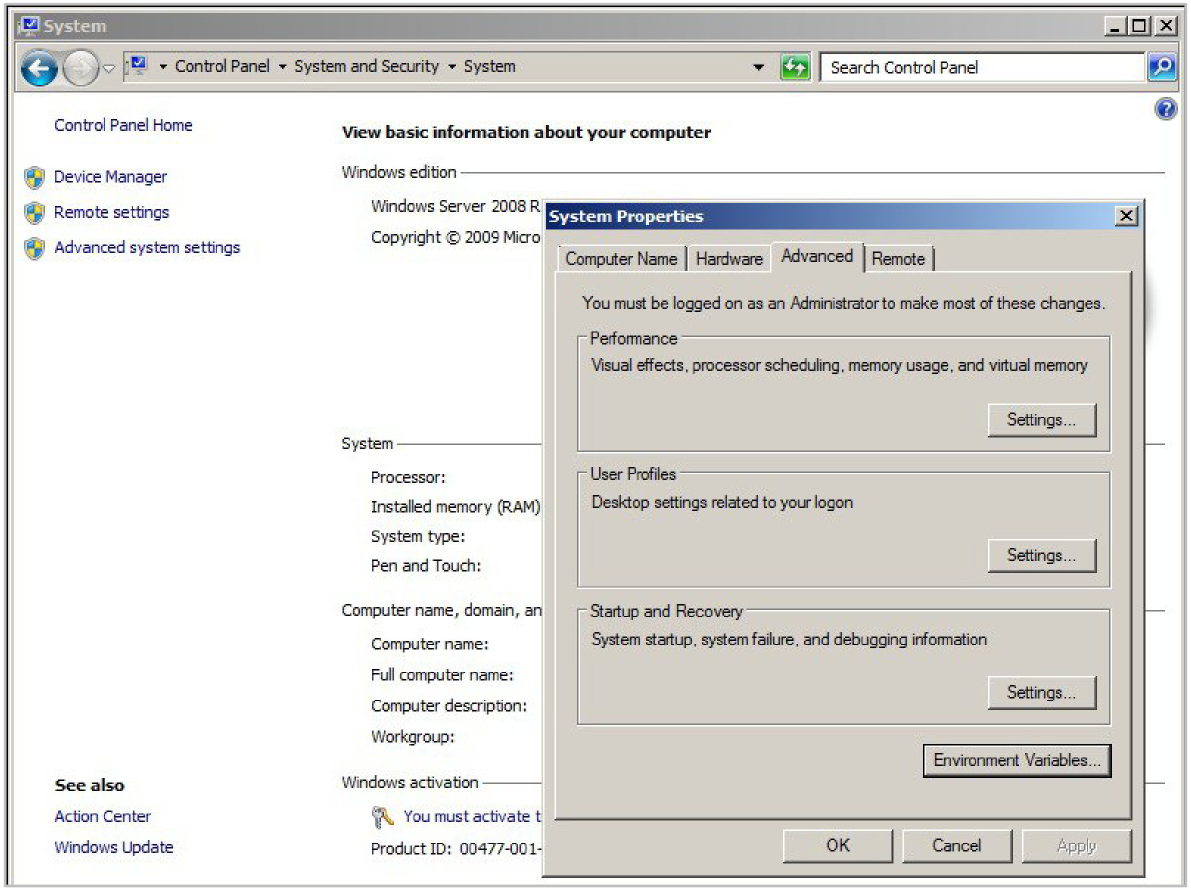
PYTHONPATH in Eclipse

- **Choose the Properties option for a source-code file**
  - Run/Debug Settings



PYTHONPATH in Windows Server 2008

- **From Control Panel | System and Security | System**
  - Environment Variables



- Module Overview
- `import` and Namespace
- `from` and Namespace
- The Standard Library

The `import` Statement

- **Syntax: `import` *`modulename`***
- **Creates an object of the module's contents**
- **Enables access to the *`modulename`*'s classes and functions**
- **Possibly creates the `.pyc` byte code file**
  - If the corresponding `.py` file is newer, recompile the `.pyc`
- **Executed once per process**
  - Later `imports` of same module name use the existing object

Module Execution

- All unenclosed statements are executed
- Attributes are created
  - `functions` or `objects`
- Occurs in a separate namespace
- Creates a module object
  - Based on the file name

mod1.py

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

Statements execute

Attribute

```
>>> import mod1
starting mod1
>>> mod1
<module 'mod1' from 'mod1.pyc'>
```

Module object created

Module Attributes

- Reside within the module namespace
- Are accessed through a qualified name
  - *`module.attribute`*

```
>>> mod1.name
'Guido'
>>> mod1.printname(3)
GuidoGuidoGuido
```

- Qualified attributes reference the proper module

mod1.py

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

mod2.py

```
"""
Another simple module
"""
print 'starting mod2'
import mod1

def printname():
    print mod1.name, 'in mod2'
```

```
>>> import mod1
starting mod1
>>> import mod2
starting mod2
>>> mod1.printname(1)
Guido
>>> mod2.printname()
Guido in mod2
```

Qualified

Chained `imports`

- **Imported file contains an `import` statement**
  - A imports B; B imports C
- **Require two levels of qualification to get embedded attributes**
  - From A, use *`B.C.attribute`*

mod1.py (C)

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

mod2.py (B)

```
"""
Another simple module
"""
print 'starting mod2'
import mod1

def printname():
    print mod1.name, 'in mod2'
```

(A)

```
>>> import mod2
starting mod2
starting mod1
>>> mod2.mod1.name
'Guido'
```

Two levels of qualification

The `import as` Statement

- **Syntax: `import` *`modulename`* `as` *`name`***
  - Access contents using *name* instead of *modulename*
  - Identifier *name* is not restricted by operating system file name

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

```
>>> import mod1 as m
starting mod1
>>> m.printname(2)
GuidoGuido
```

Examining Namespace

- **The `dir()` function displays names of module attributes**
  - The `__dict__` attribute is a dictionary of a module's objects

```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'math']
>>> __name__
'__main__'
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh' ...
>>> math.__name__
'math'
>>> for key, value in math.__dict__.items():
...     print key, '\t ===>', value
...
pow       ===> <built-in function pow>
fsum      ===> <built-in function fsum>
cosh      ===> <built-in function cosh> ...
```

Examine the current module

Name of the Python program in execution

Examine the `math` module

Name of the `math` module

Testing the __name__ Attribute

- When a module is executed as a program, the __name__ attribute is set to `'__main__'`
- When imported as a module, the __name__ attribute is set to the module name
- May be tested to execute embedded module testing code

```
def main_program():
    ...   # logic to test a module
    ...   # when executed as a program
    ...

if __name__ == '__main__':
    main_program()
```

Chapter Contents

- Module Overview
- `import` and Namespace
- ➢ **`from` and Namespace**
- The Standard Library

`from` Statement

- **Copies named attribute into the current namespace**
  - No attribute qualification needed
- **Syntax: `from` *`module`* `import` *`attributes`***

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

```
>>> from mod1 import printname
starting mod1
>>> printname(2)
GuidoGuido
```

Single attribute

Unqualified name

- **`from module import *`**
  - Imports all attributes into the current namespace
  - Not a best practice
    - May corrupt the namespace

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

```
>>> from mod1 import *
starting mod1
>>> printname(3)
GuidoGuidoGuido
>>> name
Guido
```

All attributes

Unqualified

Namespace Corruption

- **Affects same named objects within the namespace**
  - Later assignments replace previous values
- **`from` *`module`* `import *` is discouraged in Python style guide**
  - PEP 8

```
>>> name = 'Lars'
>>> name
'Lars'
>>> from mod1 import *
starting mod1
>>> name
'Guido'
```

name in current namespace changed

```
"""
A simple module
"""
print 'starting mod1'
name = 'Guido'

def printname(count):
    print name * count
```

Hands-On Exercise 6.1

In your Exercise Manual, please refer to
Hands-On Exercise 6.1: Modules

Packages

- **Directory hierarchy of module files**
- **Allow module hierarchy to follow the directory structure**
  - Instead of flat structure of modules
  - Group related modules as needed
    - System architecture, service, Python version, etc.
- **Packages are usually downloaded as compressed archives**
  - Extraction into a directory structure provides
    - `README` file for instructions
    - `setup.py` file for installation
    - Package `python` files
- **Packages installation**

```
python setup.py install
```

Downloading Packages

- **Python Package Index is a package repository with tutorials**
  - For downloading and installing
  - For creating and uploading
  - `http://pypi.python.org/pypi`
- **pip is a fundamental tool for package management**
  - Installs packages and dependencies from the repository
  - Removes packages and dependencies

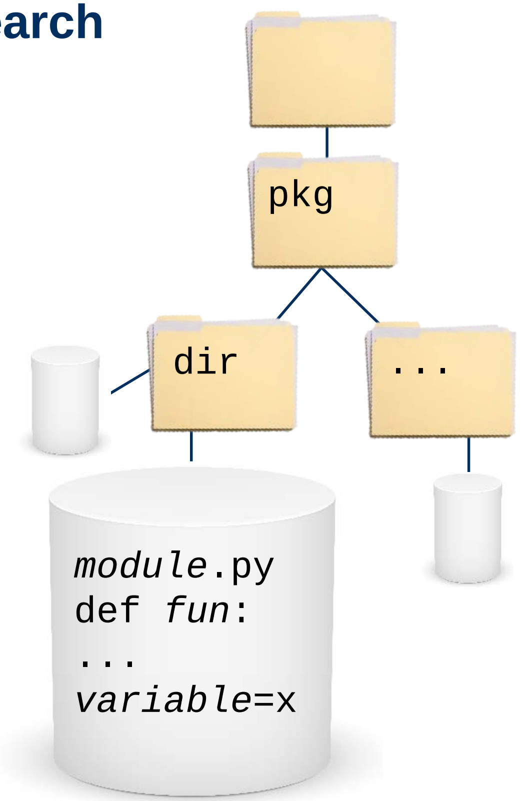
```
pip install packagename
```

Package Directory Structure

- **Base directory of the package must be in the Python search path**
- **Multiple levels of subdirectories are allowed**
- **Each subdirectory must also contain**
  - Its `.py` or `.pyc` module files
  - An `__init__.py` file
    - Executes when that directory's modules are first imported

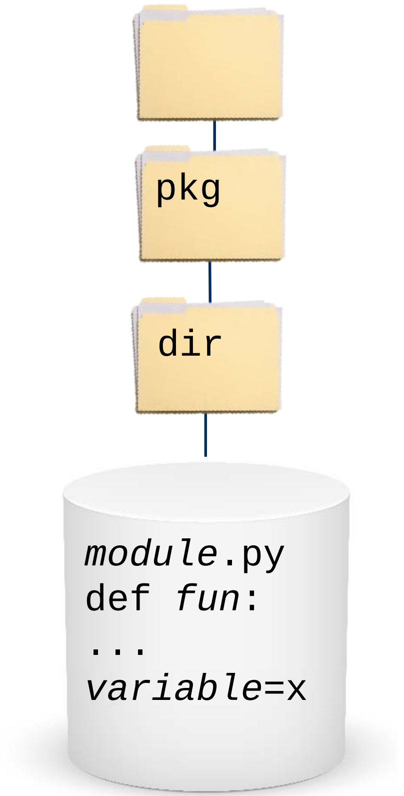
Package `import`

- **Syntax: `import` *`pkg.dir`***
- **Each `.` separates a level of directory structure**
  - No operating-system–specific separators allowed
- ***pkg* must be beneath a root directory within the search path**
- ***`dir`* must have the `__init__.py` file**
  - May be an empty file
- **Runs the code in all `__init__.py` files within the path**
  - Creates an object of the module's contents
- **Each directory becomes a namespace**



Accessing Package Modules

- **`import`** ***`pkg.dir.module`***
  - Names must be fully qualified
    - *`pkg.dir.module.fun()`*
    - *`pkg.dir.module.variable`*
- **`from`** ***`pkg.dir`*** **`import`** ***`module`***
  - Names may be qualified by module
    - *`module.fun()`* or *`module.variable`*
- **`from`** ***`pkg.dir.module`*** **`import`** ***`fun`***
  - Unqualified names are allowed
    - *`fun()`*



- Module Overview
- `import` and Namespace
- `from` and Namespace
- ➢ **The Standard Library**

Standard Library

- **Collection of modules that come with Python**
- **Not part of the language itself**
- **Interfaces to access common utilities**
  - Operating system
    - File system and utilities
  - Database access
  - Date and time information and measurement
  - GUI and network application development
  - Regular expression pattern matching
  - Archiving and compression
  - And more!

The sys Module

- Contains functions and variables that are used by Python itself

sys.version	String of Python version
sys.path	List containing search path for modules
sys.modules	Dictionary of currently loaded modules
sys.platform	String of operating system type
sys.executable	String of pathname to Python interpreter

```
>>> sys.version
'2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 64 bit
(AMD64)]'
>>> sys.path[0]
'C:\\Python27\\Lib\\idlelib',
>>> sys.path[9]
'C:\\Python27\\lib\\site-packages']
```

Command-Line Arguments

- Are passed into the program when it is started

`sys.argv`	List of command-line argument strings passed

argtest.py

```
import sys

print 'arg count is', len(sys.argv)
for word in sys.argv:
    print 'found', word
```

```
C:\> C:\Python27\python argtest.py this is it
arg count is 4
found argtest.py
found this
found is
found it
```

The os and subprocess Modules

Reference

- **Contains functions and variables used to portably query and interact with processes within the operating system**

`os.environ`	Dictionary of environment variables
`os.getpid()`	Function returning an integer process ID
`os.kill()`	Function terminating a process
`subprocess.call()`	Function executing an operating-system command

```
>>> os.environ['PYTHONPATH'].split(';')
['C:\\Python27', 'c:\\Python27\\Lib\\site-packages\\django']
>>> subprocess.call(['ping', 'localhost'])
```

- **The `subprocess.Popen()` class handles process creation**

```
>>> import subprocess
>>> pipe = subprocess.Popen(['ping', 'localhost'], shell=True,
...                          stdout=subprocess.PIPE)
>>> for pingline in pipe.stdout:
...     print pingline
...
Pinging WIN2008_64 [::1] with 32 bytes of data:
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Reply from ::1: time<1ms
Ping statistics for ::1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
```

Output shortened

The os and `os.path` Modules and the File System

- **Contain functions and variables used to**
  - Manage the file system

`os.sep`	String of directory path component separator
`os.getcwd()`	Returns a string of the current directory
`os.chdir()`	Changes the current directory
`os.listdir()`	Returns a list of directory contents
`os.mkdir()`	Creates a directory
`os.rmdir()`	Removes a directory
`os.remove()`	Removes a file

  - Query the file system

`os.path.isdir()`	Returns a Boolean, tests if path is a directory
`os.path.isfile()`	Returns a Boolean, tests if path is a file
`os.path.getsize()`	Returns the size in bytes

The `glob` Module and File Names

- Provides pattern matching on file names

`?`	Matches any single character
`[ ... ]`	Matches any single character in the set
`*`	Matches any number of any character

- `glob()` function returns a list of matching file names

File System Management Example

Reference

```
>>> import os, glob, subprocess
>>> orig = r'C:\Course\1905\Data'
>>> backup = r'C:\Course\1905\backupcsv'
>>> os.mkdir(backup)
>>> os.chdir(orig)
>>> for file in glob.glob('*.csv'):
...     subprocess.call(['copy', file, backup], shell=True)
```

Regular Expression Special Characters

Reference

^, $	Anchor pattern to beginning or ending of line
.	Any single characters
[ ], [^]	Any single character in set or not in set
*	Zero or more of preceding regular expression
+	One or more of preceding regular expression
?	Zero or one of preceding regular expression
|	Or
( )	Group
\	Following character is not special

Regular Expression Escape Sequences

Reference

- Alternate method to describe select text patterns

`\d`	Any single base-10 digit
`\D`	Any single character not a base-10 digit
`\w`	Any single alphanumeric character
`\W`	Any single nonalphanumeric character
`\s`	Any single whitespace character

Using Backslash \

Reference

- **Python uses \ as part of escape sequences in strings**
  - `'\n'` for newline, `'\t'` for tab
- **Escape sequence `\\` represents a single backslash**
- **Regular expression containing backslash must be escaped**
  - `'\\d'` matches a digit
  - `'\\\\'` matches a literal backslash
  - `'\\+'` matches one or more backslashes
- **Raw strings do not honor escape sequences**
  - Specified as `r'`*`string`*`'`
    - `r'\d'` matches a digit
    - `r'\\'` matches a literal backslash
    - `r'\+'` matches one or more backslashes

The `re` Module

Reference

- **Provides pattern-matching functions**
  - Regular expressions are symbolic notation to match text patterns
    - May contain regular characters and special characters
- **`match(`*`pattern`*`,`*`string`*`)`**
  - Finds the *`pattern`* at the beginning of the *`string`* argument
  - Returns a `match` object with `start()` and `end()` methods that return the indices
- **`search(`*`pattern`*`,`*`string`*`)`**
  - Finds the *`pattern`* anywhere in the *`string`* argument
  - Returns a `match` object with `start()` and `end()` methods

String Matching Example

Reference

```
>>> import re
>>> course = 'This is Python Programming'
>>> ans = re.match(r'[A-Z]\w+', course)
>>> ans.start()
0
>>> ans.end()
4
>>> ans = re.search(r'[Pp]\w+', course)
>>> course[ans.start():ans.end()]
Python
```

String slice

Extracting Substrings

Reference

- **`findall(`*`pattern,string`*`)`**
  - Finds all occurrences of the *`pattern`*
  - Returns a list of matching strings

```
>>> data = 'This is the perfect Python Programming string'
>>> re.findall(r'[Pp]\w+',data)
['perfect', 'Python', 'Programming']
```

Using Regular Expressions

Do Now

1. **Access the Python interpreter console using the button**
2. **Import the regular expression module and make the following assignment:**

```
>>> import re
>>> text = 'http://127.0.0.1:8000/cgi-bin/helloworld.py'
```

3. **Use regular expression functions to match and display the following strings:**
   a. One or more consecutive letters
   b. One or more consecutive digits
   c. Only the consecutive digits immediately following a colon, `':'`

Using Regular Expressions: A Solution

Do Now

```
>>> text = 'http://127.0.0.1:8000/cgi-bin/helloworld.py'

>>> re.findall(r'[a-zA-Z]+', text)
['http', 'cgi', 'bin', 'helloworld', 'py']

>>> re.findall(r'[0-9]+', text)
['127', '0', '0', '1', '8000']
>>> re.findall(r'\d+', text)
['127', '0', '0', '1', '8000']

>>> loc = re.search(r':[0-9]',text)
>>> re.findall(r'\d+',text[loc.start():])
['8000']
```

Chapter Summary

You are now able to

- **Create new modules**
- **Access additional modules**
- **Use the standard library**

Chapter 7

Managing Files and Exceptions

Chapter Objectives

After completing this chapter, you will be able to

- **Handle and raise exceptions**
- **Perform I/O with multiple types of files**

I/O = input/output

- **Exceptions**
- Files
- `pickle` and `shelve`

Keyboard Input

- **The `raw_input('`*`prompt`*`')` function returns one line from standard input**
  - Converted into a string with `\n` removed

```
def print_age_in_days(years):
    print 'Your age in days is more than', 365 * int(years)

age = raw_input('Enter your age: ')
print_age_in_days(age)
```

Prompt

Line that caused the exception

```
Enter your age: fifteen
Your age in days is more than
Traceback (most recent call last):
  File "<pyshell#192>", line 2, in <print_age_in_days>
 print 'Your age in days is more than', 365 * int(years)
ValueError: Invalid literal for int() with base 10 'fifteen'
```

Exceptions

- **Are errors generated at runtime**
- **May be raised by Python itself or manually from within a program**
- **Cause a change in the control flow of a program**
  - Default action is immediate termination
    - Includes a stack trace of the calls leading to the exception
- **It is the programmer's responsibility to provide code to handle exceptions**
- **Python's exception-handling capabilities**
  - Simplify coding
  - Increase robustness
  - Provide a uniform approach to handling errors across application code

The `try` Statement

- **General structure of exception-handling code is as follows:**

```
try:
    statements ...

except exceptionTypeA:
    statements ...

except exceptionTypeB:
    statements ...

except:
    statements ...

else:
    statements ...

finally:
    statements ...
```

Block monitored for an exception

Block to handle exceptions of type *exceptionTypeA*

Block to handle exceptions of type *exceptionTypeB*

Block to handle any other unnamed exception

Block to handle if no exception was raised

Block to execute whether an exception was raised or not

Handling a Single Exception

- **Statements within the `try` block are executed and monitored for an exception**
- **On exception, control passes to the appropriate except block**
  - Associated with the most enclosing `try`

```
def print_age_in_days(years):
    print 'Your age in days is more than', 365 * int(years)


try:
    age = raw_input('Enter your age: ')
    print_age_in_days(age)
except ValueError:
    print 'You did not input the age as an integer'
```

1 Call function within `try`

2 Raises `ValueError` exception

3 Branch to `except` for that exception

Handling Multiple Exception Types

- If present, multiple except blocks are checked sequentially
- `except(`*`exceptionA, exceptionB`*`)` defines a single block for multiple exceptions
- `except:` defines a block for any unnamed exception

```
def print_age_in_days(years):
    print 'Your age in days is more than', 365 * int(years)

try:
    age = raw_input('Enter your age: ')
    print_age_in_days(age)
except ValueError:
    print 'You did not input the age as an integer'
except EOFError:
    print 'End of file from standard input'
except:
    print 'Non Value or EOF error occurred'
```

For any unnamed exception type

The `else` and `finally` Clauses

- **`else:` defines a block that is executed if no exceptions are raised**
  - Follows all `except` clauses
    - Must have at least one `except`
- **`finally:` defines a block that is always executed**
  - Whether an exception was raised or not

```
def print_age_in_days(years):
    print 'Your age in days is more than', 365 * int(years)

try:
    age = raw_input('Enter your age: ')
    print_age_in_days(age)
except ValueError:
    print 'You did not input the age as an integer'
else:
    print age, 'was successfully converted to integer'
finally:
    print 'Input test complete'
```

Block always executes

No exception raised

Exception Instances

- **Exception instances are assigned by except *`ExceptionType`* `as` *`name`***
  - *`name`*`.args` references a tuple given to the *`ExceptionType`* constructor

```
def print_age_in_days(years):
    print 'Your age in days is more than', 365 * int(years)

try:
    age = raw_input('Enter your age: ')
    print_age_in_days(age)
except ValueError as ve:
    print 'You did not input the age as an integer'
    print 'Value Error handled', ve.args
else:
    print age, 'was successfully converted to integer'
finally:
    print 'Input test complete'
```

The `raise` Statement

- **Initiates the named exception**
  - Which may be handled or not

```
import string
def print_age_in_days(years):
    for digit in years:
        if digit not in string.digits:
            raise ValueError('Cannot convert', digit, years)
    print 'Your age in days is more than', 365 * int(years)

try:
    age = raw_input('Enter your age: ')
    print_age_in_days(age)
except ValueError as ve:
    print 'You did not input the age as an integer'
    print 'Value Error handled', ve.args
```

raise the exception

```
Enter your age: fifteen
You did not input the age as an integer
Value Error handled, ('Cannot convert', 'f', 'fifteen')
```

Hands-On Exercise 7.1

In your Exercise Manual, please refer to
Hands-On Exercise 7.1: Exceptions

- Exceptions
- **Files**
- `pickle` and `shelve`

Files

- **Built-in object type**
  - Has methods to handle reading, writing, and positioning within the file
- **Reference contents of many types**
  - Character
  - Numeric
  - Class object

The open and `close` Statements

- **Open a file syntax: `object = open('pathname' [, 'mode'])`**

Optional

- **Returns a file**
- **Specifies an opening *mode***
  - `'r'`—Opened for reading at the beginning
    - Default mode
  - `'w'`—Opened for writing at the beginning
  - `'a'`—Opened for writing at the end
  - Additional `'+'` with mode opens for both reading and writing operations
- **Specifies a file's content type within the *mode***
  - Text is the default
  - `'b'` specifies binary
- **Close a file syntax: `object.close()`**
  - Releases open file reference

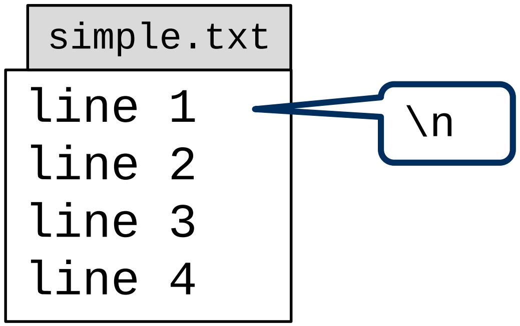
Opening Files and Exceptions

- **An `IOError` exception is raised when opening to**
  - Read a file that does not exist
  - Read or write a file without appropriate access rights
- **`IOError` exception attributes include**
  - `errno`—Error message number, `args[0]`
  - `strerror`—Error message string, `args[1]`
  - `filename`—Filename used when the exception was raised

```
try:
    infile = open('Incorrectfilename')
except IOError as ioe:
    print 'Unable to open the file'
    print 'Error number', ioe.args[0],
    print 'Message', ioe.args[1],
    print 'Filename in error', ioe.filename
```

If `open()` failed,
`close()` is not needed

- **File is a sequence of characters**
  - `'\n'` separates lines



l	i	n	e		1	\n	l	i	n	e		2	\n	...

- **`read()`: Returns the entire file contents as a single string**

- **`readline()`: Returns the next line from the file**
  - Includes the `'\n'` line delimiter
  - `rstrip()` string method can remove the `'\n'`

- **`readlines()`: Returns the entire file contents as a list of strings, including the `'\n'`**

- **`IOError` exception may be raised**

Reading a Text File Example

```
try:
    infile = open('C:/Course/1905/Data/simple.txt', 'r')
    print infile.readline().rstrip()
    print infile.readlines()
    infile.close()
except IOError as ioe:
    print 'Error number', ioe.args[0],
    print 'Message', ioe.args[1]
```

```
line 1
['line 2\n', 'line 3\n', 'line 4\n']
```

Writing a Text File

- **`write(`*`string_ref`*`)`: Writes a single string into a file**
- **`writelines(`*`list_ref`*`)`: Writes a list's contents into a file**
- **On writing, data is cached**
  - `close()` writes the cache and releases the file object
  - `flush()` followed by `os.fsync()` writes the cache and keeps the file open
- **`IOError` exception may be raised**

Data Handling Exceptions

- **Once opened, files should be closed**

```
try:
    infile = open('C:/Course/1905/Data/simple.txt', 'r')
    try:
        print infile.readline().rstrip()
        print infile.readlines()
        infile.write('line 5\n')
    except IOError:
        print 'Read or Write error on file'
    finally:
        infile.close()
except IOError as ioe:
    print 'Failed to open the file'
```

`IOError` exception raised writing to the file

File must be closed whether or not exception occurs

Using `with` to Open and Close Files

- **The `with` statement wraps a block of statements with methods defined by a *context manager***
  - If the file is opened, it will be closed
    - Even if an exception is raised

```
try:
    with open('C:/Course/1905/Data/simple.txt','r') as infile:
        print infile.readline().rstrip()
        print infile.readlines()
        infile.write('line 5\n')

except IOError:
    print 'Read or Write error on file'
```

If `open()` was successful, `close()` is guaranteed

Using Loops and Iterators for File Access

- **The file object is iterable**

```
try:
    with open('C:/Course/1905/Data/simple.txt','r') as infile:
        for dataline in infile:
            print dataline.rstrip()

except IOError:
    print 'Read or Write error on file'
```

```
line 1
line 2
line 3
line 4
```

The Standard Streams

- **Standard streams are file objects available from the sys module**
  - Already opened for reading or writing when the program starts
  - Treated as text files
- **Default to the keyboard and screen when using the Python interpreter**

1. **`sys.stdin`**
   - Provides standard input file for file methods
2. **`sys.stdout`**
   - Provides standard output file for file methods
   - Used by `print`
3. **`sys.stderr`**
   - Provides standard error file for file methods
   - Used for exception messages

Reading and Writing to Standard Streams

```
>>> import sys
>>> inline = sys.stdin.readline()
Honolulu
>>> if inline:
...     sys.stdout.write(inline)
... else:
...     sys.stderr.write('No input found')
Honolulu
```

Keyboard input

Screen output

Redirecting Streams to Files

➢ **Assigns a disk file for use as a standard stream**

- To automate testing user input
- To capture text in a log file

```
import sys
originalerr = sys.stderr
originalout = sys.stdout
errlogfile = 'C:/Course/1905/Data/errorlog.txt'
outputlogfile = 'C:/Course/1905/Data/outputlog.txt'
with open(errlogfile, 'a') as sys.stderr:
    with open(outputlogfile, 'a') as sys.stdout:
        if test_for_some_error:
            sys.stderr.write('Error\n')
        else:
            sys.stdout.write('No Error\n')

sys.stderr = originalerr
sys.stdout = originalout
```

Save the originals

Custom messages appended to the log file

Restore the originals

- Exceptions
- Files
- `pickle` and `shelve`

The `pickle` Module

- **Allows native types to be stored and retrieved from a file**
  - Dictionaries, tuples, classes, etc.
  - Without manual text conversion
- **Performs object serialization**
  - Converts native type to and from a byte sequence for storage
- **Requires an open mode of `'b'`**
  - `.pkl` file name extension is common
- **`pickle.load()`**
  - Reads an object from the `.pkl` file
- **`pickle.dump()`**
  - Writes an object to the `.pkl` file

Reading and Writing With pickle

```
import pickle

airports = {
    'HNL': 'Honolulu',
    'ITO': 'Hilo',
    'GCM': 'Grand Cayman, BWI',
    'CUR': 'Curacao, Netherland Antilles'}

class Airport(object):
    def __init__(self, citycode=None, city=None):
        self.citycode = citycode
        self.city = city

airport_dict = {}
for code in airports:
    airport_dict[code] = Airport(citycode=code,
                                 city=airports[code])
```

Create a dictionary of Airport objects

Reading and Writing With pickle

```
with open('airports.pkl', 'wb') as outfile:
    pickle.dump(airport_dict, outfile)

airport_dict = {}
with open('airports.pkl', 'rb') as infile:
    airport_dict = pickle.load(infile)

print airport_dict['HNL'].city
```

Serializes and stores the dictionary

Re-creates the dictionary from the file

```
Honolulu
```

The `shelve` Module

- **Provides keyed access to a file's contents**
  - Keys are strings
- **Allows normal dictionary operations to**
  - Retrieve the desired values
  - Update a value
  - Add new values
- **Internally uses `pickle` for the translation**
- **Can create and access `.dbm` file**

Reading and Writing With shelve

```
import shelve

airports = {
    'HNL': 'Honolulu',
    'ITO': 'Hilo',
    'GCM': 'Grand Cayman, BWI',
    'CUR': 'Curacao, Netherland Antilles'}

class Airport(object):
    def __init__(self, citycode=None, city=None):
        self.citycode = citycode
        self.city = city

df = shelve.open('airports.dbm', writeback=True)
for code in airports:
    df[code] = Airport(citycode=code,
                       city=airports[code])
```

Read and write access is allowed

Assign to the shelve object

Reading and Writing With shelve

```
print 'From the shelve'
print df['HNL'].city
df['HNL'].city = 'Lulu'
df['NRT'] = Airport(citycode='NRT', city='Tokyo')
df.sync()
for key in df:
    ap = Airport(citycode=key, city=df[key].city)
    print ap.citycode, ap.city
df.close()
```

Writes cache back into shelve file

```
From the shelve
Honolulu
CUR Curacao, Netherland Antilles
HNL Lulu
ITO Hilo
NRT Tokyo
GCM Grand Cayman, BWI
```

Hands-On Exercise 7.2

In your Exercise Manual, please refer to Hands-On Exercise 7.2: Managing Files

Chapter Summary

You are now able to

- **Handle and raise exceptions**
- **Perform I/O with multiple types of files**

Chapter 8

Accessing Relational Databases With Python

Chapter Objectives

After completing this chapter, you will be able to

- **Access relational databases within Python using**
  - `SELECT`
  - `INSERT`
  - `UPDATE`
  - `DELETE`

- **Relational Databases**
- **Using SQL**

Relational Databases

- **Store data as tables**
  - Rows are indexed by keys
    - Unique fields of data
- **Access by the structured query language (SQL)**
  - Standard programming language
- **Are implemented by many proprietary as well as open-source products**
  - Oracle, Sybase, and SQL Server are well-known commercial products
  - PostgreSQL and MySQL are well-known open-source products
  - SQLite comes with Python

MySQL

- **Most popular open-source relational database**
- **Available for many operating-system platforms**
- **Has an API for many programming languages**
- **Accessible through the `MySQLdb` module in Python**
  - `import MySQLdb`

API = application programming interface

Chapter Contents

- Relational Databases
- **Using SQL**

Steps to Accessing the Database

1. Establish a connection
2. Create a cursor for the data interchange
3. Use SQL to access the data
4. Close the connection

Step 1: Establish a Connection

- **`connect()` initiates contact with the database**
  - Requires database name and login information
    - Format is database dependent
  - Returns a connection object
    - Or raises an `OperationalError` exception
- **Connection provides methods for data access management**
  - `close()`—terminates the connection
  - `commit()`—forces write to database store
  - `rollback()`—removes changes back to last `commit()`

```
import MySQLdb

connection = MySQLdb.connect('localhost', 'user1',
'ltree', 'airline')
```

Hostname running the MySQL database

Database user name

Database password

Database name

Step 2: Create a Cursor for the Data Interchange

- **`cursor()` method creates cursor object**
- **Controlling structure for database access**
- **Provides `execute()` method for SQL statements**
  - `ProgrammingError` is raised for invalid statement
- **Provides `rowcount` attribute describing the number of rows changed or fetched**

```
curs = connection.cursor()

curs.execute('SQL statement')
if curs.rowcount:
```

Cursor name

Calls the database for the SQL execution

Test if any rows are affected

Step 3: Use SQL to Access the Data; Step 4: Close the Connection

- **SELECT statement retrieves rows**
- **SQL is passed as an argument to the `execute()` method**
- **Returns the qualified rows into the cursor object**

```
curs.execute('SELECT * FROM aircraft WHERE aircraftcode = 1')
if curs.rowcount:

connection.close()
```

The columns to be retrieved; * indicates all

Tables to be searched

SQL selection criteria

Terminates the connection

Constructing a SELECT String

- **The SQL command must be a single string**
- **May be referenced by a variable**
  - Or variables concatenated

```
query = 'SELECT * FROM flights WHERE flightnum = 1587'

curs.execute(query)
```

Passing Arguments to SQL Statements

- **Prepared statements contain fixed SQL syntax and the %s parameter**
  - Value(s) substituted from the argument list

```
craftnum = (2, )
apt = ('HNL', )

curs.execute('SELECT * FROM aircraft
                           WHERE aircraftcode = %s', craftnum)
curs.execute('SELECT * FROM airport
                           WHERE citycode = %s', apt)
```

Placeholder for argument

Extracting Data From the Cursor

- **Database rows matching the SELECT criteria are available through the cursor**
  - As a single tuple if only one row matched
  - As a tuple of tuples if multiple rows matched

```
curs.execute('SELECT city FROM airport')
for name in curs:
    print 'Airport name', name
```

- **`fetchone()` returns the next tuple**
- **`fetchall()` method returns a tuple of tuples with all remaining rows**
- **`fetchmany(`*`size`*`)` method returns *`size`* tuples within a tuple**
- **Both return None after all rows have been returned**

Inserting a Row

- **Use the SQL `INSERT INTO` *`table`* `VALUES (...)` statement**
  - Values inserted are a tuple
  - Values must meet the database field constraints
- **Call the connection's `commit()` method to update the database storage**

```
newplane1 = (5, 'Blimp')
newplane2 = (6, 'Helicopter')

curs.execute('INSERT INTO aircraft VALUES (%s, %s)',
              newplane1)
curs.execute('INSERT INTO aircraft VALUES (%s, %s)',
              newplane2)

connection.commit()
```

Tuples

Assigned left to right

Updating Data

- **Use the SQL `UPDATE` *`table`* `SET ... WHERE ...` statement**
  - Modifies the fields specified by `SET`
    - For the rows specified by `WHERE`
- **Call the connection's `commit()` method to update the database storage**

```
updateplane1 = (7, 'Blimp')
updateplane2 = ('Bell430', 6)

curs.execute('UPDATE aircraft SET aircraftcode = %s
              WHERE name = %s', updateplane1)

curs.execute('UPDATE aircraft SET name = %s
              WHERE aircraftcode = %s', updateplane2)

connection.commit()
```

Deleting Data

- Use the SQL **`DELETE FROM`** ***`table`*** **`WHERE`** statement
- Call the connection's **`commit()`** method to update the database storage

```
planecode = (6, )
planetype = ('Blimp', )

curs.execute('DELETE FROM aircraft
                    WHERE aircraftcode = %s', planecode)

curs.execute('DELETE FROM aircraft
                    WHERE name = %s', planetype)

connection.commit()
```

Hands-On Exercise 8.1

In your Exercise Manual, please refer to Hands-On Exercise 8.1: Accessing a MySQL Database

Chapter Summary

You are now able to

- **Access relational databases within Python using**
  - `SELECT`
  - `INSERT`
  - `UPDATE`
  - `DELETE`

Chapter 9

Developing GUIs With Tkinter

Chapter Objectives

After completing this chapter, you will be able to

- **Build interactive GUIs using Tkinter**
- **Create and display basic widgets**
- **Add callback functions**
- **Create widget classes within frames**

- **Tkinter**
  - **Basic Widgets and Display**
  - **Callbacks**
  - **`Entry` and `Radiobutton`**
  - **Menus**

Tkinter

- **Standard library supplied with Python for GUI creation and management**
  - Not part of the language itself
  - Is one of several GUI development frameworks for Python
    - `WxPython` and `QtPy` are others
- **Is based on the Tk library**
  - An open-source toolkit for building portable GUIs
  - Available for other programming languages
  - Originally created with the Tcl programming language
- **Provided for Python by the `Tkinter` module**
  - Allows Python to
    - Control component creation and presentation
    - Handle user interaction events

Python 3 module name is `tkinter`

Tkinter Portability

- **The Tk library has been ported to many operating systems**
  - Apple OS X
  - Microsoft Windows
  - UNIX or Linux using X Windows
- **Python programs using Tkinter should work on all platforms**
  - Require no changes
  - Maintain the platform-specific *look and feel*

- Tkinter
- ➢ **Basic Widgets and Display**
- Callbacks
- `Entry` and `Radiobutton`
- Menus

Displaying Widgets

Do Now

A. **Start Eclipse if needed**

B. **Open the `Tk_examples` project and its `label_buttons.py` file**

C. **Run the program**

Bring in the module classes and functions

```
from Tkinter import Tk, Label, Button, mainloop

def setup_gui(base, prompt='Default Instructions'):
    # Steps 2 and 3.  Create and pack widgets in a root window
    infolabel = Label(base, text=prompt, bg='white', fg='blue')
    infolabel.pack(side='left', expand=1, fill='both')

    execbutton = Button(base, text='Execute', fg='black')
    execbutton.pack(side='left')

    exitbutton = Button(base, text='Exit', fg='red')
    exitbutton.pack(side='right')
```

2

3

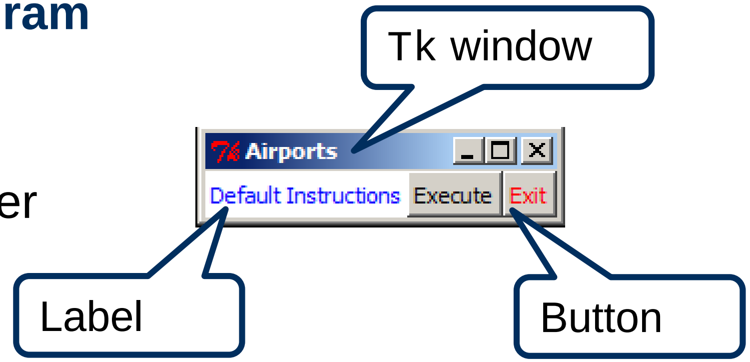
Create widget

Display widget

Displaying Widgets

Do Now

- **Notice the four main parts of GUI program**
  1. Create root window
  2. Create widgets within a window
  3. Display widgets with geometry manager
  4. Start `mainloop()`



1

4

```
# Step 1.  Create the root Tk window
root = Tk()
root.title('Airports')
setup_gui(root)

# Step 4.  Start display loop
mainloop()
```

Start display loop

Widgets

- **Standard building blocks of a GUI**
  - Labels, buttons, frames, and others
  - Provided by the Tk library as classes
- **Have attributes that describe their appearance**
  - Colors, fonts, borders, etc.
- **Created within a widget hierarchy, or tree**
  - Window manager or `root` window is the default parent
- **Assembled to present the display**
  - Geometry manager controls size and position within the layout

Tk Class Widget

- **Provide the parent objects of a widget tree**
  - Create empty windows where other widgets may be attached
  - Provide attributes that apply to the window itself
    - `title()` method sets the window title

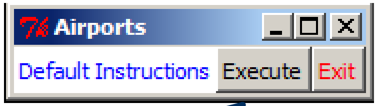
- **Has no parent**
- **Is displayed by the `mainloop()` function**

```
root = Tk()
root.title('Airports')
setup_gui(root)

mainloop()
```

Start display loop

Label and Button Widgets

- **Display text strings or images**
  - `text` attribute references a string
  - `fg` and `bg` colors



Label and Buttons

- **Are part of a widget hierarchy**
  - First argument specifies the parent widget

```
def setup_gui(base, prompt='Default Instructions'):

    infolabel = Label(base, text=prompt, bg='white', fg='blue')

    execbutton = Button(base, text='Execute', fg='black')

    exitbutton = Button(base, text='Exit', fg='red')
```

Parent widget is Tk

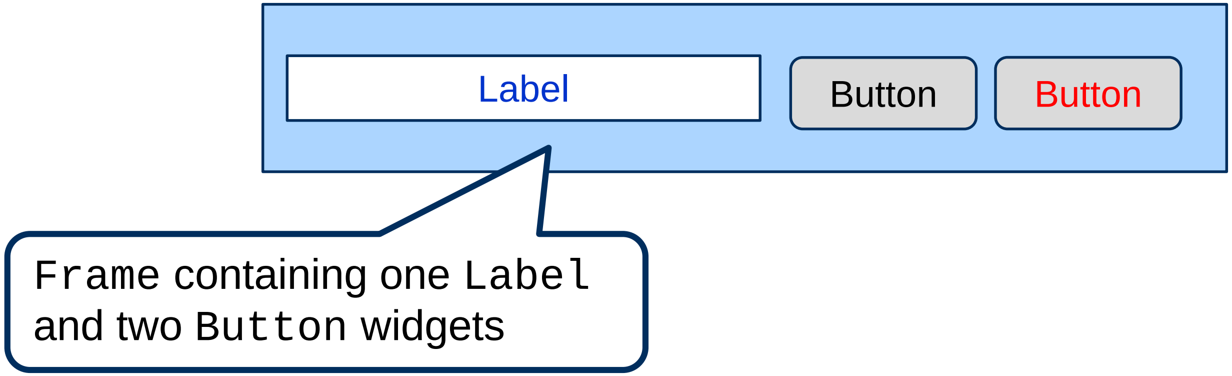
pack( ) Geometry Manager

- **Function that causes widgets to display**
- **Controls the relative layout of the widgets**
  - Attributes define the location, orientation, and expansion
  - Initial size is calculated based on the contained widgets' content
- **By default, resizing with the mouse changes only the root window size**
  - Child widget resizing is controlled by the widget's `pack()` parameters
    - `expand` defines whether the widget grows within expanded space
      - `'yes'` or `1` are equivalent
      - `'no'` or `0` are equivalent
    - `fill` describes widget horizontal and vertical growth within expanded space
      - `'x'`, `'y'`, or `'both'`

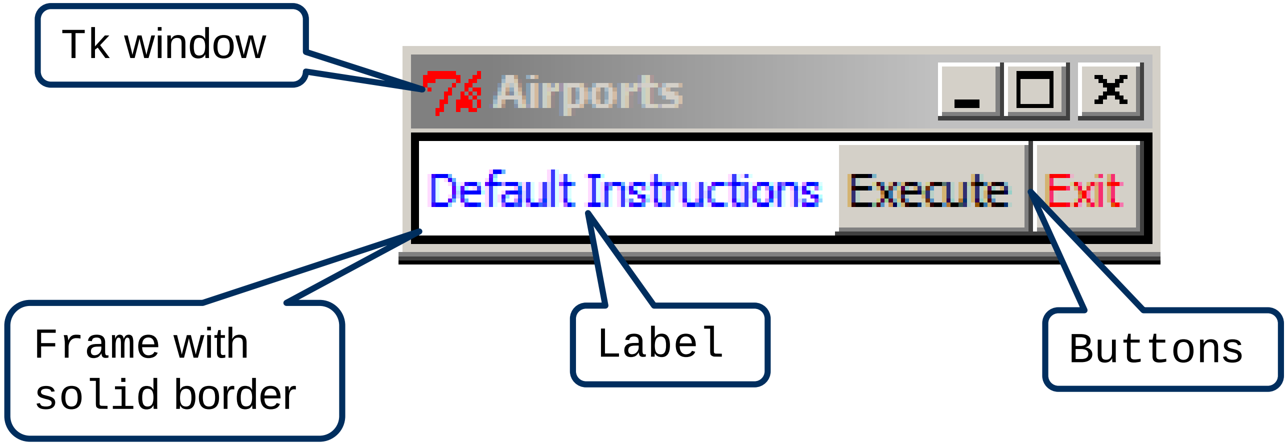
```
infolabel.pack(side='left', expand=1, fill='both')

execbutton.pack(side='left')
```

Frame Widgets

- **Provide windows where other widgets are displayed**
  - Single Tk window may contain many frames
- **Are used to create custom classes**
  - Instances inherit common layout



Frame Widgets



labels_buttons_frame.py

```
class Baseframe(Frame):
    def __init__(self, base, prompt='Default Instructions'):
        self.root = base
        self.prompt = prompt
        Frame.__init__(self, relief='solid', border=2)
        self.pack(expand=1, fill='both')
        self.setup_gui()
```

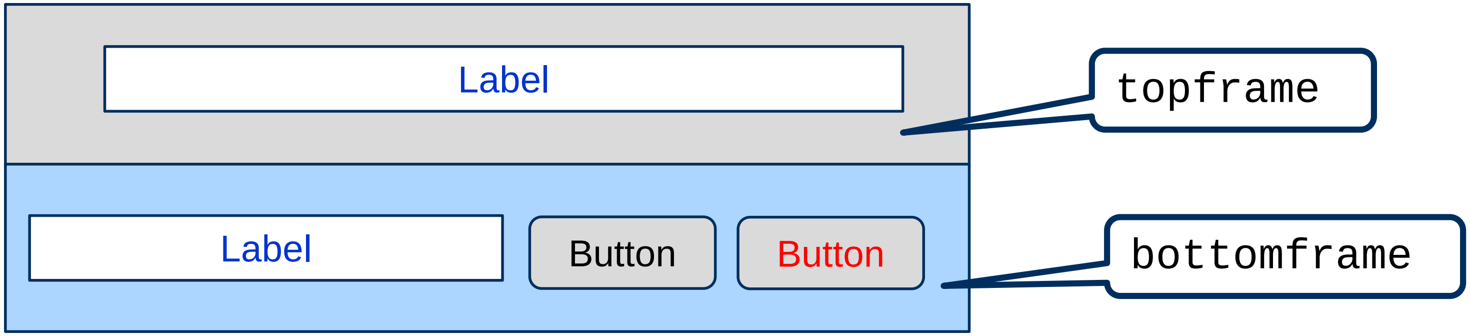
```
    def setup_gui(self):
        self.promptlabel = Label(self, text=self.prompt,
                                 bg='white', fg='blue')
        self.promptlabel.pack(side='left', expand=1,
                              fill='both')
        self.execbutton = Button(self, text='Execute',
                                 fg='black')
        self.execbutton.pack(side='left')
        self.exitbutton = Button(self, text='Exit',
                                 fg='red')
        self.exitbutton.pack(side='right')

root = Tk()
root.title('Airports')
Baseframe(base=root)
mainloop()
```

Create instance

Frames Within Frames

➢ **Frame widgets may contain additional frames**

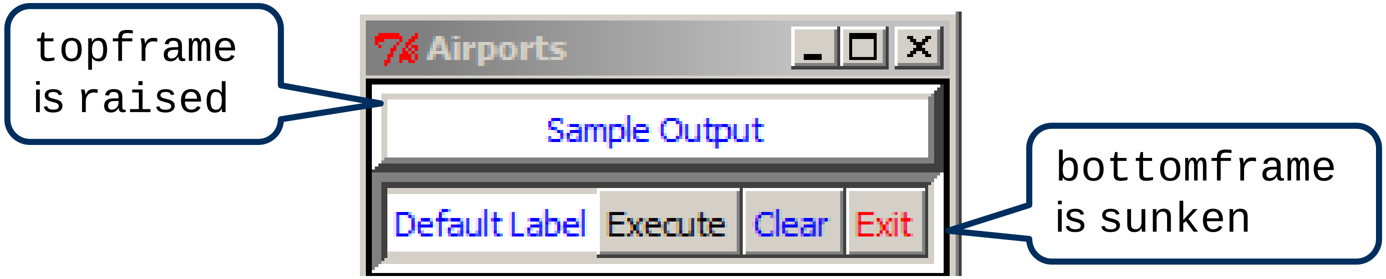
- Control widget layouts



label_buttons_frame_grid.py

```
class BaseFrame(Frame):
    def __init__(self, base, prompt='Default Label'):
        self.root = base
        self.prompt = prompt
        Frame.__init__(self, relief='solid', border=2)
        self.pack(expand=1, fill='both')
```

Frames Within Frames




```
        self.topframe = Frame(self, relief='raised', border=5)
        self.topframe.pack(side='top', expand=1, fill='both')

        self.bottomframe = Frame(self, relief='sunken',
                                 border=5)
        self.bottomframe.pack(side='bottom', expand=0)
        self.setup_gui()

    def setup_gui(self):
        self.output = Label(self.topframe,
                            text='Sample Output',
                            bg='white', fg='blue')
        self.output.pack(side='top', expand=1, fill='both')
```

The `grid` Method

➢ **Provides horizontal and vertical geometry management**
- `column` and `row` attributes
  - `column=0`, `row=0` is upper left
- `rowspan` and `columnspan` specifies height and width

```
self.promptlabel = Label(self.bottomframe,
                   text=self.prompt,
                   bg='white', fg='blue')
self.promptlabel.grid(row=0, column=0,
                      columnspan=3)
self.execbutton = Button(self.bottomframe,
                  text='Execute', fg='black')
self.execbutton.grid(row=0, column=3)
self.clearbutton = Button(self.bottomframe,
                          text='Clear', fg='blue')
self.clearbutton.grid(row=0, column=4)
self.exitbutton = Button(self.bottomframe, text='Exit',
                         fg='red')
self.exitbutton.grid(row=0, column=5)
```

Location within frame

Width

- Tkinter
- Basic Widgets and Display
- ➢ **Callbacks**
- `Entry` and `Radiobutton`
- Menus

A Callback Function

- **The reference to the callback function is passed when the button is created**
  - The `command` attribute
  - The function is not executed until the button is clicked
  - Control returns to `mainloop()`

`label_buttons_frame_grid_callback.py`

```
self.execbutton = Button(self.bottomframe,
                  text='Execute', fg='black',
                  command=self.showinfo)          Method

self.clearbutton = Button(self.bottomframe,
                   text='Clear', fg='blue',
                   command=self.clearinfo)          Method

self.exitbutton = Button(self.bottomframe, text='Exit',
                  fg='red', command=self.quit)
```

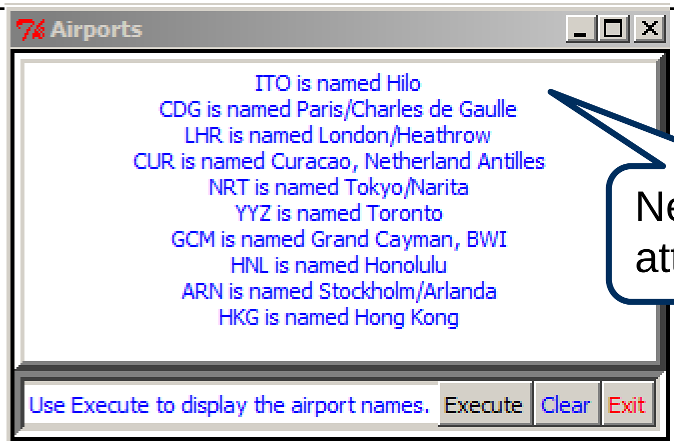
Terminates frame

Reconfiguring a Widget

- **The `configure()` method sets instance attributes in running widgets**

```
def showinfo(self):
    outs = []
    for key, value in city_code_dict.items():
        outs.append('{0} is named {1}'.format(key, value))
    outs = '\n'.join(outs)
    self.output.configure(text=outs)

def clearinfo(self):
    self.output.configure(text='')
```

Set `text` attribute in output `Label`



New `text` attribute value

ScrolledText Module

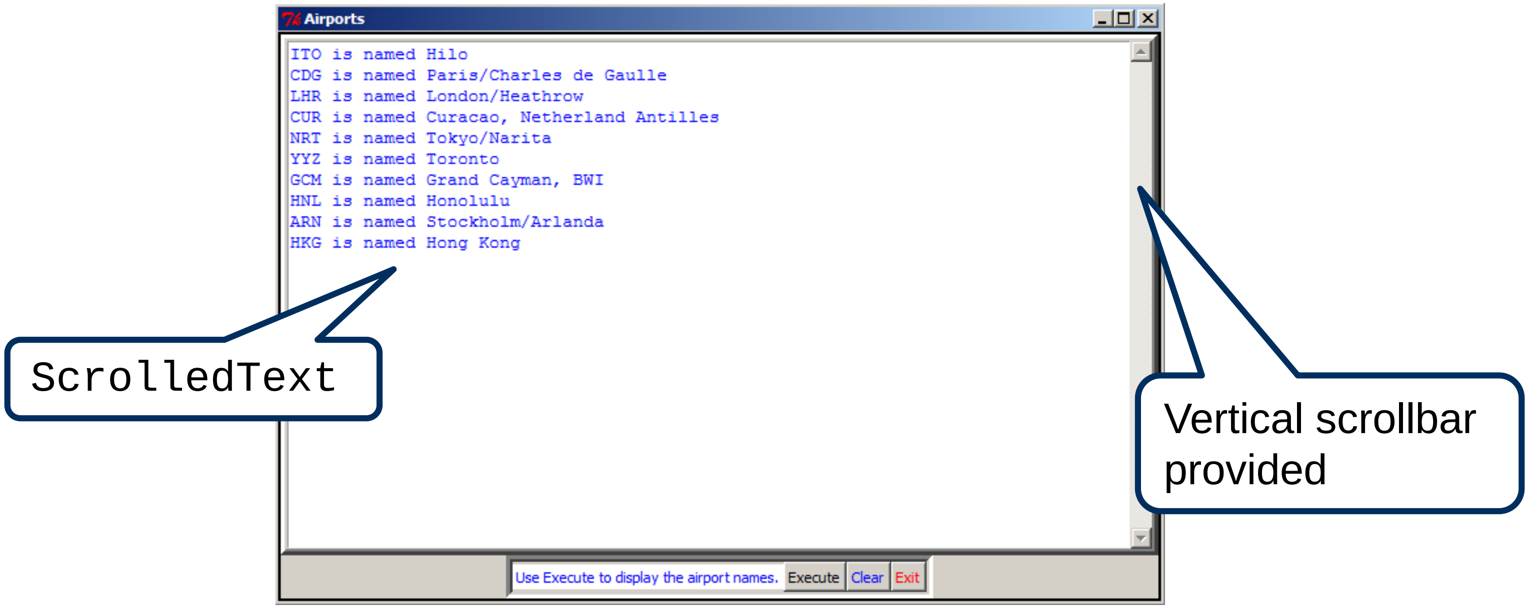
- **Provides a `ScrolledText` class**
  - Implements a `Frame` containing a `Text` widget and vertical scrollbar
  - Easier than creating them directly
- **Provides methods**
  - `insert()` adds text within the widget
  - `delete()` removes text from the widget

label_buttons_frame_grid_callback_scrolltext.py

```
from ScrolledText import ScrolledText
from Tkinter import ...
...

def setup_gui(self):
    self.output = ScrolledText(self.topframe,
                               bg='white', fg='blue')
    self.output.pack(side='top', expand=1, fill='both')

...
```

ScrolledText Module

```
def showinfo(self):
    for key, value in city_code_dict.items():
        self.output.insert('end',
        '{0} is named {1}\n'.format(key, value))

def clearinfo(self):
    self.output.delete(1.0, 'end')
```



- Tkinter
- Basic Widgets and Display
- Callbacks
- **Entry and Radiobutton**
- Menus

`Entry` Widgets

- **Present a field for keyboard input**
- **Variable of class `StringVar` references input text**
  - Assign to `Entry` widget's `textvariable` attribute
- **Provide methods to control the input area**
  - `get()` returns the entered data
  - `set()` assigns to the input area
- **`bind()` method maps keystrokes to a function**
  - `<Return>` for Enter key
  - `func` attribute references the function

Entry Example

Reference input string

`label_buttons_frame_grid_scrolltext_entry.py`

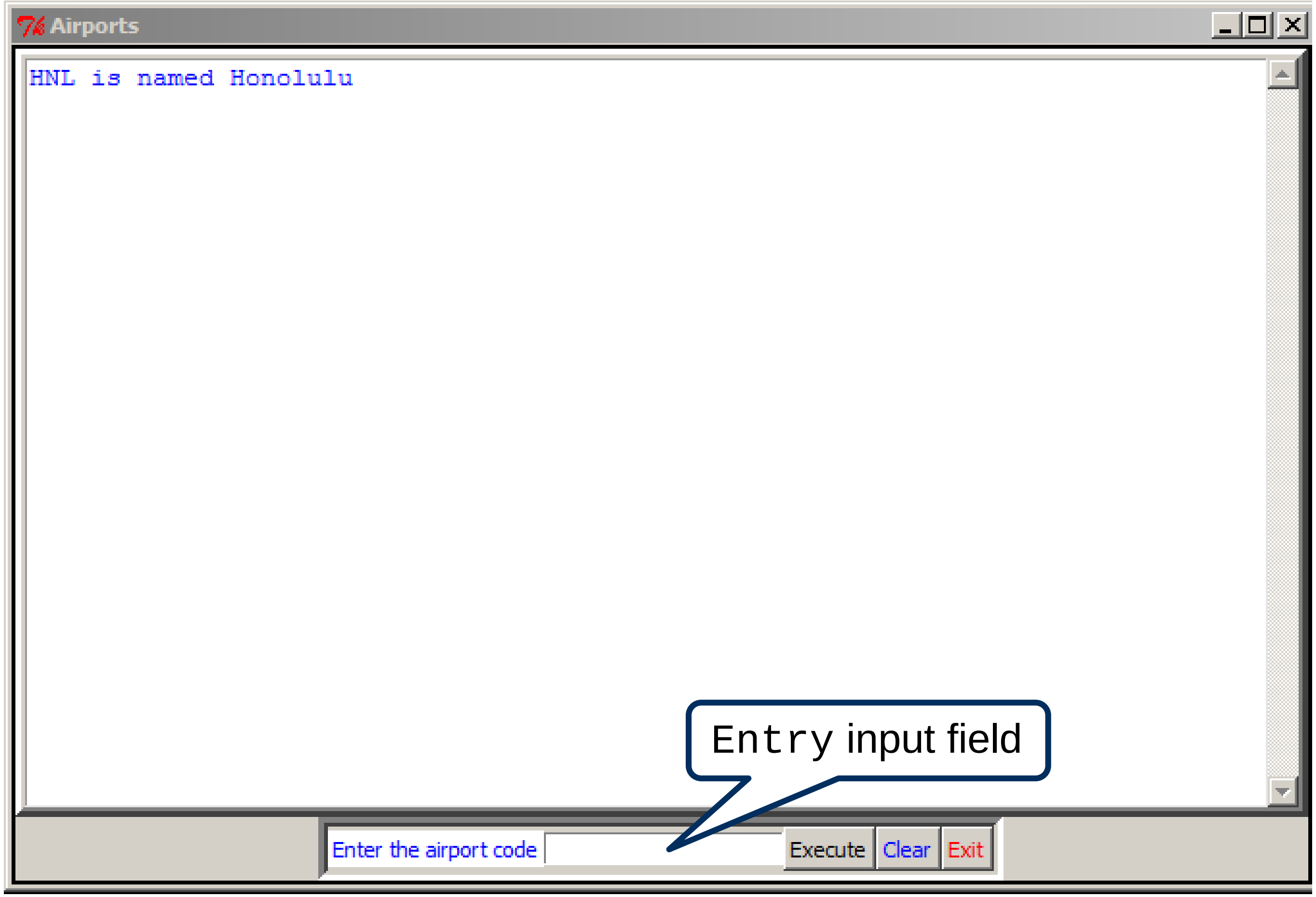
```
self.apt = StringVar()
self.input = Entry(self.bottomframe, textvariable=self.apt)
self.input.bind(sequence='<Return>', func=self.showinfo)

def showinfo(self, *args):
    airport = self.apt.get().upper()
    if airport in city_code_dict:
        msg = '{0} is named {1}\n'.format(airport,
              city_code_dict[airport])
    else:
        msg = '{0} is not an airport we
              serve\n'.format(self.apt.get())
    self.output.insert('end', msg)
    self.apt.set('')
```

Retrieve input string

Line wraps around

Reset input field to empty string

Entry Example



`Radiobutton` Widgets

- **A group of buttons that work together**
  - Only a single button can be selected at a time
- **Have `variable` and `value` attributes**
  - The same `variable` is used for all buttons in the group
  - The `value` setting defines the `variable`'s state for a particular selection

label_buttons_frame_grid_scrolltext_radio.py

```
self.apt = StringVar()
self.apt.set(' ')
col_num = 1
row_num = 0
for key in city_code_dict:
    rb = Radiobutton(self.bottomframe, text=key,
                     variable=self.apt, value=key)
    rb.grid(row=row_num, column=col_num)
    col_num += 1

...
def showinfo(self):
    airport = self.apt.get()
    ...
```

No button is preselected

Variable shared by all Radiobuttons

Assigns 'NRT' to self.apt

Chapter Contents

- Tkinter
- Basic Widgets and Display
- Callbacks
- `Entry` and `Radiobutton`
- **Menus**

Menu Widgets

- **Have characteristics similar to `Button` widgets**
  - A `label` attribute that is visible
  - A `command` attribute for a callback function
- **Are attached to a parent widget**
  - `menu` attribute of parent
- **May have submenus attached**
  - `add_cascade()` method of the parent menu

Menu Creation Steps

1. **Create top-level menu as a child of the root window**
   - Assign to the root window's `menu` attribute
   - Attachment point
2. **Create a second-level menu as a child of the top-level menu**
3. **Call the top-level menu's `add_cascade()` method to attach the second-level menu**
4. **Create selections with `add_command()` within the second-level menu**
   - Contains the `label` and `command` parameters

`label_buttons_frame_grid_scrolltext_menu.py`

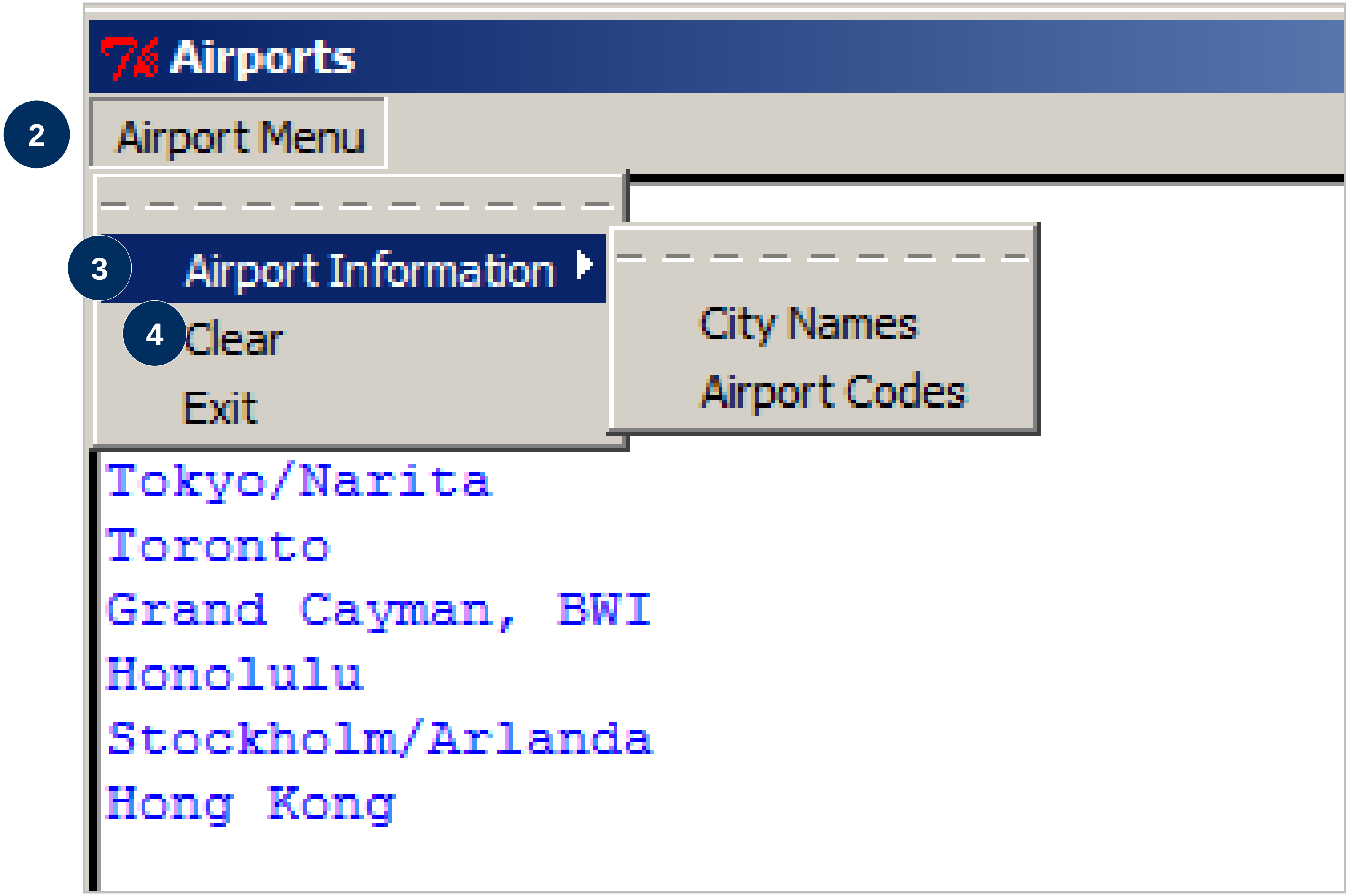
```
class MenuFrame(Frame):
    def __init__(self, base, info='Default Label'):
        self.base = base
        self.info = info
        Frame.__init__(self, self.base, relief='solid',
                       border=2)
        self.pack(expand=1, fill='both')
        self.setup_gui()
```

Starting a Menu

```
def setup_gui(self):
    self.menubar = Menu(self.base)
    self.base.configure(menu=self.menubar)

    self.airportmenu = Menu(self.menubar)
    self.menubar.add_cascade(label='Airport Menu',
                             menu=self.airportmenu)
    self.airportinfo = Menu(self.airportmenu)
    self.airportmenu.add_cascade(label='Airport Information',
                                 menu=self.airportinfo)

    self.airportmenu.add_command(label='Clear',
                                 command=self.clearinfo)
    self.airportmenu.add_command(label='Exit',
                                 command=self.base.quit)
    self.airportinfo.add_command(label='City Names',
                                 command=self.show_values)
    self.airportinfo.add_command(label='Airport Codes',
                                 command=self.show_keys)
```

1
2
3
4

Menu Example



Hands-On Exercise 9.1

In your Exercise Manual, please refer to Hands-On Exercise 9.1: GUI With Tkinter

Chapter Summary

You are now able to

- **Build interactive GUIs using Tkinter**
- **Create and display basic widgets**
- **Add callback functions**
- **Create widget classes within frames**

Chapter 10

Web Application Development With Python

Chapter Objectives

After completing this chapter, you will be able to

- **Describe web application development with Python**
- **Build a Python web application using the Django framework**

- **Web Application Development**
  - Python for Web Application Development
  - Working With Django

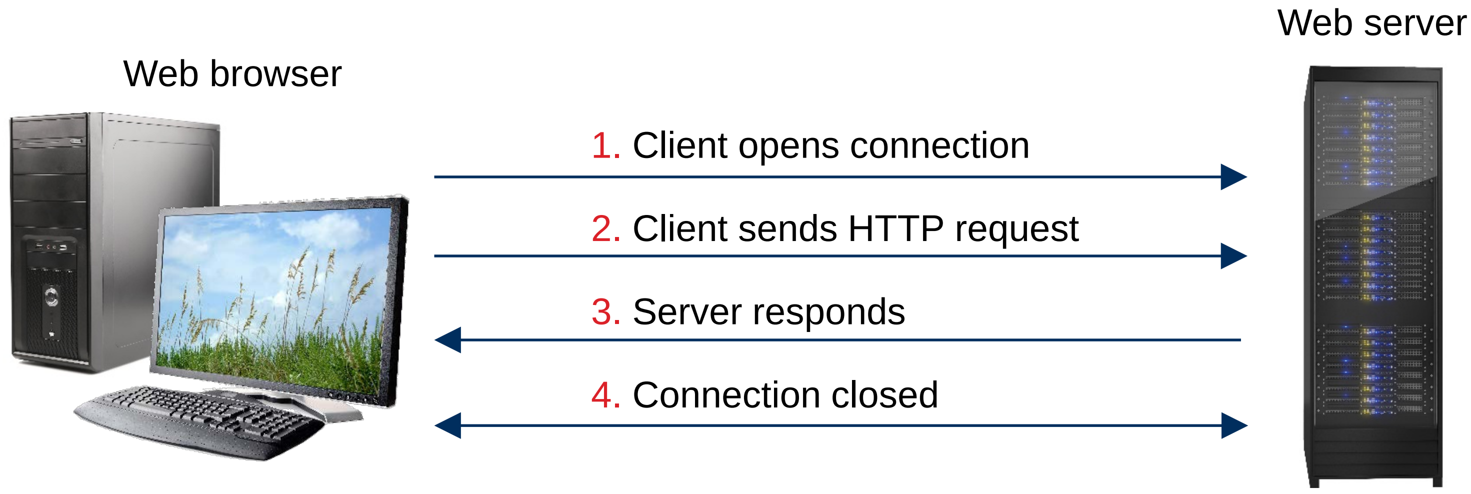
What Is a Web Application?

- **An application or system of applications that uses HTTP as its primary transport protocol**
- **The web is an excellent platform for application development**
  - Web browsers as the universal client
  - Web servers for HTML pages (static and dynamic)
  - Universal network access using the Internet/intranet

HTTP = Hypertext Transfer Protocol

Hypertext Transfer Protocol

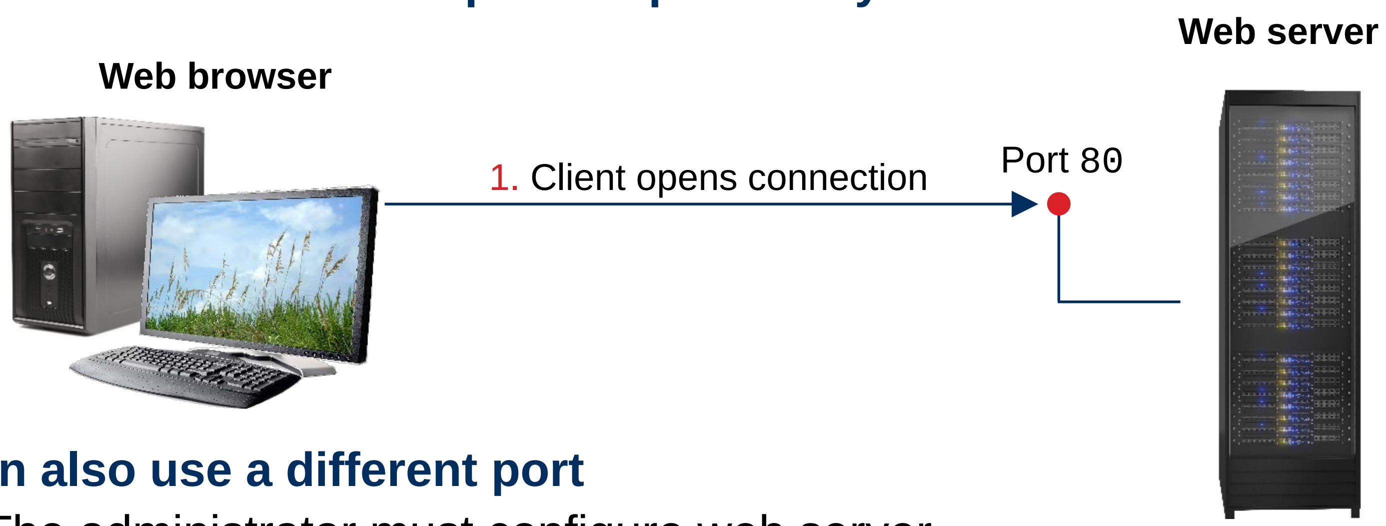
- **HTTP is the protocol for communicating on the web**
  - Stateless, TCP/IP-based protocol
  - HTTP 1.1 defined in RFC 2616, `www.faqs.org/rfcs`
- **HTTP conversation initiated when user enters URL in web browser**
  - For example, `www.learningtree.com/whats_hot.html`



RFC = Request for Comments
TCP/IP = Transmission Control Protocol/Internet Protocol
URL = uniform resource locator

Browser and Server Interaction Step 1: Client Opens Connection

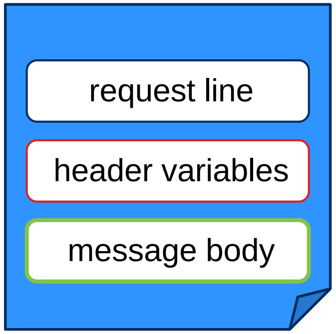
- **Client opens connection to server: `www.learningtree.com`**
  - Opens TCP/IP socket connection on a port
- **Web browsers send request to port `80` by default**



- **Can also use a different port**
  - The administrator must configure web server
  - Clients must use `http://`*`host_name`*`:`***`port`***
    - Example: `http://localhost:`**`8000`**

- **Web browser issues HTTP request**
- **An HTTP request message is composed of**
  - *Request line*: describes HTTP command
  - *Header variables*: browser information
  - *Message body*: contents of message
- ***Request line* is composed of method, resource name, protocol**

HTTP request message



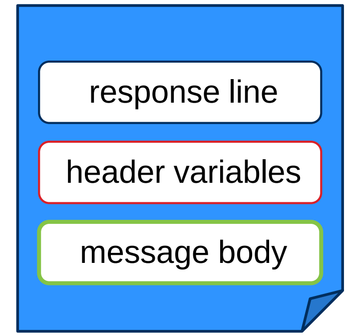

```
GET     /whats_hot.html     HTTP/1.1
```

HTTP method   Resource name   Protocol/version

Browser and Server Interaction Step 3: Server Responds

- **Server sends an HTTP response message**
  - *Response line*: server protocol and status code
  - *Header variables*: response metadata
  - *Message body*: contents of message
- **For a successful request, the server responds with the following:**

HTTP response message




```
HTTP/1.1 200 OK
Server: Apache/2.2 (Unix)
Last-Modified: Sun, 11 march 2012 08:39:21 GMT
Content-Length: 2608
Content-Type: text/html
… … …
<HTML>
<HEAD><TITLE>What's Hot at Learning Tree?</TITLE></HEAD>
<BODY>
  <H1> Hot Course covers … </H1>
  … … …
</BODY>
</HTML>
```

Response line

Header variables

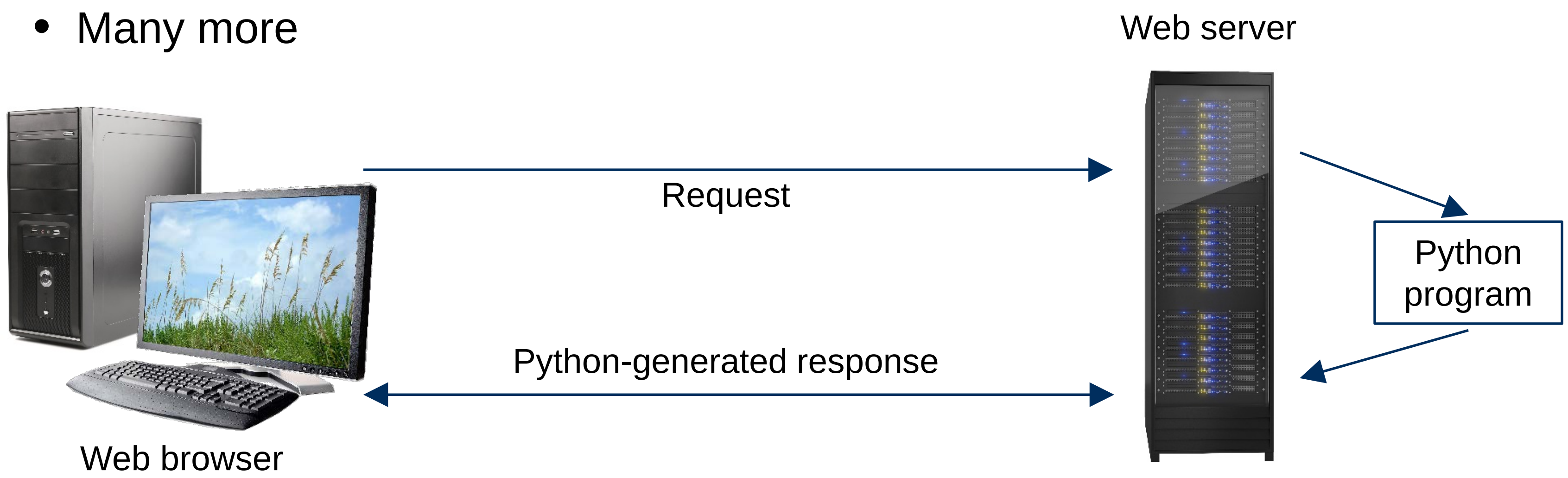
Message body

- Web Application Development
- ➢ **Python for Web Application Development**
- Working With Django

Python for Web Application Programming

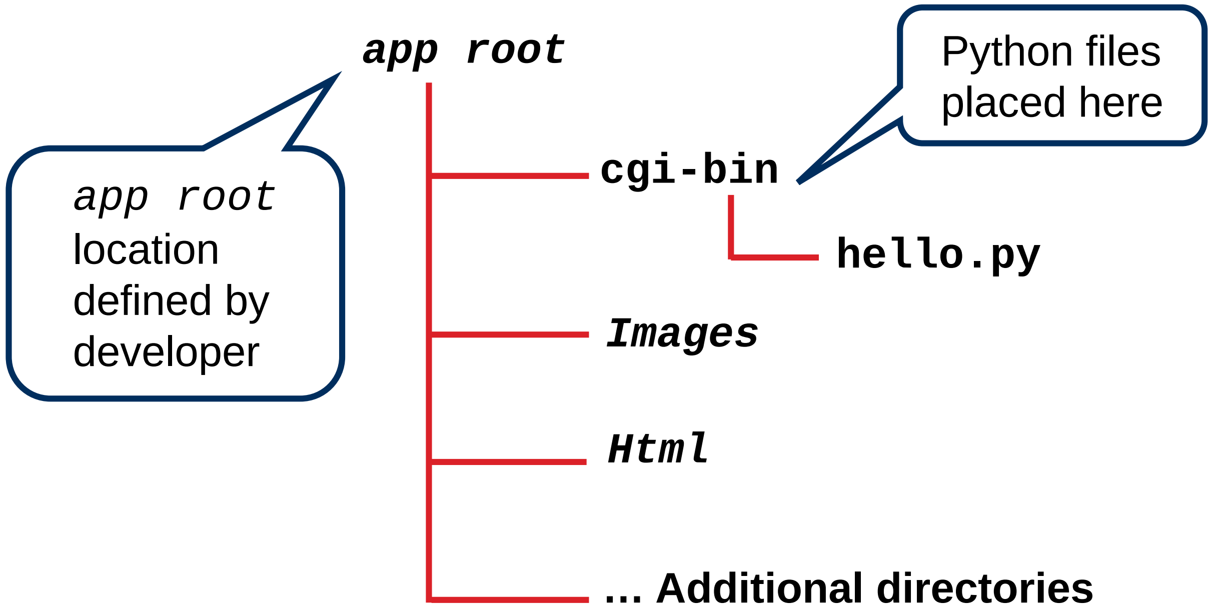
- **Web applications generate dynamic responses to user requests**
  - Use programs known as server-side scripts
  - Can be written in a variety of programming languages
    - Java, C#, Ruby, Perl, and Python

- **Python programs can be used with all major web servers**
  - Apache
  - Internet Information Services (IIS)
  - Many more



Python Web Application Structure

- **Application has a root directory**
  - All files should be below root
  - Both static files and Python programs



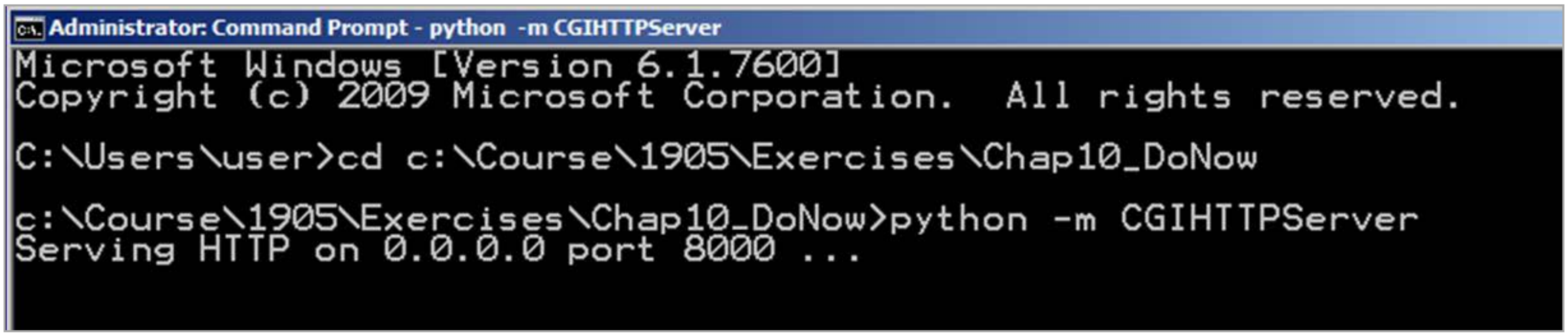
- **Python program will generate HTML page**
  - Has to set content type to `text/html`
  - Program can be a mixture of text and Python code
  - Text will be sent back to client
- **Place Python program files in `cgi-bin` folder of web application**

Set `Content-Type` for client

hello.py

```
print "Content-Type: text/html"

print """
<html>

   <body>
      <h1>Hello</h1>
   </body>


</html>

"""
```

1. **Open the Command Prompt**
2. **Navigate to the `Chap10_DoNow` folder**
   - `cd C:\Course\1905\Exercises\Chap10_DoNow`
3. **Start the Python web server on port `8000`**
   - `python –m CGIHTTPServer`



4. **Open a browser and request your `hello.py` program**
   - `http://localhost:8000/cgi-bin/hello.py`

You will develop your first Python web application; it will display the date and time

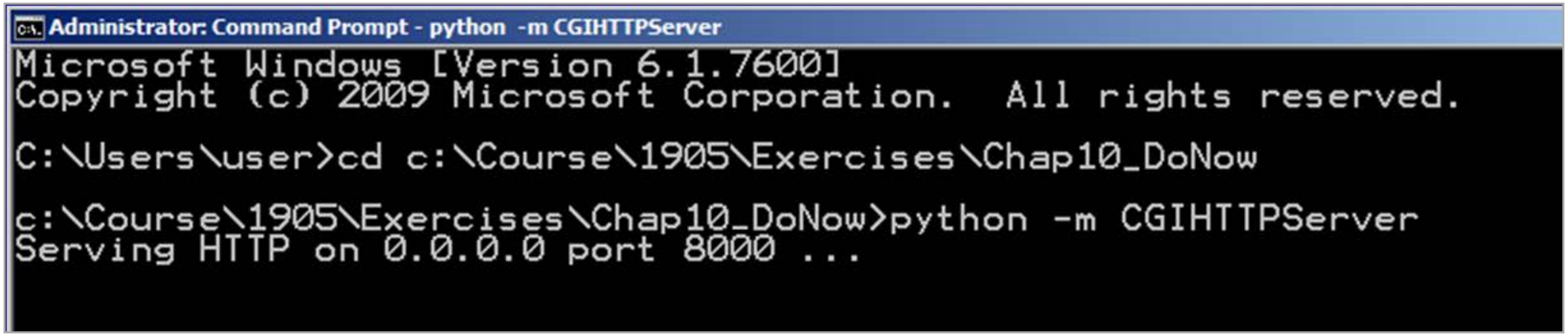
5. **In Eclipse, open the project `Chap10_DoNow`**
6. **Open the file `showdate.py`**
   - This is located in the `cgi-bin` folder
7. **Add the code that will display today's date in the browser**
   - You will need to import `date` from the module `datetime`
   - Use the `today()` method to obtain the date

```
from datetime import date

date.today()
```

Do Now

8. **Save your file**

9. **From the command prompt:**
   - Verify the web server is still running from the proper directory
   - Restart if necessary

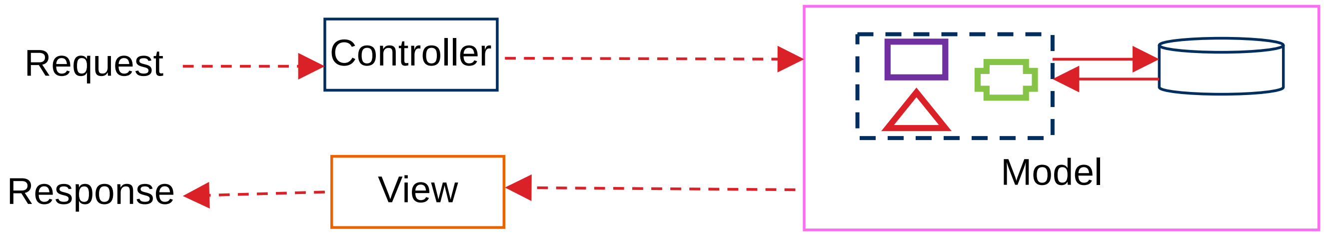


10. **Open a browser and request your `showdate.py` program**
    - `http://localhost:8000/cgi-bin/showdate.py`

11. **Close the Command Prompt when done**

Shortcomings of This Approach

- **Mixing code and HTML in the same file is not recommended**
  - Application becomes difficult to maintain
  - Difficult to reuse Python code and HTML
- **Solution is to separate different areas of functionality**
- **Model View Controller (MVC) design pattern**
  - Provides clean separation between control and presentation
    1. The *controller* handles the initial request
    2. The controller converts the request into commands for the *model*
    3. The model forwards data to *view*
    4. The view generates response using data forwarded by the model

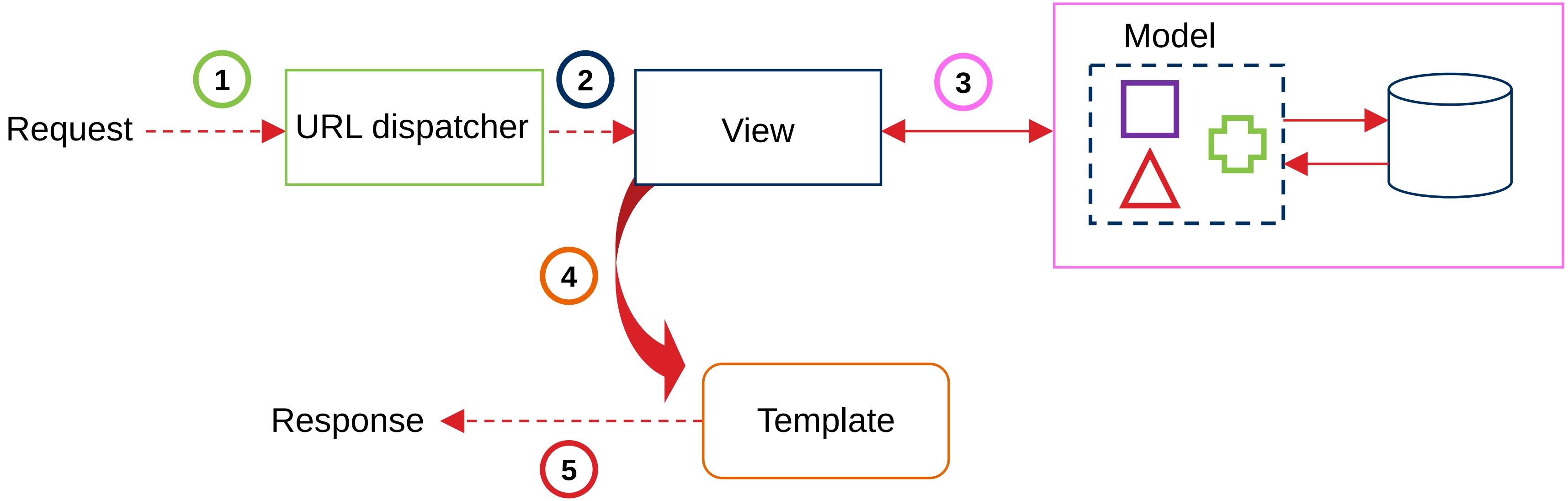


- Web Application Development
- Python for Web Application Development
- ➢ **Working With Django**

- **Open-source project**
  - Originally developed at Lawrence (Kan.) Journal-World Online
  - Django website: `https://www.djangoproject.com`
- **Python web application development framework**
  - Implements a variation of MVC
  - Enables developer to focus on building application functionality
    - Not infrastructure code
  - Designed to enable applications to be built simply and quickly
- **Django's variation on MVC uses *views* and *templates***
  - Provide clean separation between control and presentation
    - Views are Python code files
      - Process requests and provide data for templates
    - Templates generate HTML response using view-provided data

Django Design Pattern

- **HTTP requests are processed as follows:**
  1. URL dispatcher module identifies appropriate view to process request
  2. *View* decides what is being requested and delegates work to model
  3. *Model* undertakes work and returns result to view
  4. *View* selects template to be used for generating response and passes data
  5. *Template* generates HTML response using view-provided data

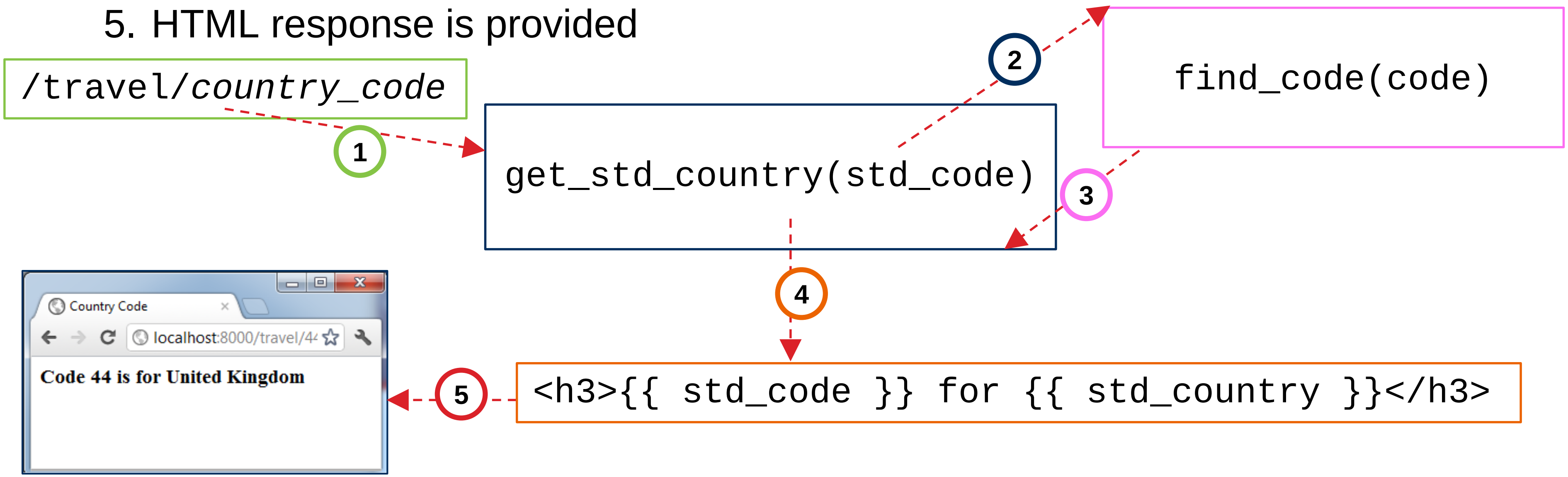


- **Application development proceeds as follows:**
  A. Sketch application flow
     – Data to be submitted with request
     – Data required by template to generate the response
  B. Implement view method
     – Processes request using model
     – Pass data (if required) to template
  C. Implement model method
     – Handle data retrieval
  D. Implement template
     – Render HTML response
  E. Map URL to view method

Step A: Sketch Application Flow

- **Our workflow will enable an STD code to be submitted**
  - Response page will display associated country name

- **Example demonstrates passing data in request**
  1. URL dispatcher calls appropriate view
  2. View delegates data processing to model
  3. Model returns results to view
  4. View passes data to template
  5. HTML response is provided



STD = Subscriber Trunk Dialing

Step B: Define View Method

- **View methods defined in `views.py`**
  - Mapped from URL by URL dispatcher module
    - Mapping defined by developer (Step E)
- **`render_to_response(`*`templateName, viewData`*`)` generates response**

views.py

```
import country_code_lookup

def get_std_country(request, std_code):
    # call model
    results = country_code_lookup.find_code(std_code)

    data_for_view = {'std_country': results,
                     'std_code': std_code}

    return  render_to_response('std_country_code.html', data_for_view)
```

Template name

Used by template in response

Step C: Implement Model Method

➢ Data handler

- Request data received from view
- Query data store
- Return results to view

country_code_lookup.py

```
country_codes = {'44': 'United Kingdom' ,
                '01': 'Canada',
                '33': 'France',
                '46': 'Sweden',
                '81': 'Japan'}

def find_code(code):
    if country_codes.get(code):
        country = country_codes[code]
    else:
        country = 'Unknown country code'
    return(country)
```

Returned to view

Step D: Implement Template

- **Templates are HTML files**
  - Containing `{{ variable  }}` for display

std_country_code.html

```
<html>
  <head>
     <title>Country Code</title>
  </head>

  <body>

    <h3> Code {{ std_code }} is for {{ std_country }}  </h3>

  </body>
</html>
```

Data provided by view

Step E: Map URL to View Method

- **URLs are mapped to view methods by URL dispatcher module**
  - Mappings defined in file `urls.py`
    - Use regular expressions to match URL patterns to view methods
- **Example URL is of the form `/travel/`*`std_code`*`/`**
  - The *`std_code`* value is passed to view method

`urls.py`

```
urlpatterns = patterns('',
  url(r'^travel/$', 'travel.views.index'),

  url(r'^travel/(?P<std_code>\d+)/$', 'travel.views.get_std_country')
)
```

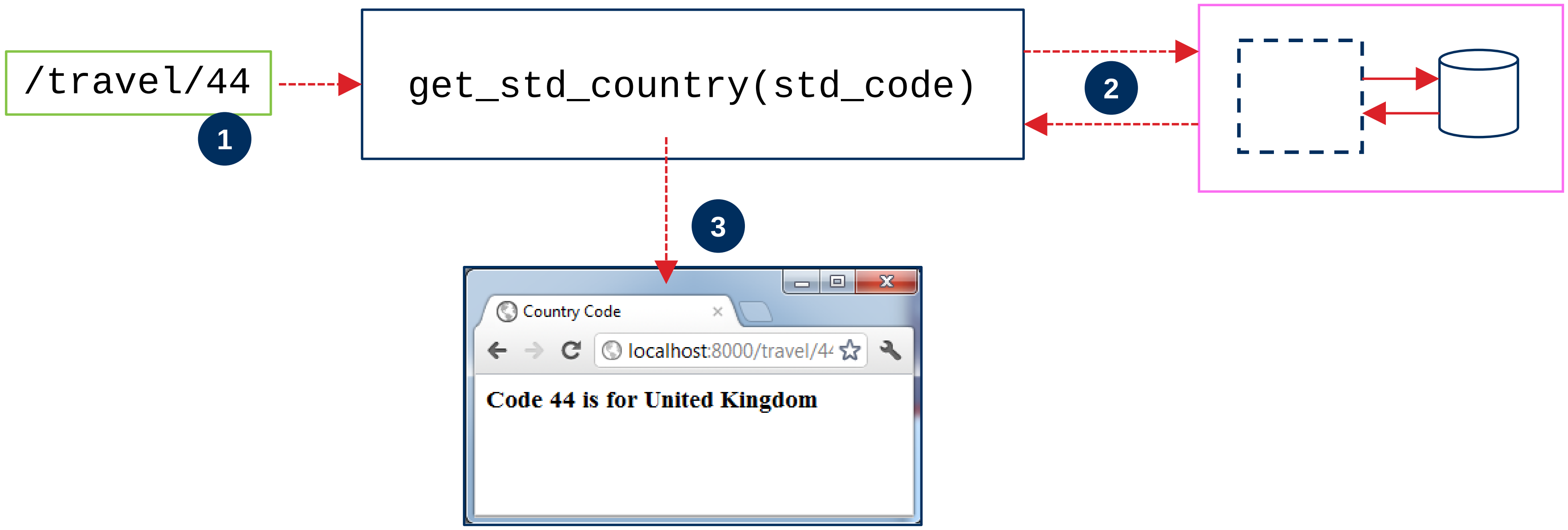
URL pattern to match

`std_code` variable references the *`country_code`* value from the URL

View method to call

Application Flow Summary

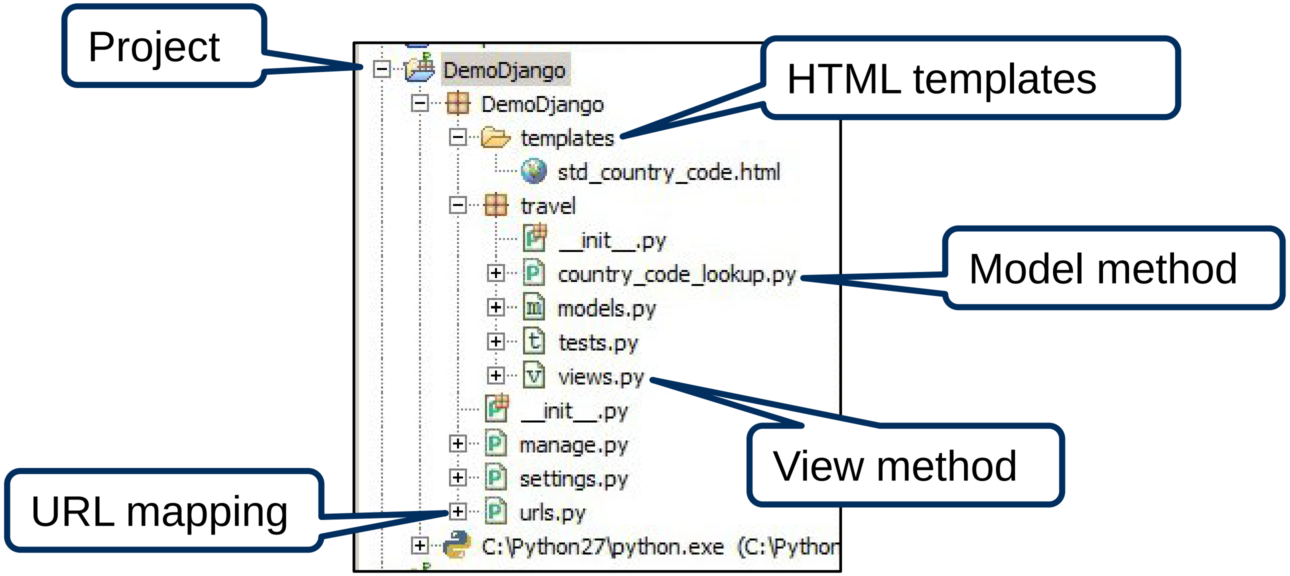
1. URL maps to a view
2. View invokes model for data processing
3. View passes results to template



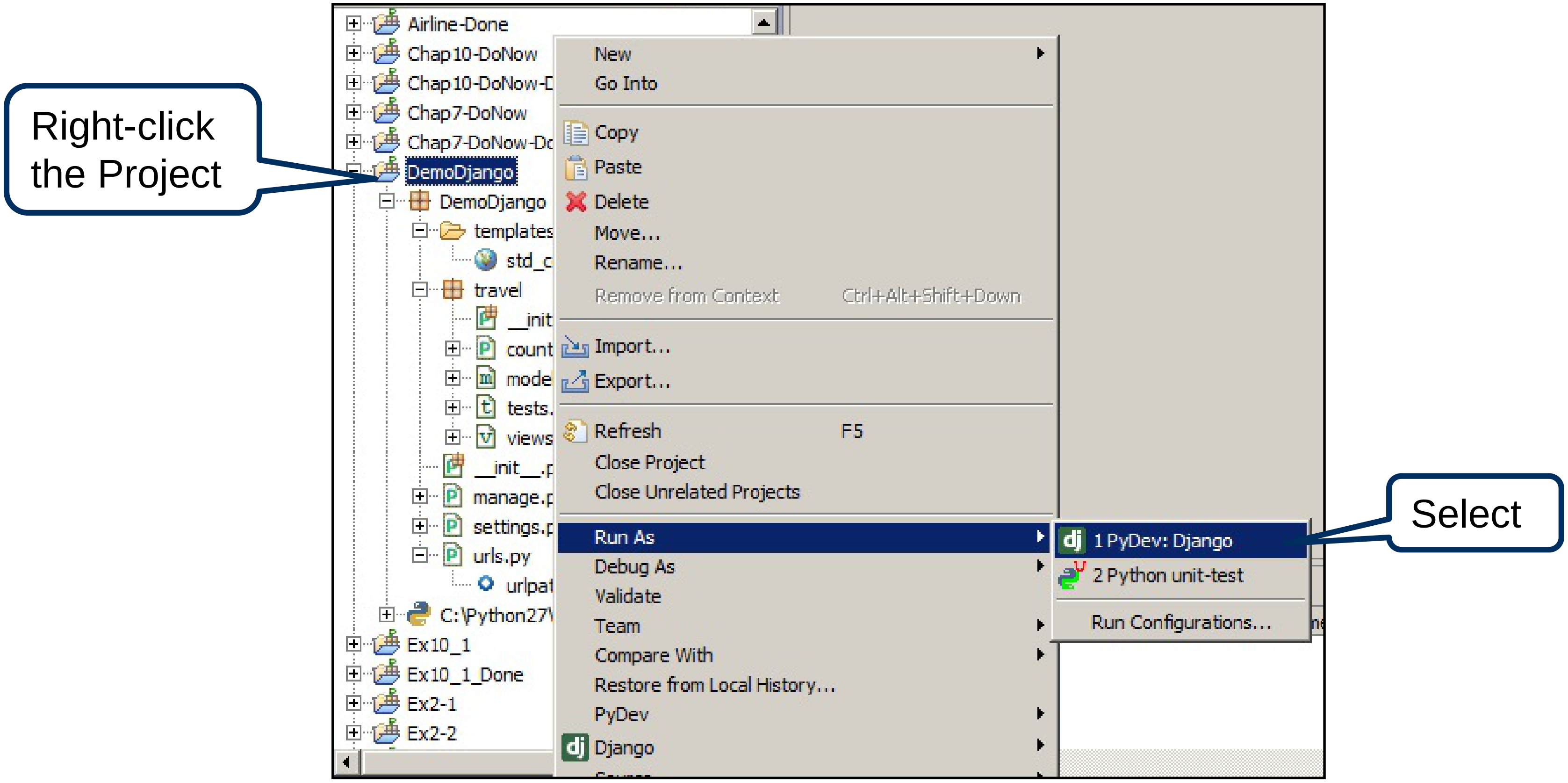
Django Project Layout

You will query a Django-powered website to look up an STD code

1. **In Eclipse, open the project `DemoDjango` to view the project infrastructure**



2. **Right-click the `DemoDjango` project; from the pop-up menu, select Run As | PyDev: Django**



3. The server startup message displays in Eclipse console:

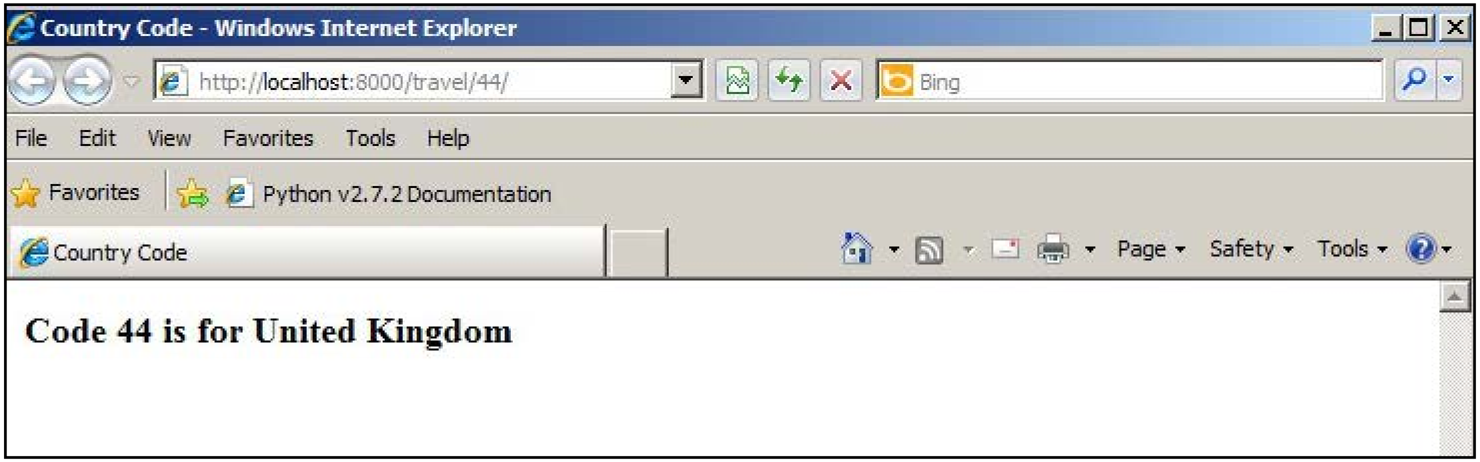
```
Development server is running at http://127.0.0.1:8000/
```

Terminates server

Do Now

4. **Open the browser, and enter the URL to request a country code**
   - `http://localhost:8000/travel/44`

5. **Template displays query result**



6. **Terminate the running server**
   - Use Red Box on Eclipse console

Hands-On Exercise 10.1

In your Exercise Manual, please refer to
Hands-On Exercise 10.1: Web Application Development With Django

Chapter Summary

You are now able to

- **Describe web application development with Python**
- **Build a Python web application using the Django framework**

Chapter 11

Course Summary

Course Summary

You are now able to

- Create, edit, and execute simple Python programs in Eclipse
- Use Python simple data types and collections of these types
- Control execution flow: conditional testing, loops, and exception handling
- Encapsulate code into reusable units with functions and modules
- Employ classes, inheritance, and polymorphism for an object-oriented approach
- Read and write data from multiple file formats
- Query relational databases using SQL statements within a Python program
- Display and manage GUI components, including labels, buttons, entry, and menus
- Create a web application with the Django framework