Learning Tree International

Cours 458G

Python Programming Introduction

458G/CN(2)/A.1/609/A.0

par Frank Schmidt

Éditeur technique :
Chris Czarnecki

EDUCATION IS OUR BUSINESS®

Course 468G

Python Programming Introduction

Learning Tree International

Training You Can Trust®

458G/CN(2)/A.1/609/A.0

PROTECTION JURIDIQUE DES PROGRAMMES D'ORDINATEUR

LOI N° 94 361 DU 10/05/94

"TOUTE COPIE DE LOGICIEL EST FORMELLEMENT INTERDITE. LE NON RESPECT DE CETTE DISPOSITION ENTRAÎNE L'EXCLUSION IMMÉDIATE DU STAGIAIRE ET PEUT DONNER LIEU À DES POURSUITES PÉNALES.

L'ANIMATEUR PEUT AUTORISER LES STAGIAIRES À COPIER DES EXERCICES RÉALISÉS, À CONDITION QU'AUCUNE PARTIE DU PROGRAMME EXÉCUTABLE, QUI FAIT L'OBJET D'UNE LICENCE D'UTILISATION, N'Y SOIT INCLUSE."

Extrait Article 25 — Réglement intérieur
(Article L.920.5.1 Code du travail)

Learning Tree International

RÈGLEMENT INTÉRIEUR
(Article L.920-5-1 du Code du Travail)

1. OBJET ET CHAMP D'APPLICATION

Article 1 : Objet

En application des dispositions du décret n° 91-1107 du 23 octobre 1991, portant application de l'article L 920-5-1 du Code du Travail, il a été établi, pour les stagiaires de Learning Tree International, un règlement intérieur qui a pour objet :

- de rappeler les principales mesures applicables en matière d'hygiène et de sécurité dans l'établissement,
- de fixer les règles applicables en matière de discipline et notamment la nature et l'échelle des sanctions applicables aux stagiaires ainsi que les droits de ceux-ci en cas de sanction.

Il sera complété ou précisé, le cas échéant, par des notes de services établies conformément à la loi dans la mesure où elles porteront prescriptions générales et permanentes dans les matières mentionnées à l'alinéa précédent.

Article 2 : Champ d'application

Ce règlement s'applique à tous les stagiaires de l'organisme.
Les dispositions du présent règlement sont applicables non seulement dans l'établissement proprement dit, mais aussi dans tout local ou espace accessoire à l'organisme, en particulier pour les formations dispensées par l'organisme en dehors de l'établissement.

2. HYGIÈNE ET SÉCURITÉ

Article 3 : Dispositions générales

En matière d'hygiène et de sécurité, chaque stagiaire doit se conformer strictement tant aux prescriptions générales qu'aux consignes particulières qui seront portées à sa connaissance par affiches, instructions, notes de services ou par tout autre moyen.

Article 4 : Respect d'autrui

Le comportement des stagiaires doit tenir compte du devoir de tolérance et de respect d'autrui dans sa personnalité et ses convictions et ne doit être en aucun cas violent physiquement ou moralement.

Article 5 : Boissons alcoolisées

L'introduction et la consommation des boissons alcoolisées pendant les heures de travail sont interdites, sauf circonstances exceptionnelles et avec l'accord de la Direction de l'établissement.

Article 6 : Tabac

En vertu du Décret du 15 novembre 2006 sur la protection des non-fumeurs, entré en vigueur le 1er février 2007, il est interdit de fumer dans tous les locaux de Learning Tree International.

Article 7 : Vols et dommages aux biens

Learning Tree International décline toute responsabilité pour les vols ou dommages aux biens pouvant survenir durant le stage de formation, au détriment des stagiaires.

Article 8 : Lieux de restauration

Les repas peuvent se prendre dans les locaux destinés à cet effet. Il est interdit d'amener des boissons chaudes ou froides dans les salles de stage équipées d'ordinateurs.

Article 9 : Tenue vestimentaire

Tout stagiaire doit être habillé de façon correcte dans les locaux de Learning Tree International.

Article 10 : Animal

Il est interdit d'introduire un animal dans les locaux de Learning Tree International.

Article 11 : Règles générales relatives à la protection contre les accidents

Tout stagiaire est tenu d'utiliser tous les moyens de protection individuels et collectifs mis à sa disposition pour éviter les accidents et de respecter strictement les consignes particulières données à cet effet.

Article 12 : Prévention médicale

Les stagiaires d'entreprise relèvent de leur entreprise et de ce fait, doivent être en règle avec celle-ci.

Article 13 : Stationnement et circulation des véhicules

Il n'est pas prévu de stationnement pour les véhicules des stagiaires. Toutefois Learning Tree International peut mettre à la disposition des stagiaires quelques places de parking dans le sous-sol. Une carte d'entrée et de sortie du parking peut être obtenue à l'accueil, contre le versement d'une caution qui sera rendue après remise de la carte à la fin du stage. Les règles de circulation, dans l'enceinte de Learning Tree International, devront être respectées.
Learning Tree International décline toute responsabilité pour les vols ou dommages aux voitures pouvant survenir durant le stage de formation.

Article 14 : Sécurité – Incendie

Toute personne présente dans les locaux de Learning Tree International prendra connaissance et appliquera les consignes de sécurité qui sont affichées sur les panneaux destinés aux informations générales.

Article 15 : Obligation d'alerte et droit de retrait

Tout stagiaire, ayant un motif raisonnable de penser qu'une situation présente un danger grave et imminent pour sa vie ou sa santé, a le droit de quitter les locaux du stage.
Toutefois, cette faculté doit être exercée de telle manière qu'elle ne puisse créer pour autrui une nouvelle situation de risque grave et imminent. Le stagiaire doit signaler immédiatement à l'animateur l'existence de la situation qu'il estime dangereuse.
Tout stagiaire ayant constaté une défaillance ou une anomalie dans les installations ou le fonctionnement des matériels est tenu d'en informer l'animateur ou le responsable de l'organisme de formation.
Tout accident, même bénin, doit être immédiatement déclaré à la Direction par la victime ou les témoins.

Article 16 : Urgence

Lorsque l'urgence le justifiera, la Direction de Learning Tree International prendra de nouvelles prescriptions qui recevront application immédiate.

Article 17 : Refus de se soumettre

Le refus du stagiaire de se soumettre aux prescriptions relatives à l'hygiène et à la sécurité pourra entraîner l'une des sanctions prévues au présent règlement.

3. ORGANISATION ET SUIVI DES STAGES

Article 18 : Emploi du temps – horaires

Learning Tree International arrête le calendrier des stages ; il est communiqué aux stagiaires.
Les horaires d'ouverture de l'établissement sont les suivants : ouverture à 8h30 le matin et fermeture à 17h30 le soir.
La ou les assistantes de formation apporteront aux stagiaires, le cas échéant, toutes précisions.
Seuls les stagiaires autorisés par écrit par un représentant habilité de Learning Tree International peuvent rester dans les locaux de l'établissement en dehors de ces horaires.

Article 19 : Assiduité, ponctualité, absences

Les stagiaires n'ont accès aux locaux de Learning Tree International que pour le déroulement des séances de formation.
Il est interdit d'introduire dans les locaux des personnes étrangères au stage.
Les stagiaires sont tenus de suivre les cours, séances d'évaluation et de réflexion, travaux pratiques, visites et stages en entreprise, et, plus généralement, toutes les séquences programmées par Learning Tree International, avec assiduité et sans interruption.
Des feuilles de présence quotidiennes seront utilisées par les responsables et devront être signées par les participants.
Learning Tree International est dégagé de toute responsabilité en cas d'absence non autorisée.

Article 20 : Travail et conditions de travail

La présence de chacun doit s'accompagner d'une participation active et de l'accomplissement d'efforts personnels.

Article 21 : Travaux pratiques

Pendant la durée du stage, des travaux pratiques sont réalisés sous le contrôle de l'animateur. Le stagiaire devra respecter les consignes de l'animateur. Le non-respect de cette clause est un des motifs qui peut donner lieu à l'exclusion immédiate du stagiaire.

Article 22 : Assurance des véhicules des stagiaires

Les stagiaires doivent vérifier que leur assurance personnelle couvre les risques encourus lorsqu'ils participent à une formation Learning Tree ou s'assurer qu'ils sont couverts par une assurance de leur employeur.

Article 23 : Enregistrement des cours

Il est formellement interdit, sauf dérogation expresse, d'enregistrer ou de filmer les séances de formation.

Article 24 : Méthodes pédagogiques, documentation et logiciels

Les méthodes pédagogiques, la documentation et les logiciels diffusés sont protégés au titre des droits d'auteur et ne peuvent être réutilisés pour un strict usage personnel ou diffusés par les stagiaires sans l'accord préaable et formel du responsable de l'organisme de formation et/ou de son auteur. Toute copie du support de cours est expressément interdite.

Toute copie de logiciels étant formellement interdite, le non respect de cette disposition entraîne l'exclusion immédiate du stagiaire et peut donner lieu à des poursuites pénales.

L'animateur peut cependant autoriser le stagiaire à copier les exercices réalisés, à condition qu'aucune partie du programme exécutable qui fait l'objet d'une licence d'utilisation ne soit incluse dans ces exercices.

Article 25 : Usage du matériel et de la documentation

Le stagiaire est tenu de conserver en bon état, d'une façon générale, tout le matériel et la documentation qui est mis à sa disposition pendant le stage.

Il ne doit pas utiliser ce matériel ou la documentation à d'autres fins que celles prévus pour le stage, et notamment à des fins personnelles, sans autorisation.

Lors de la fin du stage, le stagiaire est tenu de restituer tout matériel et document en sa possession appartenant à Learning Tree International. Cette disposition ne vise pas le support de cours que le stagiaire peut bien évidemment emporter.

4. SÉCURITÉ SOCIALE, CONGÉS MALADIE, ACCIDENT DU TRAVAIL

Article 26 : Sécurité sociale

Les stagiaires de stages agréés ou conventionnés sont affiliés à la Caisse Primaire d'Assurance Maladie de leur domicile principal (régime général ou 101) et ce, pendant toute la durée du stage. Si un stagiaire ne bénéficie pas de protection sociale, il est tenu de le signaler à la Direction.

Article 27 : Congés maladie et accident du travail

La procédure à suivre est la suivante :

Congés maladie :

Le stagiaire doit prévenir la Direction de Learning Tree International dès la première demi-journée d'absence.

Dans les 48 heures de l'arrêt, ou à son retour si celui-ci a lieu avant ce délai, le stagiaire doit fournir un certificat médical à Learning Tree International.

Sans cette pièce administrative importante pour son dossier, le stagiaire est considéré comme absent non excusé avec toutes les conséquences que cela implique. Cette communication à Learning Tree International est purement informative, la décision appartenant à l'entreprise d'origine du stagiaire.

Accident du travail ou de trajet :

Le stagiaire doit communiquer par écrit et simultanément à son entreprise pour action, et à Learning Tree International pour information, les circonstances de l'accident dans un délai de 48 heures maximum.

5. DISCIPLINE GÉNÉRALE

Article 28 : Discipline générale

Chaque stagiaire est tenu de respecter les instructions qui lui sont données par le responsable de stage.

La bonne marche de l'établissement passe notamment par l'acceptation d'une discipline élémentaire se traduisant dans les faits par l'obligation de respecter certains interdits :

- Procéder à des affichages dans des conditions non prévues par la loi ou non autorisées par la Direction de Learning Tree International ;
- Utiliser à des fins personnelles le téléphone, la télécopie, le minitel, les photocopieurs et la machine à affranchir, sans autorisation du responsable du stage et sans en acquitter le montant correspondant ;
- Organiser ou participer à des réunions dans Learning Tree International, dans des conditions non prévues par la loi ou non autorisées par la Direction du centre de formation;
- Introduire des objets ou marchandises destinés à être vendus ;
- Effectuer tout acte de nature à porter atteinte à la sécurité, à troubler le bon ordre, la discipline et de manquer de respect envers chacun pendant le déroulement du stage ou dans les autres locaux mis à la disposition des stagiaires ;
- Pénétrer ou séjourner dans les locaux de Learning Tree International en état d'ébriété ;
- Proférer des insultes ou menaces envers des membres du personnel ou envers d'autres stagiaires ;
- Se livrer à des actes répréhensibles vis à vis de la morale.

Article 29 : Mesures disciplinaires

Tout manquement par le stagiaire aux obligations résultant tant du présent règlement que des notes de services, pourra entraîner une sanction, soit un avertissement écrit, soit une exclusion, après mise en œuvre de la procédure suivante :

Le stagiaire à l'encontre duquel le directeur du centre de formation, ou son représentant, envisage de prendre une sanction, en dehors des observations verbales exprimées par l'animateur, sera convoqué pour un entretien. Lors de cet entretien, le stagiaire aura la possibilité de se faire assister par une personne de son choix, stagiaire ou salarié de Learning Tree International.

Le directeur du centre de formation ou son représentant indique le manquement constaté ainsi que le motif de la sanction envisagée et recueille les explications du stagiaire.

La sanction fait l'objet d'une décision écrite et motivée, notifiée au stagiaire, sous la forme d'une lettre qui lui est remise contre décharge, ou d'une lettre recommandée. Cette décision est immédiate et ne peut intervenir plus de quinze jours après l'entretien.

Le directeur du centre informe de la sanction prise :

- l'employeur lorsque le stagiaire est un salarié bénéficiant d'un stage dans le cadre de la formation des entreprises,
- l'employeur et l'organisme paritaire qui a pris à sa charge les dépenses de la formation lorsque le stagiaire est un salarié bénéficiant d'un stage dans le cadre d'un congé de formation.

6. ENTRÉE EN VIGUEUR DU REGLEMENT

Ce règlement entre en vigueur le 1er février 2007.

Fait à Clichy, le 1er février 2007

Overview

Learning Tree International
Training You Can Trust®

Course Objectives

Upon completion of this course, you will be able to

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

GUI = graphical user interface SQL = structured query language

Course Contents

	Introduction and Overview
Chapter 1	Python Overview
Chapter 2	Working With Numbers and Strings
Chapter 3	Collections
Chapter 4	Functions
Chapter 5	Object-Oriented Programming
Chapter 6	Modules
Chapter 7	Managing Files and Exceptions
Chapter 8	Accessing Relational Databases With Python
Chapter 9	Developing GUIs With Tkinter
Chapter 10	Web Application Development With Python
Chapter 11	Course Summary
	Next Steps

Introduction and Overview

Learning Tree International

We Bring Learning to Life™

F.6/512/F.5

About Your Course Instructor

- Background and education
- Current position
- Experience

About You and Your Fellow Participants

- **Your name**
- **Organization name**
- **Current position**
- **Background**
- **Course expectations**
  - Please complete your Pre-Course Learning Profile within your My Learning Tree® Account if have not already done so

Course Schedule

- **Each day, the course will follow this schedule:**
  - Start Class ______________________
  - Morning Break ______________________
  - Lunch ______________________
  - Resume Class ______________________
  - Afternoon Break(s) ______________________
  - End Time ______________________
  - Last-Day Course Exam ______________________

Course Logistics

- **Attendees via AnyWare™**
- **Student lounge**
- **Free Wi-Fi—login information available at reception**
- **Refreshments**
- **Restaurants**
- **Restrooms**
- **Security**
- **Emergency measures**
- **Other important items**

If we can help improve your experience, please let us know!

Maximize Your Course Experience: Questions

- **To maximize the effectiveness of your course, we encourage you to**
  - Interact frequently with your instructor and classmates
  - Request clarification and further explanation at any time before, during, and after your course
  - AnyWare™ participants: Use your microphone and AnyWare Dashboard to interact with your instructor and classmates as if you were in the physical classroom

Your questions will be of value to you and your fellow attendees

Maximize Your Course Experience: Protocols

- **Communication protocols:**
  - In-class participants:
    - Please put your mobile phone into “meeting” mode
    - Please avoid typing at the keyboard while the instructor is lecturing
  - AnyWare™ participants:
    - If you have any questions, please Chime In to notify your instructor—especially if you’ve sent your question using the Chat Pod
    - If you are using speakers, please keep the volume low, or mute your microphone when not speaking to avoid an echo

Need help?
If you experience any technical issues during your course, simply click the **Get Assistance** button on your AnyWare Dashboard, or call **1-877-653-8733**

Maximize Your Course Experience: Collaboration

- **Participating in a collaborative classroom provides many benefits:**
  - Share tips and tricks gained from your work experience
  - Help each other and discuss solutions during the exercises and activities
  - Put your minds together to develop innovative ideas
  - Complete your course with an increased depth of knowledge

Please feel free to collaborate with your classmates in-class and via AnyWare™ throughout the course

My Learning Tree®—Your Course Headquarters

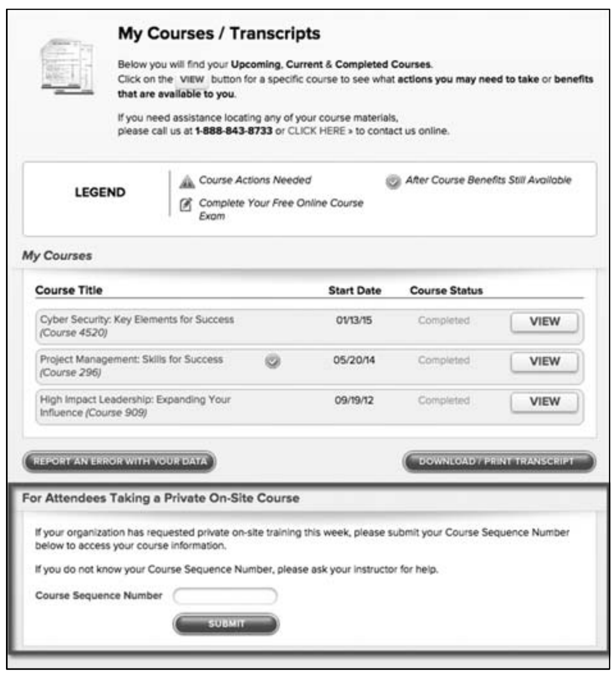
- **You will utilize your FREE personal My Learning Tree account throughout your course to**
  - Access your Hands-On Exercises
  - Enter the Recommend Your Colleagues Course Contest
  - Request your FREE Certificate of Achievement
  - Take your FREE Online Course Exam
- **PLUS, your My Learning Tree account can be used to access numerous FREE benefits after your course, such as:**
  - FREE After-Course Instructor Coaching for help and guidance back at work
  - FREE Computing Sandbox™ to practice your hands-on exercises—only for select courses
  - FREE Learning Resources and Supplemental Course Materials
  - Download and print your Course Notes, Hands-On Exercises, Transcript, and Certificates
  - GOLD CLUB Tuition Savings: SAVE $500 on select courses every week
  - *… and much more!*

Your instructor will cover many of the FREE benefits this week.

Log in to your account at: `LearningTree.com/MyLearningTree`

Fast Registration for On-Site Participants

- **For on-site participants ONLY**
  - To register and access your My Learning Tree® benefits:
    1. Log in or create your account at: `LearningTree.com/MyLearningTree`
    2. Once you've logged into your account, visit **My Courses / Transcripts**
    3. Your instructor will provide a Course Sequence Number for you to enter at the bottom of the page
  - Once you've submitted your sequence number, your course will be added to your account along with all the benefits of training with Learning Tree



My Courses / Transcripts

Below you will find your **Upcoming, Current & Completed Courses.**
Click on the VIEW button for a specific course to see what **actions you may need to take** or **benefits that are available to you.**

If you need assistance locating any of your course materials,
please call us at **1-888-843-8733** or CLICK HERE » to contact us online.

LEGEND: *Course Actions Needed* | *After Course Benefits Still Available* | *Complete Your Free Online Course Exam*

My Courses

Course Title	Start Date	Course Status	
Cyber Security: Key Elements for Success *(Course 4520)*	01/13/15	Completed	VIEW
Project Management: Skills for Success *(Course 296)*	05/20/14	Completed	VIEW
High Impact Leadership: Expanding Your Influence *(Course 909)*	09/19/12	Completed	VIEW

REPORT AN ERROR WITH YOUR DATA | DOWNLOAD / PRINT TRANSCRIPT

For Attendees Taking a Private On-Site Course

If your organization has requested private on-site training this week, please submit your Course Sequence Number below to access your course information.

If you do not know your Course Sequence Number, please ask your instructor for help.

Course Sequence Number

SUBMIT

Exam, Certifications, College Credits, and Tuition Reimbursement

- **Your free online course exam will take place on the last day of class**
  - The exam is composed of 40 multiple-choice questions
    - Third-party exam formats may differ *(e.g., ITIL Intermediate exams)*
  - If necessary, you can take the free online course exam at a later date through your My Learning Tree® account
- **Earn Learning Tree Specialist and Expert Certifications**
  - Passing the course exam may allow you to participate in Learning Tree's 2-Course Specialist and 3-Course Expert Certification Programs

- **Receive college credit units and tuition reimbursement**
  - Successfully passing your course exam may also allow you to receive college credits as certified by ACE CREDIT
    - PLUS, your certifications and college credits may qualify you for tuition reimbursement from your organization
      - Check with your HR or Training Department

ACE American Council on Education™ CREDIT® College Credit Recommendation Service

Learn more at: `LearningTree.com/Certifications`

Your Certificate of Achievement, Plus Specialist and Expert Diplomas

- **Complete this course to receive a Certificate of Achievement:**
  - Your Certificate of Achievement shows any education units you have earned
  - Your certificate will be automatically mailed to you within one week of course completion
  - You can also download your certificate from your My Learning Tree® Account on Monday following your course

- **Begin your path to Specialist and Expert Certifications:**
  - Successful completion of this course and exam may be applied toward earning Learning Tree 2-Course Specialist and/or 3-Course Expert Certifications

Track your certification progress through your My Learning Tree Account!

Log in at: `LearningTree.com/MyLearningTree`

Chapter 1

Python Overview

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

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

A Definition of Python

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

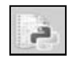
Python Philosophy

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

Chapter Contents

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

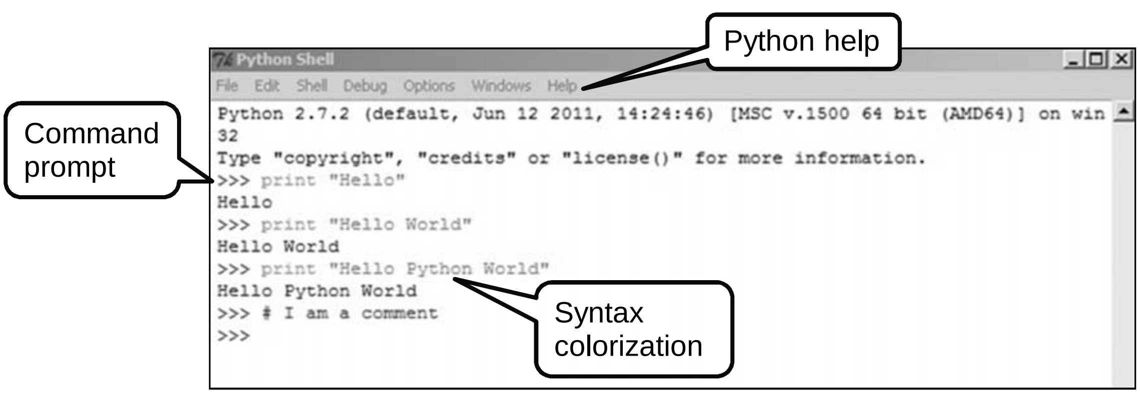
Accessing the Interpreter

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

Using IDLE

Do Now

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



Using IDLE

Do Now

2. **Enter the following statements:**

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

Press <Enter>

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

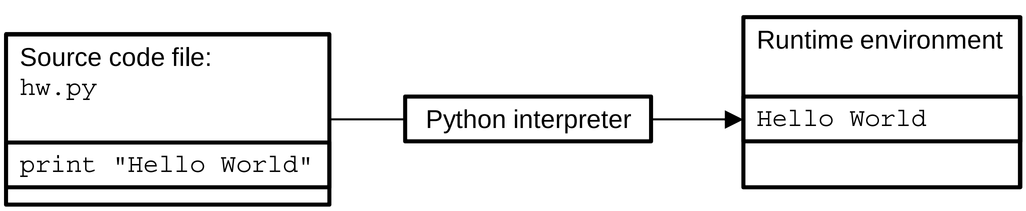
```
>>> print "Hello World"
Hello World
```

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

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

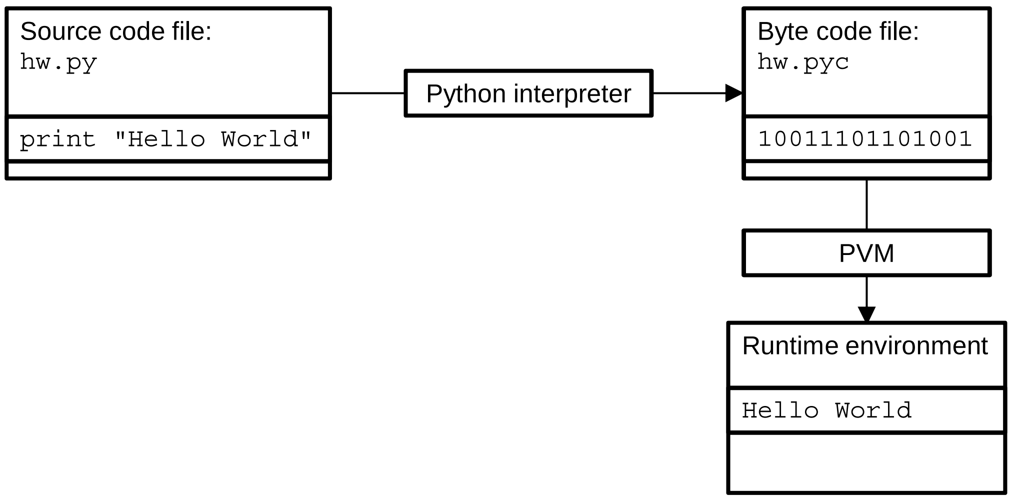

The Python Interpreter

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



The Python Interpreter

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



Using Eclipse

Do Now

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

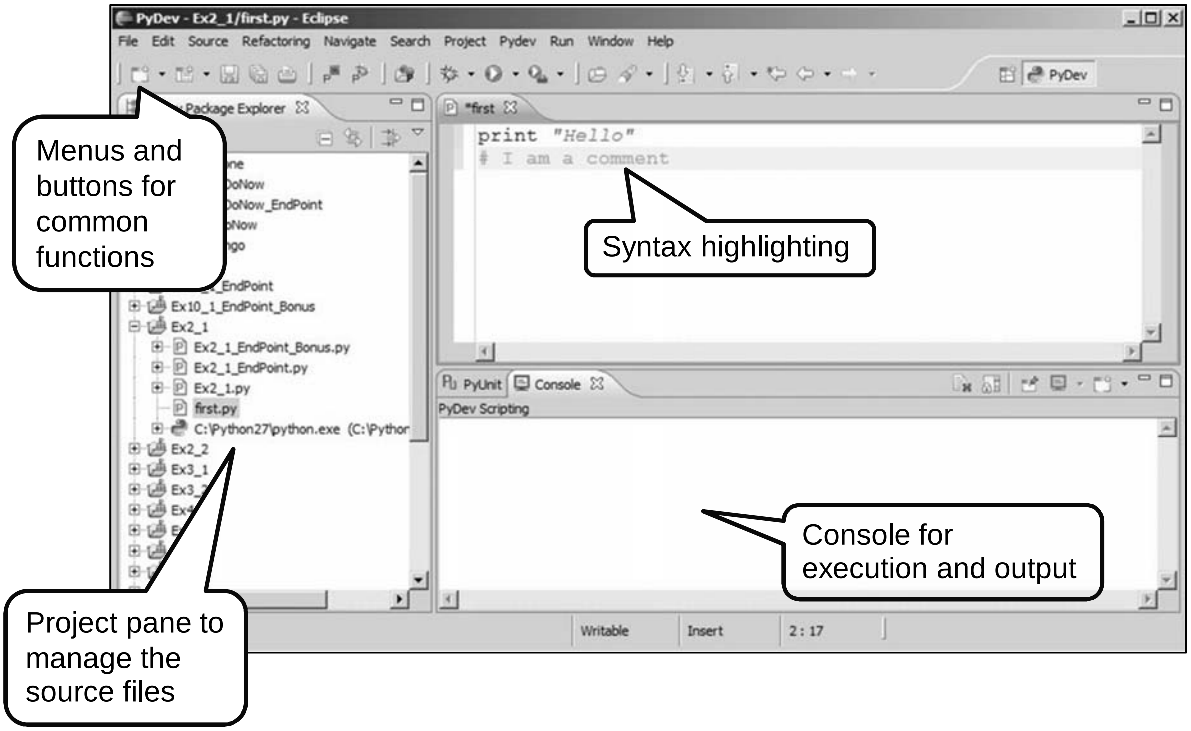
```
print "Hello"
# I am a comment
```

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

```
Hello
```

Using Eclipse

Do Now



Chapter Contents

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

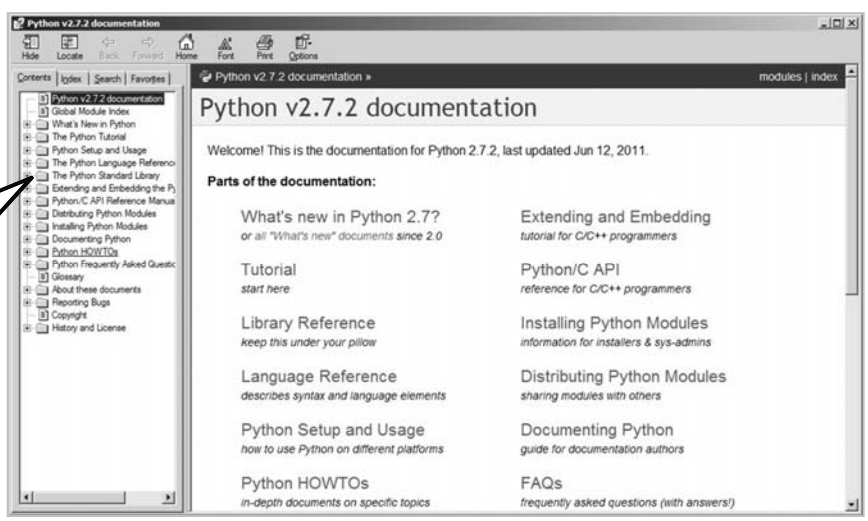
Documentation

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

PyDoc

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

From the Help menu of IDLE



HTML = hypertext markup language

Chapter Summary

You are now able to

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

Chapter 2

Working With Numbers and Strings

Learning Tree International

Training You Can Trust

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

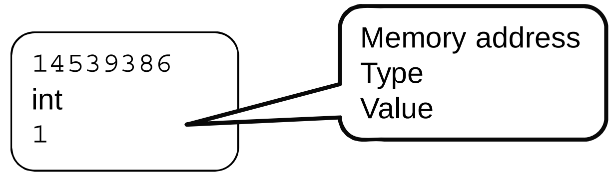
- ➡ **Objects and Variables**
- ➤ **Numeric Types and Operations**
- ➤ **String Types and Operations**
- ➤ **Conditionals**

Python Objects

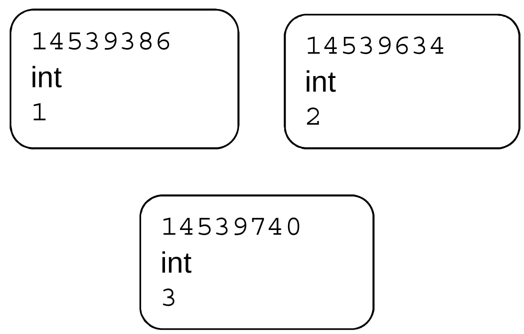
- **An object is an instance of a data value stored in a memory location**
  - Memory is allocated when the object is created
    - Built-in function `id()` shows memory address
  - Memory is reclaimed when the object is no longer referenced
    - *Garbage collection*
  - `1`, `2.5`, and `'Welcome'`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

Objects Illustrated

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




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



Variables

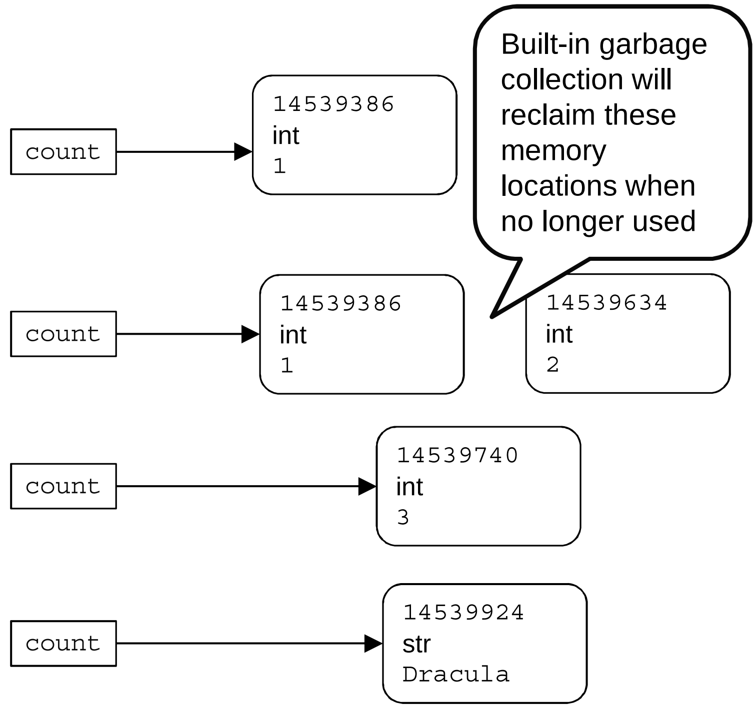
- **A variable is a named reference to an object**
  - Operations on variables use the object referenced
  - Operations are constrained by the type of the object
  - Variable instance is created when an object is assigned
    - An *identifier*
- **A variable is modified by assigning a different object**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

Variables Illustrated

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

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

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



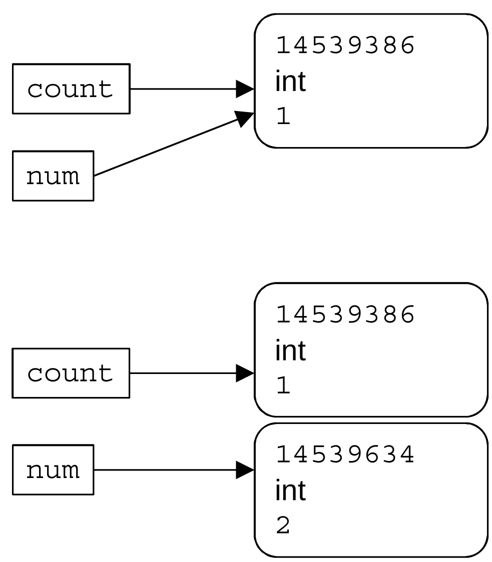
Shared Reference

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

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

Boolean result

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



Naming Rules

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

Chapter Contents

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

Python Built-In Types

- **Immutable types**
  - String literals
  - Arithmetic literals
    - Integer, floating point
- **Collection types**
  - Lists, dictionaries, and sets are mutable
  - Tuples are immutable

Numeric Objects

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

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

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

Numeric Operators

Precedence

`( )`	Raise precedence level
`a ** b`	Exponentiation

`~a`
`-a`
`+a`
	Bitwise NOT
	Negation
	Identity

`a * b`
`a / b`
`a // b`
`a % b`
	Multiplication
	True division
	Floor division
	Modulus

`a + b` `a - b`	Addition Subtraction
`a << b, a >> b`	Bit shift
`a & b`	Bitwise AND
`a ^ b`	Bitwise exclusive OR
`a | b`	Bitwise OR
`a < b, a <= b, a > b, a >= b` `a != b, a == b`	Relational Equality
`=, +=, -=, *=, /=, %=`	Assignment

Numeric Operators Example

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

Multiple precedence levels

Compound assignment

Comparisons yield Boolean results

Numeric Operation Type

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

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

All integer

Floor division, `//`, always rounds down

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

Result is floating point

Python 3 integer division always yields a floating point result

Arithmetic Typing Functions

- **The `float()` and `int()` functions return the argument in the specified type**
  - Argument may be the string representation of a numeric value
  - Argument may be an expression

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

String representation of a numeric value

Raises `ValueError` exception

Arithmetic Base Functions

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

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

Base-10 value

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

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


`print` Statement

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

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

`print` is a function in Python 3

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

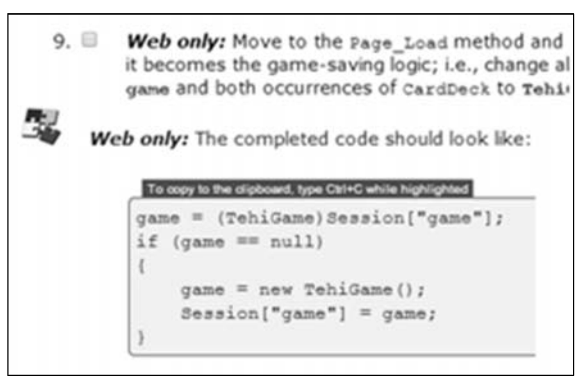
AdaptaLearn™ Enabled

- **Electronic, interactive exercise manual**
- **Offers an enhanced learning experience**
  - Some courses provide folded steps that adapt to your skill level
  - Code is easily copied from the manual
  - After class, the manual can be accessed remotely for continued reference and practice
- **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**

Using AdaptaLearn™

Do Now

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



Hands-On Exercise 2.1

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

The `math` Module

Reference

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

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

Functions from the module

Constant from the module

Chapter Contents

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

String Objects

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

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

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

String Slicing

- **A slice is a portion of a sequence**
  - Described by its offset
  - Slice boundary is a range specified in `[`*`start`*`:`*`end`*`]`

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

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

Bounded slice

Unbounded slices extend to an end

Entire sequence

String Slicing

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

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

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

Always the last reference from a sequence

Undefined slice yields an empty string

String Slicing

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

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

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

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

Every second element from `1` to `12`

String reversal

String is immutable

String Operations

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

```
>>> name = 'Guido'
>>> name + name + name
'GuidoGuidoGuido'
>>> name * 3
'GuidoGuidoGuido'
>>> name[0] * 3
'GGG'
>>> name[0] * 3 + name + name
'GGGGuidoGuido'
>>> str(-12.5)
'-12.5'
>>> str( 7 / 3.0 )
'2.33333333333'
```

Concatenation

Repetition

Type conversion

String Methods

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

New string returned

Boolean returned

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


String Methods

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

Offset

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

Use returned values for slicing

String Methods

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

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

New string returned

List returned

String returned

Delimiter in the returned string

String Formatting

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

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

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

Argument 2

Returned string

Argument 0

String Formatting

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

`n`	Width
`d`	Integer
`f`	Floating point with precision

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


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

Less than 9, pad with spaces

More than 9, no truncation

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

2 places after decimal

String Formatting

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

`n`	Width
`d`	Integer
`f`	Floating point with precision

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

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

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

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

The `string` Module

Reference

- **Standard library module providing string functions and constants**

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

Constants from the module

Function from the module

Special Strings

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

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

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

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

All strings in Python 3 are Unicode

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

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

Chapter Contents

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

Simple Comparisons

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

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

Larger Comparisons

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

All tests yield `True`

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

Second test is `False`

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

Compound Comparisons

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

Both must be `True`

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

Either yields `True`

Compound Statements

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

The `if` Statement

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

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

Indentation defines the block

Any number of optional `elifs` may follow

The single `else` is optional

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

Simple Testing

- **Empty and nonempty strings**

Yields a Boolean

Empty terminates block

Condition was `False`

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


Testing Alternatives Using `elif`

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

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

The `pass` Statement

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

Placeholder between `if` and `else`

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


Hands-On Exercise 2.2

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

Chapter Summary

You are now able to

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

Chapter 3

Collections

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

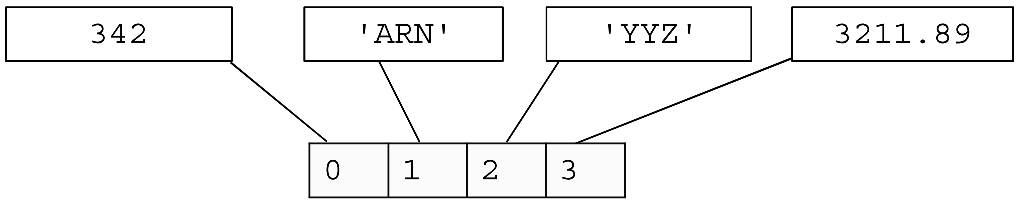
- **Lists, Dictionaries, and Tuples**
- **`for` Loops and Iterators**
- **`while` Loops**

Collections

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

List

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



List Indexing

- **Lists can be assigned**
  - A sequence of values within `[ ]`
  - Or simply `[]` to represent an empty list
- **Contents are accessed with syntax similar to strings**
  - Numeric offset range described in `[ ]`

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

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

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

List assignment

Access a single element

Use list elements like any other object

Lists are mutable

List Slicing

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

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

Result is a list

List Operators

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

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

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

List Operations

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

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

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

Assignment creates shared reference

List is copied

List Methods

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

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

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

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

Cannot extend the list by assigning a new element

List Methods Example

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

List is mutable

Remove by value

Tuple

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

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

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

Immutable

Tuple

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

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

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

____

Tuple Operations

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

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

Complex Collections

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

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

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

Inner lists

Right bracket] terminates the outer list

What would `tuple(twocodes)` return?

Sequence Unpacking

- **Multiple values from a collection are assigned**
  - Collection slice is allowed
- **Correct number of variables needed to hold all unpacked values**

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

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

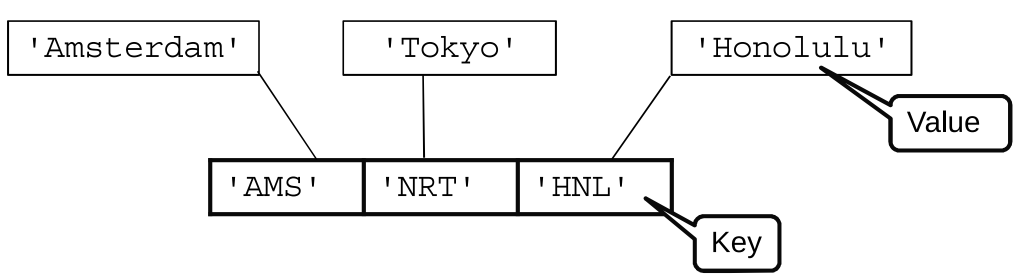
```
>>> depart, *layovers, arrive = airports
```

Hands-On Exercise 3.1

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

Dictionary

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



Dictionary Operations

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

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

Values are mutable

New entry

Dictionary Methods and Functions

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

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

Lists

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

Dictionary Methods Example

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

Duplicate keys

Returned `None`

`KeyError` exception is raised

Creating a Dictionary

- ***dict*.items()—Returns a list of key–value pairs as tuples**
- **A dictionary can be created from a sequence of key–value pairs**
  - `dict(`*arg*`)` returns a dictionary
  - *arg* is a sequence containing the key–value pairs
- ***dict* = {}—Creates an empty dictionary**

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

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

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

List of tuples

Tuple of tuples

Duplicate the dictionary

`zip()` Function

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

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

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

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

List of tuples

Convert to a dictionary

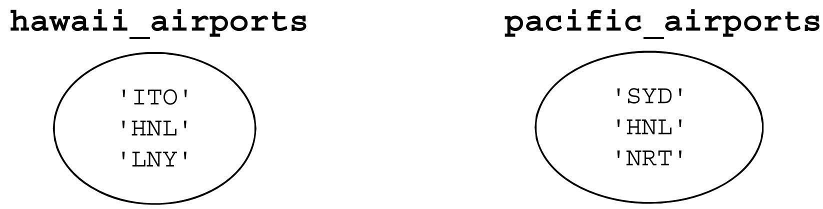
Sets

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

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

Sets

- **Support arithmetic-style operators**



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

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


Membership Quiz

- **Given the following:**

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

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

____

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

____

How could these results be assigned to a list?

____

Collection Membership Testing: `in`

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

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


Chapter Contents

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

Flow Control With Loops

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

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


The `for` Loop

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

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

Loop Through a Sequence

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

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

Final value processed in the loop

Nested Looping

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

Loop Through a Dictionary

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

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

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

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

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

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

Both print
NRT
YYZ

Tokyo
Toronto

NRT Tokyo
YYZ Toronto

Membership Quiz With a Loop

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

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

Optional Flow Control Within Loops

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

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

Optional test to terminate the loop

Optional test to return to the top of the loop

Optional block if `break` was not executed

Using `break`, `continue`, and `else` in a Loop

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

Terminate with `'HNL'`

Skip with `'HNL'`

Executed if there was no `break`

The `range` Function

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

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

Negative step used for decreasing

The `xrange` Function

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

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

`range()` is `xrange()`

Iterable Object

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

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

Create an iterable object from the list

Method to deliver each element

Exception raised when iteration is complete

Iterating Through a Sequence

- **The `for` loop uses the iteration protocol internally**

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

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

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

List Comprehension

- **An *operation* is applied to each element with a `for` loop**
  - Result is a list
- **Syntax:** `[`*`operation`* `for` *`var`* `in` *`iterable`*`]`

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

for loop applies the operation

Result is a list

Nested

List Comprehension With Conditional

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

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

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

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

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

Keys from `codes` that are also keys in `caps`

Generator Comprehension

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

Result is an iterable object

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


Additional Comprehensions Backported from Python 3

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

  `{operation for var in set if condition}`

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

```
{'HNL'}
```

Result is a set

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

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

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

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

Result is a dictionary

Chapter Contents

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

The `while` Loop

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

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

Optional test to terminate the loop

Optional test to skip to next iteration

Optional block if `break` was not executed

`while` Loop Example

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

Condition is evaluated before each pass through the loop body

Hands-On Exercise 3.2

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

Chapter Summary

You are now able to

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

Chapter 4

Functions

Learning Tree International
Training You Can Trust

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

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

Function Overview

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

Simple Function Example

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

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

Function is defined and named

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

Function is called by its name

Function is an object at a memory location

Passing Data Into a Function

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

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

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

Arguments

Keyword Parameters

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

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


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

Specify default parameter values

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

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

Keyword arguments may be passed in any order

Default is applied for `arrive` within the function

Variable-Length Parameter Lists

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

Parameter list

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

Argument list

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

Variable-Length Parameter List Example

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

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

Positional arguments are in a tuple

Keyword arguments are in a dictionary

Variable-Length Argument Lists

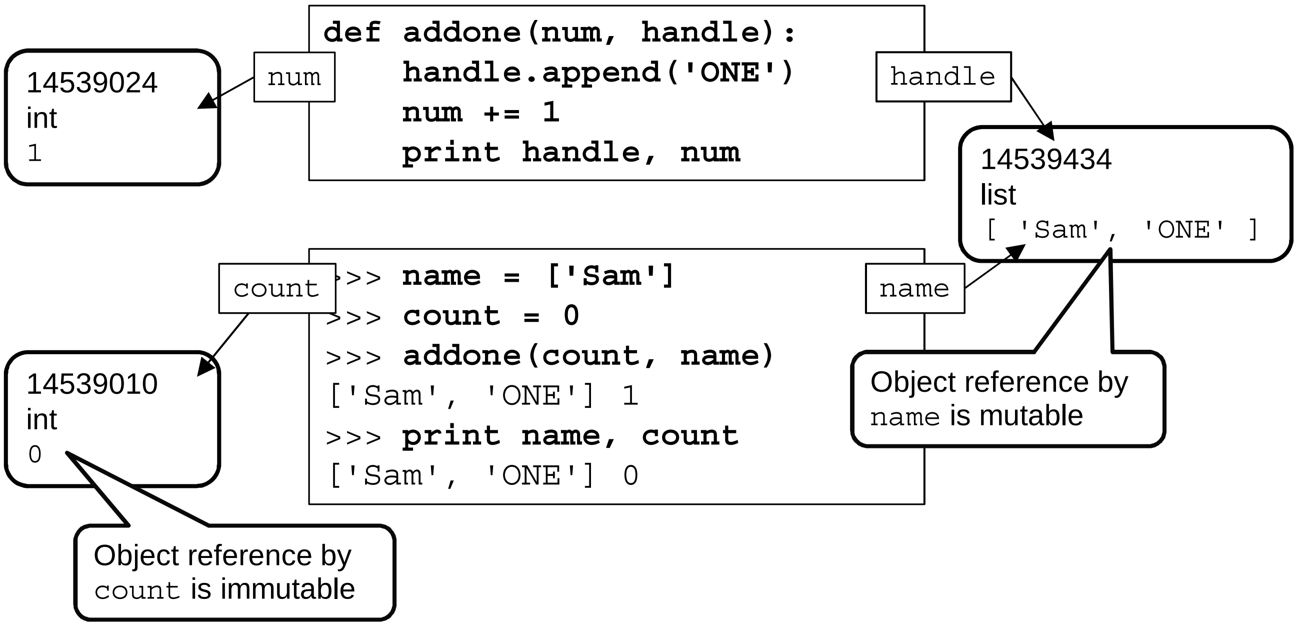
- **Functions may be called with a sequence or dictionary argument**
- **Argument name preceded by `*` will pass a collection as a sequence of positional parameters**
- **Argument name preceded by `**` will pass a dictionary as keyword parameters**

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

`employee2` without `**` is a single positional

Mutable and Immutable Arguments

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



Enclosed Functions

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

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

Enclosed functions

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

Scope

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

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

print var
fun1()
```

Enclosing function

Enclosed function

```
global
enclosing
local
```


LEGB Rule

- **Describes attribute resolution order**

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

Covered in later sections

Arguments and Scope

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

`increment()` scope

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

Global scope

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

```
function number is 6

global number is 5
```


`global` Statement

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

`area()` scope

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

Global scope

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

```
3.14
4.14

global pi is 4.14
```

14539132
float
`3.14`
`4.14`

Global object was modified by reference in the function

`return` Statement

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

Function `return` Example

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

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

Return a reference to a single object

Return a sequence

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

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

Assigned the returned value

Unpack a sequence

Functions and Polymorphism

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

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

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


Functions as Arguments

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

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

def print_italian():
    print 'Buon Giorno!'

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

Reference to the function object used as an argument

Call the function

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

Hands-On Exercise 4.1

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

Chapter Contents

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

The Lambda Expression

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

Reference to the `lambda` function object

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

Argument to the `lambda` function

The Lambda Expression

- **May be used where statements are not syntactically allowed**

Each anonymous function is a dictionary value

Dictionary keys

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

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

Argument to the `lambda` function

Hiding Function Calls in Lambda Expressions

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

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

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

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

Reference to the `lambda` object used as an argument

Executes the `lambda`

Function call with argument is the `lambda` body

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


Chapter Contents

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

Generator Function

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

Generator Function Example

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

`yield` defines the function as a generator function

`days` is the iterable object

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

First execution from top of function to `yield`

Subsequent executions proceed to next `yield`

`next()` method retrieves each item

Stopping Iteration

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

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

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

Calls `next` and handles `StopIteration` internally

Hands-On Exercise 4.2

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

Chapter Summary

You are now able to

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

Chapter 5

Object-Oriented Programming

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

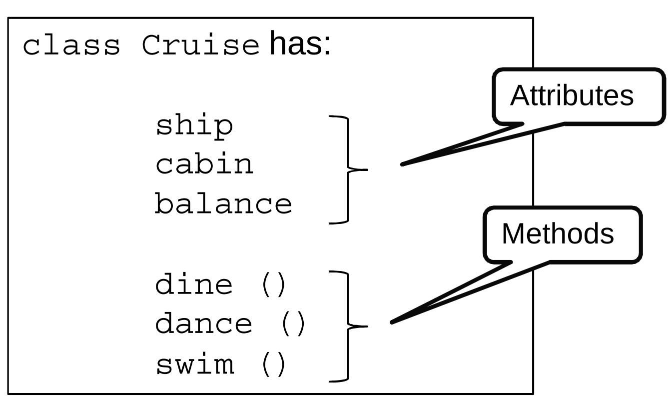
- **Define a class**
- **Create subclasses through inheritance**
- **Attach methods to classes**
- **Overload operators**

Chapter Contents

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

Class

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



The `class` Statement

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

A docstring is customary to describe the class

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

A class is an object

The `__init__()` Method

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

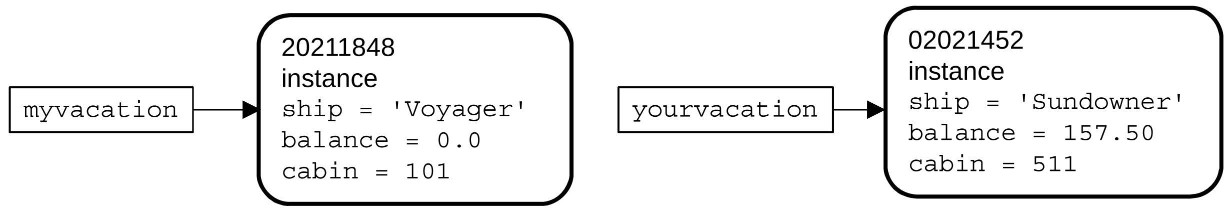
```
class Cruise(object):
    ''' This class describes a cruise.'''
    def __init__(self, ship=None, balance=0.0, cabin=0):
        self.ship = ship
        self.balance = balance
        self.cabin = cabin
```

Keyword parameters with defaults

The `self` Argument

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

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

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



`__init__()` Parameter Styles

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

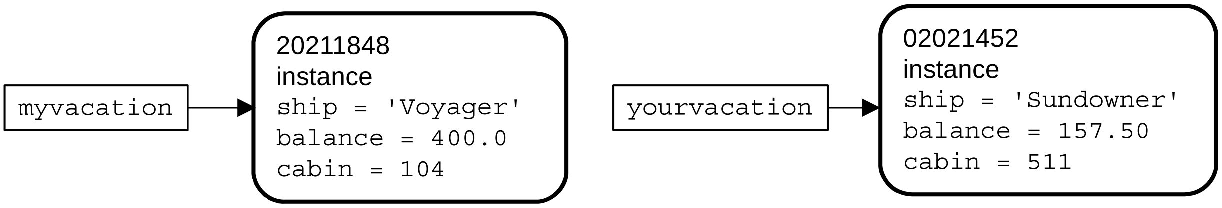
```
class Cruise(object):
    ''' This class describes a cruise.'''
    def __init__(self, shipname, bal, room):
        self.ship = shipname
        self.balance = bal
        self.cabin = room

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

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

Attribute names

Modifying Instance Attributes

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

```
myvacation.balance = 400.0

myvacation.cabin = 104
```



Methods

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

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

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

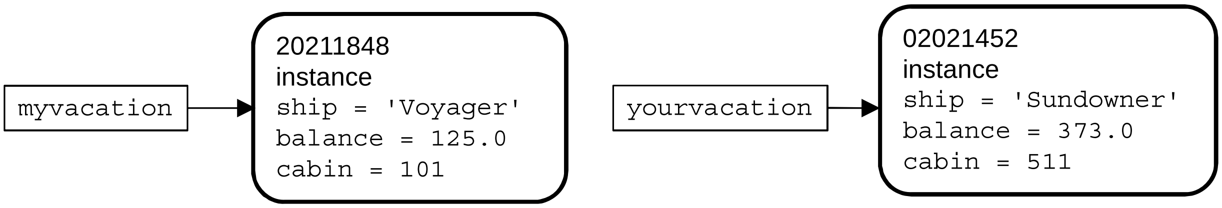
Attribute names

Modifies the instance attribute

Methods Illustrated

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

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

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



Class Variables

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

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

Class variable

Check instance attribute for membership in a class variable

Hands-On Exercise 5.1

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

Chapter Contents

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

Class Hierarchy

- **Describe the *Is A* relationship**
  - Derived/subclass is an extension of the parent/base class
    - Subclass performs class/type specific operations
- **Syntax:**

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

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

Both inherit from `Trip` class

Class Inheritance

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

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

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

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

Inheritance Hierarchy

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

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

Method within `Cruise`

Method inherited from `Trip`

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

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

Method within `Flight`

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


Subclass Instance Initialization

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

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

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

Assign `ship` attribute only

Assign all attributes in `Flight` constructor

Subclass Extension

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

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

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

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

From `Trip` class

From `Cruise` class

From `Flight` class

```
Friday
Sea Breeze
CRJ Monday
```


The `super()` Function

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

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

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

Calls the constructor from its parent class

Subclass Instance Initialization Using `super()`

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

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

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

Assigned in parent class

Assigned in the subclass

Subclass Attributes

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

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

```
Friday Sea Breeze
```

Inherited from `Trip`

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

```
Monday CRJ
```

Hide same named attribute in `Trip`

Overriding Methods

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

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

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

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

`Trip` objects call this method

`Cruise` objects call this method

Overriding Methods

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

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

```
Trip leaves on Sunday
```

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

```
Cruise Monday to Monday
```

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

```
Cruise Tuesday to Tuesday
```

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

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

Extending Methods

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

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

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

Trailing comma suppresses `print`'s newline

Call method in parent class

Handle class specific task

Extending Methods

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

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

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

Call method in subclass

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

Multiple Inheritance

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

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

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

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

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

print interview.zip
10002

print interview.day
Monday

print interview.name
Bob
```


Overloaded Operators

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

Overloaded Operators Example

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

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

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


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

Internal `__add__()` intercepts + operator

Returns a list

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


Hands-On Exercise 5.2

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

Chapter Contents

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

Decorator Function

Reference

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

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

@logit
def print_status():
    print 'Processing'

print_status()
```

```
Start logging
Processing
Stop logging
```

Equivalent to:
`print_status = logit(print_status)`

Class Methods

Reference

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

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

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

Decorator identifying the class method

```
Cruise 0.5
Sunsetsail 0.5


Cruise 0.1
Sunsetsail 0.25
```


Static Methods

Reference

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

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

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

Decorator identifying the static method

```
Cruise 0.5
Sunsetsail 0.5


Cruise 0.25
Sunsetsail 0.25
```

Abstract Class

Reference

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

`abc` Module

Reference

- **Provides tools to implement <u>A</u>bstract <u>B</u>ase <u>C</u>lasses (ABCs)**
  - `ABCMeta`—metaclass for ABCs
    - Metaclasses set up classes
  - `abstractmethod`—decorator function that requires abstract methods are overridden

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

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

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

Abstract method

Subclass must provide this method

```
Anchors aweigh
```

Chapter Summary

You are now able to

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

Chapter 6

Modules

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

➡ **Module Overview**

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

Module

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

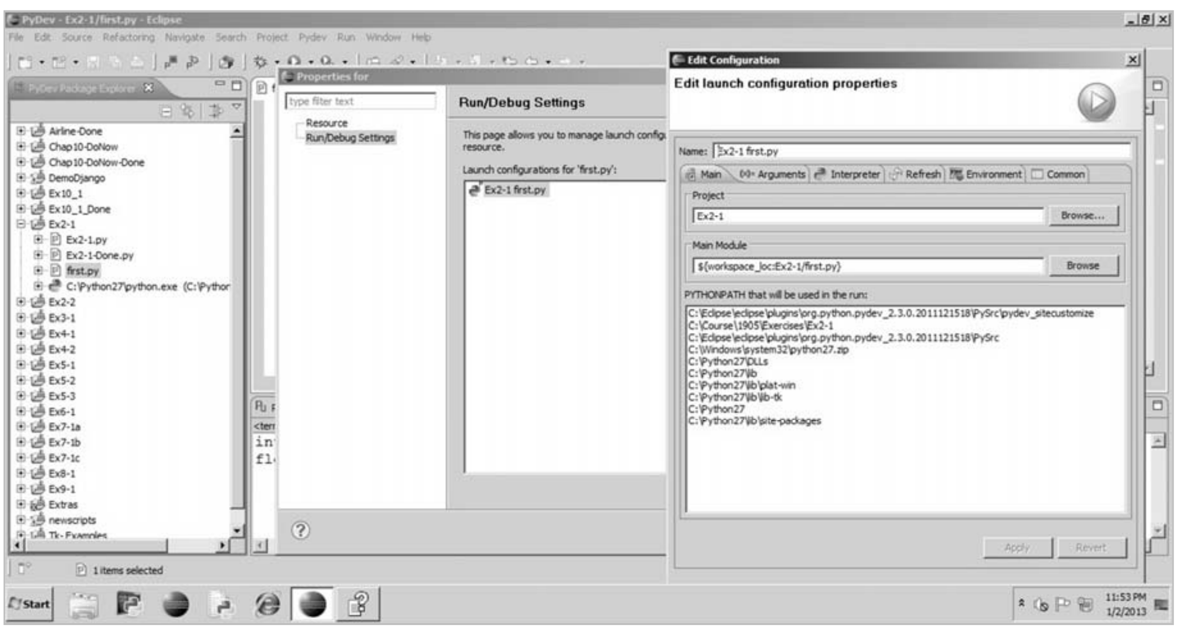
Module Files

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

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

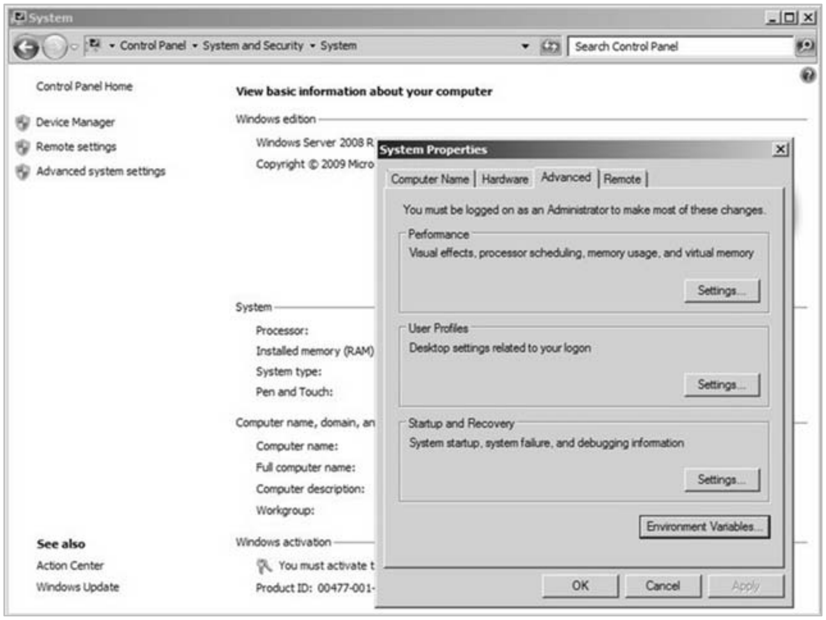
`PYTHONPATH` in Eclipse

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



`PYTHONPATH` in Windows Server 2008

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



Chapter Contents

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

The `import` Statement

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

Module Execution

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

mod1.py

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

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

Statements execute

Attribute

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

Module object created

Module Attributes

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

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


Multiple imports

- **Qualified attributes reference the proper module**

mod1.py

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

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

mod2.py

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

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

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

Qualified

Chained `imports`

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

mod1.py (C)

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

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

mod2.py (B)

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

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

(A)

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

Two levels of qualification

The `import as` Statement

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

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

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

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

Examining Namespace

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

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

Examine the current module

Name of the Python program in execution

Examine the math module

Name of the math module

Testing the `__name__` Attribute

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

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

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

Chapter Contents

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

`from` Statement

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

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

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

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

Single attribute

Unqualified name

`from` Statement

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

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

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

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

All attributes

Unqualified

Namespace Corruption

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

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

`name` in current namespace changed

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

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

Hands-On Exercise 6.1

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

Packages

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

```
python setup.py install
```

Downloading Packages

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

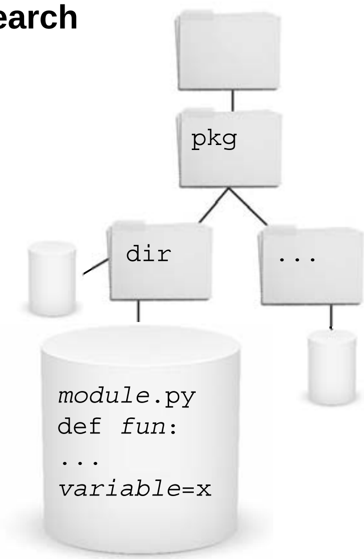
```
pip install packagename
```


Package Directory Structure

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

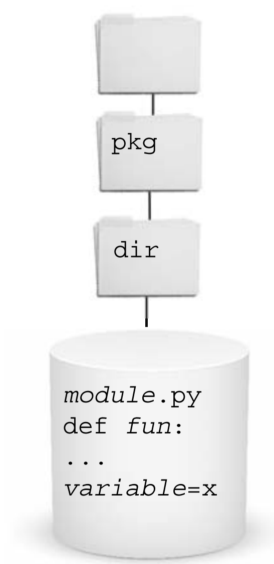
Package `import`

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



Accessing Package Modules

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



Chapter Contents

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

Standard Library

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

The `sys` Module

Reference

- Contains functions and variables that are used by Python itself

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

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


Command-Line Arguments

Reference

- Are passed into the program when it is started

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

argtest.py

```
import sys

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

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

The `os` and `subprocess` Modules

Reference

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

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

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


The `os` Module and Running Commands

Reference

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

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

Output shortened

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

Reference

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

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

  - Query the file system

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

The `glob` Module and File Names

Reference

- **Provides pattern matching on file names**

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

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

File System Management Example

Reference

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


Regular Expression Special Characters

Reference

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

Regular Expression Escape Sequences

Reference

- **Alternate method to describe select text patterns**

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

Using Backslash \

Reference

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

The `re` Module

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

String Matching Example

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

String slice

Extracting Substrings

Reference

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

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


Using Regular Expressions

Do Now

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

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

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

Using Regular Expressions: A Solution

Do Now

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

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

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

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


Chapter Summary

You are now able to

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

Chapter 7

Managing Files and Exceptions

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

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

I/O = input/output

Chapter Contents

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

Keyboard Input

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

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

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

Prompt

Line that caused the exception

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

Exceptions

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

The `try` Statement

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

```
try:
    statements ...

except exceptionTypeA:
    statements ...

except exceptionTypeB:
    statements ...

except:
    statements ...

else:
    statements ...

finally:
    statements ...
```

Block monitored for an exception

Block to handle exceptions of type *exceptionTypeA*

Block to handle exceptions of type *exceptionTypeB*

Block to handle any other unnamed exception

Block to handle if no exception was raised

Block to execute whether an exception was raised or not

Handling a Single Exception

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

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


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

1 Call function within `try`

2 Raises `ValueError` exception

3 Branch to `except` for that exception

Handling Multiple Exception Types

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

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

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

For any unnamed exception type

The `else` and `finally` Clauses

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

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

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

Block always executes

No exception raised

Exception Instances

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

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

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

The `raise` Statement

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

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

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

`raise` the exception

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


Hands-On Exercise 7.1

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

Chapter Contents

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

Files

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

The `open` and `close` Statements

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

Optional

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

Opening Files and Exceptions

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

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

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

Reading a Text File

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

simple.txt

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

\n

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

- **`read()`: Returns the entire file contents as a single string**
- **`readline()`: Returns the next line from the file**
  - Includes the `'\n'` line delimiter
  - `rstrip()` string method can remove the `'\n'`
- **`readlines()`: Returns the entire file contents as a list of strings, including the `'\n'`**
- **`IOError` exception may be raised**

Reading a Text File Example

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

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

Writing a Text File

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

Data Handling Exceptions

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

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

`IOError` exception raised writing to the file

File must be closed whether or not exception occurs

Using `with` to Open and Close Files

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

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

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

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

Using Loops and Iterators for File Access

- **The file object is iterable**

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

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

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

The Standard Streams

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

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

Reading and Writing to Standard Streams

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

Redirecting Streams to Files

- **Assigns a disk file for use as a standard stream**
  - To automate testing user input
  - To capture text in a log file

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

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

Save the originals

Custom messages appended to the log file

Restore the originals

Chapter Contents

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

The `pickle` Module

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

Reading and Writing With `pickle`

```
import pickle

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

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

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

Create a dictionary of `Airport` objects

Reading and Writing With `pickle`

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

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

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

Serializes and stores the dictionary

Re-creates the dictionary from the file

```
Honolulu
```


The `shelve` Module

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

Reading and Writing With `shelve`

```
import shelve

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

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

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

Read and write access is allowed

Assign to the `shelve` object

Reading and Writing With `shelve`

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

Writes cache back into shelve file

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

Hands-On Exercise 7.2

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

Chapter Summary

You are now able to

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

Chapter 8

Accessing Relational Databases With Python

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

➡ **Relational Databases**

➤ **Using SQL**

Relational Databases

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

MySQL

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

API = application programming interface

Chapter Contents

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

Steps to Accessing the Database

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

Step 1: Establish a Connection

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

Hostname running the MySQL database

Database user name

```
import MySQLdb

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

Database password

Database name

Step 2: Create a Cursor for the Data Interchange

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

```
curs = connection.cursor()

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

Cursor name

Test if any rows are affected

Calls the database for the SQL execution

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

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

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

connection.close()
```

The columns to be retrieved; `*` indicates all

Tables to be searched

SQL selection criteria

Terminates the connection

Constructing a `SELECT` String

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

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

curs.execute(query)
```


Passing Arguments to SQL Statements

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

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

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

Placeholder for argument

Extracting Data From the Cursor

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

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

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

Inserting a Row

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

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

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

connection.commit()
```

Tuples

Assigned left to right

Updating Data

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

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

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

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

connection.commit()
```


Deleting Data

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

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

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

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

connection.commit()
```

Hands-On Exercise 8.1

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

Chapter Summary

You are now able to

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

Chapter 9

Developing GUIs With Tkinter

Learning Tree International
Training You Can Trust

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

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

Tkinter

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

Python 3 module name is `tkinter`

Tkinter Portability

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

Chapter Contents

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

Displaying Widgets

Do Now

A. Start Eclipse if needed

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

C. Run the program

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

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

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

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

Bring in the module classes and functions

2

3

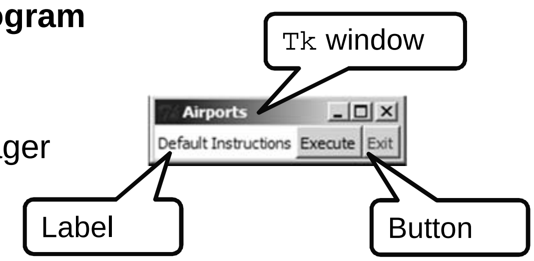
Create widget

Display widget

Displaying Widgets

Do Now

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




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

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

1

4

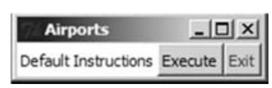
Start display loop

Widgets

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

`Tk` Class Widget

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



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

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

mainloop()
```

Start display loop

`Label` and `Button` Widgets

- **Display text strings or images**
  - `text` attribute references a string
  - `fg` and `bg` colors
- **Are part of a widget hierarchy**
  - First argument specifies the parent widget

`Label` and `Button`s

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

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

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

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

Parent widget is `Tk`

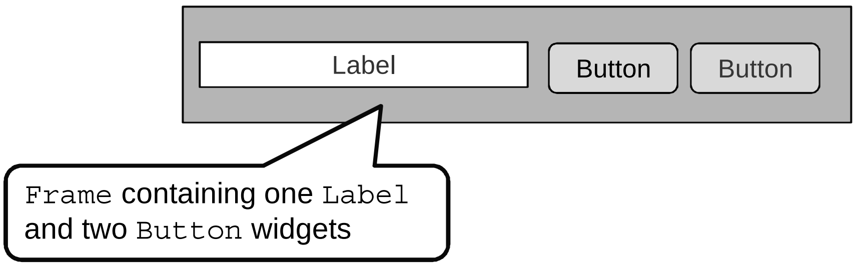
`pack()` Geometry Manager

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

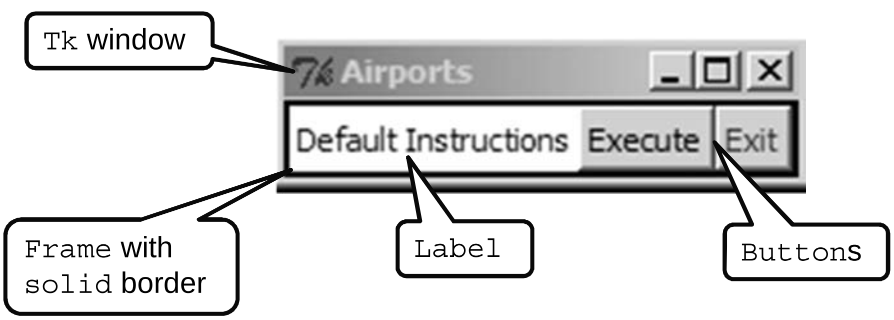
```
infolabel.pack(side='left', expand=1, fill='both')

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

`Frame` Widgets

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



`Frame` Widgets



labels_buttons_frame.py

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

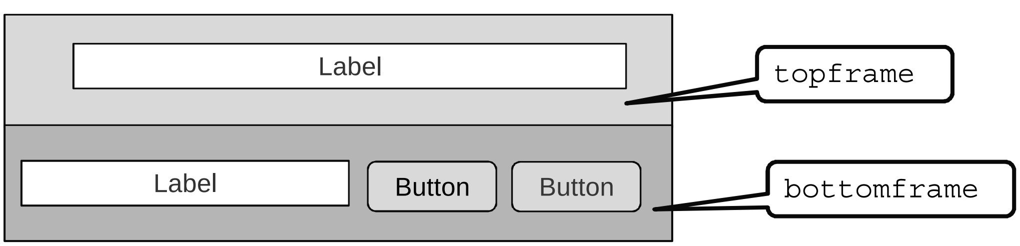
`Frame` Widgets

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

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

Create instance

Frames Within Frames

- **Frame widgets may contain additional frames**
  - Control widget layouts



label_buttons_frame_grid.py

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

Frames Within Frames

topframe is raised

bottomframe is sunken

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

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

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


The `grid` Method

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

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

Location within frame

Width

Chapter Contents

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

A Callback Function

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

label_buttons_frame_grid_callback.py

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

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

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

Method

Method

Terminates frame

Reconfiguring a Widget

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

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

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

Set `text` attribute in `output` `Label`

New `text` attribute value

`ScrolledText` Module

➤ **Provides a `ScrolledText` class**
- Implements a `Frame` containing a `Text` widget and vertical scrollbar
- Easier than creating them directly

➤ **Provides methods**
- `insert()` adds text within the widget
- `delete()` removes text from the widget

`label_buttons_frame_grid_callback_scrolltext.py`

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

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

...
```

`ScrolledText` Module

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

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

Chapter Contents

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

`Entry` Widgets

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

`Entry` Example

Reference input string

label_buttons_frame_grid_scrolltext_entry.py

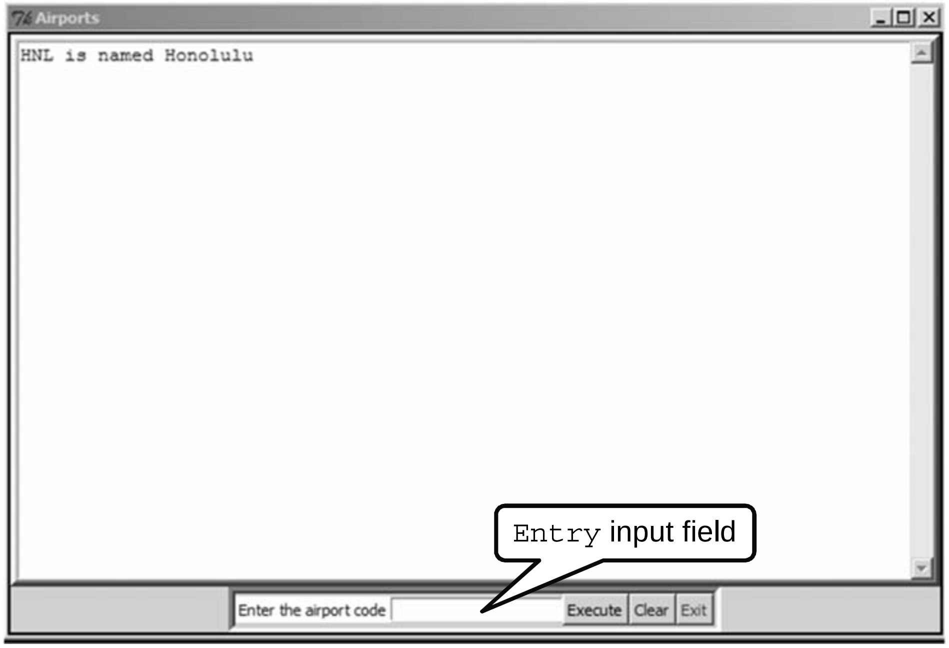
```
self.apt = StringVar()
self.input = Entry(self.bottomframe, textvariable=self.apt)
self.input.bind(sequence='<Return>', func=self.showinfo)

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

Retrieve input string

Line wraps around

Reset input field to empty string

Entry Example



Radiobutton Widgets

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

Radiobutton Selection

label_buttons_frame_grid_scrolltext_radio.py

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

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

No button is preselected

Variable shared by all `Radiobutton`s

Assigns `'NRT'` to `self.apt`

Chapter Contents

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

Menu Widgets

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

Menu Creation Steps

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

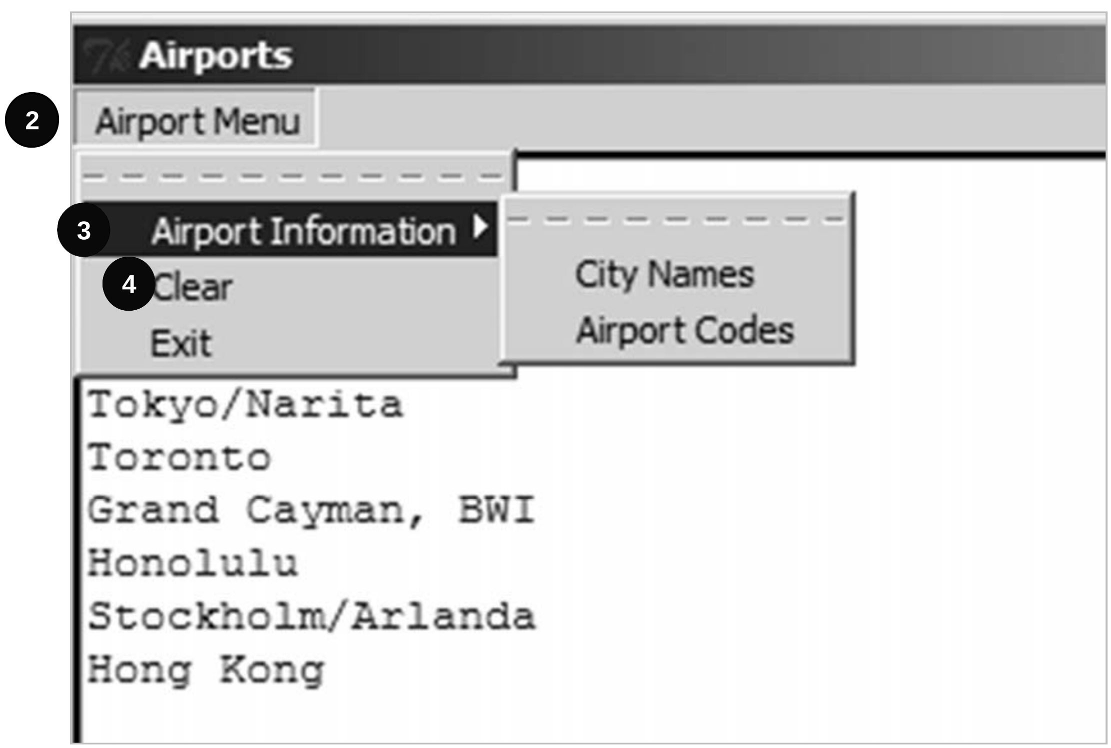
label_buttons_frame_grid_scrolltext_menu.py

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

Starting a Menu

```
def setup_gui(self):
    self.menubar = Menu(self.base)                          1
    self.base.configure(menu=self.menubar)

    self.airportmenu = Menu(self.menubar)                   2
    self.menubar.add_cascade(label='Airport Menu',
                             menu=self.airportmenu)         3
    self.airportinfo = Menu(self.airportmenu)
    self.airportmenu.add_cascade(label='Airport Information',
                                 menu=self.airportinfo)

    self.airportmenu.add_command(label='Clear',             4
                                 command=self.clearinfo)
    self.airportmenu.add_command(label='Exit',
                                 command=self.base.quit)
    self.airportinfo.add_command(label='City Names',
                                 command=self.show_values)
    self.airportinfo.add_command(label='Airport Codes',
                                 command=self.show_keys)
```


Menu Example



Hands-On Exercise 9.1

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

Chapter Summary

You are now able to

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

Chapter 10

Web Application Development With Python

Learning Tree International

Training You Can Trust®

Chapter Objectives

After completing this chapter, you will be able to

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

Chapter Contents

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

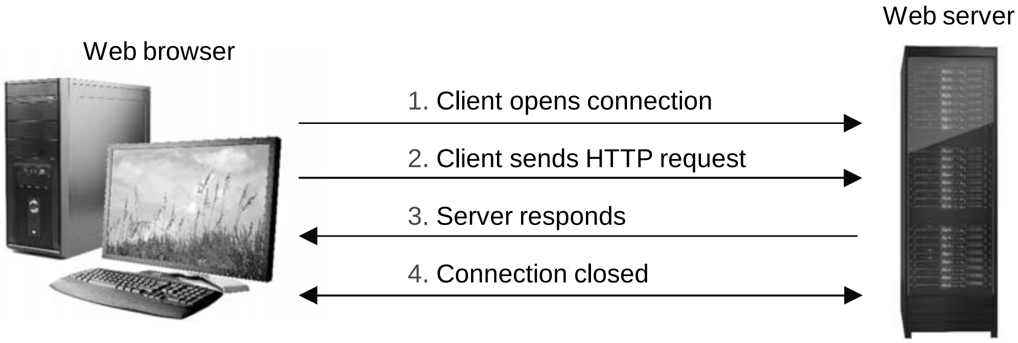
What Is a Web Application?

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

HTTP = Hypertext Transfer Protocol

Hypertext Transfer Protocol

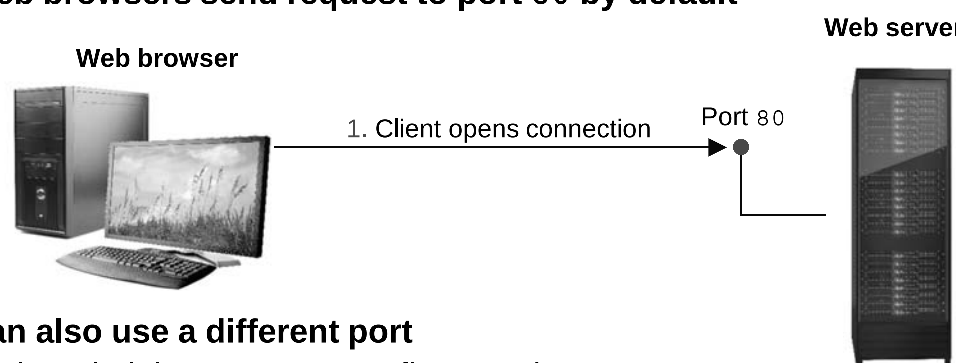
- **HTTP is the protocol for communicating on the web**
  - Stateless, TCP/IP-based protocol
  - HTTP 1.1 defined in RFC 2616, `www.faqs.org/rfcs`
- **HTTP conversation initiated when user enters URL in web browser**
  - For example, `www.learningtree.com/whats_hot.html`



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

Browser and Server Interaction Step 1: Client Opens Connection

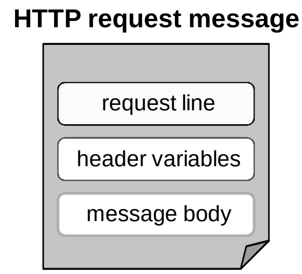
- **Client opens connection to server: `www.learningtree.com`**
  - Opens TCP/IP socket connection on a port
- **Web browsers send request to port `80` by default**

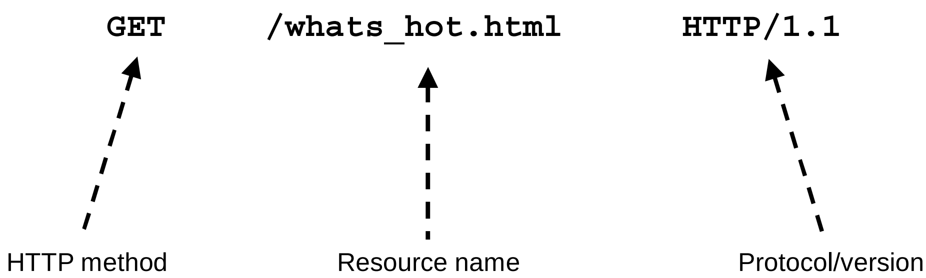


- **Can also use a different port**
  - The administrator must configure web server
  - Clients must use `http://`*`host_name`*`:`***`port`***
    - Example: `http://localhost:8000`

Browser and Server Interaction Step 2: Client Sends HTTP Request

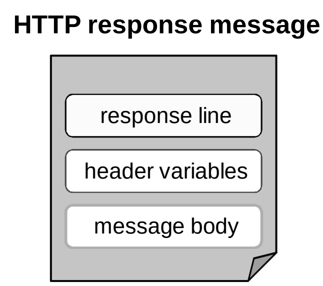
- **Web browser issues HTTP request**
- **An HTTP request message is composed of**
  - *Request line*: describes HTTP command
  - *Header variables*: browser information
  - *Message body*: contents of message
- ***Request line* is composed of method, resource name, protocol**





Browser and Server Interaction Step 3: Server Responds

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




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

Response line

Header variables

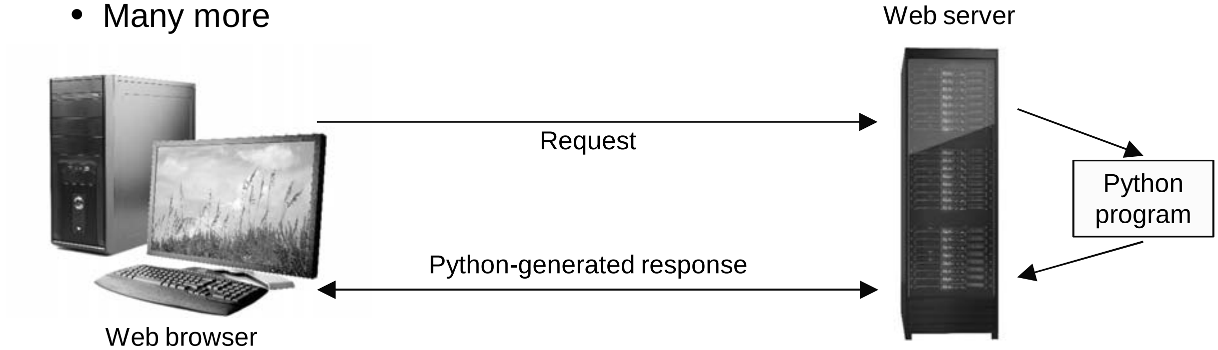
Message body

Chapter Contents

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

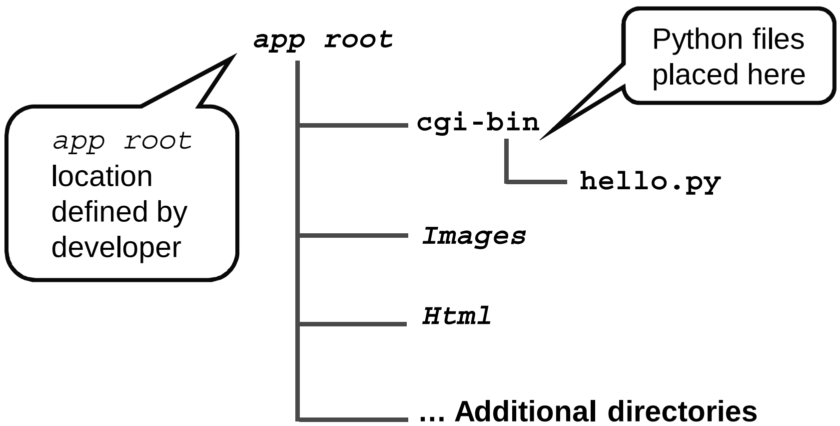
Python for Web Application Programming

- **Web applications generate dynamic responses to user requests**
  - Use programs known as server-side scripts
  - Can be written in a variety of programming languages
    - Java, C#, Ruby, Perl, and Python
- **Python programs can be used with all major web servers**
  - Apache
  - Internet Information Services (IIS)
  - Many more



Python Web Application Structure

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



Python Web Application

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

Set `Content-Type` for client

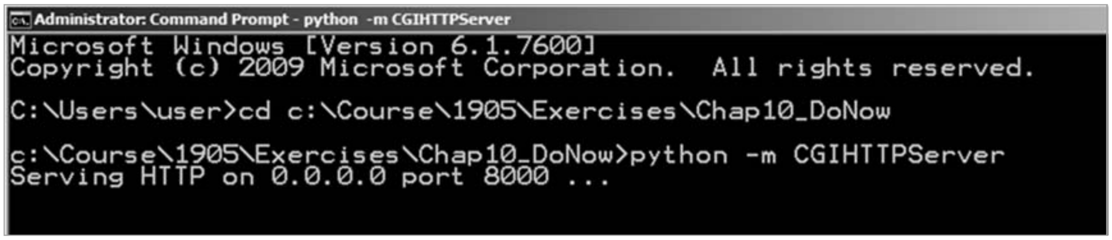
hello.py

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

print """
<html>

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


</html>

"""
```

A First Web Application

Do Now

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



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

A First Web Application

Do Now

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

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

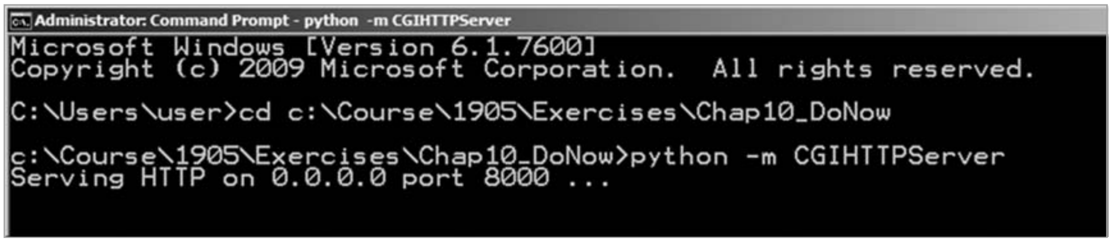
```
from datetime import date

date.today()
```

A First Web Application

Do Now

8. **Save your file**

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



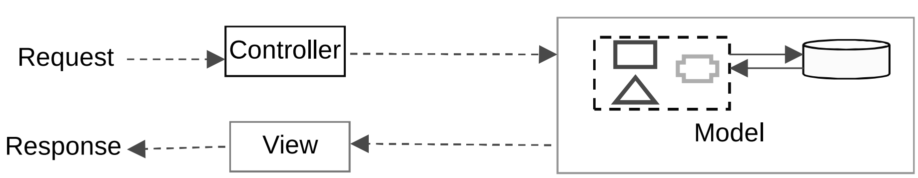
```
Administrator: Command Prompt - python  -m CGIHTTPServer
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\user>cd c:\Course\1905\Exercises\Chap10_DoNow

c:\Course\1905\Exercises\Chap10_DoNow>python -m CGIHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

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

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

Shortcomings of This Approach

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



Chapter Contents

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

Django Web Framework

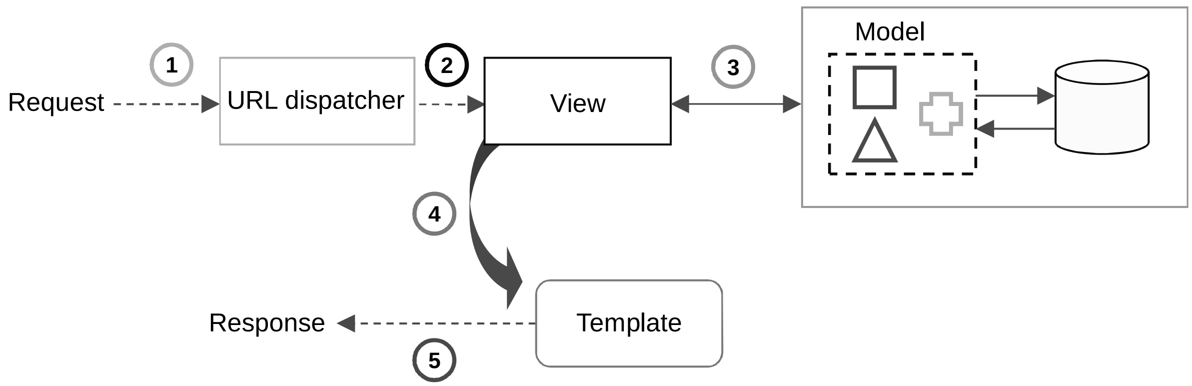
- **Open-source project**
  - Originally developed at Lawrence (Kan.) Journal-World Online
  - Django website: `https://www.djangoproject.com`
- **Python web application development framework**
  - Implements a variation of MVC
  - Enables developer to focus on building application functionality
   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

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Django Design Pattern

➤ **HTTP requests are processed as follows:**

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



Building a Web Application With Django

➤ **Application development proceeds as follows:**

A. Sketch application flow
   – Data to be submitted with request
   – Data required by template to generate the response

B. Implement view method
   – Processes request using model
   – Pass data (if required) to template

C. Implement model method
   – Handle data retrieval

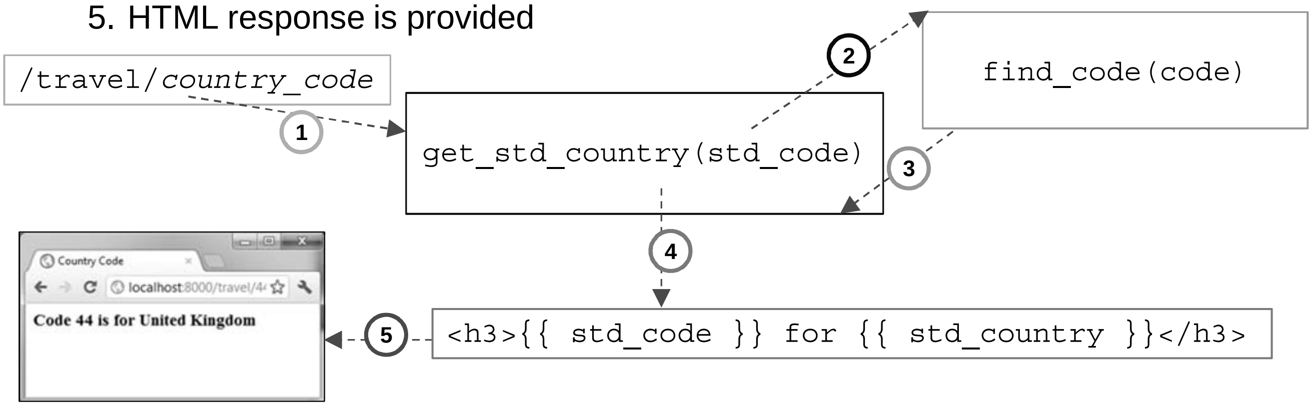
D. Implement template
   – Render HTML response

E. Map URL to view method

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Step A: Sketch Application Flow

- **Our workflow will enable an STD code to be submitted**
  - Response page will display associated country name
- **Example demonstrates passing data in request**
  1. URL dispatcher calls appropriate view
  2. View delegates data processing to model
  3. Model returns results to view
  4. View passes data to template
  5. HTML response is provided



STD = Subscriber Trunk Dialing

Step B: Define View Method

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

views.py

```
import country_code_lookup

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

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

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

Template name

Used by template in response

Step C: Implement Model Method

- **Data handler**
  - Request data received from view
  - Query data store
  - Return results to view

country_code_lookup.py

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

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

Returned to view

Step D: Implement Template

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

std_country_code.html

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

  <body>

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

  </body>
</html>
```

Data provided by view

Step E: Map URL to View Method

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

urls.py

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

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

URL pattern to match

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

View method to call

Application Flow Summary

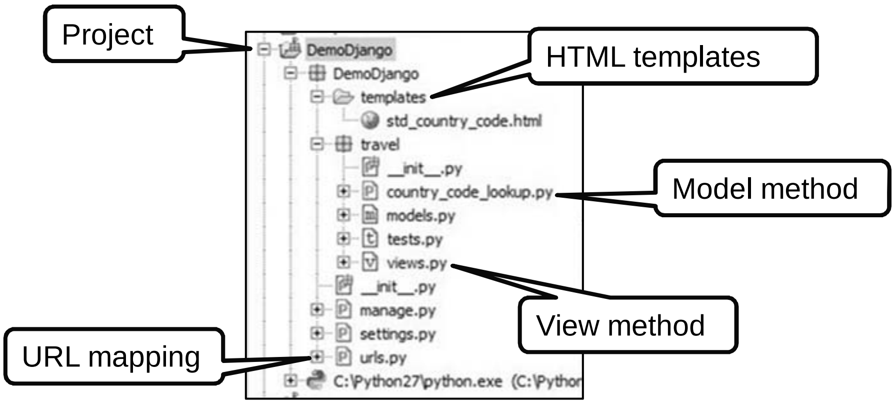
1. **URL maps to a view**
2. **View invokes model for data processing**
3. **View passes results to template**

Django Project Layout

Do Now

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

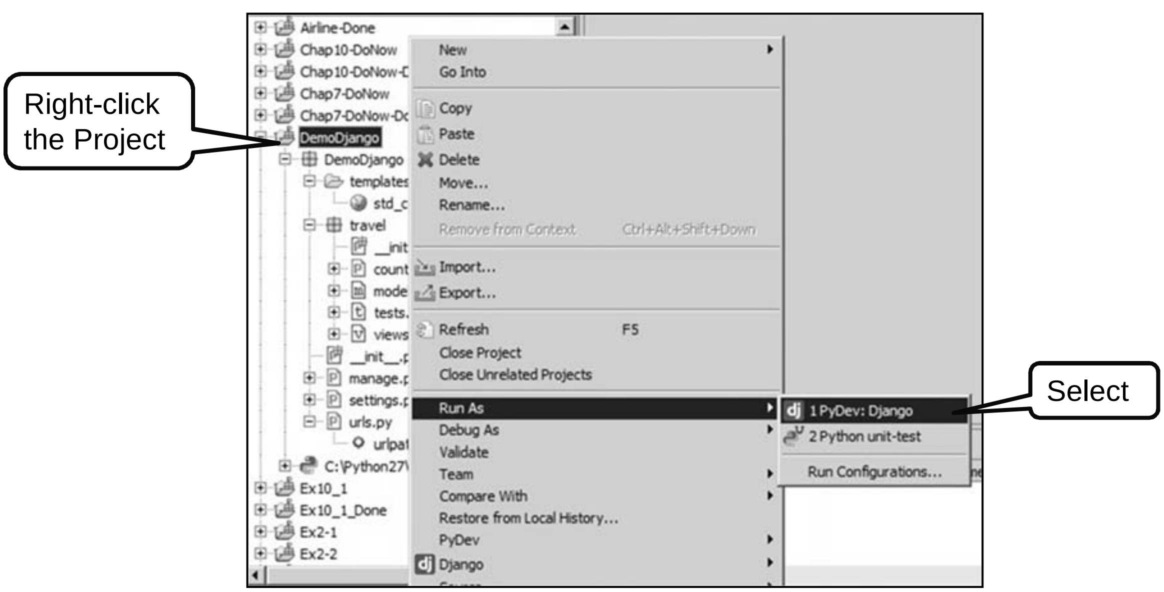
1. **In Eclipse, open the project `DemoDjango` to view the project infrastructure**



Django Demonstration

Do Now

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

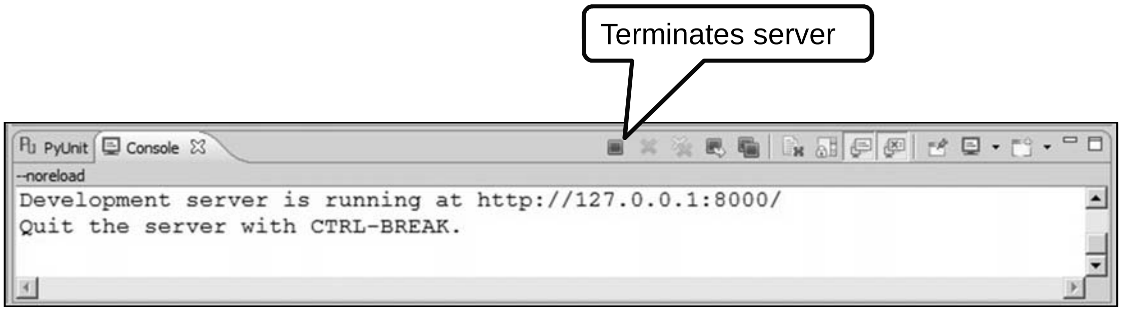


Django Demonstration

Do Now

3. The server startup message displays in Eclipse console:

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



Django Demonstration

Do Now

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

5. Template displays query result

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

Hands-On Exercise 10.1

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

Chapter Summary

You are now able to

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

Chapter 11

Course Summary

Learning Tree International

Training You Can Trust®

Course Summary

You are now able to

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

Certificate of Completion

Learning Tree International
Education You Can Trust®

EvUS/E.6/404/E.5

My Learning Tree: Continued Support

- **My Learning Tree benefits include:**
  - Print transcripts on demand
  - Gain instant access to free resources that keep you up to date on the latest technologies
  - Access to instructor-moderated forums for post-course follow-up
  - Access instructor blogs

To log in to your account, go to:
`LearningTree.com/MyLearningTree`

Follow us on Twitter: `@LearningTree`

LinkedIn group: Learning Tree International Alumni

`Facebook.com/LearningTreeIntl`

Certificate of Completion Request

➤ **Please review and complete your request for a Certificate of Completion**

- **Public course version**

Please indicate *exactly* how you would like your name to appear on the certificate

Please provide any missing information

Please correct any erroneous information

- **On-site course version**

Course Evaluation

➤ **We value your evaluation, thoughts, and suggestions**

➤ **We would like to know your level of satisfaction with**

- This course
- Your entire Learning Tree training experience

➤ **Your completion of a Course Evaluation will help us to**

- Improve the quality of instruction
- Improve the quality of the course content and the learning activities
- Provide a better educational experience for future participants
- Communicate more effectively with you and your colleagues about our education services

Course Evaluation: Please Comment About Your Experience

- **Public course version**

- **On-site course version**

In Conclusion…

Thank you, and we hope to see you in class again soon!

Appendix A

Unit Testing and Mocking

Learning Tree International

Training You Can Trust®

Chapter Objectives

By the end of this chapter, you will be able to

- **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 Nose and Pytest modules**
- **Mock dependent objects with the Mock package**

Chapter Contents

➡ **Principles of Testing**

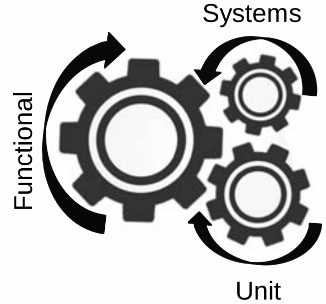
➤ **Writing Unit Tests in Python**

➤ **Executing Unit Tests With Nose and Pytest**

➤ **Using Mock Objects in Unit Tests**

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)
  - Nose and Pytest testing frameworks

Chapter Contents

- **Principles of Testing**
- ➡ **Writing Unit Tests in Python**
- **Executing Unit Tests With Nose and Pytest**
- **Using Mock Objects in Unit Tests**

Example: Person Class

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

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

    def __eq__(self, other):
        """Called when Persons are compared using == 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

    def __ne__(self, other):
        """Called when Persons are compared using != operator"""
        return not self.__eq__(other)
```

person.py

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

```
import unittest
from person import Person

class PersonTestCase(unittest.TestCase):
    """Unit tests for Person"""

    def test_init(self):
        person = Person("John", "Quincy", "Adams")

        self.assertEqual("John", person.first_name)
        self.assertEqual("Quincy", person.middle_name)
        self.assertEqual("Adams", person.last_name)

    # continued on next slide...
```

Test class extends `TestCase`

`assertEqual()` raises exception if values are not equal

Test class inherits `assertEqual()` from `TestCase`

test_person.py

Writing Unit Tests

- **`TestCase` `assert` methods raise exception if test condition is false**
  - Test runner reports failing test cases
- **Your unit test module calls `unittest.main()`**
  - Launches test runner

```
    # ... continued from previous slide

    def test_eq_instances_equal(self):
        p1 = Person("John", "Quincy", "Adams")
        p2 = Person("John", "Quincy", "Adams")
        self.assertEqual(p1, p2)

    def test_eq_instances_not_equal(self):
        p1 = Person("John", None, "Adams")
        p2 = Person("John", "Quincy", "Adams")
        self.assertNotEqual(p1, p2)

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

`assertTrue()` raises exception if expression is `False`

`assertEqual()` calls `p1.__eq__(p2)`

`assertNotEqual()` calls `p1.__ne__(p2)`

`main()` starts test runner

Running Unit Tests

- **Test runner executes test cases and reports results**

```
> cd \crs1906\examples\ch03_examples

> python -m unittest test_person.py
....
----------------------------------------------
Ran 4 tests in 0.002s

OK
```

Each "." represents a successful test case

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):
    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)
```

Function being tested returns a list

Set up expected return value

Call function and save actual return value

Compare expected value to actual value

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
```

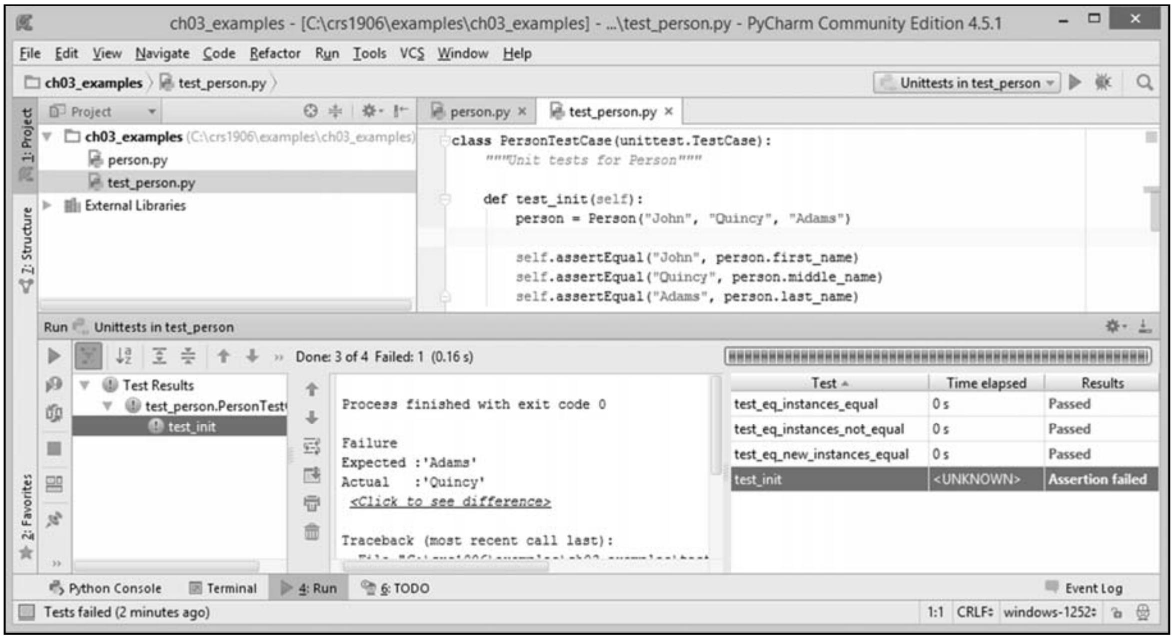
Bug in code!

```
> python -m unittest test_person_fail.py
...F
====================================================
FAIL: test_init (test_person.PersonTestCase)
----------------------------------------------------
Traceback (most recent call last):
  File "...\test_person_fail.py", line 23, in test_init
    self.assertEqual("Adams", person.last_name)
AssertionError: 'Adams' != 'Quincy'
- Adams
+ Quincy
----------------------------------------------------
Ran 4 tests in 0.003s
FAILED (failures=1)
```

Each "F" represents a failing test case

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:
    - `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):
            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="")
```

File context manger always closes file, even if exception is raised

Using explicit cleanup operation

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

Must close file explicitly

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

Chapter Contents

- Principles of Testing
- Writing Unit Tests in Python
- **Executing Unit Tests With Nose and Pytest**
- Using Mock Objects in Unit Tests

The Nose Framework

- **Nose is a popular third-party unit test framework**
- **Makes writing test cases simpler**
  - Test cases don't have to be subclasses of `unittest.TestCase`
  - Syntax for writing unit tests is simpler
- **Makes running test cases simpler**
  - Automatically searches directories for unit tests
  - Supports flexible test results reporting
- **Comes with a number of built-in plug-ins**
  - Output capture: Can capture standard output and calls to logging methods
  - Code coverage: Determines which application code was actually tested
- **Available on PyPI**
  - Installation: **`pip install nose`**

Writing Tests for Nose

- **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 defined as plain functions

Values compared with ==

`assert` statement raises `AssertionError` if test is `False`

No call to `unittest.main()`

test_person_nose.py

Running Tests With `nosetests`

- **Nose includes `nosetests` script**
  - Runs test cases in one file or multiple files

```
> cd \crs1906\examples\ch03_examples
> nosetests test_person_nose.py
....
----------------------------------------------------
Ran 4 tests in 0.003s
OK

> nosetests -v test_person_nose.py
test_person_nose.test_init ... ok
test_person_nose.test_eq_instances_equal ... ok
test_person_nose.test_eq_instances_not_equal ... ok
test_person_nose.test_eq_new_instances_equal ... ok
----------------------------------------------------
Ran 4 tests in 0.007s
OK
```

Run tests in one file

Run with verbose output

Reporting Test Failures

- **Nose displays results of failing test cases**
  - For more detail about test failure, run `nosetests -d`

```
> nosetests -d test_person_nose_fail.py
F...
======================================================================
FAIL: test_person_nose.test_init
----------------------------------------------------------------------
Traceback (most recent call last):
  ...
  File "C:\...\test_person_nose.py", line 17, in test_init
    (person.first_name, person.middle_name, person.last_name)

nose.proxy.AssertionError:
>>  assert ("John", "Quincy", "Adams") == \
  (<person.Person object>.first_name, <person.Person
   object>.middle_name, <person.Person object at ...>.last_name)
----------------------------------------------------------------------
Ran 4 tests in 0.035s

FAILED (failures=1)
```

Nose displays complete `assert` statement

Tools for Testing

- **Nose defines decorators for test cases**
- **`@raises(*exceptions)`**
  - Test passes if it raises one of the specified exceptions

```
from nose.tools import raises, timed

@raises(TypeError)
def test_raises_type_error():
    obj_under_test.do_work()
```

```
def do_work(self):
  if …:
    raise TypeError()
```

- **`@timed(limit)`**
  - Test must finish within specified time limit to pass

```
@timed(.1)   # seconds
def test_timed():
    ...
```

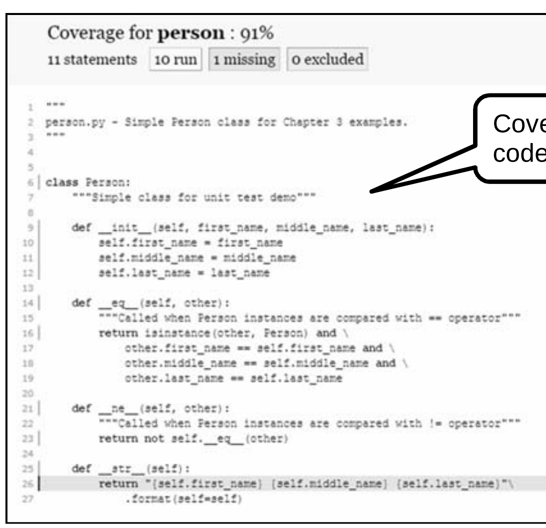
Reporting Code Coverage

- **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

- **Nose can determine what percentage of code under test was tested**
  - Run `nosetests --with-coverage --cover-html`
  - Creates `cover` directory with HTML coverage report

```
> nosetests --with-coverage --cover-html test_person_nose.py
```



Coverage report shows which code was executed by tests

Test Discovery

- **Nose will recursively search directories and run all tests**
  - Test module, class, or function must follow naming rules
    - Name must have `test` or `Test` at word boundary or following "-", "_", or "."
  - Examples
    - Class `TestView` in `test_view.py`
    - Class `ViewTest` in `viewtest.py`
    - Function `test_view_success` in `view_test.py`

```
> cd \crs1906\exercises\ticketmanor_webapp
> nosetests tests
.................................EE.....................
==============================================================
ERROR: test_get_news_max_items_1 (TestFeedReader)
--------------------------------------------------------------
Traceback (most recent call last):
...
--------------------------------------------------------------
Ran 61 tests in 13.853s
FAILED (errors=2)
```

`tests` is the top-level directory of test cases

Nose discovers and runs all tests in all subdirectories of `tests`

Choosing a Test Layout

- **Nose supports two common test layouts**

1. **Tests are in extra directory outside your actual application code**
   - Good for keeping tests separate from actual application code
   - May need to set `PYTHONPATH` to import modules if you run single tests

```
project_dir
    setup.py   # setuptools Python package metadata
    mypkg/
        __init__.py
        business_object.py
        person.py
        user_dao.py
    tests/
        __init__.py
        test_business_object.py
        test_person.py
        test_user_dao.py
```

Application package

Test package is sibling of application package

Choosing a Test Layout

2. Inline test directories in your application package

- Causes fewer problems when importing modules in test cases
- Assumes you want to distribute your tests along with your application

```
project_dir
    setup.py  # setuptools Python package metadata
    mypkg/
        __init__.py
        business_object.py
        person.py
        user_dao.py
        tests/
            __init__.py
            test_business_object.py
            test_person.py
            test_user_dao.py
```

Application package

Test package is nested under application package

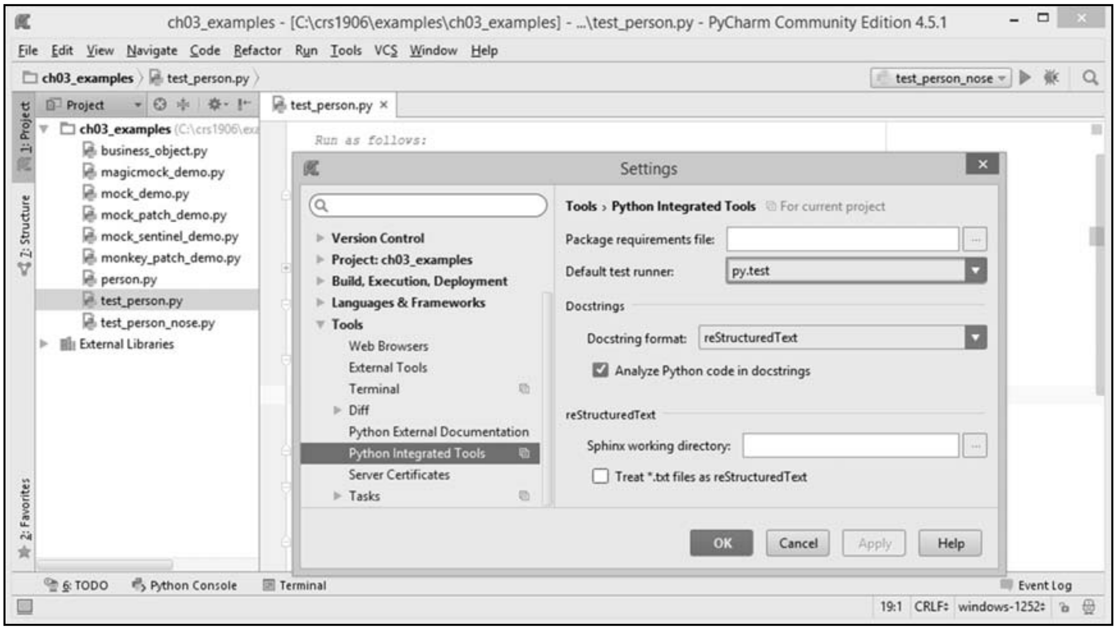
The Pytest Framework

- **Pytest is another popular third-party unit test framework**
  - Nose and Pytest modules were developed from a common code base
  - Pytest has same advantages as Nose
    - Simpler syntax for writing test cases
    - Automatic test discovery
    - Flexible test results reporting
    - Code coverage reporting
- **Pytest is available on PyPI**
  - Installation: **`pip install pytest`**
- **Pytest includes a `py.test` script**

```
> cd \crs1906\exercises\ticketmanor_webapp
> py.test tests
```

Using Nose or Pytest for Unit Tests in PyCharm

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



Chapter Contents

- **Principles of Testing**
- **Writing Unit Tests in Python**
- **Executing Unit Tests With Nose and Pytest**
- **Using Mock Objects in Unit Tests**

Chapter Contents

- Principles of Testing
- Writing Unit Tests in Python
- Executing Unit Tests With Nose and Pytest
- **Using Mock Objects in Unit Tests**

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**
  - Note that this is not a unit test; you can use monkey patching anywhere

```
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()
```

Our monkey patch function

Replace old method with new method

Call to `increment` calls new function with `counter` as first arg

monkey_patch_demo.py

Using Mock Objects in Unit Tests

- **Unit test creates mock `UserDao` object**
  - Test cases replace production DAO with mock DAO
- **Goal: test `BusinessObject.get_user()`**
  - Mock's `spec` constructor argument gives mock same interface as `UserDao`

```
class TestBusinessObject(TestCase):
  def test_get_user(self):
    expected_result = Person('Isaac', None, 'Newton')
    mock_dao = Mock(spec=UserDao)

    mock_dao.query_user.return_value = expected_result

    bus_obj = BusinessObject()
    bus_obj.user_dao = mock_dao

    actual_result = bus_obj.get_user(123)

    self.assertEquals(expected_result, actual_result)
```

Create mock DAO

Set return value of mock method

Monkey patch: replace real DAO with mock DAO

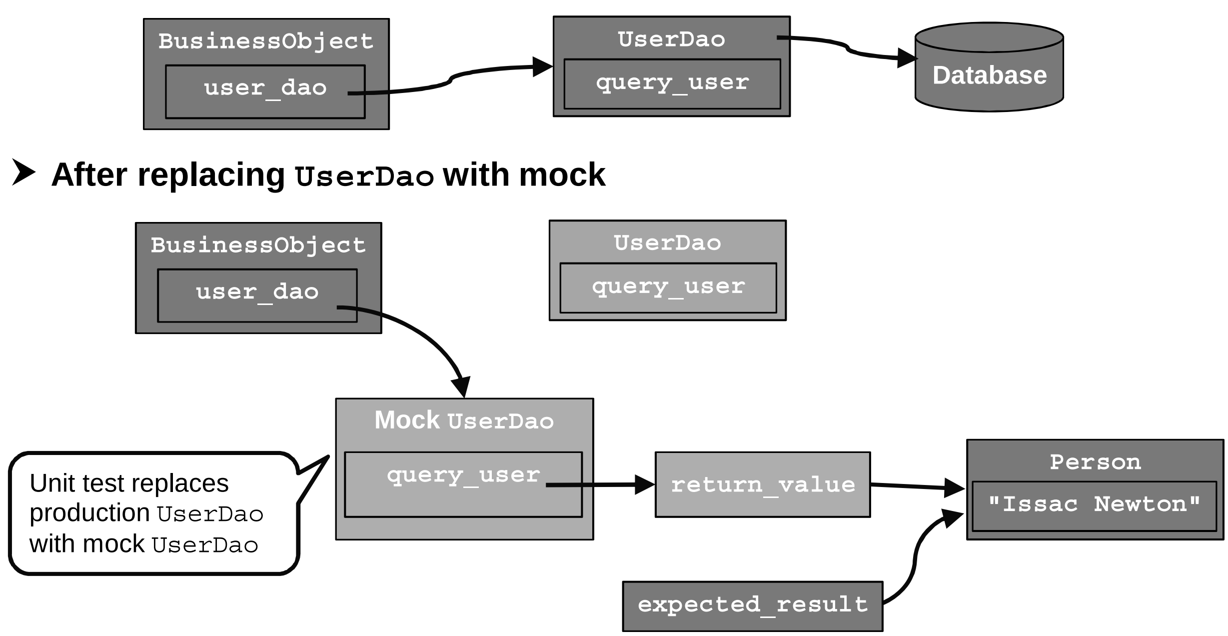
Business method uses mock DAO instead of real DAO

Verify actual result equals expected result

mock_demo.py

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

```
def test_get_user_not_found(self):
    mock_dao = Mock(spec=UserDao)
    mock_dao.query_user.return_value = None

    bus_obj = BusinessObject()
    bus_obj.user_dao = mock_dao

    user_id = 123
    with self.assertRaises(ValueError):
        bus_obj.get_user(user_id)
```

Business method should raise exception if return value is `None`

Verify business method raises exception

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(self):
    mock_dao = Mock(spec=UserDao)
    mock_dao.query_user.side_effect = sqlite3.Error('SQL error')

    bus_obj = BusinessObject()
    bus_obj.user_dao = mock_dao

    user_id = 123
    with self.assertRaisesRegex(BusinessError, '[Pp]roblem.*user'):
        bus_obj.get_user(user_id)
```

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

Verify business object handled DB error correctly

For more about mocking in unit tests, see Appendix B

Chapter Contents

- Principles of Testing
- Writing Unit Tests in Python
- Executing Unit Tests With Nose and Pytest
- Using Mock Objects in Unit Tests

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

Chapter Summary

In this chapter, you have learned to

- **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 Nose and Pytest modules**
- **Mock dependent objects with the Mock package**

The lxml.etree Tutorial

Author: Stefan Behnel

This is a tutorial on XML processing with `lxml.etree`. It briefly overviews the main concepts of the **ElementTree API**, and some simple enhancements that make your life as a programmer easier.

For a complete reference of the API, see the **generated API documentation**.

Contents

- **The Element class**
  - **Elements are lists**
  - **Elements carry attributes as a dict**
  - **Elements contain text**
  - **Using XPath to find text**
  - **Tree iteration**
  - **Serialisation**
- **The ElementTree class**
- **Parsing from strings and files**
  - **The fromstring() function**
  - **The XML() function**
  - **The parse() function**
  - **Parser objects**
  - **Incremental parsing**
  - **Event-driven parsing**
- **Namespaces**
- **The E-factory**
- **ElementPath**

A common way to import `lxml.etree` is as follows:

```
>>> from lxml import etree
```

If your code only uses the ElementTree API and does not rely on any functionality that is specific to `lxml.etree`, you can also use (any part of) the following import chain as a fall-back to the original ElementTree:

```
try:
  from lxml import etree
  print("running with lxml.etree")
except ImportError:
  try:
    # Python 2.5
    import xml.etree.cElementTree as etree
    print("running with cElementTree on Python 2.5+")
  except ImportError:
    try:
      # Python 2.5
      import xml.etree.ElementTree as etree
      print("running with ElementTree on Python 2.5+")
    except ImportError:
      try:
        # normal cElementTree install
        import cElementTree as etree
        print("running with cElementTree")
      except ImportError:
        try:
          # normal ElementTree install
          import elementtree.ElementTree as etree
          print("running with ElementTree")
        except ImportError:
          print("Failed to import ElementTree from any known place")
```

To aid in writing portable code, this tutorial makes it clear in the examples which part of the presented API is an extension of `lxml.etree` over the original **ElementTree API**, as defined by Fredrik Lundh's **ElementTree library**.

The Element class

An `Element` is the main container object for the ElementTree API. Most of the XML tree functionality is accessed through this class. Elements are easily created through the `Element` factory:

```
>>> root = etree.Element("root")
```

The XML tag name of elements is accessed through the `tag` property:

```
>>> print(root.tag)
root
```

Elements are organised in an XML tree structure. To create child elements and add them to a parent element, you can use the `append()` method:

```
>>> root.append( etree.Element("child1") )
```

However, this is so common that there is a shorter and much more efficient way to do this: the `SubElement` factory. It accepts the same arguments as the `Element` factory, but additionally requires the parent as first argument:

```
>>> child2 = etree.SubElement(root, "child2")
>>> child3 = etree.SubElement(root, "child3")
```

To see that this is really XML, you can serialise the tree you have created:

```
>>> print(etree.tostring(root, pretty_print=True))
<root>
  <child1/>
  <child2/>
  <child3/>
</root>
```

Elements are lists

To make the access to these subelements easy and straight forward, elements mimic the behaviour of normal Python lists as closely as possible:

```
>>> child = root[0]
>>> print(child.tag)
child1

>>> print(len(root))
3

>>> root.index(root[1]) # lxml.etree only!
1

>>> children = list(root)

>>> for child in root:
...     print(child.tag)
child1
child2
child3

>>> root.insert(0, etree.Element("child0"))
>>> start = root[:1]
>>> end   = root[-1:]

>>> print(start[0].tag)
child0
>>> print(end[0].tag)
```

```
child3
```

Prior to ElementTree 1.3 and lxml 2.0, you could also check the truth value of an Element to see if it has children, i.e. if the list of children is empty:

```
if root:   # this no longer works!
    print("The root element has children")
```

This is no longer supported as people tend to expect that a "something" evaluates to True and expect Elements to be "something", may they have children or not. So, many users find it surprising that any Element would evaluate to False in an if-statement like the above. Instead, use `len(element)`, which is both more explicit and less error prone.

```
>>> print(etree.iselement(root))  # test if it's some kind of Element
True
>>> if len(root):                 # test if it has children
...     print("The root element has children")
The root element has children
```

There is another important case where the behaviour of Elements in lxml (in 2.0 and later) deviates from that of lists and from that of the original ElementTree (prior to version 1.3 or Python 2.7/3.2):

```
>>> for child in root:
...     print(child.tag)
child0
child1
child2
child3
>>> root[0] = root[-1]  # this moves the element in lxml.etree!
>>> for child in root:
...     print(child.tag)
child3
child1
child2
```

In this example, the last element is *moved* to a different position, instead of being copied, i.e. it is automatically removed from its previous position when it is put in a different place. In lists, objects can appear in multiple positions at the same time, and the above assignment would just copy the item reference into the first position, so that both contain the exact same item:

```
>>> l = [0, 1, 2, 3]
>>> l[0] = l[-1]
>>> l
[3, 1, 2, 3]
```

Note that in the original ElementTree, a single Element object can sit in any number of places in any number of trees, which allows for the same copy operation as with lists. The obvious drawback is that modifications to such an Element will apply to all places where it appears in a tree, which may or may not be intended.

The upside of this difference is that an Element in `lxml.etree` always has exactly one parent, which can be queried through the `getparent()` method. This is not supported in the original ElementTree.

```
>>> root is root[0].getparent()  # lxml.etree only!
True
```

If you want to *copy* an element to a different position in `lxml.etree`, consider creating an independent *deep copy* using the `copy` module from Python's standard library:

```
>>> from copy import deepcopy

>>> element = etree.Element("neu")
>>> element.append( deepcopy(root[1]) )
```

```
>>> print(element[0].tag)
child1
>>> print([ c.tag for c in root ])
['child3', 'child1', 'child2']
```

The siblings (or neighbours) of an element are accessed as next and previous elements:

```
>>> root[0] is root[1].getprevious() # lxml.etree only!
True
>>> root[1] is root[0].getnext() # lxml.etree only!
True
```

Elements carry attributes as a dict

XML elements support attributes. You can create them directly in the Element factory:

```
>>> root = etree.Element("root", interesting="totally")
>>> etree.tostring(root)
b'<root interesting="totally"/>'
```

Attributes are just unordered name-value pairs, so a very convenient way of dealing with them is through the dictionary-like interface of Elements:

```
>>> print(root.get("interesting"))
totally

>>> print(root.get("hello"))
None
>>> root.set("hello", "Huhu")
>>> print(root.get("hello"))
Huhu

>>> etree.tostring(root)
b'<root interesting="totally" hello="Huhu"/>'

>>> sorted(root.keys())
['hello', 'interesting']

>>> for name, value in sorted(root.items()):
...     print('%s = %r' % (name, value))
hello = 'Huhu'
interesting = 'totally'
```

For the cases where you want to do item lookup or have other reasons for getting a 'real' dictionary-like object, e.g. for passing it around, you can use the `attrib` property:

```
>>> attributes = root.attrib

>>> print(attributes["interesting"])
totally
>>> print(attributes.get("no-such-attribute"))
None

>>> attributes["hello"] = "Guten Tag"
>>> print(attributes["hello"])
Guten Tag
>>> print(root.get("hello"))
Guten Tag
```

Note that `attrib` is a dict-like object backed by the Element itself. This means that any changes to the Element are reflected in `attrib` and vice versa. It also means that the XML tree stays alive in memory as long as the `attrib` of one of its Elements is in use. To get an independent snapshot of the attributes that does not depend on the XML tree, copy it into a dict:

```
>>> d = dict(root.attrib)
```

```
>>> sorted(d.items())
[('hello', 'Guten Tag'), ('interesting', 'totally')]
```

Elements contain text

Elements can contain text:

```
>>> root = etree.Element("root")
>>> root.text = "TEXT"

>>> print(root.text)
TEXT

>>> etree.tostring(root)
b'<root>TEXT</root>'
```

In many XML documents (*data-centric* documents), this is the only place where text can be found. It is encapsulated by a leaf tag at the very bottom of the tree hierarchy.

However, if XML is used for tagged text documents such as (X)HTML, text can also appear between different elements, right in the middle of the tree:

```
<html><body>Hello<br/>World</body></html>
```

Here, the `<br/>` tag is surrounded by text. This is often referred to as *document-style* or *mixed-content* XML. Elements support this through their `tail` property. It contains the text that directly follows the element, up to the next element in the XML tree:

```
>>> html = etree.Element("html")
>>> body = etree.SubElement(html, "body")
>>> body.text = "TEXT"

>>> etree.tostring(html)
b'<html><body>TEXT</body></html>'

>>> br = etree.SubElement(body, "br")
>>> etree.tostring(html)
b'<html><body>TEXT<br/></body></html>'

>>> br.tail = "TAIL"
>>> etree.tostring(html)
b'<html><body>TEXT<br/>TAIL</body></html>'
```

The two properties `.text` and `.tail` are enough to represent any text content in an XML document. This way, the ElementTree API does not require any **special text nodes** in addition to the Element class, that tend to get in the way fairly often (as you might know from classic **DOM** APIs).

However, there are cases where the tail text also gets in the way. For example, when you serialise an Element from within the tree, you do not always want its tail text in the result (although you would still want the tail text of its children). For this purpose, the `tostring()` function accepts the keyword argument `with_tail`:

```
>>> etree.tostring(br)
b'<br/>TAIL'
>>> etree.tostring(br, with_tail=False) # lxml.etree only!
b'<br/>'
```

If you want to read *only* the text, i.e. without any intermediate tags, you have to recursively concatenate all `text` and `tail` attributes in the correct order. Again, the `tostring()` function comes to the rescue, this time using the `method` keyword:

```
>>> etree.tostring(html, method="text")
b'TEXTTAIL'
```

Using XPath to find text

Another way to extract the text content of a tree is **XPath**, which also allows you to extract the separate text chunks into a list:

```
>>> print(html.xpath("string()")) # lxml.etree only!
TEXTTAIL
>>> print(html.xpath("//text()")) # lxml.etree only!
['TEXT', 'TAIL']
```

If you want to use this more often, you can wrap it in a function:

```
>>> build_text_list = etree.XPath("//text()") # lxml.etree only!
>>> print(build_text_list(html))
['TEXT', 'TAIL']
```

Note that a string result returned by XPath is a special 'smart' object that knows about its origins. You can ask it where it came from through its `getparent()` method, just as you would with Elements:

```
>>> texts = build_text_list(html)
>>> print(texts[0])
TEXT
>>> parent = texts[0].getparent()
>>> print(parent.tag)
body

>>> print(texts[1])
TAIL
>>> print(texts[1].getparent().tag)
br
```

You can also find out if it's normal text content or tail text:

```
>>> print(texts[0].is_text)
True
>>> print(texts[1].is_text)
False
>>> print(texts[1].is_tail)
True
```

While this works for the results of the `text()` function, lxml will not tell you the origin of a string value that was constructed by the XPath functions `string()` or `concat()`:

```
>>> stringify = etree.XPath("string()")
>>> print(stringify(html))
TEXTTAIL
>>> print(stringify(html).getparent())
None
```

Tree iteration

For problems like the above, where you want to recursively traverse the tree and do something with its elements, tree iteration is a very convenient solution. Elements provide a tree iterator for this purpose. It yields elements in *document order*, i.e. in the order their tags would appear if you serialised the tree to XML:

```
>>> root = etree.Element("root")
>>> etree.SubElement(root, "child").text = "Child 1"
>>> etree.SubElement(root, "child").text = "Child 2"
>>> etree.SubElement(root, "another").text = "Child 3"

>>> print(etree.tostring(root, pretty_print=True))
<root>
  <child>Child 1</child>
  <child>Child 2</child>
  <another>Child 3</another>
</root>
```

```
>>> for element in root.iter():
...     print("%s - %s" % (element.tag, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3
```

If you know you are only interested in a single tag, you can pass its name to `iter()` to have it filter for you. Starting with lxml 3.0, you can also pass more than one tag to intercept on multiple tags during iteration.

```
>>> for element in root.iter("child"):
...     print("%s - %s" % (element.tag, element.text))
child - Child 1
child - Child 2

>>> for element in root.iter("another", "child"):
...     print("%s - %s" % (element.tag, element.text))
child - Child 1
child - Child 2
another - Child 3
```

By default, iteration yields all nodes in the tree, including ProcessingInstructions, Comments and Entity instances. If you want to make sure only Element objects are returned, you can pass the `Element` factory as tag parameter:

```
>>> root.append(etree.Entity("#234"))
>>> root.append(etree.Comment("some comment"))

>>> for element in root.iter():
...     if isinstance(element.tag, basestring):  # or 'str' in Python 3
...         print("%s - %s" % (element.tag, element.text))
...     else:
...         print("SPECIAL: %s - %s" % (element, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3
SPECIAL: &#234; - &#234;
SPECIAL: <!--some comment--> - some comment

>>> for element in root.iter(tag=etree.Element):
...     print("%s - %s" % (element.tag, element.text))
root - None
child - Child 1
child - Child 2
another - Child 3

>>> for element in root.iter(tag=etree.Entity):
...     print(element.text)
&#234;
```

Note that passing a wildcard `"*"` tag name will also yield all `Element` nodes (and only elements).

In `lxml.etree`, elements provide **further iterators** for all directions in the tree: children, parents (or rather ancestors) and siblings.

Serialisation

Serialisation commonly uses the `tostring()` function that returns a string, or the `ElementTree.write()` method that writes to a file, a file-like object, or a URL (via FTP PUT or HTTP POST). Both calls accept the same keyword arguments like `pretty_print` for formatted output or `encoding` to select a specific output encoding other than plain ASCII:

```
>>> root = etree.XML('<root><a><b/></a></root>')

>>> etree.tostring(root)
b'<root><a><b/></a></root>'
```

```
>>> print(etree.tostring(root, xml_declaration=True))
<?xml version='1.0' encoding='ASCII'?>
<root><a><b/></a></root>

>>> print(etree.tostring(root, encoding='iso-8859-1'))
<?xml version='1.0' encoding='iso-8859-1'?>
<root><a><b/></a></root>

>>> print(etree.tostring(root, pretty_print=True))
<root>
  <a>
    <b/>
  </a>
</root>
```

Note that pretty printing appends a newline at the end.

In lxml 2.0 and later (as well as ElementTree 1.3), the serialisation functions can do more than XML serialisation. You can serialise to HTML or extract the text content by passing the `method` keyword:

```
>>> root = etree.XML(
...    '<html><head/><body><p>Hello<br/>World</p></body></html>')

>>> etree.tostring(root) # default: method = 'xml'
b'<html><head/><body><p>Hello<br/>World</p></body></html>'

>>> etree.tostring(root, method='xml') # same as above
b'<html><head/><body><p>Hello<br/>World</p></body></html>'

>>> etree.tostring(root, method='html')
b'<html><head></head><body><p>Hello<br>World</p></body></html>'

>>> print(etree.tostring(root, method='html', pretty_print=True))
<html>
<head></head>
<body><p>Hello<br>World</p></body>
</html>

>>> etree.tostring(root, method='text')
b'HelloWorld'
```

As for XML serialisation, the default encoding for plain text serialisation is ASCII:

```
>>> br = next(root.iter('br'))  # get first result of iteration
>>> br.tail = u'W\xf6rld'

>>> etree.tostring(root, method='text')  # doctest: +ELLIPSIS
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character u'\xf6' ...

>>> etree.tostring(root, method='text', encoding="UTF-8")
b'HelloW\xc3\xb6rld'
```

Here, serialising to a Python unicode string instead of a byte string might become handy. Just pass the name `'unicode'` as encoding:

```
>>> etree.tostring(root, encoding='unicode', method='text')
u'HelloW\xf6rld'
```

The W3C has a good **article about the Unicode character set and character encodings**.

The ElementTree class

An `ElementTree` is mainly a document wrapper around a tree with a root node. It provides a couple of methods for serialisation and general document handling.

```
>>> root = etree.XML('''\
... <?xml version="1.0"?>
... <!DOCTYPE root SYSTEM "test" [ <!ENTITY tasty "parsnips"> ]>
... <root>
...   <a>&tasty;</a>
... </root>
... ''')

>>> tree = etree.ElementTree(root)
>>> print(tree.docinfo.xml_version)
1.0
>>> print(tree.docinfo.doctype)
<!DOCTYPE root SYSTEM "test">

>>> tree.docinfo.public_id = '-//W3C//DTD XHTML 1.0 Transitional//EN'
>>> tree.docinfo.system_url = 'file://local.dtd'
>>> print(tree.docinfo.doctype)
<!DOCTYPE root PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "file://local.dtd">
```

An `ElementTree` is also what you get back when you call the `parse()` function to parse files or file-like objects (see the parsing section below).

One of the important differences is that the `ElementTree` class serialises as a complete document, as opposed to a single `Element`. This includes top-level processing instructions and comments, as well as a DOCTYPE and other DTD content in the document:

```
>>> print(etree.tostring(tree))  # lxml 1.3.4 and later
<!DOCTYPE root PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "file://local.dtd" [
<!ENTITY tasty "parsnips">
]>
<root>
  <a>parsnips</a>
</root>
```

In the original xml.etree.ElementTree implementation and in lxml up to 1.3.3, the output looks the same as when serialising only the root Element:

```
>>> print(etree.tostring(tree.getroot()))
<root>
  <a>parsnips</a>
</root>
```

This serialisation behaviour has changed in lxml 1.3.4. Before, the tree was serialised without DTD content, which made lxml lose DTD information in an input-output cycle.

Parsing from strings and files

`lxml.etree` supports parsing XML in a number of ways and from all important sources, namely strings, files, URLs (http/ftp) and file-like objects. The main parse functions are `fromstring()` and `parse()`, both called with the source as first argument. By default, they use the standard parser, but you can always pass a different parser as second argument.

The fromstring() function

The `fromstring()` function is the easiest way to parse a string:

```
>>> some_xml_data = "<root>data</root>"

>>> root = etree.fromstring(some_xml_data)
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

The XML() function

The `XML()` function behaves like the `fromstring()` function, but is commonly used to write XML literals right into the source:

```
>>> root = etree.XML("<root>data</root>")
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

There is also a corresponding function `HTML()` for HTML literals.

The parse() function

The `parse()` function is used to parse from files and file-like objects.

As an example of such a file-like object, the following code uses the `BytesIO` class for reading from a string instead of an external file. That class comes from the `io` module in Python 2.6 and later. In older Python versions, you will have to use the `StringIO` class from the `StringIO` module. However, in real life, you would obviously avoid doing this all together and use the string parsing functions above.

```
>>> some_file_like_object = BytesIO("<root>data</root>")

>>> tree = etree.parse(some_file_like_object)

>>> etree.tostring(tree)
b'<root>data</root>'
```

Note that `parse()` returns an ElementTree object, not an Element object as the string parser functions:

```
>>> root = tree.getroot()
>>> print(root.tag)
root
>>> etree.tostring(root)
b'<root>data</root>'
```

The reasoning behind this difference is that `parse()` returns a complete document from a file, while the string parsing functions are commonly used to parse XML fragments.

The `parse()` function supports any of the following sources:

- an open file object (make sure to open it in binary mode)
- a file-like object that has a `.read(byte_count)` method returning a byte string on each call
- a filename string
- an HTTP or FTP URL string

Note that passing a filename or URL is usually faster than passing an open file or file-like object. However, the HTTP/FTP client in libxml2 is rather simple, so things like HTTP authentication require a dedicated URL request library, e.g. `urllib2` or `request`. These libraries usually provide a file-like object for the result that you can parse from while the response is streaming in.

Parser objects

By default, `lxml.etree` uses a standard parser with a default setup. If you want to configure the parser, you can create a you instance:

```
>>> parser = etree.XMLParser(remove_blank_text=True) # lxml.etree only!
```

This creates a parser that removes empty text between tags while parsing, which can reduce the size of the tree and avoid dangling tail text if you know that whitespace-only content is not meaningful for your data. An example:

```
>>> root = etree.XML("<root>  <a/>   <b>  </b>     </root>", parser)
```

```
>>> etree.tostring(root)
b'<root><a/><b>  </b></root>'
```

Note that the whitespace content inside the <b> tag was not removed, as content at leaf elements tends to be data content (even if blank). You can easily remove it in an additional step by traversing the tree:

```
>>> for element in root.iter("*"):
...     if element.text is not None and not element.text.strip():
...         element.text = None

>>> etree.tostring(root)
b'<root><a/><b/></root>'
```

See `help(etree.XMLParser)` to find out about the available parser options.

Incremental parsing

`lxml.etree` provides two ways for incremental step-by-step parsing. One is through file-like objects, where it calls the `read()` method repeatedly. This is best used where the data arrives from a source like `urllib` or any other file-like object that can provide data on request. Note that the parser will block and wait until data becomes available in this case:

```
>>> class DataSource:
...     data = [ b"<roo", b"t><", b"a/", b"><", b"/root>" ]
...     def read(self, requested_size):
...         try:
...             return self.data.pop(0)
...         except IndexError:
...             return b''

>>> tree = etree.parse(DataSource())

>>> etree.tostring(tree)
b'<root><a/></root>'
```

The second way is through a feed parser interface, given by the `feed(data)` and `close()` methods:

```
>>> parser = etree.XMLParser()

>>> parser.feed("<roo")
>>> parser.feed("t><")
>>> parser.feed("a/")
>>> parser.feed("><")
>>> parser.feed("/root>")

>>> root = parser.close()

>>> etree.tostring(root)
b'<root><a/></root>'
```

Here, you can interrupt the parsing process at any time and continue it later on with another call to the `feed()` method. This comes in handy if you want to avoid blocking calls to the parser, e.g. in frameworks like Twisted, or whenever data comes in slowly or in chunks and you want to do other things while waiting for the next chunk.

After calling the `close()` method (or when an exception was raised by the parser), you can reuse the parser by calling its `feed()` method again:

```
>>> parser.feed("<root/>")
>>> root = parser.close()
>>> etree.tostring(root)
b'<root/>'
```

Event-driven parsing

Sometimes, all you need from a document is a small fraction somewhere deep inside the tree, so parsing the whole tree into memory, traversing it and dropping it can be too much overhead. `lxml.etree` supports this use case with two event-driven parser interfaces, one that generates parser events while building the tree (`iterparse`), and one that does not build the tree at all, and instead calls feedback methods on a target object in a SAX-like fashion.

Here is a simple `iterparse()` example:

```
>>> some_file_like = BytesIO("<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     print("%s, %4s, %s" % (event, element.tag, element.text))
end,    a, data
end, root, None
```

By default, `iterparse()` only generates events when it is done parsing an element, but you can control this through the `events` keyword argument:

```
>>> some_file_like = BytesIO("<root><a>data</a></root>")

>>> for event, element in etree.iterparse(some_file_like,
...                                       events=("start", "end")):
...     print("%5s, %4s, %s" % (event, element.tag, element.text))
start, root, None
start,    a, data
  end,    a, data
  end, root, None
```

Note that the text, tail, and children of an Element are not necessarily present yet when receiving the `start` event. Only the `end` event guarantees that the Element has been parsed completely.

It also allows you to `.clear()` or modify the content of an Element to save memory. So if you parse a large tree and you want to keep memory usage small, you should clean up parts of the tree that you no longer need:

```
>>> some_file_like = BytesIO(
...     "<root><a><b>data</b></a><a><b/></a></root>")

>>> for event, element in etree.iterparse(some_file_like):
...     if element.tag == 'b':
...         print(element.text)
...     elif element.tag == 'a':
...         print("** cleaning up the subtree")
...         element.clear()
data
** cleaning up the subtree
None
** cleaning up the subtree
```

A very important use case for `iterparse()` is parsing large generated XML files, e.g. database dumps. Most often, these XML formats only have one main data item element that hangs directly below the root node and that is repeated thousands of times. In this case, it is best practice to let `lxml.etree` do the tree building and only to intercept on exactly this one Element, using the normal tree API for data extraction.

```
>>> xml_file = BytesIO('''\
... <root>
...   <a><b>ABC</b><c>abc</c></a>
...   <a><b>MORE DATA</b><c>more data</c></a>
...   <a><b>XYZ</b><c>xyz</c></a>
... </root>''')

>>> for _, element in etree.iterparse(xml_file, tag='a'):
...     print('%s -- %s' % (element.findtext('b'), element[1].text))
...     element.clear()
ABC -- abc
MORE DATA -- more data
XYZ -- xyz
```

If, for some reason, building the tree is not desired at all, the target parser interface of `lxml.etree` can be used. It creates

SAX-like events by calling the methods of a target object. By implementing some or all of these methods, you can control which events are generated:

```
>>> class ParserTarget:
...     events = []
...     close_count = 0
...     def start(self, tag, attrib):
...         self.events.append(("start", tag, attrib))
...     def close(self):
...         events, self.events = self.events, []
...         self.close_count += 1
...         return events

>>> parser_target = ParserTarget()

>>> parser = etree.XMLParser(target=parser_target)
>>> events = etree.fromstring('<root test="true"/>', parser)

>>> print(parser_target.close_count)
1

>>> for event in events:
...     print('event: %s - tag: %s' % (event[0], event[1]))
...     for attr, value in event[2].items():
...         print(' * %s = %s' % (attr, value))
event: start - tag: root
 * test = true
```

You can reuse the parser and its target as often as you like, so you should take care that the `.close()` method really resets the target to a usable state (also in the case of an error!).

```
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
2
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
3
>>> events = etree.fromstring('<root test="true"/>', parser)
>>> print(parser_target.close_count)
4

>>> for event in events:
...     print('event: %s - tag: %s' % (event[0], event[1]))
...     for attr, value in event[2].items():
...         print(' * %s = %s' % (attr, value))
event: start - tag: root
 * test = true
```

Namespaces

The ElementTree API avoids **namespace prefixes** wherever possible and deploys the real namespace (the URI) instead:

```
>>> xhtml = etree.Element("{http://www.w3.org/1999/xhtml}html")
>>> body = etree.SubElement(xhtml, "{http://www.w3.org/1999/xhtml}body")
>>> body.text = "Hello World"

>>> print(etree.tostring(xhtml, pretty_print=True))
<html:html xmlns:html="http://www.w3.org/1999/xhtml">
  <html:body>Hello World</html:body>
</html:html>
```

The notation that ElementTree uses was originally brought up by **James Clark**. It has the major advantage of providing a universally qualified name for a tag, regardless of any prefixes that may or may not have been used or defined in a document. By moving the indirection of prefixes out of the way, it makes namespace aware code much clearer and easier to get right.

As you can see from the example, prefixes only become important when you serialise the result. However, the above code looks somewhat verbose due to the lengthy namespace names. And retyping or copying a string over and over again is error prone. It is therefore common practice to store a namespace URI in a global variable. To adapt the namespace prefixes for serialisation, you can also pass a mapping to the Element factory function, e.g. to define the default namespace:

```
>>> XHTML_NAMESPACE = "http://www.w3.org/1999/xhtml"
>>> XHTML = "{%s}" % XHTML_NAMESPACE

>>> NSMAP = {None : XHTML_NAMESPACE} # the default namespace (no prefix)

>>> xhtml = etree.Element(XHTML + "html", nsmap=NSMAP) # lxml only!
>>> body = etree.SubElement(xhtml, XHTML + "body")
>>> body.text = "Hello World"

>>> print(etree.tostring(xhtml, pretty_print=True))
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>Hello World</body>
</html>
```

You can also use the `QName` helper class to build or split qualified tag names:

```
>>> tag = etree.QName('http://www.w3.org/1999/xhtml', 'html')
>>> print(tag.localname)
html
>>> print(tag.namespace)
http://www.w3.org/1999/xhtml
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}html

>>> tag = etree.QName('{http://www.w3.org/1999/xhtml}html')
>>> print(tag.localname)
html
>>> print(tag.namespace)
http://www.w3.org/1999/xhtml

>>> root = etree.Element('{http://www.w3.org/1999/xhtml}html')
>>> tag = etree.QName(root)
>>> print(tag.localname)
html

>>> tag = etree.QName(root, 'script')
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}script
>>> tag = etree.QName('{http://www.w3.org/1999/xhtml}html', 'script')
>>> print(tag.text)
{http://www.w3.org/1999/xhtml}script
```

lxml.etree allows you to look up the current namespaces defined for a node through the `.nsmap` property:

```
>>> xhtml.nsmap
{None: 'http://www.w3.org/1999/xhtml'}
```

Note, however, that this includes all prefixes known in the context of an Element, not only those that it defines itself.

```
>>> root = etree.Element('root', nsmap={'a': 'http://a.b/c'})
>>> child = etree.SubElement(root, 'child',
...                          nsmap={'b': 'http://b.c/d'})
>>> len(root.nsmap)
1
>>> len(child.nsmap)
2
>>> child.nsmap['a']
'http://a.b/c'
>>> child.nsmap['b']
'http://b.c/d'
```

Therefore, modifying the returned dict cannot have any meaningful impact on the Element. Any changes to it are ignored.

Namespaces on attributes work alike, but as of version 2.3, `lxml.etree` will ensure that the attribute uses a prefixed namespace declaration. This is because unprefixed attribute names are not considered being in a namespace by the XML namespace specification (**section 6.2**), so they may end up losing their namespace on a serialise-parse roundtrip, even if they appear in a namespaced element.

```
>>> body.set(XHTML + "bgcolor", "#CCFFAA")

>>> print(etree.tostring(xhtml, pretty_print=True))
<html xmlns="http://www.w3.org/1999/xhtml">
  <body xmlns:html="http://www.w3.org/1999/xhtml" html:bgcolor="#CCFFAA">Hello World</body>
</html>

>>> print(body.get("bgcolor"))
None
>>> body.get(XHTML + "bgcolor")
'#CCFFAA'
```

You can also use XPath with fully qualified names:

```
>>> find_xhtml_body = etree.ETXPath(      # lxml only !
...     "//{%s}body" % XHTML_NAMESPACE)
>>> results = find_xhtml_body(xhtml)

>>> print(results[0].tag)
{http://www.w3.org/1999/xhtml}body
```

For convenience, you can use `"*"` wildcards in all iterators of `lxml.etree`, both for tag names and namespaces:

```
>>> for el in xhtml.iter('*'): print(el.tag)   # any element
{http://www.w3.org/1999/xhtml}html
{http://www.w3.org/1999/xhtml}body
>>> for el in xhtml.iter('{http://www.w3.org/1999/xhtml}*'): print(el.tag)
{http://www.w3.org/1999/xhtml}html
{http://www.w3.org/1999/xhtml}body
>>> for el in xhtml.iter('{*}body'): print(el.tag)
{http://www.w3.org/1999/xhtml}body
```

To look for elements that do not have a namespace, either use the plain tag name or provide the empty namespace explicitly:

```
>>> [ el.tag for el in xhtml.iter('{http://www.w3.org/1999/xhtml}body') ]
['{http://www.w3.org/1999/xhtml}body']
>>> [ el.tag for el in xhtml.iter('body') ]
[]
>>> [ el.tag for el in xhtml.iter('{}body') ]
[]
>>> [ el.tag for el in xhtml.iter('{}*') ]
[]
```

The E-factory

The `E-factory` provides a simple and compact syntax for generating XML and HTML:

```
>>> from lxml.builder import E

>>> def CLASS(*args): # class is a reserved word in Python
...     return {"class":' '.join(args)}

>>> html = page = (
...   E.html(       # create an Element called "html"
...     E.head(
...       E.title("This is a sample document")
...     ),
...     E.body(
...       E.h1("Hello!", CLASS("title")),
...       E.p("This is a paragraph with ", E.b("bold"), " text in it!"),
...       E.p("This is another paragraph, with a", "\n      ",
...         E.a("link", href="http://www.python.org"), "."),
...       E.p("Here are some reservered characters: <spam&egg>."),
```

```
...         etree.XML("<p>And finally an embedded XHTML fragment.</p>"),
...       )
...     )
... )

>>> print(etree.tostring(page, pretty_print=True))
<html>
  <head>
    <title>This is a sample document</title>
  </head>
  <body>
    <h1 class="title">Hello!</h1>
    <p>This is a paragraph with <b>bold</b> text in it!</p>
    <p>This is another paragraph, with a
      <a href="http://www.python.org">link</a>.</p>
    <p>Here are some reservered characters: &lt;spam&amp;egg&gt;.</p>
    <p>And finally an embedded XHTML fragment.</p>
  </body>
</html>
```

Element creation based on attribute access makes it easy to build up a simple vocabulary for an XML language:

```
>>> from lxml.builder import ElementMaker # lxml only !

>>> E = ElementMaker(namespace="http://my.de/fault/namespace",
...                  nsmap={'p' : "http://my.de/fault/namespace"})

>>> DOC = E.doc
>>> TITLE = E.title
>>> SECTION = E.section
>>> PAR = E.par

>>> my_doc = DOC(
...   TITLE("The dog and the hog"),
...   SECTION(
...     TITLE("The dog"),
...     PAR("Once upon a time, ..."),
...     PAR("And then ...")
...   ),
...   SECTION(
...     TITLE("The hog"),
...     PAR("Sooner or later ...")
...   )
... )

>>> print(etree.tostring(my_doc, pretty_print=True))
<p:doc xmlns:p="http://my.de/fault/namespace">
  <p:title>The dog and the hog</p:title>
  <p:section>
    <p:title>The dog</p:title>
    <p:par>Once upon a time, ...</p:par>
    <p:par>And then ...</p:par>
  </p:section>
  <p:section>
    <p:title>The hog</p:title>
    <p:par>Sooner or later ...</p:par>
  </p:section>
</p:doc>
```

One such example is the module `lxml.html.builder`, which provides a vocabulary for HTML.

When dealing with multiple namespaces, it is good practice to define one ElementMaker for each namespace URI. Again, note how the above example predefines the tag builders in named constants. That makes it easy to put all tag declarations of a namespace into one Python module and to import/use the tag name constants from there. This avoids pitfalls like typos or accidentally missing namespaces.

ElementPath

The ElementTree library comes with a simple XPath-like path language called **ElementPath**. The main difference is that you can use the `{namespace}tag` notation in ElementPath expressions. However, advanced features like value comparison and functions are not available.

In addition to a **full XPath implementation**, `lxml.etree` supports the ElementPath language in the same way ElementTree does, even using (almost) the same implementation. The API provides four methods here that you can find on Elements and ElementTrees:

- `iterfind()` iterates over all Elements that match the path expression
- `findall()` returns a list of matching Elements
- `find()` efficiently returns only the first match
- `findtext()` returns the `.text` content of the first match

Here are some examples:

```
>>> root = etree.XML("<root><a x='123'>aText<b/><c/><b/></a></root>")
```

Find a child of an Element:

```
>>> print(root.find("b"))
None
>>> print(root.find("a").tag)
a
```

Find an Element anywhere in the tree:

```
>>> print(root.find(".//b").tag)
b
>>> [ b.tag for b in root.iterfind(".//b") ]
['b', 'b']
```

Find Elements with a certain attribute:

```
>>> print(root.findall(".//a[@x]")[0].tag)
a
>>> print(root.findall(".//a[@y]"))
[]
```

In lxml 3.4, there is a new helper to generate a structural ElementPath expression for an Element:

```
>>> tree = etree.ElementTree(root)
>>> a = root[0]
>>> print(tree.getelementpath(a[0]))
a/b[1]
>>> print(tree.getelementpath(a[1]))
a/c
>>> print(tree.getelementpath(a[2]))
a/b[2]
>>> tree.find(tree.getelementpath(a[2])) == a[2]
True
```

As long as the tree is not modified, this path expression represents an identifier for a given element that can be used to `find()` it in the same tree later. Compared to XPath, ElementPath expressions have the advantage of being self-contained even for documents that use namespaces.

The `.iter()` method is a special case that only finds specific tags in the tree by their name, not based on a path. That means that the following commands are equivalent in the success case:

```
>>> print(root.find(".//b").tag)
b
>>> print(next(root.iterfind(".//b")).tag)
b
>>> print(next(root.iter("b")).tag)
b
```

Note that the `.find()` method simply returns None if no match is found, whereas the other two examples would raise a `StopIteration` exception.

Generated on: 2016-08-20.

Beautiful Soup Documentation

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.

These instructions illustrate all major features of Beautiful Soup 4, with examples. I show you what the library is good for, how it works, how to use it, how to make it do what you want, and what to do when it violates your expectations.

The examples in this documentation should work the same way in Python 2.7 and Python 3.2.

You might be looking for the documentation for Beautiful Soup 3. If so, you should know that Beautiful Soup 3 is no longer being developed, and that Beautiful Soup 4 is recommended for all new projects. If you want to learn about the differences between Beautiful Soup 3 and Beautiful Soup 4, see Porting code to BS4.

This documentation has been translated into other languages by Beautiful Soup users:

- 这篇文档当然还有中文版.
- このページは日本語で利用できます(外部リンク)
- 이 문서는 한국어 번역도 가능합니다. (외부 링크)

Getting help

If you have questions about Beautiful Soup, or run into problems, send mail to the discussion group. If your problem involves parsing an HTML document, be sure to mention *what the diagnose() function says* about that document.

Quick Start

Here's an HTML document I'll be using as an example throughout this document. It's part of a story from *Alice in Wonderland*:

```
html_doc = """
```

```
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names w
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
```

Running the “three sisters” document through Beautiful Soup gives us a `BeautifulSoup` object, which represents the document as a nested data structure:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

print(soup.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#   <p class="title">
#    <b>
#     The Dormouse's story
#    </b>
#   </p>
#   <p class="story">
#    Once upon a time there were three little sisters; and their names were
#    <a class="sister" href="http://example.com/elsie" id="link1">
#     Elsie
#    </a>
#    ,
#    <a class="sister" href="http://example.com/lacie" id="link2">
#     Lacie
#    </a>
#    and
#    <a class="sister" href="http://example.com/tillie" id="link2">
#     Tillie
#    </a>
#    ; and they lived at the bottom of a well.
#   </p>
#   <p class="story">
#    ...
#   </p>
#  </body>
# </html>
```

Here are some simple ways to navigate that data structure:

```
soup.title
# <title>The Dormouse's story</title>

soup.title.name
# u'title'
```

```
soup.title.string
# u'The Dormouse's story'

soup.title.parent.name
# u'head'

soup.p
# <p class="title"><b>The Dormouse's story</b></p>

soup.p['class']
# u'title'

soup.a
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.find(id="link3")
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

One common task is extracting all the URLs found within a page’s <a> tags:

```
for link in soup.find_all('a'):
    print(link.get('href'))
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

Another common task is extracting all the text from a page:

```
print(soup.get_text())
# The Dormouse's story
#
# The Dormouse's story
#
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
#
# ...
```

Does this look like what you need? If so, read on.

Installing Beautiful Soup

If you’re using a recent version of Debian or Ubuntu Linux, you can install Beautiful Soup with the system package manager:

```
$ apt-get install python-bs4
```

Beautiful Soup 4 is published through PyPi, so if you can't install it with the system packager, you can install it with `easy_install` or `pip`. The package name is `beautifulsoup4`, and the same package works on Python 2 and Python 3.

```
$ easy_install beautifulsoup4
```

```
$ pip install beautifulsoup4
```

(The `BeautifulSoup` package is probably *not* what you want. That's the previous major release, Beautiful Soup 3. Lots of software uses BS3, so it's still available, but if you're writing new code you should install `beautifulsoup4`.)

If you don't have `easy_install` or `pip` installed, you can download the Beautiful Soup 4 source tarball and install it with `setup.py`.

```
$ python setup.py install
```

If all else fails, the license for Beautiful Soup allows you to package the entire library with your application. You can download the tarball, copy its `bs4` directory into your application's codebase, and use Beautiful Soup without installing it at all.

I use Python 2.7 and Python 3.2 to develop Beautiful Soup, but it should work with other recent versions.

Problems after installation

Beautiful Soup is packaged as Python 2 code. When you install it for use with Python 3, it's automatically converted to Python 3 code. If you don't install the package, the code won't be converted. There have also been reports on Windows machines of the wrong version being installed.

If you get the `ImportError` "No module named HTMLParser", your problem is that you're running the Python 2 version of the code under Python 3.

If you get the `ImportError` "No module named html.parser", your problem is that you're running the Python 3 version of the code under Python 2.

In both cases, your best bet is to completely remove the Beautiful Soup installation from your system (including any directory created when you unzipped the tarball) and try the installation again.

If you get the `SyntaxError` "Invalid syntax" on the line `ROOT_TAG_NAME = u'[document]'`, you need to convert the Python 2 code to Python 3. You can do this either by installing the package:

```
$ python3 setup.py install
```

or by manually running Python's `2to3` conversion script on the `bs4` directory:

```
$ 2to3-3.2 -w bs4
```

Installing a parser

Beautiful Soup supports the HTML parser included in Python's standard library, but it also supports a number of third-party Python parsers. One is the lxml parser. Depending on your setup, you might install lxml with one of these commands:

```
$ apt-get install python-lxml
```

```
$ easy_install lxml
```

```
$ pip install lxml
```

Another alternative is the pure-Python html5lib parser, which parses HTML the way a web browser does. Depending on your setup, you might install html5lib with one of these commands:

```
$ apt-get install python-html5lib
```

```
$ easy_install html5lib
```

```
$ pip install html5lib
```

This table summarizes the advantages and disadvantages of each parser library:

Parser	Typical usage	Advantages	Disadvantages
Python's html.parser	`BeautifulSoup(markup, "html.parser")`	• Batteries included • Decent speed • Lenient (as of Python 2.7.3 and 3.2.)	• Not very lenient (before Python 2.7.3 or 3.2.2)
lxml's HTML parser	`BeautifulSoup(markup, "lxml")`	• Very fast • Lenient	• External C dependency

lxml's XML parser	`BeautifulSoup(markup, "lxml-xml")` `BeautifulSoup(markup, "xml")`	• Very fast • The only currently supported XML parser	• External C dependency
html5lib	`BeautifulSoup(markup, "html5lib")`	• Extremely lenient • Parses pages the same way a web browser does • Creates valid HTML5	• Very slow • External Python dependency

If you can, I recommend you install and use lxml for speed. If you're using a version of Python 2 earlier than 2.7.3, or a version of Python 3 earlier than 3.2.2, it's *essential* that you install lxml or html5lib–Python's built-in HTML parser is just not very good in older versions.

Note that if a document is invalid, different parsers will generate different Beautiful Soup trees for it. See Differences between parsers for details.

Making the soup

To parse a document, pass it into the `BeautifulSoup` constructor. You can pass in a string or an open filehandle:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(open("index.html"))

soup = BeautifulSoup("<html>data</html>")
```

First, the document is converted to Unicode, and HTML entities are converted to Unicode characters:

```
BeautifulSoup("Sacr&eacute; bleu!")
<html><head></head><body>Sacré bleu!</body></html>
```

Beautiful Soup then parses the document using the best available parser. It will use an HTML parser unless you specifically tell it to use an XML parser. (See Parsing XML.)

Kinds of objects

Beautiful Soup transforms a complex HTML document into a complex tree of Python objects. But you'll only ever have to deal with about four *kinds* of objects: `Tag`, `NavigableString`, `BeautifulSoup`, and `Comment`.

Tag

A `Tag` object corresponds to an XML or HTML tag in the original document:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Tags have a lot of attributes and methods, and I'll cover most of them in Navigating the tree and Searching the tree. For now, the most important features of a tag are its name and attributes.

Name

Every tag has a name, accessible as `.name`:

```
tag.name
# u'b'
```

If you change a tag's name, the change will be reflected in any HTML markup generated by Beautiful Soup:

```
tag.name = "blockquote"
tag
# <blockquote class="boldest">Extremely bold</blockquote>
```

Attributes

A tag may have any number of attributes. The tag `<b class="boldest">` has an attribute "class" whose value is "boldest". You can access a tag's attributes by treating the tag like a dictionary:

```
tag['class']
# u'boldest'
```

You can access that dictionary directly as `.attrs`:

```
tag.attrs
# {u'class': u'boldest'}
```

You can add, remove, and modify a tag’s attributes. Again, this is done by treating the tag as a dictionary:

```
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>

tag['class']
# KeyError: 'class'
print(tag.get('class'))
# None
```

Multi-valued attributes

HTML 4 defines a few attributes that can have multiple values. HTML 5 removes a couple of them, but defines a few more. The most common multi-valued attribute is `class` (that is, a tag can have more than one CSS class). Others include `rel`, `rev`, `accept-charset`, `headers`, and `accesskey`. Beautiful Soup presents the value(s) of a multi-valued attribute as a list:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ["body", "strikeout"]

css_soup = BeautifulSoup('<p class="body"></p>')
css_soup.p['class']
# ["body"]
```

If an attribute *looks* like it has more than one value, but it’s not a multi-valued attribute as defined by any version of the HTML standard, Beautiful Soup will leave the attribute alone:

```
id_soup = BeautifulSoup('<p id="my id"></p>')
id_soup.p['id']
# 'my id'
```

When you turn a tag back into a string, multiple attribute values are consolidated:

```
rel_soup = BeautifulSoup('<p>Back to the <a rel="index">homepage</a></p>')
rel_soup.a['rel']
# ['index']
rel_soup.a['rel'] = ['index', 'contents']
print(rel_soup.p)
# <p>Back to the <a rel="index contents">homepage</a></p>
```

If you parse a document as XML, there are no multi-valued attributes:

```
xml_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml')
xml_soup.p['class']
# u'body strikeout'
```

NavigableString

A string corresponds to a bit of text within a tag. Beautiful Soup uses the `NavigableString` class to contain these bits of text:

```
tag.string
# u'Extremely bold'
type(tag.string)
# <class 'bs4.element.NavigableString'>
```

A `NavigableString` is just like a Python Unicode string, except that it also supports some of the features described in Navigating the tree and Searching the tree. You can convert a `NavigableString` to a Unicode string with `unicode()`:

```
unicode_string = unicode(tag.string)
unicode_string
# u'Extremely bold'
type(unicode_string)
# <type 'unicode'>
```

You can't edit a string in place, but you can replace one string with another, using *replace_with()*:

```
tag.string.replace_with("No longer bold")
tag
# <blockquote>No longer bold</blockquote>
```

`NavigableString` supports most of the features described in Navigating the tree and Searching the tree, but not all of them. In particular, since a string can't contain anything (the way a tag may contain a string or another tag), strings don't support the `.contents` or `.string` attributes, or the `find()` method.

If you want to use a `NavigableString` outside of Beautiful Soup, you should call `unicode()` on it to turn it into a normal Python Unicode string. If you don't, your string will carry around a reference to the entire Beautiful Soup parse tree, even when you're done using Beautiful Soup. This is a big waste of memory.

BeautifulSoup

The `BeautifulSoup` object itself represents the document as a whole. For most purposes, you can treat it as a *Tag* object. This means it supports most of the methods described in Navigating the tree and Searching the tree.

Since the `BeautifulSoup` object doesn't correspond to an actual HTML or XML tag, it has no name and no attributes. But sometimes it's useful to look at its `.name`, so it's been given the special `.name` "[document]":

```
soup.name
# u'[document]'
```

Comments and other special strings

`Tag`, `NavigableString`, and `BeautifulSoup` cover almost everything you'll see in an HTML or XML file, but there are a few leftover bits. The only one you'll probably ever need to worry about is the comment:

```
markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
soup = BeautifulSoup(markup)
comment = soup.b.string
type(comment)
# <class 'bs4.element.Comment'>
```

The `Comment` object is just a special type of `NavigableString`:

```
comment
# u'Hey, buddy. Want to buy a used parser'
```

But when it appears as part of an HTML document, a `Comment` is displayed with special formatting:

```
print(soup.b.prettify())
# <b>
#  <!--Hey, buddy. Want to buy a used parser?-->
# </b>
```

Beautiful Soup defines classes for anything else that might show up in an XML document: `CData`, `ProcessingInstruction`, `Declaration`, and `Doctype`. Just like `Comment`, these classes are subclasses of `NavigableString` that add something extra to the string. Here's an example that replaces the comment with a CDATA block:

```
from bs4 import CData
cdata = CData("A CDATA block")
comment.replace_with(cdata)

print(soup.b.prettify())
# <b>
#  <![CDATA[A CDATA block]]>
# </b>
```

Navigating the tree

Here's the "Three sisters" HTML document again:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names w
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

I'll use this as an example to show you how to move from one part of a document to another.

Going down

Tags may contain strings and other tags. These elements are the tag's *children*. Beautiful Soup provides a lot of different attributes for navigating and iterating over a tag's children.

Note that Beautiful Soup strings don't support any of these attributes, because a string can't have children.

Navigating using tag names

The simplest way to navigate the parse tree is to say the name of the tag you want. If you want the <head> tag, just say `soup.head`:

```
soup.head
# <head><title>The Dormouse's story</title></head>

soup.title
# <title>The Dormouse's story</title>
```

You can do use this trick again and again to zoom in on a certain part of the parse tree. This code gets the first <b> tag beneath the <body> tag:

```
soup.body.b
# <b>The Dormouse's story</b>
```

Using a tag name as an attribute will give you only the *first* tag by that name:

```
soup.a
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

If you need to get *all* the <a> tags, or anything more complicated than the first tag with a certain name, you'll need to use one of the methods described in Searching the tree, such as *find_all()*:

```
soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

`.contents` and `.children`

A tag's children are available in a list called `.contents`:

```
head_tag = soup.head
head_tag
# <head><title>The Dormouse's story</title></head>

head_tag.contents
[<title>The Dormouse's story</title>]

title_tag = head_tag.contents[0]
title_tag
# <title>The Dormouse's story</title>
title_tag.contents
# [u'The Dormouse's story']
```

The `BeautifulSoup` object itself has children. In this case, the <html> tag is the child of the `BeautifulSoup` object.:

```
len(soup.contents)
# 1
soup.contents[0].name
# u'html'
```

A string does not have `.contents`, because it can't contain anything:

```
text = title_tag.contents[0]
text.contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

Instead of getting them as a list, you can iterate over a tag's children using the `.children` generator:

```
for child in title_tag.children:
    print(child)
# The Dormouse's story
```

.descendants

The `.contents` and `.children` attributes only consider a tag's *direct* children. For instance, the <head> tag has a single direct child–the <title> tag:

```
head_tag.contents
# [<title>The Dormouse's story</title>]
```

But the <title> tag itself has a child: the string "The Dormouse's story". There's a sense in which that string is also a child of the <head> tag. The `.descendants` attribute lets you iterate over *all* of a tag's children, recursively: its direct children, the children of its direct children, and so on:

```
for child in head_tag.descendants:
    print(child)
# <title>The Dormouse's story</title>
# The Dormouse's story
```

The <head> tag has only one child, but it has two descendants: the <title> tag and the <title> tag's child. The `BeautifulSoup` object only has one direct child (the <html> tag), but it has a whole lot of descendants:

```
len(list(soup.children))
# 1
len(list(soup.descendants))
# 25
```

.string

If a tag has only one child, and that child is a `NavigableString`, the child is made available as `.string`:

```
title_tag.string
# u'The Dormouse's story'
```

If a tag's only child is another tag, and *that* tag has a `.string`, then the parent tag is considered to have the same `.string` as its child:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

head_tag.string
# u'The Dormouse's story'
```

If a tag contains more than one thing, then it's not clear what `.string` should refer to, so `.string` is defined to be `None`:

```
print(soup.html.string)
# None
```

.strings and stripped_strings

If there's more than one thing inside a tag, you can still look at just the strings. Use the `.strings` generator:

```
for string in soup.strings:
    print(repr(string))
# u"The Dormouse's story"
# u'\n\n'
# u"The Dormouse's story"
# u'\n\n'
# u'Once upon a time there were three little sisters; and their names were\n'
# u'Elsie'
# u',\n'
# u'Lacie'
# u' and\n'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# u'...'
# u'\n'
```

These strings tend to have a lot of extra whitespace, which you can remove by using the `.stripped_strings` generator instead:

```
for string in soup.stripped_strings:
    print(repr(string))
# u"The Dormouse's story"
# u"The Dormouse's story"
# u'Once upon a time there were three little sisters; and their names were'
# u'Elsie'
# u','
# u'Lacie'
# u'and'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'...'
```

Here, strings consisting entirely of whitespace are ignored, and whitespace at the beginning and end of strings is removed.

Going up

Continuing the "family tree" analogy, every tag and every string has a *parent*: the tag that contains it.

.parent

You can access an element's parent with the `.parent` attribute. In the example "three sisters" document, the <head> tag is the parent of the <title> tag:

```
title_tag = soup.title
title_tag
# <title>The Dormouse's story</title>
title_tag.parent
# <head><title>The Dormouse's story</title></head>
```

The title string itself has a parent: the <title> tag that contains it:

```
title_tag.string.parent
# <title>The Dormouse's story</title>
```

The parent of a top-level tag like <html> is the `BeautifulSoup` object itself:

```
html_tag = soup.html
type(html_tag.parent)
# <class 'bs4.BeautifulSoup'>
```

And the `.parent` of a `BeautifulSoup` object is defined as None:

```
print(soup.parent)
# None
```

`.parents`

You can iterate over all of an element's parents with `.parents`. This example uses `.parents` to travel from an <a> tag buried deep within the document, to the very top of the document:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
for parent in link.parents:
    if parent is None:
        print(parent)
    else:
        print(parent.name)
# p
# body
# html
# [document]
# None
```

Going sideways

Consider a simple document like this:

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></b></a>")
print(sibling_soup.prettify())
# <html>
#  <body>
#   <a>
#    <b>
```

```
#     text1
#    </b>
#    <c>
#     text2
#    </c>
#   </a>
#  </body>
# </html>
```

The <b> tag and the <c> tag are at the same level: they're both direct children of the same tag. We call them *siblings*. When a document is pretty-printed, siblings show up at the same indentation level. You can also use this relationship in the code you write.

`.next_sibling` and `.previous_sibling`

You can use `.next_sibling` and `.previous_sibling` to navigate between page elements that are on the same level of the parse tree:

```
sibling_soup.b.next_sibling
# <c>text2</c>

sibling_soup.c.previous_sibling
# <b>text1</b>
```

The <b> tag has a `.next_sibling`, but no `.previous_sibling`, because there's nothing before the <b> tag *on the same level of the tree*. For the same reason, the <c> tag has a `.previous_sibling` but no `.next_sibling`:

```
print(sibling_soup.b.previous_sibling)
# None
print(sibling_soup.c.next_sibling)
# None
```

The strings "text1" and "text2" are *not* siblings, because they don't have the same parent:

```
sibling_soup.b.string
# u'text1'

print(sibling_soup.b.string.next_sibling)
# None
```

In real documents, the `.next_sibling` or `.previous_sibling` of a tag will usually be a string containing whitespace. Going back to the "three sisters" document:

```
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>
```

You might think that the `.next_sibling` of the first <a> tag would be the second <a> tag. But actually, it's a string: the comma and newline that separate the first <a> tag from the second:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

link.next_sibling
# u',\n'
```

The second <a> tag is actually the `.next_sibling` of the comma:

```
link.next_sibling.next_sibling
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
```

`.next_siblings` and `.previous_siblings`

You can iterate over a tag's siblings with `.next_siblings` or `.previous_siblings`:

```
for sibling in soup.a.next_siblings:
    print(repr(sibling))
# u',\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u' and\n'
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
# u'; and they lived at the bottom of a well.'
# None

for sibling in soup.find(id="link3").previous_siblings:
    print(repr(sibling))
# ' and\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u',\n'
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
# u'Once upon a time there were three little sisters; and their names were\n'
# None
```

Going back and forth

Take a look at the beginning of the "three sisters" document:

```
<html><head><title>The Dormouse's story</title></head>
<p class="title"><b>The Dormouse's story</b></p>
```

An HTML parser takes this string of characters and turns it into a series of events: "open an <html> tag", "open a <head> tag", "open a <title> tag", "add a string", "close the <title> tag", "open a <p> tag", and so on. Beautiful Soup offers tools for reconstructing the initial parse of the document.

`.next_element` and `.previous_element`

The `.next_element` attribute of a string or tag points to whatever was parsed immediately afterwards. It might be the same as `.next_sibling`, but it's usually drastically different.

Here's the final <a> tag in the "three sisters" document. Its `.next_sibling` is a string: the conclusion of the sentence that was interrupted by the start of the <a> tag.:

```
last_a_tag = soup.find("a", id="link3")
last_a_tag
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_a_tag.next_sibling
# '; and they lived at the bottom of a well.'
```

But the `.next_element` of that <a> tag, the thing that was parsed immediately after the <a> tag, is *not* the rest of that sentence: it's the word "Tillie":

```
last_a_tag.next_element
# u'Tillie'
```

That's because in the original markup, the word "Tillie" appeared before that semicolon. The parser encountered an <a> tag, then the word "Tillie", then the closing tag, then the semicolon and rest of the sentence. The semicolon is on the same level as the <a> tag, but the word "Tillie" was encountered first.

The `.previous_element` attribute is the exact opposite of `.next_element`. It points to whatever element was parsed immediately before this one:

```
last_a_tag.previous_element
# u' and\n'
last_a_tag.previous_element.next_element
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

`.next_elements` and `.previous_elements`

You should get the idea by now. You can use these iterators to move forward or backward in the document as it was parsed:

```
for element in last_a_tag.next_elements:
    print(repr(element))
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...</p>
```

```
# u'...'
# u'\n'
# None
```

Searching the tree

Beautiful Soup defines a lot of methods for searching the parse tree, but they're all very similar. I'm going to spend a lot of time explaining the two most popular methods: `find()` and `find_all()`. The other methods take almost exactly the same arguments, so I'll just cover them briefly.

Once again, I'll be using the "three sisters" document as an example:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names w
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

By passing in a filter to an argument like `find_all()`, you can zoom in on the parts of the document you're interested in.

Kinds of filters

Before talking in detail about `find_all()` and similar methods, I want to show examples of different filters you can pass into these methods. These filters show up again and again, throughout the search API. You can use them to filter based on a tag's name, on its attributes, on the text of a string, or on some combination of these.

A string

The simplest filter is a string. Pass a string to a search method and Beautiful Soup will perform a match against that exact string. This code finds all the <b> tags in the document:

```
soup.find_all('b')
# [<b>The Dormouse's story</b>]
```

If you pass in a byte string, Beautiful Soup will assume the string is encoded as UTF-8. You can avoid this by passing in a Unicode string instead.

A regular expression

If you pass in a regular expression object, Beautiful Soup will filter against that regular expression using its `match()` method. This code finds all the tags whose names start with the letter “b”; in this case, the <body> tag and the <b> tag:

```
import re
for tag in soup.find_all(re.compile("^b")):
    print(tag.name)
# body
# b
```

This code finds all the tags whose names contain the letter ‘t’:

```
for tag in soup.find_all(re.compile("t")):
    print(tag.name)
# html
# title
```

A list

If you pass in a list, Beautiful Soup will allow a string match against *any* item in that list. This code finds all the <a> tags *and* all the <b> tags:

```
soup.find_all(["a", "b"])
# [<b>The Dormouse's story</b>,
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

`True`

The value `True` matches everything it can. This code finds *all* the tags in the document, but none of the text strings:

```
for tag in soup.find_all(True):
    print(tag.name)
# html
# head
# title
# body
# p
# b
# p
# a
# a
```

```
# a
# p
```

A function

If none of the other matches work for you, define a function that takes an element as its only argument. The function should return `True` if the argument matches, and `False` otherwise.

Here's a function that returns `True` if a tag defines the "class" attribute but doesn't define the "id" attribute:

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

Pass this function into `find_all()` and you'll pick up all the <p> tags:

```
soup.find_all(has_class_but_no_id)
# [<p class="title"><b>The Dormouse's story</b></p>,
#  <p class="story">Once upon a time there were...</p>,
#  <p class="story">...</p>]
```

This function only picks up the <p> tags. It doesn't pick up the <a> tags, because those tags define both "class" and "id". It doesn't pick up tags like <html> and <title>, because those tags don't define "class".

If you pass in a function to filter on a specific attribute like `href`, the argument passed into the function will be the attribute value, not the whole tag. Here's a function that finds all `a` tags whose `href` attribute *does not* match a regular expression:

```
def not_lacie(href):
    return href and not re.compile("lacie").search(href)
soup.find_all(href=not_lacie)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

The function can be as complicated as you need it to be. Here's a function that returns `True` if a tag is surrounded by string objects:

```
from bs4 import NavigableString
def surrounded_by_strings(tag):
    return (isinstance(tag.next_element, NavigableString)
            and isinstance(tag.previous_element, NavigableString))

for tag in soup.find_all(surrounded_by_strings):
    print tag.name
# p
# a
# a
# a
```

```
# p
```

Now we’re ready to look at the search methods in detail.

`find_all()`

Signature: find_all(*name*, *attrs*, *recursive*, *string*, *limit*, ***kwargs*)

The `find_all()` method looks through a tag’s descendants and retrieves *all* descendants that match your filters. I gave several examples in Kinds of filters, but here are a few more:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]

soup.find_all("p", "title")
# [<p class="title"><b>The Dormouse's story</b></p>]

soup.find_all("a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.find_all(id="link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

import re
soup.find(string=re.compile("sisters"))
# u'Once upon a time there were three little sisters; and their names were\n'
```

Some of these should look familiar, but others are new. What does it mean to pass in a value for `string`, or `id`? Why does `find_all("p", "title")` find a <p> tag with the CSS class “title”? Let’s look at the arguments to `find_all()`.

The `name` argument

Pass in a value for `name` and you’ll tell Beautiful Soup to only consider tags with certain names. Text strings will be ignored, as will tags whose names that don’t match.

This is the simplest usage:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]
```

Recall from Kinds of filters that the value to `name` can be a string, a regular expression, a list, a function, or the value True.

The keyword arguments

Any argument that's not recognized will be turned into a filter on one of a tag's attributes. If you pass in a value for an argument called `id`, Beautiful Soup will filter against each tag's 'id' attribute:

```
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

If you pass in a value for `href`, Beautiful Soup will filter against each tag's 'href' attribute:

```
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

You can filter an attribute based on a string, a regular expression, a list, a function, or the value True.

This code finds all tags whose `id` attribute has a value, regardless of what the value is:

```
soup.find_all(id=True)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

You can filter multiple attributes at once by passing in more than one keyword argument:

```
soup.find_all(href=re.compile("elsie"), id='link1')
# [<a class="sister" href="http://example.com/elsie" id="link1">three</a>]
```

Some attributes, like the data-* attributes in HTML 5, have names that can't be used as the names of keyword arguments:

```
data_soup = BeautifulSoup('<div data-foo="value">foo!</div>')
data_soup.find_all(data-foo="value")
# SyntaxError: keyword can't be an expression
```

You can use these attributes in searches by putting them into a dictionary and passing the dictionary into `find_all()` as the `attrs` argument:

```
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

Searching by CSS class

It's very useful to search for a tag that has a certain CSS class, but the name of the CSS attribute, "class", is a reserved word in Python. Using `class` as a keyword argument will give you a syntax error. As of Beautiful

Soup 4.1.2, you can search by CSS class using the keyword argument `class_`:

```
soup.find_all("a", class_="sister")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

As with any keyword argument, you can pass `class_` a string, a regular expression, a function, or `True`:

```
soup.find_all(class_=re.compile("itl"))
# [<p class="title"><b>The Dormouse's story</b></p>]

def has_six_characters(css_class):
    return css_class is not None and len(css_class) == 6

soup.find_all(class_=has_six_characters)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Remember that a single tag can have multiple values for its “class” attribute. When you search for a tag that matches a certain CSS class, you’re matching against *any* of its CSS classes:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.find_all("p", class_="strikeout")
# [<p class="body strikeout"></p>]

css_soup.find_all("p", class_="body")
# [<p class="body strikeout"></p>]
```

You can also search for the exact string value of the `class` attribute:

```
css_soup.find_all("p", class_="body strikeout")
# [<p class="body strikeout"></p>]
```

But searching for variants of the string value won’t work:

```
css_soup.find_all("p", class_="strikeout body")
# []
```

If you want to search for tags that match two or more CSS classes, you should use a CSS selector:

```
css_soup.select("p.strikeout.body")
# [<p class="body strikeout"></p>]
```

In older versions of Beautiful Soup, which don’t have the `class_` shortcut, you can use the `attrs` trick mentioned above. Create a dictionary whose value for “class” is the string (or regular expression, or whatever) you want to search for:

```
soup.find_all("a", attrs={"class": "sister"})
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

The `string` argument

With `string` you can search for strings instead of tags. As with `name` and the keyword arguments, you can pass in a string, a regular expression, a list, a function, or the value True. Here are some examples:

```
soup.find_all(string="Elsie")
# [u'Elsie']

soup.find_all(string=["Tillie", "Elsie", "Lacie"])
# [u'Elsie', u'Lacie', u'Tillie']

soup.find_all(string=re.compile("Dormouse"))
[u"The Dormouse's story", u"The Dormouse's story"]

def is_the_only_string_within_a_tag(s):
    """Return True if this string is the only child of its parent tag."""
    return (s == s.parent.string)

soup.find_all(string=is_the_only_string_within_a_tag)
# [u"The Dormouse's story", u"The Dormouse's story", u'Elsie', u'Lacie', u'Tillie',
```

Although `string` is for finding strings, you can combine it with arguments that find tags: Beautiful Soup will find all tags whose `.string` matches your value for `string`. This code finds the <a> tags whose `.string` is “Elsie”:

```
soup.find_all("a", string="Elsie")
# [<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>]
```

The `string` argument is new in Beautiful Soup 4.4.0. In earlier versions it was called `text`:

```
soup.find_all("a", text="Elsie")
# [<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>]
```

The `limit` argument

`find_all()` returns all the tags and strings that match your filters. This can take a while if the document is large. If you don’t need *all* the results, you can pass in a number for `limit`. This works just like the LIMIT keyword in SQL. It tells Beautiful Soup to stop gathering results after it’s found a certain number.

There are three links in the “three sisters” document, but this code only finds the first two:

```
soup.find_all("a", limit=2)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

The `recursive` argument

If you call `mytag.find_all()`, Beautiful Soup will examine all the descendants of `mytag`: its children, its children’s children, and so on. If you only want Beautiful Soup to consider direct children, you can pass in `recursive=False`. See the difference here:

```
soup.html.find_all("title")
# [<title>The Dormouse's story</title>]

soup.html.find_all("title", recursive=False)
# []
```

Here’s that part of the document:

```
<html>
 <head>
  <title>
   The Dormouse's story
  </title>
 </head>
...
```

The <title> tag is beneath the <html> tag, but it’s not *directly* beneath the <html> tag: the <head> tag is in the way. Beautiful Soup finds the <title> tag when it’s allowed to look at all descendants of the <html> tag, but when `recursive=False` restricts it to the <html> tag’s immediate children, it finds nothing.

Beautiful Soup offers a lot of tree-searching methods (covered below), and they mostly take the same arguments as `find_all()`: `name`, `attrs`, `string`, `limit`, and the keyword arguments. But the `recursive` argument is different: `find_all()` and `find()` are the only methods that support it. Passing `recursive=False` into a method like `find_parents()` wouldn’t be very useful.

Calling a tag is like calling `find_all()`

Because `find_all()` is the most popular method in the Beautiful Soup search API, you can use a shortcut for it. If you treat the `BeautifulSoup` object or a `Tag` object as though it were a function, then it’s the same as calling `find_all()` on that object. These two lines of code are equivalent:

```
soup.find_all("a")
soup("a")
```

These two lines are also equivalent:

```
soup.title.find_all(string=True)
soup.title(string=True)
```

`find()`

Signature: find(*name*, *attrs*, *recursive*, *string*, ***kwargs*)

The `find_all()` method scans the entire document looking for results, but sometimes you only want to find one result. If you know a document only has one <body> tag, it's a waste of time to scan the entire document looking for more. Rather than passing in `limit=1` every time you call `find_all`, you can use the `find()` method. These two lines of code are *nearly* equivalent:

```
soup.find_all('title', limit=1)
# [<title>The Dormouse's story</title>]

soup.find('title')
# <title>The Dormouse's story</title>
```

The only difference is that `find_all()` returns a list containing the single result, and `find()` just returns the result.

If `find_all()` can't find anything, it returns an empty list. If `find()` can't find anything, it returns `None`:

```
print(soup.find("nosuchtag"))
# None
```

Remember the `soup.head.title` trick from Navigating using tag names? That trick works by repeatedly calling `find()`:

```
soup.head.title
# <title>The Dormouse's story</title>

soup.find("head").find("title")
# <title>The Dormouse's story</title>
```

`find_parents()` and `find_parent()`

Signature: find_parents(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_parent(*name*, *attrs*, *string*, ***kwargs*)

I spent a lot of time above covering `find_all()` and `find()`. The Beautiful Soup API defines ten other methods for searching the tree, but don't be afraid. Five of these methods are basically the same as `find_all()`, and the other five are basically the same as `find()`. The only differences are in what parts of the tree they search.

First let's consider `find_parents()` and `find_parent()`. Remember that `find_all()` and `find()` work their way down the tree, looking at tag's descendants. These methods do the opposite: they work their way *up* the tree, looking at a tag's (or a string's) parents. Let's try them out, starting from a string buried deep in the "three daughters" document:

```
a_string = soup.find(string="Lacie")
a_string
# u'Lacie'

a_string.find_parents("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

a_string.find_parent("p")
# <p class="story">Once upon a time there were three little sisters; and their names
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
#  and they lived at the bottom of a well.</p>

a_string.find_parents("p", class="title")
# []
```

One of the three <a> tags is the direct parent of the string in question, so our search finds it. One of the three <p> tags is an indirect parent of the string, and our search finds that as well. There's a <p> tag with the CSS class "title" *somewhere* in the document, but it's not one of this string's parents, so we can't find it with `find_parents()`.

You may have made the connection between `find_parent()` and `find_parents()`, and the .parent and .parents attributes mentioned earlier. The connection is very strong. These search methods actually use `.parents` to iterate over all the parents, and check each one against the provided filter to see if it matches.

`find_next_siblings()` and `find_next_sibling()`

Signature: find_next_siblings(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_next_sibling(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.next_siblings* to iterate over the rest of an element's siblings in the tree. The `find_next_siblings()` method returns all the siblings that match, and `find_next_sibling()` only returns the first one:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_next_siblings("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_next_sibling("p")
# <p class="story">...</p>
```

`find_previous_siblings()` and `find_previous_sibling()`

Signature: find_previous_siblings(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_previous_sibling(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.previous_siblings* to iterate over an element's siblings that precede it in the tree. The `find_previous_siblings()` method returns all the siblings that match, and `find_previous_sibling()` only returns the first one:

```
last_link = soup.find("a", id="link3")
last_link
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_link.find_previous_siblings("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_previous_sibling("p")
# <p class="title"><b>The Dormouse's story</b></p>
```

`find_all_next()` and `find_next()`

Signature: find_all_next(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_next(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.next_elements* to iterate over whatever tags and strings that come after it in the document. The `find_all_next()` method returns all matches, and `find_next()` only returns the first match:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_next(string=True)
# [u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
```

```
#  u';\nand they lived at the bottom of a well.', u'\n\n', u'...', u'\n']

first_link.find_next("p")
# <p class="story">...</p>
```

In the first example, the string “Elsie” showed up, even though it was contained within the <a> tag we started from. In the second example, the last <p> tag in the document showed up, even though it’s not in the same part of the tree as the <a> tag we started from. For these methods, all that matters is that an element match the filter, and show up later in the document than the starting element.

`find_all_previous()` and `find_previous()`

Signature: find_all_previous(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_previous(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.previous_elements* to iterate over the tags and strings that came before it in the document. The `find_all_previous()` method returns all matches, and `find_previous()` only returns the first match:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_previous("p")
# [<p class="story">Once upon a time there were three little sisters; ...</p>,
#  <p class="title"><b>The Dormouse's story</b></p>]

first_link.find_previous("title")
# <title>The Dormouse's story</title>
```

The call to `find_all_previous("p")` found the first paragraph in the document (the one with class=”title”), but it also finds the second paragraph, the <p> tag that contains the <a> tag we started with. This shouldn’t be too surprising: we’re looking at all the tags that show up earlier in the document than the one we started with. A <p> tag that contains an <a> tag must have shown up before the <a> tag it contains.

CSS selectors

Beautiful Soup supports the most commonly-used CSS selectors. Just pass a string into the `.select()` method of a `Tag` object or the `BeautifulSoup` object itself.

You can find tags:

```
soup.select("title")
```

```
# [<title>The Dormouse's story</title>]

soup.select("p nth-of-type(3)")
# [<p class="story">...</p>]
```

Find tags beneath other tags:

```
soup.select("body a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"  id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("html head title")
# [<title>The Dormouse's story</title>]
```

Find tags *directly* beneath other tags:

```
soup.select("head > title")
# [<title>The Dormouse's story</title>]

soup.select("p > a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"  id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("p > a:nth-of-type(2)")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

soup.select("p > #link1")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select("body > a")
# []
```

Find the siblings of tags:

```
soup.select("#link1 ~ .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"  id="link3">Tillie</a>]

soup.select("#link1 + .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

Find tags by CSS class:

```
soup.select(".sister")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("[class~=sister]")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Find tags by ID:

```
soup.select("#link1")
```

```
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select("a#link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

Find tags that match any selector from a list of selectors:

> soup.select(“#link1,#link2”) # [Elsie, # Lacie]

Test for the existence of an attribute:

```
soup.select('a[href]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Find tags by attribute value:

```
soup.select('a[href="http://example.com/elsie"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select('a[href^="http://example.com/"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select('a[href$="tillie"]')
# [<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select('a[href*=".com/el"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

Match language codes:

```
multilingual_markup = """
 <p lang="en">Hello</p>
 <p lang="en-us">Howdy, y'all</p>
 <p lang="en-gb">Pip-pip, old fruit</p>
 <p lang="fr">Bonjour mes amis</p>
"""
multilingual_soup = BeautifulSoup(multilingual_markup)
multilingual_soup.select('p[lang|=en]')
# [<p lang="en">Hello</p>,
#  <p lang="en-us">Howdy, y'all</p>,
#  <p lang="en-gb">Pip-pip, old fruit</p>]
```

Find only the first tag that matches a selector:

```
soup.select_one(".sister")
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

This is all a convenience for users who know the CSS selector syntax. You can do all this stuff with the Beautiful Soup API. And if CSS selectors are all you need, you might as well use lxml directly: it's a lot faster, and it supports more CSS selectors. But this lets you *combine* simple CSS selectors with the Beautiful Soup API.

Modifying the tree

Beautiful Soup's main strength is in searching the parse tree, but you can also modify the tree and write your changes as a new HTML or XML document.

Changing tag names and attributes

I covered this earlier, in Attributes, but it bears repeating. You can rename a tag, change the values of its attributes, add new attributes, and delete attributes:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b

tag.name = "blockquote"
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>
```

Modifying `.string`

If you set a tag's `.string` attribute, the tag's contents are replaced with the string you give:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)

tag = soup.a
tag.string = "New link text."
tag
# <a href="http://example.com/">New link text.</a>
```

Be careful: if the tag contained other tags, they and all their contents will be destroyed.

`append()`

You can add to a tag's contents with `Tag.append()`. It works just like calling `.append()` on a Python list:

```
soup = BeautifulSoup("<a>Foo</a>")
soup.a.append("Bar")

soup
# <html><head></head><body><a>FooBar</a></body></html>
soup.a.contents
# [u'Foo', u'Bar']
```

`NavigableString()` and `.new_tag()`

If you need to add a string to a document, no problem–you can pass a Python string in to `append()`, or you can call the `NavigableString` constructor:

```
soup = BeautifulSoup("<b></b>")
tag = soup.b
tag.append("Hello")
new_string = NavigableString(" there")
tag.append(new_string)
tag
# <b>Hello there.</b>
tag.contents
# [u'Hello', u' there']
```

If you want to create a comment or some other subclass of `NavigableString`, just call the constructor:

```
from bs4 import Comment
new_comment = Comment("Nice to see you.")
tag.append(new_comment)
tag
# <b>Hello there<!--Nice to see you.--></b>
tag.contents
# [u'Hello', u' there', u'Nice to see you.']
```

(This is a new feature in Beautiful Soup 4.4.0.)

What if you need to create a whole new tag? The best solution is to call the factory method `BeautifulSoup.new_tag()`:

```
soup = BeautifulSoup("<b></b>")
original_tag = soup.b

new_tag = soup.new_tag("a", href="http://www.example.com")
original_tag.append(new_tag)
original_tag
# <b><a href="http://www.example.com"></a></b>

new_tag.string = "Link text."
original_tag
# <b><a href="http://www.example.com">Link text.</a></b>
```

Only the first argument, the tag name, is required.

`insert()`

`Tag.insert()` is just like `Tag.append()`, except the new element doesn't necessarily go at the end of its parent's `.contents`. It'll be inserted at whatever numeric position you say. It works just like `.insert()` on a Python list:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.insert(1, "but did not endorse ")
tag
# <a href="http://example.com/">I linked to but did not endorse <i>example.com</i></
tag.contents
# [u'I linked to ', u'but did not endorse', <i>example.com</i>]
```

`insert_before()` and `insert_after()`

The `insert_before()` method inserts a tag or string immediately before something else in the parse tree:

```
soup = BeautifulSoup("<b>stop</b>")
tag = soup.new_tag("i")
tag.string = "Don't"
soup.b.string.insert_before(tag)
soup.b
# <b><i>Don't</i>stop</b>
```

The `insert_after()` method moves a tag or string so that it immediately follows something else in the parse tree:

```
soup.b.i.insert_after(soup.new_string(" ever "))
soup.b
# <b><i>Don't</i> ever stop</b>
soup.b.contents
# [<i>Don't</i>, u' ever ', u'stop']
```

`clear()`

`Tag.clear()` removes the contents of a tag:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.clear()
tag
# <a href="http://example.com/"></a>
```

extract()

`PageElement.extract()` removes a tag or string from the tree. It returns the tag or string that was extracted:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

i_tag = soup.i.extract()

a_tag
# <a href="http://example.com/">I linked to</a>

i_tag
# <i>example.com</i>

print(i_tag.parent)
None
```

At this point you effectively have two parse trees: one rooted at the `BeautifulSoup` object you used to parse the document, and one rooted at the tag that was extracted. You can go on to call `extract` on a child of the element you extracted:

```
my_string = i_tag.string.extract()
my_string
# u'example.com'

print(my_string.parent)
# None
i_tag
# <i></i>
```

decompose()

`Tag.decompose()` removes a tag from the tree, then *completely destroys it and its contents*:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

soup.i.decompose()

a_tag
# <a href="http://example.com/">I linked to</a>
```

replace_with()

`PageElement.replace_with()` removes a tag or string from the tree, and replaces it with the tag or string of your choice:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

new_tag = soup.new_tag("b")
new_tag.string = "example.net"
a_tag.i.replace_with(new_tag)

a_tag
# <a href="http://example.com/">I linked to <b>example.net</b></a>
```

`replace_with()` returns the tag or string that was replaced, so that you can examine it or add it back to another part of the tree.

wrap()

`PageElement.wrap()` wraps an element in the tag you specify. It returns the new wrapper:

```
soup = BeautifulSoup("<p>I wish I was bold.</p>")
soup.p.string.wrap(soup.new_tag("b"))
# <b>I wish I was bold.</b>

soup.p.wrap(soup.new_tag("div")
# <div><p><b>I wish I was bold.</b></p></div>
```

This method is new in Beautiful Soup 4.0.5.

unwrap()

`Tag.unwrap()` is the opposite of `wrap()`. It replaces a tag with whatever's inside that tag. It's good for stripping out markup:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

a_tag.i.unwrap()
a_tag
# <a href="http://example.com/">I linked to example.com</a>
```

Like `replace_with()`, `unwrap()` returns the tag that was replaced.

Output

Pretty-printing

The `prettify()` method will turn a Beautiful Soup parse tree into a nicely formatted Unicode string, with each HTML/XML tag on its own line:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
soup.prettify()
# '<html>\n <head>\n </head>\n <body>\n  <a href="http://example.com/">\n...'

print(soup.prettify())
# <html>
#  <head>
#  </head>
#  <body>
#   <a href="http://example.com/">
#    I linked to
#    <i>
#     example.com
#    </i>
#   </a>
#  </body>
# </html>
```

You can call `prettify()` on the top-level `BeautifulSoup` object, or on any of its `Tag` objects:

```
print(soup.a.prettify())
# <a href="http://example.com/">
#  I linked to
#  <i>
#   example.com
#  </i>
# </a>
```

Non-pretty printing

If you just want a string, with no fancy formatting, you can call `unicode()` or `str()` on a `BeautifulSoup` object, or a `Tag` within it:

```
str(soup)
# '<html><head></head><body><a href="http://example.com/">I linked to <i>example.com

unicode(soup.a)
# u'<a href="http://example.com/">I linked to <i>example.com</i></a>'
```

The `str()` function returns a string encoded in UTF-8. See Encodings for other options.

You can also call `encode()` to get a bytestring, and `decode()` to get Unicode.

Output formatters

If you give Beautiful Soup a document that contains HTML entities like “&lquot;”, they’ll be converted to Unicode characters:

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
```

```
unicode(soup)
# u'<html><head></head><body>\u201cDammit!\u201d he said.</body></html>'
```

If you then convert the document to a string, the Unicode characters will be encoded as UTF-8. You won’t get the HTML entities back:

```
str(soup)
# '<html><head></head><body>\xe2\x80\x9cDammit!\xe2\x80\x9d he said.</body></html>'
```

By default, the only characters that are escaped upon output are bare ampersands and angle brackets. These get turned into “&”, “<”, and “>”, so that Beautiful Soup doesn’t inadvertently generate invalid HTML or XML:

```
soup = BeautifulSoup("<p>The law firm of Dewey, Cheatem, & Howe</p>")
soup.p
# <p>The law firm of Dewey, Cheatem, &amp; Howe</p>

soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a>')
soup.a
# <a href="http://example.com/?foo=val1&amp;bar=val2">A link</a>
```

You can change this behavior by providing a value for the `formatter` argument to `prettify()`, `encode()`, or `decode()`. Beautiful Soup recognizes four possible values for `formatter`.

The default is `formatter="minimal"`. Strings will only be processed enough to ensure that Beautiful Soup generates valid HTML/XML:

```
french = "<p>Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;</p>"
soup = BeautifulSoup(french)
print(soup.prettify(formatter="minimal"))
# <html>
#  <body>
#   <p>
#    Il a dit &lt;&lt;Sacré bleu!&gt;&gt;
#   </p>
#  </body>
# </html>
```

If you pass in `formatter="html"`, Beautiful Soup will convert Unicode characters to HTML entities whenever possible:

```
print(soup.prettify(formatter="html"))
# <html>
#  <body>
#   <p>
#    Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;
#   </p>
#  </body>
# </html>
```

If you pass in `formatter=None`, Beautiful Soup will not modify strings at all on output. This is the fastest option, but it may lead to Beautiful Soup generating invalid HTML/XML, as in these examples:

```
print(soup.prettify(formatter=None))
# <html>
#  <body>
#   <p>
#    Il a dit <<Sacré bleu!>>
#   </p>
#  </body>
# </html>

link_soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a
print(link_soup.a.encode(formatter=None))
# <a href="http://example.com/?foo=val1&bar=val2">A link</a>
```

Finally, if you pass in a function for `formatter`, Beautiful Soup will call that function once for every string and attribute value in the document. You can do whatever you want in this function. Here's a formatter that converts strings to uppercase and does absolutely nothing else:

```
def uppercase(str):
    return str.upper()

print(soup.prettify(formatter=uppercase))
# <html>
#  <body>
#   <p>
#    IL A DIT <<SACRÉ BLEU!>>
#   </p>
#  </body>
# </html>

print(link_soup.a.prettify(formatter=uppercase))
# <a href="HTTP://EXAMPLE.COM/?FOO=VAL1&BAR=VAL2">
#  A LINK
# </a>
```

If you're writing your own function, you should know about the `EntitySubstitution` class in the `bs4.dammit` module. This class implements Beautiful Soup's standard formatters as class methods: the "html" formatter is `EntitySubstitution.substitute_html`, and the "minimal" formatter is `EntitySubstitution.substitute_xml`. You can use these functions to simulate `formatter=html` or `formatter==minimal`, but then do something extra.

Here's an example that replaces Unicode characters with HTML entities whenever possible, but *also* converts all strings to uppercase:

```
from bs4.dammit import EntitySubstitution
def uppercase_and_substitute_html_entities(str):
    return EntitySubstitution.substitute_html(str.upper())

print(soup.prettify(formatter=uppercase_and_substitute_html_entities))
# <html>
#  <body>
```

```
#   <p>
#    IL A DIT &lt;&lt;SACR&Eacute; BLEU!&gt;&gt;
#   </p>
#  </body>
# </html>
```

One last caveat: if you create a `CData` object, the text inside that object is always presented *exactly as it appears, with no formatting*. Beautiful Soup will call the formatter method, just in case you've written a custom method that counts all the strings in the document or something, but it will ignore the return value:

```
from bs4.element import CData
soup = BeautifulSoup("<a></a>")
soup.a.string = CData("one < three")
print(soup.a.prettify(formatter="xml"))
# <a>
#  <![CDATA[one < three]]>
# </a>
```

get_text()

If you only want the text part of a document or tag, you can use the `get_text()` method. It returns all the text in a document or beneath a tag, as a single Unicode string:

```
markup = '<a href="http://example.com/">\nI linked to <i>example.com</i>\n</a>'
soup = BeautifulSoup(markup)

soup.get_text()
u'\nI linked to example.com\n'
soup.i.get_text()
u'example.com'
```

You can specify a string to be used to join the bits of text together:

```
# soup.get_text("|")
u'\nI linked to |example.com|\n'
```

You can tell Beautiful Soup to strip whitespace from the beginning and end of each bit of text:

```
# soup.get_text("|", strip=True)
u'I linked to|example.com'
```

But at that point you might want to use the *.stripped_strings* generator instead, and process the text yourself:

```
[text for text in soup.stripped_strings]
# [u'I linked to', u'example.com']
```

Specifying the parser to use

If you just need to parse some HTML, you can dump the markup into the `BeautifulSoup` constructor, and it'll probably be fine. Beautiful Soup will pick a parser for you and parse the data. But there are a few additional arguments you can pass in to the constructor to change which parser is used.

The first argument to the `BeautifulSoup` constructor is a string or an open filehandle–the markup you want parsed. The second argument is *how* you'd like the markup parsed.

If you don't specify anything, you'll get the best HTML parser that's installed. Beautiful Soup ranks lxml's parser as being the best, then html5lib's, then Python's built-in parser. You can override this by specifying one of the following:

- What type of markup you want to parse. Currently supported are "html", "xml", and "html5".
- The name of the parser library you want to use. Currently supported options are "lxml", "html5lib", and "html.parser" (Python's built-in HTML parser).

The section Installing a parser contrasts the supported parsers.

If you don't have an appropriate parser installed, Beautiful Soup will ignore your request and pick a different parser. Right now, the only supported XML parser is lxml. If you don't have lxml installed, asking for an XML parser won't give you one, and asking for "lxml" won't work either.

Differences between parsers

Beautiful Soup presents the same interface to a number of different parsers, but each parser is different. Different parsers will create different parse trees from the same document. The biggest differences are between the HTML parsers and the XML parsers. Here's a short document, parsed as HTML:

```
BeautifulSoup("<a><b /></a>")
# <html><head></head><body><a><b></b></a></body></html>
```

Since an empty tag is not valid HTML, the parser turns it into a tag pair.

Here's the same document parsed as XML (running this requires that you have lxml installed). Note that the empty <b /> tag is left alone, and that the document is given an XML declaration instead of being put into an <html> tag.:

```
BeautifulSoup("<a><b /></a>", "xml")
# <?xml version="1.0" encoding="utf-8"?>
# <a><b/></a>
```

There are also differences between HTML parsers. If you give Beautiful Soup a perfectly-formed HTML document, these differences won't matter. One parser will be faster than another, but they'll all give you a data structure that looks exactly like the original HTML document.

But if the document is not perfectly-formed, different parsers will give different results. Here's a short, invalid document parsed using lxml's HTML parser. Note that the dangling </p> tag is simply ignored:

```
BeautifulSoup("<a></p>", "lxml")
# <html><body><a></a></body></html>
```

Here's the same document parsed using html5lib:

```
BeautifulSoup("<a></p>", "html5lib")
# <html><head></head><body><a><p></p></a></body></html>
```

Instead of ignoring the dangling </p> tag, html5lib pairs it with an opening <p> tag. This parser also adds an empty <head> tag to the document.

Here's the same document parsed with Python's built-in HTML parser:

```
BeautifulSoup("<a></p>", "html.parser")
# <a></a>
```

Like html5lib, this parser ignores the closing </p> tag. Unlike html5lib, this parser makes no attempt to create a well-formed HTML document by adding a <body> tag. Unlike lxml, it doesn't even bother to add an <html> tag.

Since the document "<a></p>" is invalid, none of these techniques is the "correct" way to handle it. The html5lib parser uses techniques that are part of the HTML5 standard, so it has the best claim on being the "correct" way, but all three techniques are legitimate.

Differences between parsers can affect your script. If you're planning on distributing your script to other people, or running it on multiple machines, you should specify a parser in the `BeautifulSoup` constructor. That will reduce the chances that your users parse a document differently from the way you parse it.

Encodings

Any HTML or XML document is written in a specific encoding like ASCII or UTF-8. But when you load that document into Beautiful Soup, you'll discover it's been converted to Unicode:

```
markup = "<h1>Sacr\xc3\xa9 bleu!</h1>"
soup = BeautifulSoup(markup)
soup.h1
# <h1>Sacré bleu!</h1>
soup.h1.string
# u'Sacr\xe9 bleu!'
```

It's not magic. (That sure would be nice.) Beautiful Soup uses a sub-library called Unicode, Dammit to detect a document's encoding and convert it to Unicode. The autodetected encoding is available as the `.original_encoding` attribute of the `BeautifulSoup` object:

```
soup.original_encoding
'utf-8'
```

Unicode, Dammit guesses correctly most of the time, but sometimes it makes mistakes. Sometimes it guesses correctly, but only after a byte-by-byte search of the document that takes a very long time. If you happen to know a document's encoding ahead of time, you can avoid mistakes and delays by passing it to the `BeautifulSoup` constructor as `from_encoding`.

Here's a document written in ISO-8859-8. The document is so short that Unicode, Dammit can't get a good lock on it, and misidentifies it as ISO-8859-7:

```
markup = b"<h1>\xed\xe5\xec\xf9</h1>"
soup = BeautifulSoup(markup)
soup.h1
<h1>νεμω</h1>
soup.original_encoding
'ISO-8859-7'
```

We can fix this by passing in the correct `from_encoding`:

```
soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
soup.h1
<h1>םולש</h1>
soup.original_encoding
'iso8859-8'
```

If you don't know what the correct encoding is, but you know that Unicode, Dammit is guessing wrong, you can pass the wrong guesses in as `exclude_encodings`:

```
soup = BeautifulSoup(markup, exclude_encodings=["ISO-8859-7"])
soup.h1
<h1>םולש</h1>
soup.original_encoding
'WINDOWS-1255'
```

Windows-1255 isn't 100% correct, but that encoding is a compatible superset of ISO-8859-8, so it's close enough. (`exclude_encodings` is a new feature in Beautiful Soup 4.4.0.)

In rare cases (usually when a UTF-8 document contains text written in a completely different encoding), the only way to get Unicode may be to replace some characters with the special Unicode character "REPLACEMENT CHARACTER" (U+FFFD, �). If Unicode, Dammit needs to do this, it will set the `.contains_replacement_characters` attribute to `True` on the `UnicodeDammit` or `BeautifulSoup` object. This lets you know that the Unicode representation is not an exact representation of the original–some data was lost. If a document contains �, but `.contains_replacement_characters` is `False`, you'll know that the � was there originally (as it is in this paragraph) and doesn't stand in for missing data.

Output encoding

When you write out a document from Beautiful Soup, you get a UTF-8 document, even if the document wasn't in UTF-8 to begin with. Here's a document written in the Latin-1 encoding:

```
markup = b'''
 <html>
  <head>
   <meta content="text/html; charset=ISO-Latin-1" http-equiv="Content-type" />
  </head>
  <body>
   <p>Sacr\xe9 bleu!</p>
  </body>
 </html>
'''

soup = BeautifulSoup(markup)
print(soup.prettify())
# <html>
#  <head>
#   <meta content="text/html; charset=utf-8" http-equiv="Content-type" />
#  </head>
#  <body>
#   <p>
#    Sacré bleu!
#   </p>
#  </body>
# </html>
```

Note that the <meta> tag has been rewritten to reflect the fact that the document is now in UTF-8.

If you don't want UTF-8, you can pass an encoding into `prettify()`:

```
print(soup.prettify("latin-1"))
# <html>
#  <head>
#   <meta content="text/html; charset=latin-1" http-equiv="Content-type" />
# ...
```

You can also call encode() on the `BeautifulSoup` object, or any element in the soup, just as if it were a Python string:

```
soup.p.encode("latin-1")
# '<p>Sacr\xe9 bleu!</p>'

soup.p.encode("utf-8")
# '<p>Sacr\xc3\xa9 bleu!</p>'
```

Any characters that can't be represented in your chosen encoding will be converted into numeric XML entity references. Here's a document that includes the Unicode character SNOWMAN:

```
markup = u"<b>\N{SNOWMAN}</b>"
snowman_soup = BeautifulSoup(markup)
tag = snowman_soup.b
```

The SNOWMAN character can be part of a UTF-8 document (it looks like ☃), but there's no representation for that character in ISO-Latin-1 or ASCII, so it's converted into "&#9731" for those encodings:

```
print(tag.encode("utf-8"))
# <b>☃</b>

print tag.encode("latin-1")
# <b>&#9731;</b>

print tag.encode("ascii")
# <b>&#9731;</b>
```

Unicode, Dammit

You can use Unicode, Dammit without using Beautiful Soup. It's useful whenever you have data in an unknown encoding and you just want it to become Unicode:

```
from bs4 import UnicodeDammit
dammit = UnicodeDammit("Sacr\xc3\xa9 bleu!")
print(dammit.unicode_markup)
# Sacré bleu!
```

```
dammit.original_encoding
# 'utf-8'
```

Unicode, Dammit's guesses will get a lot more accurate if you install the `chardet` or `cchardet` Python libraries. The more data you give Unicode, Dammit, the more accurately it will guess. If you have your own suspicions as to what the encoding might be, you can pass them in as a list:

```
dammit = UnicodeDammit("Sacr\xe9 bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

Unicode, Dammit has two special features that Beautiful Soup doesn't use.

Smart quotes

You can use Unicode, Dammit to convert Microsoft smart quotes to HTML or XML entities:

```
markup = b"<p>I just \x93love\x94 Microsoft Word\x92s smart quotes</p>"

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="html").unicode_markup
# u'<p>I just &ldquo;love&rdquo; Microsoft Word&rsquo;s smart quotes</p>'

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="xml").unicode_markup
# u'<p>I just &#x201C;love&#x201D; Microsoft Word&#x2019;s smart quotes</p>'
```

You can also convert Microsoft smart quotes to ASCII quotes:

```
UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="ascii").unicode_markup
# u'<p>I just "love" Microsoft Word\'s smart quotes</p>'
```

Hopefully you'll find this feature useful, but Beautiful Soup doesn't use it. Beautiful Soup prefers the default behavior, which is to convert Microsoft smart quotes to Unicode characters along with everything else:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'
```

Inconsistent encodings

Sometimes a document is mostly in UTF-8, but contains Windows-1252 characters such as (again) Microsoft smart quotes. This can happen when a website includes data from multiple sources. You can use `UnicodeDammit.detwingle()` to turn such a document into pure UTF-8. Here's a simple example:

```
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE QUOTATION MA
doc = snowmen.encode("utf8") + quote.encode("windows_1252")
```

This document is a mess. The snowmen are in UTF-8 and the quotes are in Windows-1252. You can display the snowmen or the quotes, but not both:

```
print(doc)
# ☃☃☃�I like snowmen!�

print(doc.decode("windows-1252"))
# â˜ƒâ˜ƒâ˜ƒ“I like snowmen!”
```

Decoding the document as UTF-8 raises a `UnicodeDecodeError`, and decoding it as Windows-1252 gives you gibberish. Fortunately, `UnicodeDammit.detwingle()` will convert the string to pure UTF-8, allowing you to decode it to Unicode and display the snowmen and quote marks simultaneously:

```
new_doc = UnicodeDammit.detwingle(doc)
print(new_doc.decode("utf8"))
# ☃☃☃“I like snowmen!”
```

`UnicodeDammit.detwingle()` only knows how to handle Windows-1252 embedded in UTF-8 (or vice versa, I suppose), but this is the most common case.

Note that you must know to call `UnicodeDammit.detwingle()` on your data before passing it into `BeautifulSoup` or the `UnicodeDammit` constructor. Beautiful Soup assumes that a document has a single encoding, whatever it might be. If you pass it a document that contains both UTF-8 and Windows-1252, it's likely to think the whole document is Windows-1252, and the document will come out looking like `â˜ƒâ˜ƒâ˜ƒ“I like snowmen!”`.

`UnicodeDammit.detwingle()` is new in Beautiful Soup 4.1.0.

Comparing objects for equality

Beautiful Soup says that two `NavigableString` or `Tag` objects are equal when they represent the same HTML or XML markup. In this example, the two tags are treated as equal, even though they live in different parts of the object tree, because they both look like “pizza”:

```
markup = "<p>I want <b>pizza</b> and more <b>pizza</b>!</p>"
soup = BeautifulSoup(markup, 'html.parser')
first_b, second_b = soup.find_all('b')
print first_b == second_b
# True

print first_b.previous_element == second_b.previous_element
```

```
# False
```

If you want to see whether two variables refer to exactly the same object, use *is*:

```
print first_b is second_b
# False
```

Copying Beautiful Soup objects

You can use `copy.copy()` to create a copy of any `Tag` or `NavigableString`:

```
import copy
p_copy = copy.copy(soup.p)
print p_copy
# <p>I want <b>pizza</b> and more <b>pizza</b>!</p>
```

The copy is considered equal to the original, since it represents the same markup as the original, but it’s not the same object:

```
print soup.p == p_copy
# True

print soup.p is p_copy
# False
```

The only real difference is that the copy is completely detached from the original Beautiful Soup object tree, just as if `extract()` had been called on it:

```
print p_copy.parent
# None
```

This is because two different `Tag` objects can’t occupy the same space at the same time.

Parsing only part of a document

Let’s say you want to use Beautiful Soup look at a document’s <a> tags. It’s a waste of time and memory to parse the entire document and then go over it again looking for <a> tags. It would be much faster to ignore everything that wasn’t an <a> tag in the first place. The `SoupStrainer` class allows you to choose which parts of an incoming document are parsed. You just create a `SoupStrainer` and pass it in to the `BeautifulSoup` constructor as the `parse_only` argument.

(Note that *this feature won’t work if you’re using the html5lib parser*. If you use html5lib, the whole document will be parsed, no matter what. This is because html5lib constantly rearranges the parse tree as it works, and if some part of the document didn't actually make it into the parse tree, it'll crash. To avoid confusion, in the examples below I'll be forcing Beautiful Soup to use Python's built-in parser.)

SoupStrainer

The `SoupStrainer` class takes the same arguments as a typical method from Searching the tree: *name*, *attrs*, *string*, and ***kwargs*. Here are three `SoupStrainer` objects:

```
from bs4 import SoupStrainer

only_a_tags = SoupStrainer("a")

only_tags_with_id_link2 = SoupStrainer(id="link2")

def is_short_string(string):
    return len(string) < 10

only_short_strings = SoupStrainer(string=is_short_string)
```

I'm going to bring back the "three sisters" document one more time, and we'll see what the document looks like when it's parsed with these three `SoupStrainer` objects:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names w
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_a_tags).prettify())
# <a class="sister" href="http://example.com/elsie" id="link1">
#  Elsie
# </a>
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>
# <a class="sister" href="http://example.com/tillie" id="link3">
#  Tillie
# </a>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_tags_with_id_link2).pre
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_short_strings).prettify
```

```
# Elsie
# ,
# Lacie
# and
# Tillie
# ...
#
```

You can also pass a `SoupStrainer` into any of the methods covered in Searching the tree. This probably isn't terribly useful, but I thought I'd mention it:

```
soup = BeautifulSoup(html_doc)
soup.find_all(only_short_strings)
# [u'\n\n', u'\n\n', u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
#  u'\n\n', u'...', u'\n']
```

Troubleshooting

diagnose()

If you're having trouble understanding what Beautiful Soup does to a document, pass the document into the `diagnose()` function. (New in Beautiful Soup 4.2.0.) Beautiful Soup will print out a report showing you how different parsers handle the document, and tell you if you're missing a parser that Beautiful Soup could be using:

```
from bs4.diagnose import diagnose
data = open("bad.html").read()
diagnose(data)

# Diagnostic running on Beautiful Soup 4.2.0
# Python version 2.7.3 (default, Aug  1 2012, 05:16:07)
# I noticed that html5lib is not installed. Installing it may help.
# Found lxml version 2.3.2.0
#
# Trying to parse your data with html.parser
# Here's what html.parser did with the document:
# ...
```

Just looking at the output of diagnose() may show you how to solve the problem. Even if not, you can paste the output of `diagnose()` when asking for help.

Errors when parsing a document

There are two different kinds of parse errors. There are crashes, where you feed a document to Beautiful Soup and it raises an exception, usually an `HTMLParser.HTMLParseError`. And there is unexpected behavior, where a Beautiful Soup parse tree looks a lot different than the document used to create it.

Almost none of these problems turn out to be problems with Beautiful Soup. This is not because Beautiful Soup is an amazingly well-written piece of software. It's because Beautiful Soup doesn't include any parsing code. Instead, it relies on external parsers. If one parser isn't working on a certain document, the best solution is to try a different parser. See Installing a parser for details and a parser comparison.

The most common parse errors are `HTMLParser.HTMLParseError: malformed start tag` and `HTMLParser.HTMLParseError: bad end tag`. These are both generated by Python's built-in HTML parser library, and the solution is to *install lxml or html5lib.*

The most common type of unexpected behavior is that you can't find a tag that you know is in the document. You saw it going in, but `find_all()` returns `[]` or `find()` returns `None`. This is another common problem with Python's built-in HTML parser, which sometimes skips tags it doesn't understand. Again, the solution is to *install lxml or html5lib.*

Version mismatch problems

- `SyntaxError: Invalid syntax` (on the line `ROOT_TAG_NAME = u'[document]'`): Caused by running the Python 2 version of Beautiful Soup under Python 3, without converting the code.
- `ImportError: No module named HTMLParser` - Caused by running the Python 2 version of Beautiful Soup under Python 3.
- `ImportError: No module named html.parser` - Caused by running the Python 3 version of Beautiful Soup under Python 2.
- `ImportError: No module named BeautifulSoup` - Caused by running Beautiful Soup 3 code on a system that doesn't have BS3 installed. Or, by writing Beautiful Soup 4 code without knowing that the package name has changed to `bs4`.
- `ImportError: No module named bs4` - Caused by running Beautiful Soup 4 code on a system that doesn't have BS4 installed.

Parsing XML

By default, Beautiful Soup parses documents as HTML. To parse a document as XML, pass in "xml" as the second argument to the `BeautifulSoup` constructor:

```
soup = BeautifulSoup(markup, "xml")
```

You'll need to *have lxml installed*.

Other parser problems

- If your script works on one computer but not another, or in one virtual environment but not another, or outside the virtual environment but not inside, it's probably because the two environments have different parser libraries available. For example, you may have developed the script on a computer that has lxml installed, and then tried to run it on a computer that only has html5lib installed. See Differences between parsers for why this matters, and fix the problem by mentioning a specific parser library in the `BeautifulSoup` constructor.
- Because HTML tags and attributes are case-insensitive, all three HTML parsers convert tag and attribute names to lowercase. That is, the markup <TAG></TAG> is converted to <tag></tag>. If you want to preserve mixed-case or uppercase tags and attributes, you'll need to *parse the document as XML.*

Miscellaneous

- `UnicodeEncodeError: 'charmap' codec can't encode character u'\xfoo' in position bar` (or just about any other `UnicodeEncodeError`) - This is not a problem with Beautiful Soup. This problem shows up in two main situations. First, when you try to print a Unicode character that your console doesn't know how to display. (See this page on the Python wiki for help.) Second, when you're writing to a file and you pass in a Unicode character that's not supported by your default encoding. In this case, the simplest solution is to explicitly encode the Unicode string into UTF-8 with `u.encode("utf8")`.
- `KeyError: [attr]` - Caused by accessing `tag['attr']` when the tag in question doesn't define the `attr` attribute. The most common errors are `KeyError: 'href'` and `KeyError: 'class'`. Use `tag.get('attr')` if you're not sure `attr` is defined, just as you would with a Python dictionary.
- `AttributeError: 'ResultSet' object has no attribute 'foo'` - This usually happens because you expected `find_all()` to return a single tag or string. But `find_all()` returns a _list_ of tags and strings–a `ResultSet` object. You need to iterate over the list and look at the `.foo` of each one. Or, if you really only want one result, you need to use `find()` instead of `find_all()`.
- `AttributeError: 'NoneType' object has no attribute 'foo'` - This usually happens because you called `find()` and then tried to access the *.foo*` attribute of the result. But in your case, `find()` didn't find anything, so it returned `None`, instead of returning a tag or a string. You need to

figure out why your `find()` call isn't returning anything.

Improving Performance

Beautiful Soup will never be as fast as the parsers it sits on top of. If response time is critical, if you're paying for computer time by the hour, or if there's any other reason why computer time is more valuable than programmer time, you should forget about Beautiful Soup and work directly atop lxml.

That said, there are things you can do to speed up Beautiful Soup. If you're not using lxml as the underlying parser, my advice is to *start*. Beautiful Soup parses documents significantly faster using lxml than using html.parser or html5lib.

You can speed up encoding detection significantly by installing the cchardet library.

Parsing only part of a document won't save you much time parsing the document, but it can save a lot of memory, and it'll make *searching* the document much faster.

Beautiful Soup 3

Beautiful Soup 3 is the previous release series, and is no longer being actively developed. It's currently packaged with all major Linux distributions:

```
$ apt-get install python-beautifulsoup
```

It's also published through PyPi as `BeautifulSoup`.:

```
$ easy_install BeautifulSoup
```

```
$ pip install BeautifulSoup
```

You can also download a tarball of Beautiful Soup 3.2.0.

If you ran `easy_install beautifulsoup` or `easy_install BeautifulSoup`, but your code doesn't work, you installed Beautiful Soup 3 by mistake. You need to run `easy_install beautifulsoup4`.

The documentation for Beautiful Soup 3 is archived online.

Porting code to BS4

Most code written against Beautiful Soup 3 will work against Beautiful Soup 4 with one simple change. All you should have to do is change the package name from `BeautifulSoup` to `bs4`. So this:

```
from BeautifulSoup import BeautifulSoup
```

becomes this:

```
from bs4 import BeautifulSoup
```

- If you get the `ImportError` "No module named BeautifulSoup", your problem is that you're trying to run Beautiful Soup 3 code, but you only have Beautiful Soup 4 installed.
- If you get the `ImportError` "No module named bs4", your problem is that you're trying to run Beautiful Soup 4 code, but you only have Beautiful Soup 3 installed.

Although BS4 is mostly backwards-compatible with BS3, most of its methods have been deprecated and given new names for PEP 8 compliance. There are numerous other renames and changes, and a few of them break backwards compatibility.

Here's what you'll need to know to convert your BS3 code and habits to BS4:

You need a parser

Beautiful Soup 3 used Python's `SGMLParser`, a module that was deprecated and removed in Python 3.0. Beautiful Soup 4 uses `html.parser` by default, but you can plug in lxml or html5lib and use that instead. See Installing a parser for a comparison.

Since `html.parser` is not the same parser as `SGMLParser`, you may find that Beautiful Soup 4 gives you a different parse tree than Beautiful Soup 3 for the same markup. If you swap out `html.parser` for lxml or html5lib, you may find that the parse tree changes yet again. If this happens, you'll need to update your scraping code to deal with the new tree.

Method names

- `renderContents` -> `encode_contents`
- `replaceWith` -> `replace_with`
- `replaceWithChildren` -> `unwrap`
- `findAll` -> `find_all`
- `findAllNext` -> `find_all_next`

- `findAllPrevious` -> `find_all_previous`
- `findNext` -> `find_next`
- `findNextSibling` -> `find_next_sibling`
- `findNextSiblings` -> `find_next_siblings`
- `findParent` -> `find_parent`
- `findParents` -> `find_parents`
- `findPrevious` -> `find_previous`
- `findPreviousSibling` -> `find_previous_sibling`
- `findPreviousSiblings` -> `find_previous_siblings`
- `nextSibling` -> `next_sibling`
- `previousSibling` -> `previous_sibling`

Some arguments to the Beautiful Soup constructor were renamed for the same reasons:

- `BeautifulSoup(parseOnlyThese=...)` -> `BeautifulSoup(parse_only=...)`
- `BeautifulSoup(fromEncoding=...)` -> `BeautifulSoup(from_encoding=...)`

I renamed one method for compatibility with Python 3:

- `Tag.has_key()` -> `Tag.has_attr()`

I renamed one attribute to use more accurate terminology:

- `Tag.isSelfClosing` -> `Tag.is_empty_element`

I renamed three attributes to avoid using words that have special meaning to Python. Unlike the others, these changes are *not backwards compatible.* If you used these attributes in BS3, your code will break on BS4 until you change them.

- `UnicodeDammit.unicode` -> `UnicodeDammit.unicode_markup`
- `Tag.next` -> `Tag.next_element`
- `Tag.previous` -> `Tag.previous_element`

Generators

I gave the generators PEP 8-compliant names, and transformed them into properties:

- `childGenerator()` -> `children`
- `nextGenerator()` -> `next_elements`
- `nextSiblingGenerator()` -> `next_siblings`
- `previousGenerator()` -> `previous_elements`
- `previousSiblingGenerator()` -> `previous_siblings`

- `recursiveChildGenerator()` -> `descendants`
- `parentGenerator()` -> `parents`

So instead of this:

```
for parent in tag.parentGenerator():
    ...
```

You can write this:

```
for parent in tag.parents:
    ...
```

(But the old code will still work.)

Some of the generators used to yield `None` after they were done, and then stop. That was a bug. Now the generators just stop.

There are two new generators, *.strings and .stripped_strings*. `.strings` yields NavigableString objects, and `.stripped_strings` yields Python strings that have had whitespace stripped.

XML

There is no longer a `BeautifulStoneSoup` class for parsing XML. To parse XML you pass in "xml" as the second argument to the `BeautifulSoup` constructor. For the same reason, the `BeautifulSoup` constructor no longer recognizes the `isHTML` argument.

Beautiful Soup's handling of empty-element XML tags has been improved. Previously when you parsed XML you had to explicitly say which tags were considered empty-element tags. The `selfClosingTags` argument to the constructor is no longer recognized. Instead, Beautiful Soup considers any empty tag to be an empty-element tag. If you add a child to an empty-element tag, it stops being an empty-element tag.

Entities

An incoming HTML or XML entity is always converted into the corresponding Unicode character. Beautiful Soup 3 had a number of overlapping ways of dealing with entities, which have been removed. The `BeautifulSoup` constructor no longer recognizes the `smartQuotesTo` or `convertEntities` arguments. (Unicode, Dammit still has `smart_quotes_to`, but its default is now to turn smart quotes into Unicode.) The constants `HTML_ENTITIES`, `XML_ENTITIES`, and `XHTML_ENTITIES` have been removed, since they configure a feature (transforming some but not all entities into Unicode characters) that no longer exists.

If you want to turn Unicode characters back into HTML entities on output, rather than turning them into UTF-8 characters, you need to use an *output formatter*.

Miscellaneous

Tag.string now operates recursively. If tag A contains a single tag B and nothing else, then A.string is the same as B.string. (Previously, it was None.)

Multi-valued attributes like `class` have lists of strings as their values, not strings. This may affect the way you search by CSS class.

If you pass one of the `find*` methods both *string* *and* a tag-specific argument like *name*, Beautiful Soup will search for tags that match your tag-specific criteria and whose *Tag.string* matches your value for *string*. It will *not* find the strings themselves. Previously, Beautiful Soup ignored the tag-specific arguments and looked for strings.

The `BeautifulSoup` constructor no longer recognizes the *markupMassage* argument. It's now the parser's responsibility to handle markup correctly.

The rarely-used alternate parser classes like `ICantBelieveItsBeautifulSoup` and `BeautifulSOAP` have been removed. It's now the parser's decision how to handle ambiguous markup.

The `prettify()` method now returns a Unicode string, not a bytestring.

Testing Your Code

Fork me on GitHub

Testing your code is very important.

Getting used to writing testing code and running this code in parallel is now considered a good habit. Used wisely, this method helps you define more precisely your code's intent and have a more decoupled architecture.

Some general rules of testing:

- A testing unit should focus on one tiny bit of functionality and prove it correct.
- Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by **`setUp()`** and **`tearDown()`** methods.
- Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable. In some cases, tests can't be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed.
- Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently, ideally automatically when you save the code.
- Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
- It is a good idea to implement a hook that runs all tests before pushing code to a shared repository.
- If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next. When coming back to work, you will have a pointer to where you were and get back on track faster.
- The first step when you are debugging your code is to write a new test pinpointing the bug. While it is not always possible to do, those bug catching tests are among the most valuable pieces of code in your project.
- Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. `square()` or even `sqr()` is ok in running code, but in testing code you would have names such as `test_square_of_number_2()`, `test_square_negative_number()`. These function names are displayed when a test fails, and should be as descriptive as possible.
- When something goes wrong or has to be changed, and if your code has a good set of tests, you or other maintainers will rely largely on the testing suite to fix the problem or modify a given behavior. Therefore the testing code will be read as much as or even more than the running code. A unit test whose purpose is unclear is not very helpful in this case.
- Another use of the testing code is as an introduction to new developers. When someone will have to work on the code base, running and reading the related testing code is often the best thing that they can do to start. They will or should discover the hot spots, where most difficulties arise, and the corner cases. If they have to add some functionality, the first step should be to add a test to ensure that the new functionality is not already a working path that has not been plugged into the interface.

The Basics

Unittest

`unittest` is the batteries-included test module in the Python standard library. Its API will be familiar to anyone who has used any of the JUnit/nUnit/CppUnit series of tools.

v: latest

Creating test cases is accomplished by subclassing **`unittest.TestCase`**.

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

As of Python 2.7 unittest also includes its own test discovery mechanisms.

unittest in the standard library documentation

Doctest

The **doctest** module searches for pieces of text that look like interactive Python sessions in docstrings, and then executes those sessions to verify that they work exactly as shown.

Doctests have a different use case than proper unit tests: they are usually less detailed and don't catch special cases or obscure regression bugs. They are useful as an expressive documentation of the main use cases of a module and its components. However, doctests should run automatically each time the full test suite runs.

A simple doctest in a function:

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """

    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

When running this module from the command line as in `python module.py`, the doctests will run and complain if anything is not behaving as described in the docstrings.

Tools

py.test

py.test is a no-boilerplate alternative to Python's standard unittest module.

```
$ pip install pytest
```

Despite being a fully-featured and extensible test tool, it boasts a simple syntax. Creating a test suite is as easy as writing a module with a couple of functions:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

v: latest

and then running the *py.test* command

```
$ py.test
=========================== test session starts ============================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

================================= FAILURES =================================
_______________________________ test_answer ________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:5: AssertionError
========================= 1 failed in 0.02 seconds =========================
```

is far less work than would be required for the equivalent functionality with the unittest module!

py.test

Nose

nose extends unittest to make testing easier.

```
$ pip install nose
```

nose provides automatic test discovery to save you the hassle of manually creating test suites. It also provides numerous plugins for features such as xUnit-compatible test output, coverage reporting, and test selection.

nose

tox

tox is a tool for automating test environment management and testing against multiple interpreter configurations

```
$ pip install tox
```

tox allows you to configure complicated multi-parameter test matrices via a simple ini-style configuration file.

tox

Unittest2

unittest2 is a backport of Python 2.7's unittest module which has an improved API and better assertions over the one available in previous versions of Python.

If you're using Python 2.6 or below, you can install it with pip

```
$ pip install unittest2
```

You may want to import the module under the name unittest to make porting code to newer versions of the module easier in the future

```
import unittest2 as unittest
```

v: latest

```
class MyTest(unittest.TestCase):
    ...
```

This way if you ever switch to a newer Python version and no longer need the unittest2 module, you can simply change the import in your test module without the need to change any other code.

unittest2

mock

unittest.mock is a library for testing in Python. As of Python 3.3, it is available in the standard library.

For older versions of Python:

```
$ pip install mock
```

It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

For example, you can monkey-patch a method:

```
from mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')

thing.method.assert_called_with(3, 4, 5, key='value')
```

To mock classes or objects in a module under test, use the `patch` decorator. In the example below, an external search system is replaced with a mock that always returns the same result (but only for the duration of the test).

```
def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()

# SearchForm here refers to the imported class reference in myapp,
# not where the SearchForm class itself is imported from
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results runs a search and iterates over the result
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Mock has many other ways you can configure it and control its behavior.

mock

v: latest

PYTHON
TESTING*<http://pythontesting.net/>*

Python Software Development and Software Testing (posts and podcast)

pytest introduction

January 15, 2013 By Brian<http://pythontesting.net/author/brian-2/>

I think of pytest as the run-anything, no boilerplate, no required api, use-this-unless-you-have-a-reason-not-to test framework.
This is really where testing gets fun.
As with previous intro's on this site, I'll run through an overview, then a simple example, then throw pytest at my markdown.py project. I'll also cover fixtures, test discovery, and running unittests with pytest.

Contents

- No boilerplate, no required api
- pytest example
- Running pytest
- pytest fixtures
- Testing markdown.py
- Test discovery
- Running unittests from pytest
- Running doctests from pytest
- More pytest info (links)
- Examples on github
- Next

No boilerplate, no required api

The doctest*<http://pythontesting.net/framework/doctest-introduction/>* and unittest*<http://pythontesting.net/framework/unittest-introduction/>* both come with Python.

They are pretty powerful on their own, and I think you should at least know about those frameworks, and learn how to run them at least on some toy examples, as it gives you a mental framework to view other test frameworks.

Note:
The module *unnecessary_math* is non-standard and can be found here: implementation of unnecessary_math.py

With unittest, you a very basic test file might look like this:

```
import unittest
from unnecessary_math import multiply

class TestUM(unittest.TestCase):

    def test_numbers_3_4(self):
        self.assertEqual( multiply(3,4), 12)
```

The style of deriving from `unittest.TestCase` is something unittest shares with it's xUnit counterparts like JUnit.

I don't want to get into the history of xUnit style frameworks. However, it's informative to know that inheritance is quite important in some languages to get the test framework to work right.

But this is Python. We have very powerful introspection and runtime capabilities, and very little information hiding. Pytest takes advantage of this.

An identical test as above could look like this if we remove the boilerplate:

```
from unnecessary_math import multiply

def test_numbers_3_4():
    assert( multiply(3,4) == 12 )
```

Yep, three lines of code. (Four, if you include the blank line.)
There is no need to `import unnittest`.

There is no need to derive from `TestCase`.
There is no need to for special `self.assertEqual()`, since we can use Python's built in `assert` statement.

This works in pytest. Once you start writing tests like this, you won't want to go back.

However, you may have a bunch of tests already written for doctest or unittest.
Pytest can be used to run doctests and unittests.
It also claims to support some twisted trial tests (although I haven't tried this).

You can extend pytest using plugins you pull from the web, or write yourself.
I'm not going to cover plugins in this article, but I'm sure I'll get into it in a future article.

You will sometimes see pytest referred to as py.test.
I use this convention:
pytest : the project
py.test : the command line tool that runs pytest
I'm not sure if that's 100% accurate according to how the folks at pytest.org use the terms.

pytest example

Using the same unnecessary_math.py module that I wrote in the doctest intro<*http://pythontesting.net/framework/doctest-introduction/#example*>, this is some example test code to test the 'multiply' function.

```
from unnecessary_math import multiply

def test_numbers_3_4():
    assert multiply(3,4) == 12

def test_strings_a_3():
    assert multiply('a',3) == 'aaa'
```

Running pytest

To run pytest, the following two calls are identical:

```
python -m pytest test_um_pytest.py
py.test test_um_pytest.py
```

And with verbose:

```
python -m pytest -v test_um_pytest.py
py.test -v test_um_pytest.py
```

I'll use `py.test`, as it's shorter to type.

Here's an example run both with and without verbose:

```
> py.test test_um_pytest.py
============================= test session starts =======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 2 items

test_um_pytest.py ..

=========================== 2 passed in 0.05 seconds ====================


> py.test -v test_um_pytest.py
============================= test session starts =======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4 -- C:\python27\python.exe
collecting ... collected 2 items

test_um_pytest.py:12: test_numbers_3_4 PASSED
test_um_pytest.py:15: test_strings_a_3 PASSED

=========================== 2 passed in 0.02 seconds ====================
```

pytest fixtures

Although unittest does allow us to have setup and teardown, pytest extends this quite a bit.
We can add specific code to run:

- at the beginning and end of a module of test code (setup_module/teardown_module)

- at the beginning and end of a class of test methods (setup_class/teardown_class)
- alternate style of the class level fixtures (setup/teardown)
- before and after a test function call (setup_function/teardown_function)
- before and after a test method call (setup_method/teardown_method)

We can also use pytest style fixtures, which are covered in pytest fixtures nuts and bolts<*http://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/*>.

I've modified our simple test code with some fixture calls, and added some print statements so that we can see what's going on.
Here's the code:

```
from unnecessary_math import multiply

def setup_module(module):
    print ("setup_module      module:%s" % module.__name__)

def teardown_module(module):
    print ("teardown_module   module:%s" % module.__name__)

def setup_function(function):
    print ("setup_function    function:%s" % function.__name__)

def teardown_function(function):
    print ("teardown_function function:%s" % function.__name__)

def test_numbers_3_4():
    print 'test_numbers_3_4  <============================ actual test cc
    assert multiply(3,4) == 12

def test_strings_a_3():
    print 'test_strings_a_3  <============================ actual test cc
    assert multiply('a',3) == 'aaa'


class TestUM:

    def setup(self):
        print ("setup             class:TestStuff")

    def teardown(self):
        print ("teardown          class:TestStuff")

    def setup_class(cls):
        print ("setup_class       class:%s" % cls.__name__)

    def teardown_class(cls):
        print ("teardown_class    class:%s" % cls.__name__)

    def setup_method(self, method):
```

```
        print ("setup_method      method:%s" % method.__name__)

    def teardown_method(self, method):
        print ("teardown_method   method:%s" % method.__name__)

    def test_numbers_5_6(self):
        print 'test_numbers_5_6  <============================ actual tes
        assert multiply(5,6) == 30

    def test_strings_b_2(self):
        print 'test_strings_b_2  <============================ actual tes
        assert multiply('b',2) == 'bb'
```

To see it in action, I'll use the **-s** option, which turns off output capture.

This will show the order of the different fixture calls.

```
> py.test -s test_um_pytest_fixtures.py
============================= test session starts ======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 4 items

test_um_pytest_fixtures.py ....

=========================== 4 passed in 0.07 seconds ===================
setup_module       module:test_um_pytest_fixtures
setup_function     function:test_numbers_3_4
test_numbers_3_4   <============================ actual test code
teardown_function  function:test_numbers_3_4
setup_function     function:test_strings_a_3
test_strings_a_3   <============================ actual test code
teardown_function  function:test_strings_a_3
setup_class        class:TestUM
setup_method       method:test_numbers_5_6
setup              class:TestStuff
test_numbers_5_6   <============================ actual test code
teardown           class:TestStuff
teardown_method    method:test_numbers_5_6
setup_method       method:test_strings_b_2
setup              class:TestStuff
test_strings_b_2   <============================ actual test code
teardown           class:TestStuff
teardown_method    method:test_strings_b_2
teardown_class     class:TestUM
teardown_module    module:test_um_pytest_fixtures
```

Testing markdown.py

The test code to test markdown.py is going to look a lot like the unittest version*<http://pythontesting.net/framework/unittest-introduction/#example_markdown>*, but without the boilerplate.

I'm also using an API adapter*<http://pythontesting.net/strategy/software-api-cli-interface-adapters/>* introduced in a previous post.

Here's the code to use pytest to test markdown.py:

```
from markdown_adapter import run_markdown

def test_non_marked_lines():
    print ('in test_non_marked_lines')
    assert run_markdown('this line has no special handling') == \
            'this line has no special handling</p>'

def test_em():
    print ('in test_em')
    assert run_markdown('*this should be wrapped in em tags*') == \
            '<p><em>this should be wrapped in em tags</em></p>'

def test_strong():
    print ('in test_strong')
    assert run_markdown('**this should be wrapped in strong tags**') == \
            '<p><strong>this should be wrapped in strong tags</strong></p
```

And here's the output:

```
> py.test test_markdown_pytest.py
============================= test session starts ======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 3 items

test_markdown_pytest.py F.F

=================================== FAILURES ===========================
____________________________ test_non_marked_lines ______________________

    def test_non_marked_lines():
        print ('in test_non_marked_lines')
>       assert run_markdown('this line has no special handling') ==
                'this line has no special handling</p>'
E       assert 'this line ha...cial handling' == '<p>this line ... handli
E         - this line has no special handling
E         + this line has no special handling
E         ? +++                                ++++

test_markdown_pytest.py:14: AssertionError
------------------------------- Captured stdout -------------------------
in test_non_marked_lines
_________________________________ test_strong ___________________________

    def test_strong():
        print ('in test_strong')
>       assert run_markdown('**this should be wrapped in strong tags**')
                '<p><strong>this should be wrapped in strong tags</strong
E       assert '**this shoul...strong tags**' == '<p><strong>th...</stror
E         - **this should be wrapped in strong tags**
E         + <strong>this should be wrapped in strong tags</strong>

test_markdown_pytest.py:24: AssertionError
------------------------------- Captured stdout -------------------------
in test_strong
====================== 2 failed, 1 passed in 0.30 seconds ==============
```

You'll notice that all of them are failing. This is on purpose, since I haven't implemented any real markdown code yet.

However, the formatting of the output is quite nice.
It's quite easy to see why the test is failing.

Test discovery

The unittest module comes with a 'discovery' option.
Discovery is just built in to pytest.
Test discovery was used in my examples to find tests within a specified module.
However, pytest can find tests residing in multiple modules, and multiple packages, and even find unittests and doctests.
To be honest, I haven't memorized the discovery rules.
I just try to do this, and at seems to work nicely:

- Name my test modules/files starting with 'test_'.
- Name my test functions starting with 'test_'.
- Name my test classes starting with 'Test'.
- Name my test methods starting with 'test_'.
- Make sure all packages with test code have an '**init**.py' file.

If I do all of that, pytest seems to find all my code nicely.
If you are doing something else, and are having trouble getting pytest to see your test code,
then take a look at the pytest discovery documentation*<http://pytest.org/latest/example/pythoncollection.html>*.

Running unittests from pytest

To show how pytest handles unittests, here's a sample run of pytest on the simple unittests I wrote in the unittest introduction*<http://pythontesting.net/framework/unittest-introduction/>*:

```
> py.test test_um_unittest.py
============================= test session starts =======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 2 items

test_um_unittest.py ..
```

```
=========================== 2 passed in 0.07 seconds ====================
> py.test -v test_um_unittest.py
============================= test session starts =======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4 -- C:\python27\python.exe
collecting ... collected 2 items

test_um_unittest.py:15: TestUM.test_numbers_3_4 PASSED
test_um_unittest.py:18: TestUM.test_strings_a_3 PASSED

=========================== 2 passed in 0.06 seconds ====================
```

As you can see, I didn't provide any extra options, pytest finds unittests automatically.

Running doctests from pytest

You can run some doctests from pytest, according to the documentation. However, with my examples of putting doctests in text files*<http://pythontesting.net/framework/doctest-introduction/>*, I can't figure out a way to get pytest to run them.

I've tried several attempts, and keep getting into import error problems:

```
> py.test --doctest-modules test_unnecessary_math.txt
============================= test session starts =======================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 1 items

test_unnecessary_math.txt F

=================================== FAILURES ============================
__________________________________ [doctest] ____________________________
001 This is a doctest based regression suite for unnecessary_math.py
002 Each '>>>' line is run as if in a python shell, and counts as a test.
003 The next line, if not '>>>' is the expected output of the previous li
004 If anything doesn't match exactly (including trailing spaces), the te
005
006 >>> from unnecessary_math import multiply
UNEXPECTED EXCEPTION: ImportError('No module named unnecessary_math',)
Traceback (most recent call last):

  File "C:\python27\lib\doctest.py", line 1289, in __run
    compileflags, 1) in test.globs

  File "<doctest test_unnecessary_math.txt[0]>", line 1, in <module>

ImportError: No module named unnecessary_math

E:\python_notes\repo\markdown.py-dev\simple_example\test_unnecessary_math
=========================== 1 failed in 0.06 seconds ====================
```

If anyone out there knows what I'm doing wrong, please let me know. Thanks in advance.

More pytest info (links)

- pytest.org – main site, great info
- pytest fixtures – start of a series of fixture posts
- pytest fixtures easy example
- pytest xUnit style fixtures
- pytest fixtures nuts and bolts – Don't miss this!

Examples on github

All of the examples here are available in the markdown.py project*<https://github.com/variedthoughts/markdown.py>* on github.

Next

In the next post, I'll throw nose at the sampe problems.

Related posts:

1. **unittest introduction** ***(unittest introduction)***
2. **doctest introduction** ***(doctest introduction)***
3. **pytest fixtures easy example** ***(pytest fixtures easy example)***
4. **nose introduction** ***(nose introduction)***
5. **pytest xUnit style fixtures** ***(pytest xUnit style fixtures)***

Filed Under: pytest*<http://pythontesting.net/category/framework/pytest/>*
Tagged With: frameworks*<http://pythontesting.net/on/frameworks/>*, pytest*<http://pythontesting.net/on/pytest/>*

Python Testing with unittest, nose, pytest.
Get up to speed fast on pytest, unittest, and nose.

All in the comfort of your own e-reader.

1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such *extension modules* can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header `"Python.h"`.

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

> **Note:** The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the **`ctypes`** module or the cffi library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.

1.1. A Simple Example

Let's create an extension module called `spam` (the favorite food of Monty Python fans...) and let's say we want to create a Python interface to the C library function **`system()`**. [1] This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file `spammodule.c`. (Historically, if a module is called `spam`, the C file containing its implementation is called `spammodule.c`; if the module name is very long, like `spammify`, the module name can be just `spammify.c`.)

The first line of our file can be:

```
#include <Python.h>
```

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

> **Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

All user-visible symbols defined by `Python.h` have a prefix of `Py` or `PY`, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, `"Python.h"` includes a few standard header files: `<stdio.h>`, `<string.h>`, `<errno.h>`, and `<stdlib.h>`. If the latter header file does not exist on your system, it declares the functions **`malloc()`**, **`free()`** and **`realloc()`** directly.

The next thing we add to our module file is the C function that will be called when the Python expression `spam.system(string)` is evaluated (we'll see shortly how it ends up being called):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression `"ls -l"`) to the arguments passed to the C function. The C function always has two arguments, conventionally named *self* and *args*.

For module functions, the *self* argument is *NULL* or a pointer selected while initializing the module (see **`Py_InitModule4()`**). For a method, it would point to the object instance.

The *args* argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function **`PyArg_ParseTuple()`** in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return *NULL* immediately (as we saw in the example).

1.2. Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a *NULL* pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is *NULL* no exception has occurred. A second global variable stores the "associated value" of the exception (the second argument to `raise`). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback` (see the section on module `sys` in the Python Library Reference). It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the "associated value" of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don't need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred()`. This returns the current exception object, or *NULL* if no exception has occurred. You normally don't need to call `PyErr_Occurred()` to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (usually *NULL* or `-1`). It should *not* call one of the `PyErr_*()` functions — one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling `PyErr_*()`, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*()` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling `PyErr_Clear()`. The only time C code should call `PyErr_Clear()` is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

Every failing `malloc()` call must be turned into an exception — the direct caller of `malloc()` (or `realloc()`) must call `PyErr_NoMemory()` and return a failure indicator itself. All the object-creating functions (for example, `PyInt_FromLong()`) already do this, so this note is only relevant to those who call `malloc()` directly.

Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and `-1` for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don't use `PyExc_TypeError` to mean that a file couldn't be opened (that should probably be `PyExc_IOError`). If something's wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

```
static PyObject *SpamError;
```

and initialize it in your module's initialization function (**`initspam()`**) with an exception object (leaving out the error checking for now):

```
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
}
```

Note that the Python name for the exception object is **`spam.error`**. The `PyErr_NewException()` function may create a class with the base class being `Exception` (unless another class is passed in instead of *NULL*), described in Built-in Exceptions.

Note also that the **`SpamError`** variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing **`SpamError`** to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of `PyMODINIT_FUNC` as a function return type later in this sample.

The **`spam.error`** exception can be raised in your extension module using a call to `PyErr_SetString()` as shown below:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

1.3. Back to the Example

Going back to our example function, you should now be able to understand this statement:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns *NULL* (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as `const char *command`).

The next statement is a call to the Unix function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()`: it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```
return Py_BuildValue("i", sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning `void`), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the `Py_RETURN_NONE` macro):

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a *NULL* pointer, which means “error” in most contexts, as we have seen.

1.4. The Module’s Method Table and Initialization

Function

I promised to show how **spam_system()** is called from Python programs. First, we need to list its name and address in a "method table":

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

Note the third entry (`METH_VARARGS`). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be `METH_VARARGS` or `METH_VARARGS | METH_KEYWORDS`; a value of `0` means that an obsolete variant of **PyArg_ParseTuple()** is used.

When using only `METH_VARARGS`, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via **PyArg_ParseTuple()**; more information on this function is provided below.

The **METH_KEYWORDS** bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use **PyArg_ParseTupleAndKeywords()** to parse the arguments to such a function.

The method table must be passed to the interpreter in the module's initialization function. The initialization function must be named **initname()**, where *name* is the name of the module, and should be the only non-`static` item defined in the module file:

```
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

Note that PyMODINIT_FUNC declares the function as `void` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

When the Python program imports module **spam** for the first time, **initspam()** is called. (See below for comments about embedding Python.) It calls **Py_InitModule()**, which creates a "module object" (which is inserted in the dictionary `sys.modules` under the key `"spam"`), and inserts built-in function objects into the newly created module based upon the table (an array of **PyMethodDef** structures) that was passed as its second argument.

`Py_InitModule()` returns a pointer to the module object that it creates (which is unused here). It may abort with a fatal error for certain errors, or return *NULL* if the module could not be initialized satisfactorily.

When embedding Python, the **`initspam()`** function is not called automatically unless there's an entry in the **`_PyImport_Inittab`** table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling **`initspam()`** after the call to `Py_Initialize()`:

```
int
main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();

    ...
```

An example may be found in the file `Demo/embed/demo.c` in the Python source distribution.

Note: Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a **`fork()`** without an intervening **`exec()`**) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures. Note also that the `reload()` function can be used with extension modules, and will call the module initialization function (**`initspam()`** in the example), but will not load the module again if it was loaded from a dynamically loadable object file (`.so` on Unix, `.dll` on Windows).

A more substantial example module is included in the Python source distribution as `Modules/xxmodule.c`. This file may be used as a template or simply read as an example.

1.5. Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter Building C and C++ Extensions with distutils) and additional information that pertains only to building on Windows (chapter Building C and C++ Extensions on Windows) for more information about this.

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the `Modules/` subdirectory, but then you must first rebuild `Makefile` there by running '**make** Makefile'. (This is necessary each time you change the `Setup` file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```

1.6. Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called "callback" functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won't dwell on how to call the Python parser with a particular string as input — if you're interested, have a look at the implementation of the `-c` command line option in `Modules/main.c` from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

This function must be registered with the interpreter using the `METH_VARARGS` flag; this is described in section The Module's Method Table and Initialization Function. The `PyArg_ParseTuple()` function and its arguments are documented in section Extracting Parameters in Extension Functions.

The macros `Py_XINCREF()` and `Py_XDECREF()` increment/decrement the reference count of an object and are safe in the presence of *NULL* pointers (but note that *temp* will not be *NULL* in this context). More info on them in section Reference Counts.

Later, when it is time to call the function, you call the C function `PyObject_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in NULL, or an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` returns a Python object pointer: this is the return value of the Python function. `PyObject_CallObject()` is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t *NULL*. If it is, the Python function terminated by raising an exception. If the C code that called `PyObject_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling `PyErr_Clear()`. For example:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

1.7. Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

```
int PyArg_ParseTuple(PyObject *arg, char *format, ...);
```

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained in Parsing arguments and building values in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

Some example calls:

```
int ok;
int i, j;
long k, l;
const char *s;
int size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
    /* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
    /* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
    /* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* A pair of ints and a string, whose size is also returned */
    /* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

1.8. Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                char *format, char *kwlist[], ...);
```

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a *NULL*-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

Note: Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_INCREF(Py_None);

    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};
```

```
void
initkeywdarg(void)
{
  /* Create the module and add the functions */
  Py_InitModule("keywdarg", keywdarg_methods);
}
```

1.9. Building Arbitrary Values

This function is the counterpart to **`PyArg_ParseTuple()`**. It is declared as follows:

```
PyObject *Py_BuildValue(char *format, ...);
```

It recognizes a set of format units similar to the ones recognized by **`PyArg_ParseTuple()`**, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with `PyArg_ParseTuple()`: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

```
Py_BuildValue("")                        None
Py_BuildValue("i", 123)                  123
Py_BuildValue("iii", 123, 456, 789)      (123, 456, 789)
Py_BuildValue("s", "hello")              'hello'
Py_BuildValue("ss", "hello", "world")    ('hello', 'world')
Py_BuildValue("s#", "hello", 4)          'hell'
Py_BuildValue("()")                      ()
Py_BuildValue("(i)", 123)                (123,)
Py_BuildValue("(ii)", 123, 456)          (123, 456)
Py_BuildValue("(i,i)", 123, 456)         (123, 456)
Py_BuildValue("[i,i]", 123, 456)         [123, 456]
Py_BuildValue("{s:i,s:i}",
              "abc", 123, "def", 456)    {'abc': 123, 'def': 456}
Py_BuildValue("((ii)(ii)) (ii)",
              1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5, 6))
```

1.10. Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions **`malloc()`** and **`free()`**. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with **`malloc()`** should eventually be returned to the pool of available memory by exactly one call to **`free()`**. It is important to call **`free()`** at the right time. If a block's address is forgotten but **`free()`** is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls **`free()`** for a block and then continues to use the block, it creates a conflict with re-use of the block through another **`malloc()`** call. This is called *using freed memory*. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It’s easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it’s important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of “automatic” to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn’t need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we’ll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (`__del__()` methods). When there are such finalizers, the detector exposes the cycles through the `gc` module (specifically, the `garbage` variable in that module). The `gc` module also exposes a way to run the detector (the `collect()`

function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the `--without-cycle-gc` option to the **configure** script on Unix platforms (including Mac OS X) or by removing the definition of `WITH_CYCLE_GC` in the `pyconfig.h` header on other platforms. If the cycle detector is disabled in this way, the `gc` module will not be available.

1.10.1. Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly — rather, it makes a call through a function pointer in the object's *type object*. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let's first introduce some terms. Nobody "owns" an object; however, you can *own a reference* to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to *borrow* [2] a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely. [3]

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

1.10.2. Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyInt_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyInt_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them — even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership — they are "normal.")

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred from the function to its caller.

1.10.3. Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value `0`, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in **`bug()`**? You bet! Assuming that the list passed into **`bug()`** is accessible to the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4. NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them *NULL* pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return *NULL* only to indicate that an exception occurred. The reason for not testing for *NULL* arguments is that functions often pass the objects they receive on to other function — if each function were to test for *NULL*, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for *NULL* only at the "source:" when a pointer that may be *NULL* is received, for example, from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for *NULL* pointers — however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`Pytype_Check()`) don't check for *NULL* pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with *NULL* checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never *NULL* — in fact it guarantees that it is always a tuple. [4]

It is a severe error to ever let a *NULL* pointer "escape" to the Python user.

1.11. Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).

1.12. Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type "collection" which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them `static`, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared `static`, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section The Module's Method Table and Initialization Function). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (`void *`). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

Whichever method you choose, it's important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (`const char *`); you're permitted to pass in a *NULL* name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule's name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C API.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of `void` pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section A Simple Example. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding "spam" to every command). This function `PySpam_System()` is also exported to other extension modules.

The function `PySpam_System()` is a plain C function, declared `static` like everything else:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function `spam_system()` is modified in a trivial way:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return Py_BuildValue("i", sts);
}
```

In the beginning of the module, right after the line

```
#include "Python.h"
```

two more lines must be added:

```
#define SPAM_MODULE
#include "spammodule.h"
```

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module's initialization function must take care of initializing the C API pointer array:

```
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

    if (c_api_object != NULL)
```

```
        PyModule_AddObject(m, "_C_API", c_api_object);
}
```

Note that `PySpam_API` is declared `static`; otherwise the pointer array would disappear when **`initspam()`** terminates!

The bulk of the work is in the header file `spammodule.h`, which looks like this:

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
 (*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */
```

All that a client module must do in order to have access to the function **`PySpam_System()`** is to call the function (or rather macro) **`import_spam()`** in its initialization function:

```
PyMODINIT_FUNC
initclient(void)
{
    PyObject *m;

    m = Py_InitModule("client", ClientMethods);
    if (m == NULL)
        return;
    if (import_spam() < 0)
        return;
    /* additional initialization can happen here */
}
```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section Capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

Footnotes

[1] An interface for this function already exists in the standard module `os` — it was chosen as a simple and straightforward example.

[2] The metaphor of "borrowing" a reference is not completely correct: the owner still has a copy of the reference.

[3] Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

[4] These guarantees don't hold when you use the "old" style calling convention — this is still found in much existing code.

Achevé d'imprimer le 30 septembre 2016 sur les presses

de Learning Tree International France

ISBN : 978-2-7525-1109-6

EDUCATION IS OUR BUSINESS®