

Course 1906

Advanced Python: Best Practices and Design Patterns

by

Michael Woinoski

Technical Editor:

Alexander Lapajne



Copyright

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their
respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or
transmitted in any form or by any means, electronic, mechanical, photocopying,
recording or otherwise, or translated into any language, without the prior written
permission of the publisher.

Copying software used in this course is prohibited without the express
permission of Learning Tree International, Inc. Making unauthorized copies of
such software violates federal copyright law, which includes both civil and
criminal penalties.

Introduction and Overview



LEARNING TREE
INTERNATIONAL

Course Objectives

- ▶ Employ best practices and design patterns to build Python applications that are reliable, efficient, and testable
- ▶ Exploit Python's OOP features for stable, maintainable code
- ▶ Automate unit testing with the Unittest, Pytest, and Mock modules
- ▶ Measure and improve performance of Python applications using profiling tools, code optimization techniques, and PyPy
- ▶ Install and distribute Python programs and modules using standard Python tools, including Pip and Python Packaging Index (PyPI)



OOP = object-oriented programming

Course Objectives

- ▶ Create and manage concurrent threads of control
- ▶ Generate and consume REST web service requests and responses



REST = representational state transfer

Course Contents

Introduction and Overview

Chapter 1 Object-Oriented Programming in Python

Chapter 2 Design Patterns in Python

Chapter 3 Unit Testing and Mocking

Chapter 4 Error Detection and Debugging Techniques

Chapter 5 Measuring and Improving Performance

Chapter 6 Design Patterns II

Chapter 7 Installing and Distributing Modules

Chapter 8 Concurrent Execution

Chapter 9 Interfacing With REST Web Services and Clients

Chapter 10 Course Summary

Next Steps

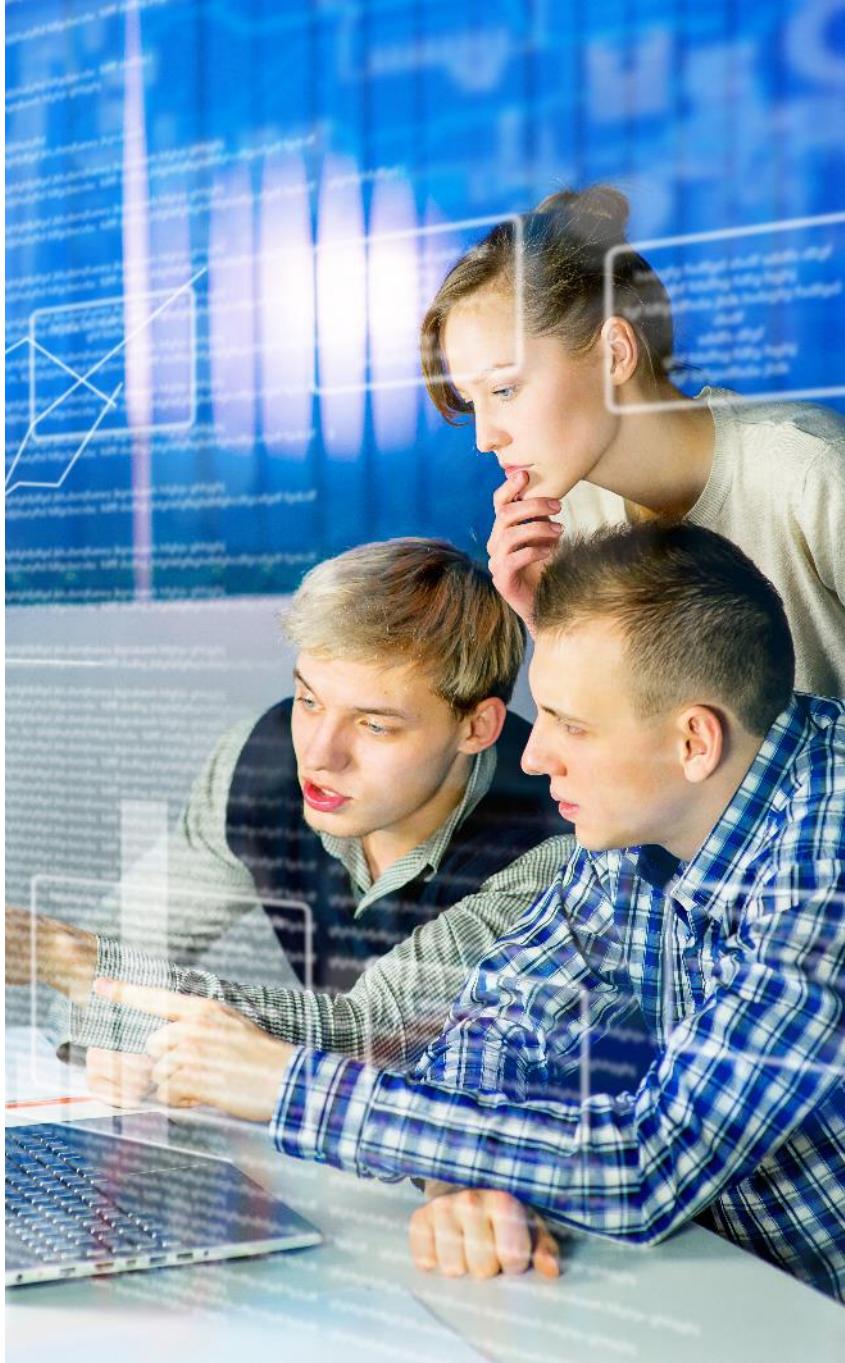
Course Contents

Appendix A Extending Python

Appendix B Advanced Python Features

Goals of the Course

- ▶ **Gain experience implementing design patterns in Python**
 - Experienced programmers may use same design patterns in all languages
- ▶ **You'll be able to apply Python best practices in your code**
 - Applying best practices results in better code
- ▶ **You'll be able to read and understand complex Python code**
 - We'll look at some Pythonic "magic" so you will understand how it works



Expected Background

- ▶ Real-world experience in Python is expected to the level of [Course 1905, Introduction to Python Training](#)
 - Familiarity with the core Python language features
 - Defining modules and packages
 - Built-in data types, decisions (`if/elif/else`), looping (`while, for`)
 - Defining and calling functions
 - Exception handling: `try/except/finally, raise`
 - Object-oriented programming in Python

Our Classroom Environment

► **Software installed on your workstations:**

Python 2, Python 3	Python interpreters
PyCharm, IDLE	Python IDEs (PyCharm for exercises)
Notepad++, VS Code, Atom, vim	Text editors
MySQL	Database for examples and exercises
WSL Ubuntu Linux	Windows Subsystem for Linux with Ubuntu
Clink	bash line editing and command history for Windows command prompt

Programming Environment

- ▶ **You'll complete hands-on exercises in a Windows environment**
 - Most examples and exercises will run unchanged on UNIX, Linux, and Mac
- ▶ **Examples and exercises are compatible with Python 3**
 - Some Python 3 changes are not backward compatible with Python 2
 - Lecture material highlights differences between Python versions
 - Python 3 syntax is identified this symbol: 
- ▶ **Exercise instructions are written for PyCharm IDE**
 - For interactive use, PyCharm includes a Python console

Hands-On Exercises

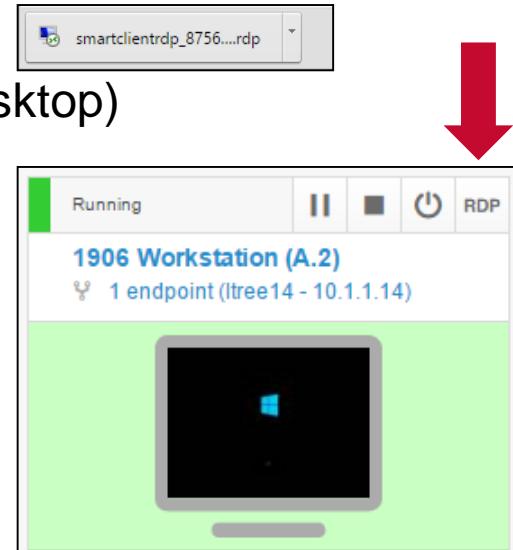
- ▶ **Exercises appear throughout the course**
 - To reinforce your learning
- ▶ **All exercises have bonus sections to complete if you have more time**
 - You won't have time to finish them all in class!
- ▶ **Hints are available for all of the coding exercises**
 - Open a web browser and visit the course home page
- ▶ **All exercises, solutions, and course examples are available for download**
 - Visit My Learning Tree
 - Navigate to My Courses/Transcripts
 - Find Python Best Practices and Design Patterns and click View button
 - Under Course Notes and Exercise Manuals, click the link for Download your CD1
- ▶ **You will do the hands-on exercises on a remote Virtual Machine (VM)**

Accessing Your Virtual Machine (VM)

Do Now

► Perform these steps to access your VM:

1. Open Chrome and navigate to <https://www.learningtree.com>
2. Log in to My Learning Tree
 - Your instructor will give you a temporary password if you need it
3. Click **Course History > Course Enrollments > Advanced Python Best Practices and Design Patterns > Details**
4. Click **Access My Virtual Machine**
5. Click the **RDP** button (above the thumbnail of the desktop)
6. Click **Download RDP**
7. Click the downloaded file in the Chrome status bar
8. Click **Connect**
9. Click **Yes** in the warning dialog about the remote computer's identity
10. Enter your credentials
 - Username: user
 - Password: pw
11. Right-click the desktop > Screen Resolution and select the resolution of your monitor



Course Notes

- ▶ **The Course Notes are organized to facilitate learning**
 - Not intended to be used as a primary reference
- ▶ **The Course Notes provide the big picture, then slowly fill in more detail**
 - Suggested reference: standard Python documentation
 - Available as Windows help files or online
 - Online:
<https://docs.python.org/3/>
 - For more information, see Supplemental Course Materials (SCM) on My Learning Tree



Chapter 1

Object-Oriented

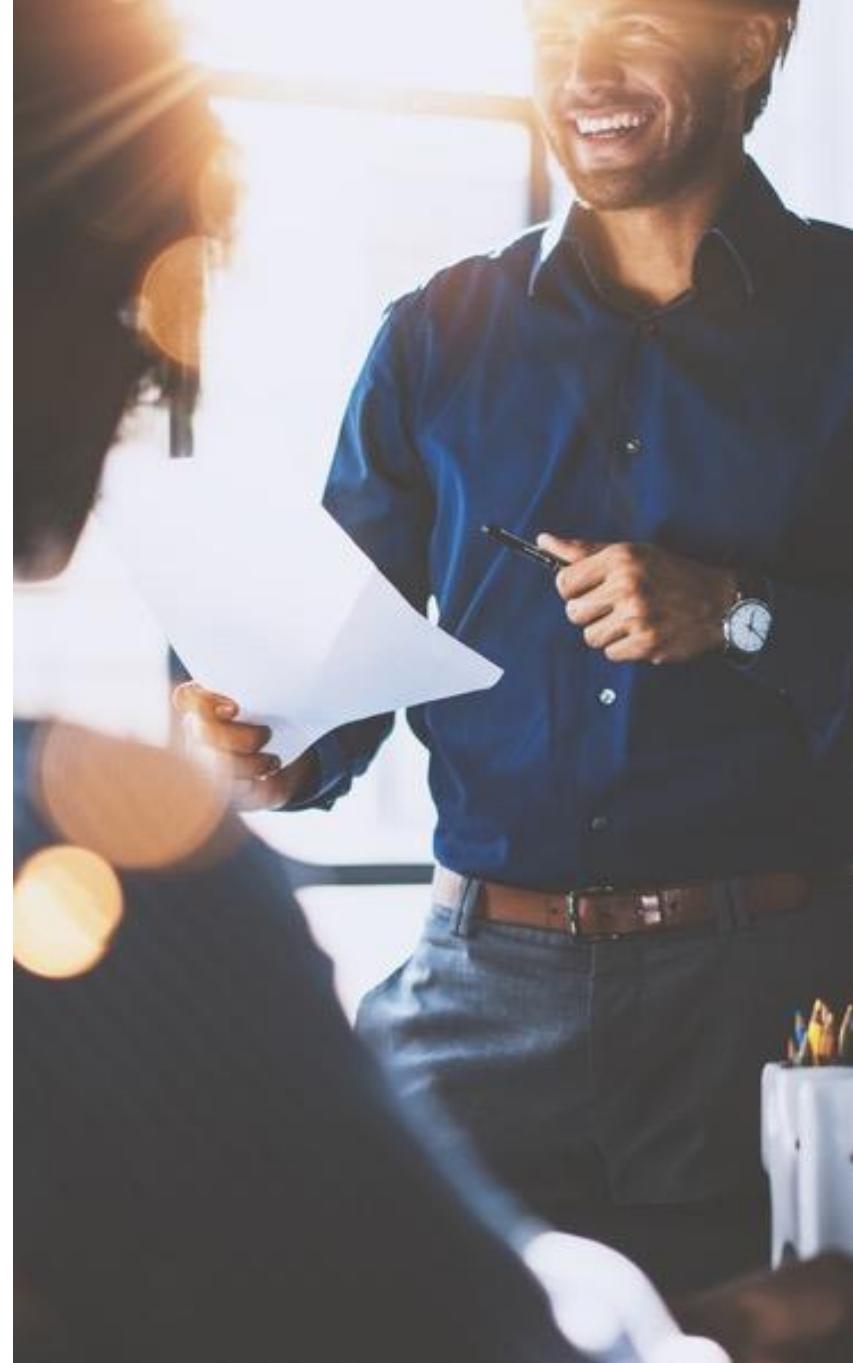
Programming in Python



LEARNING TREE
INTERNATIONAL

Objectives

- ▶ Define Python classes and create objects
- ▶ Extend classes to define subclasses using inheritance



Contents

Python Feature Review

- ▶ OOP in Python
- ▶ Type Hinting
- ▶ Hands-On Exercise 1.1



Review: List Data Type

► **list: Mutable ordered sequence of items**

- Create a list with []

```
cities = ['London', 'Ottawa', 'Stockholm']
populations = [8.6, 0.9, 0.9] # populations in millions
```

► **To select a list item, use a numeric index**

```
print('The value of', cities[0], 'is', populations[0])
```

► **List operations**

- Select item by index: populations[1]
- Select range of items (slice): cities[0:2]
 - Selects ['London', 'Ottawa']
- Append item: cities.append('Washington DC')
- Process all list items using for loop:

```
for city in cities:
    print(city.upper())
```

Selects items start to (end - 1)

Review: Tuple Data Type

► tuple: Immutable ordered sequence of items

- *Immutable*: You can't add or delete items
- Create tuple with ()

```
item = ('pi', 3.1415927)
```

Create tuple of two items

► Tuples can be indexed like lists

```
print('The value of', item[0], 'is', item[1])
```

```
item[0] = 'e' # Error
```

You can't modify the tuple

Retrieve first item in tuple

List Comprehensions

► List comprehension: Concise syntax for creating new lists

- Syntax: [*expression-with-var* for *var* in *list*]
- Execution:
 1. Loop over list, assign each item to var
 2. Each pass through the loop, evaluate *expression-with-var*
 3. Add the value of the expression to a new list

```
populations = [8.6, 0.9, 0.9]  
nums = [1, 2, 3, 4, 5, 6]
```

Evaluate the
expression str(p)...

...for each item in the
list populations

```
pop_strs = [str(p) for p in populations]
```

```
squares = [x**2 for x in nums]
```

Evaluate the
expression x**2...

...for each item in
the list nums

Equivalent for loop:

```
pop_strs = []  
for p in populations:  
    pop_strs.append(str(p))
```

List Comprehensions With Conditions

► List comprehension can include an **if** condition

- Syntax: [*expr-with-var* for *var* in *List* if *test-with-var*]
- *expr-with-var* is evaluated for a list item only if the test is true
- Order of evaluation:
 1. for loop
 2. if test
 3. If the test is true, *expr-with-var*

Evaluate expression...

...for all items...

...greater than 1

```
large_pop_strs = [str(p) for p in populations if p > 1]
```

```
even_squares = [x**2 for x in nums if x % 2 == 0]
```

Evaluate expression...

...for all items...

...that are even

Review: Dictionary Data Type

► dict: set of key/value pairs used as a lookup table

- Create a dict with '{}' and a list of keys and values separated by ':'

```
cap_lookup = { 'Canada': 'Ottawa',
                'Sweden': 'Stockholm',
                'United Kingdom': 'London' }
country = 'Canada'
print('Capital of', country, 'is ', cap_lookup[country])
cap_lookup['United States'] = 'Washington DC'
```

Look up value using key as index

Adds or replaces value

► Loop through all keys in dict with a for loop

```
for country in cap_lookup.keys(): # same as for country in cap_lookup:
    print('Capital of', country, 'is ', cap_lookup[country])
```

► Loop through all items in dict using tuples

```
for country, capital in cap_lookup.items():
    print('Capital of', country, 'is ', capital)
```

Each dictionary item is a tuple

Review: Function Definitions

- ▶ Define a function with the **def** keyword
- ▶ Function can have variable length argument list with *****

```
cap_lookup = {  
    'Canada': 'Ottawa',  
    'Sweden': 'Stockholm',  
    ...}  
  
def get_capital(*countries):  
    return [cap_lookup.get(country) for country in countries]  
  
caps = get_capital('Canada', 'Sweden')
```

Receives all arguments
in a single tuple

Pass any number
of arguments

Loop over tuple
of arguments

Review: Function Definitions

- ▶ **Function parameters can have *default values***
 - Those arguments can be omitted when the function is called
- ▶ **Values can be passed to function as *keyword arguments***
 - Order of keyword arguments can be changed

```
def print_greeting(name, end=10, step=1):  
    for i in range(1, end+1, step):  
        print('{}. Hi, {}'.format(i, name))  
  
print_greeting('Adam', 5, 2) # all values passed  
print_greeting('Adam', 4)    # step defaults to 1  
print_greeting()           # TypeError: name has no default  
print_greeting(end=5, name='Adam')
```

No quotes around
keyword argument names

If keyword syntax is used,
arguments can be in any order

Review: Function Definitions

- ▶ You can access keyword arguments using dictionary syntax
 - In this example, `**kwargs` is a dictionary of keyword arguments

```
def print_keyword_args(**kwargs):  
    for name, value in kwargs.items():  
        print("{}'s value is {}".format(name, value))  
  
print_keyword_args(john=10, jill=12, david=15)
```

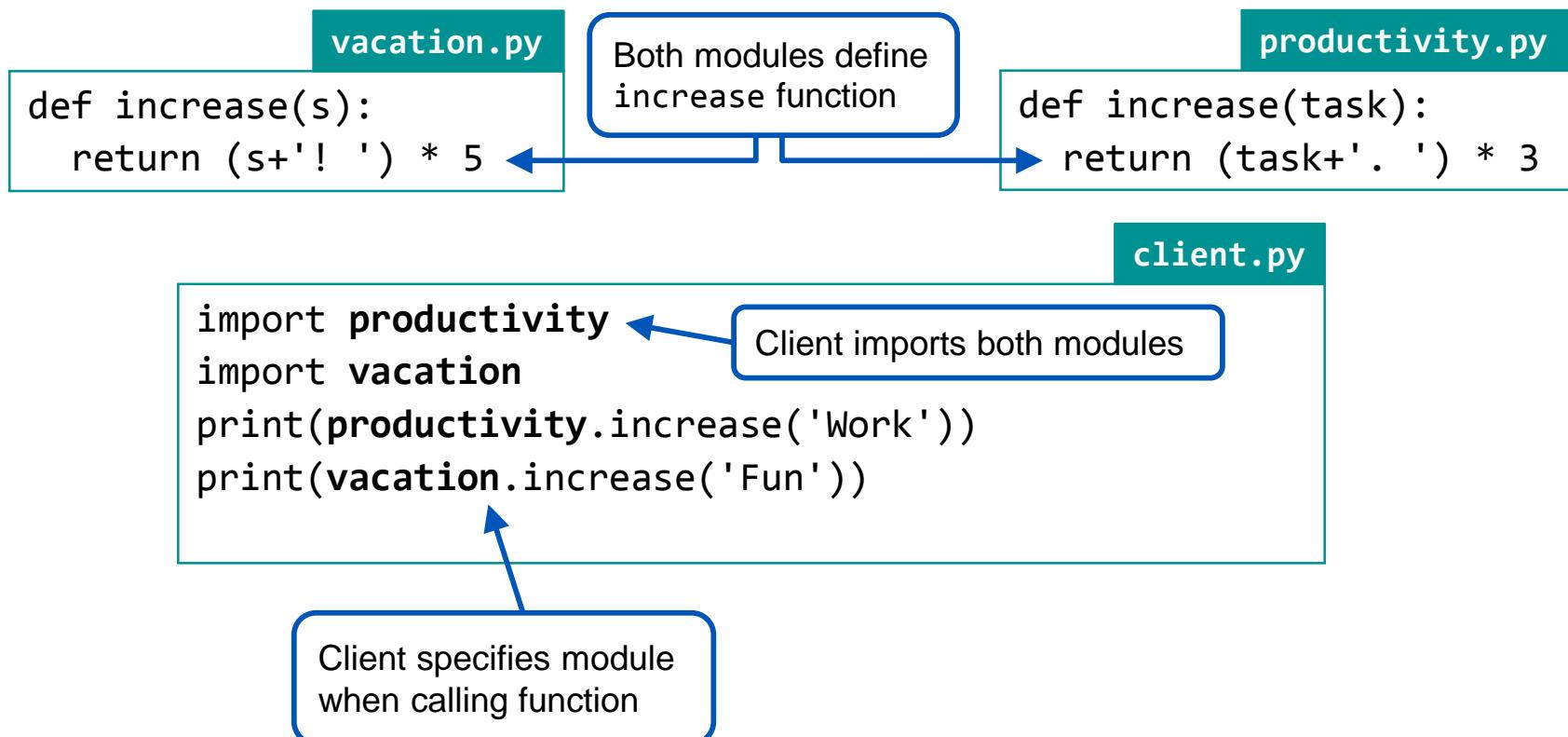
- ▶ Output

```
john's value is 10  
david's value is 15  
jill's value is 12
```

Review: Importing Python Modules

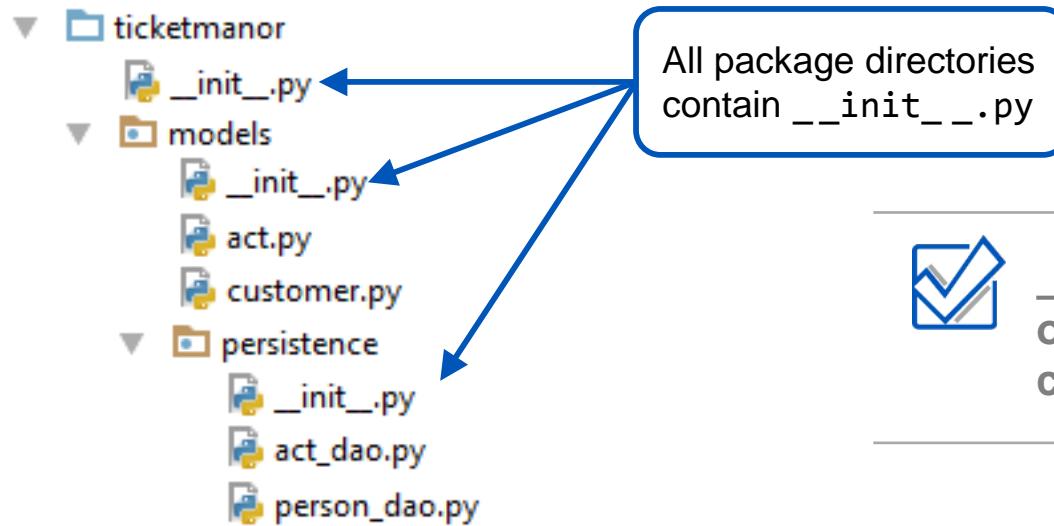
► Each Python source file defines a *module*

- Each module defines a *namespace*
- Names in one namespace do not conflict with names in other namespaces
- Names must be *imported* before use



Review: Packages

- **Modules can be defined in a *package hierarchy***
 - Packages make it easier to organize applications with many modules
- **A directory defines a package if it contains a file named `__init__.py`**



`__init__.py` may be empty, or it may define functions, classes, etc.

- **import statements specify the path through the package hierarchy**

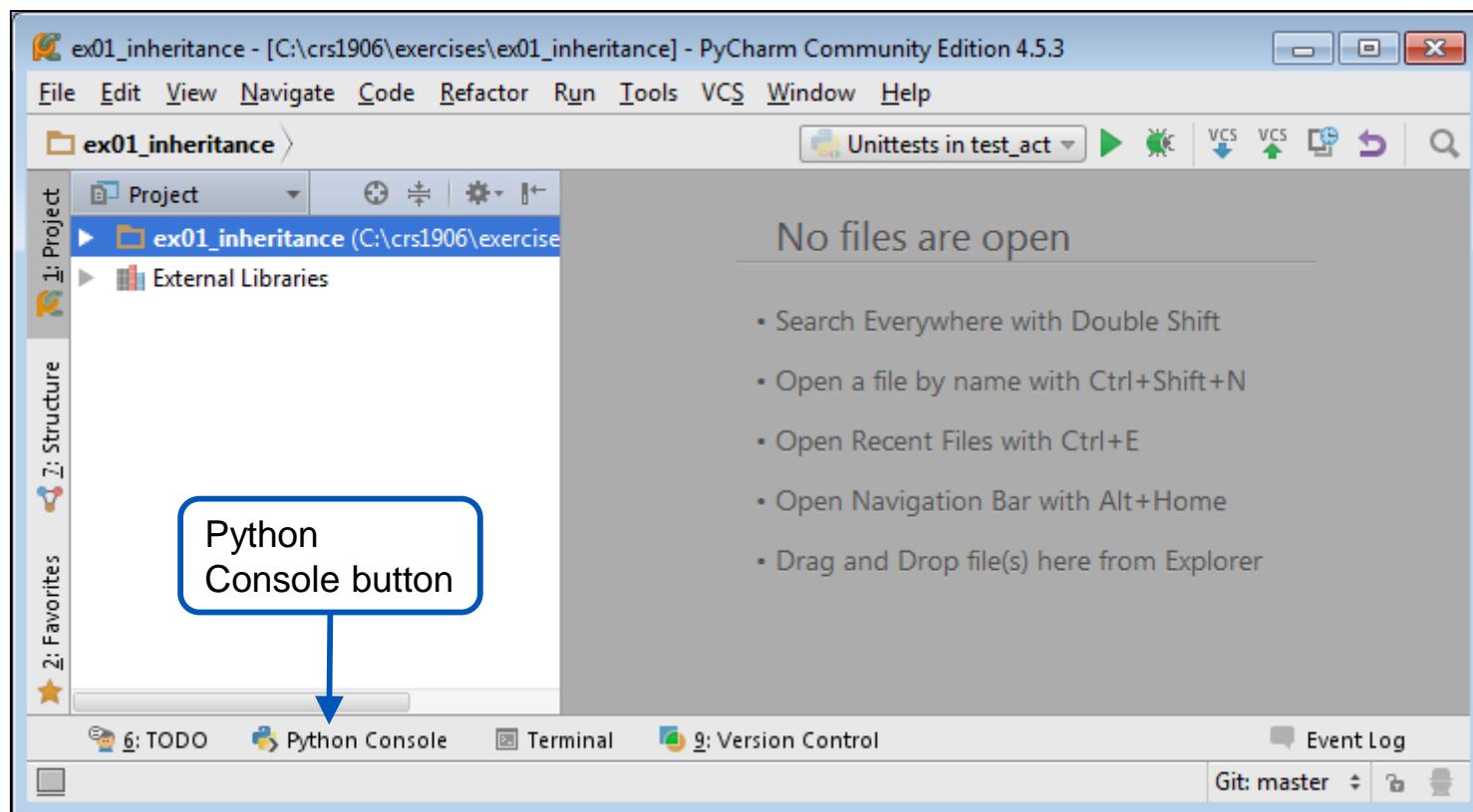
client.py

```
import ticketmanor.models.act
import ticketmanor.models.persistence.person_dao
```

Executing Python Statements Interactively

Do Now

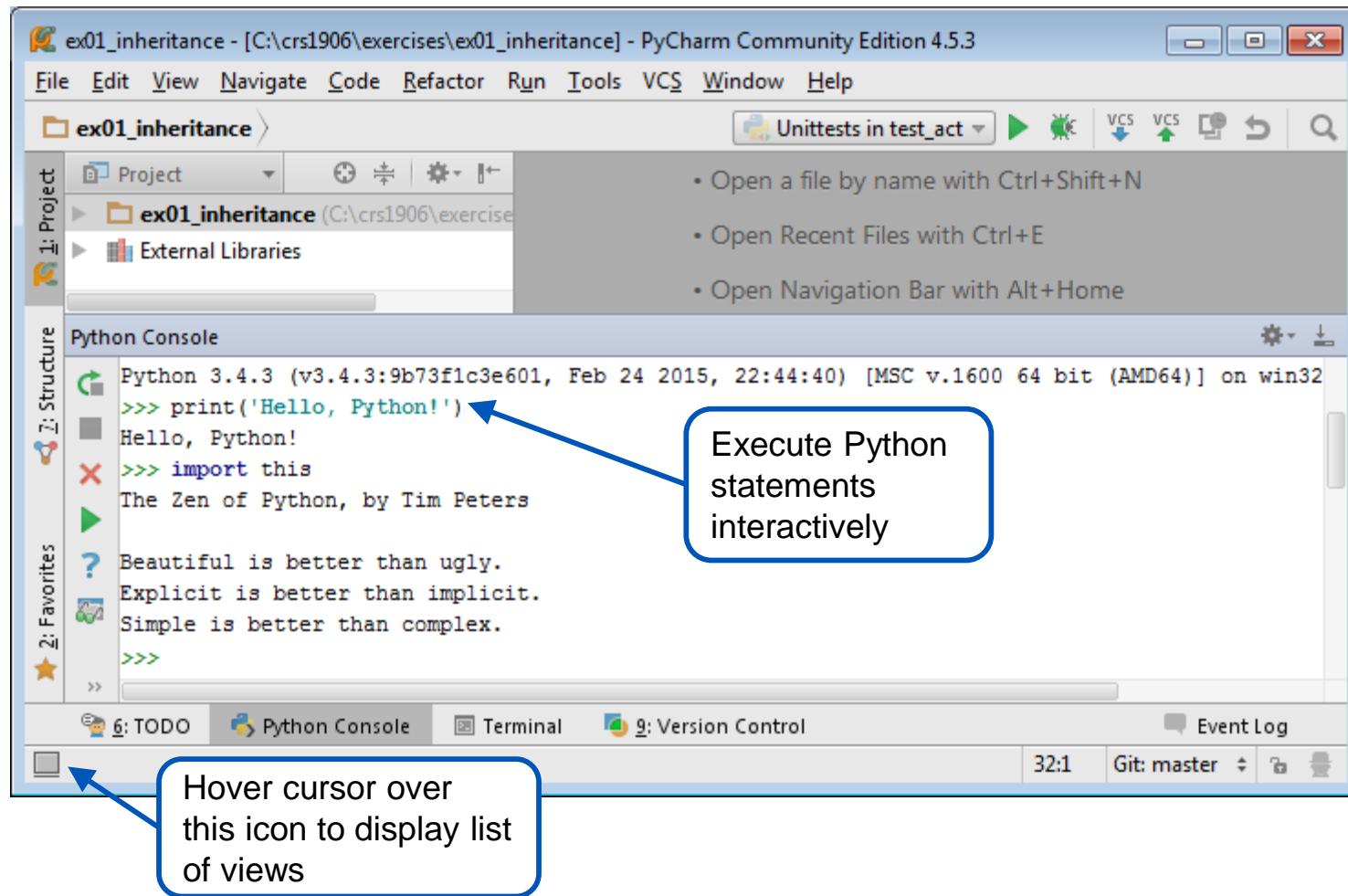
- ▶ Start the PyCharm IDE from the icon on the desktop
- ▶ Click the Python Console button



Executing Python Statements Interactively

Do Now

3. Enter Python statements in the console and examine the output



Contents

- ▶ Python Feature Review

OOP in Python

- ▶ Type Hinting
- ▶ Hands-On Exercise 1.1



Review: Python Object-Oriented Programming

- ▶ **Python is an object-oriented language**
 - Supports classes, objects, encapsulation, inheritance, polymorphism
- ▶ **Define classes with the `class` keyword**
 - First argument to an instance method is a reference to the current object

simple_counter.py

```
class SimpleCounter:
```

```
    """A simple example class"""
```

```
    def __init__(self, start):
```

```
        self.count = start
```

Python calls `__init__()` to initialize a new object

```
    def increment(self, incr=1):
```

```
        self.count += incr
```

```
        return self.count
```

count attribute “springs into existence” when assigned

```
    def __str__(self):
```

```
        return 'count={}'.format(self.count)
```

Instance method; `self` is reference to current object

Python calls `__str__()` to convert `SimpleCounter` to string

Review: Python Object-Oriented Programming

- ▶ Each object has its own copy of the instance attributes
- ▶ Python doesn't support private attributes or methods

```
counter1 = SimpleCounter(0)  
counter2 = SimpleCounter(100)
```

Create two
SimpleCounter
instances

```
print(counter1.count)  
counter1.count += 1  
print(counter1.count)
```

You can access instance
attributes directly

```
counter2.increment(5)  
msg = 'Counter2 value: ' + str(counter2)  
print(msg)
```

Reference to object is
passed as hidden first arg

counter1

count

0

counter2

count

100

Built-in str() function calls
object's __str__()

Review: Static Methods

- ▶ **Classes may have *static methods***
 - Static method can't reference current instance or class
 - Used to bundle utility methods with classes
- ▶ **Static methods are identified with `@staticmethod` decorator**
 - Decorator: A function that modifies the behavior of classes or other functions
 - Decorators can be applied with the syntax `@decorator-name`

```
import time
class Event:
    def __init__(self):
        self.timestamp = Event.now()

    @staticmethod
    def now():
        """Return current date and time as a string"""
        return time.strftime("%Y-%m-%d %H:%M:%S")
```

Use class name to call static method

Static method doesn't have a self parameter

Review: Class Data Attributes and Class Methods

► Classes may have *class data attributes*

- Class attributes are initialized outside all instance methods
- All instances of the same class share the same class attributes
- Use class name or class object to reference class data attributes

► Classes may have *class methods*

- Method definition includes built-in `@classmethod` decorator
- Method receives a class parameter instead of an instance parameter
- Unlike static methods, class methods support polymorphic method calls

```
class SimpleCounter:  
    how_many_counters = 0  
  
    def __init__(self, start):  
        self.count = start  
        SimpleCounter.how_many_counters += 1  
  
    @classmethod  
    def get_instance_count(cls):  
        return cls.how_many_counters  
        # or SimpleCounter.how_many_counters
```

how_many_counters	0
counter1	count 0
counter2	count 100

Review: Class Inheritance

► Classes can extend other classes

- Subclass inherits methods from superclass

simple_counter.py

```
class UpAndDownCounter(SimpleCounter):
    """UpAndDownCounter defines a decrement operation"""

    def __init__(self, start, min=0):
```

```
        super().__init__(start)
        self.min = min
```

Subclass `__init__()` calls superclass `__init__()` using built-in function `super()`

Py2: `super(UpAndDownCounter, self).__init__(start)`

```
    def decrement(self, decr=1):
        if self.count - decr > self.min:
            self.count -= decr
        return self.count
```

Subclass references attributes created in superclass

```
    def __str__(self):
        return '{{},min={}}'
                .format(super().__str__(), self.min)
```

Subclass `__str__()` function calls superclass's `__str__()`

Note: `super()`, not `super`

Review: Class Inheritance

► Create subclass instances as before

Initialize subclass instance

```
up_and_down = UpAndDownCounter(10, min=1)
```

Call inherited
superclass method

```
up_and_down.increment()
```

Call method
defined in subclass

```
up_and_down.decrement()
```

```
print('up_and_down = ', str(up_and_down))
```

Implied call to `__str__()`



Superclass definition

```
class SimpleCounter:  
    def __init__(self, start):  
        self.count = start  
    ...  
    def increment(self, incr=1):  
        ...
```

Private Attributes of Classes

- ▶ **Python does not have private data attributes or methods**
 - All data attributes and methods may be accessed directly by client code
- ▶ **Python convention: Attributes whose names start with '_' are private**
 - Treat them as implementation details that might change without notice
 - But interpreter does *not* enforce this

```
class SimpleCounter:  
    def __init__(self, start):  
        self._count = start  
  
    def increment(self, incr=1):  
        self._count += incr  
        return self._count  
  
    def _serialize(self):  
        ...  
  
    def save(self):  
        ...  
        self._serialize()
```

_count should not be accessed directly by outside code

_serialize() should not be called by outside code

Properties

► Classes can define *properties*

- A property is a data attribute that has methods to get, set, or delete its value
 - Allows class author to change implementation without breaking other code
 - But Python still does not prevent direct access of data attribute
- Example: new requirement: SimpleCounter must keep history of values

```
class SimpleCounter:  
    def __init__(self, start=0):  
        self._count_history = [start] # replace single count with list  
  
    @property  
    def count(self):  
        return self._count_history[-1] # return latest value in history  
  
    @count.setter  
    def count(self, value):  
        self._count_history.append(value) # add value to history  
  
c = SimpleCounter()  
c.count = 10      # A hidden call to the setter method  
print(c.count)   # A hidden call to the getter method
```

Getter method

Setter method

Contents

- ▶ Python Feature Review
- ▶ OOP in Python

Type Hinting

- ▶ Hands-On Exercise 1.1



Type Hinting

► Python 3.5+ supports *type hints* in code

- Can specify data type of variables, and function parameters and return values
- Hints for built-in types use int, float, bool, str, bytes

```
def stringify(num: int) -> str:  
    return str(num)
```

Type of parameter is int

```
def sum(num1: int, num2: int) -> int:  
    return num1 + num2
```

Type of function's return value is str

```
def add(num1: int, my_float: float = 3.5) -> float:  
    return num1 + my_float
```

Each function parameter has its own type hint

```
def print_sum(num1: int, num2: int) -> None:  
    print("{} + {} = {}".format(num1, num2, sum(num1, num2)))
```

Argument default value goes after type

```
If function doesn't return a  
value, its return type is None
```

The MyPy Module

- ▶ **CPython interpreter ignores type hints**
 - Code will still run, even if types are used incorrectly
- ▶ **Most Python IDEs will process type hints and warn about type mismatches**
- ▶ **To perform static type checking outside your IDE, install and run MyPy**
 - mypy doesn't run your program, it only performs static type checking

```
> pip install mypy  
> mypy file_to_check.py
```

- ▶ **MyPy usage example:**

type_hints_example.py

```
def stringify(num: int) -> str:  
    return str(num)
```

```
result = stringify('one')
```

Argument has wrong data type

```
> mypy type_hints_example.py
```

```
type_hints_example.py:24: error: Argument 1 to "stringify" has  
incompatible type "str"; expected "int"
```

Type Hinting for Collections and Classes

- To give hints for standard collection types, import typing module
 - typing module defines types List, Set, Dict, Tuple

```
from typing import List, Dict

def stringify_list(nums: List[int]) -> Dict[str, int]:
    return { (stringify(val), val) for val in nums }
```

Argument type
and return type
are collections

- Instance attributes and methods can use type hints

```
class UpAndDownCounter(SimpleCounter):
    count: int
    min: int

    def __init__(self, start: int, min: int = 0) -> None:
        self.count = start
        self.min = min

    def increment(self, incr: int = 1) -> int:
        self.count += incr
        return self.count

    def write_to_file(self, filepath: str) -> None:
        ...
```

Declare types of instance attributes
(note: these are *not* class attributes)

self does not have a type hint

Types for method arguments
and return values are
declared as usual

Contents

- ▶ Python Feature Review
- ▶ OOP in Python
- ▶ Type Hinting

Hands-On Exercise 1.1



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)**



- 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)

9. **Web only:** Move to the `Page_Load` method and it becomes the game-saving logic; i.e., change all `game` and both occurrences of `cardDeck` to `TehiGame`.

 **Web only:** The completed code should look like:

```
To copy to the clipboard, type Ctrl+C while highlighted
game = (TehiGame)Session["game"];
if (game == null)
{
    game = new TehiGame();
    Session["game"] = game;
}
```

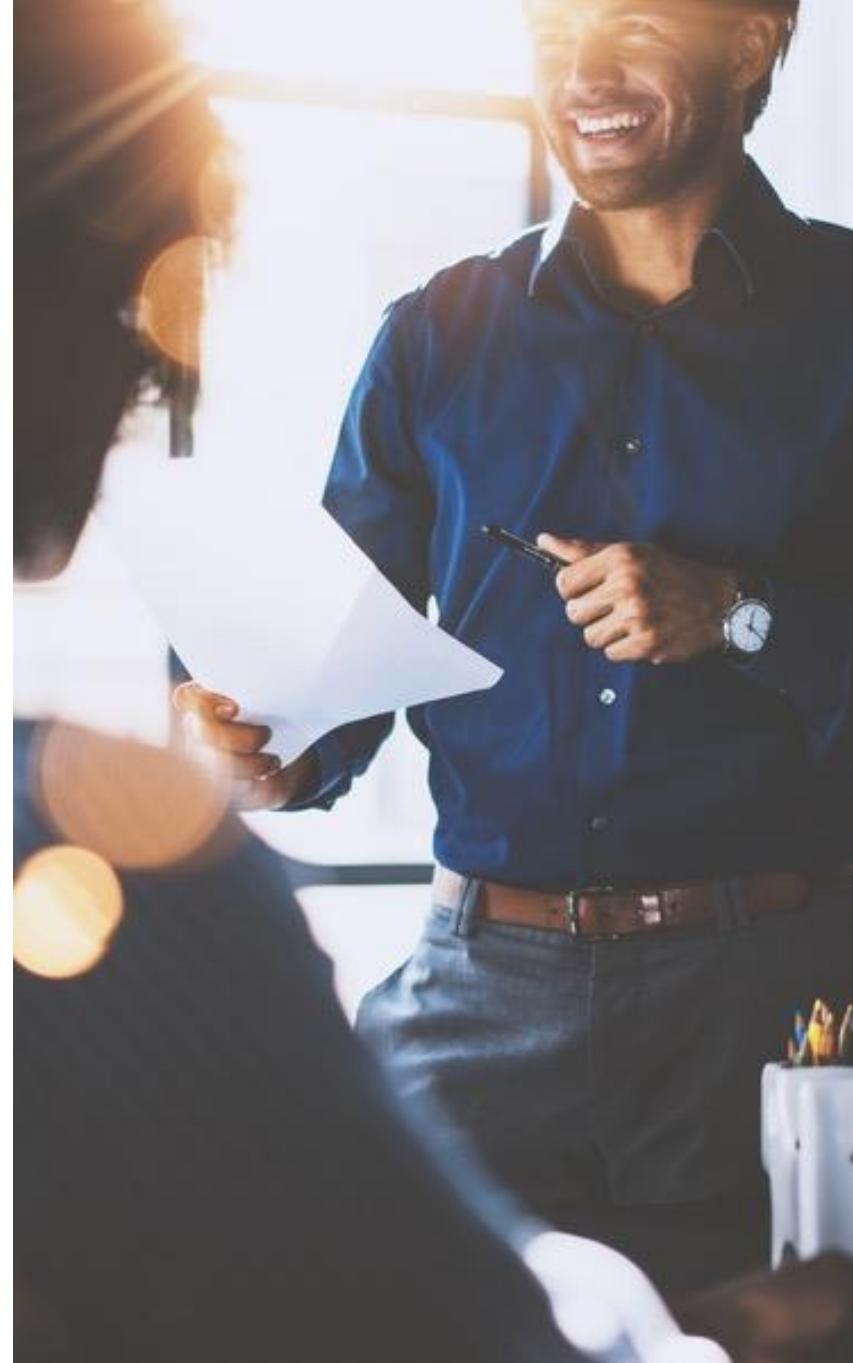
Hands-On Exercise 1.1

In your Exercise Manual, please refer to
Hands-On Exercise 1.1: Python Object-Oriented Programming



Objectives

- ▶ Define Python classes and create objects
- ▶ Extend classes to define subclasses using inheritance



Chapter 2

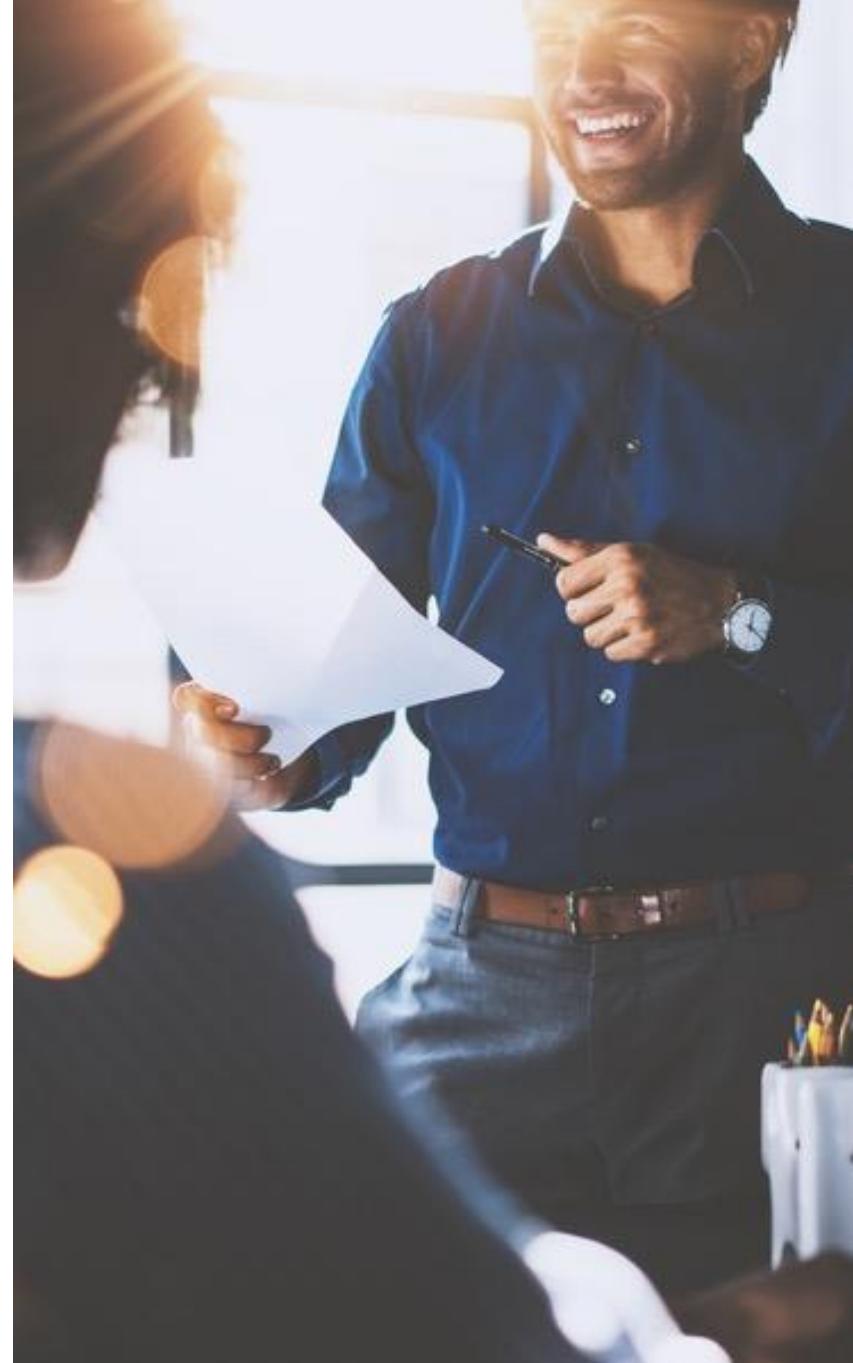
Design Patterns in Python



LEARNING TREE
INTERNATIONAL

Objectives

- ▶ Define a skeleton algorithm in the Template Method design pattern
- ▶ Loop over a collection with the Iterator design pattern
- ▶ Process collections efficiently with generator functions and generator expressions



Contents

The Template Method Pattern

- ▶ Defining and Using Abstract Base Classes
- ▶ The Iterator Pattern
- ▶ Generator Functions and Expressions
- ▶ Hands-On Exercise 2.1



Design Patterns

- ▶ ***Design pattern:* Proven, reusable solution to a specific design issue**
 - Popularized in “Design Patterns: Elements of Reusable Object-Oriented Software,” by Gamma, Helm, et al.
 - Known as Gang of Four (GoF) patterns
 - Described general patterns of object creation, structure, and behavior
- ▶ **Many other design pattern books have been written since GoF**
 - May target specific languages or problem domains
- ▶ **We'll discuss a number of common design patterns**
 - We'll focus on Pythonic implementations of each pattern



The Template Method Pattern

► Use case: Processing HTTP requests

- Many processing steps are the same for all HTTP requests
 - Get request data from URL of GET request or body of POST request
 - Convert request data from strings to language-specific types
 - Generate an HTML response
- Some processing steps are unique to each application
 - Business rules for processing request data
 - Create content of response

► Goal: Separate generic processing from application-specific steps

- But don't duplicate code
- Avoid copy-and-paste *anti-pattern*

The Template Method Pattern

- ▶ **Solution: *Template Method* design pattern**
 - Superclass method defines template of request processing algorithm
 - Superclass method delegates some steps to subclass using “hook” methods
 - Subclass implements hook methods to customize processing steps
 - Superclass template method controls when hook methods are called
- ▶ **Result: Subclass customizes processing without changing overall flow**
- ▶ **UML class diagram of Template Method**

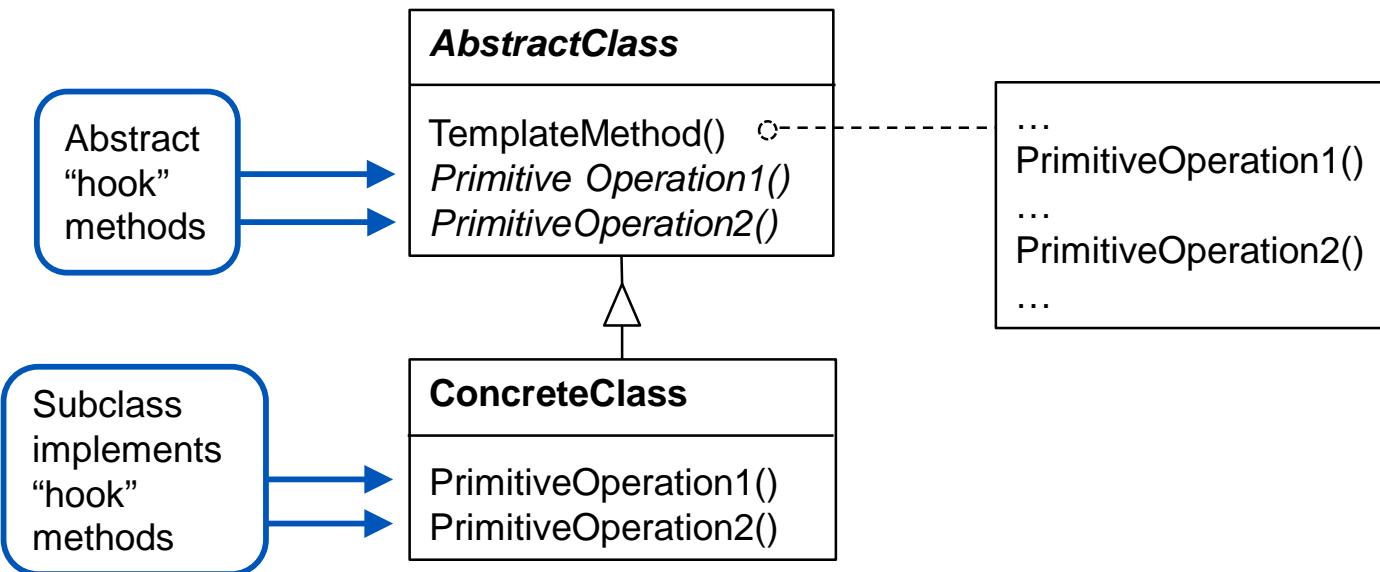


Image Source: https://commons.wikimedia.org/wiki/File:Template_Method_design_pattern.png

The Template Method Pattern

http_server.py

```
class HttpRequestProcessor:
    def handle_one_request(self):
        request_data = self.get_request_data(request) ← Generic processing
        if self.command == 'GET':
            resp, status = self.do_get(request_data)
        elif self.command == 'POST':
            resp, status = self.do_post(request_data) ← Superclass calls
                                                        subclass methods for
                                                        application-specific work
        self.return_response(resp, status) ← Generic processing

    def get_request_data(self, request):
        ...
    def return_response(self, response_data, status):
        ...

class AddUserFormProcessor(HttpRequestProcessor):
    def do_get(self, request_data):
        name = request_data['name']
        ...
    def do_post(self, request_data): ← Subclass implements
                                    hook methods
        name = request_data['name']
        ...
```

Template method

Generic processing

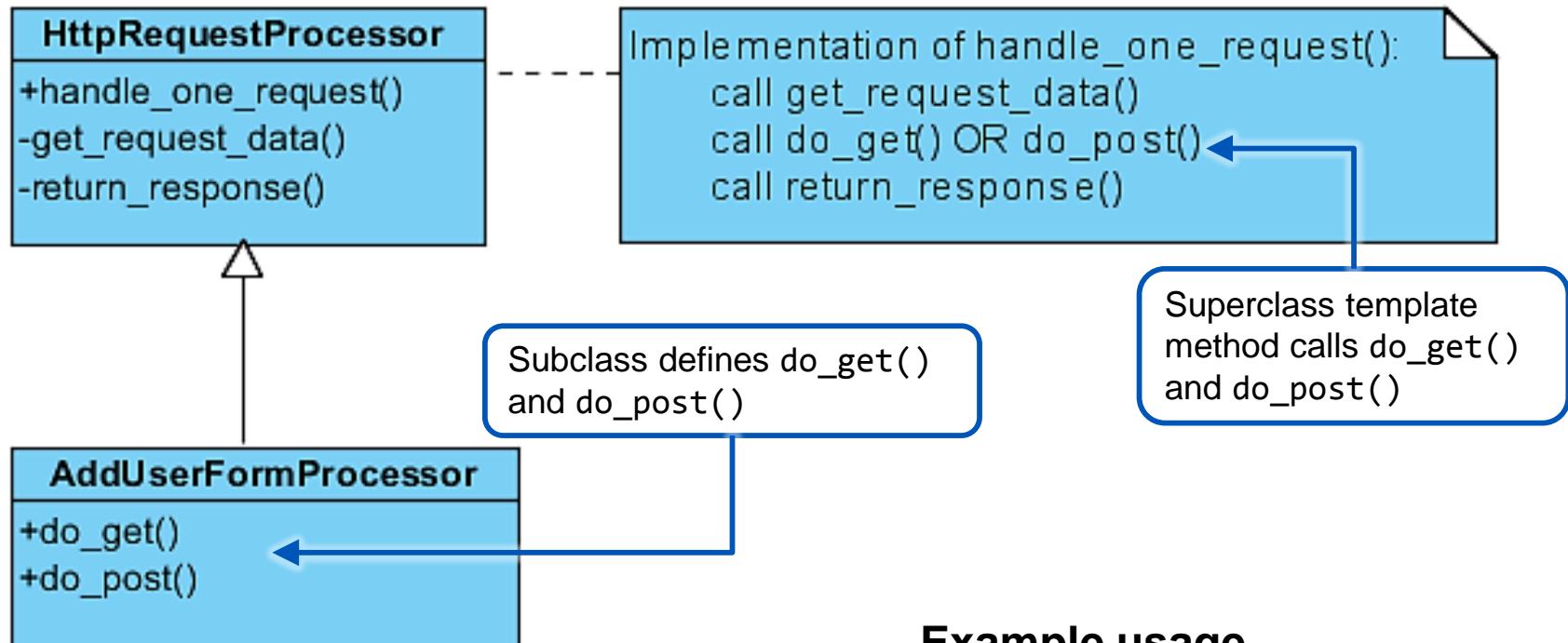
Superclass calls subclass methods for application-specific work

Generic processing

Superclass calls do_get() and do_post() but does not implement them

The Template Method Pattern

► UML class diagram of Template Method implementation



The Template Method Pattern

- ▶ **Characteristics of Template Method design pattern**
 - Superclass controls sequencing of method calls
 - Subclasses implement hook methods for application-specific tasks
 - Subclasses don't need to implement generic processing
- ▶ **Consequences**
 - Subclass logic can't accidentally change workflow
 - Generic logic in superclass can be reused by many subclasses
 - Less code duplication
 - Enforces Don't Repeat Yourself (DRY) principle



Template Method in Action

Do Now

1. Switch to PyCharm and open
C:\crs1906\examples\ch02_examples
2. Open http_server.py
 - Examine the code marked with TODO comments
3. Select "Run http_server" in the Run menu, then click the Run button
4. Switch to a command prompt and execute the following commands
 - Use the curl command to send an HTTP request
 - Note the response from the HTTP server

```
> curl -X GET "http://localhost:7777/adduser?name=Homer&job=chaos"  
{"name": "Homer", "job": "chaos"}
```

Send GET data in URL

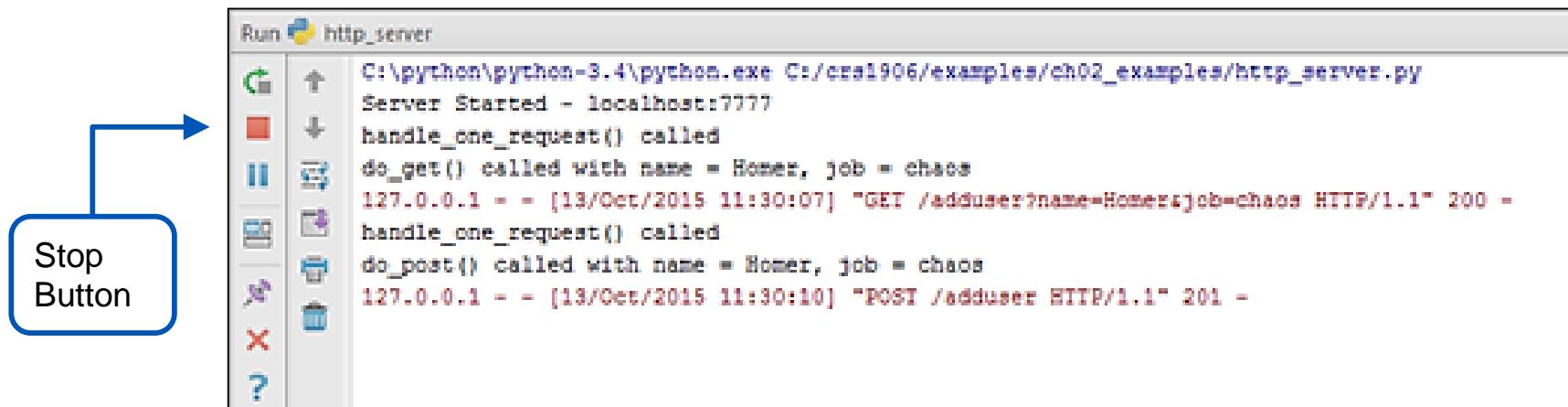
```
> curl -X POST --data "name=Homer&job=chaos" http://localhost:7777/adduser  
{"name": "Homer", "job": "chaos"}
```

Send POST data in body
of HTTP request

Template Method in Action

Do Now

5. Switch to PyCharm and note the console output from the HTTP server



The screenshot shows the PyCharm Run tool window titled "Run http_server". The left sidebar contains icons for running, stopping, pausing, and other actions. The main pane displays the following console output:

```
C:\python\python-3.4\python.exe C:/crs1906/examples/ch02_examples/http_server.py
Server Started - localhost:7777
handle_one_request() called
do_get() called with name = Homer, job = chaos
127.0.0.1 - - [13/Oct/2015 11:30:07] "GET /adduser?name=Homer&job=chaos HTTP/1.1" 200 -
handle_one_request() called
do_post() called with name = Homer, job = chaos
127.0.0.1 - - [13/Oct/2015 11:30:10] "POST /adduser HTTP/1.1" 201 -
```

6. Click the Stop button to shut down the HTTP server

Parsing Internet News Feeds

- ▶ Another use case for Template Method: Parsing Internet news feeds
- ▶ News feed formats may differ
 - RSS vs. AtomPub
 - XML vs. JSON
- ▶ But some processing will be the same for all news feed formats
 - Normalize date formats
 - Fetch images from URLs in feed
 - Parse raw format into Python data
- ▶ Define NewsFeedParser class with methods for generic processing
 - NewsFeedParser defines abstract methods for format-specific processing
 - Subclasses: RssNewsFeedParser, AtomPubNewsFeedParser
- ▶ You'll implement this use case in Hands-On Exercise 2.1

AtomPub = Atom Publishing Protocol

JSON = JavaScript Object Notation

RSS = Rich Site Summary

XML = extensible markup language

Contents

- ▶ The Template Method Pattern

Defining and Using Abstract Base Classes

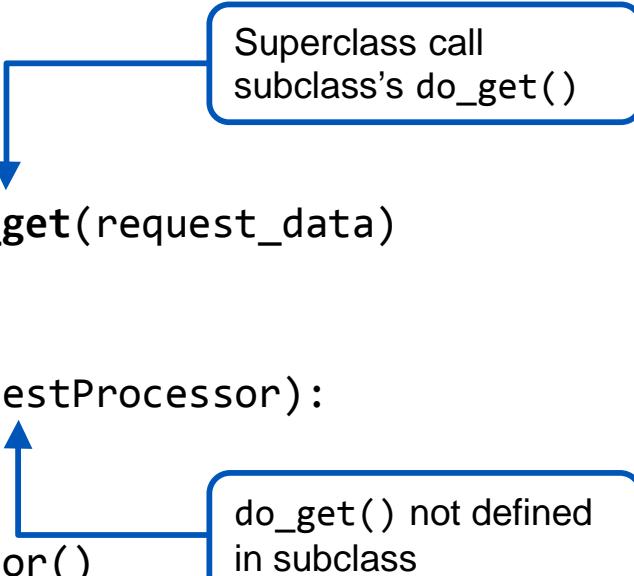
- ▶ The Iterator Pattern
- ▶ Generator Functions and Expressions
- ▶ Hands-On Exercise 2.1



Unimplemented Methods

- In `HttpRequestProcessor` example, what happens if subclass does not implement a required method?

```
class HttpRequestProcessor:  
    def handle_one_request(self):  
        ...  
        if self.command == 'GET':  
            resp, status = self.do_get(request_data)  
        ...  
  
class AddUserFormProcessor(HttpRequestProcessor):  
    ...  
  
http_processor = AddUserFormProcessor()  
http_processor.handle_one_request(...)
```



- Causes a runtime error
 - `AttributeError: object has no attribute 'do_get'`

Unimplemented Methods

- Superclass could define stub methods that raise exceptions if called

```
class HttpRequestProcessor:  
    def do_get(self, request):  
        raise NotImplementedError('do_get()')  
  
    def do_post(self, request):  
        raise NotImplementedError('do_post()')  
    ...
```

Superclass method raises exception if called

- But error won't be detected until `do_get()` or `do_post()` is called
 - Missing method might not be discovered until application is deployed
- We need a way to catch these errors early

Abstract Base Classes

► Python supports Abstract Base Classes (ABC)

- ABC can be used as base class only; cannot be instantiated directly
- Methods with `@abstractmethod` must be implemented by subclasses
- Python syntax checkers (PyCharm, Pylint) flag unimplemented methods

```
from abc import ABCMeta, abstractmethod

class HttpRequestProcessor(metaclass=ABCMeta):
    def handle_one_request(self):
        if self.command == 'GET':
            resp, status = self.do_get(request_data)
        elif self.command == 'POST':
            resp, status = self.do_post(request_data)
        ...
        @abstractmethod
        def do_get(self, request):
            pass
        @abstractmethod
        def do_post(self, request):
            pass
```

Superclass call
subclass's `do_get()`

Superclass definition of
`do_get()` is empty

Abstract Base Classes

- Subclass of abstract class must implement abstract methods
 - Otherwise subclass is also abstract

```
class MyRequestProcessor(HttpServletRequestProcessor):  
    pass
```

```
processor = MyRequestProcessor()
```

Subclass doesn't implement abstract methods

Raises TypeError with message:
“Can’t instantiate abstract class
MyRequestProcessor with abstract
methods do_get, do_post”

Contents

- ▶ The Template Method Pattern
- ▶ Defining and Using Abstract Base Classes

The Iterator Pattern

- ▶ Generator Functions and Expressions
- ▶ Hands-On Exercise 2.1



Looping Through Collections

- ▶ **The *Iterator* design pattern describes how to loop through collections**
 - Goal: Access a collection's elements without exposing the implementation
- ▶ **Python's built-in for loop meets most requirements for iteration**
 - for loops can handle looping over strings, tuples, lists, and sets
 - You don't need to implement iteration for simple use cases
- ▶ **However, sometimes you need more control when processing a collection**
 - Example: Looping over a collection of images in a GUI application
 - Initial iteration through images returns scaled-down thumbnails
 - Accessing an image at a specific index returns the full image
- ▶ **Python has features that let you control exactly how iteration works**
 - You can define special iterator methods `__iter__()` and `__next__()`, or
 - You can define generator functions and generator expressions

Iterator Methods

- To customize iteration for a class, define two special methods
 - 1. `__iter__()`: Called when class instance is used on a for loop
 - Must return an iterable object (an object that defines `__next__()`)
 - 2. `__next__()`: Iterator method called each pass through the for loop
 - for loop calls built-in `next()`, which calls class's `__next__()`
 - `__next__()` must raise `StopIteration` when iteration is done
- Example: class Reverse: for looping over a sequence backward

reverse_class.py

```
class Reverse:  
    def __init__(self, sequence):  
        self.sequence = sequence  
        self.index = len(sequence)  
  
    def __iter__(self):      ← Definition of __iter__()  
        return self  
  
    def __next__(self):      ← Definition of __next__()  
        if self.index == 0:  
            raise StopIteration()  
        self.index -= 1  
        return self.sequence[self.index]
```

Python Iterators

► Example of use of Reverse

```
# Normal iteration through a list
nums = [10, 20, 30, 40, 50]
for item in nums:
    print(item)

# Custom iteration with Reverse class
rev = Reverse(nums)
for item in rev: ← Python calls Reverse's __iter__() at start of loop
    print(item) ← Python calls Reverse's __next__() each pass
                  through loop

# No need for temporary variable
for item in Reverse(nums):
    print(item)
```

Customizing Indexing

- ▶ To customize indexing operation, define method `__getitem__()`
 - Called when instance is indexed with '[]'

double_list.py

```
class DoubleList:  
    def __init__(self, items):  
        self.items = items  
  
    def __getitem__(self, index):  
        return self.items[index] * 2
```

```
double_list = DoubleList([100, 200, 300])  
print(double_list[0])    # prints 200
```

Class defines `__getitem__()`

Python calls DoubleList's `__getitem__()` when object is indexed

Contents

- ▶ The Template Method Pattern
- ▶ Defining and Using Abstract Base Classes
- ▶ The Iterator Pattern

Generator Functions and Expressions

- ▶ Hands-On Exercise 2.1



Generator Functions

- **Generator functions are tools for creating iterators with minimal code**
 - yield statement returns next data value
 - yield statement stops iteration temporarily
 - Each time next() is called, generator resumes where it left off
 - Generator remembers local variable values and which statement executed last
 - Python creates __iter__() and __next__() methods automatically
- **Generator function implementation of reverse iteration**

```
reverse_generator.py
```

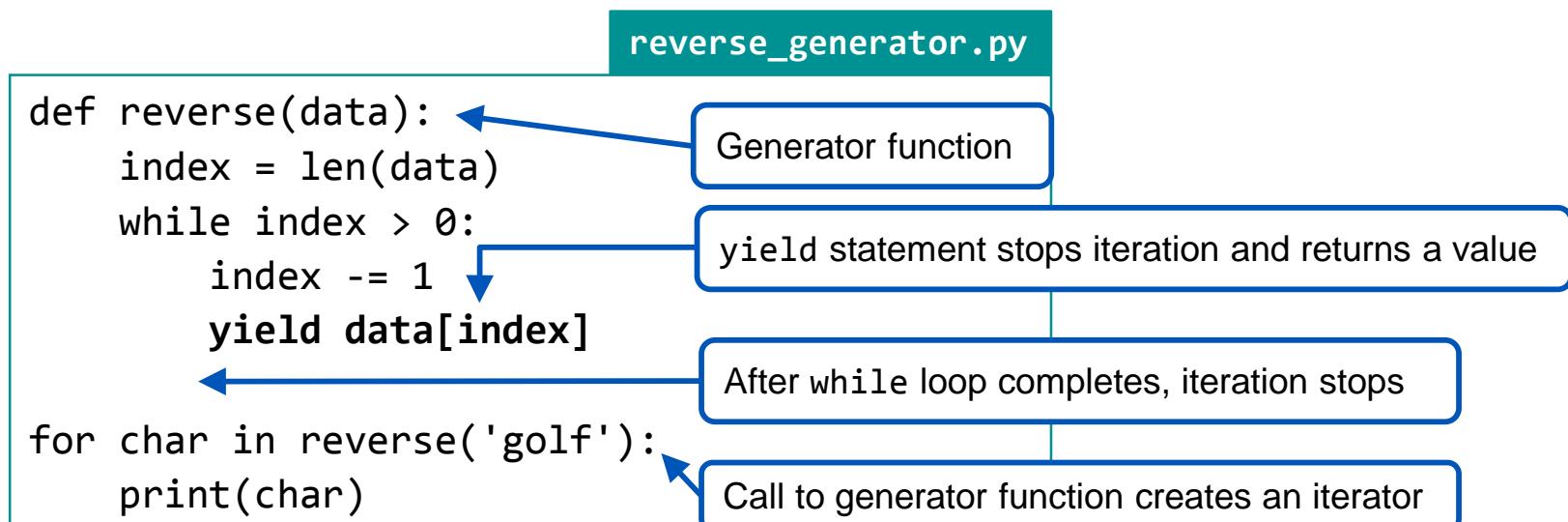
```
def reverse(data):  
    index = len(data)  
    while index > 0:  
        index -= 1  
        yield data[index]  
  
for char in reverse('golf'):  
    print(char)
```

Generator function

yield statement stops iteration and returns a value

After while loop completes, iteration stops

Call to generator function creates an iterator



Generator Function Call Sequence

► **Python processes generator function as follows:**

1. Interpreter reads generator function definition
2. Interpreter adds code that returns an iterator
3. The `for` calls generator function
4. Generator function returns iterator
5. The `for` calls iterator's `__next__()`
6. Iterator's `__next__()` executes code in body of generator until `yield`
7. Later calls to `__next__()` will resume execution where previous call ended
8. At end of generator function, interpreter raises `StopIteration`

Steps 1 through 5 occur before the execution of your code in the generator function



Generator Functions

- ▶ **Advantage of generators over lists: large lists consume lots of memory**
 - Generators don't require all values of sequence to be available immediately
 - Generator uses very little memory
- ▶ **Example: Generator function version of DoubleList class**
 - yield statement “bookmarks” place in function loop
 - When next() is called again, generator function picks up where it left off

```
def double_it(data):  
    for value in data:  
        yield value * 2  
  
values = [10, 20, 30, 40]  
for num in double_it(values):  
    print(num)
```

yield remembers place in loop

Prints 20, 40, 60, 80

- ▶ **Generator functions provide a compact syntax for implementing the Iterator design pattern**

Generator Expressions

- **Generator expression is an even more compact way of writing generators**
 - Syntax is the same as a list comprehension, without '[...]'

generator_expr.py

```
powers = [2, 4, 8, 16, 32]
msg = ', '.join(str(p) for p in powers)

graduates = [('Paul', 3.0),
              ('George', 2.0),
              ('Ringo', 1.0),
              ('John', 4.0)] # student name, grade

top = max((grad[1], grad[0]) for grad in graduates)
```

Generator expression.
Same as: for p in powers:
yield str(p)

3. top gets
(4.0, 'John')

2. max() compares
tuples element by
element

1. Generator expression
yields tuples

Contents

- ▶ The Template Method Pattern
- ▶ Defining and Using Abstract Base Classes
- ▶ The Iterator Pattern
- ▶ Generator Functions and Expressions

Hands-On Exercise 2.1



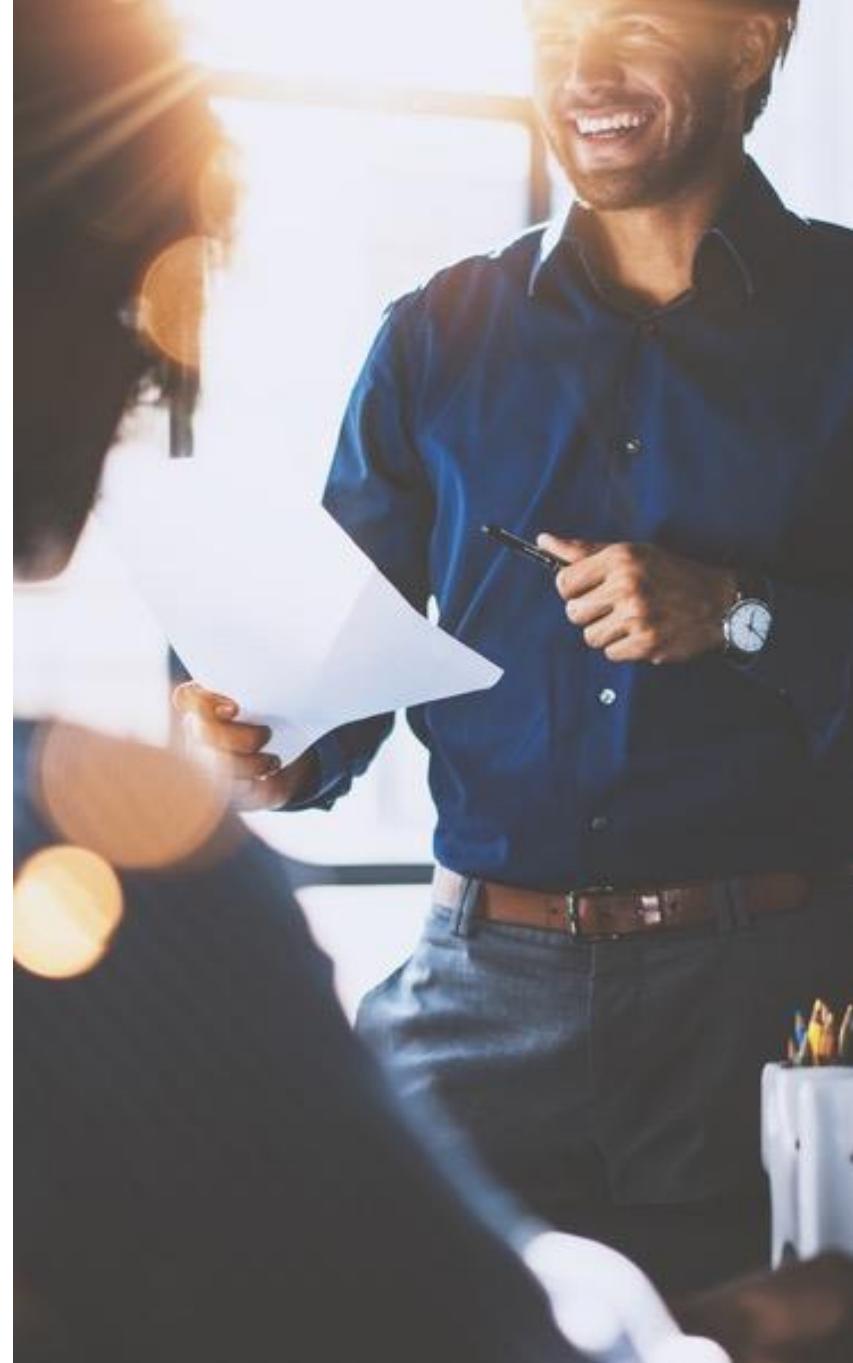
Hands-On Exercise 2.1

In your Exercise Manual, please refer to
**Hands-On Exercise 2.1: Implementing Design
Patterns in Python**



Objectives

- ▶ Define a skeleton algorithm in the Template Method design pattern
- ▶ Loop over a collection with the Iterator design pattern
- ▶ Process collections efficiently with generator functions and generator expressions



Chapter 3

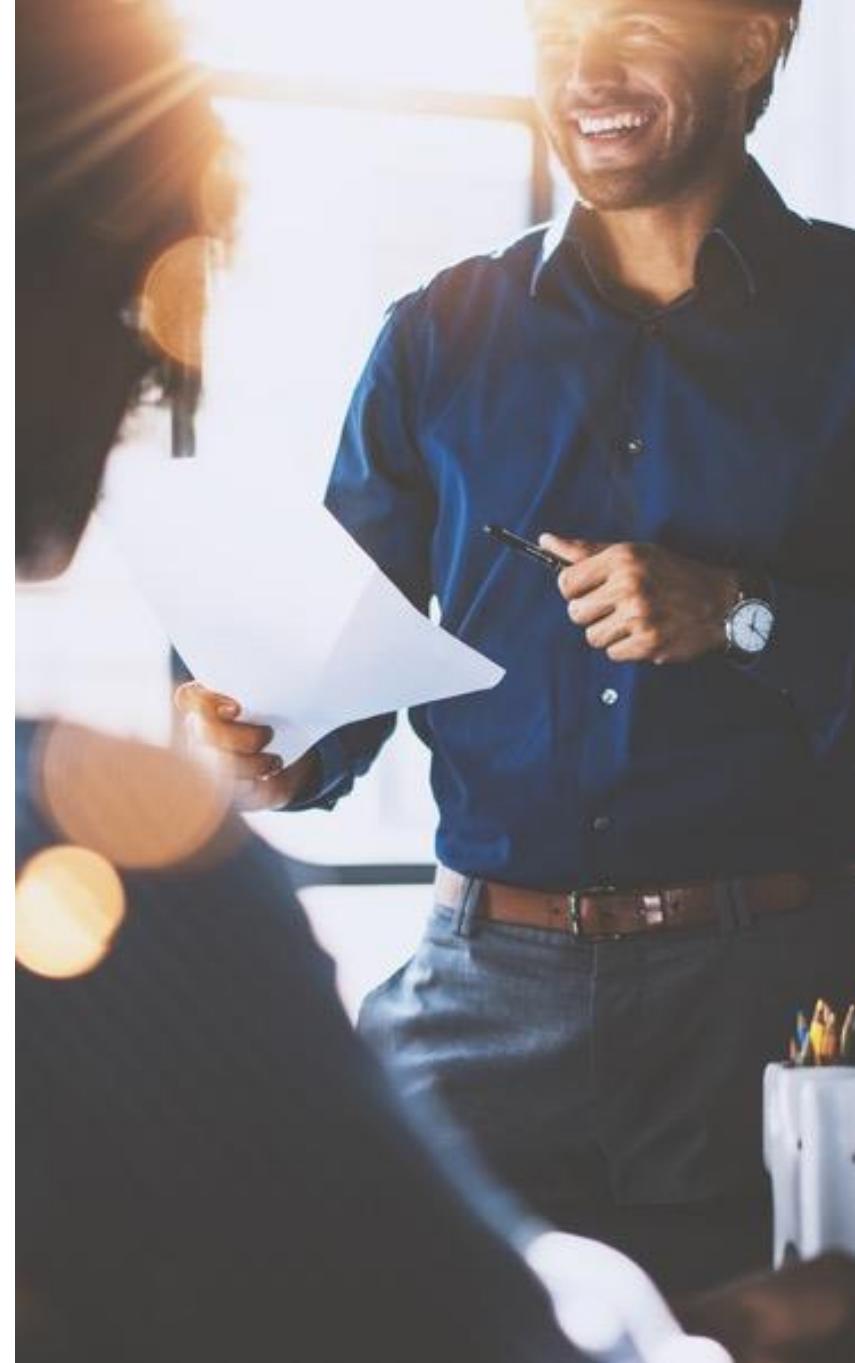
Unit Testing and Mocking



LEARNING TREE
INTERNATIONAL

Objectives

- ▶ **Describe the differences between unit testing, integration testing, and functional testing**
- ▶ **Write and run unit tests for Python modules and classes**
- ▶ **Apply best practices in your test setup and execution**
- ▶ **Simplify automated testing with the Pytest module**
- ▶ **Mock dependent objects with the Mock package**



Contents

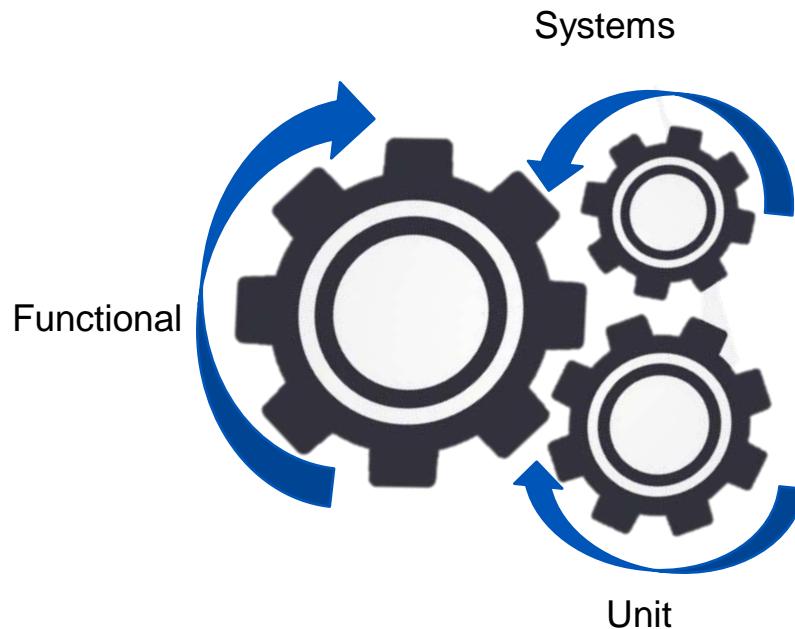
Principles of Testing

- ▶ Writing Unit Tests in Python
- ▶ Executing Unit Tests With Pytest
- ▶ Hands-On Exercise 3.1
- ▶ Using Mock Objects in Unit Tests
- ▶ Optional Hands-On Exercise 3.2



Testing Principles

- ▶ **Good software is thoroughly tested**
 - Requires planning up front in development cycle
 - Requires commitment from developers and project managers
- ▶ **Good code is testable**
 - Designed and written with testing in mind
- ▶ **Dedicated test organizations write and run certain tests**
 - QA/QC: System test, acceptance test
 - Performance group: Load test
 - Security team: Security test
- ▶ **Software developers may write three types of tests**
 - Unit tests: test one component
 - Integration tests: test interaction of several components
 - Functional tests: test full application



QA/QC = quality assurance/quality control

Unit Testing

- ▶ ***Unit test: Tests one component in complete isolation***
- ▶ **Dependencies on other components are provided by test harness**
 - Stub and mock objects, in-memory databases, fake HTTP servers
- ▶ **If a unit test fails, you know exactly which component caused the error**
- ▶ **Unit tests may be written before the component is written**
 - Test-Driven Development (TDD)
 - Unit test defines component's functional requirements
- ▶ **Unit tests are usually automated**
 - Often a task in the nightly build or Continuous Integration (CI) process
- ▶ **Python provides unit testing tools**
 - Standard unittest module
 - mock module (includes Mock and MagicMock classes)
 - Pytest testing framework

Contents

- ▶ Principles of Testing

Writing Unit Tests in Python

- ▶ Executing Unit Tests With Pytest
- ▶ Hands-On Exercise 3.1
- ▶ Using Mock Objects in Unit Tests
- ▶ Optional Hands-On Exercise 3.2



Example: Person Class

- We'll write unit test cases for the Person class

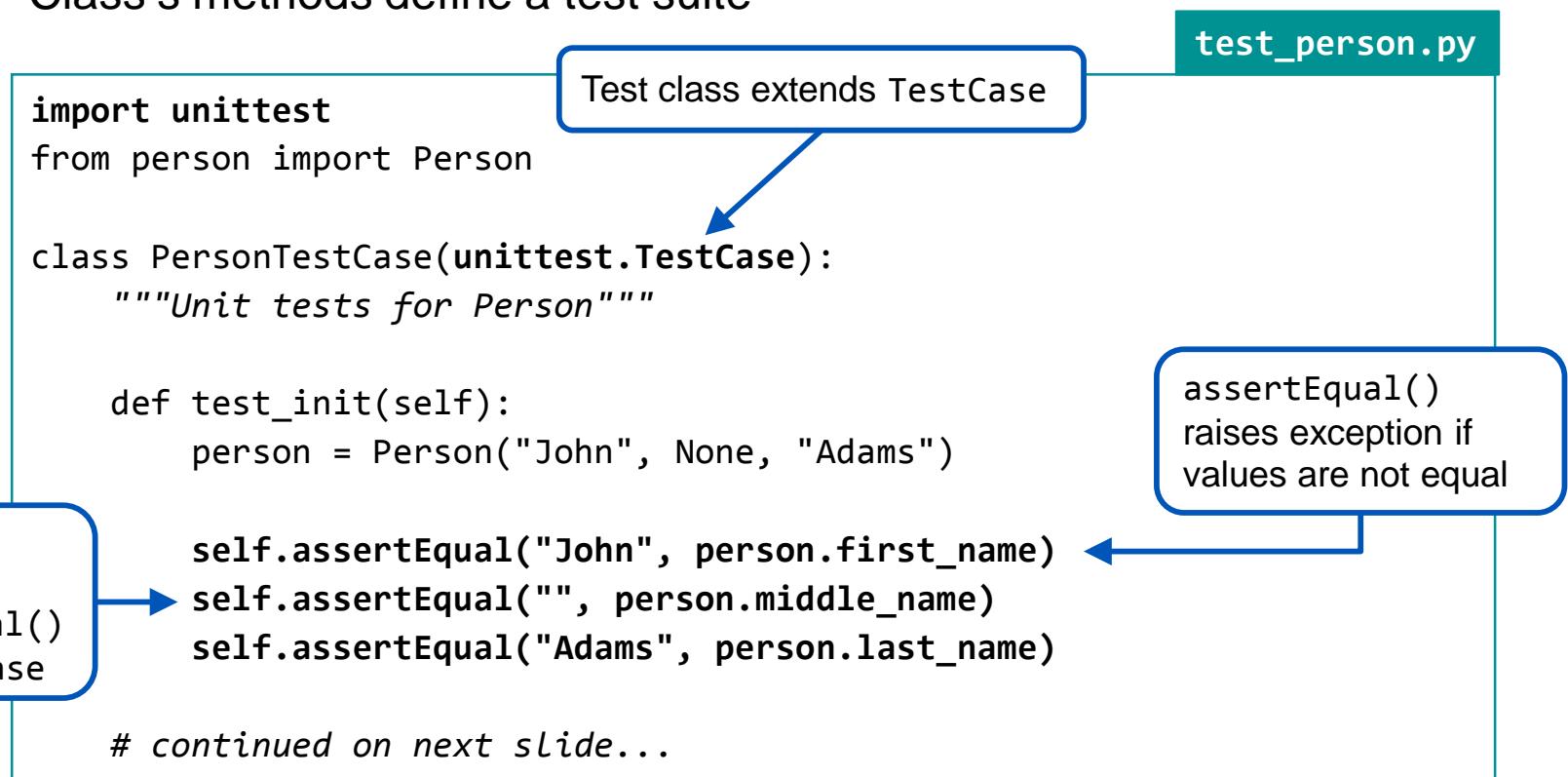
person.py

```
class Person:  
    """ Simple class for a unit test demo """  
  
    def __init__(self, first_name, middle_name, last_name):  
        """ Initialize attributes. Replace None with empty string. """  
        self.first_name = first_name if first_name else ""  
        self.middle_name = middle_name if middle_name else ""  
        self.last_name = last_name if last_name else ""  
  
    def full_name(self):  
        """  
        Join the name attributes with exactly one space between them  
        and no leading or trailing spaces.  
        """  
  
        names = [self.first_name, self.middle_name, self.last_name]  
        return " ".join(n for n in names if n)
```

Writing Unit Tests

► Standard unittest module supports automated unit tests

- Define a class that extends unittest.TestCase
- Define methods whose names begin with test
 - Each method is a test case
 - Class's methods define a test suite



Writing Unit Tests

- Your unit test module calls `unittest.main()`
 - Launches test runner
- Test runner reports failing test cases
 - If the test condition of an `assert...()` method is false
 - If the code being tested raises an exception that's never caught

```
# ... continued from previous slide

def test_full_name(self):
    person = Person("John", "Quincy", "Adams")
    full_name = person.full_name()
    self.assertEqual("John Quincy Adams", full_name)

def test_full_name_empty_middle(self):
    person = Person("John", "", "Adams")
    full_name = person.full_name()
    self.assertEqual("John Adams", full_name)

if __name__ == '__main__':
    unittest.main()
```

The diagram illustrates the execution flow of the provided Python code. It consists of several annotated sections:

- Initialize object to be tested:** Points to the line `person = Person("John", "Quincy", "Adams")`.
- Call method being tested:** Points to the line `full_name = person.full_name()`.
- Verify results are correct:** Points to the line `self.assertEqual("John Quincy Adams", full_name)`.
- main() starts test runner:** Points to the final line `unittest.main()`.

Running Unit Tests

- ▶ Test runner executes test cases and reports results

```
> cd \crs1906\examples\ch03_examples  
  
> python -m unittest test_person.py  
.....  
-----  
Ran 10 tests in 0.002s  
  
OK
```

Each "." represents a successful test case

Running Unit Tests

► Test runner reports failures

```
class Person:  
    def __init__(self, first_name, middle_name, last_name):  
        self.first_name = first_name  
        self.middle_name = middle_name  
        self.last_name = middle_name
```

Bug in code!

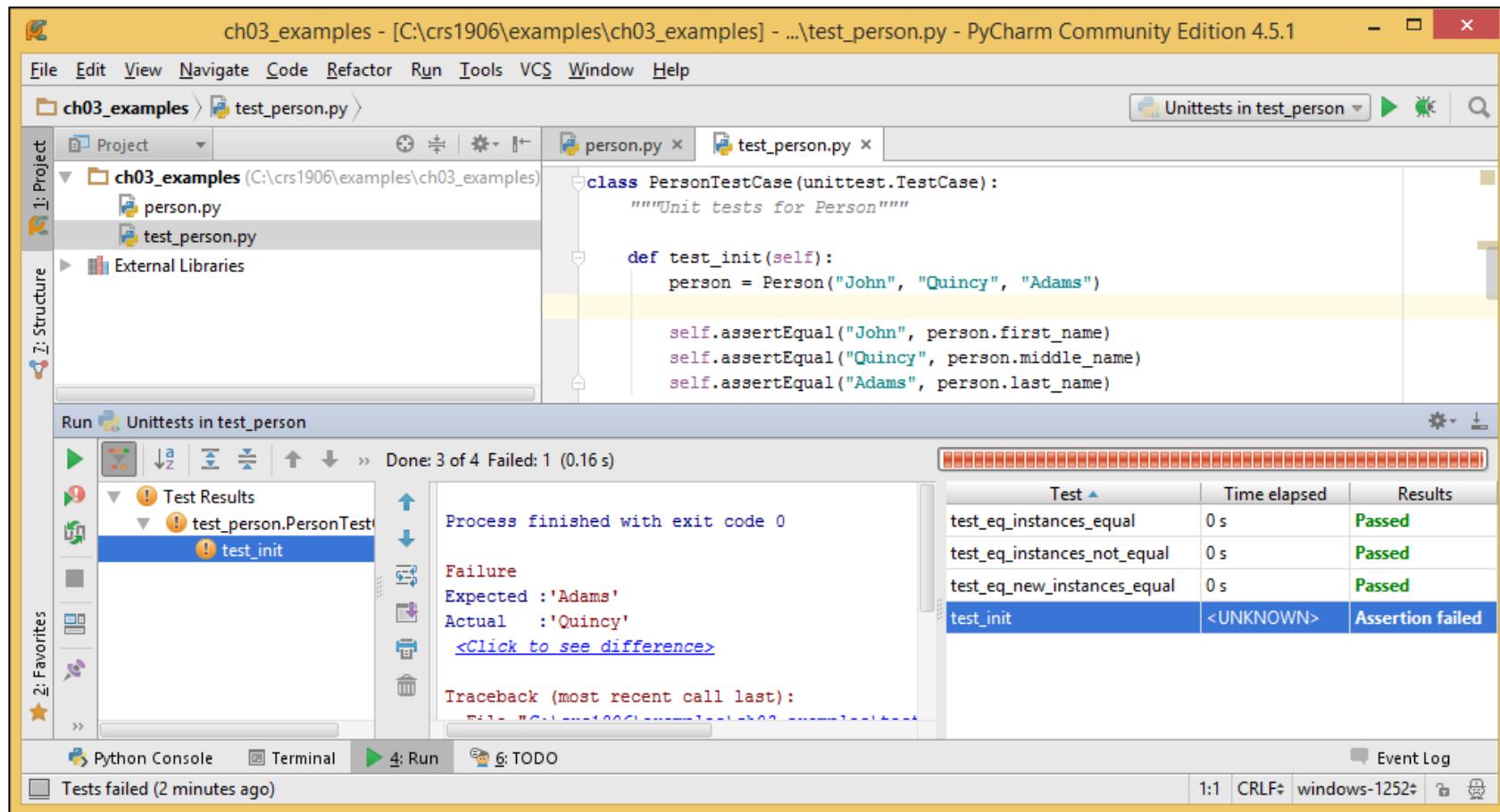
Each "F"
represents a failing
test case

```
> python -m unittest test_person_fail.py  
.F...  
=====  
FAIL: test_empty_middle (test_person_fail.PersonTestCase)  
-----  
Traceback (most recent call last):  
  File "...\\test_person_fail.py", line 34, in test_empty_middle  
    self.assertEqual("John Adams", full_name)  
AssertionError: 'John Adams' != 'John'  
  - John Adams  
  + John  
-----  

```

Running Unit Tests in PyCharm

- ▶ PyCharm has a built-in test runner
 - Right-click test module in Project window | Run 'Unittests in test_person'



Other Assert Methods

- ▶ **TestCase defines many assert methods**
- ▶ **Examples are:**
 - **assertTrue(expr): Verifies bool(expr) is True**

```
self.assertTrue(len(input_list) > 1)
```

- **assertIsNotNone(expr): Verifies expr is not None**

```
person = Person('William', 'Shakespeare')
self.assertIsNotNone(person.last_name)
self.assertTrue(person.last_name.strip())
```

- **assertRegex(str, regex): Verifies str matches regex**

```
self.assertRegex(address.us_zipcode, r'^\d{5}(-\d{4})?$',
```

Testing for Exceptions

- ▶ **TestCase methods can verify that exceptions are raised when appropriate**
 - `assertRaises(exc_type)`: Verify exception of type `exc_type` is raised
 - `assertRaisesRegex(exc_type, regex)`: Verify exception's string value matches `regex`
- ▶ **assertRaises returns a context manager so it can be used on with block**
- ▶ **Example: Person constructor should raise ValueError on bad input**

```
class Person:  
    def __init__(self, first_name, last_name):  
        if not last_name:  
            raise ValueError('Last name cannot be empty')  
        ...  
class PersonTest:  
    def test_verify_exception():  
        with self.assertRaises(ValueError): ← assertRaises() returns a  
            person = Person('William', None)  
  
    def test_verify_exception_message():  
        with self.assertRaisesRegex(ValueError, r'[Ll]ast.*[Nn]ame'):  
            person = Person('William', None)
```

assertRaises() returns a
context manager

Review: Context Managers

- ▶ Many standard functions perform implicit operations in a `with` statement
 - Function returns a *context manager*
 - Context manager performs actions on entering and exiting the `with` block
- ▶ Often used for implicit cleanup operations
 - No need to explicitly call cleanup methods
- ▶ Example: Reading a file

Using a `with` Statement

```
with open('index.html') as f:  
    for line in f:  
        print(line, end="")
```

Using explicit cleanup operation

```
try:  
    f = open('index.html')  
    for line in f:  
        print(line, end="")  
finally:  
    try:  
        f.close()  
    except NameError:
```

Must close file explicitly

If `open()` fails, reference to
`f` raises `NameError`



The file context manager always closes the file, even if an exception is raised

Comparing Entire Objects

- You can often simplify unit tests by comparing entire objects
 - Instead of comparing all attributes individually

```
def test_instances_equal(self):  
    expected = Person("John", "Quincy", "Adams")  
    actual = personRepository.find(last_name="Adams")  
    self.assertEqual(expected, actual)  
  
def test_instances_not_equal(self):  
    expected = Person("John", "Quincy", "Adams")  
    actual = personRepository.find(last_name="Lincoln")  
    self.assertNotEqual(expected, actual)
```

Compare entire objects instead of comparing first name and middle name and last name

Verify objects are not equal

- To compare two objects directly, the class needs an `__eq__` method*

*Note: There are two underscores before and after "eq".

The `__eq__` Method

► Define the `__eq__` method to compare two objects

- `__eq__` is known as a “magic” method
- When you compare two objects with `==`, Python “magically” calls `__eq__`

```
class Person:  
    ...  
    def __eq__(self, other):  
        """Called when Persons are compared using the == operator"""  
        return isinstance(other, Person) and \  
            other.first_name == self.first_name and \  
            other.middle_name == self.middle_name and \  
            other.last_name == self.last_name  
  
john = Person("John", "Quincy", "Adams")  
adams = Person("John", "Quincy", "Adams")  
if john == adams: # True
```

Person's `__eq__` method defines what "equal" means for two Person objects

► Without `__eq__`, an object is equal only to itself

- By default, the objects' data attributes are *not* compared

► `unittest.assertEqual(arg1, arg2)` calls arg1's `__eq__`

Comparing Lists

- ▶ Python supports list comparison using `==` and `!=`
 - Lists are equal if they have same length and all items compare equal

```
expected = [...]
actual = [...]
if expected == actual: # compare lists using ==
```

- ▶ Unit tests can easily check values that are lists

```
def no_middle_names(*args: Person) -> List[Person]:
    return [p for p in args
            if not p.middle_name]
```

...
def test_no_middle_names(self):
 p1 = Person('Pat', '', 'Drie')
 p2 = Person('Jesse', '', 'Lee')

 expected = [p1, p2]
 actual = no_middle_names(p1, p2)
 self.assertEqual(expected, actual)

args is a sequence of Person

Function being tested returns a list

Set up expected return value

Call function and save actual return value

Compare expected value to actual value

Contents

- ▶ Principles of Testing
- ▶ Writing Unit Tests in Python

Executing Unit Tests With Pytest

- ▶ Hands-On Exercise 3.1
- ▶ Using Mock Objects in Unit Tests
- ▶ Optional Hands-On Exercise 3.2



The Pytest Framework

- ▶ **Pytest is a third-party unit test framework**
- ▶ **Makes writing test cases simpler**
 - Test cases don't have to be subclasses of `unittest.TestCase`
 - Syntax for writing tests is simpler
- ▶ **Makes running test cases simpler**
 - Automatically searches directories for unit tests
 - Supports flexible test results reporting
- ▶ **Has many useful plug-ins**
 - Code coverage: Determines which application code was actually tested
 - “Picked” tests: Run tests only for modules you changed since last commit
- ▶ **Install Pytest and plugins from PyPI**
 - `pip install pytest pytest-cov pytest-picked`

Writing Tests for Pytest

► Test cases can be simpler than tests written with unittest module

- Test functions don't have to be defined in TestCase subclass
- You can verify behavior with Python's assert statement
- Tests can use standard Python operators ==, !=, etc.

```
def test_init():
    person = Person("John", "Quincy", "Adams")
    assert ("John", "Quincy", "Adams") == \
        (person.first_name, person.middle_name, person.last_name)

def test_eq_instances_equal():
    p1 = Person("John", "Quincy", "Adams")
    p2 = Person("John", "Quincy", "Adams")
    assert p1 == p2
```

Test cases are defined as plain functions

Compare values with ==

If test is False, assert statement raises AssertionError

test_person_pytest.py

- No call to unittest.main()

Running Tests With pytest

► Pytest includes pytest script

- Runs test cases from one file or multiple files

```
> cd \crs1906\examples\ch03_examples
> pytest test_person_pytest.py
.....
collected 4 items
test_person_pytest.py .... [100%]
=====
4 passed in 0.03 seconds =====
```

Run tests in one file

```
> pytest -v test_person_pytest.py
test_person_pytest.py::test_init PASSED [ 25%]
test_person_pytest.py::test_eq_instances_equal PASSED [ 50%]
test_person_pytest.py::test_eq_instances_not_equal PASSED [ 75%]
test_person_pytest.py::test_eq_new_instances_equal PASSED [100%]

=====
4 passed in 0.03 seconds =====
```

Run with verbose output

Reporting Test Failures

► Pytest displays results of failing test cases

- For more detail about test failure, run pytest -v

```
> pytest test_person_pytest_fail.py
test_person_pytest_fail.py F... [100%]
=====
      FAILURES =====
      test_init
=====
def test_init():
    person = Person("John", "Quincy", "Adams")
>     assert ("John", "Quincy", "Adams") == \
           (person.first_name, person.middle_name, person.last_name)
E     AssertionError: assert ('John', 'Quincy', 'Adams') ==
                           ('John', 'Quincy', 'Quincy')
E         At index 2 diff: 'Adams' != 'Quincy'
E         Use -v to get the full diff

test_person_pytest_fail.py:17: AssertionError
=====
  1 failed, 3 passed in 0.07 seconds =====
```

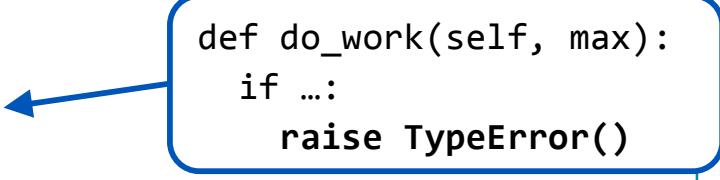
Pytest displays complete assert statement

Testing for Exceptions

- ▶ Pytest lets you verify that exceptions are raised appropriately
- ▶ `raises(exception-type)`
 - Test passes if method call raises the specified exception
 - Test fails if no exception or a different type of exception

```
from pytest import raises

def test_exception_if_wrong_type():
    with raises(TypeError):
        obj_under_test.do_work("123")
```

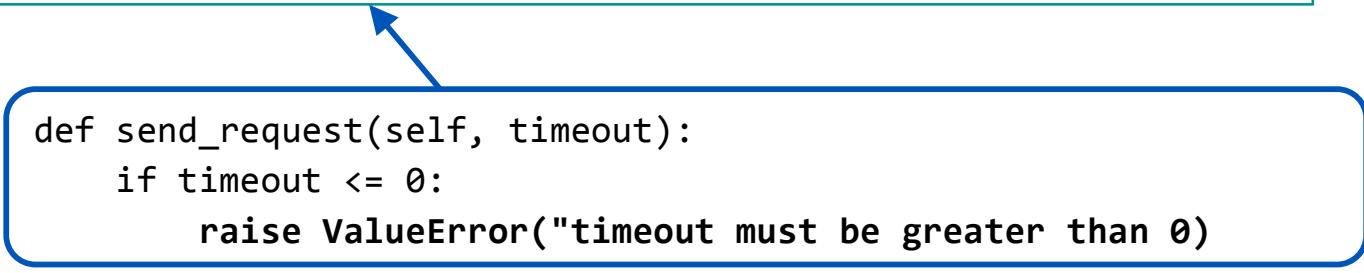


```
def do_work(self, max):
    if ...:
        raise TypeError()
```

A blue arrow points from the line "obj_under_test.do_work("123")" in the test code to the start of the "do_work" method definition.

- ▶ Add `match` argument to `raises()` to test exception's message

```
def test_negative_timeout_raises_ValueError():
    with raises(ValueError, match=r"[Gg]reater than (zero|0)"):
        obj_under_test.send_request(-1)
```



```
def send_request(self, timeout):
    if timeout <= 0:
        raise ValueError("timeout must be greater than 0")
```

A blue arrow points from the line "obj_under_test.send_request(-1)" in the test code to the start of the "send_request" method definition.

Reporting Code Coverage

- ▶ All your test cases pass—Great!
- ▶ But did you test all your code?
 - Maybe your tests execute only 50% of your code
 - What about the code you didn't test?
- ▶ How do you know which code wasn't tested?
- ▶ Don't guess—Analyze your test case *code coverage*



Reporting Code Coverage

- ▶ Pytest can determine what percentage of code was tested by unit tests
 - Runs all tests in the current directory: `pytest --cov --cov-report html`
 - Creates `htmlcov` directory with HTML coverage report

```
> pytest --cov --cov-report html
```

Coverage report shows which code was executed by tests

```
Coverage for person.py : 91%
11 statements 10 run 1 missing 0 excluded

1 """
2 person.py - Simple Person class for Chapter 3 examples.
3 """
4
5
6 class Person:
7     """Simple class for unit test demo"""
8
9     def __init__(self, first_name, middle_name, last_name):
10         self.first_name = first_name
11         self.middle_name = middle_name
12         self.last_name = last_name
13
14     def __eq__(self, other):
15         """Called when Person instances are compared with == operator"""
16         return isinstance(other, Person) and \
17             other.first_name == self.first_name and \
18             other.middle_name == self.middle_name and \
19             other.last_name == self.last_name
20
21     def __ne__(self, other):
22         """Called when Person instances are compared with != operator"""
23         return not self.__eq__(other)
24
25     def __str__(self):
26         return f"{self.first_name} {self.middle_name} {self.last_name}"
```

Test Discovery

- ▶ Pytest will recursively search directories and run all tests
 - Test file name must match the pattern *_test.py or test_*.py
 - Name of test class (if defined) must start with Test
 - Test function and method names must start with test
 - Examples
 - Function `test_view_success` in `view_test.py`
 - Method `test_mobile` in class `TestView` in `test_view.py`

```
> cd \crs1906\exercises\ticketmanor_webapp
> pytest tests -W ignore::DeprecationWarning
collected 94 items

tests\models\test_act.py . [ 1%]
tests\models\test_address.py .... [ 5%]
tests\models\test_customer.py ..... [ 10%]
...
tests\util\test_util.py ... [ 97%]
tests\views\test_login_view.py .. [100%]
===== 94 passed, 3 warnings in 14.98s =====
```

tests is the top-level directory of test cases

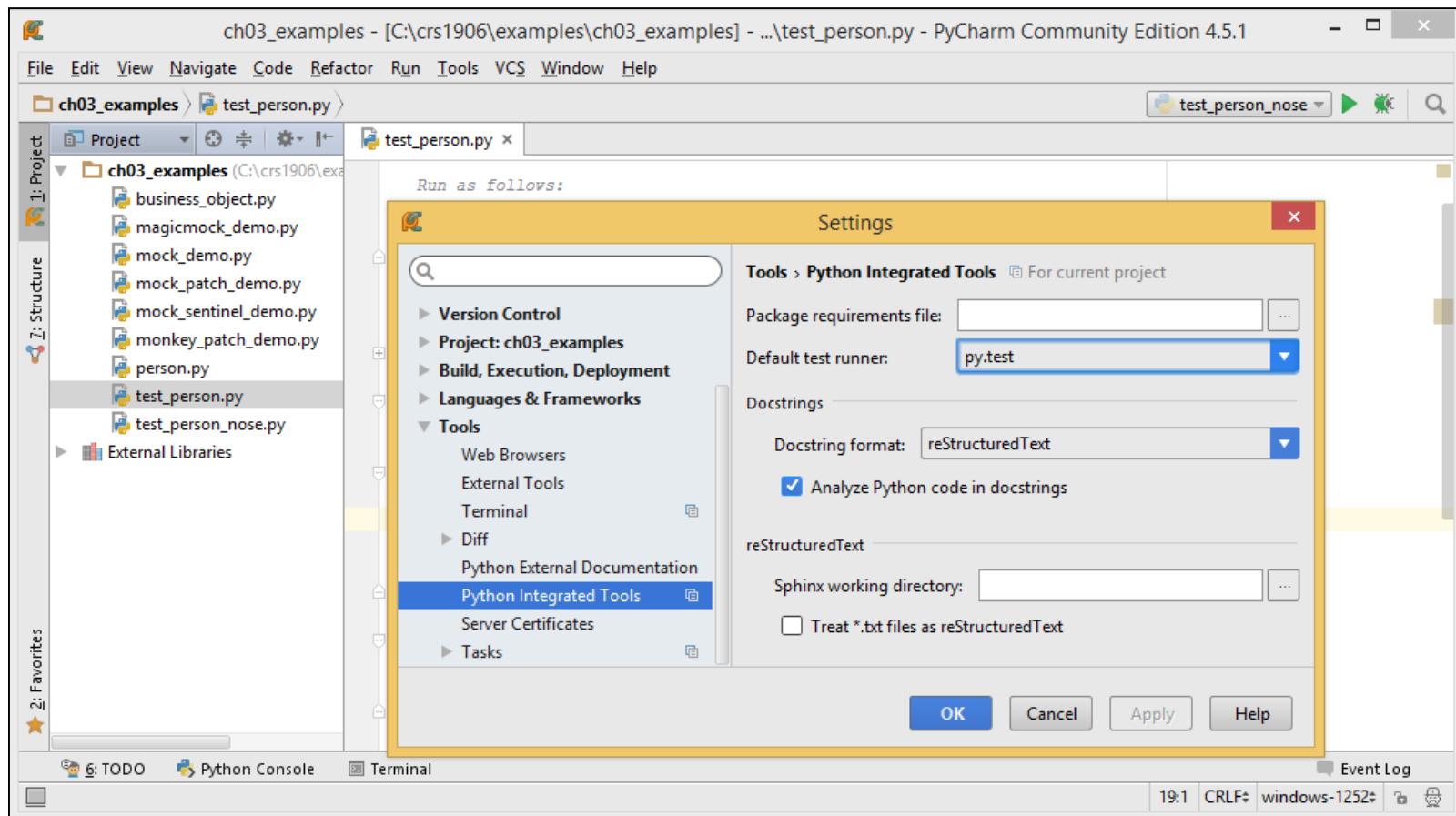
Pytest discovers and runs all tests in all subdirectories of tests

The Nose Framework

- ▶ **Nose is another third-party unit test framework**
 - Nose and Pytest modules were developed from a common code base
 - Nose has same advantages as Pytest
- ▶ **Nose is no longer under active development**
 - Replaced by Nose2 module, but Nose2 has not gained wide acceptance
 - Most projects are switching to Pytest

Using Pytest or Nose for Unit Tests in PyCharm

- ▶ PyCharm can use Pytest or Nose to run unit tests
 - Select File | Settings | Tools | Python Integrated Tools | Default test runner



Contents

- ▶ Principles of Testing
- ▶ Writing Unit Tests in Python
- ▶ Executing Unit Tests With Pytest

Hands-On Exercise 3.1

- ▶ Using Mock Objects in Unit Tests
- ▶ Optional Hands-On Exercise 3.2



Hands-On Exercise 3.1

In your Exercise Manual, please refer to
Hands-On Exercise 3.1: Unit Testing



Contents

- ▶ Principles of Testing
- ▶ Writing Unit Tests in Python
- ▶ Executing Unit Tests With Pytest
- ▶ Hands-On Exercise 3.1

Using Mock Objects in Unit Tests

- ▶ Optional Hands-On Exercise 3.2



Testing Objects With Dependencies

- Most classes depend on other classes to do some of their work
 - Example: BusinessObject delegates database access to UserDao
 - UserDao implements Data Access Object (DAO) design pattern

ch03_examples/business_object.py

```
class UserDao: # encapsulates database access
    def __init__(self): ... # create connection to database

    def query_user(self, user_id): ... # query database

class BusinessObject:
    def __init__(self):
        self.user_dao = UserDao()

    def get_user(self, user_id)
        try:
            user = self.user_dao.query_user(user_id)
            if user is None:
                raise ValueError('invalid ID')
            return user
        except sqlite3.Error:
            raise BusinessError('Problem fetching user')

BusinessObject constructor satisfies dependency
BusinessObject uses dependency to access DB
```

Mock Objects

- ▶ **Problem: BusinessObject has hardcoded dependency on UserDao**

```
class BusinessObject:  
    def __init__(self):  
        self.user_dao = UserDao()
```

- ▶ **Unit tests should test classes in complete isolation**

- But creating a BusinessObject also creates a UserDao
 - So unit tests of BusinessObject also test UserDao

- ▶ **Problem: UserDao may need connection to production database**

- ▶ **Solution: In unit tests, replace UserDao instance with a *mock object***

- Mock object will have the same interface as UserDao
 - But mock DAO's methods return static values
 - Mock DAO doesn't need a database connection
 - Mock objects can verify that their methods were called correctly

- ▶ **Standard module unittest.mock makes it easy to define mock objects**

- Added in Python 3.3; available for earlier Python versions as mock module



Review: Monkey Patching

- ▶ **unittest.mock utilizes *monkey patching***
 - Monkey patch: Piece of Python code that modifies other code at runtime
- ▶ **Use cases for monkey patching**
 - Unit tests: Replace reference to dependent object or replace method with stub
 - Production code: Patch third-party code as a workaround to a bug
- ▶ **Example of monkey patching—you can use monkey patching anywhere**
 - This is not a unit test

monkey_patch_demo.py

```
class SimpleCounter: # __init__() definition omitted...
    def increment(self, incr=1):
        self.count += incr
    def debug_incr(obj, incr=1):
        obj.count += incr
        print('new value =', obj.count)
SimpleCounter.increment = debug_incr
counter = SimpleCounter()
counter.increment()
```

The diagram illustrates the flow of monkey patching in the provided Python code. It consists of three callout boxes with arrows pointing to specific parts of the code:

- A blue arrow points from the text "Our monkey patch function" to the line `def debug_incr(obj, incr=1):`.
- A blue arrow points from the text "Replace old method with new method" to the line `SimpleCounter.increment = debug_incr`.
- A blue arrow points from the text "Call to increment calls new function with counter as first arg" to the line `counter.increment()`.

Using Mock Objects in Unit Tests

- ▶ **Goal: test BusinessObject.get_user()**
 - get_user() calls UserDao.query_user()
- ▶ **Unit test replaces the production UserDao with a mock object**
 - We add a query_user() method to the mock
 - We tell the Mock query_user() what to return

mock_demo.py

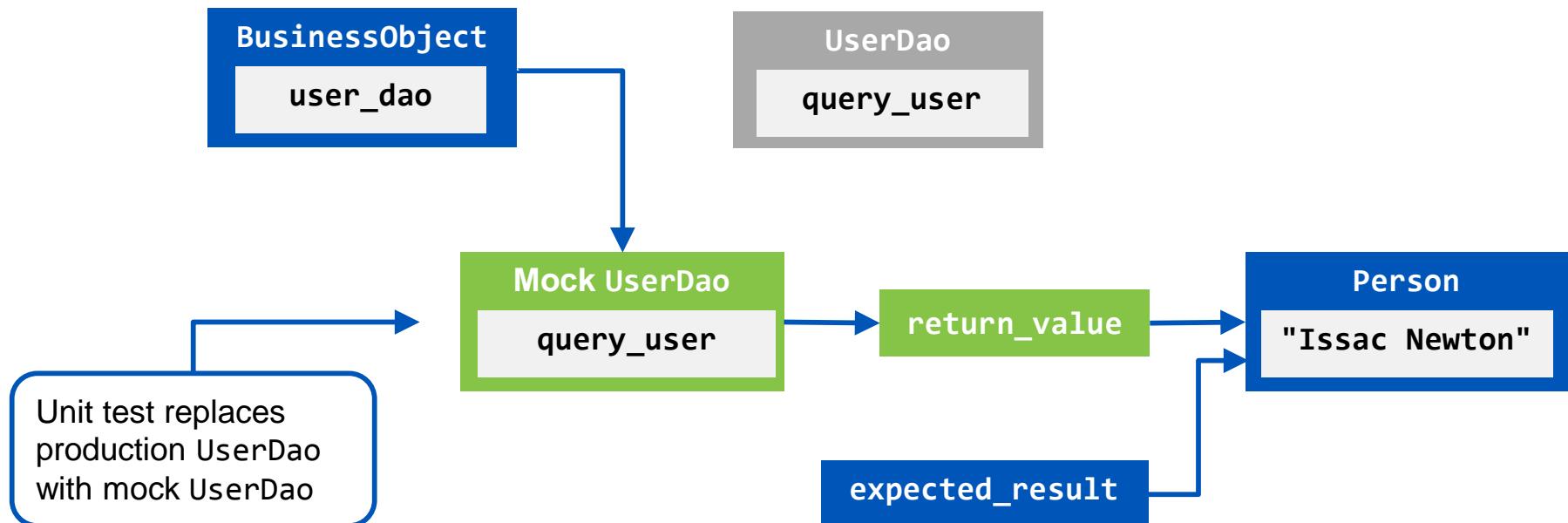
```
class TestBusinessObject(TestCase):  
    def test_get_user(self):  
        expected_result = Person('Isaac', None, 'Newton')  
        mock_dao = Mock()  
        Create mock object  
        mock_dao.query_user.return_value = expected_result  
        Set return value of  
        mock method  
        bus_obj = BusinessObject()  
        bus_obj.user_dao = mock_dao  
        Monkey patch: replace real DAO  
        with mock DAO  
        actual_result = bus_obj.get_user(123)  
        Business method uses mock  
        DAO instead of real DAO  
        self.assertEquals(expected_result, actual_result)  
        Verify actual result equals expected result
```

Using Mock Objects in Unit Tests

► Before using mock



► After replacing UserDao with mock



Using Mock Objects to Trigger Error Conditions

- ▶ **Mock can return values intended to trigger error conditions**
 - Test cases verify that class under test handles errors correctly

```
from pytest import raises

def test_get_user_not_found_raises_ValueError(self):
    mock_dao = Mock()
    mock_dao.query_user.return_value = None
    bus_obj = BusinessObject()
    bus_obj.user_dao = mock_dao

    with raises(ValueError):
        bus_obj.get_user(123)
```

Configure the mock query_user() method to return None

Test case succeeds if business method raises ValueError

```
class BusinessObject:
    def get_user(self, user_id)
        ...
        user = self.user_dao.query_user(user_id)
        if user is None:
            raise ValueError('invalid ID')
        ...
```

Raising Exceptions From Mock Objects

► Mock can raise exceptions

- `side_effect` attribute tells mock which exception to raise
- Test case verifies the class being tested handles exceptions correctly

```
def test_get_user_dao_error_raises_BusinessError(self):
    mock_dao = Mock()
    mock_dao.query_user.side_effect = sqlite3.Error('SQL error')

    bus_obj = BusinessObject()
    bus_obj.user_dao = mock_dao

    with raises(BusinessError, match=r'[Pp]roblem.*user'):
        bus_obj.get_user(123)
```

Configure the mock `query_user()` method to raise a DB error

Test case succeeds if `get_user()` raises `BusinessError` and if `BusinessError`'s message attribute matches the pattern

```
class BusinessObject:
    def get_user(self, user_id)
        try:
            user = self.user_dao.query_user(user_id)
            ...
        except sqlite3.Error:
            raise BusinessError('Problem fetching user')
```

Specifying the Mock Object's Interface

- ▶ By default, Mock allows any method call, even if not defined in UserDao
- ▶ To restrict method calls, add the spec constructor argument
 - `Mock(spec=UserDao)`
 - Mock has the same interface as UserDao
 - Will raise an error if code being tested calls a method not defined in UserDao

```
class TestBusinessObject(TestCase):  
    def test_get_user(self):  
        ...  
        mock_dao = Mock(spec=UserDao)  
        ...
```

Mock raises AttributeError if called
with method not defined in UserDao



Contents

- ▶ Principles of Testing
- ▶ Writing Unit Tests in Python
- ▶ Executing Unit Tests With Pytest
- ▶ Hands-On Exercise 3.1
- ▶ Using Mock Objects in Unit Tests

Optional Hands-On Exercise 3.2



Optional Hands-On Exercise 3.2

In your Exercise Manual, please refer to
**Optional Hands-On Exercise 3.2: Unit Testing
With Mocks**

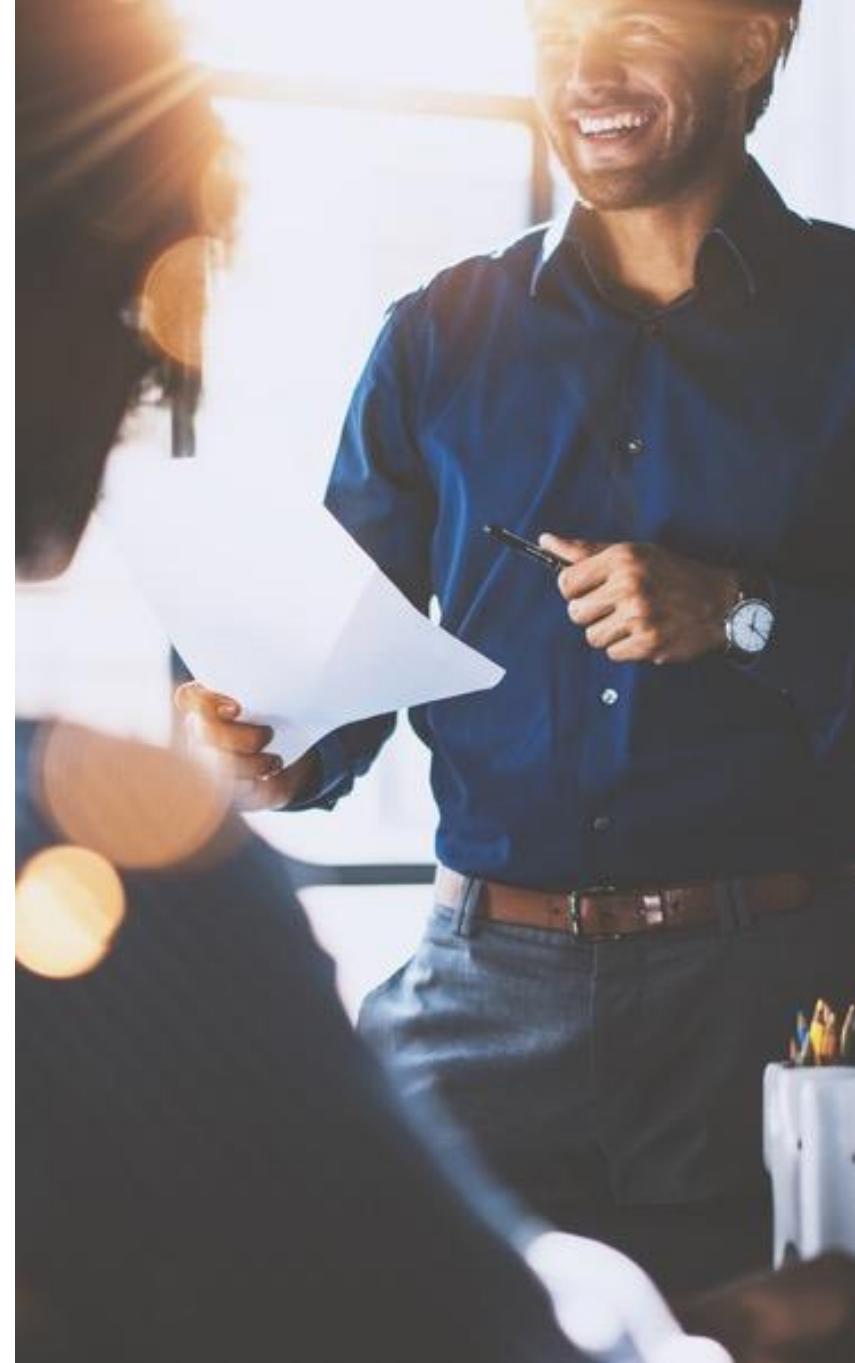


Best Practices for Testing

- ▶ **Write automated unit tests for all new code**
 - Ideally, before you write the code itself
- ▶ **Use mock objects to satisfy dependencies between classes**
 - But use verification of mocks sparingly to avoid brittle test code
- ▶ **Run unit tests every time you change the code**
- ▶ **Include automated unit tests as part of your build process**
 - Include test results in project's Definition of Done (DoD)
 - Examples: "All unit tests must pass," "98% of unit tests must pass"
 - Include required test coverage in DoD
 - Example: "At least 95% of code must be covered by unit tests"
- ▶ **Strategy for fixing bugs**
 1. Write a unit test that reproduces the bug before attempting to fix it
 2. Run the unit test and verify that it fails
 3. Fix the bug
 4. Run the unit test again, and verify that it succeeds

Objectives

- ▶ **Describe the differences between unit testing, integration testing, and functional testing**
- ▶ **Write and run unit tests for Python modules and classes**
- ▶ **Apply best practices in your test setup and execution**
- ▶ **Simplify automated testing with the Pytest module**
- ▶ **Mock dependent objects with the Mock package**

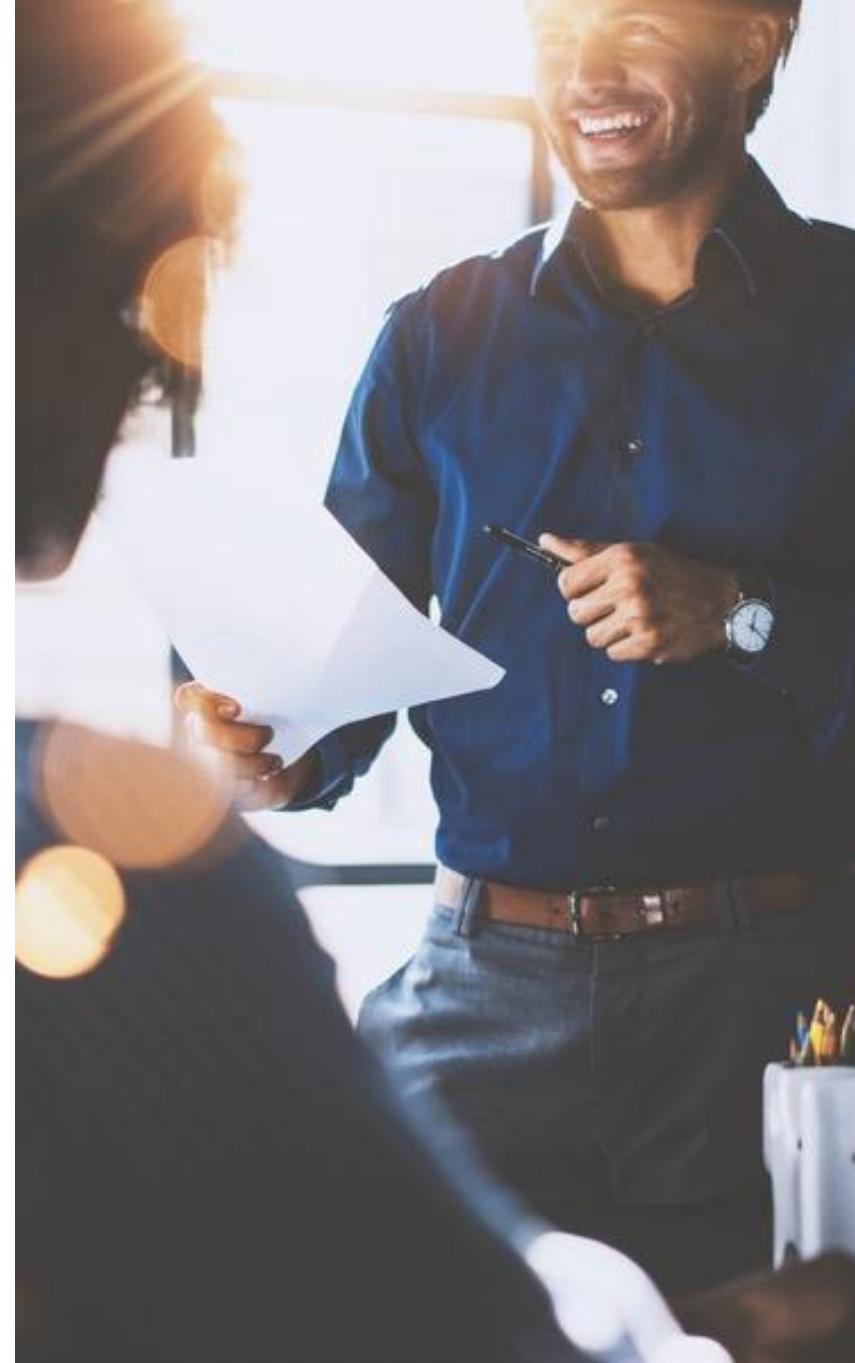


Chapter 4

Error Detection and Debugging Techniques

Objectives

- ▶ Raise user-defined exceptions
- ▶ Log messages from Python modules for auditing and debugging
- ▶ Verify your code with Pylint
- ▶ Trace program execution with the PyCharm IDE



Contents

Defining and Raising Exceptions

- ▶ Logging Messages
- ▶ Verifying Code With Pylint
- ▶ Tracing Execution With IDEs
- ▶ Hands-On Exercise 4.1



Review: Exception Handling

- ▶ Python's exception mechanism supports error notification and handling
 - When code detects an error, it *raises* an exception
 - By default, Python interpreter prints a *traceback* (list of active functions)

```
>>> elements = {'H': 'hydrogen', 'He': 'Helium'}  
>>> elements['Li']  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>  
    KeyError: 'Li'
```

- ▶ Code can handle exceptions with **try** statement

```
try:  
    elements = {'H': 'hydrogen',  
                'He': 'helium'}  
    print(elements['Li'])  
except KeyError as ke:  
    print('Error accessing item', ke)  
finally:  
    print('Done with elements')
```

Output

Error accessing item 'Li'
Done with elements

Executed if KeyError
exception raised in try

Always executed

Getting Information About an Exception

- ▶ **sys.exc_info() returns 3-tuple with details of current exception**
 - Type of exception
 - Exception object itself
 - Traceback object with traceback
- ▶ **Standard traceback module can extract, format, and print tracebacks**
 - To print traceback to sys.std_err, call traceback.print_exc()
 - Other traceback methods let you manipulate stack frames
- ▶ **Example from tkinter GUI application**

```
try:  
    ...  
except:  
    exc_type, exc_value, exc_tb = sys.exc_info()  
    traceback.print_exc()  
    self.errorDialog("Error... {} {}"  
                    .format(exc_type, exc_value))
```

Write traceback to
sys.std_err

Display exception
information in error dialog

User-Defined Exceptions

- ▶ You define new exception classes by extending `Exception`

exceptions.py

```
class TicketException(Exception):  
    pass
```

- ▶ Your code raises new exception when it detects an error condition

- Exception constructor sets `args` attribute to tuple of constructor arguments
- Calling code handles new exception just like built-in exceptions

```
def process_ticket(ticket):  
    if not ticket.date_time:  
        raise TicketException(  
            'No date/time for event',  
            ticket.event_name)  
  
try:  
    t = Ticket('XKCD Con', None, 85.0)  
    process_ticket(t)  
    print('Ticket {} is ok'.format(t))  
except TicketException as te:  
    print(te.args)
```

Raise exception

Output

```
('No date/time for event',  
 'XKCD Con')
```

Print exception's args

Contents

- ▶ Defining and Raising Exceptions

Logging Messages

- ▶ Verifying Code With Pylint
- ▶ Tracing Execution With IDEs
- ▶ Hands-On Exercise 4.1



Logging Error Messages

- ▶ Python's standard logging implements a fast, flexible logging framework
 - Unlike print(), logging methods have many configuration options
 - Messages can be logged at different levels: ERROR, INFO, DEBUG, etc.
 - Logging level can be different for individual Python packages
 - Logging level can be configured in code or from an external configuration file
 - Format of messages can be different for different output destinations
- ▶ Logger class defines logging methods
 - Create Logger instances by calling `logging.getLogger(logger_name)`
 - Use `__name__` to get package-qualified name of current module
 - Call Logger methods to log messages

```
import logging
logger = logging.getLogger(__name__)
...
logger.info('Program started at %s', time.strftime('%X %x'))
total, used, free = shutil.disk_usage('/')
if used / total > 0.9: # disk is more than 90% full
    logger.warning('Root dir is %d% full', 100 * used / total)
```

Substitutions in messages:
%s string
%d numeric

`logging_demo.py`

Logger Attributes

- Logger methods create messages with standard logging levels

Logger Method	Message Level
logger.critical()	CRITICAL
logger.error() logger.exception()	ERROR
logger.warning()	WARNING
logger.info()	INFO
logger.debug()	DEBUG



High Priority
Low Priority

- Message is logged if message level is equal to or higher than logger level
 - Default logger level is WARNING
 - Message destination is determined by Handler class

```
logger = logging.getLogger('logging_demo')
logger.setLevel(logging.INFO) # Log Levels INFO and higher
logger.addHandler(logging.FileHandler('demo.log'))
logger.debug('Debug message') # DEBUG-Level message is not Logged
logger.info('Info message') # INFO-Level message is Logged
```

Logging Demo

Demo

- ▶ Open Python console and execute the following commands

```
>>> import logging
>>> logger = logging.getLogger('logging_demo') ←
>>> logger.addHandler(logging.StreamHandler())
>>> logger.setLevel(logging.INFO)
>>> logger.warning('Warning message')
Warning message
>>> logger.info('Info message')
Info message ←
>>> logger.debug('Debug message')
>>> logger.setLevel(logging.WARNING)
>>> logger.warning('Warning message')
Warning message ←
>>> logger.info('Info message')
>>> logger.debug('Debug message')
>>>
```

By default, StreamHandler logs to sys.stderr

Level is INFO, so debug message is not logged

Level is WARNING, so info and debug messages are not logged

Standard Log Handlers

- ▶ Choose a Handler class to suit your logging requirements

Handler Class	Description
FileHandler	Writes messages to files
RotatingFileHandler	Writes messages to files, with support for maximum log file sizes and log file rotation
TimedRotatingFileHandler	Writes messages to files, rotating log file at certain timed intervals
SMTPHandler	Sends messages to designated email address
HTTPHandler	Sends messages to HTTP server

- ▶ Example: Send a text message by logging to SMTPHandler

```
from logging.handlers import SMTPHandler
smtp_host = 'my.smtp.host.com'
sender = 'python@my.host.com'
to = '5551234567@txt.att.net'
subj = 'Warning from python app'
logger.addHandler(SMTPHandler(smtp_host, sender, to, subj))
logger.warning('Better check the server') # send text message
```

Formatting Log Messages

- **Formatter classes let you choose format of log messages**
 - Format codes in format string are substituted when message is written

```
log_format = '%(asctime)s:%(levelname)s:%(message)s'  
handler = logging.FileHandler('logging_demo.log')  
handler.setFormatter(logging.Formatter(log_format))  
logger.addHandler(handler)
```

Format Code	Description
%(asctime)s	Date and time as a string
%levelname)s	Log level
%message)s	Log message
%funcname)s	Name of function that contains logging call
%name)s	Name of logger
%module)s	Module name
%threadName)s	Thread name

Reducing Logging Overhead

- ▶ **Logger method calls usually have very little overhead**
 - If message level is lower than configured logging level, nothing is written
 - However, creating the message itself may be slow

```
logger.setLevel(logging.INFO) # Log INFO and higher only  
complex_object = MyBigComplexObj()  
logger.debug('object = ' + repr(complex_object))
```

Debug message isn't written,
but `repr()` is still called

- ▶ **To skip creation of messages, test Logger level before calling method**

```
if logger.isEnabledFor(logging.DEBUG):  
    logger.debug('object = ' + repr(complex_object))
```

Call to `repr()` is skipped unless
logging level is DEBUG or lower

File-Based Logging Configuration

- ▶ Logging can be configured in a text file instead of Python code
 - Logging levels can be changed without modifying code
 - Handler types can be changed
- ▶ Read configuration file by calling `logging.config.fileConfig()`

```
import logging
import logging.config
logging.config.fileConfig('logging.conf')
logger = logging.getLogger('logging_demo')
logger.info('Program started at %s', time.strftime('%X %x'))
```

Read logging configuration from file

No code required to configure logging

Logging Configuration File logging.conf

```
[loggers]
keys=root,logging_demo ← Define names of loggers,
                           handlers, and formatters

[handlers]
keys=console_handler,file_handler ←

[formatters]
keys=simple_formatter,file_formatter ←

[logger_root] ← Configure root Logger
level=DEBUG
handlers=console_handler

[logger_logging_demo] ← Configure logging_demo Logger
level=DEBUG
handlers=console_handler,file_handler ← Messages are written to two handlers
qualname=logging_demo
propagate=0

# continued on next slide...
```

Logging Configuration File logging.conf

```
[handler_console_handler]
class=StreamHandler
level=WARNING
formatter=simple_formatter
args=(sys.stdout,)

[handler_file_handler]
class=handlers.RotatingFileHandler
level=DEBUG
formatter=file_formatter
args=('logging_demo.log', 'a', 10000000, 5)
# args: file name, file mode, max bytes, number of backups

[formatter_simple_formatter]
format=%(levelname)s:%(message)s

[formatter_file_formatter]
format=%(asctime)s|%(name)s|%(levelname)s|%(message)s
# An empty datefmt value substitutes ISO8601 format date/times
datefmt=
```

Configure file_handler Handler

RotatingFileHandler limits log file size

Configure file_formatter Formatter

Best Practices: When to Use Logging

- This table describes which tools are best for common tasks

Task	Best Tool
Display console output for ordinary messages	<code>print()</code>
Report events that occur during normal operation	<code>logging.info()</code> or <code>logging.debug()</code>
Issue a warning if there is nothing the client application can do about the situation, but the event should still be noted	<code>logging.warning()</code>
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (for example, error handler in a long-running server process)	<code>logging.error()</code> , <code>logging.exception()</code> , or <code>logging.critical()</code>

Contents

- ▶ Defining and Raising Exceptions
- ▶ Logging Messages

Verifying Code With Pylint

- ▶ Tracing Execution With IDEs
- ▶ Hands-On Exercise 4.1



Tools to Check Your Code

- ▶ **Code that runs with no obvious problems may still contain subtle bugs**
 - Python tools can perform an automated code inspection
- ▶ **Pylint: Checks for errors in Python code**
 - Enforces PEP 008 coding standards
 - Checks for bad “code smells”: Poor coding styling, anti-patterns, etc.
 - Documentation is available at <http://docs.pylint.org>
- ▶ **PyCharm automatically detects potential problems**

Advanced Code Inspection With Pylint

- ▶ **Pylint:** Very strict code inspection tool that “picks the lint” out of code
 - Reports potential problems the Python interpreter ignores
- ▶ **Pylint basic usage:** `pylint [options] module_or_package`
 - Analyzes single module or all modules in package
- ▶ **Example: Analyze one file using default options**

```
> pylint simple_crypt.py
No config file found, using default configuration
***** Module ch04_examples.simple_crypt
C: 14, 0: Exactly one space required around assignment
    encoded=encoded + letters[x]           ← Pylint messages
          ^ (bad-whitespace)
C: 23, 0: Final newline missing (missing-final-newline)
C:  1, 0: Missing module docstring (missing-docstring)
C:  3, 0: Invalid constant name "shift" (invalid-name)
...

```

Global evaluation

Your code has been rated at 6.00/10

← Code evaluation

Pylint Message Types

- ▶ **Default text output**
 - MESSAGE_TYPE: LINE_NUM:[OBJECT:] MESSAGE
- ▶ **Message types**
 - Convention: Programming standard violation
 - Refactor: Bad code smell
 - Informational: Potentially interesting information (doesn't affect final grade)
 - Warning: Python-specific problems
 - Error: Bug in the code
 - Fatal: An error occurred that terminated Pylint processing
- ▶ **Pylint generates many false positive messages**
 - Add --errors-only or -E option to restrict messages to errors only

```
> pylint -E simple_crypt.py
No config file found, using default configuration
```

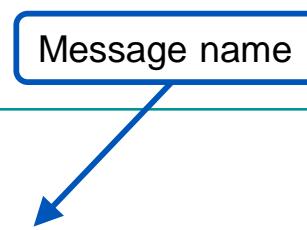
No errors reported

Getting Help for Messages

- ▶ Pylint can give detailed explanation of one of its messages
 - Syntax: `pylint --help-msg=message-name`
- ▶ Example

```
> pylint simple_crypt.py
...
C: 7, 0: Invalid constant name "shift" (invalid-name)
...
> pylint --help-msg=invalid-name
:invalid-name (C0103): *Invalid %s name "%s"%s*
Used when the name doesn't match the regular expression associated
to its type (constant, variable, class...). This message belongs to
the basic checker.
```

Message name



Configuring Pylint

- ▶ You can configure Pylint with configuration file
 - Select default report format
 - Selectively enable or disable messages
- ▶ Pylint looks for its configuration file in the following order
 - `pylintrc` or `.pylintrc` in current directory
 - `pylintrc` in parent modules of current module
 - `.pylintrc` or `.config/pylintrc` in your home directory
- ▶ Pylint can generate template configuration

```
> pylint --generate-rcfile > pylintrc
```
- ▶ For details, see <https://docs.pylint.org/run.html>

Pylint configuration is written to `pylintrc`

Sample pylintrc File

[REPORTS]

```
# Set output format: text, parseable, colorized, msvs, html  
output-format=text
```

```
# Put messages in a separate file for each module / package  
files-output=no
```

```
# Tells whether to display a full report or only the messages  
reports=no
```

[MESSAGES CONTROL]

```
# Only show warnings with the listed confidence levels.
```

```
# Valid Levels: HIGH, INFERENCE, INFERENCE_FAILURE, UNDEFINED  
confidence=
```

...

```
# many more settings ...
```

Disabling Selected Pylint Messages

- ▶ Pylint output can be extremely verbose
- ▶ May result in “warning fatigue”
 - You see so many warnings, you start to ignore them
- ▶ Selectively disable messages in `pylintrc` to filter out harmless messages
 - Useful if Pylint reports problems in code from third-party libraries
 - To display list of Pylint codes: `pylint --list-msg`

`pylintrc`

```
# Disable the given messages
disable=invalid-name,wrong-spelling-in-comment
```

- ▶ You can disable messages with `--disable` command-line option:

```
> pylint --disable=invalid-name,wrong-spelling-in-comment ...
```
- ▶ You can disable warnings in Python code with special Pylint comments

```
shift = 3 # pylint: disable=invalid-name
```

Error Detection in IDEs

- ▶ PyCharm detects PEP 8 nonconformance, potential bugs, etc.
 - Code inspection can be customized using IDE settings

The screenshot shows the PyCharm interface with the file `simple_crypt.py` open. The code implements a Caesar cipher. A callout box with a blue border and arrow points to the right margin, where several pink vertical lines (warning markers) are visible. Another callout box with a blue border and arrow points to a tooltip for a specific warning.

`import string`

`shift = 3`

`choice = input("would you like to encode or decode?")`

`word = (input("Please enter text"))`

`letters = string.ascii_letters + string.punctuation +`

`encoded = ''`

`if choice == "encode":`

`for letter in word:`

`if letter == ' ':`

`encoded = encoded + ' '`

`else:`

`x = letters.index(letter) + shift`

`encoded = encoded + letters[x]`

`if _____:`

`if letter == ' ':`

`encoded = encoded + ' '`

`else:`

`x = letters.index(letter) - shift`

`encoded = encoded + letters[x]`

`print(encoded)`

Markers flag lines with warnings

Hover cursor over warning indicator to see message

PEP 8: missing whitespace around operator

Contents

- ▶ Defining and Raising Exceptions
- ▶ Logging Messages
- ▶ Verifying Code With Pylint

Tracing Execution With IDEs

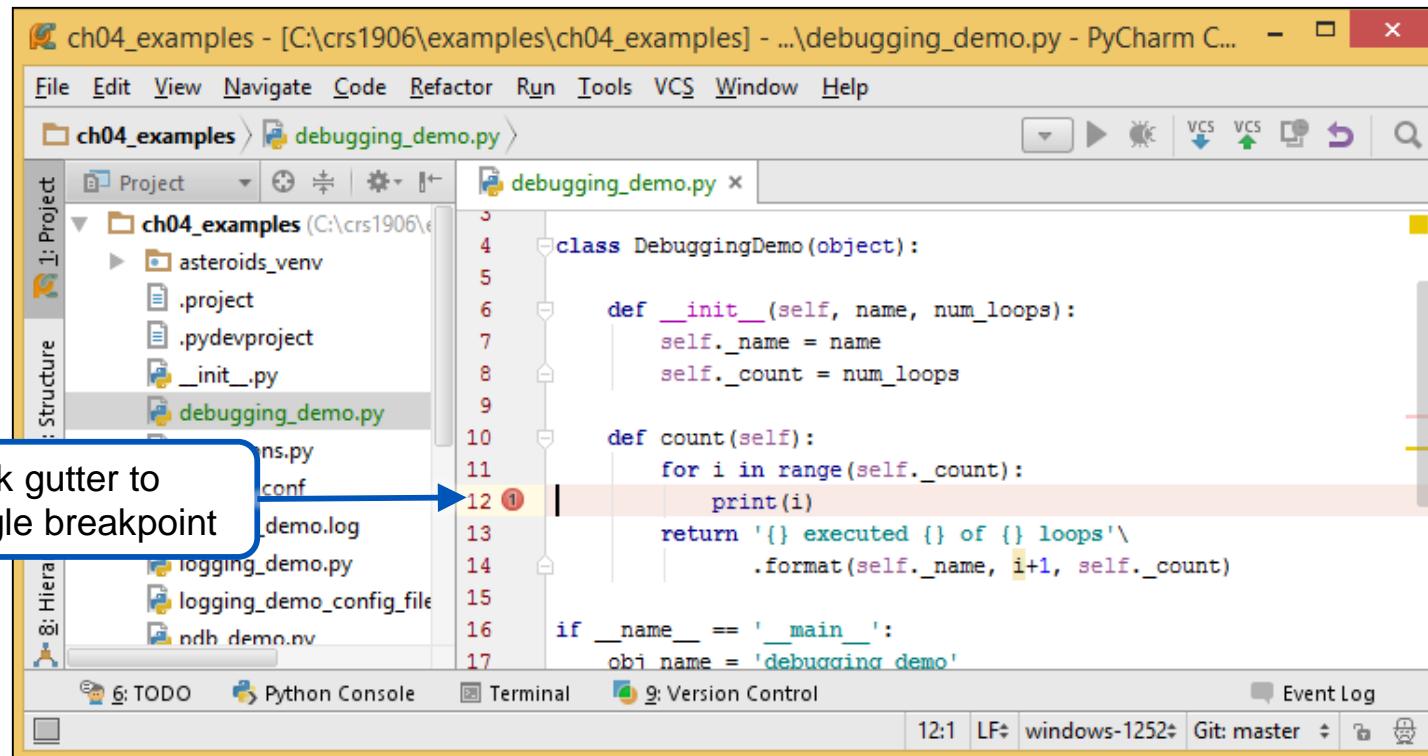
- ▶ Hands-On Exercise 4.1



Debugging With Python IDEs

Demo

- ▶ Python IDEs support interactive debugging
- ▶ Your instructor will demo PyCharm's debugger, and you will try it yourself
- ▶ Demo script: ch04_examples/debugging_demo.py
 - Set a breakpoint at line 12

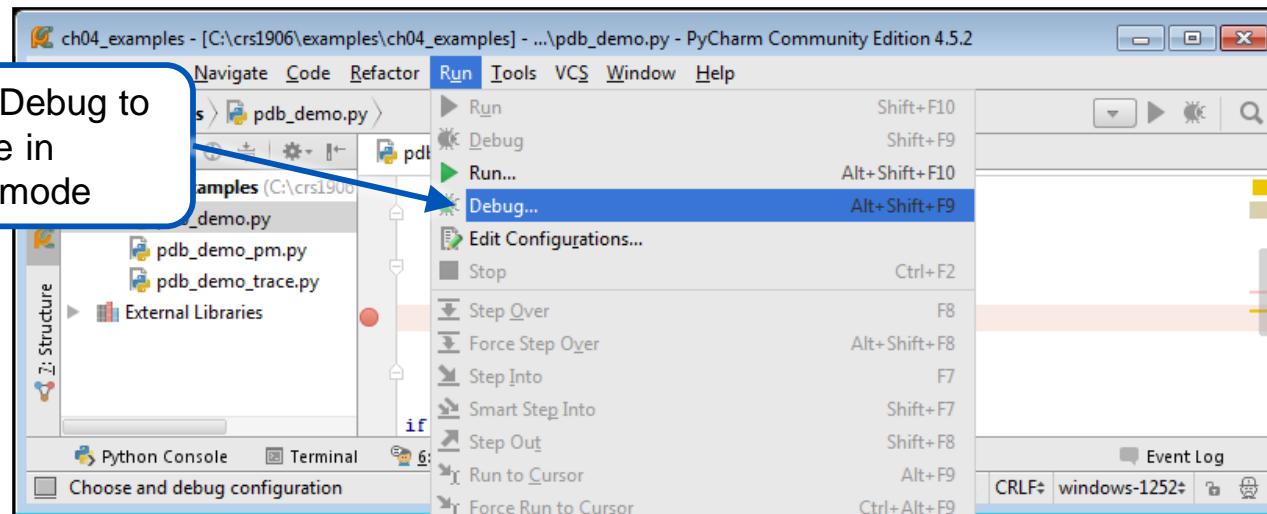


Debugging With Python IDEs

Demo

► Select Run | Debug

- PyCharm switches to Debug view

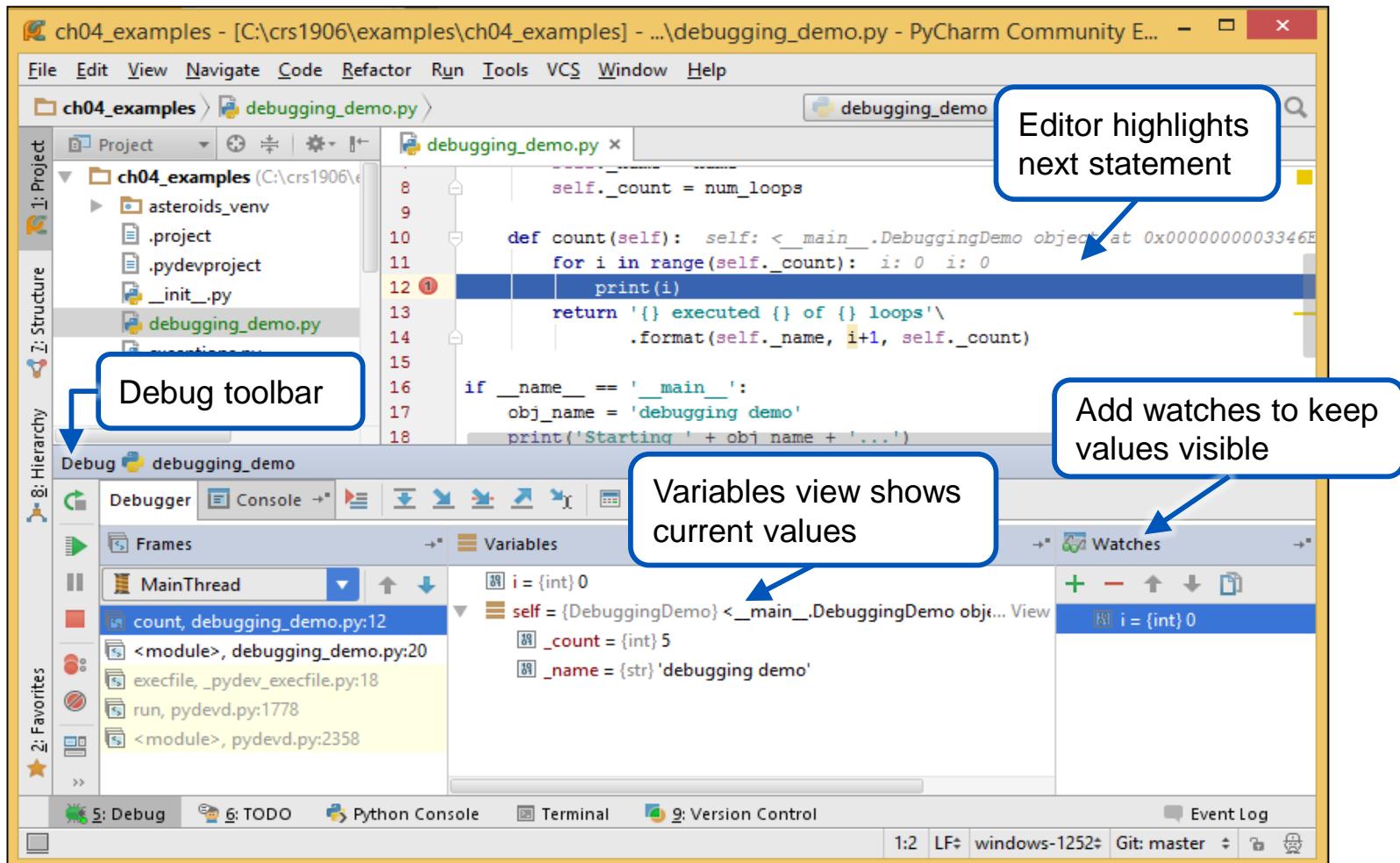


► Perform debug actions with the Debug toolbar

- Single-step through code
- Display values of variables
- Add watches to keep values always visible

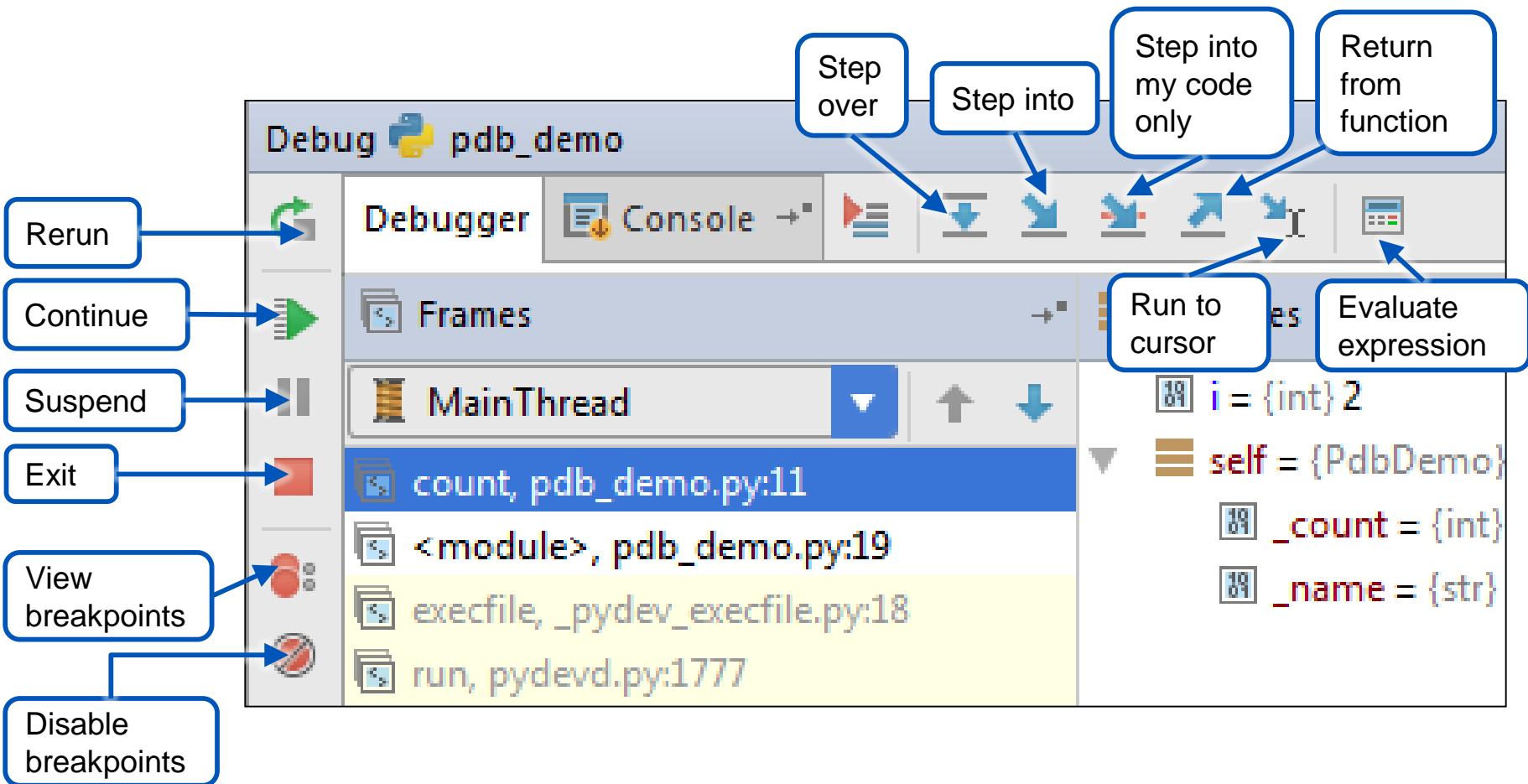
Debugging With PyCharm

Demo



Debugging With PyCharm

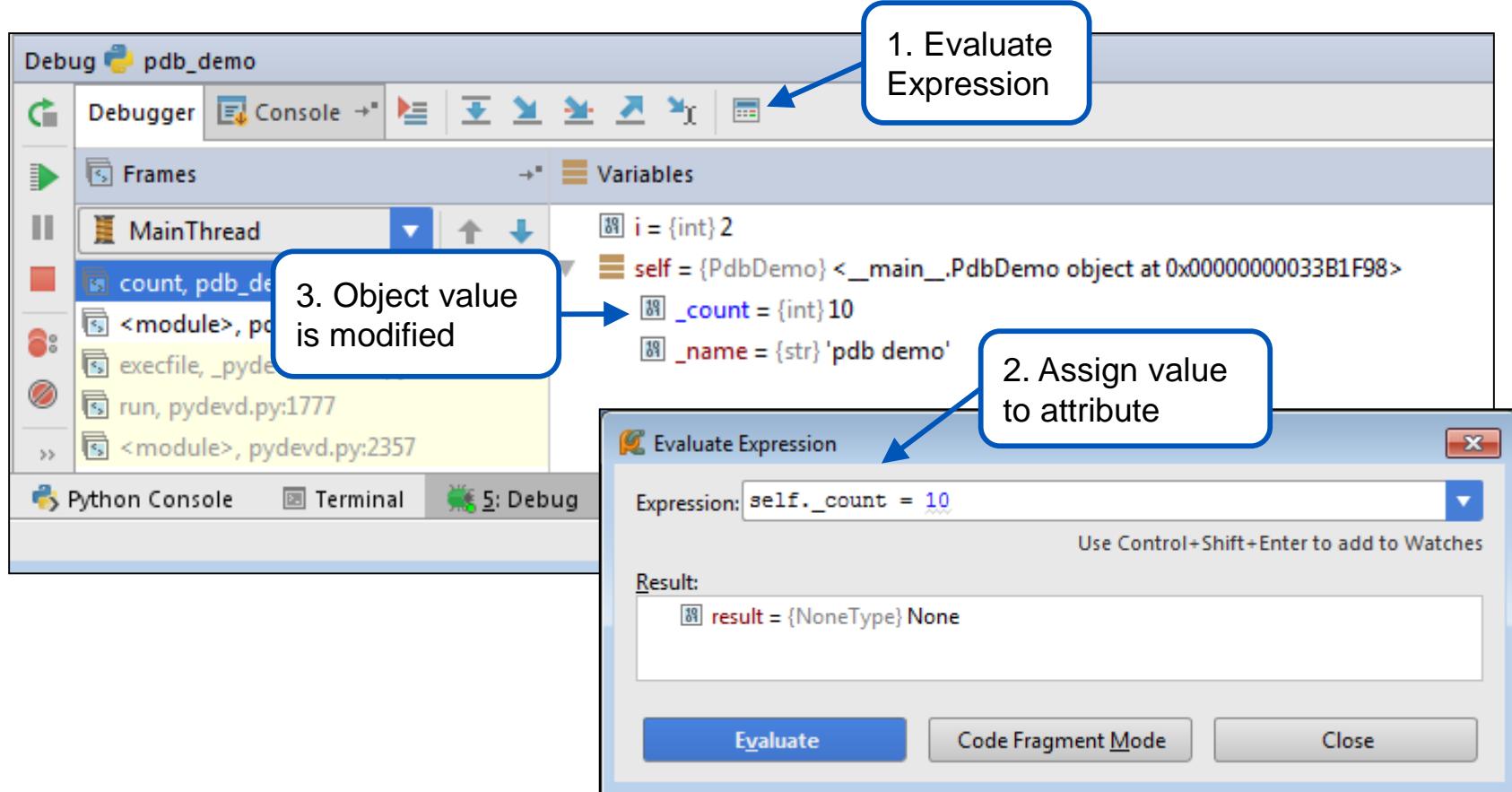
Reference



Modifying Variables During Execution

Reference

- To modify a variable or object attribute, use Evaluate Expression



Contents

- ▶ Defining and Raising Exceptions
- ▶ Logging Messages
- ▶ Verifying Code With Pylint
- ▶ Tracing Execution With IDEs

Hands-On Exercise 4.1



Hands-On Exercise 4.1

In your Exercise Manual, please refer to
**Hands-On Exercise 4.1: Error Detection and
Debugging**



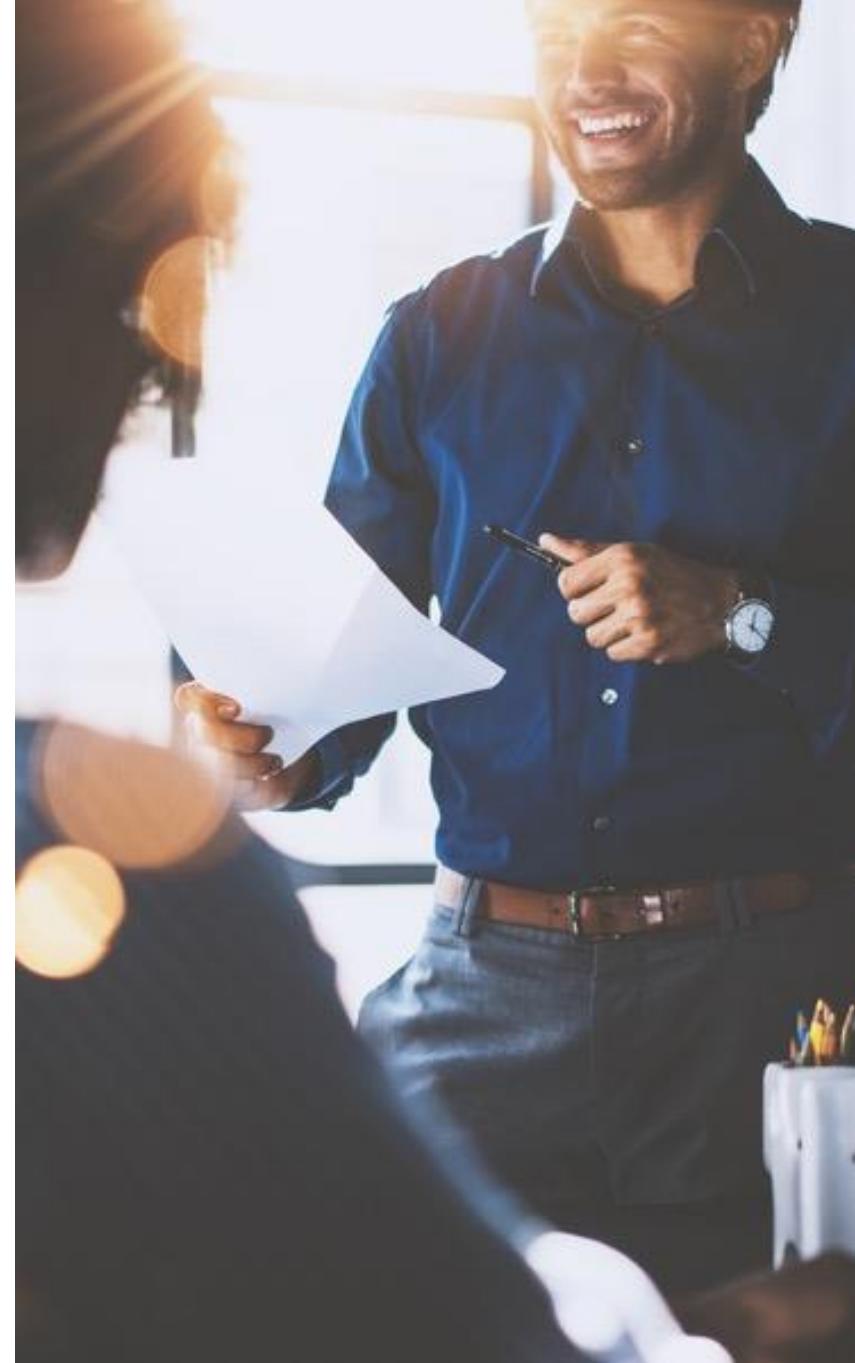
Best Practices: Error Detection and Debugging

- ▶ **Use `Logger.debug()` to log application state regularly**
 - To avoid massive log files, use `RotatingFileHandler` or `TimedRotatingFileHandler`
 - In production configuration file, set logging level to `INFO` or `WARNING`
- ▶ **Make Pylint usage a formal part of your development process**
 - Example: Define Pylint as a commit hook in your SCMS
 - Code can't be committed if Pylint reports errors or global evaluation < 9.0
 - Example: Code review can't be scheduled until Pylint issues are resolved
- ▶ **Check your code frequently with `pylint -E`**
 - Run `pylint` with all checks enabled before major milestones
- ▶ **Pick your favorite interactive debugger and master it**

SCMS = source code management system

Objectives

- ▶ Raise user-defined exceptions
- ▶ Log messages from Python modules for auditing and debugging
- ▶ Verify your code with Pylint
- ▶ Trace program execution with the PyCharm IDE

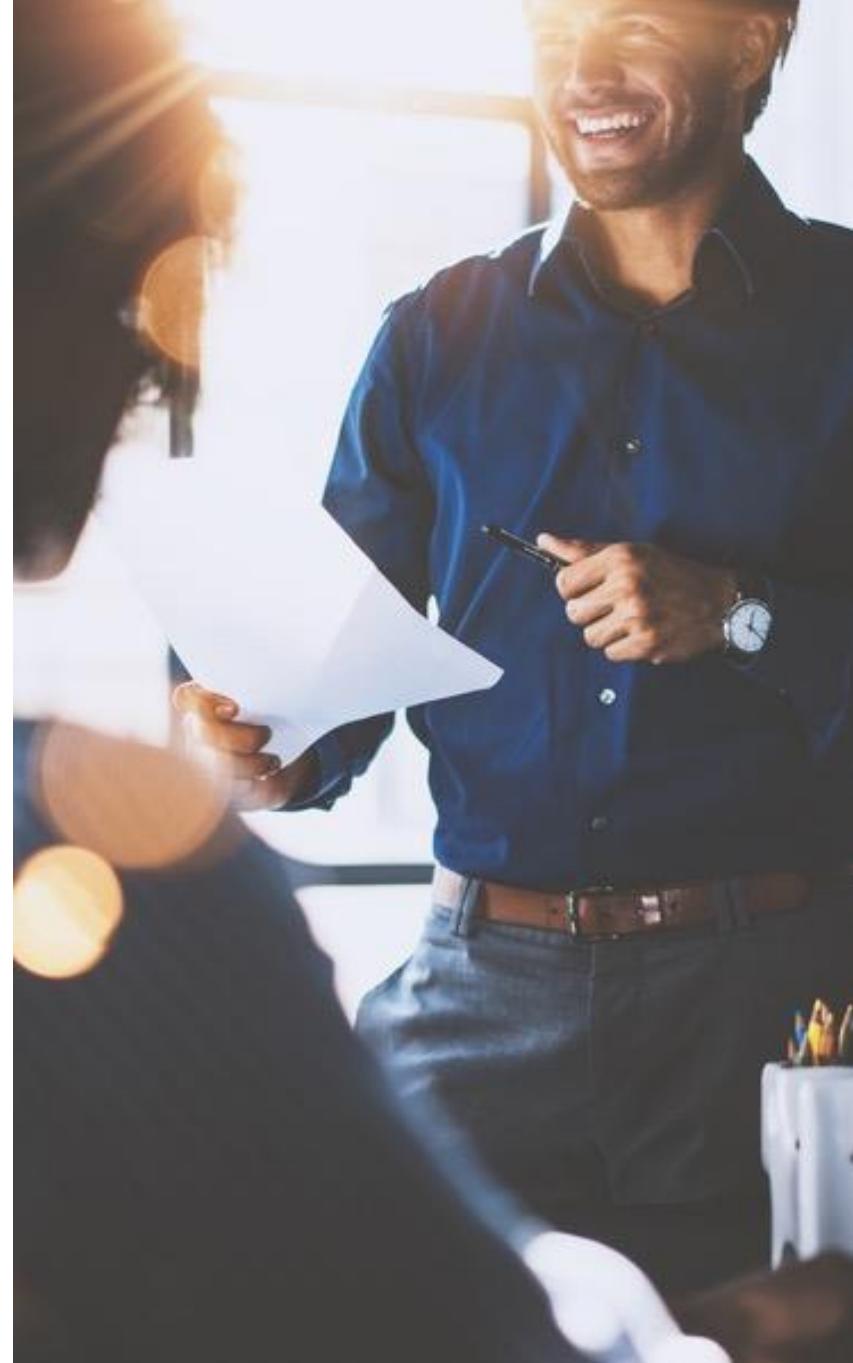


Chapter 5

Measuring and Improving Performance

Objectives

- ▶ Use profilers to monitor and measure execution of Python applications
- ▶ Employ Python language features to enhance performance
- ▶ Replace the standard CPython implementation with PyPy



Contents

Profiling Application Execution

- ▶ Python Features for Improving Performance
- ▶ Replacing CPython With PyPy
- ▶ Hands-On Exercise 5.1



Profiling Application Execution

- ▶ **Your applications must meet performance requirements**
 - They must be “fast enough” (not necessarily “as fast as possible”)
- ▶ **How do you know if application is fast enough?**
 - If it isn’t, how can you improve it?
- ▶ **Before trying to improve application performance, you need to measure it**
 - You need tools for timing software execution
- ▶ **Use a *profiler* to identify performance bottlenecks**
 - Profiler does an automated analysis of your application’s performance
 - Creates an *execution profile* that makes it easy to identify bottlenecks
 - Execution profile provides a benchmark for future measurements
- ▶ **Accurate performance measurements help focus optimization efforts**
 - You work on component with worst bottleneck
 - After optimizing, run with profiler again and compare results to benchmark
 - If no improvement, back out changes

Tools for Profiling Performance

- ▶ Python has many good options for measuring application performance
- ▶ Use time command in *nix bash or cygwin for rough estimates

```
ch05_examples $ time python fibonacci.py
real    0m0.063s
user    0m0.016s
sys     0m0.047s
```

user + sys = actual CPU time used

- ▶ Use standard Python modules for timing and profiling
 - timeit: Simple way to time small bits of Python code
 - cProfile: Measures execution time of functions and generates a report
- ▶ Use third-party profilers (command-line or GUI)

timeit Module

- ▶ Standard timeit module times execution of Python statements
- ▶ timeit defines a command-line interface
 - Example: Time execution of main() in fibonacci.py

```
> cd \crs1906\examples\ch05_examples
> python -m timeit --setup "import fibonacci" "fibonacci.main()"
100 loops, best of 3: 19.1 msec per loop
```

Setup statement
executes once

Statement to
be timed

- ▶ timeit executes a timing run by calling a statement many times in a loop
 - Then it repeats the loop several times (default 3)
- ▶ Options set number of loops and number of repeats

```
> python -m timeit -n 50 -r 5 -s "import fibonacci" "fibonacci.main()"
50 loops, best of 5: 19.5 msec per loop
```

timeit Module

► timeit also defines a callable interface

- timeit() method repeats timing loop three times
- repeat() method lets you change the number of loops
- By default, timeit() disables Garbage Collection (GC)
 - You can re-enable GC in setup string

```
> python  
>>> from timeit import timeit, repeat  
>>> timeit('fibonacci.main()', setup='import fibonacci', number=100)  
1.8912353385695668  
  
>>> timeit('fibonacci.main()', setup='gc.enable(); import fibonacci',  
number=100)  
1.8701257516789289  
  
>>> repeat('fibonacci.main()', setup='import fibonacci', repeat=5, number=50)  
[0.944113357982701, ...]
```

Default is 10^{**6}

Enable GC

List of 5 results

Specifies how many timing runs

timeit Module

- You can call timeit from your code

fibonacci.py

```
def main():
    ...

if __name__ == '__main__':
    from timeit import timeit
    loops = 100
    total_time = timeit("main()", setup="from __main__ import main",
                         number=loops)
    print('Called main() {} times, average time was {:.2f} seconds'
          .format(loops, total_time/loops))
```

Import main() in
setup string



The cProfile Module

- ▶ **Standard cProfile module times execution of Python applications**
 - Implemented as a C extension
 - Adds very little overhead to program execution
- ▶ **cProfile writes text report to standard output**
 - Each line contains statistics for one function call
 - Lines are sorted by module name and function name
 - You can modify cProfile's behavior with command-line options
- ▶ **pstats module formats profiler report**
 - Reads output of cProfile
 - Allows you to interactively sort and display report contents

Profiling With cProfile

► Run cProfile as a script from the command line

Note option -m cProfile

```
> cd \crs1906\examples\ch05_examples
> venv\Scripts\activate
> python -m cProfile mandelbrot.py 1000 50 | more
Main took 0:00:08.380971
Total sum of elements (for validation): 731119
    3864638 function calls (3861808 primitive calls) in 10.599 seconds

Ordered by: standard name

  ncalls  tottime percall cumtime  percall filename:lineno(function)
  ...
    1    0.000    0.000    0.003    0.003 main.py:1(<module>)
    1    0.000    0.000    0.000    0.000 main.py:81(TestProgram)
    1    0.009    0.009   10.600   10.600 mandelbrot.py:1(<module>)
1    5.137    5.137    8.379    8.379 mandelbrot.py:10(calculate_z_serial)
    1    0.398    0.398   10.586   10.586 mandelbrot.py:26(calc)
    1    0.000    0.000    0.000    0.000 memmap.py:1(<module>)
  ...
```

Most time spent
in this function

Sorted by filename and
line number

cProfile Statistics

- Report gives statistics on program execution

Column	Description
ncalls	Number of calls to this function
tottime	Total time spent in function (excluding time in calls to subfunctions)
percall	Time per call (tottime / ncalls)
cumtime	Cumulative time spent in this and all subfunctions
percall	Time per call (cumtime / primitive calls)

- Output shows most time spent in `calculate_z_serial()` function
 - `calculate_z_serial()` has highest tottime and percall values

Sorting Report Data

► Choose sort column using command-line option -s

```
> python -m cProfile -s tottime mandelbrot.py 1000 50 | more
Main took 0:00:08.158564
Total sum of elements (for validation): 731119
    3864828 function calls (3861998 primitive calls) in 10.142 seconds

Ordered by: internal time
              Sorted by total time spent in function
              (excluding calls to subfunctions)

      ncalls  tottime   percall  cumtime  percall   filename:lineno(function)
          1    4.888     4.888    8.170    8.170  mandelbrot.py:10(calculate_z_serial)
  564676    3.281     0.000    3.281    0.000  {built-in method abs}
          1    1.136     1.136    1.136    1.136  {built-in method system}
          1    0.333     0.333   10.129   10.129  mandelbrot.py:26(calc)
  258608    0.211     0.000    0.211    0.000  {method 'append' of 'list'}
...
...
```

► To save human-readable report, redirect output to a file

```
> python -m cProfile mandelbrot.py 1000 50 > profile_rep
```

Sorting Profiler Report

► Common sort fields

Column	Description
calls	Call count
ncalls	Call count
cumtime	Time spent in function (including subfunction calls)
tottime	Time spent in function (excluding subfunction calls)
file	File name
module	File name (alias for file)
nfl	Name/file/line
percall	Time per call (cumtime / primitive calls)

Saving Reports

- ▶ **Problem:** to get a different sort order, you need to run your program again
 - Not practical if program takes a long time to run
- ▶ **Solution:** use standard pstats module
 - pstats lets you interactively sort and display cProfile's report
- ▶ **To save report to be formatted by pstats, use cProfile's -o option**

```
> python -m cProfile -o mandelbrot.prof mandelbrot.py 1000 50
```

Save binary output that
can be read by pstats

Displaying Profiling Reports With pstats

- **pstats module displays reports generated by cProfile**
 - pstats command-line interface can manipulate report interactively

```
> python -m cProfile -o mandelbrot.prof mandelbrot.py 1000 50
> python -m pstats mandelbrot.prof
Welcome to the profile statistics browser.
mandelbrot.prof%
mandelbrot.prof% help
Documented commands (type help <topic>):
=====
EOF  add  callees  callers  help  quit  read  reverse  sort  stats  strip
mandelbrot.prof% sort
Valid sort keys ...:
totime -- internal time
module -- file name
calls -- call count
cumulative -- cumulative time
...
time -- internal time
mandelbrot.prof% sort time
```

Start pstats

help displays list of commands

Displays valid sort keys

Sort report by time

Formatting Reports With pstats

```
mandelbrot.prof% help stats
```

Displays help for stats command

Print statistics from the current stat object.

Arguments may be:

- * An integer maximum number of entries to print.
- * A decimal fractional number between 0 and 1, controlling what fraction of selected entries to print.
- * A regular expression; only entries with function names that match it are printed.

```
mandelbrot.prof% stats 10
```

Displays formatted report

```
Fri Jul 10 14:51:24 2015      mandelbrot.prof
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	5.380	5.380	8.675	8.675	mandelbrot.py:10(calculate...)
3564676	3.287	0.000	3.287	0.000	{built-in method abs}
1	1.225	1.225	1.225	1.225	{built-in method system}
1	0.468	0.468	11.206	11.206	mandelbrot.py:26(calc)
258594	0.338	0.000	0.338	0.000	{method 'append' of 'list' ...}
...					

GUI Profilers: SnakeViz

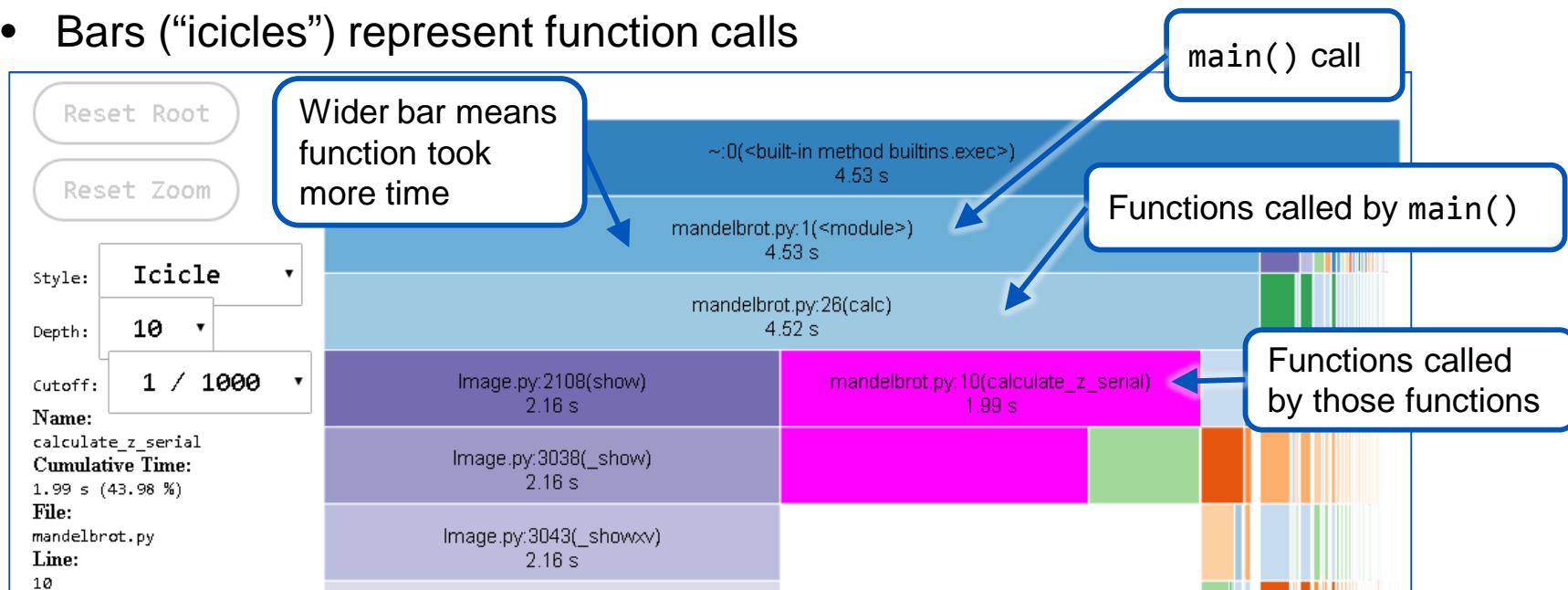
► SnakeViz: Displays cProfile results file graphically

- Starts a simple web server, displays results file in web browser

```
> pip install snakeviz  
> python -m cProfile -o mandelbrot.prof mandelbrot.py 1000 50  
> snakeviz mandelbrot.prof
```

► SnakeViz displays execution profile as a graph

- Bars (“icicles”) represent function calls



Source: Graphic image from SnakeViz, open-source software. <https://jiffyclub.github.io/snakeviz/>.

SnakeViz Demo

Demo

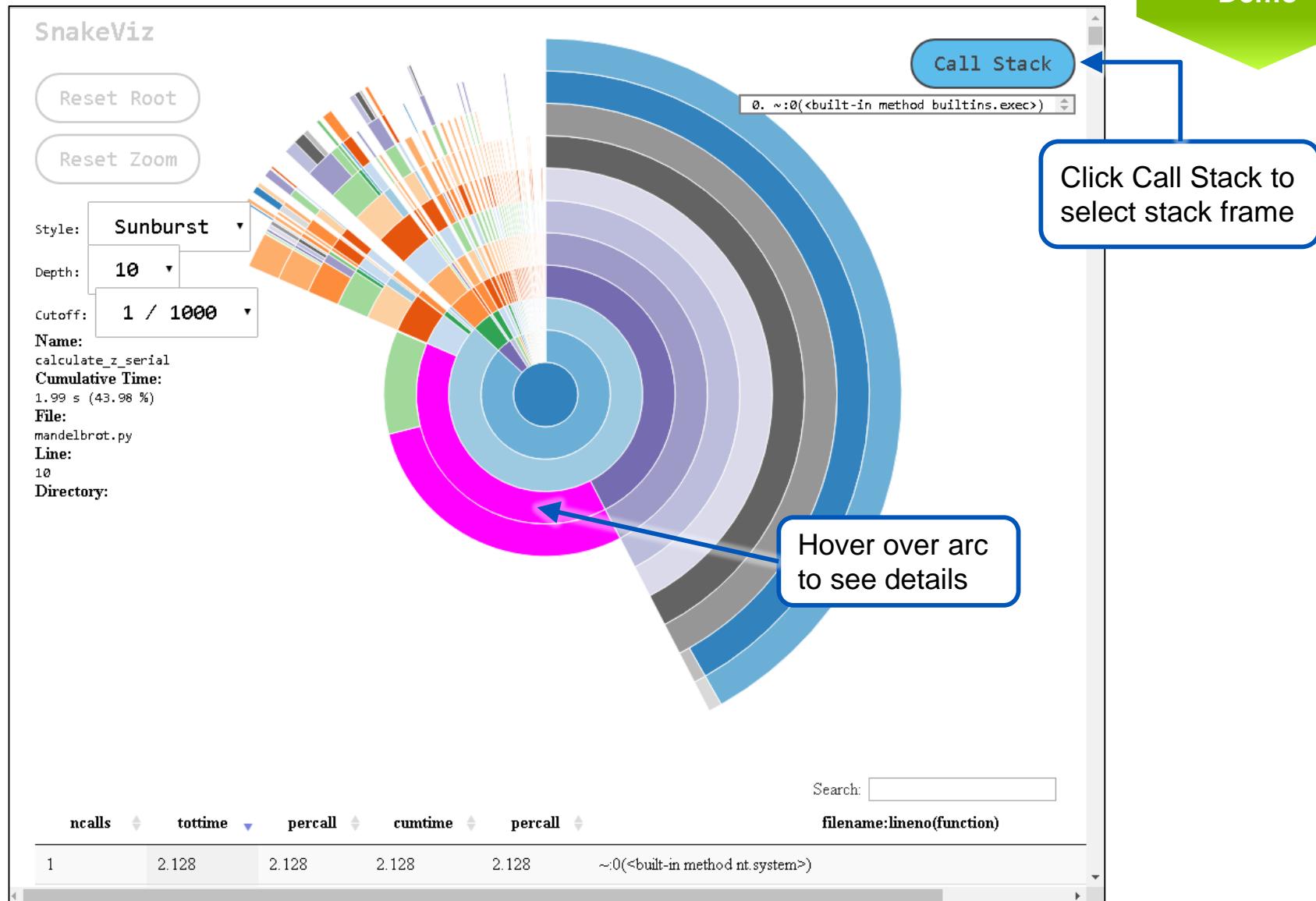
- ▶ Your instructor will now demo SnakeViz
- ▶ Steps to start SnakeViz:

```
> cd \crs1906\examples\ch05_examples  
> venv\Scripts\activate  
> python -m cProfile -o mandelbrot.prof mandelbrot.py 1000 50  
> snakeviz mandelbrot.prof
```

- ▶ Interacting with SnakeViz
 1. Click Call Stack button
 2. In sunburst, select arc for calc function (2nd layer from center, purple arc)
 3. Now you've drilled into the call for calc
 - Hover over an arc to see its execution profile
 - Click an arc to drill down into it
 4. To get back to initial display, select stack frame 0 from call stack

SnakeViz Demo

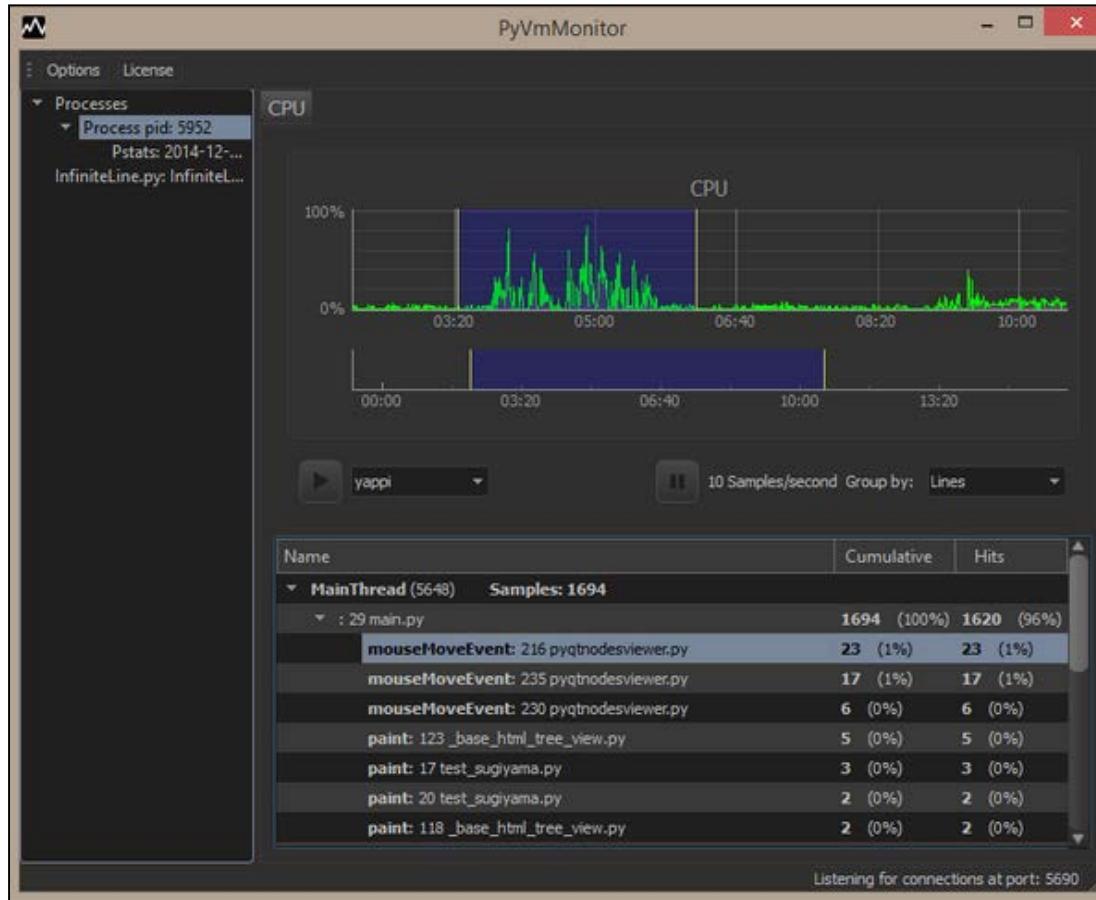
Demo



Source: Graphic image from SnakeViz, open-source software. <https://jiffyclub.github.io/snakeviz/>.

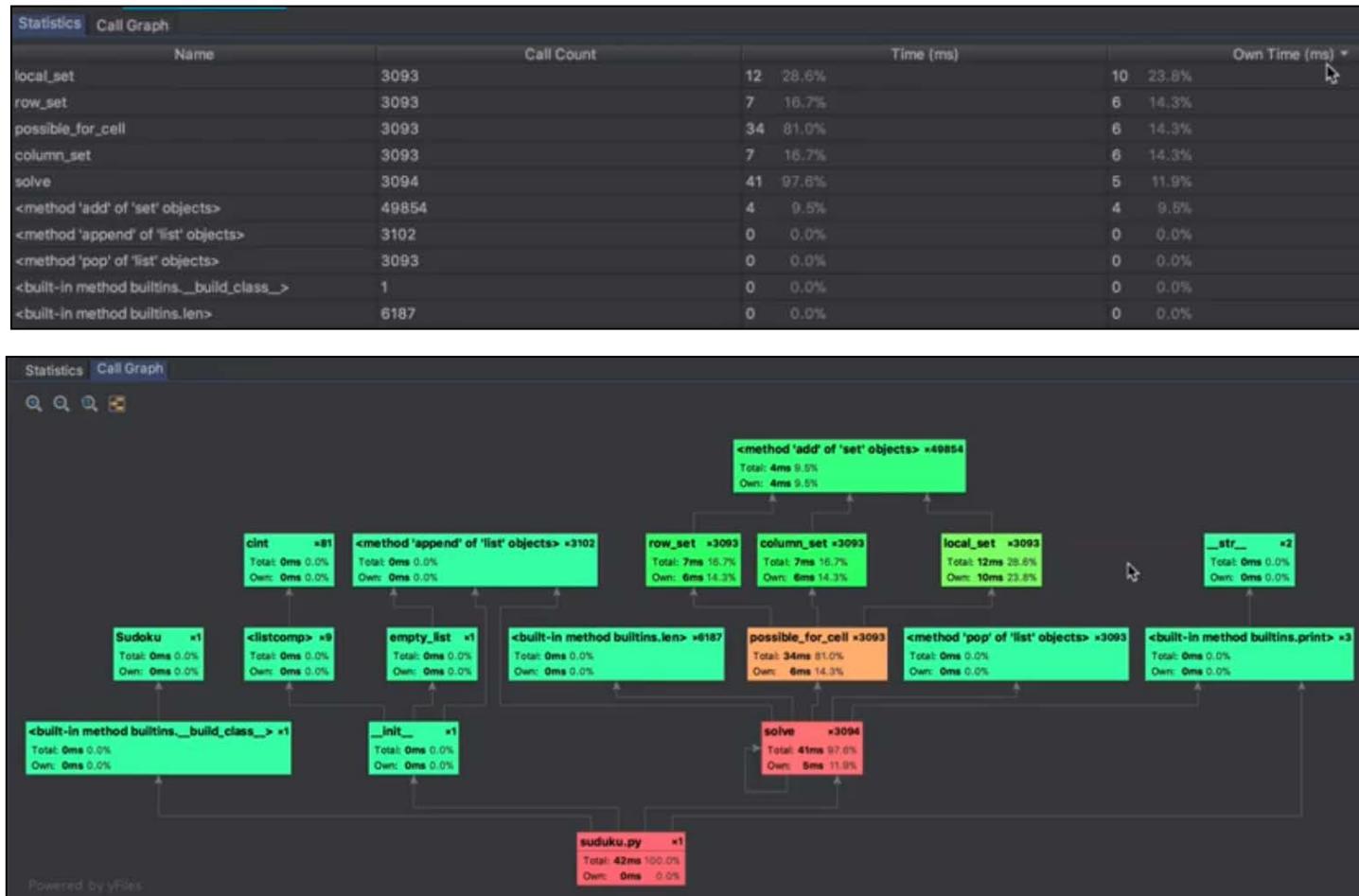
GUI Profilers: PyVmMonitor

- ▶ **PyCharm integrates with commercial PyVmMonitor profiler**
 - Requires PyVmMonitor to be installed separately



GUI Profilers: PyCharm

- PyCharm Professional Edition has built-in GUI profiler
 - Displays execution statistics and a call graph



Contents

- ▶ Profiling Application Execution

Python Features for Improving Performance

- ▶ Replacing CPython With PyPy
- ▶ Hands-On Exercise 5.1



Strategies for Improved Performance

- ▶ **We'll explore strategies for improving application performance**
 - Don't apply these strategies unless performance really is a problem
 - ▶ **Start with clean designs and simple, understandable, error-free code**
 - Don't start optimizing until your code is thoroughly tested and correct
 - ▶ **Measure performance before starting optimizations**
 - If application meets performance requirements, leave it as is
 - ▶ **Before optimizing, build a regression test suite**
 - Allows you to try different optimizations with confidence
 - ▶ **Targeted optimizations save huge amounts of development and testing**
 - Optimizing a SQL query may yield more than months of tweaking algorithms
-
- “Premature optimization is the root of all evil.”*

—Donald Knuth
-
- “The best performance improvement is the transition from the nonworking to the working state.”*

—John Ousterhout

Strategies for Improved Performance

- ▶ **Use a profiler to identify performance bottlenecks**
 - Measure, don't guess
 - Focus optimization work on component with worst bottleneck
 - Run `timeit` before and after optimizing
 - Run regression test suite
- ▶ **Determine whether application is *CPU-bound* or *I/O-bound***
 - CPU-bound: Most application time is spent executing instructions
 - I/O-bound: Most application time is spent waiting for I/O to complete
- ▶ **CPU-bound applications benefit from code optimizations**
 - Tuning algorithms
 - Moving initialization code out of loops
 - Making functions shorter to leverage hardware cache
- ▶ **For I/O-bound applications, look for ways to reduce I/O**
 - Keep I/O out of loops
 - Use cached or buffered I/O to move larger amounts of data with fewer calls

Python Idioms for Performance

- ▶ Certain Python idioms can improve application performance
- ▶ Building large strings with concatenation can be expensive
 - Each '+' operation creates a temporary string, which is then discarded
 - To build large strings, store inputs in a list and join them in one operation

string_cat_demo.py

```
input_lines = ... # big list of lines

r = ''
for line in input_lines:
    r += line.lower()

r = ''
for line in input_lines:
    r = ''.join((r, line.lower()))

ra = []
for line in input_lines:
    ra.append(line.lower())
r = ''.join(ra)

r = ''.join([line.lower() for line in input_lines])
r = ''.join(line.lower() for line in input_lines)
```

The diagram illustrates five different Python string manipulation idioms:

- String concatenation:** Points to the first snippet where `r += line.lower()` is used.
- Join individual strings:** Points to the second snippet where `r = ''.join((r, line.lower()))` is used.
- Append strings to list, then a single join:** Points to the third snippet where `ra.append(line.lower())` and `r = ''.join(ra)` are used.
- List comprehension:** Points to the fourth snippet where `r = ''.join([line.lower() for line in input_lines])` is used.
- Generator expression:** Points to the fifth snippet where `r = ''.join(line.lower() for line in input_lines)` is used.

Results of String Concatenation Test

► Output of previous example

```
> python string_cat_demo.py
__main__.to_lower_string_concat(): 0.0220 seconds
__main__.to_lower_join_individual_strings(): 1.7735 seconds
__main__.to_lower_append_strings_to_list(): 0.0042 seconds
__main__.to_lower_list_comprehension(): 0.0032 seconds
__main__.to_lower_generator_expression(): 0.0035 seconds
```

► Timing results

Strategy	Result
String concatenation	Slow
Join individual strings	Very slow
Append strings to list, then a single join	Fast
Add strings to list comprehension, then join	Very fast
Iterator with generator expression, then join	Very fast

Python Idioms for Performance

- ▶ **Use local variables whenever possible**
 - Python accesses local variables much more efficiently than global variables
- ▶ **Avoid import statements inside a function**
 - Each time the function is called, the `import` statement executes again
 - Module will be imported only once, but the extra imports are inefficient
- ▶ **Function call overhead in Python is relatively high**
 - Where appropriate, functions should handle data aggregates

Do this:

```
def to_lower(lines):  
    r = []  
    for line in lines:  
        r.append(line.lower())  
    return ''.join(r)  
  
input_lines = [...]  
result = to_lower(input_lines)
```

One call to
`to_lower()`

Don't do this:

```
def to_lower(line, buffer):  
    buffer.append(line.lower())  
  
input_lines = [...]  
r = []  
for line in input_lines:  
    to_lower(line, r)  
result = ''.join(r)
```

Many calls to
`to_lower()`

Compiled Python Code

- ▶ **Python interpreter automatically generates “compiled” files**
 - Compiled files contain platform-independent Python byte code
 - Files have extension .pyc
- ▶ **CPython interpreter “executes” byte code directly**
 - Interpreter does not convert byte code to native machine code
- ▶ **.pyc file is generated when module is imported**
 - Standalone scripts don't generate .pyc files
- ▶ **Python interpreter's “optimization” flags have little impact on performance**
 - -O: Removes assert statements from .pyc
 - -OO: Removes __doc__ strings from .pyc
 - Prior to Python 3.5, optimized code was written to .pyo file

Contents

- ▶ Profiling Application Execution
- ▶ Python Features for Improving Performance

Replacing CPython With PyPy

- ▶ Hands-On Exercise 5.1



PyPy Python Interpreter

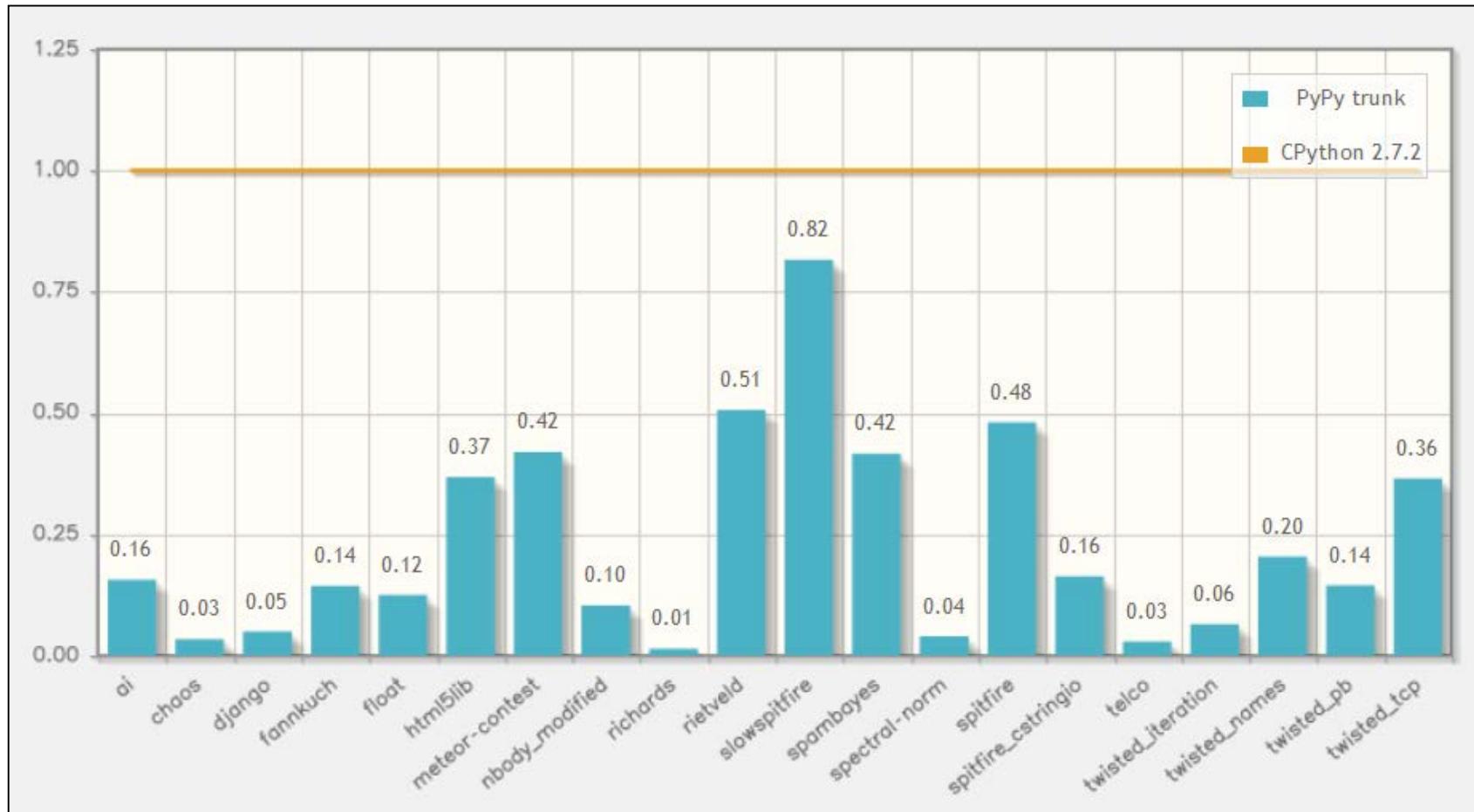
- ▶ **PyPy is a drop-in replacement for standard CPython interpreter**
 - Web site: <http://pypy.org>
- ▶ **Implements the full Python language**
 - Supports Python 2.7 and Python 3.x
- ▶ **Goals: Better Python performance with no code changes**
 - High-performance garbage collector (GC)
 - High-performance Just-In-Time (JIT) compiler
- ▶ **PyPy's JIT compiler translates your Python code as it executes**
 - Compiler generates machine code that (usually) runs faster than bytecode
 - No pre-compile step required
- ▶ **Python programs may use less memory**
 - Most Python programs run unchanged with PyPy

"If you want your code to run faster, you should probably just use PyPy."

—Guido van Rossum

PyPy Performance

- Graph compares PyPy and CPython running common Python frameworks



Source: Graphic image is from <http://speed.pypy.org>.

Installing PyPy

- ▶ **Download PyPy:** <http://pypy.org/download.html>
 - Choose Python version (2.7 or 3.x)
 - Select appropriate binary distribution
- ▶ **Unzip/uncompress downloaded archive**
- ▶ **Add PYPY_HOME and PYPY_HOME/bin to beginning of PATH**
- ▶ **Set up PyPy with the usual Python development tools**
 - Install pip with pypy3 -m ensurepip
 - Use PyPy's pip3 to install modules in PyPy

```
set PATH=%PYPY_HOME%;%PYPY_HOME%\bin;%PATH%
pypy3 -m ensurepip
pip3 install virtualenv
```

Install virtualenv in pypy

- ▶ **Now run your Python scripts using pypy3 instead of python**

Running PyPy

Do Now

- ▶ Run the following commands

```
> cd \crs1906\examples\ch05_examples  
> use_python37  
> python primes_python.py  
...  
Total execution time: 8.9 seconds  
> use_pypy  
> pypy3 primes_python.py  
...  
Total execution time: 0.3 seconds
```

Note difference in execution times

PyPy Caveats

- ▶ **Most Python code runs unchanged with PyPy**
 - However, most C extension modules won't work
 - Packages that work:
<https://bitbucket.org/pypy/compatibility/wiki/Home>
- ▶ **Most Python code runs faster with PyPy**
 - However, some Python code runs slower (sometimes *much* slower)

```
> cd C:\crs1906\examples\ch05_examples
> use_python37 && python string_cat_demo.py
__main__.to_lower_string_concat : 0.0255 seconds
...
> use_pypy && pypy string_cat_demo.py
__main__.to_lower_string_concat : 3.4860 seconds
```

Function ran far slower
with PyPy

- ▶ **PyPy's version of Python language is behind CPython's version**
 - Latest Python features may not be available
- ▶ **Linux version of PyPy is easy to install and configure**
 - But Windows PyPy version may not be as reliable
 - Examples: Problems running tkinter

Contents

- ▶ Profiling Application Execution
- ▶ Python Features for Improving Performance
- ▶ Replacing CPython With PyPy

Hands-On Exercise 5.1



Hands-On Exercise 5.1

In your Exercise Manual, please refer to
**Hands-On Exercise 5.1: Measuring and
Improving Performance**



Best Practices: Enhancing Performance

- ▶ Start with clean designs and simple, understandable, error-free code
- ▶ Measure performance before starting optimizations
 - If application meets performance requirements, leave it as is
- ▶ Use a profiler to identify performance bottlenecks
 - Focus optimization work on one component with worst bottleneck
 - After optimizing, run timings again and compare results to benchmark



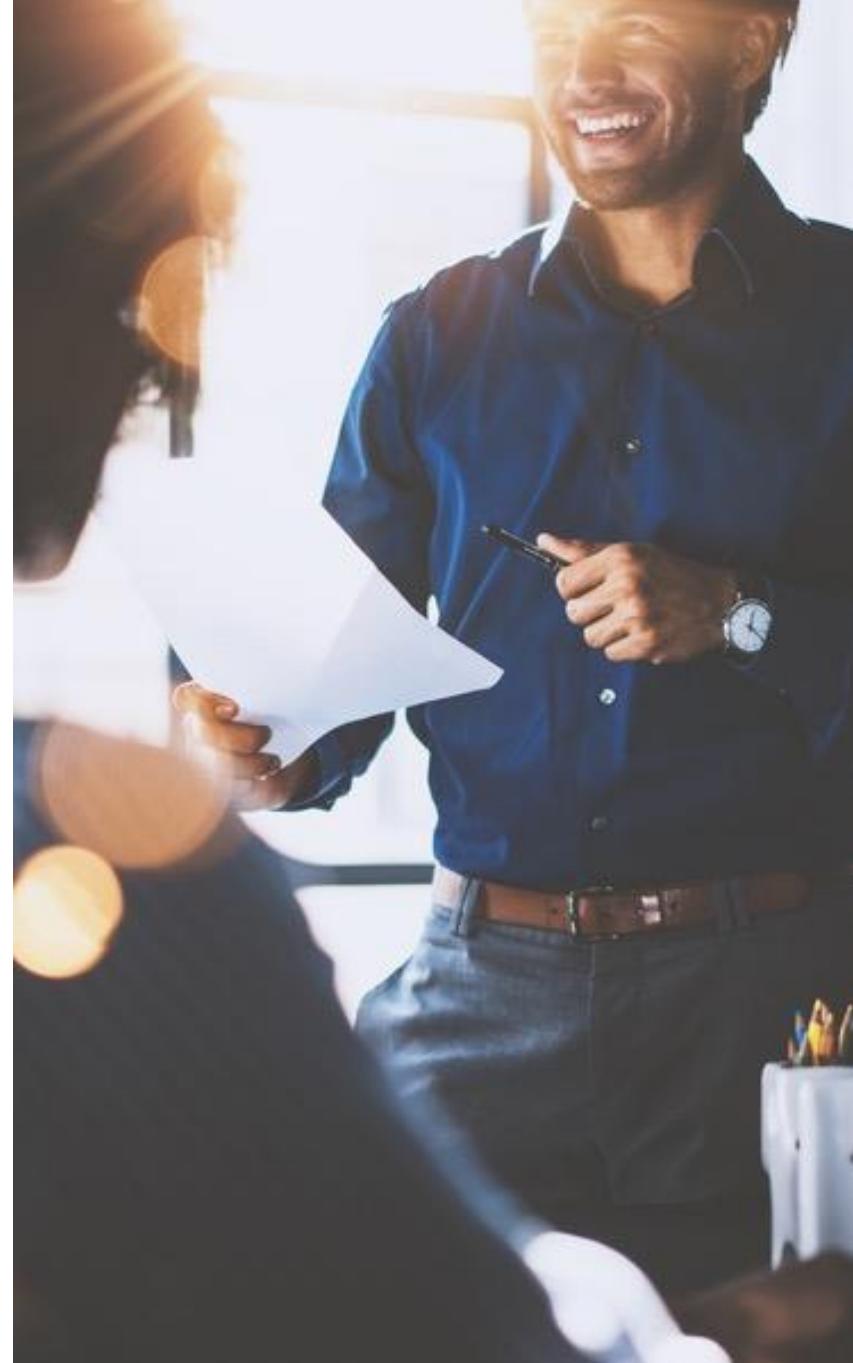
Best Practices: Enhancing Performance

- ▶ **Use `timeit` for performance benchmarks**
 - `cProfile` does not give accurate results
 - `cProfile` shows you the problems, `timeit` tells you whether you fixed them
- ▶ **For performance gains with no code changes, replace CPython with PyPy**
- ▶ **If PyPy doesn't help, try Cython**
 - Cython: superset of Python
 - Requires minor code changes, but may improve performance dramatically



Objectives

- ▶ Use profilers to monitor and measure execution of Python applications
- ▶ Employ Python language features to enhance performance
- ▶ Replace the standard CPython implementation with PyPy

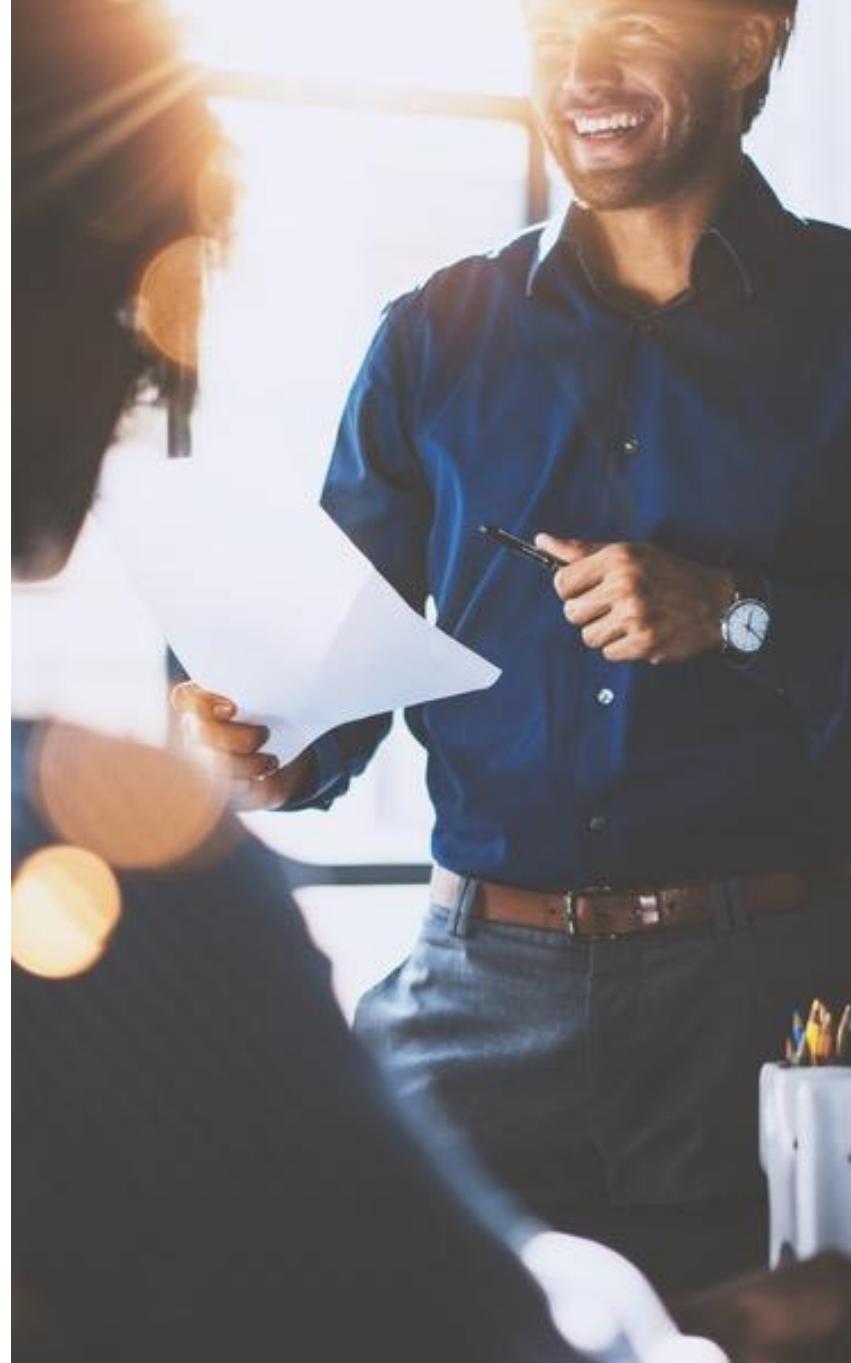


Chapter 6

Design Patterns II

Objectives

- ▶ Create loosely coupled many-to-many relationships with the Observer design pattern
- ▶ Attach additional responsibilities to an object dynamically using the Decorator design pattern
- ▶ Control access to objects with the Proxy design pattern



Contents

Observer Design Pattern

- ▶ **Decorator Design Pattern**
- ▶ **Proxy Design Pattern**
- ▶ **Hands-On Exercise 6.1**



Maintaining Consistency Between Objects

- ▶ **Design problem: Need to maintain consistency between related objects**
 - Example: Different graphical views of the same data
 - Example: Software components receiving events from the same hardware
- ▶ **We don't want objects to be tightly coupled**
 - Objects shouldn't need to know each other's implementation details
 - Should be easy to add new views of existing data
- ▶ **Solution: The *Observer* design pattern**

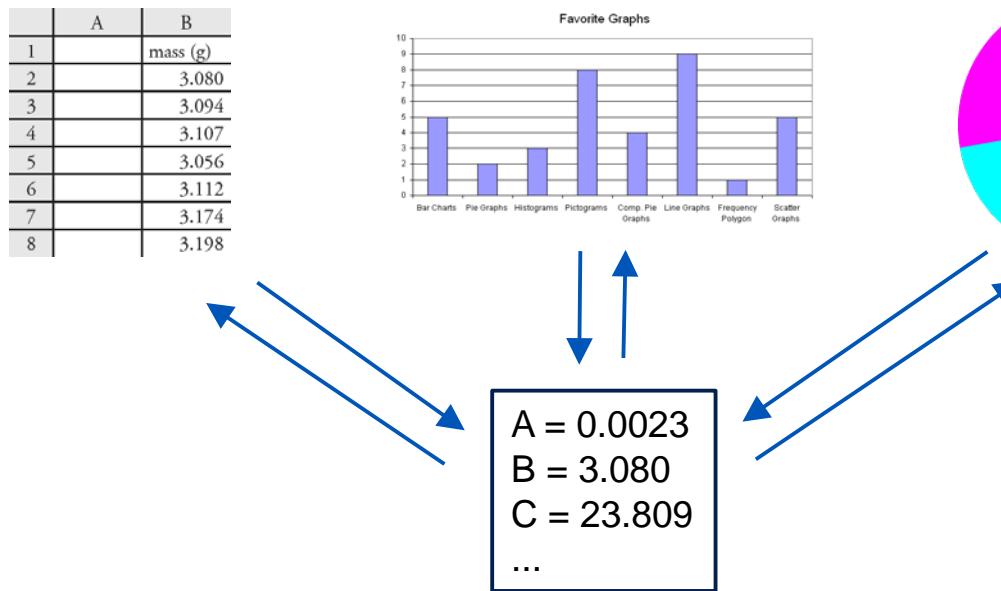
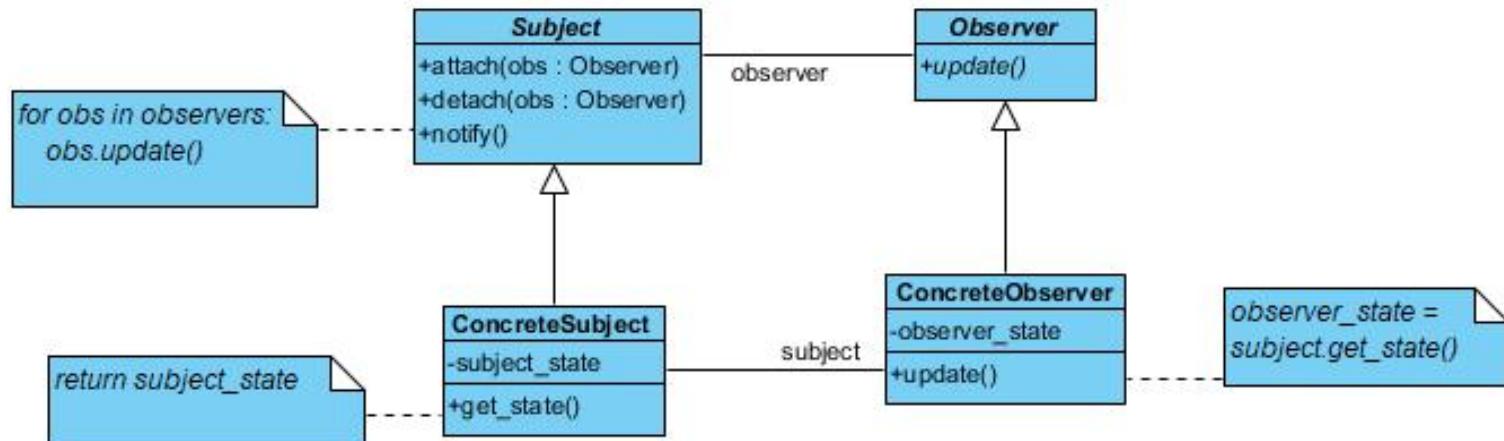


Image Source: Generated by Michael Woinoski.

Observer Design Pattern

- ▶ **Observer pattern defines dependency between *Subject* and *Observers***
 - When Subject changes state, all Observers are notified
 - Observers update themselves with new state of Subject
- ▶ **Loosely coupled many-to-many relationship**
 - Subject doesn't need to know concrete Observer classes
 - Observer needs to know Subject's state, but not details of Subject itself
 - Subject may have many Observers, and Observer may observe multiple Subjects
- ▶ **UML class diagram of Observer pattern**



UML = Unified Modeling Language

Image Source: Generated by Michael Woinoski.

Observer Design Pattern

- ▶ **Observer pattern appears in many contexts**
 - Callback functions: Framework calls user code to notify it of events
 - Publish/subscribe (Pub/Sub): Subscribers are notified when a message arrives
- ▶ **Examples of Observer pattern**
 - Model/View/Controller (MVC) architecture: Model is Subject, View is Observer
 - Database triggers: Database table is Subject, trigger is Observer
 - Django signal system: Django component is Subject, user code is Observer
- ▶ **Consequences of Observer pattern**
 - Subjects and Observers can vary independently
 - You can add Observers without modifying Subject or other Observers

Implementation of Observer Pattern

observer.py

```
class Subject(metaclass=abc.ABCMeta):
    def __init__(self):
        self._observers = set()

    def observer_attach(self, observer, *observers):
        for obs in (observer,) + observers:
            self._observers.add(obs)
            obs.update(self)

    def observer_detach(self, observer):
        self._observers.discard(observer)

    def observer_notify(self):
        for obs in self._observers:
            obs.update(self)

class Observer(metaclass=abc.ABCMeta):
    @abstractmethod
    def update(self, subject):
        pass
```

Create empty observers set

Attach one or more observers

Notify new observers of current state

Remove an observer

Notify observers of new state

Concrete subclasses must override abstract method

Discussion of Implementation of Observer Pattern

- ▶ **Subject constructor creates empty set for Observers**
- ▶ **observer_attach() method accepts one or more Observer arguments**
- ▶ **In for loop, first Observer and remaining arguments are concatenated**

```
def observer_attach(self, observer, *observers):  
    for obs in (observer,) + observers:  
        self._observers.add(obs)  
        observer.update(self)
```
- ▶ **If argument list might be very long, use more efficient itertools.chain()**

```
def observer_attach(self, observer, *observers):  
    for obs in itertools.chain((obs,), observers):
```

Subject and Observer Subclasses

```
class ConcreteSubject(Subject):
    def __init__(self, value=None):
        super().__init__()
        self._state = None
        self.state = value
    @property
    def state(self):
        return self._state
    @state.setter
    def state(self, value):
        if self._state != value:
            self._state = value
            self.observer_notify()
class ConcreteObserver(Observer):
    def update(self, subject):
        print('subject state = {}'
              .format(subject.state))
```

The diagram illustrates the flow of the Subject and Observer pattern through annotated code snippets:

- Initialize _state data member**: Points to the line `self._state = None`.
- Call state property setter**: Points to the line `self.state = value`.
- Called when state property is accessed**: Points to the `@property` block.
- Called when state property is set**: Points to the `@state.setter` block.
- Notify observers of new state**: Points to the line `self.observer_notify()`.
- Concrete Observer overrides update()**: Points to the `update` method definition.
- Call state getter method**: Points to the line `.format(subject.state))`.

Concrete Subject and Concrete Observer Implementation

- ▶ **Concrete Subject subclass defines state as a property**
 - When state property is accessed, Python calls @property method
 - When state property is set, Python calls @property.setter method
 - Data member referenced in setter must be initialized before setter is called
- ▶ **Concrete Observer subclass overrides update()**
 - Subject passes itself as the argument to Observer's update()
 - Observer *pulls* data from Subject:

```
class Subject:  
    def observer_notify(self):  
        for obs in self._observers:  
            obs.update(self)
```

```
class ConcreteObserver(Observer):  
    def update(self, subject):  
        self.do_work(subject.state)
```

- ▶ **Alternatively, Subject can *push* its state to Observer's update()**

```
class Subject:  
    def observer_notify(self):  
        for obs in self._observers:  
            obs.update(self.state)
```

```
class ConcreteObserver(Observer):  
    def update(self, data):  
        self.do_work(data)
```

Run Sample Observer Script

Do Now

1. Switch to PyCharm

- Open project C:\crs1906\examples\ch06_examples

2. Open observer.py and review client code in the main() function

```
def main():
    initial_state = 1
    subject = ConcreteSubject(initial_state)

    observer1 = ConcreteObserver('observer1')
    observer2 = ConcreteObserver('observer2')

    subject.observer_attach(observer1, observer2)

    subject.state = 100
    subject.state = 200
```

Run Sample Observer Script

Do Now

3. Right-click observer.py and select “Run observer.py”

- Note that the observers are notified every time the subject changes

```
Creating subject with initial state 1...
```

```
Creating two observers...
```

```
Attaching observers to subject...
```

```
observer1.update() called, subject state = 1
```

```
observer2.update() called, subject state = 1
```

```
Changing subject state to 100...
```

```
observer1.update() called, subject state = 100
```

```
observer2.update() called, subject state = 100
```

```
Changing subject state to 200...
```

```
observer1.update() called, subject state = 200
```

```
observer2.update() called, subject state = 200
```

Contents

- ▶ Observer Design Pattern

Decorator Design Pattern

- ▶ Proxy Design Pattern
- ▶ Hands-On Exercise 6.1



Adding Responsibilities to Objects

- ▶ **Design problem: Need to add responsibilities to objects**
 - Example: GUI Text widget may need a border and/or scrollbars
 - But not every Text widget needs a border or scrollbars
 - We want to add borders or scrollbars only when needed
 - Ideally, with no changes to code of Text widget or the client of Text widget
- ▶ **Solution: The *Decorator* design pattern**

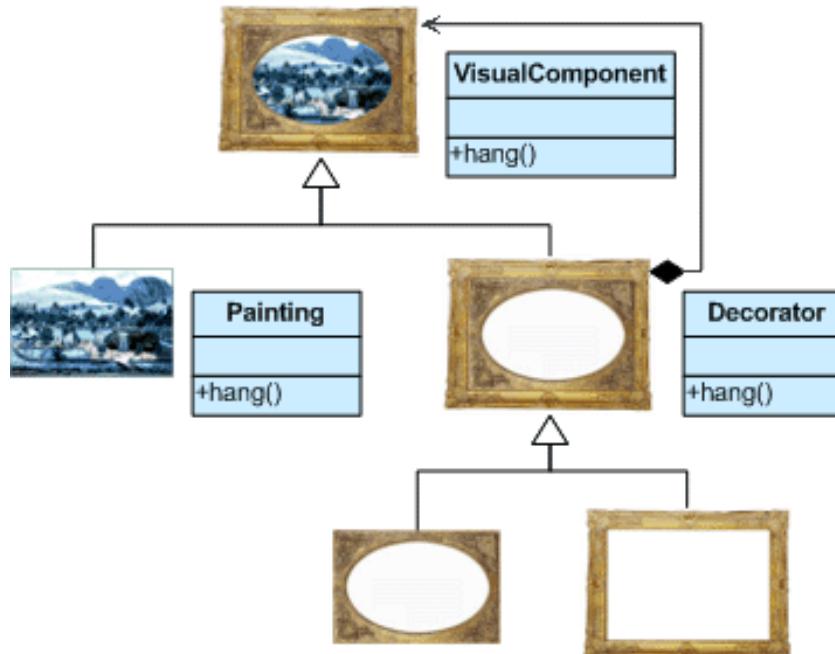
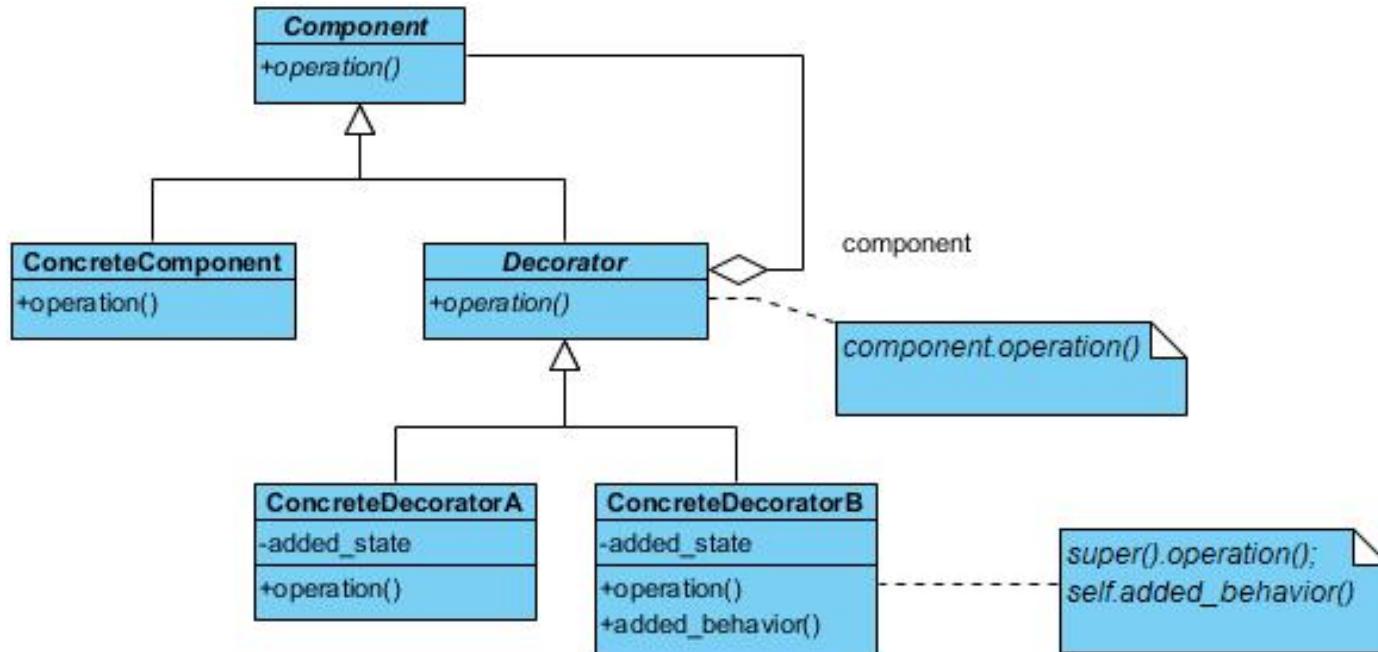


Image Source: Created and generated by Michael Woynoski.

Decorator Design Pattern

► UML diagram of Decorator pattern



► Common use cases for Decorator pattern

- Applying database transactions to methods
- Logging function arguments at entry and return value at exit
- Profiling function execution

Characteristics of the Decorator Pattern

- **Decorator pattern lets you add responsibilities to objects as needed**
 - Each Decorator has a single responsibility
 - Decorators can be chained



- **Example**



- **Client object and target object are unaware of the presence of decorators**
 - Decorators can be added and removed with no changes to client or target
- **Python supports the Decorator pattern using decorator functions**
 - Decorator can be applied using monkey patching or special decorator syntax
- **Python Web site maintains a library of submitted decorator code**
 - <https://wiki.python.org/moin/PythonDecoratorLibrary>

Python Decorators

► Example: Execution profiling decorator

- In this example, decorator is applied with monkey patching

decorators/profiling_decorator.py

```
def profile_call(target_func):
    def wrapper(*args, **kwargs):
        start = time.process_time()
        result = target_func(*args, **kwargs)
        end = time.process_time()
        print('{}: {} secs'.format(target_func.__name__, end - start))
        return result
    return wrapper

def read_file(filepath, pass_phrase):
    return ...

read_file = profile_call(read_file)
data = read_file('magic_spells.bin', 'Hogwarts')
```

Nested function definition

Call target function

Decorator function returns reference to nested wrapper function

Define target function

Replace target function with wrapper function

Target function name is now an alias for wrapper function

Runtime Call Sequence of Decorator Functions

- When you call target function, Python calls decorator function instead
 - For a working example, see decorators/profiling_monkey_patch.py

```
def profile_call(target_func):  
    def wrapper(...):  
        ...  
        result = target_func(...)  
        ...  
        return result  
    return wrapper
```

```
def read_file(...):  
    return ...
```

```
read_file = profile_call(read_file)
```

```
data = read_file(...)
```

1. Interpreter reads function definitions

6. wrapper adds actions, then calls target function

3. profile_call returns reference to nested wrapper function

2. You call profile_call, passing reference to target function

4. target function is replaced by wrapper

5. You call the function using target reference, which is now wrapper

Applying Decorators with "@"

- **Decorator can be applied to target function with `@decorator-name`**
 - No need for monkey patch

decorators/profiling_at_decorator.py

```
def profile_call(target_func):  
    def wrapper(...):  
        ...  
    return wrapper  
  
@profile_call  
def read_file(...):  
    ...  
  
data = read_file(...)
```

Define decorator as before and apply
decorator directly to target function

- **Output**

read_file: 0.2002 secs

Logging Decorator

► Example: Logging decorator

- Logs function name and arguments on entry and function results on exit

decorators/logging_decorator.py

```
def log_call(fn):
    def log_args_and_results(*args):
        print("{}({}): enter".format(
            fn.__name__, ", ".join(str(a) for a in args)))
        result = fn(*args) ← Call target function
        print("{}({}): exit, result: {}".format(
            fn.__name__, ", ".join(str(a) for a in args), result))
        return result

    return log_args_and_results

@log_call ← Apply decorator
def read_file(...):
```

Print target function result

Print target function name and arguments

Working With Decorators

Do Now

1. Switch to PyCharm and open `decorators/logging_decorator.py`
 - Examine the `log_call()` decorator function definition
2. Open `decorators/logging_demo.py`
 - Note `@log_call` on the `nsum()` function definition
3. Right-click `logging_demo.py` and select “Run `logging_demo.py`”
 - Note the output from the logging decorator

```
...
nsum(1000000): enter
nsum(1000000): exit, result: 500000500000
in main, nsum(1000000) = 500000500000
...
```

Output from logging
decorator

Working With Decorators

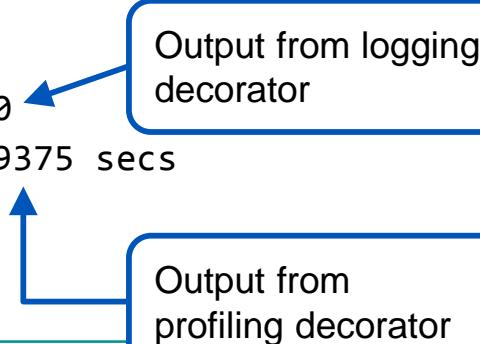
Do Now

4. Open `decorators/profiling_decorator.py`
 - Examine `profile_call()` decorator function definition
5. Edit `decorators/logging_demo.py`
 - Add `@profile_call` to `nsum()` function definition

```
@profile_call  
@log_call  
def nsum(n):
```

6. Switch to PyCharm and execute `logging_demo.py`
 - Note output from both decorators

```
...  
nsum(1000000): enter  
nsum(1000000): exit, result: 500000500000  
log_args_and_results(1000000) ran in 0.09375 secs  
  
in main, nsum(1000000) = 500000500000  
...
```



Contents

- ▶ Observer Design Pattern
- ▶ Decorator Design Pattern

Proxy Design Pattern

- ▶ Hands-On Exercise 6.1



Controlling Access to an Object

- ▶ **Many common use cases require controlling access to an object**
 - Deferring object initialization until its first reference
 - Example: Loading large images on demand
 - Example: Executing database queries for nested objects when accessed
 - Looking up and calling methods on remote objects
 - Securing access to sensitive objects
 - Performing extra actions when an object is accessed
 - Example: Reference counting
- ▶ **But we want to keep client code as simple as possible**
 - Client objects shouldn't be aware of access control logic
 - Interface of controlled object must be the same as an uncontrolled object
- ▶ **Solution: *Proxy* design pattern**

Proxy Design Pattern

- ▶ **Proxy pattern provides a “stand-in” or placeholder for another object**
 - Proxy controls access to target object
- ▶ **Participants**
 - Subject: Defines common interface for RealSubject and Proxy
 - RealSubject: Object being proxied; implements Subject interface
 - Proxy: Has reference to RealSubject; implements Subject interface

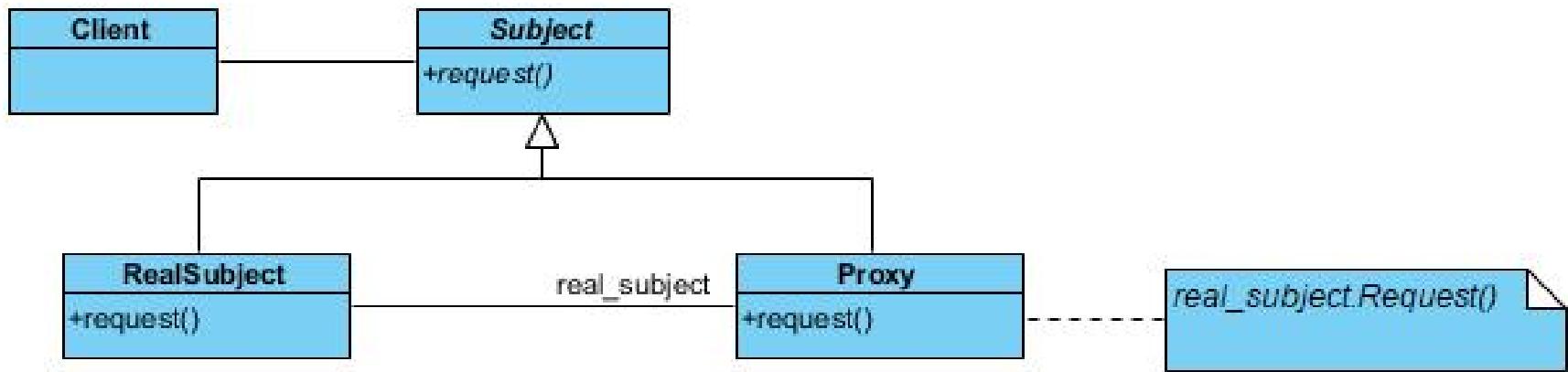
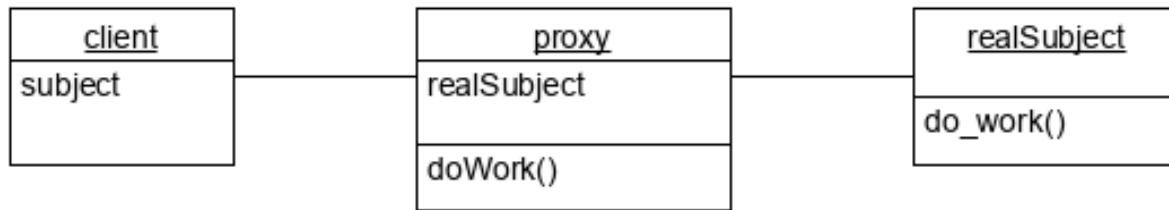


Image Source: Generated by Michael Woinoski.

Proxy Design Pattern

- ▶ Client has reference to Proxy instead of RealSubject
- ▶ Client calls methods of Subject interface
 - Proxy and RealSubject both implement Subject interface
 - Client is not aware that it is calling methods of Proxy instead of RealSubject



- ▶ Types of proxies
 - *Virtual proxy*: Lazy-loads target object when it's first referenced
 - *Remote proxy*: Looks up and calling methods on remote objects
 - *Protection proxy*: Secures access to sensitive objects
 - *Reference proxy*: Performs extra actions when an object is accessed

Implementing the Proxy Design Pattern

- ▶ **UML diagrams for Decorator and Proxy are similar, but purposes differ**
 - Decorator: Adds a responsibility to a target object
 - Target object still performs its usual task
 - Proxy: Controls access to an object
 - May completely replace RealSubject's existing functionality
- ▶ **Python implementation options**
 1. Define Subject as abstract class
 - Proxy and RealSubject both extend Subject
 2. Subject is not implemented at all
 - Proxy just implements same methods as RealSubject
 3. Proxy extends RealSubject, overriding methods that need special treatment
 - Good for proxying classes from external libraries or other projects

Example of Proxy Pattern

► Example of proxy that lazy-loads large images

- Proxy is implemented with an abstract base class

image_proxy.py

```
class Image(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_content(self, path): pass

class ConcreteImage(Image):
    def __init__(self, path):
        self.image_content = ... # read image content from file

    def get_content(self):
        return self.image_content

class LazyLoadingImage(Image):
    def __init__(self, path):
        self.path = path
        self.concrete_image = None

    def get_content(self):
        if not self.concrete_image:
            self.concrete_image = ConcreteImage(self.path)
        return self.concrete_image.get_content()
```

Abstract base class defines interface

Concrete subclass loads image immediately

Proxy doesn't load image until needed

When client requests image, proxy delegates to concrete subclass

Lazy-Loading Images

► Example of client code that uses proxy

- If client's `display_image()` is not called, image is never loaded from disk

```
class ImageClient:  
    def __init__(self, image):  
        self.image = image  
  
    def display_image(self):  
        content = self.image.get_content()  
        ... # render image content  
  
def main():  
    proxy = LazyLoadingImage('LargeMagellanicCloud.jpg')  
  
    client = ImageClient(proxy)  
  
    if len(sys.argv) > 1 and sys.argv[1] == '--display-image':  
        client.display_image()
```

image argument can be `ConcreteImage` or `LazyLoadingImage`

`LazyLoadingImage` loads image only if client fetches it

Driver creates `LazyLoadingImage`

Driver passes proxy to `ImageClient` constructor

Client's `display_image()` is called only if command line option is present

Working With a Proxy

Do Now

- ▶ Execute the driver without displaying an image
 - Note that the ConcreteImage constructor is never called

```
> cd \crs1906\examples\ch06_examples
> python image_proxy.py
LazyLoadingImage.__init__("LargeMagellanicCloud.jpg")
ImageClient.__init__(LazyLoadingImage("LargeMagellanicCloud.jpg"))
```

- ▶ Now execute the following command to display an image
 - Note that the proxy calls the ConcreteImage constructor

```
> python image_proxy.py --display-image
LazyLoadingImage.__init__("LargeMagellanicCloud.jpg")
ImageClient.__init__(LazyLoadingImage("LargeMagellanicCloud.jpg"))
ConcreteImage.__init__("LargeMagellanicCloud.jpg")
```

ConcreteImage constructor is called

Proxy Characteristics

- ▶ **Proxy enforces the Single Responsibility Principle**
 - `ConcreteImage` responsibility: load an image
 - `LazyLoadingImage` responsibility: implement lazy loading
- ▶ **Consequences of Proxy pattern**
 - Client code doesn't change
 - Target code doesn't change
 - Proxy logic is encapsulated in a single class
- ▶ **Easy to make `LazyLoadingImage` generic**
 - Generic proxy can do lazy loading for any class
 - See `ch06_examples/generic_lazy_loading_proxy.py`

Contents

- ▶ Observer Design Pattern
- ▶ Decorator Design Pattern
- ▶ Proxy Design Pattern

Hands-On Exercise 6.1



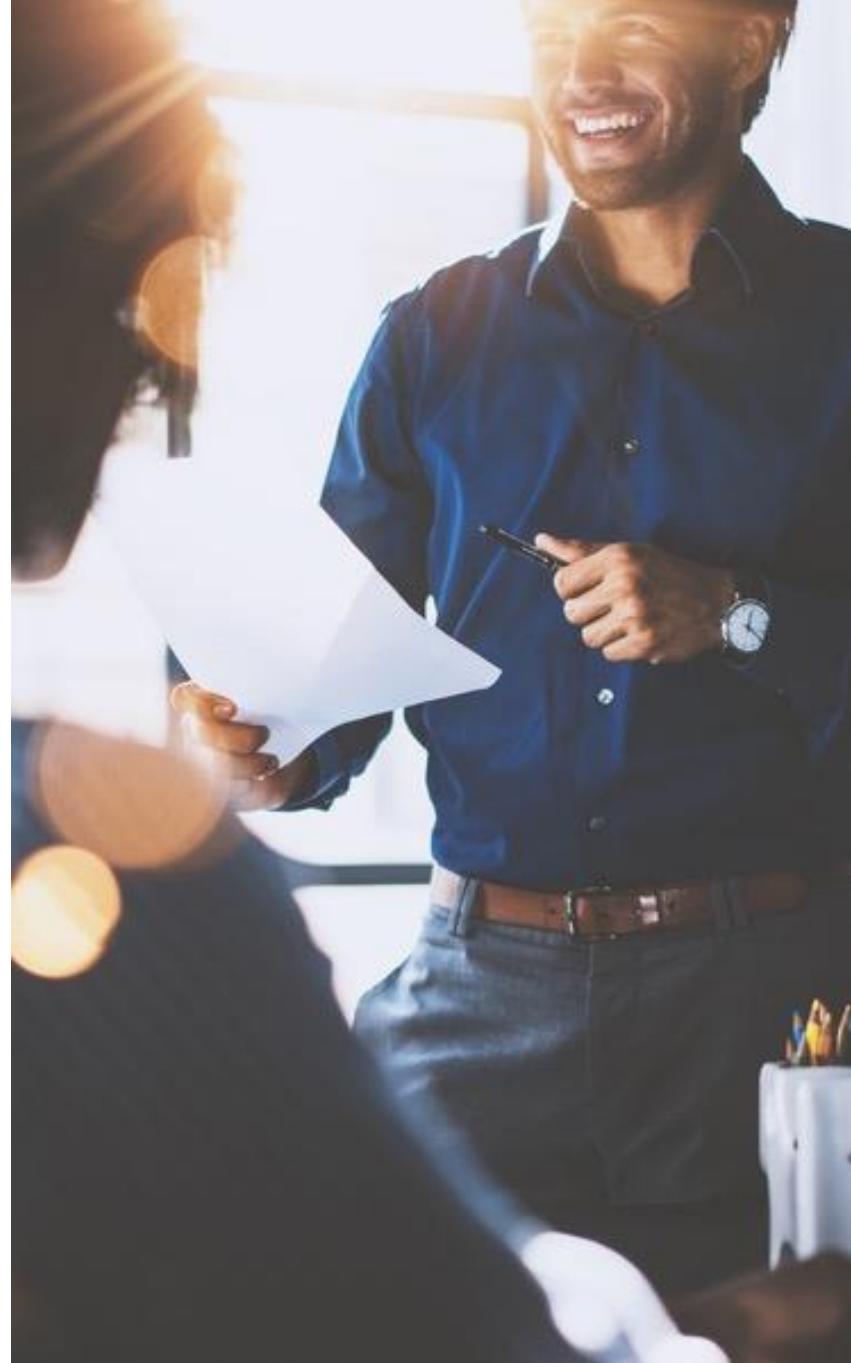
Hands-On Exercise 6.1

In your Exercise Manual, please refer to
**Hands-On Exercise 6.1: Applying the
Decorator, Observer, and Proxy Design
Patterns**



Objectives

- ▶ Create loosely coupled many-to-many relationships with the Observer design pattern
- ▶ Attach additional responsibilities to an object dynamically using the Decorator design pattern
- ▶ Control access to objects with the Proxy design pattern

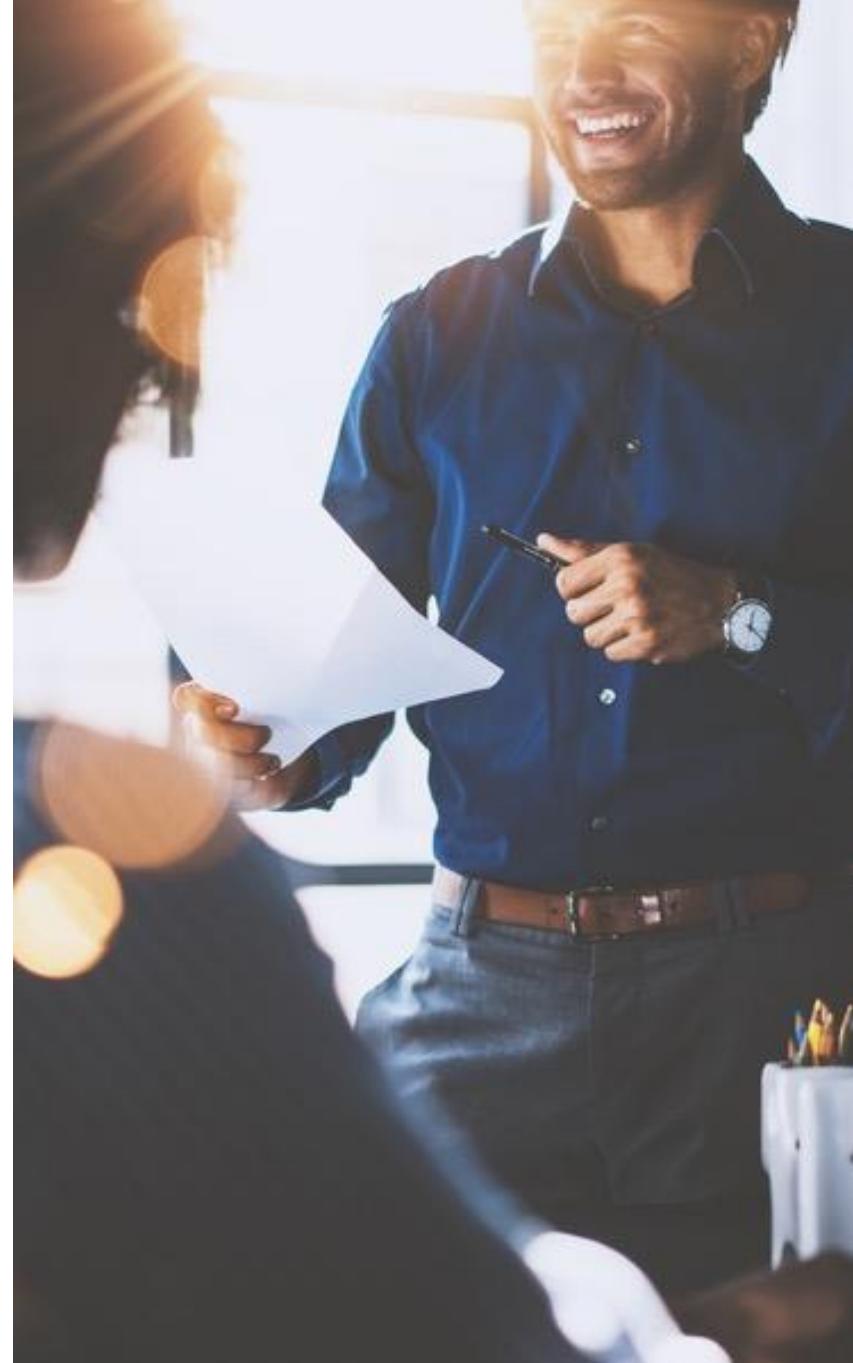


Chapter 7

Installing and Distributing Modules

Objectives

- ▶ **Install modules from the PyPI repository using pip**
- ▶ **Establish isolated Python environments with venv and virtualenv**
- ▶ **Package Python modules and applications for distribution**
- ▶ **Upload your Python modules to a local repository**



Contents

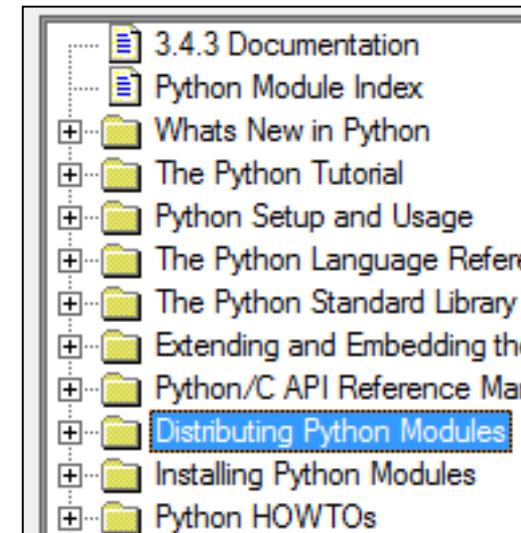
Installing and Updating Python Modules

- ▶ **Creating Virtual Python Environments**
- ▶ **Packaging and Distributing Modules and Applications**
- ▶ **Installing Packages from a Local Repository**
- ▶ **Hands-On Exercise 7.1**



Installing and Distributing Python Modules

- ▶ **Python standard library includes tools for installing and updating modules**
 - Modules can be installed from public repositories or from your project's local repository
- ▶ **Standard library also includes tools for packaging and distributing your modules and applications**
 - Defines best practices for packaging and distribution
 - Assures your project can be installed and updated easily
- ▶ **Certain third-party packaging and distribution tools are in widespread use**
 - We'll discuss several useful examples
- ▶ **Python documentation has relevant chapters**
 - Installing Python Modules
 - Distributing Python Modules

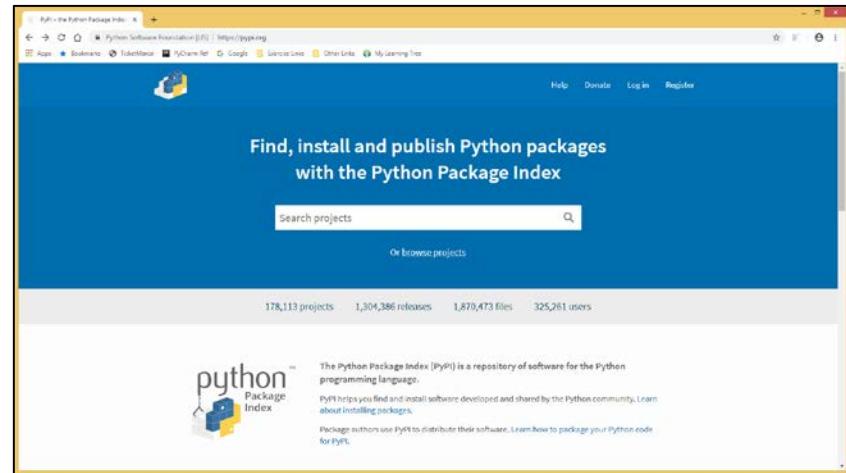


Key Terms for Packaging and Distribution

- ▶ **PyPI: Public repository of open source packages**
- ▶ **Python Packaging Authority (PyPA): Group responsible for maintenance of standard packaging tools**
 - Recommended reading: PyPA's *Python Packaging User Guide*
 - See <https://packaging.python.org/en/latest/>
- ▶ **distutils: Python's original build and distribution system**
 - Direct use of distutils is being phased out
- ▶ **setuptools: Drop-in replacement for distutils**
 - Recommended by PyPA as alternative to distutils
- ▶ **wheel: Adds bdist_wheel command to setuptools**
 - Produces “wheels” or “wheel files”
 - Cross-platform binary packaging format defined in PEP 491
 - Allows Python packages to be installed without being built locally even if the package includes binary extension

PyPI Repository

- PyPI repository has thousands of freely available Python modules
 - If it's Python and it's good, it's on PyPI
 - <http://pypi.python.org>
- Use PyPI's search facility to find Python modules
- A module's home page has links for downloading the module and documentation
- After initial setup, you usually install modules from PyPI using pip



Warning:
Packages on PyPI are *not* scanned for viruses or malicious code!

The pip Command

- ▶ **pip is the standard tool for installing Python modules**
 - By default, pip searches the global PyPI repository for modules
- ▶ **To install a new module, run pip install**

`pip install slothtest`
- ▶ **To upgrade an installed module and its dependencies, run pip install -U**

`pip install -U pytest`
- ▶ **pip can upgrade itself**
 - On *nix and Mac OS X, use pip to upgrade itself:

`pip install -U pip`
 - On Windows, run the pip module as a script:

`python -m pip install -U pip`
- ▶ **To see all installed modules: pip list**
- ▶ **For help: pip -h and pip install -h**

Installing From Wheel Files

- ▶ ***Wheel: Binary package format***
 - Can contain C/C++ extension pre-built for specific platform and architecture
 - Binary packages can be installed without building from source
 - Pure Python packages can also use wheel format
- ▶ **A wheel is a ZIP-format archive with .whl extension**
 - A wheel's filename includes Python version and target CPU architecture
- ▶ **Packages from wheels are usually installed to site-packages directory**
 - User may choose different directories for code, data, and docs
- ▶ **The wheel format is a more portable replacement for the older Egg format**
 - Install wheels with pip
 - Install eggs with easy_install
- ▶ **PyPA recommends using wheels for binary packaging**

Installing Packages

Do Now

1. Open a command prompt and change directory

```
> cd \crs1906\examples\ch07_examples
```

2. We'll install the restview package; first, confirm that it is not installed

```
> restview description.rst  
'restview' is not recognized ...
```

3. Install the restview package from PyPI or a source distribution

```
> pip install restview
```

Install from PyPI

```
> pip install ./restview-2.6.1.tar.gz
```

Install from local file

4. Confirm that restview command is installed (browser displays page)

```
> restview description.rst
```

5. Shut down restview server (use <Ctrl><C>)

Contents

- ▶ Installing and Updating Python Modules

Creating Virtual Python Environments

- ▶ Packaging and Distributing Modules and Applications
- ▶ Installing Packages from a Local Repository
- ▶ Hands-On Exercise 7.1



Virtual Environments

- ▶ ***Problem: Several versions of Python are in widespread use***
 - Popular Python packages may require different versions of Python
 - You may need to test your application with multiple Python versions
 - Python versions 2.6, 2.7, and 3+ are all in current use
- ▶ ***Solution: Create virtual Python environments***
 - Described in PEP 405
 - Each virtual environment is completely isolated from other environments
 - Each has its own Python interpreter, libraries, and third-party modules
 - You install required dependencies separately in each virtual environment
- ▶ ***Recommendations for creating virtual environments***
 - `venv` module (standard with Python 3.3+)
 - `virtualenv` module

venv Module

- ▶ **venv is the standard tool for managing Python virtual environments**
- ▶ **Copies Python interpreter and essential libraries to new directory**
 - Doesn't share libraries with other venv environments
- ▶ **Updates to global library don't affect virtual environments**
 - You can update global Python version without breaking existing applications
- ▶ **Updates to virtual environments don't affect global library**
 - Useful for testing new packages and new Python versions

Creating a Virtual Environment

- ▶ **To create new virtual environment:** `python -m venv venv-directory`
 - `venv` creates *venv-directory*
 - Copies Python interpreter to subdirectory for executables
 - *nix: *venv-directory/bin*
 - Windows: *venv-directory\Scripts*
 - Python 3.3: Doesn't install pip, setuptools, or wheel
 - Python 3.4+: Installs pip and setuptools (but not wheel)
- ▶ **venv creates a script named `activate`**
- ▶ **`activate` adds new Python interpreter's location to PATH**
 - Now `python` command starts interpreter in virtual environment's directory
 - To undo changes to PATH, run `deactivate`

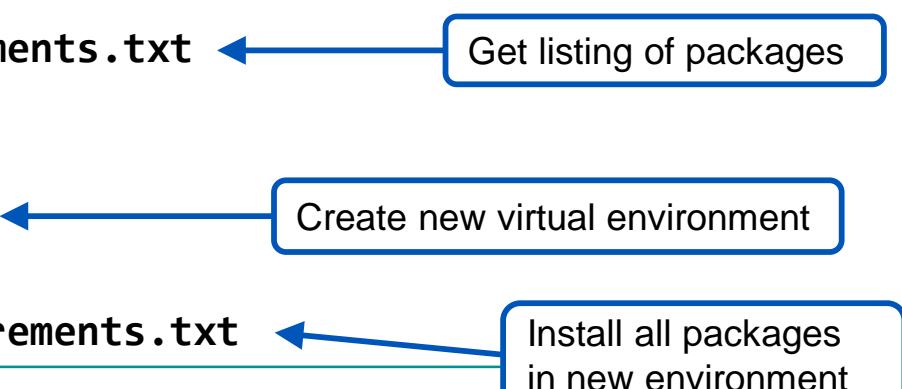
Working With Virtual Environments

- ▶ **New virtual environment is completely clean**
 - Contains no third-party packages that were added to site's Python installation
 - On *nix, --system-site-packages adds links to all system-wide packages
 - Requires OS support for symbolic links (not supported on Windows)

```
python -m venv --system-site-packages venv-directory
```

- ▶ **Virtual environment can't be moved or copied to another computer**
 - pip freeze prints a listing of all packages installed in current environment
 - Create a duplicate environment with pip install -r

```
venv\Scripts\activate
pip freeze > \temp\requirements.txt
deactivate
cd \new\parent\dir
python -m venv venv_copy
venv_copy\Scripts\activate
pip install -r \temp\requirements.txt
```



The diagram illustrates the process of creating a duplicate virtual environment. It shows three steps: 1. "Get listing of packages" (pip freeze > \temp\requirements.txt), 2. "Create new virtual environment" (python -m venv venv_copy), and 3. "Install all packages in new environment" (pip install -r \temp\requirements.txt). Arrows point from each step to its corresponding command.

Creating a Virtual Environment

Demo

► Example: Create Python 3.8 virtual environment on Windows

```
> python --version
```

```
Python 3.7.4
```

Currently using Python 3.7

```
> cd \crs1906\examples\ch07_examples
```

Create Python 3.8 virtual environment

```
> C:\Python\Python3.8\python -m venv py38_venv
```

```
> py38_venv\Scripts\activate
```

```
(py38_venv) > path
```

```
PATH=C:\crs1906\examples\ch07_examples\py38_venv\Scripts;...
```

```
(py38_venv) > python --version
```

```
Python 3.8.0
```

Now using Python 3.8

```
(py38_venv) > deactivate
```

```
> python --version
```

```
Python 3.7.4
```

Back to Python 3.7

Creating a Virtual Environment

1. From a command prompt, change to the example directory

```
cd \crs1906\examples\ch07_examples
```

2. Create a virtual environment

```
python -m venv new_venv
```

3. Display the new directory's content

- Note the minimal site-packages contents

```
dir new_venv
dir new_venv\Lib\site-packages
```

4. Activate the virtual environment

```
new_venv\Scripts\activate
```

virtualenv Module

- ▶ **virtualenv is the original tool for managing Python virtual environments**
 - Used for Python versions 3.3 and earlier
- ▶ **venv is based on virtualenv**
 - Operation of both tools is very similar
- ▶ **Installation: pip install virtualenv**
- ▶ **To create a virtual environment: virtualenv *venv-directory***

Contents

- ▶ Installing and Updating Python Modules
- ▶ Creating Virtual Python Environments

Packaging and Distributing Modules and Applications

- ▶ Installing Packages from a Local Repository
- ▶ Hands-On Exercise 7.1



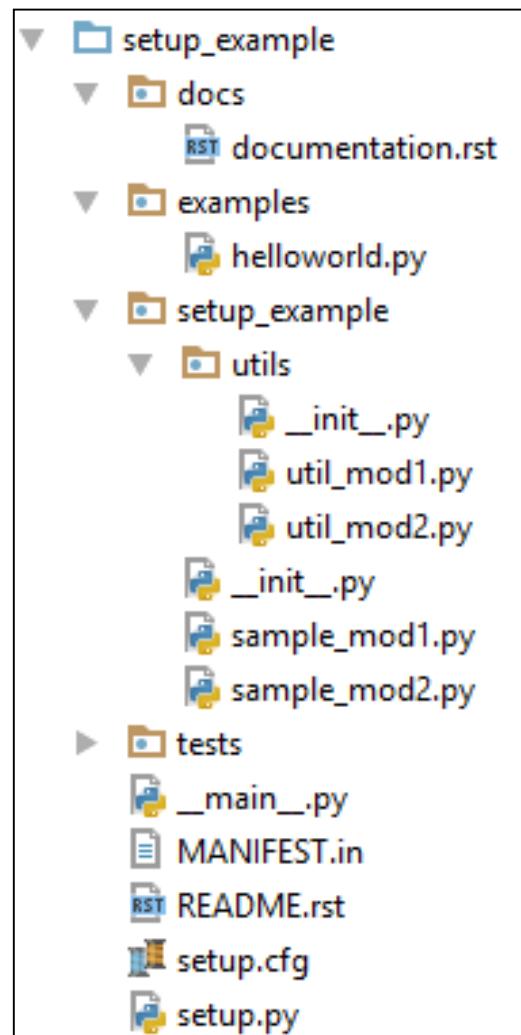
Packaging Your Project

- ▶ You often need to make your project available to other Python developers
 - Developers on your project
 - Developers on other projects within your organization
 - Developers on the web
- ▶ Your project should be easy to find, install, and update
- ▶ We'll discuss best practices for packaging and distribution
- ▶ By following these recommendations, you make your project easy to use
 - Others can find your project on PyPI or a local repository
 - Others can install and upgrade your project using setuptools and pip



Project Structure

- ▶ PyPA recommends certain initial files for projects
 - setup.py: Project build and installation script
 - README.rst: Overview of project
 - ReStructured Text (RST) format
 - <your package>: Top-level package with same name as project
- ▶ We'll discuss some of these files in detail
- ▶ Add other directories and files as required
 - Documentation
 - Examples
 - Unit tests



setup.py Script

- ▶ **Most important file for packaging your project is setup.py**
 - Stored in the root directory of your project
- ▶ **setup.py calls setuptools.setup() to configure project**
- ▶ **setup.py defines command-line interface for running packaging tasks**

Task	Command
Build source distribution	<code>python setup.py sdist</code>
Build wheel distribution	<code>python setup.py bdist_wheel</code>
Remove temporary build files	<code>python setup.py clean</code>
Install package in development mode	<code>python setup.py develop</code>
Install package	<code>python setup.py install</code>
Get general help	<code>python setup.py --help</code>
Get help for commands	<code>python setup.py --help-commands</code>

setup.py Script

setup_example/setup.py

```
from setuptools import setup, find_packages
```

```
setup(  
    name='setup_example',  
    version='1.0.0',  
    packages=find_packages(exclude=['tests*']),  
    description='Example module',  
    install_requires=['requests', 'Pillow']  
)
```

find_packages() discovers all Python packages in the project

install_requires lists dependent modules

setup.py Script

- ▶ **Arguments to setuptools.setup() set project's packaging attributes**
- ▶ **packages: List of all packages in the project**
 - `setuptools.find_packages()` searches project and returns list of all packages
 - `exclude` option lists directories to be ignored
- ▶ **install_requires: List of required packages**
 - When your project is installed, pip will install required packages, if necessary

Building and Installing a Wheel

Demo

```
> deactivate  
> cd \crs1906\examples\ch07_examples\setup_example
```

```
> python setup.py sdist bdist_wheel
```

Generate source distribution
and wheel distribution file

```
> dir dist
```

```
07/17/2015 ... setup_example-1.0.0-py3-none-any.whl
```

```
07/17/2015 ... setup_example-1.0.0.tar.gz
```

```
> cd dist
```

```
> python -m venv testenv
```

```
> testenv\Scripts\activate
```

```
> pip install setup_example-1.0.0-py3-none-any.whl
```

Install from wheel

```
...
```

```
Successfully installed setup-example-1.0.0
```

```
> dir testenv\Lib\site-packages\setup_example
```

```
...
```

```
07/17/2015 ... sample_mod1.py
```

```
07/17/2015 ... sample_mod2.py
```

```
07/17/2015 ... __init__.py
```

```
...
```

Package installed to venv's
site-packages

Verifying the Installation

Demo

```
> python
>>> from setup_example.sample_mod1 import func1
>>> func1()
In setup_example.sample_mod1.func1()
In setup_example.utils.util_mod1.util_func1()

> pip uninstall setup_example
Uninstalling setup-example-1.0.0:
Would remove:
...
Proceed (y/n)? y
Successfully uninstalled setup-example-1.0.0

> dir testenv\Lib\site-packages\setup_example
File Not Found

> deactivate
```

Import modules from
new package as usual

Installing to a Per-User Directory

Demo

- By default, packages are installed to a global site-packages directory
 - Add --user option to setup.py to install to user's private site-packages
 - User's site-package is automatically added to site.path

```
> cd ..  
> python setup.py install --user  
...  
Finished processing dependencies for setup-example==1.0.0  
  
> dir \Users\student\AppData\Roaming\Python\Python37\site-packages  
...  
01/15/2020 ... setup_example-1.0.0-py3.7.egg  
  
> python  
>>> import sys  
>>> sys.path  
['', ...  
'C:\Users\user\AppData\Roaming\Python\Python37\site-packages', ...]  
>>> from setup_example.sample_mod1 import func1  
>>> func1()  
>>> quit()
```

The diagram illustrates the process of installing a package to a per-user directory. It shows a sequence of terminal commands and their corresponding effects on the Python environment.

- Install module using setup.py:** A blue arrow points from the command `python setup.py install --user` to the resulting file `setup_example-1.0.0-py3.7.egg`.
- Package installed to user's private directory:** A blue arrow points from the file `setup_example-1.0.0-py3.7.egg` to the Python session, indicating it is located in the user's private site-packages directory.
- Egg file contains your source files plus installation metadata:** A blue arrow points from the file `setup_example-1.0.0-py3.7.egg` to the Python session, explaining its contents.
- User's site-packages added to sys.path:** A blue arrow points from the Python session back to the command `sys.path`, indicating the path has been updated.

Contents

- ▶ Installing and Updating Python Modules
- ▶ Creating Virtual Python Environments
- ▶ Packaging and Distributing Modules and Applications

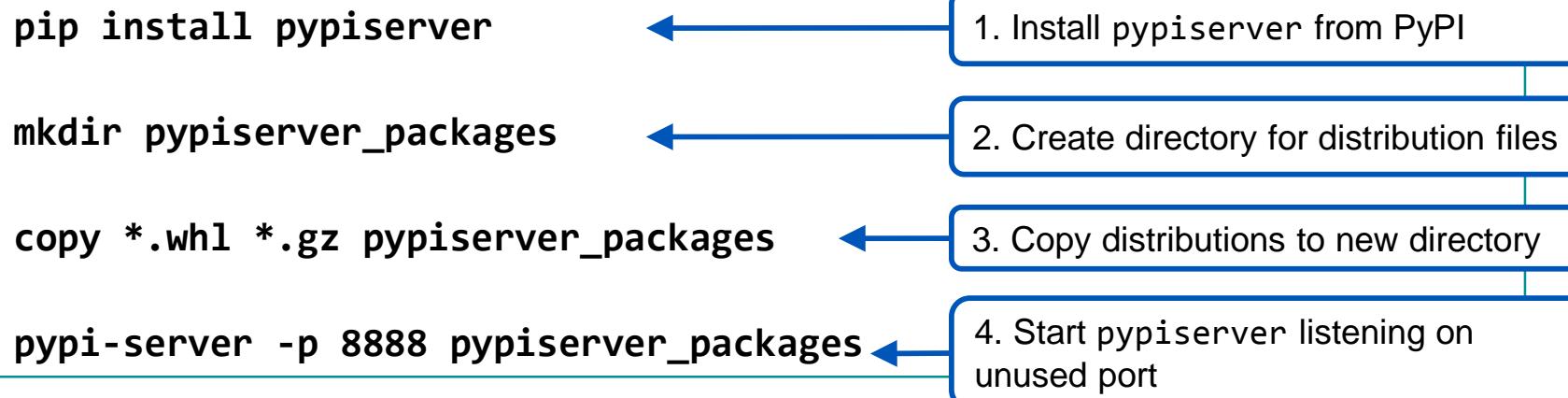
Installing Packages from a Local Repository

- ▶ Hands-On Exercise 7.1



Setting Up a Local Repository

- ▶ **Problem:** You want to reduce the risk of installing a harmful module
 - PyPI packages are not scanned for viruses or malicious code
- ▶ **Solution:** Set up a local repository of modules
- ▶ **pypiserver is a minimal PyPI-compatible server**
 - Can be used to upload and serve Python wheel and source distributions
 - Compatible with pip
 - Stores packages inside a regular directory
- ▶ Installation and configuration



Installing Packages From Local Repositories

- ▶ Clients specify local server as the index URL when installing packages

```
pip install --index-url http://localhost:8888/simple/ mypackage
```

- ▶ To configure pip to use local server by default, do one of the following

- Set INDEX_URL environment variable

```
set INDEX_URL=http://localhost:8888/simple/
```

- Add the following lines to *HOME_DIR/.pip/pip.conf*

```
[global]
index-url = http://localhost:8888/simple/
```

- ▶ Now clients can install with pip as usual

- If package is not available locally, pip install fails
- Ensures that only trusted modules can be installed

```
pip install mypackage
```

Contents

- ▶ **Installing and Updating Python Modules**
- ▶ **Creating Virtual Python Environments**
- ▶ **Packaging and Distributing Modules and Applications**
- ▶ **Installing Packages from a Local Repository**

Hands-On Exercise 7.1



Hands-On Exercise 7.1

In your Exercise Manual, please refer to
**Hands-On Exercise 7.1: Installation and
Distribution**



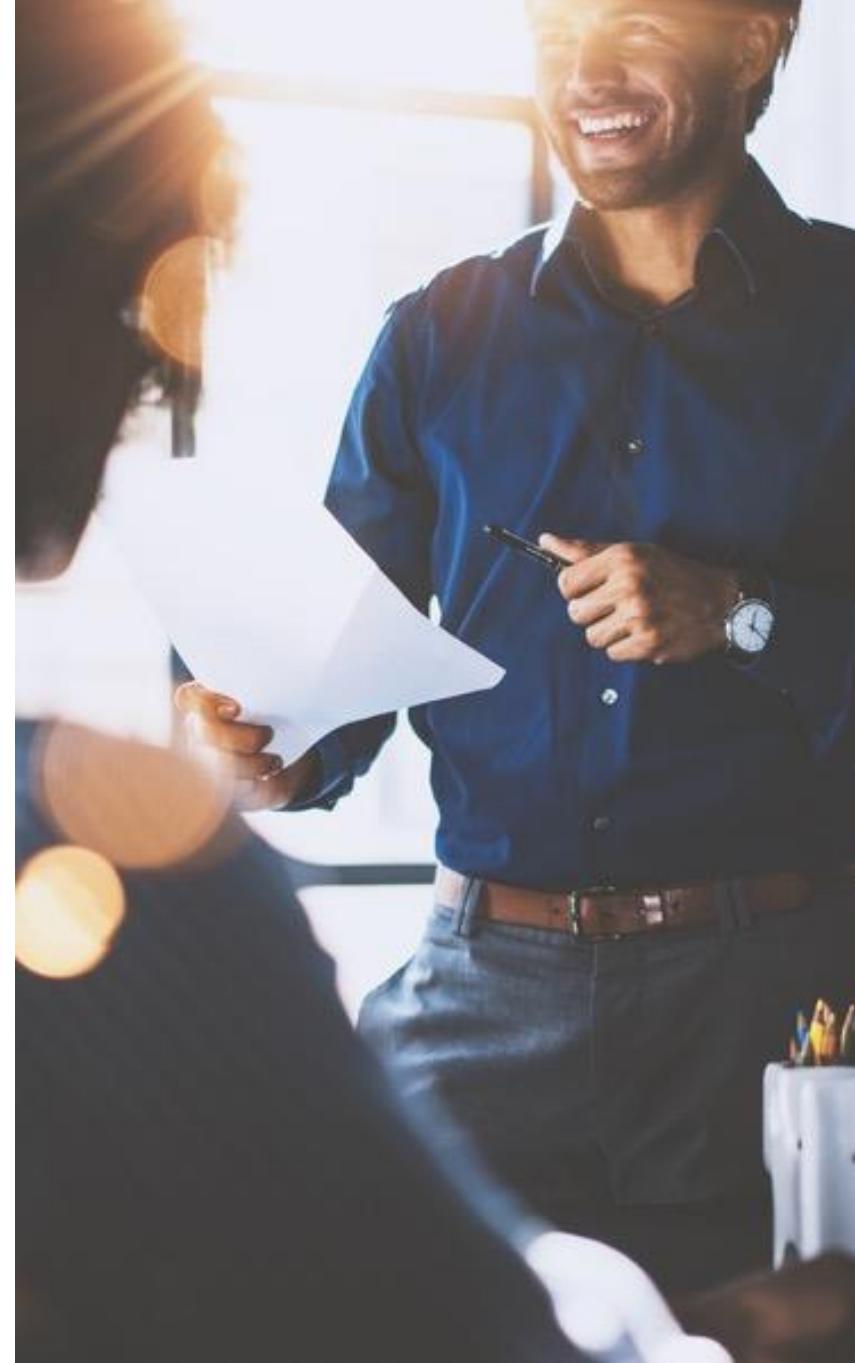
Best Practices: Module Installation and Distribution

- ▶ Use pip instead of easy_install to install modules
- ▶ Follow recommendations in Python Packaging User Guide for building distributions
- ▶ In your setup.py, import setuptools instead of distutils
- ▶ After building a distribution package, test that it installs correctly in a new virtual environment



Objectives

- ▶ **Install modules from the PyPI repository using pip**
- ▶ **Establish isolated Python environments with venv and virtualenv**
- ▶ **Package Python modules and applications for distribution**
- ▶ **Upload your Python modules to a local repository**

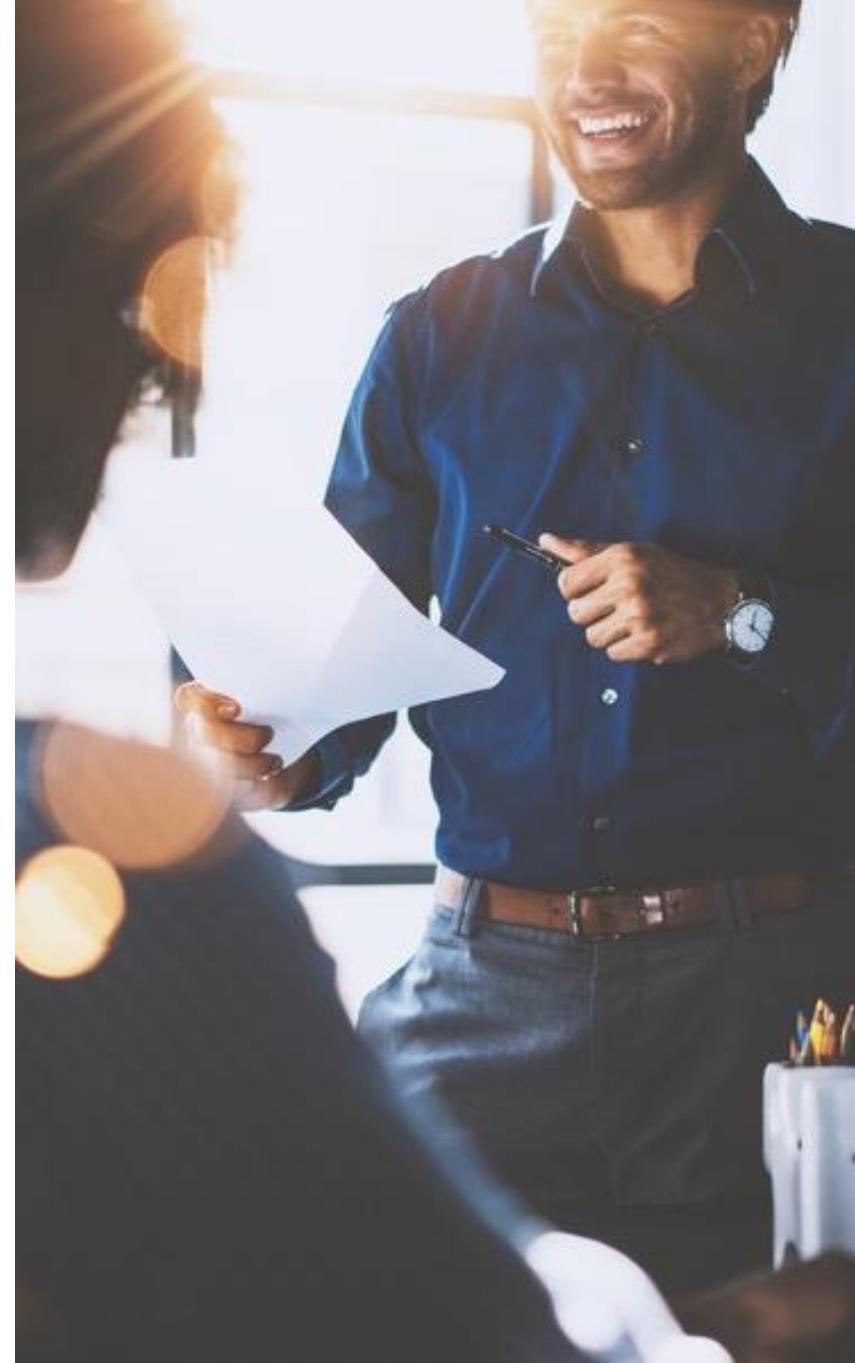


Chapter 8

Concurrent Execution

Objectives

- ▶ **Create and manage multiple threads of control with the Thread class**
- ▶ **Synchronize threads with locks**
- ▶ **Parallelize execution with Process pools and Executor classes**
- ▶ **Simplify parallel applications by using Queue objects**



Contents

Overview of Concurrency

- ▶ The threading Module
- ▶ Hands-On Exercise 8.1
- ▶ The multiprocessing Module
- ▶ The current.futures Module
- ▶ Hands-On Exercise 8.2



Concurrency and Parallelism in Python

- ▶ **Python supports *concurrency* and *parallelism***
 - Concurrency: Multiple tasks execute on a single CPU core
 - Parallelism: Multiple tasks execute simultaneously on multiple cores
- ▶ **Python interpreter supports *threads***
 - All threads share one CPU core using timeslicing
 - Python threads are never executed in parallel
 - Standard Python modules for threads
 - `threading`
 - `concurrent.futures`
- ▶ **Python interpreter supports parallelism using *processes***
 - Processes run simultaneously on multiple CPUs or multiple CPU cores
 - Standard Python modules for processes
 - `subprocess`
 - `multiprocessing`
 - `concurrent.futures`

Overview of Python Modules

- ▶ **subprocess module: provides easy access to external applications**
 - subprocess functions create heavy-weight processes managed by OS
 - Processes run executables (external commands or applications)
 - subprocess allows you to retrieve command output and exit status
 - Available in Python 2.4+
- ▶ **threading module: runs Python code in multiple threads of control**
 - Method calls create light-weight thread instances
 - Each thread executes a Python function
 - Python threads never run simultaneously, even with multiple CPUs or cores
 - Provide concurrency but not parallelism
 - Available in Python 2.2+

Overview of Python Modules

- ▶ **multiprocessing module: spawns processes managed by OS**
 - Processes are new instances of Python interpreter
 - Supports true parallelism, but has more overhead than threads
 - Supports shared queues, shared memory, process pools
 - multiprocessing module is powerful, but API is complex
 - Available in Python 2.6+
- ▶ **concurrent.futures: high-level interface for asynchronous execution**
 - Can easily access power of multiprocessing module
 - Added in Python 3.2, backported to Python 2.6 and 2.7

Contents

- ▶ Overview of Concurrency

The threading Module

- ▶ Hands-On Exercise 8.1
- ▶ The multiprocessing Module
- ▶ The current.futures Module
- ▶ Hands-On Exercise 8.2



Overview of threading Module

- ▶ **threading module supports light-weight threads**
- ▶ **Threads support concurrency, not parallelism, even on multi-core CPUs**
- ▶ **Python implements a Global Interpreter Lock (GIL)**
 - Some C components of standard Python implementation are not thread-safe
 - Python uses GIL to prevent multiple threads from executing simultaneously
- ▶ **A thread needs to acquire GIL before it can execute**
 - Thread won't run until owning thread releases GIL
- ▶ **Threads are good for *I/O-bound* applications**
 - I/O-bound: Most application time is spent waiting for I/O to complete
 - Result: Threads help performance of I/O-bound applications
- ▶ **Threads are bad for *CPU-bound* applications**
 - CPU-bound: Most application time is spent executing instructions
 - Python interpreter and OS must frequently switch between threads
 - Result: Threads hurt performance of CPU-bound applications

Simple Thread Example

- Your application code is usually executed by a single main thread
 - Main thread can create child Thread instances and call start() to run them
 - Main thread can wait for a thread to complete by calling thread's join()

```
thread_demo1.py
```

```
from threading import Thread
from zipfile import ZipFile, ZIP_DEFLATED
def zip_it(infile, outfile):
    with ZipFile(outfile, 'w', ZIP_DEFLATED) as f
        f.write(infile)
    print('Finished', infile)
background = Thread(target=zip_it,
                    args=('inventory.csv', 'inventory.zip'))
background.start()
print('Main thread running')
background.join()
print('Main thread waited until background was done.')
```

New thread will execute zip_it()

Create thread to execute zip_it()

Thread will pass args to zip_it()

Main thread starts child thread.
Child thread calls zip_it()

Main thread waits for child to complete

Threads Calling Methods

- ▶ Threads can call instance methods as well as plain functions
 - Thread constructor's target argument value is *ClassName.method_name*
 - First value in Thread constructor's args tuple is passed to method as self

thread_demo2.py

```
class Zipper:  
    def zip_it(self, infile, outfile): ←  
        f = ZipFile(outfile, 'w', ZIP_DEFLATED)  
        f.write(infile)  
        f.close()  
  
zip_instance = Zipper('my_zip') ←  
  
background = Thread(target=Zipper.zip_it,  
                    args=(zip_instance, 'inventory.csv', 'inventory.zip'))  
  
background.start() ←  
  
background.join() ←  
  
print('Main thread waited until background was done.')
```

zip_it() is a method in Zipper class

Create Zipper instance

target references zip_it() method

zip_instance is passed to zip_it() method as self

Creating Subclasses of Thread

- ▶ Instead of creating Thread instances directly, you can subclass Thread
 - Useful if task is more complex than a single function
 - When you call Thread.start(), Python calls Thread subclass's run()

thread_demo3.py

```
class AsyncZip(Thread):  
    Define subclass of Thread  
    def __init__(self, infile, outfile):  
        super().__init__()  
        Invoke Thread constructor  
        self.infile = infile  
        self.outfile = outfile  
  
    Override inherited run method  
  
    def run(self):  
        with ZipFile(self.outfile, 'w', ZIP_DEFLATED) as f  
            ... # same code as previous zip_it()  
  
background = AsyncZip('inventory.csv', 'inventory.zip')  
  
Create instance of Thread subclass  
  
background.start()  
Main thread starts child thread. Child thread calls run()  
  
print('Main thread running')  
background.join()  
Main thread waits for child thread to complete
```

Retrieving Values From Thread Instances

► Thread.run() does not return a value

- If child thread needs to return results, add attributes to your Thread subclass
- Main thread can retrieve results after joining child thread

thread_demo4.py

```
class DiskUsage(Thread):  
    ...  
    def run(self):  
        ...  
        self.results = ... # results of calculation  
  
disk_usage_thread = DiskUsage()  
disk_usage_thread.start()  
...  
disk_usage_thread.join()  
disk_usage = disk_usage_thread.results
```

Thread's run() method stores result of calculation in an attribute

Main thread gets child thread's results

Creating Multiple Threads

Do Now

1. Switch to PyCharm and open project ch08_examples
 - File | Open | C:\crs1906\examples\ch08_examples
2. Right-click yinyang_singlethreaded.py | Run '...'
 - The drawing is created by one thread
3. Open yinyang_multithreaded.py and examine main() method

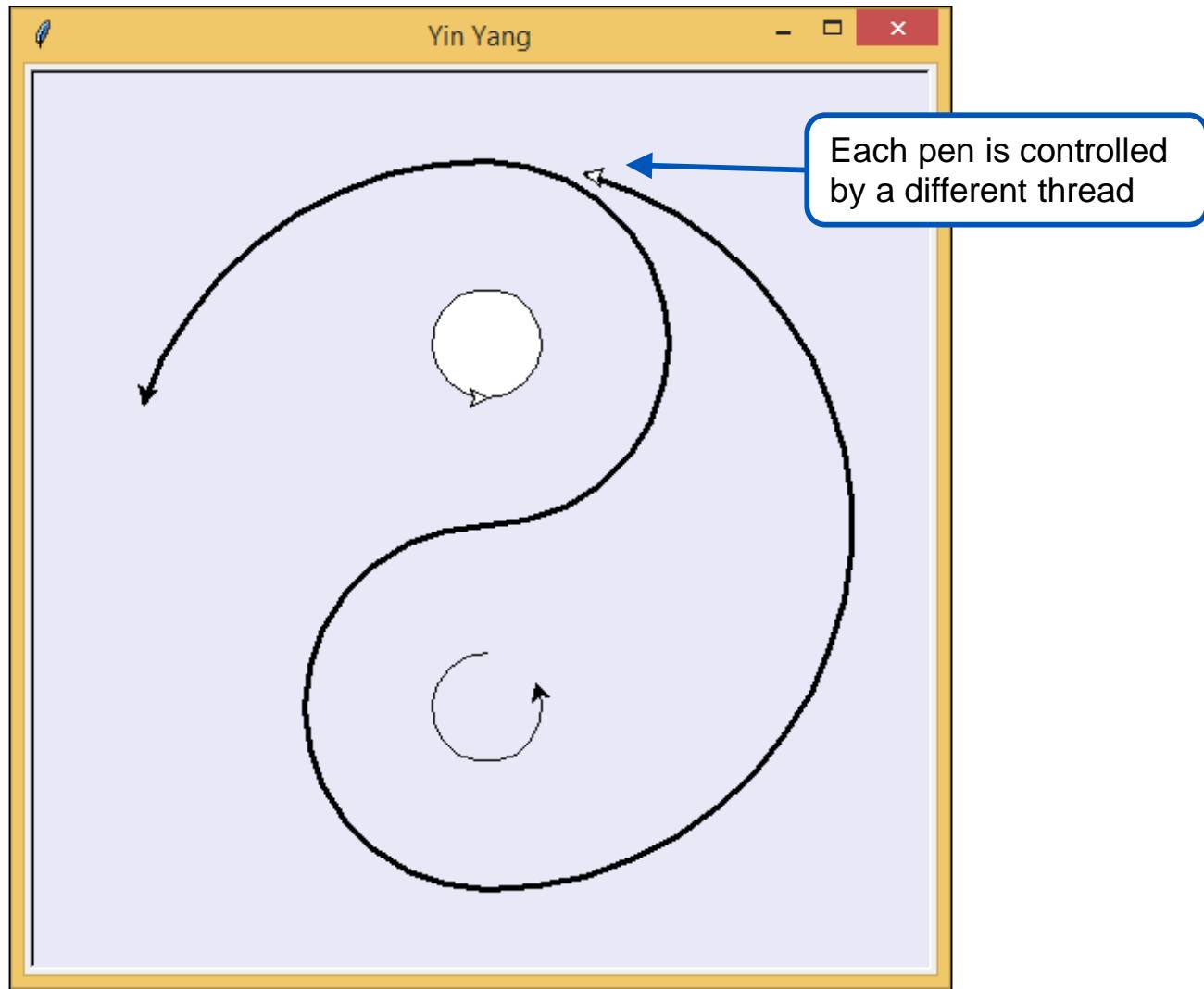
```
yin_thread = Thread(target=draw_yin)
yang_thread = Thread(target=draw_yang)
yin_thread.start()
yang_thread.start()
```

Main thread creates
two threads for drawing

4. Run yinyang_multithreaded.py
 - Note two pens drawing the image “simultaneously”
5. Open yinyang_multi_four_threads.py and examine main()
 - Creates four threads for drawing
6. Run yinyang_multi_four_threads.py

Creating Multiple Threads

Do Now



Race Conditions

- ▶ **Race condition:** multiple threads access shared resource simultaneously
 - Can result in corruption of data and incorrect results
- ▶ **Example: application samples data from five different sensors**
 - We define a Counter class to track total number of measurements so far
 - Code is in ch08_examples/thread_race.py

```
class Counter:  
    def __init__(self):  
        self.count = 0  
  
    def increment(self, offset=1):  
        self.count += offset  
  
class SampleSensors:  
    counter = Counter()  
  
    def worker(self, how_many):  
        for i in range(how_many):  
            self.read_from_sensor()  
            SampleSensors.counter.increment()  
  
    ... # continued on next slide
```

Class attribute: one instance of Counter will be shared by all threads

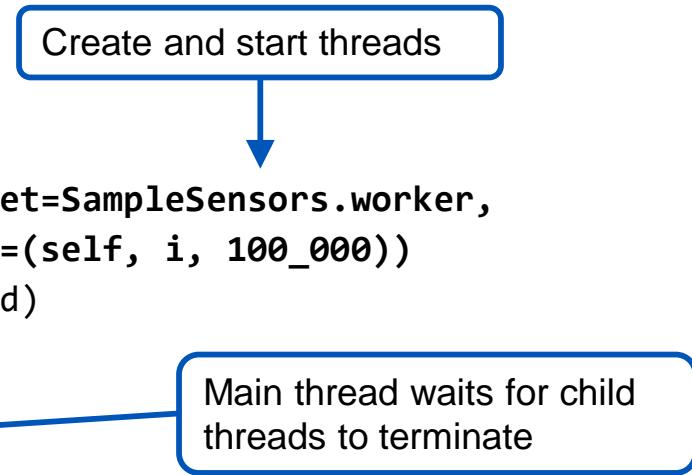
Each thread will run worker()

worker() updates shared Counter

Race Conditions

- Main program creates Counter instance and five threads

```
class SampleSensors:  
    counter = Counter()  
  
    def worker(self): ... # from previous slide  
  
    def sample_sensors(self):  
        threads = []  
        for i in range(5):  
            thread = Thread(target=SampleSensors.worker,  
                            args=(self, i, 100_000))  
            threads.append(thread)  
            thread.start()  
        for thread in threads:  
            thread.join()  
  
        print(f'Counter should be 500000, found {SampleSensors.counter.count}')
```



- Output

Counter should be 500000, found 499998

Oops! Counter value is wrong

Preventing Race Conditions

- ▶ Problem is caused by two threads updating count at the same time
 - If a thread is interrupted in the middle of an update, the result may be wrong
- ▶ Critical sections of code must be synchronized among threads
 - The increment operation in Counter.increment() looks *atomic*

```
class Counter:  
    def increment(self, offset=1):  
        self.count += offset
```

- But the generated byte code requires several separate operations
- ```
1. value = getattr(counter, 'count')
2. result = value + offset
3. setattr(counter, 'count', result)
```

Increment operation requires several steps, may be interrupted


- Race condition may result
    - Thread A runs worker() but is interrupted after getting value at Step 1
    - Thread B runs worker() from beginning to end, updating shared counter
    - Thread A resumes at Step 2 and overwrites Thread B's update

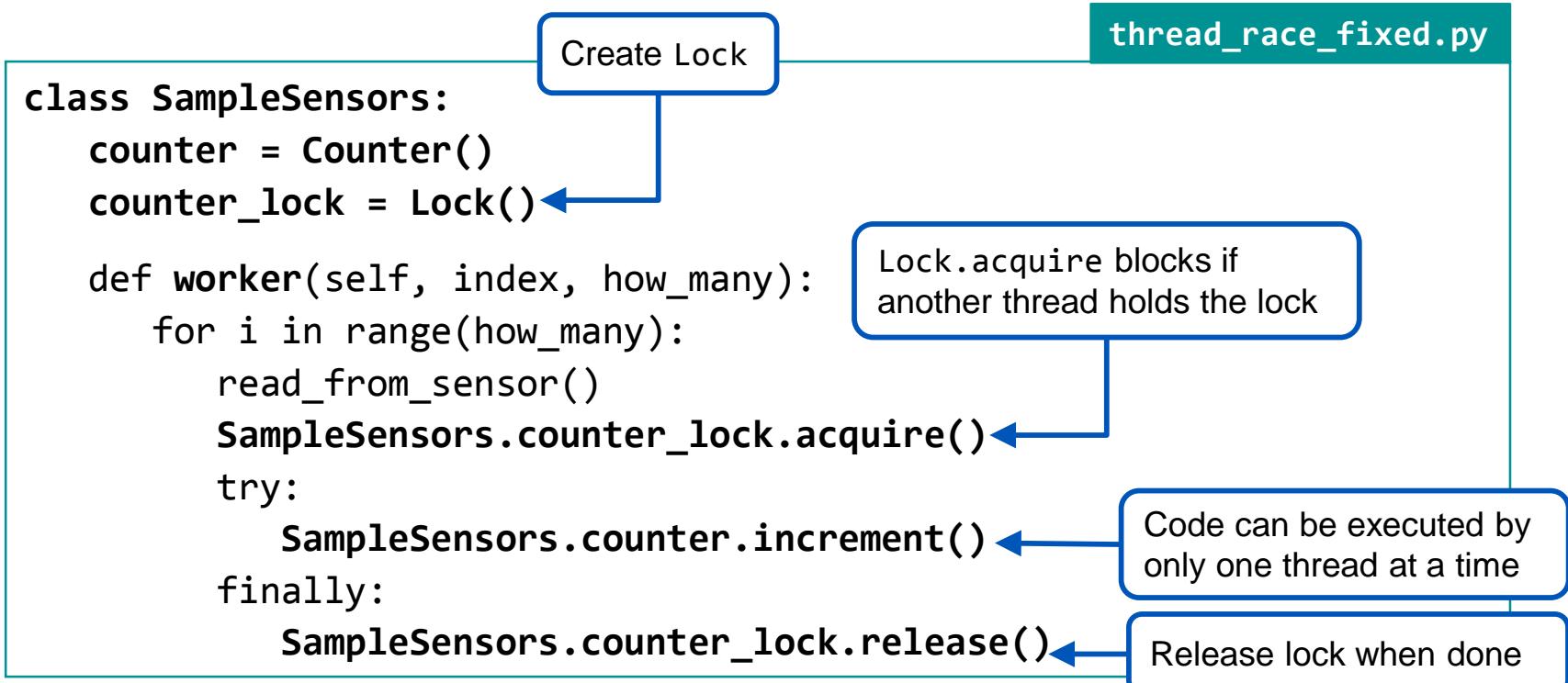
# Preventing Race Conditions

---

- ▶ **Solution to race condition problem: `threading.Lock` class**
- ▶ **Lock instance can be held by only one thread at a time**
  - Thread A acquires lock and starts executing code
  - Thread B attempts to acquire lock before executing code
  - If Thread A is still holding the lock, Python suspends Thread B
  - When Thread A releases the lock, Thread B can acquire it and continue
- ▶ **Lock class supports synchronization of threads**
  - Implements mutual exclusion (*mutex*)

# Preventing Race Conditions

- **Modified code creates a Lock object before incrementing counter**
  - Threads must acquire the Lock before executing the critical section



- **Result after adding synchronization**

Counter value is correct

Counter should be 500000, found 500000

# Python Idiom: Managing Locks

- ▶ **Lock class implements the context manager protocol**
  - Python idiom: Simplify code using the with statement
    - Lock.acquire is called at entry to with
    - Lock.release is called at exit to with (even if exception occurs)
  - No try/finally required

```
def worker(self, index, how_many):
 for i in range(how_many):
 read_from_sensor()
 with SampleSensors.counter_lock:
 SampleSensors.counter.increment()
```

with statement  
simplifies code

# Contents

---

- ▶ Overview of Concurrency
- ▶ The threading Module

## Hands-On Exercise 8.1

- ▶ The multiprocessing Module
- ▶ The current.futures Module
- ▶ Hands-On Exercise 8.2



## Hands-On Exercise 8.1

In your Exercise Manual, please refer to  
**Hands-On Exercise 8.1: Concurrency**



# Contents

---

- ▶ Overview of Concurrency
- ▶ The threading Module
- ▶ Hands-On Exercise 8.1

## The multiprocessing Module

- ▶ The current.futures Module
- ▶ Hands-On Exercise 8.2



# Parallel Processing

- ▶ Certain types of applications are computationally intensive
  - Image processing
  - Big Data analysis
  - Weather prediction
  - Simulations: Scientific, medical, financial
- ▶ These applications are typically CPU-bound
- ▶ Parallel processing often helps performance of CPU-bound applications
  - Multiple CPUs or CPU cores perform parts of the computation simultaneously
- ▶ Python supports parallel processing with the `multiprocessing` module



# The multiprocessing Module

---

- ▶ **The multiprocessing module provides true parallel execution**
  - Manages processes at the OS level, like the subprocess module
  - But processes created with multiprocessing can run any Python code
  - Not restricted by the GIL
- ▶ **Process run a new instance of the Python interpreter**
  - Start up overhead can be significant
  - Processes can be pooled to reduce startup overhead
- ▶ **Processes require synchronization**
  - Processes can use high-level synchronization techniques
    - Synchronized queues
    - Pipes: Output of one process becomes input to next process
  - multiprocessing also defines Lock classes for low-level synchronization

# Creating Process Instances

## ► Example: zip a file using a Process

- Child process will import current module
- Add test to ensure that main code is not executed when imported by child

multiprocessing\_demo1.py

```
from multiprocessing import Process

def zip_it(infile, outfile):
 with ZipFile(outfile, 'w', ZIP_DEFLATED) as f:
 f.write(infile)
 print('Finished', infile)

if __name__ == "__main__":
 child_process = Process(target=zip_it,
 args=('inventory.csv', 'inventory.zip'))
 child_process.start()
 print('Main process continues')

 child_process.join()
 print('Main process waited for child')
```

Same zip\_it() function as in Thread demo

Test protects code from recursive execution when imported

Create Process to run zip\_it()

Process class has same API as Thread class

# Subclassing Process

## ► You can define subclasses of Process

- Use same technique as defining subclasses of Thread
- When you call `Process.start`, new Python interpreter calls `run` method

`multiprocessing_demo2.py`

```
class AsyncZip(Process):
 def __init__(self, infile, outfile): ← Define Process subclass
 super().__init__()
 self.infile = infile
 self.outfile = outfile

 def run(self): ← Override inherited run method
 with ZipFile(self.outfile, 'w', ZIP_DEFLATED) as f
 ... # same code as before

if __name__ == '__main__':
 child_process = AsyncZip('inventory.csv', 'inventory.zip') ← Create and start Process subclass instance
 child_process.start()
 print('Main process continues to run in foreground')
 child_process.join()
 print('Main process waited until child_process was done.')

```

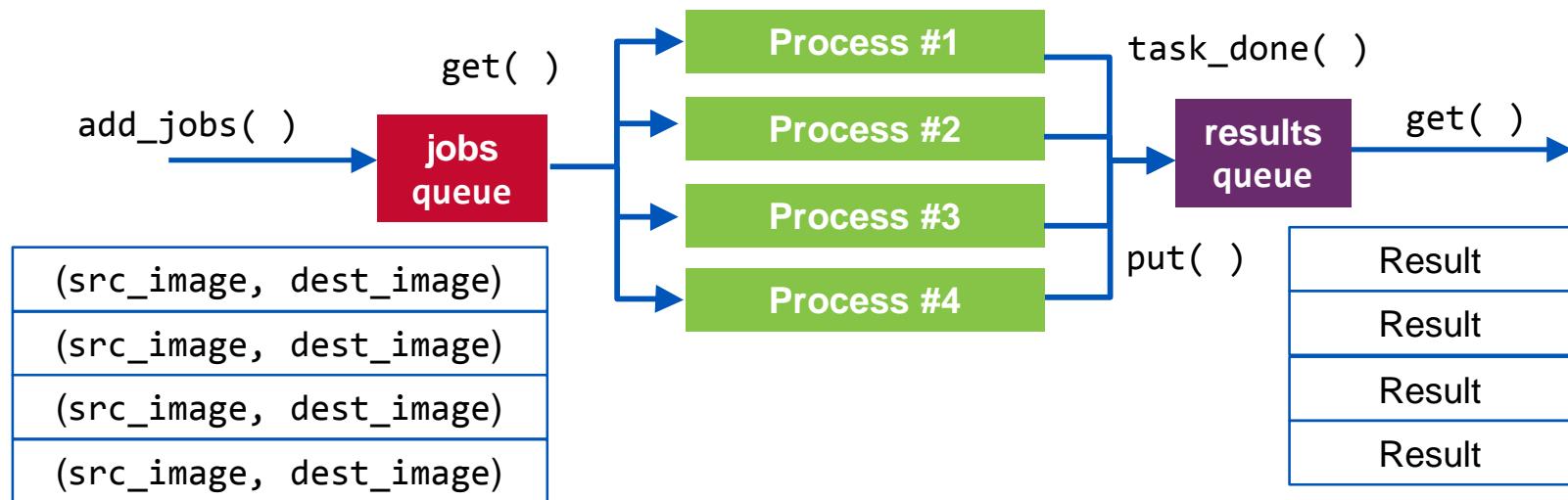
# Synchronized Queues

---

- ▶ **Common use case: multiple processes handle data as it arrives**
  - Must ensure two processes don't try to handle the same data
- ▶ **multiprocessing.Lock can synchronize processes**
  - But synchronizing with locks can be error prone and difficult to test
  - Acquiring and releasing locks may lead to deadlock conditions
- ▶ **Solution: multiprocessing.Queue class**
  - One process adds data to Queue, other processes get data from Queue
  - Queue class implements locking internally
    - Processes reading data from Queue won't conflict
- ▶ **Queues are best choice for inter-process communication and coordination**
  - Applications are easier to design, more readable, more reliable
- ▶ **Example: parallel RSS feed reader**
  - Multiple feeds can be downloaded concurrently by multiple processes
  - Each process adds its feed to a shared Queue

# Synchronizing Processes With Queues

- ▶ **multiprocessing.Queue and JoinableQueue allow high-level synchronization**
  - Processes can be synchronized without explicit locking
  - Code is less prone to deadlocks and other synchronization issues
- ▶ **Example: CPU-bound parallel image scaling application**
  - Application creates processes first, then starts adding jobs to jobs queue
  - Process gets a job from job queue, then puts result on results queue



# Parallel Image Scaling Application With Queues

```
def main():
 scale(size, smooth, src_dir, dest_dir, num_procs)

def scale(size, smooth, src_dir, dest_dir, num_procs):
 jobs = multiprocessing.JoinableQueue()
 results = multiprocessing.Queue()
 create_processes(size, smooth, jobs, results, num_procs)
 for src_image, dest_image in get_jobs(src_dir, dest_dir):
 jobs.put((src_image, dest_image))
 jobs.join()
 while not results.empty():
 result = results.get()
 ... # use result

def create_processes(size, smooth, jobs, results, num_procs):
 for _ in range(num_procs):
 process = Process(target=worker,
 args=(size, smooth, jobs, results))
 process.daemon = True
 process.start()

continued on next slide...
```

scale() args are from command line

Create input and output queues

Wait for input queue to empty

Get results from output queue

Create and start Process instance

Create daemon process

imagescale-q-m.py

# Parallel Image Scaling Application

```
def worker(size, smooth, jobs, results):
 while True:
 try:
 src_image, dest_image = jobs.get() ← Function executed by processes
 result = scale_one(size, smooth, src_image, dest_image)
 results.put(result) ← Process waits for input
 finally:
 jobs.task_done() ← Do the work and add result to output queue
 Tell input queue this input
 has been processed

def get_jobs(src_dir, dest_dir):
 src_and_dest = []
 for src_file_name in os.listdir(src_dir): ← Get the file path for each
 source and destination file
 ...
 src_and_dest.append((src_file_path, dest_file_path)) ← Add a tuple to the list
 return src_and_dest

def scale_one(size, smooth, src_image, dest_image):
 ... # perform scaling operation
 return Result(...)
```

# Daemon Processes

- ▶ **Interpreter exits when all remaining processes are *daemon* processes**
  - Daemon processes do background work for Python interpreter
  - Example: garbage collector
- ▶ **By default, processes you create are not daemon processes**
  - Interpreter won't exit until all non-daemon processes complete
- ▶ **To create a daemon process, set its `daemon` attribute before starting it**

```
process = Process(target=worker, args=...)
process.daemon = True ←
process.start()
```

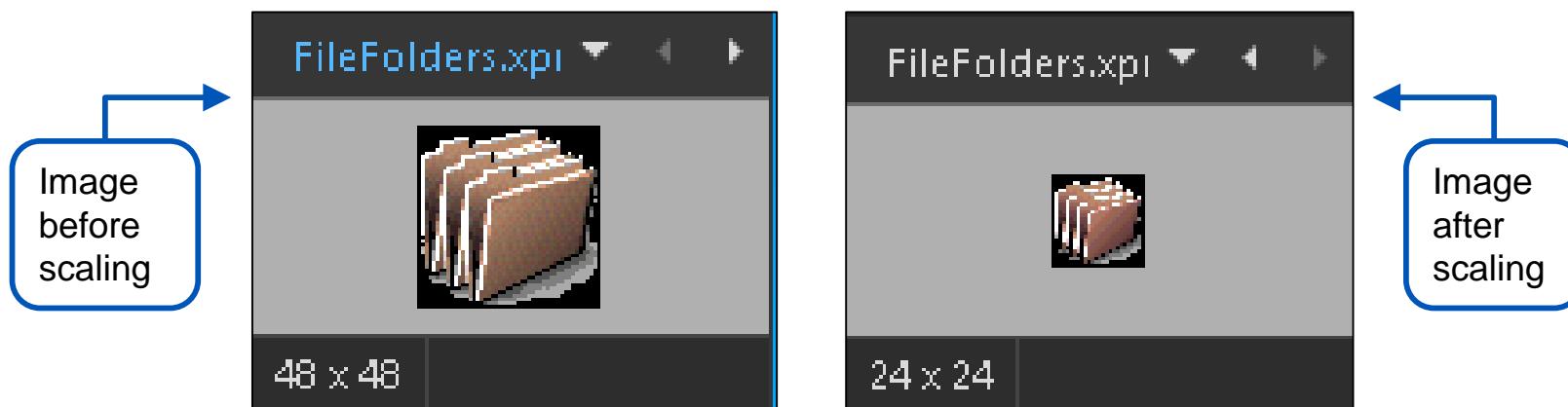
Daemon process doesn't prevent  
interpreter from exiting

# Execute Parallel Image Scaling Application

Do Now

## 1. Switch to a command prompt; execute the following commands

```
> cd \crs1906\examples\ch08_examples
> dir imagescale_in\FileFolders.xpm
07/27/2019 10:01 PM 11,972 FileFolders.xpm
> del imagescale_out*.*
> python imagescale-q-m.py imagescale_in imagescale_out
copied 2 scaled 37 using 2 processes
> dir imagescale_out\FileFolders.xpm
10/08/2019 03:54 PM 1,226 FileFolders.xpm
> imagescale_in\FileFolders.xpm
> imagescale_out\FileFolders.xpm
```



# Contents

---

- ▶ Overview of Concurrency
- ▶ The threading Module
- ▶ Hands-On Exercise 8.1
- ▶ The multiprocessing Module

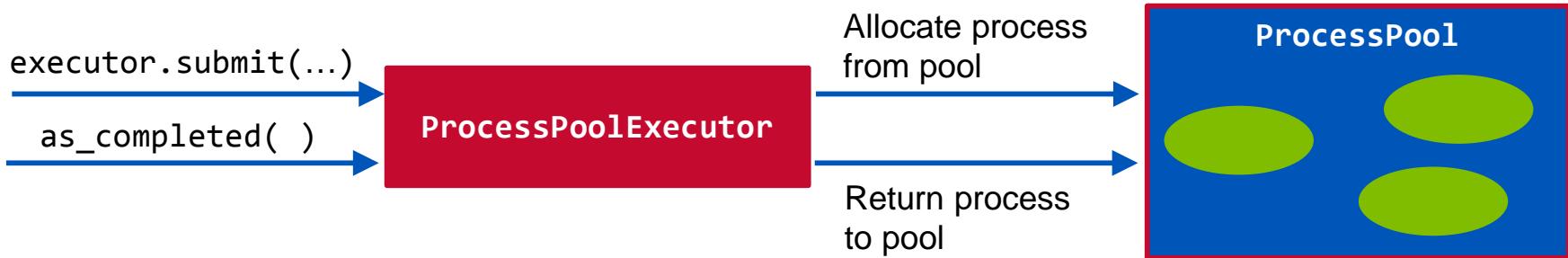
## The `current.futures` Module

- ▶ Hands-On Exercise 8.2



# concurrent.futures Module

- ▶ concurrent.futures has a high-level API for threads and processes
  - Advantage: you write less code and less explicit synchronization
- ▶ Use Executor instances to create and execute threads and processes
  - ThreadPoolExecutor, ProcessPoolExecutor
  - Executor instance maintains a *pool* of threads or processes
    - When task is submitted, Executor allocates a Process from pool
    - When task is complete, its Process instance is returned to pool
    - Next task can reuse existing Process from pool
- ▶ Executor.submit() method executes code concurrently
  - Returns an instance of Future
  - Use Future to check status, get result, or cancel concurrent execution



# Image Scaling Application Version 2

## ► Example: image scaling application using ProcessPoolExecutor

- No queues or locks; all process synchronization is done with Futures
- ProcessPoolExecutor is a context manager, can use with statement
  - At end of with, Executor.shutdown() is called automatically
  - Executor waits for all pending futures to complete executing

imagescale-m.py

```
from concurrent.futures import ProcessPoolExecutor

def scale(size, smooth, src_dir, dest_dir, num_procs):
 futures = set() # create empty set
 with ProcessPoolExecutor(max_workers=num_procs) as executor:
 for src_image, dest_image in get_jobs(src_dir, dest_dir):
 future = executor.submit(scale_one, size, smooth,
 src_image, dest_image)
 futures.add(future)

 summary = wait_for(futures)
 ...
continued on next slide...
```

Create Executor

Get a process from the pool and execute function scale\_one

# Image Scaling Application Version 2

```
def wait_for(futures):
 ...
 try:
 for future in concurrent.futures.as_completed(futures):
 err = future.exception() ← Get exception, if any
 if err is None:
 result = future.result() ← Get return value of
 ...
 except KeyboardInterrupt: # Ctrl-C
 ...
 for future in futures:
 future.cancel()
 return ... # summary of number of images copied, etc.

def scale_one(size, smooth, src_image, dest_image):

 return Result(...) ← future.result()
 returns this value
```

Returns iterator that yields Futures as processes complete

Get exception, if any

Get return value of scale\_one()

future.result()  
returns this value

# Mapping Collections Asynchronously

## ► Common use case: asynchronously process a collection of data

- Executor.map() spawns a thread or process for each item in the collection
- Returns iterator over function return values, not Futures
- Exception from function is raised when retrieving result

executor\_map\_demo.py

```
def ftp_get_file(args):
 site, file, user, pw = args
 ...
 return args.site, args.file

if __name__ == '__main__':
 ftp_args = [
 ('ftp.kernel.org', 'flock-2.0.tar.gz',
 'anonymous', 'mike@wxyz.me'), ...]
 with ThreadPoolExecutor(max_workers=20) as executor:
 results = executor.map(ftp_get_file, ftp_args, timeout=5)
 try:
 for site, file in results:
 print('Got', file, site)
 except Exception as e:
 print("Exception: " + str(e))
```

The diagram illustrates the execution flow of the `executor_map_demo.py` script. It starts with the definition of `ftp_get_file`, which takes arguments `site`, `file`, `user`, and `pw`. A callout notes that any exception in this function is raised when iterating to its result. The script then defines a list of tuples `ftp_args` containing pairs of `(site, file)` and `(user, pw)`. A callout indicates that this is a 'List of tuples'. Following this, a `with ThreadPoolExecutor(max_workers=20)` block is entered. Inside, the `executor.map` method is called with `ftp_get_file`, `ftp_args`, and `timeout=5`. A callout notes that this 'Spawns threads to execute `ftp_get_file()`'. The resulting iterator `results` is then iterated over using a `for` loop. A callout notes that this 'Iterates over results in order of function calls'. Finally, an `except` block handles any exceptions, printing them to the console.

# Contents

---

- ▶ Overview of Concurrency
- ▶ The threading Module
- ▶ Hands-On Exercise 8.1
- ▶ The multiprocessing Module
- ▶ The current.futures Module

## Hands-On Exercise 8.2



## Hands-On Exercise 8.2

In your Exercise Manual, please refer to  
**Hands-On Exercise 8.2: Multiprocessing**



# Design Considerations

---

- ▶ **Some applications fit naturally into multithreaded designs**
  - Example: web server needs to handle many requests simultaneously
  - Example: web client needs to download many resources simultaneously
- ▶ **However, applications may use multiprocessing strictly for performance**
  - Often requires drastic redesign of application
  - Multiprocessing applications are difficult to design, write, test, and debug
- ▶ **If performance is highest priority, multiprocessing may not be best choice**
  - Alternatives to multiprocessing for improved performance
    - Use a different interpreter or a compiler
      - PyPy: optimized Python interpreter
      - Numba: module that compiles Python code to optimized machine code
    - Convert Python code to C-language extension modules using Cython
    - Rewrite critical sections in C/C++

# Best Practices: Concurrency

---

- ▶ **Use concurrent.futures**
  - Adds high-level support for both threads and processes
- ▶ **For CPU-bound applications, use processes**
- ▶ **For I/O-bound applications, use threads**
- ▶ **Avoid using shared data**



# Best Practices: Concurrency

---

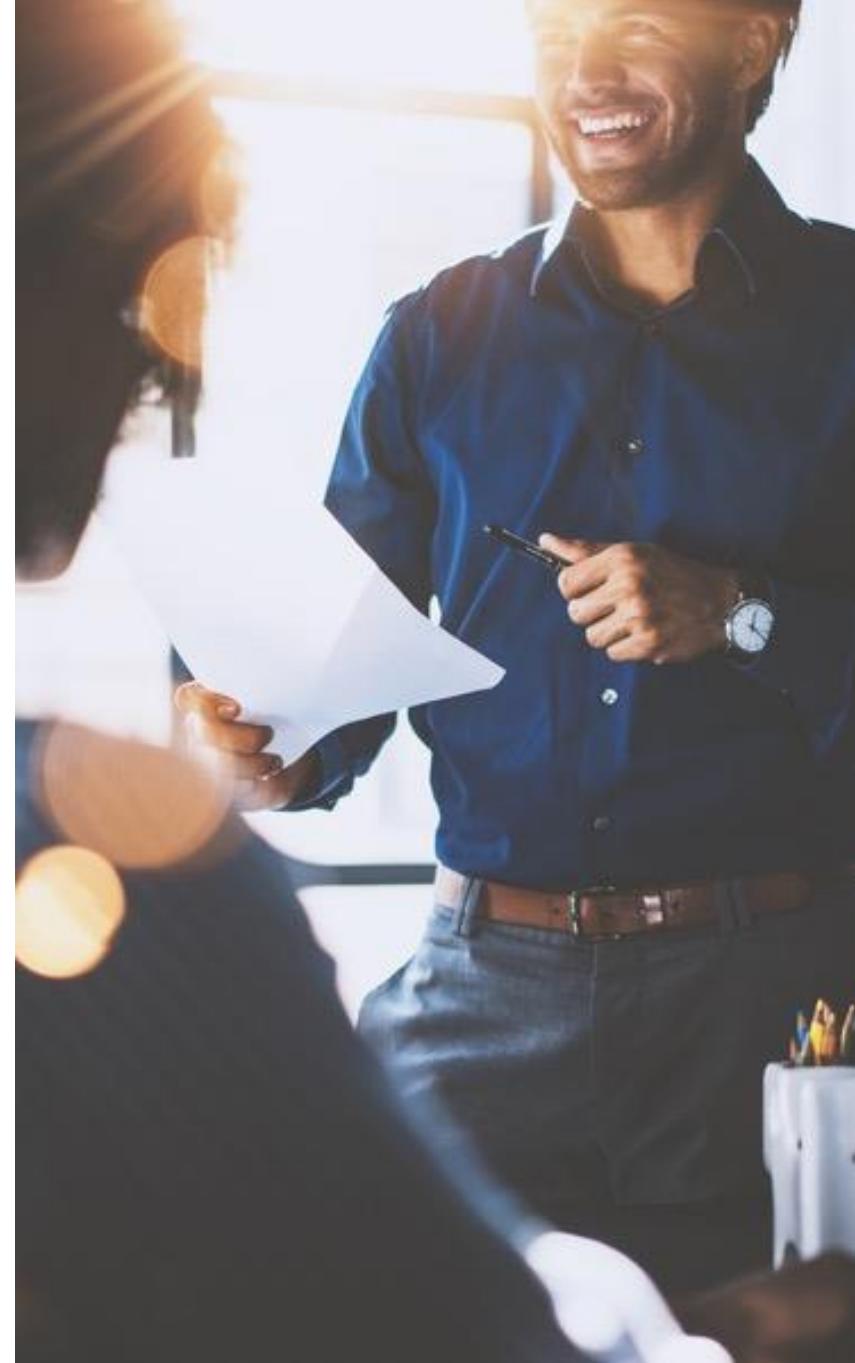
- ▶ If you must share data, do one of the following (in order of preference)
  1. Share data as read-only immutable types or deep copies
  2. Use a thread-safe/process-safe type
    - Processes: `multiprocessing.Queue` and `JoinableQueue`
    - Threads: `queue.Queue`
  3. Implement safe read-write access
    - Threads: Use `threading.Lock` and related classes
    - Processes: Use `multiprocessing.Lock` and related classes



# Objectives

---

- ▶ **Create and manage multiple threads of control with the Thread class**
- ▶ **Synchronize threads with locks**
- ▶ **Parallelize execution with Process pools and Executor classes**
- ▶ **Simplify parallel applications by using Queue objects**



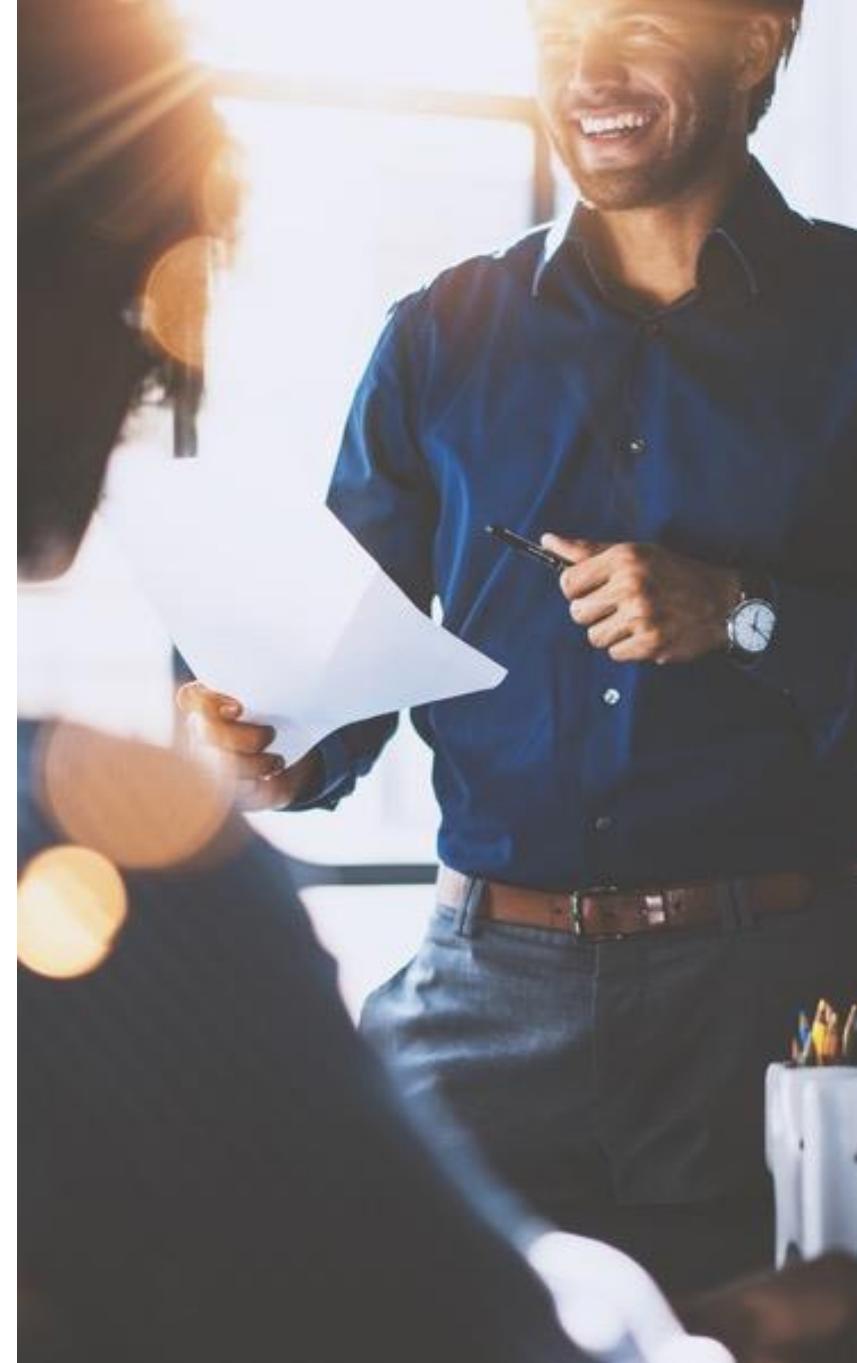
# Chapter 9

# Interfacing With REST Web Services and Clients

# Objectives

---

- ▶ Build a Python implementation of a REST web service
- ▶ Develop a REST web service that supports Ajax clients with JSON data
- ▶ Write Python clients that send requests to a REST web service
- ▶ Read and write JSON data in REST clients



# Contents

---

## REST Web Services

- ▶ JSON and Ajax
- ▶ Implementing a REST Service
- ▶ REST Clients
- ▶ Hands-On Exercise 9.1



# Web Services

---

- ▶ **Scenario: Our application needs to communicate with other applications**
  - Easy to implement if we control all applications
    - We write all of the applications in Python and run them on our network
- ▶ **But what if applications are owned by customers or business partners?**
  - Our application must be able to communicate via the Internet
  - Customer's software may be written in a different language
    - Java, C#, C++, etc.
  - Business partner's host may have a different OS
    - \*nix, Mac OS, Windows, etc.
- ▶ **Solution: Web services: Software systems that support *interoperable* network interaction**
  - Messages are formatted as XML, JSON, or other standard formats
  - Messages sent using HTTP or other web-related protocols
- ▶ **Interoperability is the biggest advantage**
  - Applications may be implemented with different languages and platforms
- ▶ **Implementation is simple and inexpensive**

# REST Web Services

---

- ▶ **Representational state transfer (REST) defines a type of web service**
- ▶ **To interact with a REST service, client sends HTTP request**
  - Service maps HTTP “verbs” (GET, POST, PUT, etc.) to different operations
  - GET may mean “fetch resource,” POST may mean “add resource,” etc.
- ▶ **REST service sends data to client in body of HTTP response**
  - REST is very flexible: Data can be XML, JSON, plain text, HTML
- ▶ **Services and clients can be written in any language on any OS**
  - Only requirement is support for HTTP requests
- ▶ **REST services are widely used**
  - Public REST APIs: Google Search, Amazon S3, Twitter, Facebook
  - Server-side support for Ajax web applications and mobile applications
- ▶ **Services can be tested with browser-based tools**
  - Chrome’s Postman extension, Firefox’s RESTClient add-on

# Example REST Service

---

- ▶ Building a REST service requires choosing HTTP methods and URLs
- ▶ Assign different meanings to different HTTP methods
  - GET: Retrieve resource
  - POST: Create resource
  - PUT: Update resource
  - DELETE: Delete resource
- ▶ Design URLs for requests
  - One HTTP method may respond differently to different URL patterns
- ▶ Example: REST to-do task service: manages your to-do list
  - GET `http://localhost:5000/todo/api/v1.0/tasks`
    - Returns all tasks
  - GET `http://localhost:5000/todo/api/v1.0/tasks/1342`
    - Returns task with ID 1342
- ▶ Choose format of message bodies: XML, JSON, or other format
  - We'll choose JSON

# A REST To-Do Service

## ► Sample HTTP requests and responses for to-do service

- GET returns all to-do tasks

```
GET http://localhost:5000/todo/api/v1.0/tasks
```

- GET with task ID responds with one task

```
GET http://localhost:5000/todo/api/v1.0/tasks/51
```

- POST inserts task: Request body contains title, description, and 'done' flag

```
POST http://localhost:5000/todo/api/v1.0/tasks
```

Request  
body

```
{"title": "Eat", "description": "Lunch", "done": false}
```

- PUT updates task: URL references task, request body has update values

```
PUT http://localhost:5000/todo/api/v1.0/tasks/51
```

Request  
body

```
{"done": true}
```

- DELETE deletes task: URL references task

```
DELETE http://localhost:5000/todo/api/v1.0/tasks/51
```

# Testing REST Services

Do Now

1. Open a command prompt and start the REST server:

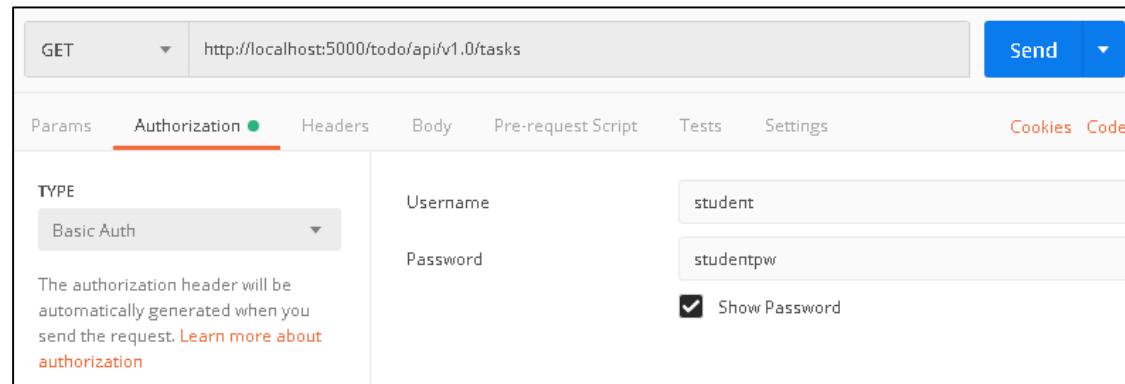
```
cd \crs1906\examples\ch09_examples\todo-api
python rest_server_json.py
```

2. Double-click the Postman icon on desktop
3. Select the GET method
4. Enter this URL: <http://localhost:5000/todo/api/v1.0/tasks>

5. Click Authorization tab

6. Select type Basic Auth
  - Username: student
  - Password: studentpw

7. Click Send



# Response From GET Request

- ▶ Note that client and service are completely decoupled
  - Client and service can be implemented in different language and OS

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:5000/todo/api/v1.0/tasks
- Auth Type:** Basic Auth (selected)
- Username:** student
- Password:** studentpw (with Show Password checked)
- Request Preview:** Preview Request button
- Response Status:** Status: 200 OK, Time: 526ms, Size: 460 B
- Response Headers:** Body, Cookies, Headers (4), Test Results
- Response Content (Pretty View):**

```
1 {
2 "tasks": [
3 {
4 "description": "Needs to work on irregular verbs",
5 "done": false,
6 "id": 1,
7 "title": "Teach cat Spanish"
8 },
9 {
10 "description": "Remember what happened last time",
11 "done": false,
12 "id": 2,
13 "title": "Untangle Gordian knot"
14 }
15]
}
```
- Annotations:** A blue callout box labeled "JSON response" points to the "tasks" array in the response body.

# Contents

---

- ▶ REST Web Services

## JSON and Ajax

- ▶ Implementing a REST Service
- ▶ REST Clients
- ▶ Hands-On Exercise 9.1



# JSON—An Alternative to XML

- ▶ REST services commonly use JavaScript Object Notation (JSON) for data
- ▶ JSON is a simple, lightweight alternative to XML
  - Python objects can be marshalled to JSON instead of XML

```
{ "windSpeed" : 8.5,
 "isSpeedInKnots" : false,
 "direction" : "NW" }
```

JSON

```
<windConditions>
 <windSpeed>8.5</windSpeed>
 <isSpeedInKnots>false</isSpeedInKnots>
 <direction>NW</direction>
</windConditions>
```

XML

# Comparing JSON and XML

---

## ► Advantages of JSON

- More compact than XML
  - May reduce network transmission time
- Simple syntax makes it faster to process than XML
- Very easy for JavaScript clients to process

## ► Disadvantages of JSON

- JSON's simple data model doesn't support complex use cases
  - JSON data types: Strings, numbers, booleans, objects, and arrays
- XML data model is much richer
  - Supports user-defined data types, inheritance, attributes, id references
- JSON doesn't support namespaces
  - Makes it difficult to merge JSON documents from different projects
- No standard for JSON schemas

## ► Is JSON better than XML?

- No—JSON is more appropriate *for certain applications*

# JSON Format Overview

## ► Objects

- Objects in JSON consist of *members* in curly brackets ({} )

## ► Object members

- Each members consist of a *name* and a *value* separated by a colon (:)
- Members are separated by commas

```
{ "windSpeed" : 8.5,
 "isSpeedInKnots" : false,
 "direction" : "NW" }
```

No datatype info

## ► Member names are strings

- Always enclosed in double quotes

## ► Member values

- Strings enclosed in double quotes
- Numbers (no quotes)
- true, false, null (no quotes)

# Converting Python Data to JSON

► **Standard json module converts Python objects to and from JSON**

- `json.dumps()` converts built-in Python objects to JSON
  - `dumps()` input can be numbers, strings, lists, tuples, dictionaries
  - Returns a string
- `json.loads()` converts JSON string to Python dictionary
  - Returns a dict

```
>>> import json
>>> amount = {'value': 235.0, 'currency': 'EUR'}
>>> json.dumps(amount)
'{"currency": "EUR", "value": 235.0}'
>>> task = '{"title": "Run Pylint", "done": false}'
>>> json.loads(task)
{'done': False, 'title': 'Run Pylint'}
>>>
```

# Converting User-Defined Classes to JSON

- **json.dumps() needs help to convert user-defined classes to JSON**
  - Pass reference to conversion function as default argument
  - Conversion function should return a built-in Python type

amount\_to\_json.py

```
class Amount:
 def __init__(self, value, currency):
 self.value = value
 self.currency = currency

 def get_attr_values(obj):
 return obj.__dict__

amt_obj = Amount(65.4, 'SEK')
amt_json = json.dumps(amt_obj, default=get_attr_values)
```

Type of \_\_dict\_\_ is a  
built-in dictionary

- Generated JSON

```
{ "value" : 65.4, "currency" : "SEK" }
```

# JSON Arrays

---

- ▶ **Arrays**
  - JSON arrays contain comma-separated items in [...]
- ▶ **Python lists are converted to JSON arrays**

```
>>> ticker_symbols = ['LTRE', 'GOOGL', 'TSLA']
>>> json.dumps(ticker_symbols)
'["LTRE", "GOOGL", "TSLA"]'
```

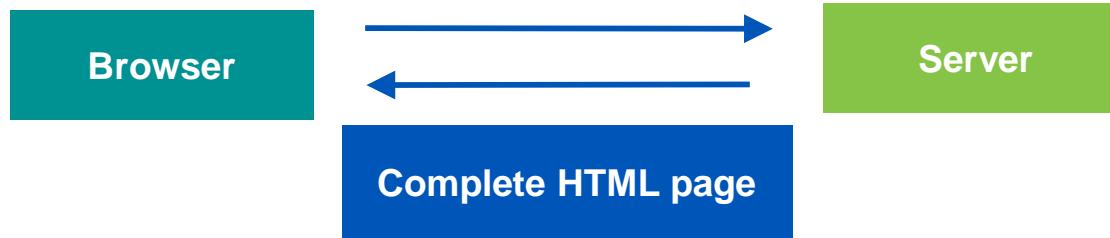
- ▶ **Arrays can contain objects**

```
>>> amounts = [Amount(235.0, 'EUR'), Amount(65.4, 'SEK')]
>>> json.dumps(amounts, default=get_attr_values)
'[{"value": 235.0, "currency": "EUR"},\n {"value": 65.4, "currency": "SEK"}]'
```

- Generated JSON

# Supporting Ajax Clients With REST Services

- ▶ **REST is often used with Asynchronous JavaScript and XML (Ajax)**
  - A technique for creating rich, interactive web applications
- ▶ **In traditional web applications, nothing happens until you click Submit**
  - All data that you enter into the form is sent to the server in one request
  - Server responds to request with a complete new HTML page
  - To change even a small section of page, browser must render entire page



HTML = hypertext markup language

# Ajax Compared to Traditional Web Requests

- ▶ In Ajax, requests are **asynchronous**
  - JavaScript in browser sends request without waiting for Submit button
  - REST service sends response with small bit of XML or JSON data
    - JavaScript in browser parses response
  - JavaScript quickly updates small sections of page with new data



- ▶ Page updates much faster than with a traditional HTML request
  - Page updates may include animation
  - Page reacts to mouse motion
- ▶ Most sophisticated Web sites use Ajax
  - Amazon, Google Maps, Netflix, Facebook, etc.

# Contents

---

- ▶ REST Web Services
- ▶ JSON and Ajax

## Implementing a REST Service

- ▶ REST Clients
- ▶ Hands-On Exercise 9.1



# Flask Microframework

---

- ▶ Python does not have built-in support for developing REST services
- ▶ Flask: lightweight Python web application framework
  - <https://palletsprojects.com/p/flask/>
- ▶ Easy to implement REST services with Flask
  - Add Flask decorators to map requests to functions
  - Add templates to URLs to pull data from request
  - Use Flask's facilities for converting to and from JSON
- ▶ Other Python web application frameworks support REST services
  - Pyramid: <https://trypyramid.com/>
  - Django: <https://www.djangoproject.com/>



Flask logo: © 2010 by Armin Ronacher. Some rights reserved. Use of logo does not indicate endorsement of Learning Tree International by Flask.

# Flask-Based REST Service

- ▶ Example adds Flask features to Python functions
  - @app.route decorator: maps HTTP request to a function
    - Flask checks incoming request's URL and HTTP method
    - If both match @app.route() arguments, Flask calls function
  - jsonify() function serializes Python dictionary to JSON
- ▶ Sample GET request:

```
GET http://localhost:5000/todo/api/v1.0/tasks
```

todo-api/rest\_server\_json.py

```
from flask import (Flask, jsonify, abort, request,
 make_response, url_for, Response)

BASE_URI = '/todo/api/v1.0/tasks'

@app.route(BASE_URI, methods=['GET'])
def get_tasks():
 tasks = rest_server_dao.get_all_tasks()
 return jsonify({'tasks': tasks})
```

Map URL and HTTP method to function

Serialize dictionary to JSON

# Getting Data From Request URL

- Route URL can include templates to capture portions of request URL
  - Syntax: <*data-type*: *param-name*>
  - Matching portion of URL is passed to function as *param-name*
- Flask's abort() function sets HTTP error response status
- Sample GET request with data in URL:

```
GET http://localhost:5000/todo/api/v1.0/tasks/51
```

```
@app.route(BASE_URI + '/<int:task_id>',
 methods=['GET'])
def get_task(task_id):
 task = rest_server_dao.get_task(task_id)
 if task is None:
 abort(404)
 return jsonify({'task': task})
```

Capture last URL component as task\_id

# Responding to POST Requests

- ▶ POST request body contains JSON data
  - Flask converts JSON in request to Python dictionary
  - Converted dictionary is available in function as `request.json`
- ▶ Sample POST request:

```
POST http://localhost:5000/todo/api/v1.0/tasks
{"title": "Eat", "description": "Lunch", "done": false}
```

```
@app.route(BASE_URI, methods=['POST'])
def create_task():
 if not request.json or \
 'title' not in request.json:
 abort(400)
 title = request.json['title']
 desc = request.json.get('description', '')
 done = request.json.get('done', False)
 task = rest_server_dao.create_task(title, desc, done)
 return jsonify({'task': task}), 201
```

request.json has data from request body

Returns HTTP status 201 Created

# Responding to PUT Requests

- ▶ PUT handling is similar to POST
- ▶ Sample PUT request:

```
PUT http://localhost:5000/todo/api/v1.0/tasks/51
{"done": true}
```

```
@app.route(BASE_URI + '/<int:task_id>', methods=['PUT'])
def update_task(task_id):
 if not request.json:
 abort(400)
 ...
 task = rest_server_dao.update_task(...)
 return jsonify({'task': task}), 202
```

Returns HTTP status  
202 Accepted

# Responding to DELETE Requests

- ▶ **DELETE request and response do not have bodies**
  - Function returns Flask Response object with HTTP status
  - DELETE usually returns HTTP status 204 (No content)
- ▶ **Sample DELETE request:**

```
DELETE http://localhost:5000/todo/api/v1.0/tasks/51
```

```
@app.route(BASE_URI + '/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
 if not rest_server_dao.delete_task(task_id):
 abort(404)
 return Response(status=204) ←
```

Returns HTTP status  
204 No Content

# Adding Authentication

## ► Flask includes features for HTTP Basic Authentication

- `@auth.get_password` decorates authentication callback function
  - Callback function returns password for username parameter
- Request to functions with `@auth.login_required` require credentials

```
@auth.get_password
def get_password(username):
```

Authentication callback function

```
 return rest_server_dao.get_password(username)
```

```
@auth.error_handler
```

```
def unauthenticated():
 return make_response(
 jsonify({'error': 'Unauthenticated'}), 401)
```

Handler for unauthenticated access attempts

```
@app.route(BASE_URI, methods=['GET'])
```

```
@auth.login_required
```

```
def get_tasks():
 ...
```

Request must include valid credentials

# Adding Data to a Request Body

Do Now

## 1. Use Postman to test a POST request to a REST service

- Change HTTP method to POST
- Click **Body** tab, then select **raw** and enter the following text:

```
{"title": "Run Pylint", "description": "Find bugs", "done": false}
```

The screenshot shows the Postman application interface. At the top, there are tabs for 'Builder' (selected), 'Runner', and various icons for import, sync, sign in, and support. Below the header, the URL is set to `http://localhost:5000...` and the method is set to `POST`. The 'Body' tab is highlighted with a red circle, and the 'raw' option is selected under the body type dropdown. The JSON payload is entered into the body editor:

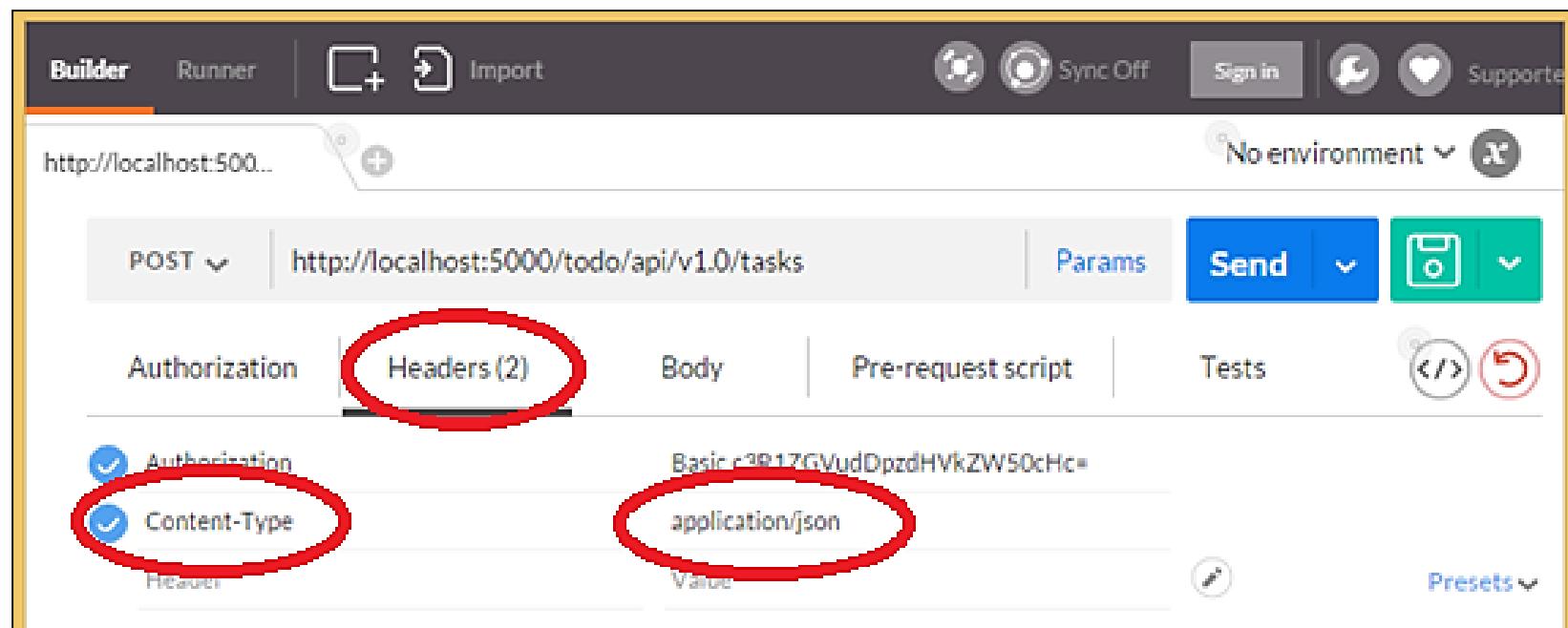
```
1 {"title": "Run Pylint", "description": "Find bugs", "done": false}
```

# Adding Data to a Request Body

## 2. Click Headers

- Enter **Content-Type** with value `application/json`
- Enter **Accept** with value `application/json`

## 3. Click Send



# Adding Data to a Request Body

Do Now

## 4. Response from POST request

The screenshot shows the Postman application interface. At the top, there are tabs for 'Builder' and 'Runner', along with various icons for 'Import', 'Sync Off', 'Sign in', and 'Supporters'. Below the header, the URL 'http://localhost:500...' is displayed, followed by a dropdown for 'No environment' and a delete button. The main action bar includes 'POST', the URL 'http://localhost:5000/todo/api/v1.0/tasks', 'Params', a 'Send' button, and a download icon.

Below the action bar, the status bar shows 'Body' (selected), 'Cookies', 'Headers(4)', 'Tests', 'Status 201 CREATED', and 'Time 18 ms'. Under the 'Body' tab, there are three options: 'Pretty' (selected), 'Raw', and 'Preview'. The 'Raw' view displays the JSON response:

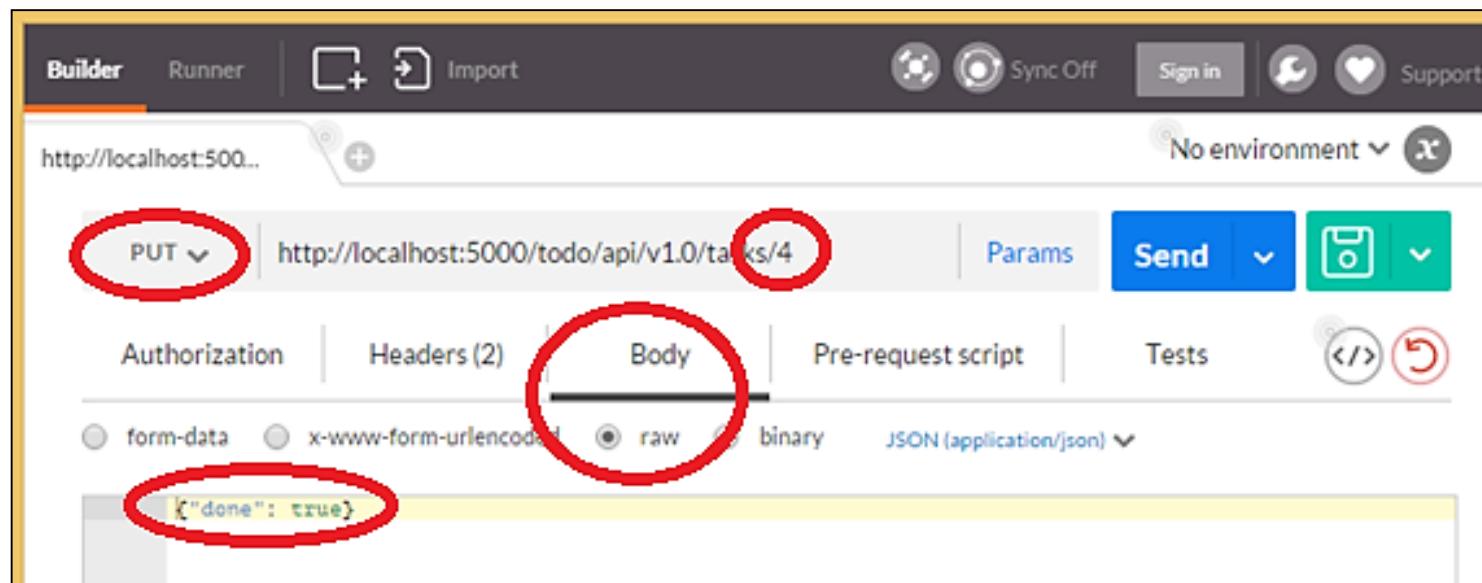
```
{
 "task": {
 "description": "Find bugs",
 "done": false,
 "id": 4,
 "label": "Run Pylint"
 }
}
```

A red oval highlights the 'id' field value '4' in the JSON response.

# Adding Data to a Request Body

## 5. Use Postman to test a PUT request to a REST service

- Change HTTP method to PUT
- Copy id value from POST response and add it to the end of the URL field
- Click **Headers** and verify Content-Type is application/json
- Click **Body** tab, then select **raw** and enter the following text:  
  { "done": true}
- Click **Send**



# Adding Data to a Request Body

Do Now

## 6. Response from PUT request

The screenshot shows the Postman application interface. At the top, there are tabs for 'Builder' (selected), 'Runner', and 'Import'. On the right side of the header are icons for 'Sync Off', 'Sign in', and 'Support'. Below the header, the URL 'http://localhost:500...' is entered, and the status is 'No environment'. The method is set to 'PUT' and the endpoint is 'http://localhost:5000/todo/api/v1.0/tasks/4'. Below the URL, there are buttons for 'Params', 'Send', and a green 'Collection' icon. The main content area shows the response details. Under 'Body', the tab is selected, showing a JSON response. The response body is:

```
{
 "task": {
 "done": true,
 "id": 4,
 "title": "Run Pylint"
 }
}
```

The 'done' key in the JSON object is circled in red.

# REST Security Considerations

- ▶ **Be cautious when using HTTP GET for REST requests**
  - Web servers often log complete URL of all GET requests including parameters and their values
    - Even when using HTTPS
    - Log files may be readable by ordinary users
  - Browsers may store GET parameter values in history
  - URL may be passed in HTTP Referrer header to third-party sites
- ▶ **For security, always use POST instead of GET for sensitive data**



# Contents

---

- ▶ REST Web Services
- ▶ JSON and Ajax
- ▶ Implementing a REST Service

## REST Clients

- ▶ Hands-On Exercise 9.1



# The Requests Package

- ▶ Python REST clients need a framework to send HTTP requests
- ▶ Standard Python HTTP modules provide low-level interfaces
  - `http.client`, `urllib` (Python 3),  
`urllib2` (Python 2)
- ▶ The Requests package is recommended for higher-level client interface
  - Available on PyPI:  
`pip install requests`



# Sending REST Requests

## ► Send GET request with `requests.get()`

- Set HTTP headers as dictionary in header parameter
- Pass credentials as tuple in auth parameter
- Response object's `json()` method converts JSON data to Python dictionary

todo-api/test\_rest\_server.py

```
import json, requests
base_url = 'http://localhost:5000/todo/api/v1.0/tasks'
def get_task(task_id):
 http_headers = {'Accept': 'application/json'}
 creds = ('student', 'studentpw')
 url = base_url + '/' + task_id

 r = requests.get(url, auth=creds, headers=http_headers)

 if r.status_code != 200:
 raise RuntimeError('Oops')

 json_result = r.json()

 task = json_result['task']
 return task['title'], task['description'], task['done']
```

The diagram illustrates the execution flow of the `get_task` function. It starts with the import statement and the definition of `base_url`. A callout box points to `base_url` with the note: "Production version must use https, not http!". The `get_task` function body follows, showing the creation of `http_headers`, `creds`, and `url`. An annotation "Send GET request" points to the `requests.get` call. The response `r` is then tested for a successful status code (200) with the annotation "Test HTTP status". If the test fails, a `RuntimError` is raised. Finally, the response body is converted to JSON with `r.json()` and assigned to `json_result`. The last step, extracting the task details from `json_result`, is annotated as "Get response body".

# Sending POST Requests

## ► Send POST request with `requests.post()`

- Pass JSON data in request body with `json` parameter

```
def create_task(title, description, done):
 http_headers = {'Content-Type': 'application/json', ←
 'Accept': 'application/json'}
 creds = {'student', 'studentpw'}
 task = {
 'title': title,
 'description': description,
 'done': done,
 }
 r = requests.post(base_url, auth=creds, headers=http_headers,
 json=task) ←
 if r.status_code != 201:
 raise RuntimeError('Problem creating task')
 return r.json()
```

**Send POST request**

**Content-Type: data type the client is sending**

**Dictionary will be converted to JSON in request body**

# Sending PUT and DELETE Requests

- `request.put()` and `request.delete()` have similar usage

```
def update_task(id, task):
 url = base_url + "/" + id
 http_headers = {'Content-Type': 'application/json'}
 creds = ('student', 'studentpw')
 r = requests.put(url, auth=creds, ← Send PUT request
 headers=http_headers, json=task)

 if r.status_code != 202:
 raise RuntimeError('Problem updating task')

def delete_task(id):
 url = base_url + "/" + id
 r = requests.delete(url, auth=('student', 'studentpw')) ← Send DELETE request

 if r.status_code != 204:
 raise RuntimeError('Problem deleting task')
```

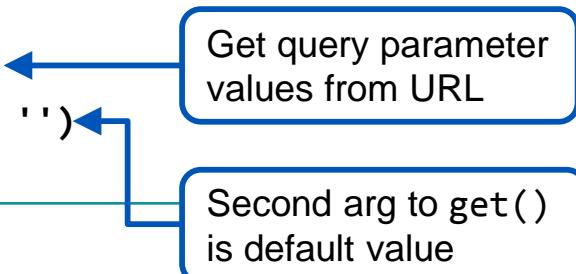
# Adding Parameters to Request URL

- REST service may accept query parameters using standard URL encoding
  - URL syntax: *url?param1=value1&param2=value2&...*
  - Service gets parameter values using `request.args.get('arg-name')`
  - Client sets values by passing a dictionary as `params` argument
- Request

```
GET http://glamorshack.com/rest/123?timezone=EST&debug=True
```

- Service method

```
@app.route(BASE_URI + '/<int:task_id>', methods=['GET'])
def get_task(task_id):
 tz = request.args.get('timezone', '')
 debug_mode = request.args.get('debug', '')
```



- Client code

```
...
query_params = {'timezone': 'EST', 'debug': True}
r = requests.get(url, params=query_params)
```



# Contents

---

- ▶ REST Web Services
- ▶ JSON and Ajax
- ▶ Implementing a REST Service
- ▶ REST Clients

## Hands-On Exercise 9.1



## Hands-On Exercise 9.1

In your Exercise Manual, please refer to  
**Hands-On Exercise 9.1: Interfacing With  
REST Web Services and Clients**



# Best Practices: REST Web Services

---

- ▶ **Don't include sensitive data in REST URLs, even if using TLS/SSL**
  - HTTP server logs complete URLs after decryption
- ▶ **Follow HTTP standard's definition of HTTP methods**
  - GET fetches resource, PUT updates resource, etc.
- ▶ **Follow HTTP standard's definitions of status codes**
  - 200 OK, 201 Created, 202 Accepted, etc.
- ▶ **Use JSON format for simple requests and Ajax processing**
- ▶ **Use XML format for complex requests**
- ▶ **Document REST services thoroughly**
  - Client developers need details of URL formats and message body contents

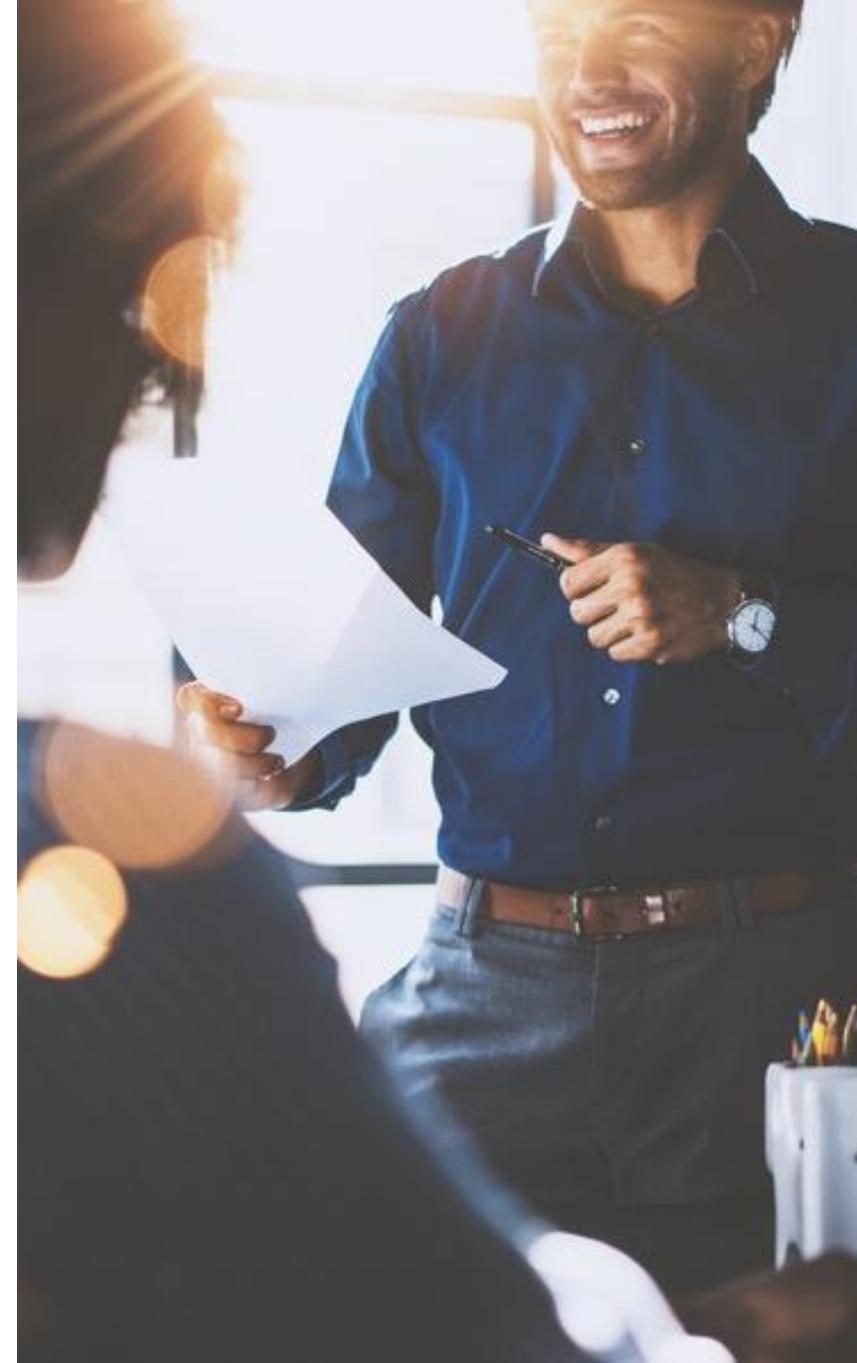
SSL = Secure Sockets Layer

TLS = Transport Layer Security

# Objectives

---

- ▶ Build a Python implementation of a REST web service
- ▶ Develop a REST web service that supports Ajax clients with JSON data
- ▶ Write Python clients that send requests to a REST web service
- ▶ Read and write JSON data in REST clients



# Chapter 10

# Course Summary



LEARNING TREE™  
INTERNATIONAL

# Course Objectives

- ▶ Employ best practices and design patterns to build Python applications that are reliable, efficient, and testable
- ▶ Exploit Python's OOP features for stable, maintainable code
- ▶ Automate unit testing with the Unittest, Pytest, and Mock modules
- ▶ Measure and improve performance of Python applications using profiling tools, code optimization techniques, and PyPy
- ▶ Install and distribute Python programs and modules using standard Python tools, including Pip and Python Packaging Index (PyPI)



OOP = object-oriented programming

# Course Objectives

---

- ▶ Create and manage concurrent threads of control
- ▶ Generate and consume REST web service requests and responses



REST = representational state transfer

# Appendix A

# Extending Python

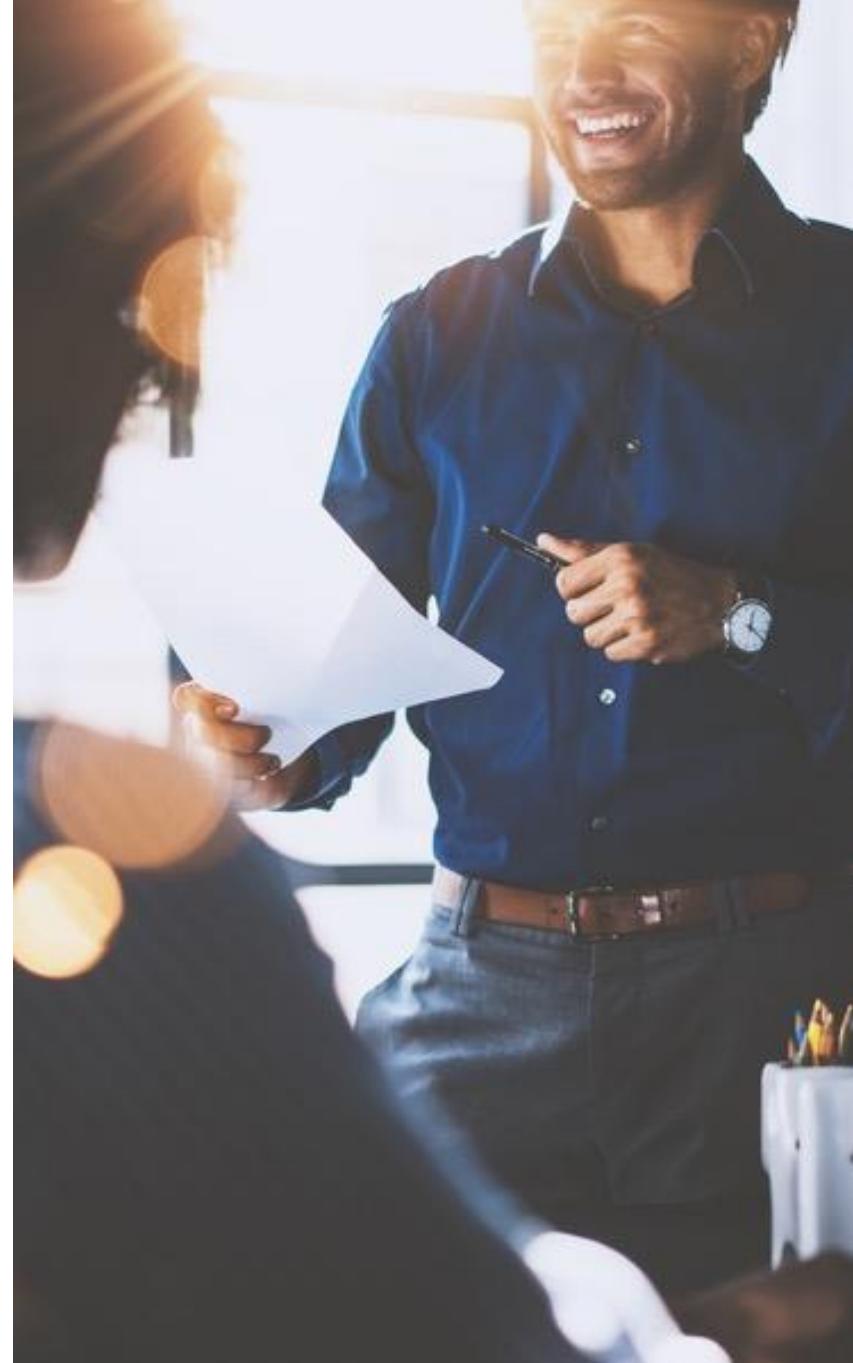


LEARNING TREE  
INTERNATIONAL

# Objectives

---

- ▶ Call C functions from Python applications using the `ctypes` module
- ▶ Extend Python's functionality with C extension modules
- ▶ Rewrite Python modules in the Cython language to improve application performance



# Contents

---

## Calling C Functions From Python

- ▶ Writing C Extension Modules
- ▶ The Cython Language
- ▶ Hands-On Exercise A.1



# Integrating C and C++ With Python

---

- ▶ **Code written in C/C++ is very close to “bare metal”**
  - C/C++ runtime libraries add very little overhead to compiled programs
- ▶ **Rewriting portions of a Python script in C/C++ can give big speed gains**
  - In the exercise, you’ll profile Python scripts before and after integrating C code
- ▶ **In this appendix, we’ll discuss several ways of integrating Python and C/C++**
  - Calling C/C++ functions using the standard `ctypes` module
  - Writing extension modules in C
  - Using the Cython framework
- ▶ **Note: There are no exam questions about C or C++ language details**

# ctypes Module

---

- ▶ **Standard ctypes module supports integrating C and C++ with Python**
  - Python modules can call C or C++ functions
  - C and C++ functions can call Python functions
- ▶ **To call functions, we need to pass data between languages**
  - Both languages must understand data used as arguments and return values
- ▶ **ctypes defines datatypes available to both C/C++ code and Python**
  - See next slide
- ▶ **ctypes requires a C or C++ compiler to be installed**
  - \*nix: cc (classic C compiler), gcc, g++ (open source)
  - Mac OS X: XCode
  - Windows: Microsoft Visual C++, gcc, g++
- ▶ **To be callable from Python, C functions must be compiled into a library**
  - On \*nix: .so file (shared object); on Windows: .dll file (DLL)

DLL = dynamic link library

# Compatible Fundamental Data Types

- **ctypes defines a number of primitive data types compatible with C**

ctypes Type	C Type	Python Type
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character Unicode string
c_int	int	int
c_long	long	int
c_float	float	float
c_double	double	float
c_char_p	char *	bytes object or None
c_wchar_p	wchar_t *	string or None

```
import ctypes
ans = c_int(24) # create a c_int instance
msg = c_wchar_p('answer is') # create a Unicode string
print('{} {}'.format(msg.value, ans.value)) # get values
ans.value = 42 # objects are mutable
```

# Calling C Functions From Python

- ▶ **Step 1: Define C header file and C source file**
  - This example builds a Windows library; steps for \*nix are similar
- ▶ **Sample C header file**

```
#ifndef BUILDING_PRIMES_DLL
#define PRIMES_DLL __declspec(dllexport)
#else
#define PRIMES_DLL __declspec(dllimport)
#endif

int PRIMES_DLL is_prime(int n);
```

The diagram illustrates the structure of a C header file named `primes.h`. It is annotated with several callouts:

- A blue box labeled "C compiler directives" points to the preprocessor directives (`#ifdef`, `#define`, `#else`, `#endif`) at the top of the code.
- A blue box labeled "Declarations of C function" points to the function declaration `int PRIMES_DLL is_prime(int n);`.
- A blue box labeled "primes.h" is positioned in the top right corner of the code area.
- An annotation box with a blue border and arrow points to the `__declspec(dllexport)` macro definition. It contains the text: "If we're building DLL, export function declarations".
- An annotation box with a blue border and arrow points to the `__declspec(dllimport)` macro definition. It contains the text: "If we're building a client, import function declarations".

# Calling C Functions From Python

## ► Sample C source file

primes.c

```
#include "primes.h"
PRIMES_DLL int is_prime(int n) {
 int i;
 int has_factor = 0;
 ...
 return !has_factor;
}
```

C source file includes header file

is\_prime function takes an int parameter and returns an int

## ► Step 2: Compile C code and create library

- This example creates a Windows DLL

Input is C source file

```
gcc -DBUILDING_PRIMES_DLL -o primes.o -c primes.c
gcc primes.o -shared -o primes.dll
```

Output is DLL

# Calling C Functions From Python

## ► Step 3: Call C function from Python module

primes.py

```
from ctypes import cdll, c_int

library = 'primes.dll'
library_mod = cdll.LoadLibrary(library) Load dynamic link library

is_prime = library_mod.is_prime Assign function reference to a variable
is_prime.argtypes = (c_int,) Define types of arguments and result
is_prime.restype = c_int

num = 29
result = is_prime(num) Call C function using function reference

print(f'{num} is {"not " if result == 0 else ""}prime')
```

Use result

# Passing Arrays to C Functions

## ► Python lists are complex objects that can grow or shrink

- C and C++ arrays are simple fixed-size blocks of contiguous memory
- Array parameters to C/C++ functions may be declared `int*` (pointer to `int`)

primes.h

```
int PRIMES_DLL primes_c(int how_many, int *p);
```

primes\_c() function populates array with prime numbers

Array of `int` is declared as “pointer to `int`”

## ► Passing arrays between Python and C requires special handling

- We need to create a new class for each different array length and item type
- We then call the new class's constructor to create the array

# Passing Arrays to C Functions

## ► Example of creating an array in Python

- `ctypes.c_int` class overloads the `*` operator
- `c_int * 10000` creates a new class on-the-fly: Array of 10000 int

ctypes\_demo.py

```
from ctypes import c_int, POINTER
...
primes_c = library_mod.primes_c
primes_c.argtypes = (c_int, POINTER(c_int))
primes_c.restype = c_int

ctypes_array_class = c_int * 10000
ctypes_array = ctypes_array_class()
c_result = primes_c(10000, ctypes_array)
last_5 = [ctypes_array[i]
 for i in range(9995, 10000)]
```

Function argument declared as `int*`

Create new array class

Call new class constructor to create array

Pass empty array to C function

Access array elements using sequence notation

# Demo of Calling C Functions From Python

Do Now

## 1. In PyCharm, open project appA\_examples

- Note the source files in the ctypes\_demo folder
- Makefile contains configuration for make build tool

## 2. Open a command prompt and enter the following commands

```
> cd \crs1906\examples\appA_examples\ctypes_demo
> use_anaconda
> make
rm -f primes.o primes.dll core
gcc -DBUILDING_PRIMES_DLL -Wall -Og -g -o primes.o -c primes.c
gcc primes.o -shared -o primes.dll
> python ctypes_demo.py
Loading DLL from primes.dll
28 is not prime
29 is prime
...
...
```

# Contents

---

- ▶ Calling C Functions From Python

## Writing C Extension Modules

- ▶ The Cython Language
- ▶ Hands-On Exercise A.1



# Writing Python Extension Modules in C

- ▶ **Calling C functions directly requires a lot of setup in Python**
  - If project calls C functions frequently, results in lots of repetitive code
- ▶ **To eliminate setup code, write a Python extension module in C**
  - Other Python modules can call your extension module as usual
  - Caveat: This technique requires experience with C coding
- ▶ **Example: Extension module for primes functions from previous section**
  - Step 1: Remove `__declspec()` from C header file
  - Step 2: Write extension wrapper functions in C
  - Step 3: Create driver script for `setuptools` package
  - Step 4: Build extension module
  - Step 5: Import extension module from Python modules
- ▶ **Step 1: Remove `__declspec()` from function declarations in C header file**

primes.h

```
int is_prime(int n);
int primes_c(int how_many, int *p);
```

No `__declspec()`

# Extension Wrapper Functions

## ► Step 2: Write extension wrapper functions in C

primes\_ext\_mod.c

```
#include "Python.h"
#include "primes.h"

static PyObject *py_is_prime(PyObject *self, PyObject *args) {
 int x; /* variable for parameter */
 int result; /* variable for result */

 if (!PyArg_ParseTuple(args, "i", &x)) {
 return NULL;
 }
 result = is_prime(x); // Call our C function
 return Py_BuildValue("i", result); // Build a Python value
} // continued on next slide...
```

Define variables for parameters and result

Copy Python numeric parameter to C int variable

Call our C function

Build a Python value for the result

# Module Configuration Functions

primes\_ext\_mod.c

```
/* Module method table */
static PyMethodDef PrimesMethods[] = {
 {"is_prime", py_is_prime, METH_VARARGS, "Test"}, ← Maps Python names
 {"primes_c", py_primes_c, METH_VARARGS, "Generate"}, → to C function names
 {NULL, NULL, 0, NULL}
};

/* Module structure */
static struct PyModuleDef primes_ext_mod = {
 PyModuleDef_HEAD_INIT,
 "primes_ext_mod", /* name of module */
 "Primes extension module", /* Doc string (may be NULL) */
 -1, /* Size of per-interpreter state or -1 */
 PrimesMethods /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC PyInit_primes_ext_mod(void) {
 return PyModule_Create(&primes_ext_mod);
}
```

Describes overall structure of module

Initializes extension module

# Writing Python Extension Modules in C

## ► Step 3: Create driver script for setuptools package

- Call to `setuptools.setup()` includes `ext_modules` argument
- Extension class describes a single C/C++ extension module
  - Extension constructor accepts arguments for C/C++ compiler
- `setup()` will invoke C/C++ compiler to build extension module

setup.py

```
from setuptools import setup, Extension
setup(name="primes_ext_mod",
 ext_modules=[
 Extension("primes_ext_mod",
 ["primes.c", "primes_ext_mod.c"],
 include_dirs=['.'])
])
```

The code shows a `setup()` call with a `name` parameter set to `"primes_ext_mod"`. This is annotated with a callout box labeled "Extension module name". Below it, another callout box labeled "List of source files" points to the list of source files `["primes.c", "primes_ext_mod.c"]` passed to the `Extension` constructor.

# Writing Python Extension Modules in C

## ► Step 4: Use setuptools to build extension module

- `--inplace`: Put extension module in source directory, not library directory

```
> python setup.py build_ext --inplace
```

## ► Step 5: Import extension module from Python modules

- Python code calls extension module's functions directly
- Arguments and results are ordinary Python objects

`primes_ext_mod_demo.py`

```
from primes_ext_mod import is_prime
n = 1023
result = is_prime(n)
```

Import function from extension module

Call function from extension module

# Passing Python Lists and Arrays

- ▶ **Passing Python lists to extension function requires more configuration**
  - Code shown here works for Python array instances or NumPy arrays

primes\_ext\_mod.c

```
/* int primes_c(int, int *) */
static PyObject *py_primes_c(PyObject *self, PyObject *args) {
 int howmany;
 PyObject *bufobj;
 Py_buffer view;
 double result;

 if (!PyArg_ParseTuple(args, "iO", &howmany, &bufobj)) {
 return NULL;
 }

 if (PyObject_GetBuffer(bufobj, &view,
 PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
 return NULL;
 }

 /* continued on next slide... */
```

Get the passed int and Python object

Extract buffer information and store in view

# Passing Python Lists and Arrays

primes\_ext\_mod.c

```
if (view.ndim != 1) {
 PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
 PyBuffer_Release(&view);
 return NULL;
}

if (strcmp(view.format, "i") != 0) {
 PyErr_SetString(PyExc_TypeError, "Expected an array of int");
 PyBuffer_Release(&view);
 return NULL;
}

result = primes_c(howmany, view.buf);

PyBuffer_Release(&view);
return Py_BuildValue("i", result);
}
```

Check type of items in array

Pass raw buffer to C function

Indicate we're done working with buffer

# Python Code to Pass Array

- ▶ Python code passes an array.array instance to function
  - No special setup required

primes\_ext\_mod\_demo.py

```
from array import array
from primes_ext_mod import primes_c
primes_buffer = array('i', [0] * 10000)
result = primes_c(how_many, primes_buffer)
last_5 = [primes_buffer[i]
 for i in range(9995, 10000)]
print("tail of primes_array: {}".format(last_5))
```

The diagram illustrates the interaction between Python code and a C extension module. It shows the following steps:

- Import function from extension module:** The line `from primes_ext_mod import primes_c` is annotated with this callout.
- Create array:** The line `primes_buffer = array('i', [0] * 10000)` is annotated with this callout.
- Pass array to function from extension module:** The line `result = primes_c(how_many, primes_buffer)` is annotated with this callout.
- Retrieve values from array as usual:** The line `last_5 = [primes_buffer[i] for i in range(9995, 10000)]` is annotated with this callout.

# Demo of Python Extension Module

Do Now

1. In PyCharm, open project appA\_examples
  - Note the source files in the extension\_mod\_demo folder
2. Open a command prompt and enter the following commands

```
> cd \crs1906\examples\appA_examples\extension_mod_demo
> python setup.py build_ext --inplace
running build_ext
building 'primes_ext_mod' extension
...
Generating code
Finished generating code
...
> python primes_ext_mod_demo.py
28 is not prime
1023 is prime
tail of primes_array: [104707, 104711, 104717, 104723, 104729]
```

# Contents

---

- ▶ Calling C Functions From Python
- ▶ Writing C Extension Modules

## The Cython Language

- ▶ Hands-On Exercise A.1



# Cython C-Extensions for Python

---

- ▶ **Cython is an optimizing static compiler**
  - Compiles standard Python modules to native machine language
  - Compiled code may execute many times faster than Python byte code
  - Disadvantage: Compiled code is not platform-independent
    - Unlike Python code, it won't run on different OS or different hardware
- ▶ **Cython defines a language that is a superset of Python**
  - Adds language features for declaring types of variables and functions
  - Makes static optimizations more effective
  - You can get big performance gains without writing any C/C++ code
- ▶ **If you need to call C/C++, it's much easier to use Cython than ctypes**
  - Writing C extension modules is also easier in Cython
- ▶ **Cython integrates well with NumPy scientific programming library**
  - Adds advanced support for large, multidimensional arrays and matrices
- ▶ **Cython project: <http://cython.org/>**
  - Documentation: <http://docs.cython.org/>

# Installing Cython

---

- ▶ **Cython requires a C or C++ compiler**
- ▶ **Installing on UNIX or Linux**
  - Use the system's installed C or C++ compiler (for example, gcc)
  - `pip install cython`
- ▶ **Installing on Windows**
  - Cython integrates best with MS Visual C++ compiler
    - Works well with Community edition of MS Visual Studio
  - Cython also supports open-source MinGW GNU C/C++ compiler
    - See <https://github.com/cython/cython/wiki/InstallingOnWindows>

# Integrating Cython Into Projects

## ► Cython filenames have .pyx extension

- Python files with Cython language extensions
- Allows declaring variables with C data types
- May include all Python operators, built-in functions, and standard modules

integrate.pyx

```
def f(double x): ← Declare parameter as C double
 return x**2 - x

def integrate_f(double a, double b, int N):
 cdef int i
 cdef double s, dx } Declare local variables as C
 int and double

 s = 0
 dx = (b - a) / N
 for i in range(N): ← Python built-in function
 s += f(a + i * dx)
 return s * dx
```

# Integrating Cython Into Projects

- ▶ **Cython compiles .pyx file to .c file, and compiles .c file to linkable library**
  - On \*nix: .so file (shared object)
  - On Windows: .pyd file (DLL)
- ▶ **Build your application using built-in setuptools module**
  - Create setup.py to invoke Cython at build time

setup.py

```
from setuptools import setup
from Cython.Build import cythonize

setup(name='Cython Demo',
 ext_modules=cythonize('*/*.pyx',
 compiler_directives={'language_level': '3'}))
```

Cython will process .pyx files

- Run setup.py to build Cython extensions in current directory

```
> python setup.py build_ext --inplace
```

- ▶ Now you can import your Cython extension like a plain Python module

# Running Cython

Do Now

## 1. Run the following commands at a command prompt

```
> cd \crs1906\examples\appA_examples
> clean
> python setup.py build_ext --inplace
> python
 >>> import integrate
 >>> integrate.integrate_f(0, 100, 10**7)
328333.283833309
 >>> import timeit
 >>> timeit.timeit('integrate.integrate_f(0, 100, 10**7)',
 setup='import integrate', number=1)
0.865146184780468
 >>> import integrate_python
 >>> timeit.timeit('integrate_python.integrate_f(0, 100, 10**7)',
 setup='import integrate_python', number=1)
4.386683852277315
```

Build Cython extensions

Import module and call functions as usual

Run pure Python version for comparison

Wow! *Big* difference in performance!

# Calling C Functions From Python

- ▶ Cython supports calling C functions from Python
- ▶ C functions can take Python numbers or strings as arguments
  - Return values can be numbers or strings
- ▶ C functions can't accept or return Python lists directly
  - Python code can allocate memory buffer using standard C library functions
  - C function can populate memory buffer as an array
  - Python can copy buffer items to Python list
- ▶ Example: C function that calculates prime numbers

primes\_c.c

```
int primes_c(int how_many, int *p) {
 int n = 2, k = 0, i = 0;
 while (i < k && n % p[i] != 0) {
 ...
 p[k] = n;
 }
 return how_many;
}
```

Parameter is array of int

Array operations

# Calling C Functions From Python

► **C function requires a header file with function declaration**

- Header file is imported by calling module

primes\_c.h

```
int primes_c(int how_many, int *p);
```

► **Cython file has special comments that set build configuration options**

- Build process reads comments for names of C source files

primes\_c\_wrapper.pyx

```
distutils: language = c
distutils: sources = primes_c.c
from libc.stdlib cimport malloc, free
cdef extern from "primes_c.h":
 int primes_c(int how_many, int p[])
continued on next slide...
```

Build options identify C source files

Import standard C memory management functions

Import C header file and declare C function

# Calling C Functions From Python

- ▶ Cython function allocates memory for C function's array argument
  - After C function returns, Cython function frees memory

primes\_c\_wrapper.pyx

```
cdef int *buffer = <int *>calloc(how_many, sizeof(int))

Declarres a pointer to an int

if not buffer:
 raise MemoryError()
try:
 result = primes_c(how_many, buffer)

 return [buffer[i] for i in range(result)]

finally:
 free(buffer)
```

Allocate memory for C array

Pass allocated memory to C function

Create list comprehension from array items assigned by C function

Return allocated memory to system

# Calling C Functions From Python

Do Now

1. **appA\_examples\test\_primes.py defines three functions**
  - calculate\_python: Pure Python version
  - calculate\_cython: Cython version with C data type declarations
  - calculate\_c\_wrapper: Cython version that calls external C function
2. **Run the following command at a command prompt**

```
> pytest -s test_primes.py
...
test_primes.calculate_python : 2.1601 seconds
test_primes.calculate_cython : 0.0513 seconds
test_primes.calculate_c_wrapper : 0.0510 seconds
```

Cython version is as fast as the C version



# Contents

- ▶ Calling C Functions From Python
  - ▶ Writing C Extension Modules
  - ▶ The Cython Language

## Hands-On Exercise A.1



## Hands-On Exercise A.1

In your Exercise Manual, please refer to  
**Hands-On Exercise A.1: Extending Python**



# Discussion: Integrating C/C++ With Python

---

- ▶ **If performance is highest priority, consider rewriting critical sections in C**
  - Call C/C++ functions using ctypes module
  - Write C/C++ extension modules
    - Build modules using Cython or setuptools module and setup.py
- ▶ **But rewriting Python in C can be difficult and error prone**
  - C/C++ compilers miss many coding errors involving pointer manipulation
  - Developer is responsible for memory management
  - Often multiple Python components must be rewritten in C
    - Increases testing needs
    - Increases risk
- ▶ **Be sure to do a thorough risk analysis before committing to C/C++**

# Best Practices: Enhancing Performance

---

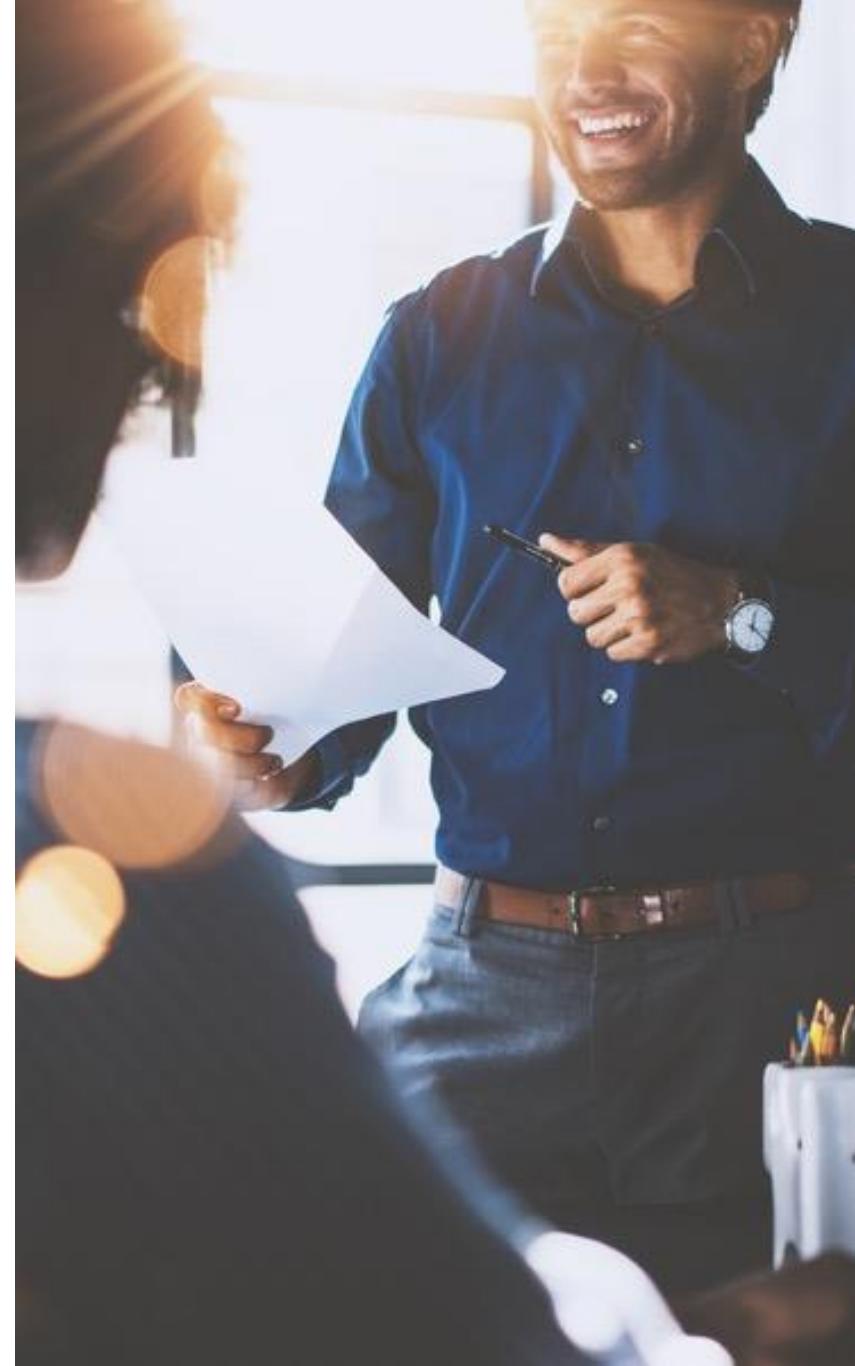
- ▶ For calling C functions or writing simple extensions, use ctypes module
- ▶ For performance gains with few code changes, use Cython .pyx files
  - But remember, this has an impact on portability of your application
- ▶ For integrating large C libraries, use Cython



# Objectives

---

- ▶ Call C functions from Python applications using the `ctypes` module
- ▶ Extend Python's functionality with C extension modules
- ▶ Rewrite Python modules in the Cython language to improve application performance



# Appendix B

# Advanced Python Features



LEARNING TREE  
INTERNATIONAL

# Objectives

---

- ▶ Introspect class attributes at runtime
- ▶ Use the State design pattern to build a class whose behavior changes based on its inputs
- ▶ Decouple object creation from object usage with the Factory Method design pattern
- ▶ Write test cases with advanced features of the Mock module
- ▶ Replace the standard CPython implementation with Jython
- ▶ Simplify concurrent applications with the AsyncIO library
- ▶ Synchronize threads using Queues
- ▶ Schedule recurring events using the sched module
- ▶ Interact with an XML-based REST service
- ▶ Port Python 2 code to Python 3
- ▶ Debug applications with Python's standard pdb module

# Contents

# Introspection and Metaclasses

- ▶ State Design Pattern
  - ▶ Factory Method Design Pattern
  - ▶ Mock Module Advanced Features
  - ▶ The Jython Interpreter
  - ▶ The Asyncio Module
  - ▶ Synchronizing Threads With Queues
  - ▶ The sched Module
  - ▶ Handling XML REST Responses
  - ▶ Porting Python 2 Code to Python 3
  - ▶ Debugging With pdb
  - ▶ Hands-On Exercise B.1



# Introspection

---

- ▶ **Python includes features for *introspection***
  - Introspection: Examining and modifying class and object attributes at runtime
- ▶ **Introspection methods**

<code>dir(object)</code>	Returns list of <i>object</i> 's attribute names
<code>vars(object)</code>	Returns <i>object</i> 's <code>__dict__</code> attribute
<code>callable(object)</code>	True if <i>object</i> is callable (function or class)
<code>getattr()</code> , <code>setattr()</code>	Get and set attributes of an object
<code>isinstance()</code> , <code>issubclass()</code>	Test types of objects and classes

- ▶ **Special data attributes**

<code>__dict__</code>	Dictionary of object's attributes and values
<code>__name__</code>	Name of object
<code>__class__</code>	Class of object
<code>__doc__</code>	The object's doc comment ( """...""" )

# Working With Introspection

Do Now

## 1. Enter the following statements into the Python console

```
>>> class X:
... """Simple class"""
... def f(self):
... """A method"""
... return 99
...
>>> obj = X()
>>> dir(obj) ← Get list of attribute names
[..., 'f']
>>> obj.msg = 'Hi'
>>> dir(obj)
[..., 'f', 'msg']
>>> obj.__dict__ ← Get all attribute names and values
{'msg': 'Hi'}
```

# Working With Introspection

## 2. Enter the following statements into the Python console

```
>>> obj.msg
'Hi'
>>> getattr(obj, 'msg') ← Get value of msg attribute
'Hi'
>>> callable(obj.msg) ← Is value of msg attribute
False
a function or class?
>>> callable(obj.f)
True
>>> callable(getattr(obj, 'f'))
True
>>> getattribute(obj, 'f')()
99
>>> X.__doc__ ← Get value of msg attribute
'Simple class'
>>> X.f.__doc__
'A method'
>>> getattr(X, 'f').__doc__
'A method'
```

# Introspection Code Example

- ▶ **Example: Print class's method names and docstrings**
  - Code is in appB\_examples/introspection\_demo.py

```
def print_docstrings(arg):
 """Prints method names and their docstrings"""
 method_names = [method for method in dir(arg)
 if callable(getattr(arg, method))]

 docstrings = ['{} {}'.format(name.ljust(20),
 '\n'.join(getattr(arg, name).__doc__.split()))
 for name in method_names]

 print('\n'.join(docstrings))

class MyClass:
 def do_work(self):
 """Do work for the class."""
 ...

print_docstrings(MyClass) # print the docstrings of the class's methods
```

Get list of methods

Get list of method doc strings with newlines replaced by spaces

# Metaclasses

---

- ▶ **Everything in Python is an object**
  - Every object is an instance of a class
  - `msg = 'Hello' # msg is an instance of str class`
  - `sc = SimpleCounter(0) # sc is an instance of SimpleCounter class`
- ▶ **Python classes are also objects**
  - `SimpleCounter` is an object
  - Question: What is the class of `SimpleCounter`?
  - Answer: `SimpleCounter` is instance of a built-in class named `type`
- ▶ **The `type` class is a *metaclass***
  - Metaclass: the class of a class
- ▶ **It's helpful to know the basics of metaclasses**
  - Primary benefit: It helps you understand Python documentation
  - But you'll rarely need to write metaclasses for your applications

# Viewing Class Attributes

1. In PyCharm: File | Open | C:\crs1906\examples\appB\_examples
2. Select Tools | Python Console

```
>>> from simple_counter import SimpleCounter
>>> msg = 'Hello'
>>> msg.__class__
<class 'str'>
```

Type of msg is str

```
>>> sc = SimpleCounter(0)
>>> sc.__class__
<class 'simple_counter.SimpleCounter'>
```

Type of sc is  
SimpleCounter

```
>>> SimpleCounter.__class__
<class 'type'>
```

Type of SimpleCounter is type

# Metaclasses

---

- ▶ As classes are factories for objects, metaclasses are factories for classes
- ▶ Class definitions create several items
  - Class name
  - Dictionary of class attributes (methods and class data attributes)
  - List of base classes
- ▶ Metaclass is responsible for taking those items and creating the class
  - Items are passed as arguments to metaclass `__new__()` and `__init__()`
- ▶ When a module is loaded, interpreter reads class definition
  - Interpreter calls metaclass `__new__()` and `__init__()`
  - `__new__()`: Class object is being created (memory is being allocated)
  - `__init__()`: Class object creation is complete
- ▶ Metaclass `__new__()` and `__init__()` are called only once
  - Metaclass can perform one-time initializations for a class

# Metaclass Example

```
class MyMeta(type): # base class of metaclass is type
 def __new__(meta_cls, cls_name, base_cls, cls_attrs):
 print('in MyMeta.__new__(): cls_name: ', cls_name)
 print('Metaclass: ', meta_cls)
 print('Base classes: ', base_cls)
 print('Class attributes: ', cls_attrs)
 return super().__new__(meta_cls, cls_name, base_cls,
cls_attrs)

 def __init__(cls, cls_name, base_cls, cls_attrs):
 print('in MyMeta.__init__(): cls_name: ', cls_name)
 print('Class: ', cls)
 print('Base classes: ', base_cls)
 print('Class attributes: ', cls_attrs)
 super().__init__(cls_name, base_cls, cls_attrs)

class MaxCounter(SimpleCounter, metaclass=MyMeta):
 max = 0 # class attribute, not instance attribute
```

# Metaclasses

## ► Output of previous example

```
---- in MyMeta.__new__(): cls_name: MaxCounter ----
Metaclass: <class '__main__.MyMeta'>
Base classes: (<class 'simple_counter.SimpleCounter'>,)
Class attributes: {'__module__': '__main__', ..., 'max': 0}

---- in MyMeta.__init__(): cls_name: MaxCounter ----
Class: <class '__main__.MaxCounter'>
Base classes: (<class 'simple_counter.SimpleCounter'>,)
Class attributes: {'__module__': '__main__', ..., 'max': 0}
```

# Metaclasses

## ► Example from standard `string.Template` module

- Metaclass `__init__()` performs one-time initializations for class

```
class TemplateMetaclass(type):
 default_pattern = '...' # complex regular expression (RE) pattern

 def __init__(cls, name, bases, cls_attrs):
 super().__init__(name, bases, cls_attrs)
 if 'pattern' in cls_attrs: # check if class has 'pattern' attr
 pattern = cls.pattern # if so, use the class's pattern
 else:
 pattern = TemplateMetaclass.default_pattern
 cls.pattern = re.compile(pattern) # RE is compiled only once

class Template(metaclass=TemplateMetaclass):
 ...

class MyTemplate(Template): # subclass can define a pattern
 pattern = 'This is my template pattern'
 ...
```

# Multiple Inheritance

- ▶ **Python classes can extend several classes**
  - Subclass inherits methods from all superclasses

```
class Subclass(Superclass1, Superclass2, Superclass3):
 ...
 obj = Subclass()
 obj.do_something()
```



Call methods as usual

- ▶ **If `do_something()` is not defined in subclass, Python searches superclasses**
  - Depth-first (Superclass1, then Superclass1's superclass, etc.)
  - Left-to-right (Superclass1, then Superclass2, then Superclass3)

# Multiple Inheritance

## ► Example: initialization with multiple inheritance

multiple\_inheritance.py

```
class FlightReservation:
 def __init__(self, airline):
 self.airline = airline

class HotelReservation:
 def __init__(self, room_type):
 self.room_type = room_type

class FlightAndHotelRes(FlightReservation, HotelReservation):
 def __init__(self, airline, room_type):
 FlightReservation.__init__(self, airline)
 HotelReservation.__init__(self, room_type)

res = FlightAndHotelRes('Delta', 'King')
print('Airline:', res.airline) # FlightReservation attribute
print('Room type:', res.room_type) # HotelReservation attribute
```



res

airline Delta  
room\_type King

Call superclass constructors

Access attributes as usual

# Avoiding Multiple Inheritance

- ▶ **Multiple inheritance can make code complex and difficult to maintain**
- ▶ **Multiple inheritance may be replaced by *composition***
  - Create instances of classes as data attributes
  - No inheritance required

```
class FlightAndHotelRes:
 def __init__(self, airline, room_type):
 self.flight_res = FlightReservation(airline)
 self.hotel_res = HotelReservation(room_type)

 @property
 def airline(self):
 return self.flight_res.airline

 @property
 def room_type(self):
 return self.hotel_res.room_type

res = FlightAndHotelRes('Delta', 'King')
print('Airline:', res.airline)
print('Room type:', res.room_type)
```

No superclasses

Create data attributes for flight and hotel info

Define properties to access data

Access flight and hotel info

# Mix-In Classes

## ► The most practical use case for multiple inheritance is *mix-in classes*

- Each mix-in class defines one well-defined bit of functionality
- Example: Override Boolean comparison operators
- Goal: Allow classes to override operators with as little new code as possible

`mixin_class.py`

```
class ComparableMixin:
 """

 This class has methods that override !=, <, >, and >= operators
 Subclasses must define overrides for <= and ==
 """

 def __ne__(self, other): ← Implements != operator
 return not (self == other)

 def __lt__(self, other): ← Implements < operator
 return self <= other and (self != other)

 def __gt__(self, other): ← Implements > operator
 return not self <= other

 def __ge__(self, other): ← Implements >= operator
 return self == other or self > other
```

# Mix-In Classes

- Other classes can add mix-in functionality as required

```
class SimpleCounter:
 def __init__(self, start=0):
 self._count = start

 ...

class UpAndDownCounter(SimpleCounter, ComparableMixin):
 ...
 def __le__(self, other): ← Implements <= operator
 return self._count <= other._count
 def __eq__(self, other): ← Implements == operator
 return self._count == other._count

counter0 = UpAndDownCounter(0)
counter1 = UpAndDownCounter(1)
assert counter0 <= counter1 ← Operators defined in UpAndDownCounter
assert counter0 == counter1 ← Operators defined in UpAndDownCounter
assert counter0 < counter1 ← Operators defined in ComparableMixin
assert counter0 != counter1 ← Operators defined in ComparableMixin
assert counter0 > counter1 ← Operators defined in ComparableMixin
assert counter0 >= counter1 ← Operators defined in ComparableMixin
```

# Contents

---

- ▶ Introspection and Metaclasses

## State Design Pattern

- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# Reacting to Events

---

- ▶ **A component's response to input may depend on previous inputs**
  - Example: Parser processes input token differently based on previous tokens
  - Example: Network connection reacts to connection requests differently when the connection is established, listening, or closed
  - Example: Hardware's response to signal depends on signals received earlier
- ▶ **In all of these examples, the component has different *states***
  - Component's response to an input event depends on its current state
- ▶ **We need a way to design components with state-dependent behavior**
- ▶ **Solution: The *State* pattern**

# State Design Pattern

- ▶ **State pattern allows an object to alter its behavior when its state changes**
  - Object appears to change its class
- ▶ **Also known as *State Machine* pattern**
- ▶ **Useful when object behavior changes based on previous inputs**
- ▶ **Applicable when objects have different modes of behavior**
  - Each mode is a state

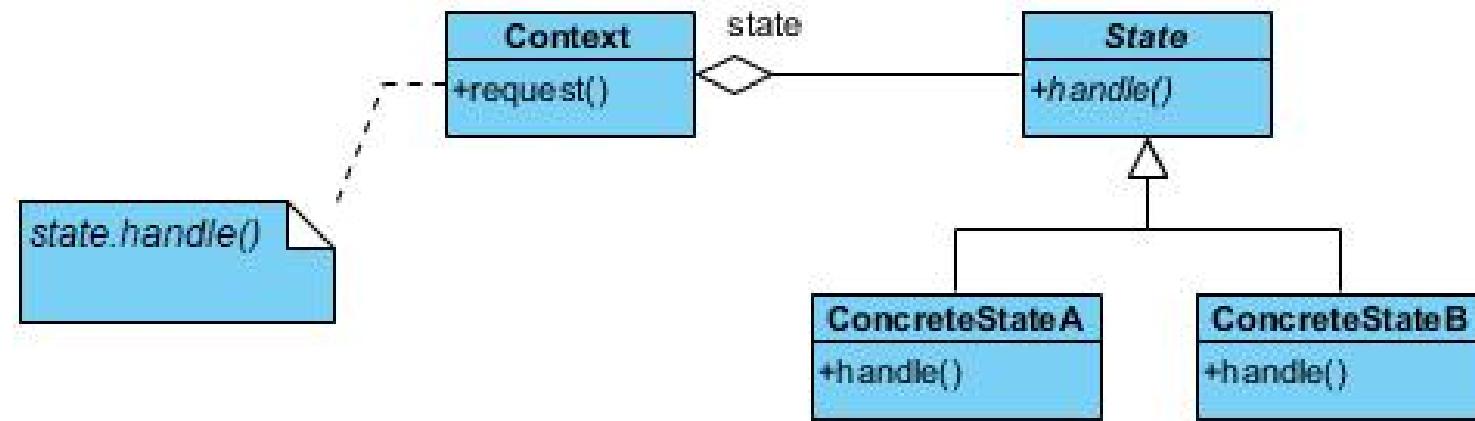
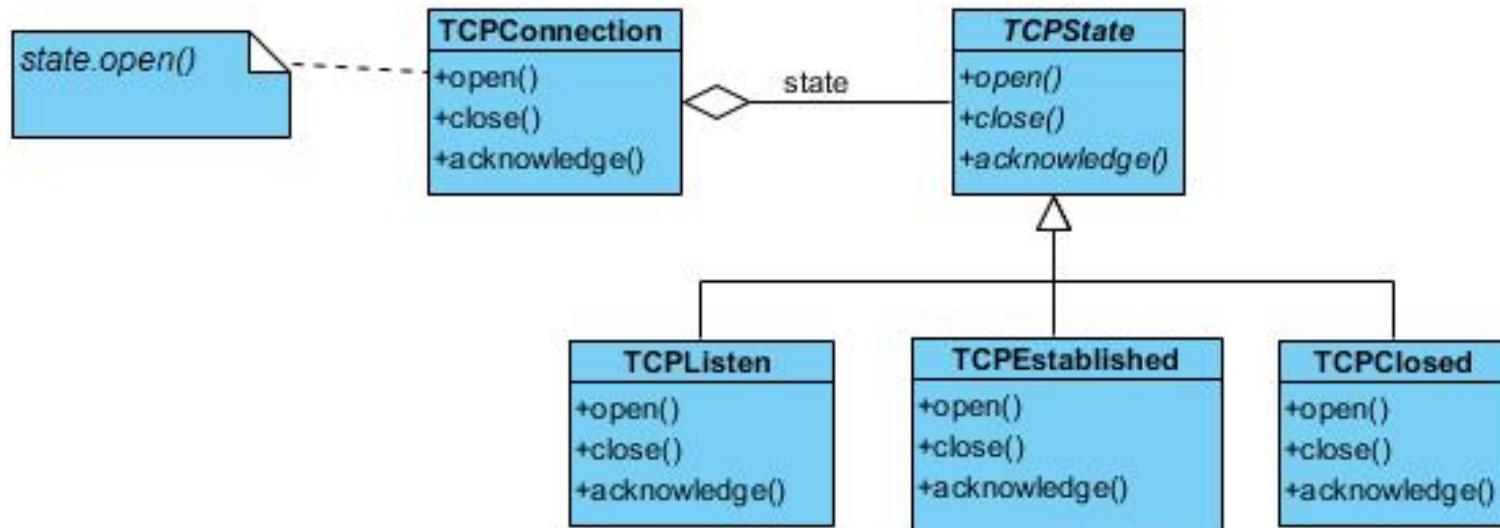


Image Source: Generated by Michael Woinoski.

# State Design Pattern

## ► Usages of State design pattern

- Programming language parsers
- Hardware device drivers
- TCP driver: Connection has states *established*, *listen*, and *closed*
  - Inputs (events) to connection are requests to *open*, *acknowledge*, and *close*
  - Connection responds to each event differently depending on its state

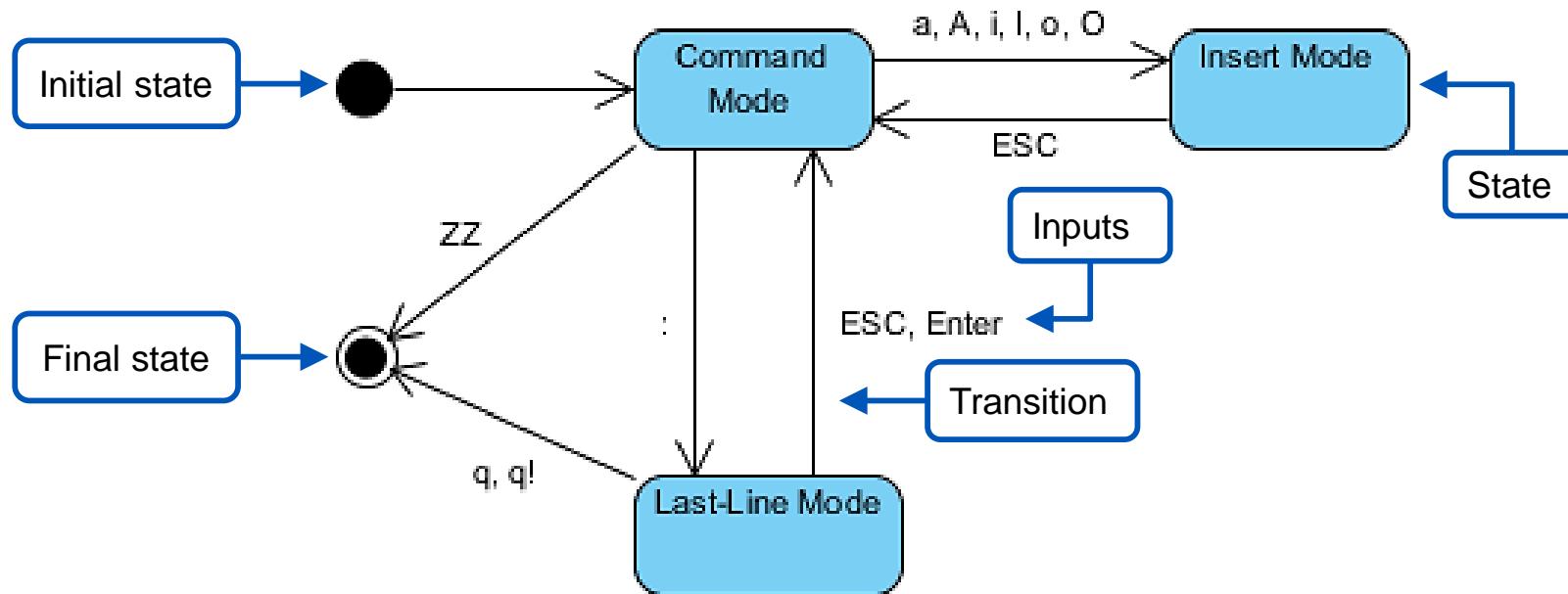


TCP = Transmission Control Protocol

Image Source: Generated by Michael Woynoski.

# Documenting State Machine Behavior

- ▶ Documented with UML State Machine diagram
  - Rounded rectangles represent states
  - Arrows between rectangles represent transitions between states
  - Labels on arrows represent inputs that cause the state transition to occur
- ▶ Example: \*nix vi editor has three modes: **command**, **insert**, and **last-line**
  - Depending on mode, keyboard input has different results



\*nix = UNIX-like system: UNIX, Linux, Mac OS X, etc.  
Image Source: Generated by Michael Woinoski.

# State Machine Implementation

---

- ▶ State machine implementation adds features to execute application logic
- ▶ State may have ***enter actions*** and ***exit actions***
  - Enter action: Function executed when transitioning to this state
  - Exit action: Function executed when transitioning from this state
- ▶ Transitions may have ***events, guards, and actions***
  - Event: Input that triggers this transition
  - Guard: Boolean condition that determines whether transition occurs
  - Action: Function executed when this transition occurs
- ▶ Traditional coding techniques usually involve nested if-else decisions
  - If there are states and events, code is very complex
- ▶ Python implementation is simpler
  - Actions and guards are function references or lambdas
  - State has dictionary that maps events to transitions
    - Events can be arbitrary objects

# Generic State Machine Implementation

```
class State:
 def __init__(self, name, *, transitions=None,
 enter_action=None, exit_action=None):
 self.name = name
 self.enter_action = enter_action
 self.exit_action = exit_action
 self.transitions = None
 if transitions:
 self.transitions = {t.event: t for t in transitions}

 def add_transition(self, transition):
 self.transitions[transition.event] = transition

class Transition:
 def __init__(self, *, event, target, guard=None, action=None):
 self.event = event
 self.target = target
 self.guard = guard
 self.action = action
```

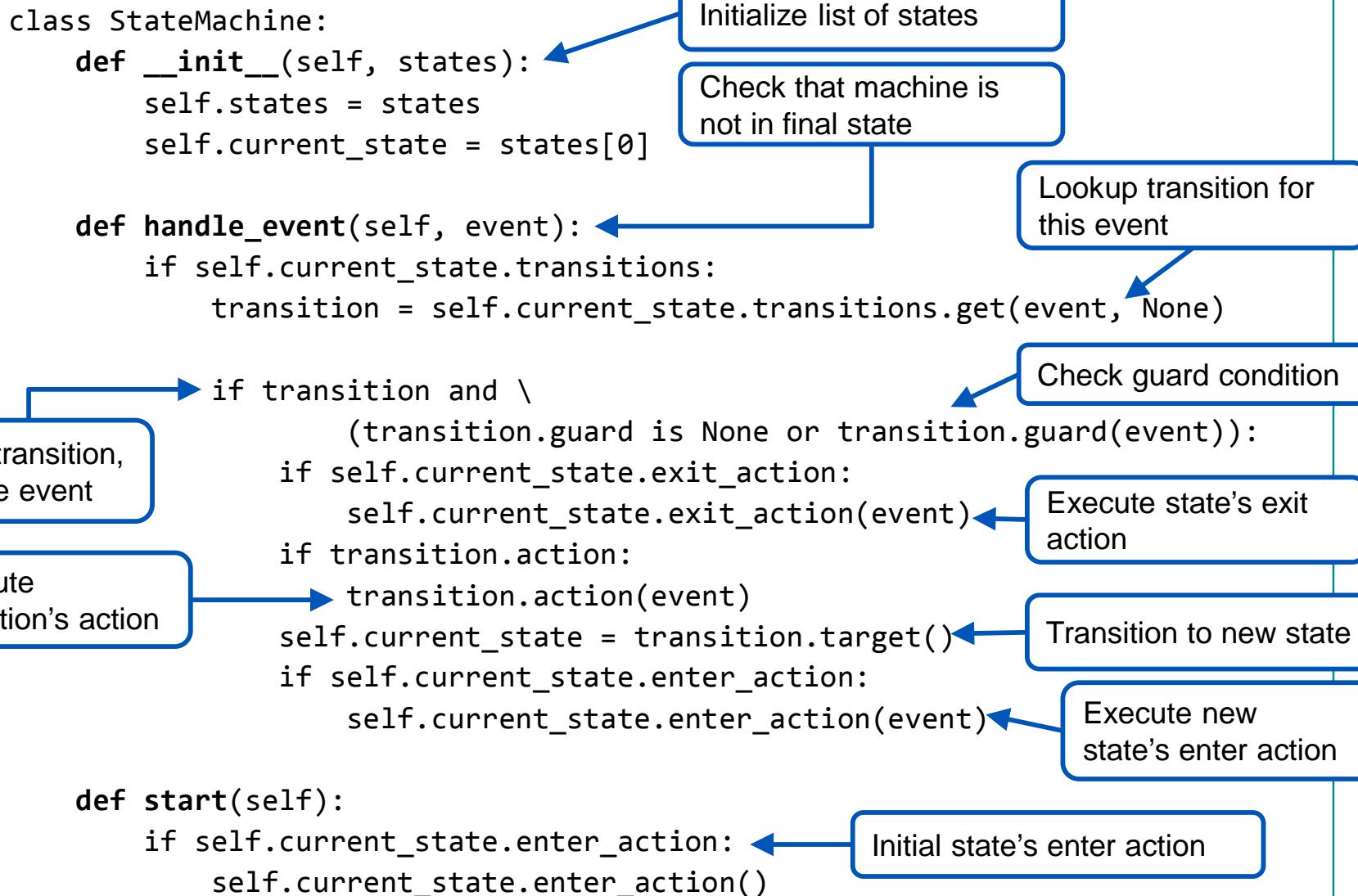
Actions are function references

Create mapping of events to transitions

Target state

guard and action are function references

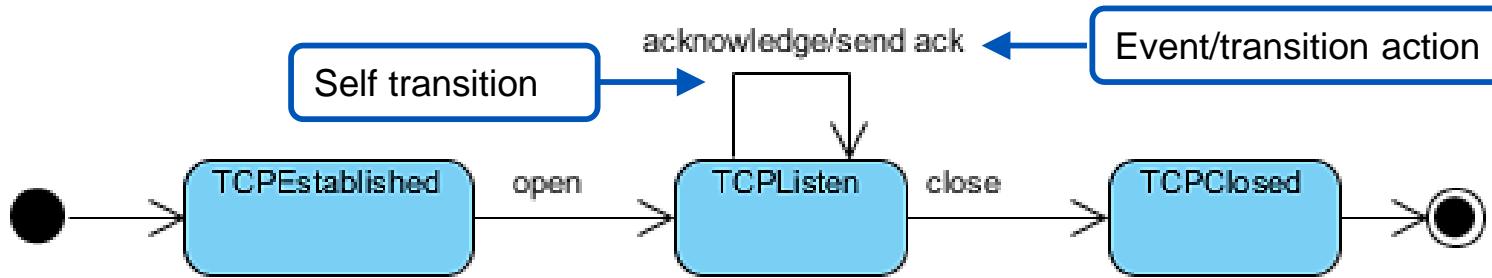
# Generic State Machine Implementation



# Usage of State Machine Implementation

## ► Example: Simulation of TCP connection

- Transition target defined as lambda so it can reference other states



```
class Event(Enum):
 open, acknowledge, close = range(3)

class TCPConnection(StateMachine):
 def __init__(self):
 self.tcp_established = State(
 name='TCPEstablished',
 enter_action=lambda: print('TCPEstablished state'),
 transitions=[Transition(event=Event.open,
 target=lambda: self.tcp_listen)])
```

# continued on next slide...

# Usage of State Machine Implementation

```
self.tcp_listen = State(← Define TCPListen state
 name='TCPListen',
 enter_action=lambda event: print('Entered TCPListen'),
 transitions=[
 Transition(event=Event.acknowledge, ← TCPListen has two
 action=self.send_ack,
 target=lambda: self.tcp_listen),
 Transition(event=Event.close, ←
 target=lambda: self.tcp_closed)])
self.tcp_closed = State(← Define TCPClosed state
 name='TCPClosed',
 enter_action=lambda event: print('Entered TCPClosed'))
states = [self.tcp_established, self.tcp_listen, self.tcp_closed]
super().__init__(states)

def send_ack(self, event):
 print('Executing TCPConnection.send_ack({})'.format(event))

continued on next slide...
```

# Usage of State Machine Implementation

## ► Driver code

```
events = [Event.close, Event.acknowledge, Event.open,
 Event.acknowledge, Event.open, Event.acknowledge,
 Event.close, Event.acknowledge, Event.open]

conn = TCPConnection()
conn.start()

for event in events:
 conn.handle_event(event)
```

Events are ignored if they occur  
when the current state has no  
transition for that event

## ► Output

```
Entered TCPEstablished state
Entered TCPListen state
Executing TCPConnection.send_ack(Event.acknowledge)
Entered TCPListen state
Executing TCPConnection.send_ack(Event.acknowledge)
Entered TCPListen state
Entered TCPClosed state
```

Ignored close and acknowledge  
events in TCPEstablished state

Ignored open event in  
TCPListen state

Ignored acknowledge and open  
events in TCPClosed state

# Run Sample State Machine Script

Do Now

## 1. Edit

C:\crs1906\examples\appB\_examples\test\_tcp\_connection.py

- If event occurs in state with no transition for that event, it is ignored

```
def test_tcp_connection(self):
 conn = TCPConnection()
 conn.start() # initial state is established
 self.assertEqual(conn.tcp_established, conn.current_state)

 event_next_state_list = [
 (Event.close, conn.tcp_established), # ignored
 (Event.acknowledge, conn.tcp_established), # ignored
 (Event.open, conn.tcp_listen), # open connection
 (Event.acknowledge, conn.tcp_listen), # send ack
 (Event.open, conn.tcp_listen), # ignored
 (Event.acknowledge, conn.tcp_listen), # send ack
 (Event.close, conn.tcp_closed), # close connection
 (Event.acknowledge, conn.tcp_closed), # ignored
 (Event.open, conn.tcp_closed), # ignored
]
 for event, result_state in event_next_state_list:
 conn.handle_event(event)
 self.assertEqual(result_state, conn.current_state)
```

# Run Sample State Machine Script

Do Now

2. Switch to a command prompt, execute the following command

```
pytest test_tcp_connection.py
```

3. Verify test case passes

```
===== test session starts =====
platform win32 -- Python 3.5.0 -- py-1.4.30 -- pytest-2.7.2
rootdir: C:\crs1906\examples\appB_examples, infile:
collected 1 items

test_tcp_connection.py .

===== 1 passed in 0.11 seconds =====
```

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern

## Factory Method Design Pattern

- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# Factory Method Design Pattern

---

- ▶ **Problem with our Proxy implementation: Driver calls proxy's constructor**
  - Constructor call can't be modified dynamically
- ▶ **Solution: Implement *Factory Method* design pattern**
  - Factory class defines method that creates concrete subclass of Image
- ▶ **Type of returned image can be changed by configuring factory class**
  - Factory configuration could be read from external configuration file

# Factory Method Implementation

- ▶ **Factory class maintains map of classes and their replacements**
  - `get_instance()` method returns replacement type if registered

```
class ObjectFactory:
```

```
 class_map = {}
```

Factory method

```
@classmethod
```

```
def register(cls, type, replacement_type):
```

```
 cls.class_map[type] = replacement_type
```

Register a type and its replacement

Type of desired object

```
@classmethod
```

```
def get_instance(cls, type, *args, **kwargs):
```

```
 returned_type = cls.class_map.get(type, type)
```

```
 return returned_type(*args, **kwargs)
```

Other args passed to class constructor

```
 ↑
```

Call constructor for returned type

If type is not in map, use type as default value

# continued on next slide...

# Factory Method Implementation

```
class ObjectFactory:
 ...

 def main(show_it):
 ObjectFactory.register(ConcreteImage, ImageProxy)

 image = ObjectFactory.get_instance(ConcreteImage, '...')
 client = ObjectFactory.get_instance(ImageClient, image)
 client.display_image()
```

Register proxy as replacement  
for ConcreteImage

Get objects from factory

# Proxy Script With Object Factory

Do Now

## 1. Edit

C:\crs1906\examples\appB\_examples\image\_proxy\_factory\_method.py

## 2. Open a command prompt and execute the following command

- Note message from ConcreteImage constructor: Image file was loaded

```
> python image_proxy_factory_method.py --display-image
...
ConcreteImage.__init__("LargeMagellanicCloud.jpg")
```

## 3. Execute the following command

- Note ConcreteImage constructor is not called: Image file was not loaded

```
> python image_proxy_factory_method.py
```

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern

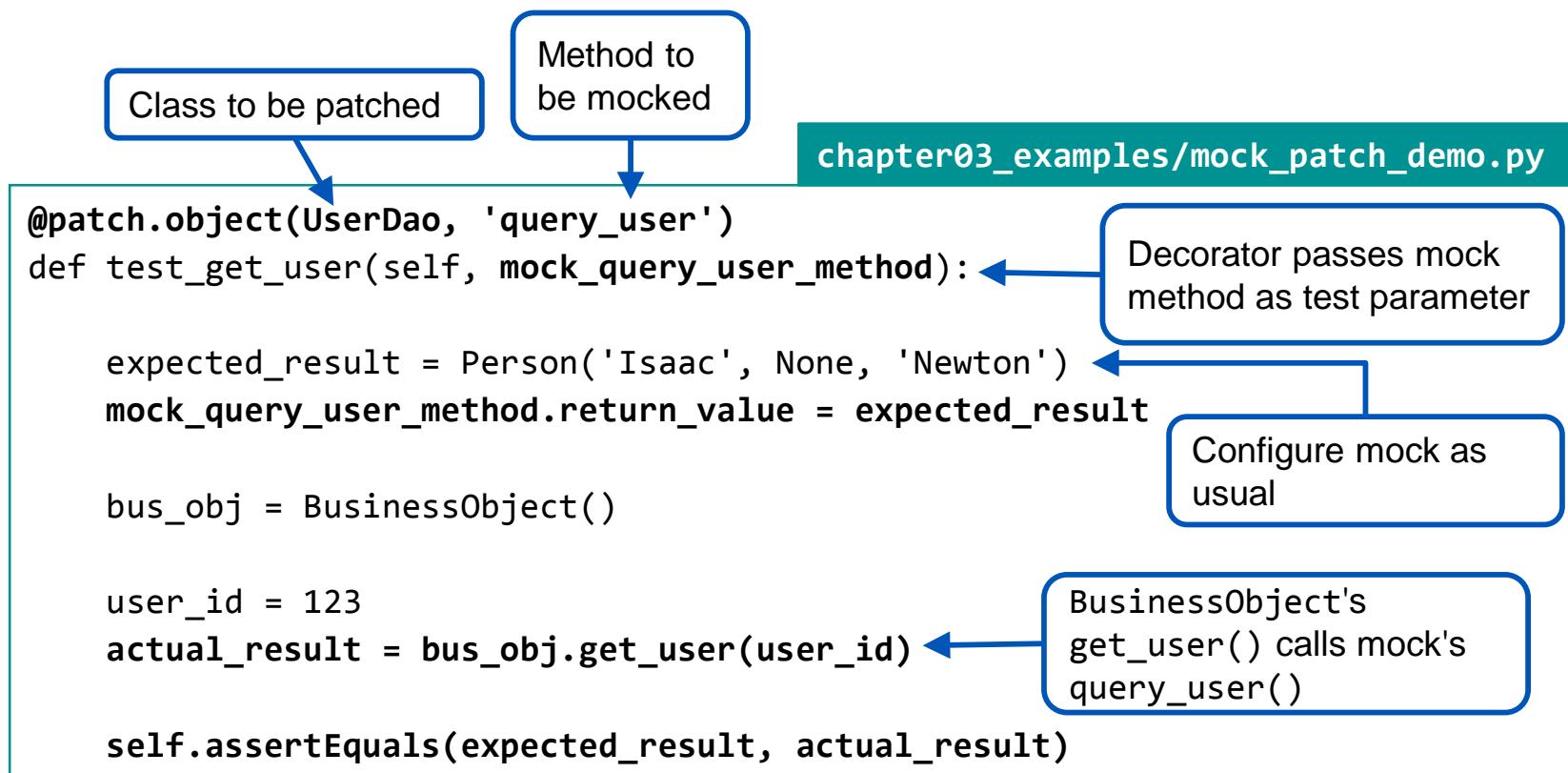
## Mock Module Advanced Features

- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# Patching Classes With `@patch.object`

- ▶ **`@patch.object` decorator simplifies test code**
  - `@patch.object` creates Mock for one method instead of the entire class
    - `@patch.object` performs monkey patching for that one method only
  - No need for you to explicitly call Mock constructor



# Patching Classes With @patch

- ▶ **Problem: BusinessObject constructor still calls UserDao constructor**
  - UserDao constructor attempts to connect to database
  - To avoid database errors, test cases must set up database
- ▶ **Solution: Use @patch decorator to replace UserDao class completely**
  - Any calls to class constructor will create a Mock instead
  - @patch passes mock *class*, not mock *instance*, as test method parameter
    - You must call mock class's constructor to get a Mock object

```
@patch('models.bus_obj.UserDao')
def test_get_user_patch_dao(self, mock_user_dao_class):

 mock_dao = mock_user_dao_class() ← Call mock class constructor
 Parameter is mock class

 expected_result = Person('Isaac', None, 'Newton')
 mock_dao.query_user.return_value = expected_result

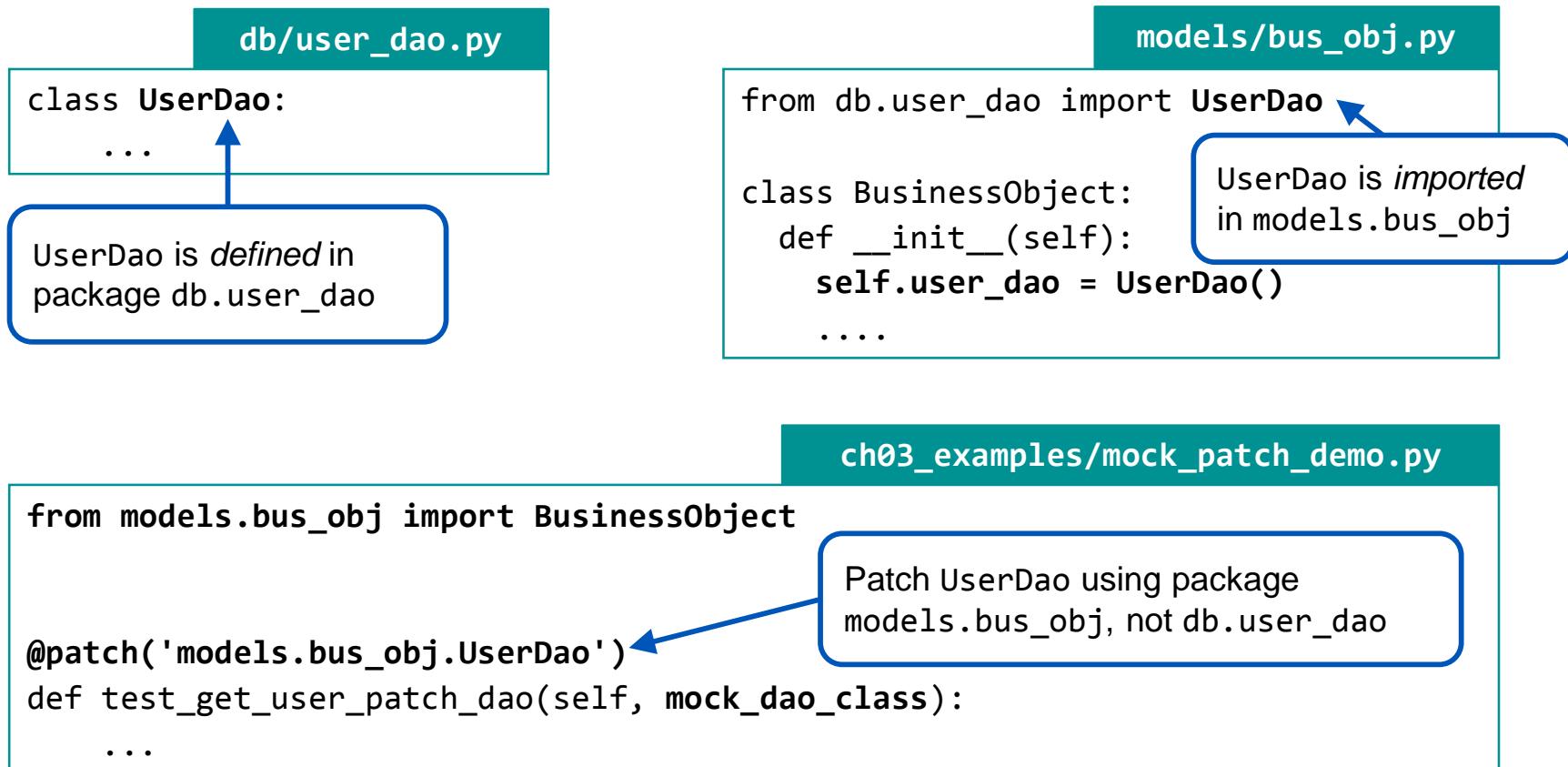
 bus_obj = BusinessObject() ← Business object never calls
 UserDao constructor

 actual_result = bus_obj.get_user(123)

 self.assertEqual(expected_result, actual_result)
```

# Patching Classes With @patch

- ▶ Patch a class where it is *imported*, not where it is *defined*
  - This is vital when using @patch



# MagicMock

---

- ▶ **Classes can define *magic* methods**
  - Called by Python when objects are used with operators or built-in functions

<code>__str__</code>	Called by built-in function <code>str()</code>
<code>__getitem__</code>	Called to get value from indexed instance: <code>value = obj[key]</code>
<code>__setitem__</code>	Called to set value in indexed instance: <code>obj[key] = value</code>
<code>__contains__</code>	Called by <code>in</code> operator: <code>if value in list:</code>

- ▶ **Mocks created with Mock class do not implement magic methods**
- ▶ **MagicMock is a Mock subclass that implements magic methods**
  - Use MagicMock to mock classes that override magic methods
  - You can also use MagicMock mock built-in types (`dict`, `list`, etc.)

# Testing With MagicMock

magic\_mock\_demo.py

```
class Department:
 def __init__(self):
 self.persons = {}

 def add_employee(self, person):
 self.persons[person.id] = person

 def get_employee(self, id):
 return self.persons[id]

class TestDepartment(TestCase):
 def test_add_employee(self):
 dept = Department()
 mock_dict = MagicMock()
 dept.persons = mock_dict
 emp = Person(2785, 'Bobby', 'James', 'Fischer')
 dept.add_employee(emp)
 mock_dict.__setitem__.assert_called_with(2785, emp)

 expected = Person(2690, 'Boris', 'Vasilievich', 'Spassky')
 mock_dict.__getitem__.return_value = expected
 actual = dept.get_employee(2690)
 self.assertEqual(expected, actual)
```

Class uses built-in dict

Test goal: verify that Department methods use dict attribute appropriately

Replace built-in dict with MagicMock

Verify magic method was called

# Summary of Techniques for Mocking

---

- The following table summarizes the techniques for mocking objects

Task	Technique
Replace a class completely	<code>@patch()</code>
Replace one method of a class	<code>@patch.object()</code>
Replace selected methods	<code>Mock()</code> , <code>MagicMock()</code>
Replace magic methods	<code>MagicMock()</code>

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features

## The Jython Interpreter

- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



- ▶ **Jython is a Python 2.7 interpreter written in Java**
  - Java applications can call Jython scripts
  - Jython scripts can use Java classes, including standard Java library classes
- ▶ **Jython interpreter compiles Python source to Java byte code**
  - Java byte code is executed by Java Virtual Machine (JVM)
  - Performance may be much better than CPython
- ▶ **Useful if you want Python scripts to be able to use Java libraries**
  - Or if you want Java code to call Python scripts
- ▶ **In concurrent applications, Jython threads are mapped to Java threads**
  - Multiple Jython threads can execute on multiple CPU cores in parallel
  - Can result in significant performance gains compared with standard Python
    - CPython threads always share a single CPU core
- ▶ **Download installer from <http://www.jython.org/downloads.html>**
  - Jython book: <http://www.jython.org/jythonbook/en/1.0/>

# Using Standard Java Classes in Jython

## ► Example interactive session with Jython

```
> jython Start Jython interpreter
>>> from java.util import Date Import Java's standard Date
>>> d1 = Date() class and instantiate
>>> d1
Sun Jun 28 11:57:32 EDT 2015
>>> repr(d1) Python built-in functions
u'Sun Jun 28 11:57:32 EDT 2015' work with Java objects
>>> str(d1)
'Sun Jun 28 11:57:32 EDT 2015'
>>> dir(d1)
['__class__', '__eq__', '__ge__', 'after', 'before', ...]
>>> d2 = Date() Compare Java Date instances
>>> d2 > d1 using Python operator
True
```

# Using Custom Java Classes in a Jython Script

- Jython script can import custom Java classes

HelloWorld.java

```
package com.foo;

public class HelloWorld {
 public void hello() {
 System.out.println("Hello World!");
 }

 public void hello(String name) {
 System.out.printf("Hello %s!\n", name);
 }
}
```

```
>>> from com.foo import HelloWorld
>>> h = HelloWorld()
>>> h.hello()
Hello World!
>>> h.hello('Python')
Hello Python!
```

Import Java HelloWorld class and instantiate

Call Java methods

# Importing Python Classes in Java Applications

- ▶ **Java application can import Jython classes**
  - Requires considerably more set up than accessing Java from Jython
  - We'll review a simple example here
- ▶ **Python class must implement Java interface**

BuildingType.java

```
package com.ltree;

public interface BuildingType {
 public String getBuildingName();
 public String getBuildingAddress();

 public int getBuildingId();
 public void setBuildingName(String name);

 public void setBuildingAddress(String address);
 public void setBuildingId(int id);
}
```

Java interface must define setters and getters for attributes



# Jython Class

## ► Jython class implements Java interface

building.py

```
from com.ltree import BuildingType
class Building(BuildingType):
 def __init__(self):
 self.name = None
 self.address = None
 self.id = -1
 def getBuildingName(self):
 return self.name
 def setBuildingName(self, name):
 self.name = name
 ...

```

Import Java interface

Implement Java interface

Constructor can't take arguments

Attributes are accessed with getter and setter methods

Define getters and setters for other attributes

# Java Application

## ► Create Jython object using object factory

Main.java

```
package com.ltree;
import org.plyjy.factory.JythonObjectFactory;
public class Main {
 public static void main(String[] args) {
 // Obtain an instance of the object factory
 JythonObjectFactory factory = JythonObjectFactory.getInstance();

 // Call createObject() method on object factory by
 // passing Java interface and name of Jython module/class
 BuildingType building = (BuildingType) factory.createObject(
 BuildingType.class, "Building");

 // Populate the object with values using the setter methods
 building.setBuildingName("BUILDING-A");
 building.setBuildingAddress("100 MAIN ST.");
 building.setBuildingId(1);

 System.out.println(building.getBuildingName());
 ...
 }
}
```

# Numba JIT Compiler

---

- ▶ **Numba framework supports JIT compilation for selected functions**
  - Like PyPy, Numba's JIT compiler translates your Python code as it executes
  - But Numba runs with standard CPython implementation
    - Has no dependency on PyPy
- ▶ **Add Numba's @jit decorator to functions to be JIT-compiled**
  - Numba uses reflection to infer argument types when function is called

```
from numba import jit

@jit
def sum2d(arr):
 ...
 big_array = ...
 result = sum2d(big_array)
```

- ▶ **Numba is a third-party module available on PyPI**
  - Installation: pip install numba
  - Documentation: <http://numba.pydata.org>

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter

## The **Asyncio** Module

- ▶ Synchronizing Threads With Queues
- ▶ The `sched` Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With `pdb`
- ▶ Hands-On Exercise B.1



# AsyncIO Library

---

- ▶ **AsyncIO**
  - Python 3.4 added `asyncio` library
    - Implemented with *coroutine functions*
    - Coroutine: a function that can be suspended and resumed at many points
  - Python 3.7 added high-level API with new keywords `async` and `await`
    - `async`: identifies an *awaitable* object (function, loop, or context manager)
    - `await`: suspends execution on an awaitable object
- ▶ **High-level API hides details of event loop**
  - Make it easier to write asynchronous code
- ▶ **By itself, AsyncIO runs your coroutines in a *single thread***
  - AsyncIO by itself won't make your code run faster
- ▶ **Libraries built on AsyncIO can use Thread pools to improve performance**
  - `aiofiles` module provides asynchronous file I/O functions
  - `aiohttp` and `aiodns` modules provide asynchronous HTTP functions

# AsyncIO Example

## ► **asyncio\_zip\_files.py** zips files concurrently

- `asyncio.gather()` blocks until all tasks are complete

```
async def read_file(file):
 async with aiofiles.open(file, "rb") as f:
 return await f.read()

def compress(data):
 return gzip.compress(data)

async def write_file(file, contents):
 async with aiofiles.open(file, "wb") as f:
 await f.write(contents)

async def zip_file(file):
 contents = await read_file(file)
 zipped_contents = compress(contents)
 await write_file(file + ".gzip", zipped_contents)

async def main(files):
 tasks = []
 for file in files:
 task_obj = zip_file(file)
 tasks.append(task_obj)
 await asyncio.gather(*tasks)

asyncio.run(main(...))
```

Define coroutine `read_file()`

Call `async open()` instead of native `open()`

`await` suspends `read_file()` coroutine until read operation is complete

Suspend `write_file()` coroutine until write operation is complete

Call to coroutine creates a Task object but doesn't execute code

`asyncio.gather()` schedules Task objects to be executed

# AsyncIO Example Output

## ► Output of previous example

```
> python asyncio_zip_files.py input_file_*.txt
starting to read input_file_1.txt
starting to read input_file_2.txt
starting to read input_file_3.txt
starting to read input_file_4.txt
starting to read input_file_5.txt
 done reading input_file_2.txt
starting to write input_file_2.txt.gzip
 done reading input_file_4.txt
starting to write input_file_4.txt.gzip
 done reading input_file_5.txt
starting to write input_file_5.txt.gzip
 done reading input_file_1.txt
starting to write input_file_1.txt.gzip
 done reading input_file_3.txt
starting to write input_file_3.txt.gzip
 done writing 1063 bytes to input_file_2.txt.gzip
 done writing 970638 bytes to input_file_4.txt.gzip
 done writing 9445788 bytes to input_file_5.txt.gzip
 done writing 16680122 bytes to input_file_1.txt.gzip
 done writing 21177138 bytes to input_file_3.txt.gzip
```

Coroutine `read_file()` is called for 5 input files

Reading of files 4 and 5 is complete before files 1 and 3

Writing to file 5 before reading file 1 is complete

Writing of files 4 and 5 is complete before files 1 and 3

# Synchronous File Zipping Code Output

- **sync\_zip\_files.py performs synchronous file zipping operations**
  - Compare output with asynchronous example

```
> python sync_zip_files.py input_file_*.txt
starting to read input_file_1.txt
 done reading input_file_1.txt
starting to write input_file_1.txt.gzip
 done writing 16680122 bytes to input_file_1.txt.gzip
starting to read input_file_2.txt
 done reading input_file_2.txt
starting to write input_file_2.txt.gzip
 done writing 1063 bytes to input_file_2.txt.gzip
starting to read input_file_3.txt
 done reading input_file_3.txt
starting to write input_file_3.txt.gzip
 done writing 21177138 bytes to input_file_3.txt.gzip
starting to read input_file_4.txt
 done reading input_file_4.txt
starting to write input_file_4.txt.gzip
 done writing 970638 bytes to input_file_4.txt.gzip
starting to read input_file_5.txt
 done reading input_file_5.txt
starting to write input_file_5.txt.gzip
 done writing 9445788 bytes to input_file_5.txt.gzip
```

All operations are performed synchronously

# Asynchronous Downloads from Multiple URLs

- **asyncio\_url\_download.py** downloads from multiple URLs concurrently
  - Note use of `asyncio.as_completed()` to schedule Tasks
  - As each Task completes, `as_completed()` returns a Future for the Task
  - The Future will contain the Task's result when the Task completes

```
import aiohttp, aiofiles

async def get_html(session, url):
 resp = await session.request('GET', url=url)
 data = await resp.text()
 return data

async def main(urls):
 async with aiohttp.ClientSession() as session:
 tasks = [get_html(session, url) for url in urls]
 for future in asyncio.as_completed(tasks):
 next_html = await future
 filename = ...
 async with aiofiles.open(filename, "w") as f:
 await f.write(next_html)

wiki_urls = ...
asyncio.run(main(wiki_urls))
```

The diagram illustrates the execution flow of the `main()` function. It starts with the `async with aiohttp.ClientSession() as session:` block, which is annotated with a callout box: "Call `get_html()` to create a Task for each URL". Inside this block, the `tasks` variable is assigned a list of `get_html()` tasks for each URL. This is followed by a `for` loop that iterates over the `tasks` list using `asyncio.as_completed(tasks)`. Each iteration is annotated with a callout box: "Schedule all Tasks, then process results greedily as Tasks complete". Inside the loop, the `next_html` variable is assigned the result of the completed Task, which is annotated with another callout box: "Submit HTTP GET request asynchronously". Finally, the `filename` is set and an `aiofiles.open()` block is executed to write the downloaded HTML to a file, which is annotated with a callout box: "Write downloaded text asynchronously".

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module

## Synchronizing Threads With Queues

- ▶ The `sched` Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With `pdb`
- ▶ Hands-On Exercise B.1



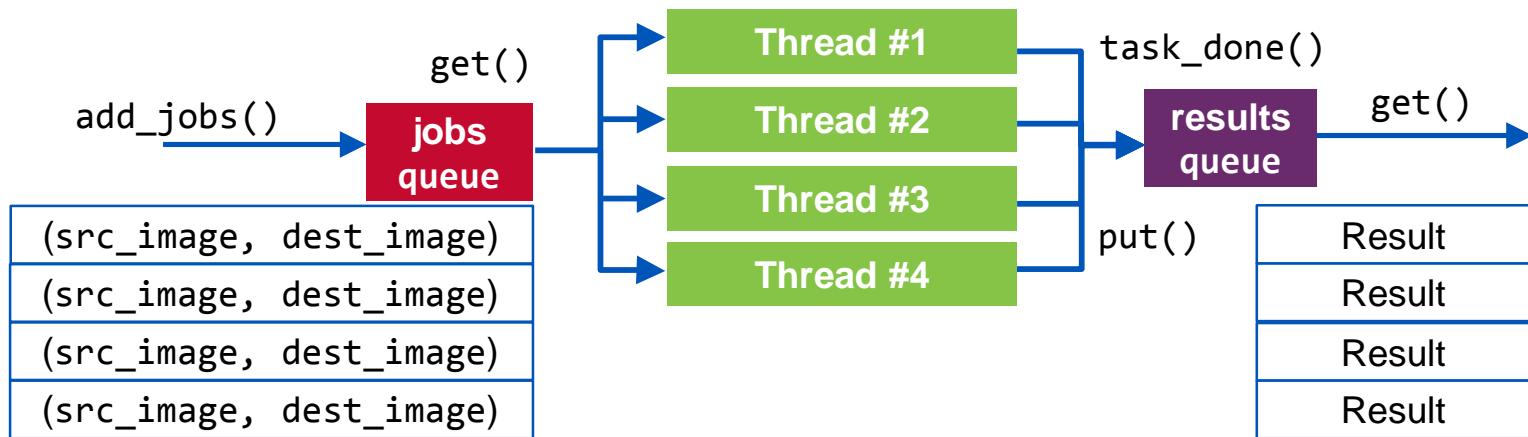
# High-Level Thread Synchronization

---

- ▶ **Common use case: Multiple threads process data as it arrives**
  - Synchronizing with locks can be error prone and difficult to test
  - Acquiring and releasing locks may lead to deadlock conditions
- ▶ **Solution: queue.Queue class**
  - As data arrives, application adds it to Queue
  - Threads remove data from Queue and process it
  - Advantage of using Queue class: Queue implements locking internally
    - You don't need to add explicit locking to prevent race conditions
- ▶ **Queues are best choice for inter-thread communication and coordination**
  - Applications are easier to design, more readable, more reliable
- ▶ **Example: Multithreaded RSS feed reader**
  - Application downloads content from many RSS feeds
  - Each feed will be handled by a thread
  - Several downloads can execute concurrently
- ▶ **This application is I/O-bound, so multithreading yields good performance**

# Synchronizing Threads With Queues

- ▶ Application will create two queues
  - Input queue (jobs): Each item is the URL of an RSS feed to download
  - Output queue (results): Each item is feed content converted to HTML
- ▶ Main thread starts several child threads
  - Each thread gets an item from the jobs queue
    - If jobs queue is empty, thread waits
  - After thread processes an item, it puts its output on results queue
    - Thread notifies jobs queue by calling `task_done()`
    - Each call to `get()` must have a corresponding call to `task_done()`
  - Main thread waits for jobs queue to empty, then gets items from results queue



# Queue Example

```
def main():
 jobs_q = queue.Queue()
 results_q = queue.Queue()
 create_threads(limit, jobs_q, results_q, how_many_threads)
 todo = add_jobs("whatsnew.dat", jobs_q)
 process(todo, jobs_q, results_q, how_many_threads)

def create_threads(limit, jobs_q, results_q, how_many_threads):
 for _ in how_many_threads:
 thread = Thread(target=worker, args=(limit, jobs_q, results_q))
 thread.daemon = True
 thread.start()

def worker(limit, jobs_q, results_q):
 while True:
 try:
 feed = jobs_q.get()
 ok, result = Feed.read(feed, limit)
 results_q.put(result)
 finally:
 jobs_q.task_done()
```

Put result on output queue

Create queues for inputs and outputs

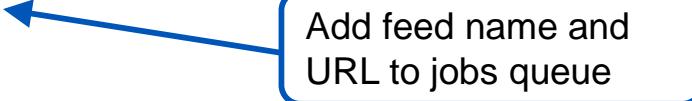
Each thread executes worker

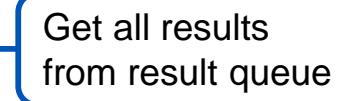
Daemon threads don't prevent interpreter from exiting

Get next item from input queue

Tell input queue that processing on item is complete

# Queue Example

```
def add_jobs(filename, jobs_q):
 for todo, feed in enumerate(Feed.iter(filename), start=1):
 jobs_q.put(feed)
 return todo
 
 Add feed name and URL to jobs queue

def process(todo, jobs_q, results_q, how_many_threads):
 jobs_q.join()
 
 join() waits for all inputs to be processed
 done, filename = output(results_q)
 ...
 
 Get all results from result queue

def output(results_q):
 filename = ... # open output file
 while not results_q.empty(): # Safe because all jobs have finished
 result = results_q.get()
 ... # write result to output file
 
 # return number of results processed
```

# Storing Thread-Specific State

- Your application may need to store data specific to current thread
  - Create thread-local storage using `threading.local()`
  - Thread local object is not shared with other threads

```
class MySocketConn:
 def __init__(self):
 self.local = threading.local()

 def read_socket(self):
 self.local.sock = socket(...)
 ...

 def read_conn(conn):
 conn.read_socket()
 ...

conn_obj = MySocketConn()
Thread(target=read_conn, (conn_obj,)).start()
Thread(target=read_conn, (conn_obj,)).start()
```

Each thread gets a unique local object

Each thread has its own socket

read\_conn is called by multiple thread

One MySocketConn instance is shared by multiple threads

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues

## The sched Module

- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# sched Module

- sched module provides basic support for event scheduling

```
import sched, time

def do_task(name):
 print('Running ' + name + ' at ' + time.strftime("%X %x"))
 print('Starting at ' + time.strftime("%X %x"))
scheduler = sched.scheduler(time.time, time.sleep)
scheduler.enter(2, 1, do_task, ('first',))
scheduler.enter(3, 1, do_task, ('second',))
scheduler.run()
```

Run do\_task()  
after 3-second delay

Priority: if tasks run  
at same time,  
highest priority runs  
first

Pass standard  
time functions as  
arguments

Argument list to  
do\_task()

- Output

```
Starting at 15:53:57 07/02/15
Running first at 15:53:59 07/02/15
Running second at 15:54:00 07/02/15
```

# sched Module

► **sched.event returns an event object**

- May be used to cancel an action

```
event = scheduler.enter(60, 1, do_task, ('first',))
...
event.cancel()
```

► **sched module can schedule action for a specific time**

```
start_str = '2015-11-10 16:16'
fmt = '%Y-%m-%d %H:%M'
start_time = time.mktime(time.strptime(start_str, fmt))

scheduler.enterabs(start_time, priority, do_task, ('abs time',))
scheduler.run()
```

► **sched is not as flexible as UNIX cron or MS Windows at commands**

- datetime module has utilities for calculating time until a given date, etc.
- Third-party scheduling modules on PyPI: apscheduler, schedule

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module

## Handling XML REST Responses

- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# Sending and Consuming XML

## ► XML is a common format for more complex REST services

- Example: Google news feeds in RSS and AtomPub formats
- <http://news.google.com/news/feeds?q=python&output=rss>

The screenshot shows the Postman application interface. At the top, there's a header bar with 'GET' selected, the URL 'http://news.google.com/news/feeds?q=python&output=rss', and a 'Send' button. Below the header are tabs for 'Authorization', 'Headers (1)', 'Body', 'Pre-request script', and 'Tests'. The 'Authorization' tab is checked, showing a basic auth token: 'Basic c3R1ZGVudDpzdHVkZW50cHc='. The 'Body' tab is selected, displaying the XML response from the API. The XML content is as follows:

```
1 <rss version="2.0">
2 <channel>
3 <generator>NFE/1.0</generator>
4 <title>python - Google News</title>
5 <link>http://news.google.com/news?hl=en&pz=1&ned=us&q=python</link>
6 <language>en-US</language>
7 <webMaster>news-feedback@google.com</webMaster>
8 <copyright>© 2015 Google</copyright>
9 <pubDate>Thu, 23 Jul 2015 02:55:50 GMT</pubDate>
10 <lastBuildDate>Thu, 23 Jul 2015 02:55:50 GMT</lastBuildDate>
11 <image>
12 <title>python - Google News</title>
13 <url>http://www.gstatic.com/news-static/img/logo/en_us/news.gif</url>
14 <link>http://news.google.com/news?hl=en&pz=1&ned=us&q=python</link>
15 </image>
16 <item>
17 <title>Report python sightings - Marco Island Sun Times - Marco Island Sun Times</title>
18 <link>http://news.google.com/news/url?sa=t&fd=R&ct2=us&usg=AFQjCNF8DtZE7x9eH8aEuzfrdXKhTmttzQ&clid=c3a7d30bb8a4878e06b80cf16
```

# Parsing and Creating XML Documents

- ▶ **Simple API for XML: Standard `xml.etree.ElementTree` module**
  - `ElementTree` can process XML from a file or a string
  - It can also create and write XML
- ▶ **Example XML response from REST service**

```
<task>
 <title>Get jetpack serviced</title>
 <description>Thruster response is sluggish</description>
 <done>False</done>
 <id>143</id>
</task>
```

# Parsing XML Documents

## ► Function that reads XML from a string

- `ElementTree.fromstring()` returns Element for root element of XML

```
def parse_task_xml(self, xml_str): # rest_server_xml.py
 task_element = ElementTree.fromstring(xml_str)
 title = task_element.findtext('./title')
 desc = task_element.findtext('./description')
 done = task_element.findtext('./done')
 id = task_element.findtext('./id')
 if done: # if done is None, just return it
 done = (done == 'True') # otherwise, convert string to bool
 return title, desc, done, id
```

## ► ElementTree uses **XPath** expressions to locate XML elements

- XPath: Query language for selecting nodes (elements) in XML documents
- Standard defined by World Wide Web Consortium (W3C)

## ► XPath syntax is similar to file path syntax

- To locate `<task>` element's `<title>` child element: `/task/title`
- To locate current element's `<done>` child element: `./done`

# Reading an XML REST Response

## ► Example of code to read XML response from GET request

```
task_id = 143
url = base_url + '/' + task_id
creds = ('student', 'studentpw')
http_hdrs = {'Accept': 'application/xml'}
```

```
r = requests.get(url, auth=creds, headers=http_hdrs)
```

```
if r.status_code != 200:
 raise RuntimeError('Problem getting XML for task')
```

```
xml_result = r.text
```

```
title, desc, done, id = parse_task_xml(xml_result)
```

Get XML text from  
response body

Pass XML from response body to  
parse\_task\_xml() function on previous slide

# Creating XML

## ► Steps to create XML string

- Use ElementTree methods to create tree of elements
- Use ElementTree.tostring() to serialize tree of elements to string

```
def create_task_xml(self, task):
 task_element = ElementTree.Element('task') ← Create root element
 <task>
 for field in ('title', 'description',
 'done', 'id'):
 if task.get(field) is not None:
 element = ElementTree.SubElement(task_element, field)
 element.text = str(task[field])
 return ElementTree.tostring(task_element, encoding='unicode')
```

Create subelements of <task>

Serialize element tree to  
string

# Adding XML to POST Request Body

## ► Example of POST request with XML in body

- Pass XML data in request body with data parameter

```
task = {
 'title': 'Teach cat Spanish',
 'description': 'Needs to work on irregular verbs',
 'done': False
}
http_hdrs = {'Content-Type': 'application/xml'}
creds = ('student', 'studentpw')

xml = create_task_xml(task) ← Pass dictionary to function
 on previous slide

r = requests.post(url, auth=creds, headers=http_hdrs, data=xml)

if r.status_code != 202:
 raise RuntimeError("Can't create task")
```

Add XML string to body of POST request

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses

## Porting Python 2 Code to Python 3

- ▶ Debugging With pdb
- ▶ Hands-On Exercise B.1



# Porting Python 2 Modules to Python 3

---

- ▶ **Python 3 provides tools to port Python 2 modules**
  - See “Porting Python 2 Code to Python 3 Code” in Python Standard Documentation under “Python HOWTOs”
- ▶ **2to3 tool converts Python 2 code to Python 3**
  - Code may require no other changes to run under Python 3
- ▶ **Default operation: Write changes to standard output**

```
2to3 my_module.py
```

- ▶ **Add -w option to write changes back to source file**

```
2to3 -w my_module.py
```

# Supporting Both Python 2 and Python 3

---

- ▶ After running 2to3, your code no longer runs under Python 2
- ▶ Instead, you can modify code to run under both Python 2 and Python 3
- ▶ Strategy
  1. Only worry about supporting Python 2.7 (not earlier Python 2 versions)
  2. Use caniusepython3 to find dependencies incompatible with Python 3
  3. Make sure you have good test coverage (including automated unit tests)
  4. Learn the differences between Python 2 and 3
  5. Use Modernize or Futurize modules to update your code
  6. Use Pylint to catch problems with your Python 3 support
  7. Use continuous integration to stay compatible with Python 2 and 3
- ▶ For details, see <https://docs.python.org/3/howto/pyporting.html>

# Modernize Module

- ▶ **Modernize: More conservative than Futurize**
  - Targets subset of Python compatible with both Python 2 and Python 3
- ▶ **Usage**

```
pip install modernize six
python-modernize -w example.py
```

- ▶ **Modernize relies on Six compatibility library**
  - Six replaces incompatible Python features with wrappers
  - Wrappers work in both Python 2 and Python 3

```
symbols = {'LTRE': 'Learning Tree', 'PYPL': 'PayPal'}
vals = symbols.viewvalues() # OK in Python 2, fails in Python 3
import six
vals = six.viewvalues(symbols) # OK in Python 2 and Python 3
```

- ▶ **For details, see**  
<http://python-modernize.readthedocs.org/en/latest/fixers.html>

# Futurize Modules

---

- ▶ **Futurize: More aggressive than Modernize**
  - Tries to make Python 3 idioms and practices exist in Python 2
  - Uses standard `__future__` module to backport Python 3 features
- ▶ **Apply Futurize incrementally**
  - Each stage makes more changes to Python 2 code
- ▶ **Stage 1: futurize --stage1 -w \*.py**
  - After stage 1, verify modified code still runs under Python 2
  - Code won't yet run under Python 3
- ▶ **Stage 2: futurize --stage2 -w \*.py**
  - After stage 2, re-run test suite with Python 3
  - Some changes won't work under Python 2
- ▶ **Finally, add wrappers from `__future__` to re-enable Python compatibility**
  - Rerun test suite with Python 2 and 3
- ▶ See [http://python-future.org/automatic\\_conversion.html](http://python-future.org/automatic_conversion.html)

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3

## Debugging With pdb

- ▶ Hands-On Exercise B.1



# The pdb Debugger

---

- ▶ **Python's standard pdb module provides an interactive debugger**
  - Supports setting breakpoints, including conditional breakpoints
  - Can single-step through source code
  - Allows inspection of stack frames and source code listing
  - Evaluates arbitrary Python code in the context of any stack frame
- ▶ **pdb also supports post-mortem debugging**
  - Can examine state of application after a program crash
- ▶ **pdb has a simple command line interface**
  - Not as intuitive as graphical debugger in PyCharm
  - But pdb has advantages
    - Always available in Python
    - Lightweight, small memory footprint
    - Works over ssh connections with remote systems

# pdb Command Summary

---

<b>h</b> [ <i>cmd</i> ]	Display help (for <i>cmd</i> )
<b>l</b> [ <i>begin[, end]</i> ]	List module source code
<b>ll</b>	Long listing of current function
<b>expr</b>	Evaluate expression in context of current execution state
<b>!stmt</b>	Execute a Python statement
<b>(empty Line)</b>	Re-execute last pdb command
<b>b</b> <i>lineno</i>	Set breakpoint on <i>lineno</i>
<b>b</b> <i>function</i>	Set breakpoint on first line of <i>function</i>
<b>c</b>	Continue execution until next breakpoint
<b>cl</b> [ <i>num</i> ]	Clear breakpoint number <i>num</i>
<b>n</b>	Step over next statement
<b>s</b>	Step into next statement
<b>r</b>	Resume execution until current function returns
<b>run</b>	Restart program execution without losing breakpoints
<b>w</b>	Shows where you are in the execution stacktrace
<b>q</b>	Quit pdb

# Code Example

- We'll use the following code for a pdb example

```
"""pdb_demo.py - Example of pdb"""
class PdbDemo:

 def __init__(self, name, num_loops):
 self._name = name
 self._count = num_loops

 def count(self):
 for i in range(self._count):
 print(i)
 return '{} executed {} of {} loops'\
 .format(self._name, i+1, self._count)

if __name__ == '__main__':
 obj_name = 'pdb demo'
 print('Starting ' + obj_name + '...')
 pd = PdbDemo(obj_name, 5)
 result = pd.count()
 print(result)
```

# Demo of pdb

Do Now

- To run pdb script from command line, execute `python -m pdb`

List current function

```
> cd \crs1906\examples\appB_examples
> python -m pdb pdb_demo.py ← Execute pdb module
> c:\...\pdb_demo.py(1)<module>()
-> """pdb_demo.py - Example of pdb"""
(Pdb) 11
```

Set breakpoint at line 16

```
1 -> """pdb_demo.py - Example of pdb"""
...
15 if __name__ == '__main__':
16 obj_name = 'pdb demo'
17 print('Starting ' + obj_name + '...')

(Pdb) b 16
```

Continue execution

```
Breakpoint 1 at c:\...\pdb_demo.py:16 ← Location in module
```

Step over next statement

```
(Pdb) c
> c:\...\pdb_demo.py(16)<module>()
-> obj_name = 'pdb demo'

(Pdb) n
> c:\...\pdb_demo.py(17)<module>()
-> print('Starting ' + obj_name + '...')
```

Next statement to be executed

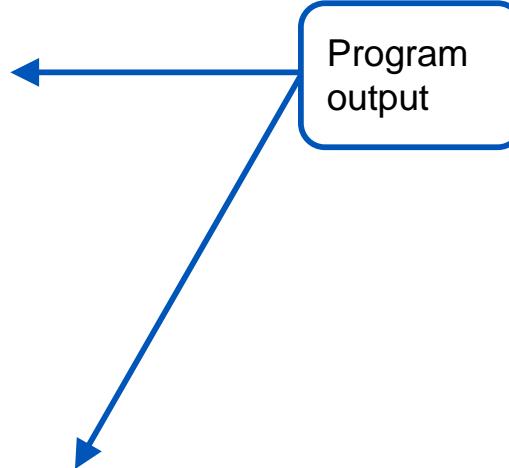
# Demo of pdb

Do Now

Evaluate an expression  
(variable value)

```
(Pdb) obj_name
'pdb demo'
(Pdb) n
Starting pdb demo...
> c:\...\pdb_demo.py(18)<module>()
-> pd = PdbDemo(obj_name, 5)
(Pdb) n
> c:\...\pdb_demo.py(19)<module>()
-> result = pd.count()
(Pdb) n
0
1
2
3
4
> c:\...\pdb_demo.py(20)<module>()
-> print(result)
(Pdb) result
'pdb demo executed 5 of 5 loops'
```

Program  
output



# Starting pdb From Interactive Interpreter

- ▶ You can start pdb from an interactive Python interpreter
  - Useful for debugging modules with no main function
  - Example: Debug PdbDemo class's count method

```
> python
>>> import pdb
>>> import pdb_demo
>>> pdb.run('pdb_demo.PdbDemo("from python", 5).count()')
> <string>(1)<module>()
(Pdb) s
--Call--
> c:\...\pdb_demo.py(5)__init__()
-> def __init__(self, name, num_loops):
(Pdb) l
 1 """pdb_demo.py - Example of pdb from chapter 4"""
 :
(Pdb) b 10
Breakpoint 1 at c:\...\pdb_demo.py:10
(Pdb) c
> c:\...\pdb_demo.py(10)count()
-> for i in range(self._count):
```

Start Python, import pdb,  
import your module

Instantiate class and call  
a method

Step to start executing  
your code

Debug as usual

# Starting pdb From Your Code

- **pdb module defines set\_trace() method that you call from your code**
  - Useful for debugging code with complex setup

```
import pdb
class PdbDemo:
 ...
 def count(self):
 for i in range(self._count):
 pdb.set_trace()
 print(i)
```

Import pdb module

Call set\_trace method

- **pdb starts automatically when execution reaches set\_trace() call**

```
> python pdb_demo_trace.py
> c:\...\pdb_demo_trace.py(14)count()
-> print(i)
(Pdb) 1
12 for i in range(self._count):
13 pdb.set_trace()
14 -> print(i)
```

Run application as usual

pdb starts at call to set\_trace

# Post-Mortem Debugging

- **pdb lets you examine application state after a program failure**
  - Python's `-i` option starts interactive interpreter when application terminates
  - `pdb.pm()` runs a pdb post-mortem on the crashed program

```
> python -i pdb_demo_pm.py ← Start Python with -i option
...
Traceback (most recent call last):
 File "C:\...\pdb_demo_pm.py", line 13, in count
 .format(self.name, i+1, self._count)
AttributeError: 'PdbDemo' object has no attribute 'name' ← Oops!
>>> import pdb
>>> pdb.pm() ← Call pdb.pm()
> c:\...\pdb_demo_pm.py(13)count()
-> .format(self.name, i+1, self._count)
(Pdb) 1
 12 return '{} executed {} of {} loops'\n 13 -> .format(self.name, i+1, self._count)
(Pdb) self
 _name=demo, _count=5 ← pdb calls object's
 (Pdb) i
 4
```

Examine state of application at time of failure

# Debugging Unit Tests

- ▶ Pytest's `--pdb` option starts pdb on every test case failure

```
> pytest --pdb test_atom_news_feed_parser.py
...
 assert expected_results[i][name] == actual_result[name]
E assert 'https://t0.g...ges?q=tbn:...' == 'https://t0.gs...
E - https://t0.gstatic.com/images?q=tbn:...
E + https://t0.gstatic.com/images?q=tbn:...<div class=
E ?
 +++++++
```

Run Pytest with  
--pdb option

Pytest flags  
failed test case

```
test_rss_news_feed_parser_no_class.py:19: AssertionError
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
> ...\\test_rss_news_feed_parser_no_class.py(19)...
-> assert expected_results[i][name] == actual_result[name]
(Pdb) actual_result[name]
'https://t0.gstatic.com/images?q=tbn:...<div class='
(Pdb)
```

Pytest starts pdb  
at test failure

# Contents

---

- ▶ Introspection and Metaclasses
- ▶ State Design Pattern
- ▶ Factory Method Design Pattern
- ▶ Mock Module Advanced Features
- ▶ The Jython Interpreter
- ▶ The Asyncio Module
- ▶ Synchronizing Threads With Queues
- ▶ The sched Module
- ▶ Handling XML REST Responses
- ▶ Porting Python 2 Code to Python 3
- ▶ Debugging With pdb

## Hands-On Exercise B.1



## Hands-On Exercise B.1

In your Exercise Manual, please refer to  
**Hands-On Exercise B.1: Debugging With pdb**



# Objectives

---

- ▶ Introspect class attributes at runtime
- ▶ Use the State design pattern to build a class whose behavior changes based on its inputs
- ▶ Decouple object creation from object usage with the Factory Method design pattern
- ▶ Write test cases with advanced features of the Mock module
- ▶ Replace the standard CPython implementation with Jython
- ▶ Simplify concurrent applications with the AsyncIO library
- ▶ Synchronize threads using Queues
- ▶ Schedule recurring events using the sched module
- ▶ Interact with an XML-based REST service
- ▶ Port Python 2 code to Python 3
- ▶ Debug applications with Python's standard pdb module