Parsing and Writing XML with Python

Luis Meneses
Welcome to DHSI 2019!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered the day before our courses begin), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes - your instructors will let you know - or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

Do check in there first if you need anything that’s not in this coursepak.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2019”.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help....
DHSI Wi-Fi

Network name: DHSI
Passkey: dhsi2019
The 2019 schedule is just taking shape nicely! A very few things to confirm, add, etc, still but this is the place to be to find out what is happening when / where ...

---

Sunday, 2 June 2019 [DHSI Registration + Suggested Outings]

If you're here a day or two before we begin, or staying a day or two afterwards, here are a few ideas of things you might consider doing ....

Suggested Outing 1, Botanical Beach (self-organised; car needed)

A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach; we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful undersea life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.

Suggested Outing 2, Butchart Gardens (self-organised)

A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Peninsula. About an hour there by public bus from UVic, or 30 minutes by car.

Suggested Outing 3, Salt Spring Island (self-organised; a full day, car/bus + ferry combo)

Why not take a day to explore and celebrate the funky, laid back, Canadian gulf island lifestyle on Salt Spring Island. Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever ....

Suggested Outing 4, Paddling Victoria's Inner Harbour (self-organised)

A shorter time, seeing Victoria's beautiful city centre from the waterways that initially inspired its foundation. A great choice if the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.

And more!

Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hungry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker's, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!

---

9:00 to 4:00

Early Class Meeting: 4. [Foundations] DH For Department Chairs and Deans (David Strong Building C124, Classroom)

Further details are available from instructors in mid May to those registered in the class. Registration materials will be available in the classroom.

---

3:00 to 5:00

DHSI Registration (MacLaurin Building, Room A100)

After registration, many will wander to Cadboro Bay and the pub at Smuggler's Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub OR even into the city for a nice meal.

---

Monday, 3 June 2019

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Jannaya Friggstad Jensen.

---

7:45 to 8:15

Last-minute Registration (MacLaurin Building, Room A100)
Welcome, Orientation, and Instructor Overview (MacLaurin A144)
- Welcome to the Territory
- Welcome to DHSI: Ray Siemens, Alyssa Arbuckle
- Welcome from UVic: Jonathan Bengtson (University Librarian), Alexandra D'Arcy (Associate Dean Research, Humanities)

8:30 to 10:00

Classes in Session (click for details and locations)

- 1. [Foundations] Digitisation Fundamentals and their Application (Clearihue A103, Lab)
- 2. [Foundations] Introduction to Computation for Literary Criticism (Clearihue A102, Lab)
- 4. [Foundations] DH For Department Chairs and Deans (David Strong Building C124, Classroom)
- 5. [Foundations] Developing a Digital Project (With Omeka) (Clearihue A031, Lab)

10:15 to Noon

- 8. [Foundations] Fundamentals of Programming/Coding for Human(s)ists (Clearihue A108, Lab)
- 9. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab)
- 10. Sound and Digital Humanities (Cornett A120, Classroom)
- 11. Critical Pedagogy and Digital Praxis in the Humanities (Clearihue D132, Classroom)
- 12. Digital Humanities for Japanese Culture: Resources and Methods (McPherson Library A003, Classroom)
- 14. Retro Machines & Media (McPherson Library 129, Classroom)
- 15. Geographical Information Systems in the Digital Humanities (Clearihue A105, Lab)
- 16. Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images (Cornett A121, Classroom)
- 17. Web APIs with Python (Human and Social Development A170, Lab)
- 18. Ethical Data Visualization: Taming Treacherous Data (Cornett A128, Classroom)
- 19. Linked Open Data and the Semantic Web (Cornett A132, Classroom)
- 20. Palpability and Wearable Computing (McPherson Library A025, Classroom)
- 22. Information Security for Digital Researchers (David Strong Building C114, Classroom)

12:15 to 1:15

Lunch break / Unconference Coordination Session (MacLaurin A144)
(Grab a sandwich and come on down!)

Discussion topics, scheduling, and room assignments from among all DHSI rooms will be handled at this meeting.

1:30 to 4:00

Classes in Session

Institute Lecture: Jacqueline Wernimont (Dartmouth C): "Sex and Numbers: Pleasure, Reproduction, and Digital Biopower"
Chair: Anne Cong-Huyen (U Michigan)
(MacLaurin A144)

4:10 to 5:00

Abstract: Drawing from Numbered Lives (MIT 2018), this talk will consider a long history of sex-number entanglement in Anglo-American Cultures. Drawing on historical and contemporary objects and practices, Wernimont will ask "in what ways do theories of biopower, critical gender and critical race studies, and media studies" suggest that we can understand this set of entanglements and their impacts. NB: While relevant, this talk will not include discussions of sexual trauma or violence. It will include frank discussion of sex acts and various ways of translating sexual behavior into numbers.

5:00 to 6:00

Opening Reception (University Club)

Tuesday, 4 June 2019

9:00 to Noon

Classes in Session

12:15 to 1:15

Lunch break / Unconference

"Mystery" Lunches

1:30 to 4:00

Classes in Session

DHSI Conference and Colloquium Lightning Talk Session 1 (MacLaurin A144)
4:15 to 5:15
Chair: Kim O'Donnell (Simon Fraser U)
- Marion Grant (Ryerson U), “Visualizing Networks: Yellow Nineties Print and Performance”
- Kristen Starkowski (Princeton U), “Mapping Minor Characters: Quantifying and Visualizing Character Space in Dickens’s Novels and in their Adaptations”
- Leah Henrickson (Loughborough U), “Who is the author of the computer-generated text?”

6:00 to 8:00
DHSI Newcomer’s Gathering (Grad House Restaurant, Graduate Student Centre)
Come down, buy meal and a beverage, and make some new friends!

9:00 to Noon
Classes in Session

Wednesday, 5 June 2019

12:15 to 1:15
This presentation introduces Manifold Scholarship, a Mellon-funded digital publishing platform developed by the CUNY Graduate Center, The University of Minnesota Press, and Cast Iron Coding. Manifold allows you to create beautiful, dynamic open access projects that can include text, images, video, embedded resources, and social annotation. We will provide an overview of Manifold and demonstrate how faculty, students and staff in the digital humanities can use Manifold to publish open access scholarly works, conduct and participate in peer review, and create custom edited versions of public domain course texts and OER.

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 2 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)
- Catherine Ryu (Michigan State U), “Tone Perfect: Developing a Multimodal Audio Database for Mandarin Chinese as an Open Source”
- Jessica Linzel (Brock U), “The Shopkeeper Aristocracy: Mapping Trade Networks in Colonial Niagara”
- Kirsten Painter (U Washington), “From Bogatyrs to Bread: Digitization & Online Exhibition of Rare Russian Children's Books at the U Washington”

6:00 to 7:00
"Half Way There!" [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)
Bring your DHSI nametag and enjoy your first tipple on us! [A great opportunity for an interest group meet-up ....]

Thursday, 6 June 2019

9:00 to Noon
Classes in Session

12:15 to 1:15
"Mystery" Lunches

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 3 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)
- Colleen Kolba (U South Florida), “What Comics can Teach our Students about Multimodal Literacy”
- Trish Baer (ETCL; U Victoria), “Preserving Digital Legacies: Archived Websites and Digital Discoverability”
- Suchismita Dutta (U Miami), “The Importance of Archival Transcription for Genre Building”
- Jeffrey Lawler (California State U, Long Beach), “Twining our way through the Past: Video Game Authoring as History Pedagogy”
Friday, 7 June 2019 [DHSI; ADHO Pedagogy SIG Conference Opening]

9:00 to Noon  Classes in Session

12:15 to 1:15  Lunch Reception / Course E-Exhibits (MacLaurin A100)

1:30 to 1:50  Remarks, A Week in Review (MacLaurin A144)

2:00 to 3:00  Joint Institute Lecture (DHSI and ADHO Pedagogy SIG Conference):
Matt Gold (CUNY Graduate Center and Association for Computers and the Humanities); “Thinking Through DH: Proposals for Digital Humanities Pedagogy”
Chair: Diane Jakacki (Bucknell U) (MacLaurin A144)

Abstract: How do we teach digital humanities, and how should DH be taught? What, indeed, should we teach when we teach DH? This talk will present a proposal for grounding digital humanities pedagogical practice in the research interests of our students and the epistemological foundations of our methods rather than through an approach grounded more central in data and methods.

3:30 to 5:00  Joint Reception: DHSI and ADHO Pedagogy SIG Conference (University Club)
E-Poetry Event (Chris Tanasescu)
Watch this space for details, including how to participate!

Saturday, 8 June 2019 [Conference, Colloquium, and Workshop Sessions]

8:00 to 9:00  Conference / Workshop Registration (MacLaurin A100)

The day's events are included with your DHSI registration. If you're not registered in DHSI, you're very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We'll have a nametag waiting for you!

Coffee, Tea, &c?  Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

9:00 to 4:00  DHSI Conference and Colloquium Sessions
ADHO Pedagogy SIG Conference Sessions
Right2Left Workshop Sessions

9:00 to 4:00  All Day DHSI Workshop Session (click for workshop details and free registration for DHSI participants)
55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] (David Strong Building C124, Classroom)

9:00 to 9:10  Informal Greetings, Room Set-up (Lobby, outside Hickman 105)

Session 1
DHSI Colloquium and Conference (Hickman 105)
Digital Humanities & Literature, Chair: Kim O'Donnell (Simon Fraser U)
- Youngmin Kim (Dongguk U), “Transdiscursivity in the Convergence of Digital Humanities and World Literature”
- Caroline Winter (U Victoria), “Digitizing Adam Smith’s Literary Library”
- Kaitlyn Fralick (U Victoria); Kailey Fukushima (U Victoria); Sarah Karlson (U Victoria), “Victorian Poetry
9:10 to 10:30

**ADHO Pedagogy SIG Conference (Hickman 110)**

Chair: Katherine Faull (Bucknell U)

- Aaron Tucker and Nada Savicevic (Ryerson U), “Write Here, Right Now: An Open Source eTextbook for the Flipped Classroom”
- Heather McAlpine (U Fraser Valley), “Digital Meters: Using Text Encoding to Teach Literature in the Undergraduate Classroom”
- Tiina H. Airaksinen (U Helsinki), “Digital Humanities in Cultural Studies: Creating a MOOC course for University Students and A-Level Students”

Right2Left Workshop (Hickman 116)


10:30 to 10:40

**Break**

10:40 to Noon

**Session 2**

DHSI Colloquium and Conference (Hickman 105)

Digital Humanities & Society, Chair: Eleanor Reed (Hastings C)

- Joel Zapata (Southern Methodist U), “Uncovering the Southern Plains’ Mexican American Civil Rights Movement”
- Brendan Mackie (U California, Berkeley), “Visualizing Long-Term Cultural Change: An Example From The Birth of Civil Society”

**ADHO Pedagogy SIG Conference (Hickman 110)**

Chair: Laura Estill (St Francis Xavier U)

- Jane Jackson (Chinese U of Hong Kong), “Interrogating digital spaces for intercultural meaning-making”
- Christopher Church, Katherine Hepworth (U Nevada, Reno), “We’re STEAMed! A call for balancing technical instruction and disciplinary content in the digital humanities”

Right2Left Workshop (Hickman 116)

- Edward “Eddie” Surman (Claremont Graduate U), “Qualitative Digital Text Analysis and #Right2Left Languages: A Demonstration of Atlas.ti using the Hebrew Bible”

Noon to 1:10

**Lunch** (We recommend Mystic Market on weekends!)

1:10 to 2:30

**Session 3**

DHSI Colloquium and Conference (Hickman 105)

Digital Humanities & Community, Chair: Claire Carlin (U Victoria)

- Pia Russel (U Victoria); Emily Stremel (U Victoria), “Mentorship and disability: Supporting disabled employees in digital humanities”
- Amy Lueck (Santa Clara U), “Virtually Emplacing Indigenous Memory”
- Md. Shehabul Alam (National U Bangladesh), “Integrating Library Service with Union Information and Service Center: A Joint Initiative towards Digital Bangladesh”
- Veronica Gomez (Instituto de Humanidades y Ciencias Sociales (IHuCSo) - UNL-CONICET), “Latin American E-literature and Location: The Nation Revisited in Electronic Literature Organization (ELO)”

**ADHO Pedagogy SIG Conference (Hickman 110)**

Chair: Chris Tănăsescu (UC Louvain)

- Laura Estill (St Francis Xavier U), “One Assignment, Three Ways: Assessing DH Projects in a Literature Course”
- Francesca Giannetti (Rutgers U, New Brunswick), “So near while apart: Correspondence Editions as Critical Library Pedagogy and Digital Humanities Methodology”

**Right2Left Workshop (Hickman 116)**

- Najla Jarkas (American U Beirut) and David Joseph Wrisley (NYU Abu Dhabi), “RTL Software Localization and Digital Humanities: the Case Study of Translating Voyant Tools into Arabic”
### Session 4

DHSI Colloquium and Conference *(Hickman 105)*
- Ashleigh Casserme-Stanfield (U Chicago), “Sonifying Hamlet and Reading the Room”

ADHO Pedagogy SIG Conference *(Hickman 110)*
Chair: Aaron Tucker (Ryerson U)
- Youngmin Kim (Dongguk U), “Teaching Digital Humanities and World Literature in Class”
- Alice Fleerackers, Juan Pablo Alperin, Esteban Morales, Remi Kalir (Simon Fraser U, U Colorado Denver), “Online annotations in the classroom: How, why, and what do students learn from annotating course material?”
- Andie Silva (York C and Graduate Center, CUNY), “Keeping it Local: Undergraduate DH as Feminist Practice”

Right2Left Workshop *(Hickman 116)*
- Joanna Byszuk (Institute of Polish Language, Polish Academy of Sciences, Warsaw/Computational Stylistics Group) and Alexey Khismatulin (Institute of Oriental Manuscripts, Russian Academy of Sciences, Saint Petersburg), "Attribution of Authorship for Medieval Persian Quasidas with Stylometry”
- Ilan Benattar (New York U), "#Right2Left Biblical Translations in Jewish Textual History: Case Studies in Judeo-Arabic and Judeo-Spanish”

### Sunday, 9 June 2019 [Workshop Sessions]

**8:00 to 5:00**
The day’s events are included with your DHSI registration. If you’re not registered in DHSI, you’re very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We’ll have a nametag waiting for you!

**Coffee, Tea, &c?**
Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

**9:00 to 4:00**

<table>
<thead>
<tr>
<th>All Day Workshop Sessions (click for workshop details and free registration for DHSI participants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] <em>(David Strong Building C124, Classroom)</em></td>
</tr>
<tr>
<td>56. Pedagogy of the Digitally Oppressed: Anti-Colonial DH Methods and Praxis [9 June; All Day] <em>(Hickman 116, Classroom)</em></td>
</tr>
<tr>
<td>57. Natural Language Processing and Network Coding Apps for Text &amp; Textual Corpus Analysis in the Humanities [9 June; All Day] <em>(David Strong Building C114, Classroom)</em></td>
</tr>
</tbody>
</table>

**9:00 to Noon**

<table>
<thead>
<tr>
<th>AM Workshop Sessions (click for workshop details and free registration for DHSI participants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>59. 3D Visualization for the Humanities [9 June; AM] <em>(Cornett A229, Classroom)</em></td>
</tr>
<tr>
<td>60. It’s All Relational: AbTeC’s Indigenous Video Game Workshops as Storytelling Praxis [9 June; AM] <em>(Cornett A121, Classroom)</em></td>
</tr>
<tr>
<td>61. Spatial DH: De-Colonizing Cultural Territories Online [9 June; AM] <em>(Clearihue D130, Classroom)</em></td>
</tr>
<tr>
<td>63. Creating a CV for Digital Humanities Makers [9 June; AM] <em>(David Strong Building C109, Classroom)</em></td>
</tr>
</tbody>
</table>

**Noon to 1:00**
Lunch (We recommend Mystic Market on weekends!)

**1:00 to 4:00**

<table>
<thead>
<tr>
<th>PM Workshop Sessions (click for workshop details and free registration for DHSI participants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65. Indigenous Futurities in the Classroom and Beyond [9 June; PM] <em>(Cornett A121, Classroom)</em></td>
</tr>
<tr>
<td>66. DHSI Knits: History of Textiles and Technology [9 June; PM] <em>(Cornett A121, Classroom)</em></td>
</tr>
<tr>
<td>68. Linked Open Datafication for Humanities Scholars [9 June; PM] <em>(McPherson Library A003, Classroom)</em></td>
</tr>
<tr>
<td>69. Stylo - WYSIWYM Text Editor for Humanities Scholars [9 June; PM] <em>(McPherson Library A025, Classroom)</em></td>
</tr>
</tbody>
</table>

After the day, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter’s Pub OR even into the city for a bite to eat.
Your hosts for the week are Ray Siemens and Jannaya Friggstad Jensen.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:45 to 8:15</td>
<td>DHSI Last-minute Registration (MacLaurin A100)</td>
</tr>
<tr>
<td>8:30 to 10:00</td>
<td>Welcome, Orientation, and Instructor Overview (MacLaurin A144)</td>
</tr>
<tr>
<td>10:15 to Noon</td>
<td>Classes in Session (click for details and locations)</td>
</tr>
<tr>
<td></td>
<td>29. [Foundations] Understanding The Predigital Book: Technologies of Inscription (McPherson Library A003, Classroom)</td>
</tr>
<tr>
<td></td>
<td>30. [Foundations] Databases for Digital Humanists (McPherson Library 210, Classroom)</td>
</tr>
<tr>
<td></td>
<td>32. [Foundations] Music Encoding Fundamentals and their Applications (Clearihue A030, Lab)</td>
</tr>
<tr>
<td></td>
<td>33. Digital Storytelling (Cornett A120, Classroom)</td>
</tr>
<tr>
<td></td>
<td>34. Text Mapping as Modelling (Clearihue D131, Classroom)</td>
</tr>
<tr>
<td></td>
<td>35. Stylometry with R: Computer-Assisted Analysis of Literary Texts (Clearihue A102, Lab)</td>
</tr>
<tr>
<td></td>
<td>36. Open Access and Open Social Scholarship (Clearihue D130, Classroom)</td>
</tr>
<tr>
<td></td>
<td>37. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (Cornett A229, Classroom)</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference Coordination Session (MacLaurin A144)</td>
</tr>
<tr>
<td></td>
<td>(Grab a sandwich and come on down!)</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Institute Lecture: Angel David Nieves (San Diego State U): &quot;3D Mapping and Forensic Traces of Testimony: Documenting Apartheid-Era Crimes Through the Digital Humanities&quot; Chair: Constante Crompton (U Ottawa) (MacLaurin A144)</td>
</tr>
<tr>
<td></td>
<td>Abstract: In 1989 the killing of a queer, 14-year-old youth in Winnie Mandela's house named Stompie Seipei (an event that few in South Africa are willing to recall, let alone discuss, in any detail) -- is perhaps one of the most glaring examples where the queer and activist community was suppressed or erased from anti-apartheid/liberation histories. Digital humanities may actually help both reconstruct and recover a history that is still very early in the telling, despite what is commonly believed about the liberation struggle and the contributions of queer activists in the dismantling of apartheid. Perhaps it could explain why a youth such as Seipei was killed -- or at the very least, provide a more complex and messy narrative that permits one to know more how the history of queer anti-apartheid activists was suppressed. This talk outlines a methodology for &quot;messy thinking and writing&quot; in the digital humanities that -- through a queer and feminist intersectional framework -- permits a more complex layering of oral histories and 3D historical reconstructions.</td>
</tr>
<tr>
<td>5:00 to 6:00</td>
<td>Reception (University Club)</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
</tr>
<tr>
<td>Time</td>
<td>Event</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Conference and Colloquium Lightning Talk Session 4 (MacLaurin A144)&lt;br&gt;Chair: Lindsey Seatter (U Victoria)&lt;br&gt;• Ashley Caranto Morford (U Toronto); Kush Patel (U Michigan); Arun Jacob (McMaster U), “#OurDHLs anti-colonial: Questions and challenges in dismantling colonial influences in digital humanities pedagogy”&lt;br&gt;• Julia King (U Bergen), “Developing Network Visualizations of Syon Abbey's Books, 1415-1539”&lt;br&gt;• Luis Meneses (ETCL; U Victoria), “Identifying Changes in the Political Environment in Ecuador”&lt;br&gt;• Alicia Brown (Texas Christian U), “Digital Cartography of the Ancient World”&lt;br&gt;• Laura Horak (Carleton U), “Building the Transgender Media Portal”&lt;br&gt;• Andrew Boyles Peterson (Michigan State U), “Last Mile Tracking: Implications of Rental Scooter Surveillance”</td>
</tr>
<tr>
<td>6:00 to 8:00</td>
<td>DHSI Newcomer's Gathering (Grad House Restaurant, Graduate Student Centre)&lt;br&gt;Come down, buy meal and a beverage, and make some new friends!</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference&lt;br&gt;“Mystery” Lunches&lt;br&gt;Presentation: An Introduction Jupyter Notebooks for Researchers (MacLaurin A144)&lt;br&gt;This presentation introduces Jupyter Notebooks for researchers, via a partnership between Compute Canada and the Pacific Institute for the Mathematical Sciences (PIMS) including a large number of Canadian institutions. Read more <a href="#">here</a>. Presenting is James Colliander, PIMS Director and team.</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Institute Lecture: Karina van Dalen-Oskam (Huygens Institute and U Amsterdam; Alliance of Digital Humanities Organizations): “The Riddle of Literary Quality: Some Answers”&lt;br&gt;Chair: Aaron Mauro (Penn State, Behrend C) (MacLaurin A144)&lt;br&gt;Abstract: What is literature, and can you measure it? That is the key question of the project The Riddle of Literary Quality. “The Riddle” is a research project of the Huygens Institute for the History of the Netherlands (Amsterdam) in collaboration with the Fryeke Akademy (Leeuwarden) and the Institute for Logic, Language and Computation (University of Amsterdam). The Riddle combines computational analysis of writing style with the results of a large online survey of readers, completed by almost 14,000 participants. In my talk, I will go into</td>
</tr>
</tbody>
</table>

**Wednesday, 12 June 2019**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference&lt;br&gt;“Mystery” Lunches&lt;br&gt;[Instructor lunch meeting]</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Institute Lecture: Karina van Dalen-Oskam (Huygens Institute and U Amsterdam; Alliance of Digital Humanities Organizations): “The Riddle of Literary Quality: Some Answers”&lt;br&gt;Chair: Aaron Mauro (Penn State, Behrend C) (MacLaurin A144)&lt;br&gt;Abstract: What is literature, and can you measure it? That is the key question of the project The Riddle of Literary Quality. “The Riddle” is a research project of the Huygens Institute for the History of the Netherlands (Amsterdam) in collaboration with the Fryeke Akademy (Leeuwarden) and the Institute for Logic, Language and Computation (University of Amsterdam). The Riddle combines computational analysis of writing style with the results of a large online survey of readers, completed by almost 14,000 participants. In my talk, I will go into</td>
</tr>
</tbody>
</table>

**Thursday, 13 June 2019**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference&lt;br&gt;“Mystery” Lunches&lt;br&gt;[Instructor lunch meeting]</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Institute Lecture: Karina van Dalen-Oskam (Huygens Institute and U Amsterdam; Alliance of Digital Humanities Organizations): “The Riddle of Literary Quality: Some Answers”&lt;br&gt;Chair: Aaron Mauro (Penn State, Behrend C) (MacLaurin A144)&lt;br&gt;Abstract: What is literature, and can you measure it? That is the key question of the project The Riddle of Literary Quality. “The Riddle” is a research project of the Huygens Institute for the History of the Netherlands (Amsterdam) in collaboration with the Fryeke Akademy (Leeuwarden) and the Institute for Logic, Language and Computation (University of Amsterdam). The Riddle combines computational analysis of writing style with the results of a large online survey of readers, completed by almost 14,000 participants. In my talk, I will go into</td>
</tr>
</tbody>
</table>
some of the main results of the project.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch Reception / Course E-Exhibits (MacLaurin A100)</td>
</tr>
<tr>
<td>1:30 to 2:00</td>
<td>Closing, DHSI in Review (MacLaurin A144)</td>
</tr>
</tbody>
</table>

**Contact info:**  
institut@uvic.ca  P: 250-472-5401  F: 250-472-5681
Introduction

XML has become a widely adopted format for data interchange -mainly due its simplicity and adaptability for general use. Nowadays it is a common practice to parse and transform XML with XSLTs. However, other programming languages present alternatives to this task that with a different workflow over XSLTs. In this workshop, we will outline the essential building blocks for parsing and writing XML using Python and Open Source Libraries such as Beautiful Soup and lxml.

The scope of this workshop is not limited to only XML. We will also present examples involving the parsing of documents in other forms of SGML derived-markup -such as TEI and HTML.

This is a hands-on course. Consider this offering in complement with, and / or to be built on by: Text Encoding Fundamentals and their Application; Text Analysis with Python and the Natural Language ToolKit; and more!
Draft Course Schedule (actual implementation may vary)

<table>
<thead>
<tr>
<th>Day</th>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1   | Introduction     | • Introduction  
• Why are we doing this? (XSLTs work just fine)?  
• Python libraries: BeautifulSoup and lxml  
• Environment check and getting started  
• Python: V2 or V3? Unicode does matter  
• Time for hands-on exercises |
| 2   | Parsing XML      | • Loading files  
• Using BeautifulSoup (batteries are included)  
• Manipulating XML: search, replace, append and modify  
• Speed and memory considerations: Working with large files  
• Time for hands-on exercises |
| 3   | Writing XML      | • lxml and the elementTree API  
• Unicode support  
• Time for hands-on exercises or to work on individual projects |
| 4   | Hands on work    | • Web scraping: frameworks and tools  
• Parsing and converting to/from other formats: XML and JSON  
• Time for hands-on exercises or to work on individual projects |
| 5   | Wrap up          | • Show and Tell session setup  
• Conclusions  
• Future work and what is next? |

Useful Resources

Python: [https://www.python.org](https://www.python.org)  
Dive into Python 3: [https://www.diveinto.org/python3/](https://www.diveinto.org/python3/)  
Beautiful Soup: [https://www.crummy.com/software/BeautifulSoup/](https://www.crummy.com/software/BeautifulSoup/)  
Beautiful Soup Documentation: [https://www.crummy.com/software/BeautifulSoup/bs4/doc/](https://www.crummy.com/software/BeautifulSoup/bs4/doc/)  
Lxml: [https://lxml.de](https://lxml.de)
Basics of Python
Parsing and Writing XML with Python

Dr. Luis Meneses

Electronic Textual Cultures Lab
University of Victoria

Adapted from CSCE 110, Fall 2016 - Texas AM University
Python was developed in 1989 by Guido van Rossum in the Netherlands.

Python was released for public distribution in early 1991.

How did Python begin?

- van Rossum was having a hard time getting the job done with the existing tools available.
- He envisioned that there was an easier way to get things done.

While Python has been around for over 20 years, Python is still relatively new to general software development. However, it has a lot of support from the community and new users are finding themselves programming in Python (and loving it) everyday.
Since you are learning a new language, it is very important that you experiment with the material.

Be curious about trying different things.

Don’t be afraid to make a mistake or crash your computer. That’s how we learn best.
Now, we are ready to start programming!

- We will take it slow and easy in the beginning. I want you to get comfortable getting acquainted with the basics.
- Following along in class is not enough. **You must type in the programs on your computer.**
  - It’s the best way to see how a program works or executes.
  - It’s how you learn to correct your mistakes (programming errors or bugs).
  - Writing and reading code is the only way to learn how to program.
Python's `print` statement is the tool for displaying program output. Type the following in the Python shell. The Python shell is represented by `>>>`.

Python shell example.

```python
>>> print ('Hello, World!')
Hello, World!
```
1 Type the following text into a text editor.

```python
print ('hello, world!')
```

2 Save the above text in a file called `hello.py`.

3 Run your hello.py program.
Our Second Program

Type the following text into a text editor. Save the file as hello2.py.

```python
name = input('Please enter your name: ')
print ('Hello', name, 'good to see you!')
```
Write a program that asks for a person’s first name, then middle, and then last. Afterwards, it should greet the person using their full name.

Example #1.

First name: Maria
Middle name: Elena
Last name: Perez

Howdy Maria Elena Perez!

Example #2.

First name: Eric
Middle name: Joseph
Last name: Smith

Howdy Eric Joseph Smith!
Reading Exercise: What is the output of this program?

```python
straightline.py

1 x = 3
2 y = 4
3 z = x + y
4 z = z + 1
5 x = y
6 y = 5
7 print (x)
8 print (y)
9 print (z)
```

Execution order: 1, 2, 3, 4, 5, 6, 7, 8, 9
Ask the user to enter a number. Print out the square of the number. Print the result in a full sentence that ends in a period. Sample output is shown below.

Example #1.
Enter a number: 5
The square of 5 is 25.

Example #2.
Enter a number: 901
The square of 901 is 811801.
Using the Python Shell as a Calculator

Python shell examples.

```python
>>> 3 + 4
7
>>> 15 / 3
5
>>> 12 * 10 + 4
124
>>> 4.2 + 10 / 5
6.2
>>> 9 / 2
4.5
>>> 9.0 / 2
4.5
>>> 9.0 // 2
4.0
>>> 9 // 2
4
```
Python supports five basic numerical types. The most common numerical types we will use are int, bool, and float.

- **int** (signed integers)
- **bool** (Boolean values)
- **float** (floating point real numbers)

**Examples**

- **int:** 100, 200, -437
- **bool:** True, False
- **float:** 3.456, -33.55, 34.1
Mathematical, Comparison, and Conjunctive Operators

- **Mathematical operators:** +, −, *, /, //, %, **
  - addition (+)
  - subtraction (−)
  - multiplication (*)
  - division (/)
  - floor division (//)
  - exponentiation (**)

- **Comparison operators:** <, <=, >, >=, ==, !=
  - strictly less than (<)
  - less than or equal to (<=)
  - strictly greater than (>)
  - greater than or equal to (>=)
  - equal to (==)
  - not equal to (!=)

- **Expression conjunctive operators:** and, or, not
Example: Mathematical Operators

Python shell examples.

```python
>>> 2 ** 3 + 10
18
>>> 2 ** 3 ** 2
512
>>> 10 % 2
0
>>> 10 % 7
3
>>> (3.5 + 2.5) * 2
12
>>> 9.0 / 2
4.5
>>> 9.0 // 2
4.0
>>> 9 // 2
4
```
Example: Comparison Operators

Python shell examples.

```python
>>> 9 < 100
True
>>> 10 != 15
True
>>> 10 == 15
False
```
Example: Conjunctive Operators

Python shell examples.

```python
>>> True and True
True
>>> True and False
False
>>> (False or True) and True
True
>>> not True
False
>>> 10 < 5 or 15 != 60
True
```

Conjunctive operators are best used with Boolean values.
A variable holds a value. The value can be a number, a series of characters, or something more complex. As the name suggests, variables can vary by changing what pieces of information they hold or point to.

Examples of different data types for the variable x.

```python
>>> x = True
>>> type(x)
<class 'bool'>
>>> x = 4.658
>>> type(x)
<class 'float'>
>>> x = 10
>>> type(x)
<class 'int'>
```

Notice in this example, the variable x took on 3 different values.
Rules for forming Python variable names

- The first character must be a letter or underscore. Additional characters can be alphanumeric or underscore.
- No variables can begin with a number.
- No symbols other than alphanumerics or underscores are ever allowed.
- No variable can be the same as keywords, which form the foundation of the language.
- Variables are case-sensitive.
These words have a special meaning in Python. They cannot be used to name variables.

<table>
<thead>
<tr>
<th></th>
<th>False</th>
<th>None</th>
<th>True</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>as</td>
<td>assert</td>
<td>break</td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>def</td>
<td>del</td>
<td>elif</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>except</td>
<td>finally</td>
<td>for</td>
<td></td>
</tr>
<tr>
<td>from</td>
<td>global</td>
<td>if</td>
<td>import</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>is</td>
<td>lambda</td>
<td>nonlocal</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>or</td>
<td>pass</td>
<td>raise</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>try</td>
<td>while</td>
<td>with</td>
<td></td>
</tr>
<tr>
<td>yield</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assignment Statements (1)

An assignment statement associates a variable name on the left of the equal sign with the value of an expression calculated from the right of the equal sign.

In an assignment statement, the right hand side (RHS) is evaluated first. Then, the value from the RHS is assigned to the variable on the left-hand side (LHS). Once a variable is assigned a value, the variable can be used in place of that value.

```
assignments.py

1 counter = 0
2 miles = 1000.0
3 counter = counter + 1
4 kilometers = 1.609 * miles
5 print (miles)
6 print (counter)
7 print (kilometers)
```
Python’s usage of the equal sign is not mathematical. The mathematical usage of the equal sign is represented by two equal signs (==).

assignments2.py

```python
x = 10           # assigns the integer 10 to x
y = x == 10      # since x is equal to 10, y is assigned True
z = x != 10      # since x is not equal to 10, z is assigned False

print(x)
print(y)
print(z)
print(x == 50)
```
Strings are a contiguous set of characters in between quotation marks. There are three different types of quotation marks.

- Single quotes: 'Hello, world!'
- Double quotes: "Hello, world!"
- Triple quotes: '''Hello, world!'"

Additional examples

'Good morning.'
"Go Aggs!"
'''Mary shouted, "Watch out!"'''
Accessing the Characters in a String

Consider the string ‘ORANGE’. The position (or index) of each of the characters in the string are given below.

<table>
<thead>
<tr>
<th>index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>characters:</td>
<td>O</td>
<td>R</td>
<td>A</td>
<td>N</td>
<td>G</td>
<td>E</td>
</tr>
</tbody>
</table>

Python shell examples.

```python
>>> x = 'ORANGE'
>>> x[0]
'O'
>>> x[2]
'A'
>>> x[-1]
'E'
>>> x[-5]
'R'
```
Suppose you have a string variable called \( x \). Subsets of strings can be taken using the following operators.

- \( x[a] \) gives the character from the string \( x \) in location \( a \).
- \( x[a : b] \) gives the characters from the string \( x \) starting at index \( a \) and ending at index \( b - 1 \).
- \( x[a : b : c] \) gives the characters from the string \( x \) starting at index \( a \) and ending at index \( b - 1 \), where each index is incremented by \( c \).

When using slicing notation,

- if position \( a \) is missing, the default value is 0;
- if position \( b \) is missing, the default value is the length of string; and
- if position \( c \) is missing, the default value is 1.
Example: Slicing Strings

```python
>>> x = "Texas A&M University"
>>> x[0]
'T'
>>> x[2:5]
'xas'
>>> x[0:5:2]
'Txs'
>>> x[4:]
's A&M University'
>>> x[:6]
'Texas '
>>> x[-1]
'y'
>>> x[:−1]
'Texas A&M Universit'
```
The plus (+) sign is the string concatenation operator.
The asterisk (*) is the repetition operator.

Python shell examples.

```python
>>> x = 'apple'
>>> x * 2
'appleapple'
>>> y = 'banana'
>>> x + y
'applebanana'
>>> x + ' ' + y
'apple banana'
>>> 'cat' * 2 + ' ' + 'dog' * 3
'catcat dogdogdog'
```
The **in** operator is used to determine if a string contains an element of interest. The **not in** operator determines whether a string doesn’t contain the element of interest.

**Python shell examples.**

```python
x = 'banana'
>>> 'a' in 'apple'
True
>>> 'k' in x
False
>>> 'ppl' in 'apple'
True
>>> 'ale' in 'apple'
False
>>> 'abc' not in x
True
```
Strings are immutable sequences. A program can refer to elements or subsequences of strings. However, strings cannot be modified in place.

**Python shell examples.**

```python
>>> x = 'Marc'
>>> x
'Marc'
>>> y = x[0]
>>> y
'M'
>>> x[3] = 'k'
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'str' object does not support item assignment
```
Sometimes, you’d like your code to do different things depending on the current state. For this you can use an `if` statement:

```python
if <condition>:
    <do this>
```

If the condition is True, then the `<do this>` statements are executed.
```python
print 'begin'
a = 34
if a > 0:
    print 'a is positive'
print 'end'
```
Execution order
1, 2, 3, 4, 5

Output
begin
a is positive
end
If you need to choose between 2 options, use an **if-else** statement.

```python
if <condition>:
    <do this>
else:
    <do that>
```

If the condition is True, then the `<do this>` statements are executed. Otherwise, the `<do that>` statements are executed.
Example: if-else.py

```python
print ('begin')
a = 34
if a > 0:
    print ('a is positive')
else:
    print ('a is non-positive')
print ('end')
```
Execution order
1, 2, 3, 4, 7

Output
begin
a is positive
end
If you need more than two conditions (if-else), then use the **if-elif-else** statement. With this statement, you can have as many elif statements as needed.

```python
if <condition>:
    <do this>
elif <condition2>:
    <do this2>
elif <condition3>:
    <do this3>
else:
    <do that>
```
Example: if-elif-else.py

```python
print ('begin')
a = 34
if a == 0:
    print ('a is zero')
elif a > 0:
    print ('a is positive')
else:
    print ('a is negative')
print ('end')
```
<table>
<thead>
<tr>
<th>Execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 5, 6, 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
</tr>
<tr>
<td>a is positive</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Assume that the user enters '75' (without the quotes). What is the execution order and output of the above program?
Write a program to determine whether a user enters an even or odd number.

Example #1.
Enter a number: 17
17 is an odd number.

Example #2.
Enter a number: 1092732
1092732 is an even number.
# Determine whether the user entered an even or odd number.

user_input = input('Please enter a number: ')
number = int(user_input)

if (number % 2 == 0):
    print ('You entered an even number.
else:
    print ('You entered an odd number.')
# This program looks at the last character
# of the string to determine if it
# is even or odd.

# This program also uses '\'' to show how to
# continue a line. There is no space after the
# continuation character '\''.

number = input('Enter a number: ')
last_digit = number[-1]

if last_digit == '0' or last_digit == '2' 
or last_digit == '4' or last_digit == '6' 
or last_digit == '8':
    print (number, 'is an even number. ')
else:
    print (number, 'is an odd number. ')
It is often necessary to do an operation repeatedly. You can use a **while** loop for that.

```
while <condition>:
    <body of loop>
```

The statements in `<body of loop>` will be executed as long as the condition evaluates to True. Once the condition evaluates to False, the loop is exited.
What is the execution order and output of the above program?
Execution order
1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 6, 7

Output
before loop
count = 0
count = 1
count = 2
after loop
count = 3
Show the output of the above program when the user types in the strings: 'apple', 'hello', 'dog', and 'secret' (without the quotes). After playing with the program with different inputs, give a brief statement explaining what the following program does in general.
Infinite loops

When working with while loops, sooner or later you will accidentally send Python into a never-ending loop. Here is an example. (See the next slide for stopping a program in an infinite loop.)

infinite-loop.py

```python
i=0
while i < 10:
    print (i)
```

In this program, the value of `i` never changes. Thus, the condition `i < 10` is always true and Python continuously prints zeroes. How would you fix the above program so that it doesn’t get stuck in an infinite loop?
Write a program to play The Guessing Game, where the goal is to guess a secret number between 1 and 100.

Example output.

Welcome to The Guessing Game

Enter a number between 1 and 100: 87
Too high.

Enter a number between 1 and 100: 70
Too low.

Enter a number between 1 and 100: 80
Too low.

Enter a number between 1 and 100: 85
Too high.

Enter a number between 1 and 100: 83

Congratulations! You got it in 5 guesses.
Interesting computer games require randomness. Python comes with a module, called random, that allows us to use random numbers in our programs.

What is a module?
The core part of the Python language consists of things like if statements, math operators, while loops, print, and raw_input. Everything else is contained in modules. If we want to use something from a module, we have to import it. In other words, we must tell Python that we want to use it.
random.randint\((a, b)\) will return a random integer between \(a\) and \(b\) including both \(a\) and \(b\). That is, random.randint\((1,3)\) will randomly return the integer 1, 2, or 3.

```python
>>> import random
>>> random.randint(1, 10)
7
>>> random.randint(1, 100)
64
>>> x = random.randint(-37, 56)
>>> x
-14
>>> random.randint(1, 10)
7
```

The random numbers generated will be different each time we run the program.
```python
import random

print ('=================================')
print ('Welcome to The Guessing Game')
print ('=================================

secret_num = random.randint(1, 100)
guess = int(input('Enter a number between 1 and 100: '))
attempts = 1

while guess != secret_num:
    if guess < secret_num:
        print ('Too low.')
    else:
        print ('Too high.')
    guess = int(input('
Enter a number between 1 and 100: '))
    attempts += 1

print ('
Congratulations! You got it in %d guesses.' % (attempts))
```
Functions encapsulate a task. Here’s how they are defined in Python.

```python
def function_name (formal_parameters):
    '''Optional comment, which is called a docstring, describing your function.'''
    <function body>
    return <some value or values> # optional
```

You call the function as follows:

```python
function_name (actual_parameters)
```

This has the effect of executing the function body with the formal parameters replaced by the actual parameters.
What is the execution order and output of the following program?

```python
def print_msg():
    print('I love Python!')

def is_even(num, divisor):
    print(num % divisor == 0)

print_msg()
is_even(10, 2)
is_even(7, 10)
```
Execution order
1, 4, 7, 1, 2, 8, 4, 5, 9, 4, 5

Output
I love Python!
True
False
What is the execution order and output of the following program?
What is the execution order and output of the above program?
What is the execution order and output of the following program assuming the user enters the name 'Rosana' (without the quotes)?

```python
def happy():
    ''' Prints a line of the song. '''
    return 'Happy Birthday to you!'

def happy_birthday(name):
    ''' Prints the Happy Birthday! song. '''
    print(happy())
    print(happy())
    print('Happy Birthday, dear %s.' % (name))
    print(happy())

def main():
    ''' The main driver of the program. '''
    name = input('Enter your name: ')
    print()
    happy_birthday(name)

main()
```
What is the execution order and output of the following program assuming the user wants to convert 25°F to Celsius? Repeat the exercise assuming the user wants to convert 30°C to Fahrenheit.

```python
# Converts the temperature to Celsius or Fahrenheit.

def to_fahrenheit(c):
    ''' Convert Celsius to Fahrenheit '''
    return (c * 9.0/5.0) + 32

def to_celsius(f):
    ''' Convert Fahrenheit to Celsius '''
    return (f - 32) * 5.0/9.0

def main():
    type = input('Convert to Celsius or Fahrenheit (c or f)? ')
    if type == 'c':
        temperature = int(input('Enter Fahrenheit temperature: '))
        celsius = to_celsius(temperature)
        print ('%d Fahrenheit is %d Celsius.' % (temperature, celsius))
    else:
        temperature = int(input('Enter Celsius temperature: '))
        fahrenheit = to_fahrenheit(temperature)
        print ('%d Celsius is %d Fahrenheit.' % (temperature, fahrenheit))

main()
```
Why have functions?

- **Divide-and-Conquer Problem Solving:** Functions divide programs into smaller, more manageable pieces.

- **Simplification/Readability** (duplication removal): Anywhere that multiple lines of code are needed, a function can replace those lines. If the replacement can be done in multiple places, the result is simpler code.

- **Clean Namespaces:** Functions allow us to keep our variable namespace clean (local variables only “live” as long as the function does). In other words, function f1() can use a variable called x, and function f2() can also use a variable called x and there is no confusion.

- **Abstraction:** By encapsulating details, functions provide a programmer with a high-level view of program elements whose details can be filled in later—possibly by someone else.
## Built-in Functions

<table>
<thead>
<tr>
<th>abs()</th>
<th>divmod()</th>
<th>input()</th>
<th>ord()</th>
<th>type()</th>
</tr>
</thead>
<tbody>
<tr>
<td>all()</td>
<td>enumerate()</td>
<td>int()</td>
<td>pow()</td>
<td>vars()</td>
</tr>
<tr>
<td>any()</td>
<td>eval()</td>
<td>isinstance()</td>
<td>print()</td>
<td>zip()</td>
</tr>
<tr>
<td>ascii()</td>
<td>exec()</td>
<td>issubclass()</td>
<td>property()</td>
<td><strong>import</strong>()</td>
</tr>
<tr>
<td>bin()</td>
<td>filter()</td>
<td>iter()</td>
<td>range()</td>
<td></td>
</tr>
<tr>
<td>bool()</td>
<td>float()</td>
<td>len()</td>
<td>repr()</td>
<td></td>
</tr>
<tr>
<td>bytearray()</td>
<td>format()</td>
<td>list()</td>
<td>reversed()</td>
<td></td>
</tr>
<tr>
<td>bytes()</td>
<td>frozenset()</td>
<td>locals()</td>
<td>round()</td>
<td></td>
</tr>
<tr>
<td>callable()</td>
<td>getattr()</td>
<td>map()</td>
<td>set()</td>
<td></td>
</tr>
<tr>
<td>chr()</td>
<td>globals()</td>
<td>max()</td>
<td>setattr()</td>
<td></td>
</tr>
<tr>
<td>classmethod()</td>
<td>hasattr()</td>
<td>memoryview()</td>
<td>slice()</td>
<td></td>
</tr>
<tr>
<td>compile()</td>
<td>hash()</td>
<td>min()</td>
<td>staticmethod()</td>
<td></td>
</tr>
<tr>
<td>complex()</td>
<td>help()</td>
<td>next()</td>
<td>str()</td>
<td></td>
</tr>
<tr>
<td>delattr()</td>
<td>hex()</td>
<td>object()</td>
<td>sum()</td>
<td></td>
</tr>
<tr>
<td>dict()</td>
<td>id()</td>
<td>oct()</td>
<td>super()</td>
<td></td>
</tr>
<tr>
<td>dir()</td>
<td>input()</td>
<td>open()</td>
<td>tuple()</td>
<td></td>
</tr>
</tbody>
</table>
Functions: Local and Global Variables

```python
v = 15  # global

def f1():
    v = 17  # local
    print ('v(f1):', v)  # local
    v += 1  # local
    print ('v(f1):', v)  # local

def f2():
    print ('v(f2):', v)  # global

f1()
f2()
print ('v:', v)  # global
```

Basics of Python
The above program has an error. Find it and explain why it is in fact an error.
Ordered collection of data.

- Example: `some_data = ['dog', 78, 87.0, 'gorilla']`
- Elements can be of different types (heterogeneous)
- Can have a mixture of strings, ints, floats, lists, etc.

Accessing items in a list is similar to accessing characters in a string.

- Can create sublists by specifying an index range using the slicing operators `[:]` or `[::-]`.
- You can change individual elements directly ("mutable"). Unlike strings, each element in a list can be modified.
List Examples (Python Shell)

```python
>>> aList = [10, 20, 30, 40]  # list creation

>>> aList
[10, 20, 30, 40]

>>> aList[0]  # indexing individual elements
10

>>> aList[2:]  # creating sublist
[30, 40]

>>> aList[:3]  # creating sublist
[10, 20, 30]

>>> aList[1] = 50  # mutable

>>> aList
[10, 50, 30, 40]

>>> aList[1:5:2]  # creating sublist
[50, 40]
```
```python
import random

def generate_random_numbers(n):
    number_list = []
    i = 0

    while i < n:
        number_list += [random.randint(-100, 100)]
        i += 1

    return number_list

def list_sums(numbers):
    ''' Compute numbers[0] + numbers[-1],
    numbers[1] + numbers[-2], etc. 
    '''

    i = 0
    new_list = []
    while i < int(len(numbers) / 2):
        new_list += [numbers[i] + numbers[-i-1]]
        i += 1

    return new_list
```

Using built-in functions: list-example.py

```python
new_list += [numbers[i] + numbers[-(i+1)]]
i += 1

if len(numbers) % 2 == 1:
    index = int(len(numbers) / 2)
    new_list += [numbers[index]]

return new_list

def main():
    num_vals = int(input("How many integers?: "))
    numbers = generate_random_numbers(num_vals)

    print ('Number list:', numbers)
    print ('Maximum:', max(numbers))
    print ('Minimum:', min(numbers))
    print ('List sums:', list_sums(numbers))

main()
```
For loops are another construct in Python that supports repetition.

```python
for <var> in s:
    <body of loop>
```

For each element of `s`, the for loop executes the `<body of loop>` until there are no more elements in `s`. The valid data types for `s` are objects that support iteration (or iterables) like strings, lists, and tuples. `s` cannot be an int, float, or bool.
Example: for and while loops

The following two programs produce the exact same output.

while.py

```python
a_string = 'names'
i = 0
while i < len(a_string):
    print (a_string[i])
i += 1
```

Execution order: 1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3

for.py

```python
a_string = 'names'
for letter in a_string:
    print (letter)
```

Execution order: 1, 2, 3, 2, 3, 2, 3, 2, 3, 2
Example: for and while loops

Output of while.py and for.py

names
The following two programs produce the exact same output.

**while2.py**

```python
name_list = ['Walter', 'Nicole', 'Steven']
i = 0
while i < len(name_list):
    print(name_list[i], 'Smith')
i += 1
```

Execution order: 1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3

**for2.py**

```python
name_list = ['Walter', 'Nicole', 'Steven']
for name in name_list:
    print(name, 'Smith')
```

Execution order: 1, 2, 3, 2, 3, 2, 3, 2
Example: for and while loops using lists

Output of while2.py and for2.py

Walter Smith
Nicole Smith
Steven Smith
When to use for loop instead of a while loop?

For loops that run a predetermined amount of time, then use a for statement. If you cannot predetermine how often a loop repeats, then you must use a while loop.

For example, in the Guessing Game, we can’t predetermine how many times the user will need to guess a number between 1 and 100. We know the user will need between 1 and 7 guesses if they use an optimal playing strategy. But, we don’t know what that value (i.e., number of guesses) is ahead of time.
If you need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. The range function creates a generator object that can be used to iterate over a sequence.

Python shell examples:

```python
>>> numbers = range(3)  # creates a range object
>>> for number in numbers:
...     print(number)
...     ...
0
1
2
>>> numbers
range(0, 3)
>>> type(numbers)
<class 'range'>
>>> list(numbers)  # convert range object to list
[0, 1, 2]
```
More uses of the range function. All of the examples shown below convert the range object into a list of integers.

Python shell examples.

```python
>>> # create a list of integers from 5 to 9
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> # create a list integers from 0 to 9 in increments of 3
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> # create a list of integers from -10 to -100 in increments of -30
>>> list(range(-10, -100, -30))
[-10, -40, -70]
```
Example: counting with for loops

The following two programs produce the exact value for total on Line B. However, they have different values for i on Line A.

```
while3.py

1 total = 0
2 i = 1
3 while i <= 3:
4   total += i
5   i += 1
6 print (i)               # Line A
7 print (total)           # Line B
```

Execution order: 1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 6, 7

```
for3.py

1 total = 0
2 for i in range(1,4):
3   total += i
4 print (i)               # Line A
5 print (total)           # Line B
```

Execution order: 1, 2, 3, 2, 3, 2, 3, 2, 4, 5
Example: counting with for and while loops

Output of while3.py
4
6

Output of for3.py
3
6
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:

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*: 
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')

(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'):
    (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:

(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 (for Python 2)

$ apt-get install python3-bs4 (for Python 3)

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. Make sure you use the right version of pip or easy_install for your Python version (these may be named pip3 and easy_install3 respectively if you're using 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:

```bash
$ python3 setup.py install
```

or by manually running Python’s `2to3` conversion script on the `bs4` directory:

```bash
$ 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:

```bash
$ 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:

```bash
$ apt-get install python-html5lib
$ easy_install html5lib
$ pip install html5lib
```

This table summarizes the advantages and disadvantages of each parser library:

<table>
<thead>
<tr>
<th>Parser</th>
<th>Typical usage</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

```python
from bs4 import BeautifulSoup

with open("index.html") as fp:
    soup = BeautifulSoup(fp)

soup = BeautifulSoup("<html>data</html>")
```
First, the document is converted to Unicode, and HTML entities are converted to Unicode characters:

```python
BeautifulSoup("Sacré 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:

```python
soup = BeautifulSoup('"<b class="boldest">Extremely bold</b>"')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

### Name

Every tag has a name, accessible as `.name`:

```python
tag.name
# u'b'
```

If you change a tag's name, the change will be reflected in any HTML markup generated by Beautiful Soup:

```python
tag.name = "blockquote"
tag
# <blockquote class="boldest">Extremely bold</blockquote>
```

### Attributes

A tag may have any number of attributes. The tag `<b id="boldest">` has an attribute “id” whose value is “boldest”. You can access a tag's attributes by treating the tag like a dictionary:

```python
tag['id']
# u'boldest'
```

You can access that dictionary directly as `.attrs`:
tag.attrs
# {u'id': 'boldest'}

You can add, remove, and modify a tag’s attributes. Again, this is done by
treating the tag as a dictionary:

tag['id'] = 'verybold'
tag['another-attribute'] = 1
tag
# <b another-attribute="1" id="verybold"></b>

del tag['id']
del tag['another-attribute']
tag
# </b></b>

tag['id']
# KeyError: 'id'
(tag.get('id'))
# 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"></p>')</ncss_soup.p['class']
# ['body']

css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ['body', 'strikeout']

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>')</n
rel_soup.a['rel']  # ['index']
rel_soup.a['rel'] = ['index', 'contents']
(rel_soup.p)
# <p>Back to the <a rel="index contents">homepage</a></p>

You can use `get_attribute_list` to get a value that's always a list, string, whether or not it's a multi-valued attribute

id_soup.p.get_attribute_list('id') # ['my id']

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**

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]”:

```python
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 want about is the comment:

```python
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`:

```python
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:

```python
(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)

(soup.b.prettify())
# <b>
# <![CDATA[A CDATA block]]>
# </b>

Navigating the tree¶

Here's the “Three sisters” HTML document again:

def 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 were
<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>

</body>
</html>"

from bs4 import BeautifulSoup
soup = BeautifulSoup(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>]

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:
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:
    (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:
    (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`:

```
.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:

```python
for string in soup.strings:
    (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:

```python
for string in soup.stripped_strings:
    (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.

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:

```python
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:

```python
title_tag.string.parent
# <title>The Dormouse's story</title>
```

The parent of a top-level tag like <html> is the BeautifulSoup object itself:

```python
html_tag = soup.html
type(html_tag.parent)
# <class 'bs4.BeautifulSoup'>
```

And the .parent of a BeautifulSoup object is defined as None:

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:

```python
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:
        (parent)
    else:
        (parent.name)
# p
# body
# html
# [document]
# None
Going sideways

Consider a simple document like this:

```python
sibling_soup = BeautifulSoup("<a><b>text1</b></c><text2</c></b></a>")
(sibling_soup.prettify())
# <html>
#  <body>
#   <a>
#    <b>
#     text1
#    </b>
#   </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:

```python
sibling_soup.b.next_sibling
# <c>text2</c>
```

```python
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`:

The strings “text1” and “text2” are not siblings, because they don’t have the same parent:

```python
sibling_soup.b.string
# u'text1'
```

```python
(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:

```python
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:

```python
link.next_sibling.next_sibling
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
```

You can iterate over a tag's siblings with `.next_siblings` or `.previous_siblings`:

```python
for sibling in soup.a.next_siblings:
    (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
```

```python
for sibling in soup.find(id="link3").previous_siblings:
    (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:
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.

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:

```python
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”:

```python
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 </a> 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:

```python
last_a_tag.previous_element
# u' and

last_a_tag.previous_element.next_element
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

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:
    (repr(element))
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...
# 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 were
<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>

```

```
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:

```python
tag = soup.find_all('b')  # [b'The Dormouse\'s story']
```

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 `search()` method. This code finds all the tags whose names start with the letter “b”; in this case, the <body> tag and the <b> tag:

```python
import re
for tag in soup.find_all(re.compile("^b")):
    (tag.name)  # body  # b
```

This code finds all the tags whose names contain the letter ‘t’:

```python
for tag in soup.find_all(re.compile("t")):
    (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:

```python
tags = soup.find_all(['a', 'b'])  # [b'The Dormouse\'s story'],  # <a href='http://example.com/elsie' id='link1'>Elsie</a>,  # <a href='http://example.com/lacie' id='link2'>Lacie</a>,  # <a 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):
    (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:

```python
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">The Dormouse's story</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:

```python
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 BeautifulSoup

def surrounded_by_strings(tag):
    return (isinstance(tag.next_element, BeautifulSoup) and
            isinstance(tag.previous_element, BeautifulSoup))

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()**

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:

```python
soup.find_all("title")
# [<title>The Dormouse's story</title>]

soup.find_all("p", "title")
# [<p class="title">The Dormouse's story</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>]
```

```python
import re
soup.find(string=re.compile("sisters"))
# u'Once upon a time there were three little sisters; and their names were
```

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:

```python
soup.find_all("title")
# [<title>The Dormouse's story</title>]
```

### 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:

```python
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:

```python
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

This code finds all tags whose `id` attribute has a value, regardless of what the value is:

```python
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:

```python
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:

```python
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:

```python
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

You can't use a keyword argument to search for HTML's `name` element, because Beautiful Soup uses the `name` argument to contain the name of the tag itself. Instead, you can give a value to `name` in the `attrs` argument:
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_`:

```python
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`:

```python
soup.find_all(class_=re.compile("itl"))
# [<p class="title">The Dormouse's story</p>]
```

```python
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:

```python
css_soup = BeautifulSoup('<p class="body strikeout"></p>')</ncss_soup.find_all("p", class_="strikeout")
# [<p class="body strikeout"></p>]
```

```python
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:

```python
css_soup.find_all("p", class_="body strikeout")
# [<p class="body strikeout"></p>]
```

But searching for variants of the string value won't work:

```python
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:

```python
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:

```python
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

```python
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"]
```

```python
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)
```

```python
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', u'...']
```

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”:

```python
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`:

```python
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:
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:

```python
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
<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:
These two lines are also equivalent:

```python
django_soup.find_all("a")
django_soup("a")
```

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:

```python
django_soup.find_all('title', limit=1)
# [<title>The Dormouse's story</title>]

django_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`:

```python
Remember the `soup.head.title` trick from Navigating using tag names? That trick works by repeatedly calling `find`:

```python
soup.head.title
# <title>The Dormouse's story</title>

django_soup.find("head").find("title")
# <title>The Dormouse's story</title>
```

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:

```python
a_string = soup.find(string="Lacie")
a_string  
# u'Laclie'

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 were
#  <a class="sister" href="http://example.com/elsie" id="link1">Elise</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()¶
```

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()¶

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">The Dormouse's story</p>

find_all_next() and find_next()¶

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', u', u'Lacie', u' and
# u'Tillie',
# u';
# nand they lived at the bottom of a well.', u'
# u'...', u'
# ]

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()**

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:

```python
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**

As of version 4.7.0, Beautiful Soup supports most CSS4 selectors via the SoupSieve project. If you installed Beautiful Soup through pip, SoupSieve was installed at the same time, so you don't have to do anything extra.

BeautifulSoup has a .select() method which uses SoupSieve to run a CSS selector against a parsed document and return all the matching elements. Tag has a similar method which runs a CSS selector against the contents of a single tag.

(Earlier versions of Beautiful Soup also have the .select() method, but only the most commonly-used CSS selectors are supported.)

The SoupSieve documentation lists all the currently supported CSS selectors, but here are some of the basics:

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")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

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>]

There's also a method called `select_one()`, which finds only the first tag that matches a selector:

soup.select_one(".sister")
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

If you've parsed XML that defines namespaces, you can use them in CSS
selectors:

from bs4 import BeautifulSoup
xml = """<tag xmlns:ns1="http://namespace1/" xmlns:ns2="http://namespace2/"
    <ns1:child>I'm in namespace 1</ns1:child>
    <ns2:child>I'm in namespace 2</ns2:child>
</tag> """
soup = BeautifulSoup(xml, "xml")

soup.select("child")
# ["<ns1:child>I'm in namespace 1</ns1:child>, <ns2:child>I'm in namespace 2</ns2:child>"]

soup.select("ns1\child", namespaces=namespaces)
# ["<ns1:child>I'm in namespace 1</ns1:child>"]

When handling a CSS selector that uses namespaces, Beautiful Soup uses the namespace abbreviations it found when parsing the document. You can override this by passing in your own dictionary of abbreviations:

namespaces = dict(first="http://namespace1/", second="http://namespace2/"
soup.select("second\child", namespaces=namespaces)
# ["<ns1:child>I'm in namespace 2</ns1:child>"]

All this CSS selector stuff is a convenience for people who already know the CSS selector syntax. You can do all of this with the Beautiful Soup API. And if CSS selectors are all you need, you should parse the document with lxml: it's a lot faster. But this lets you combine 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>')</n
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 .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']

**extend()**

Starting in Beautiful Soup 4.7.0, .extend() also supports a method called .extend(), which works just like calling .extend() on a Python list:
soup = BeautifulSoup("<a>Soup</a>")
soup.a.extend(['s', '', 'on'])

soup
# <html><head></head><body><a>Soup's on</a></body></html>
soup.a.contents
# [u'Soup', u's', u', u'on']

**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</b>!--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")
orignal_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.
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:

```python
mrkup = '<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></a>
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 tags or strings immediately before something else in the parse tree:

```python
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></b><i>stop</i>
```

The insert_after() method inserts tags or strings immediately following something else in the parse tree:

```python
div = soup.new_tag('div')
div.string = 'ever'
soup.b.i.insert_after(" you ", div)
soup.b
# <b><i>Don't</i> you <div>ever</div> stop</b>
soup.b.contents
# [<i>Don't</i>, u' you', <div>ever</div>, u'stop']
```

clear()

Tag.clear() removes the contents of a tag:
tag.clear()
tag
# <a href="http://example.com/"></a>

```python
extract()
```

`PageElement.extract()` removes a tag or string from the tree. It returns the tag or string that was extracted:

markups = 'I linked to example.com'/a'
soup = BeautifulSoup(markup)
a_tag = soup.a

```python
i_tag = soup.i.extract()
a_tag
# <a href="http://example.com/"></a>
i_tag
# <i>example.com</i>
```

```
(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:

```python
my_string = i_tag.string.extract()
my_string
# u'example.com'
```

```
(my_string.parent)
# None
(i_tag)
# <i></i>
```

```python
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 &lt;i&gt;example.com&lt;/i&gt;&lt;/a&gt;

soup = BeautifulSoup(markup)

a_tag = soup.a

a_tag.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 a separate line for each HTML/XML tag and string:

You can call prettify() on the top-level BeautifulSoup object, or on any of its Tag objects:

(soup.a.prettify())

# <a href="http://example.com/">
# I linked to
# &lt;i&gt;
# example.com
# &lt;/i&gt;
# &lt;/a&gt;

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
# &lt;i&gt;example.com&lt;/i&gt;&lt;/a&gt;&lt;/body&gt;&lt;/html>''

unicode(soup.a)

# u'&lt;a href="http://example.com/">I linked to &lt;i&gt;example.com&lt;/i&gt;&lt;/a&gt;''

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 ":quot;:, they'll be converted to Unicode characters:

```python
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
unicode(soup)
# u'&lt;html&gt;&lt;head&gt;&lt;/head&gt;&lt;body&gt;&lt;u201cDammit!&lt;u201d he said.&lt;/body&gt;&lt;/html&gt;'
```

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:

```python
str(soup)
# 'The law firm of Dewey, Cheatem, &amp; Howe'
```

By default, the only characters that are escaped upon output are bare ampersands and angle brackets. These get turned into “&amp;”, “&lt;”, and “&gt;”, so that Beautiful Soup doesn't inadvertently generate invalid HTML or XML:

```python
soup = BeautifulSoup("<p>The law firm of Dewey, Cheatem, &amp; Howe</p>")
soup.p
# <p>The law firm of Dewey, Cheatem, &amp; Howe</p>
```

```html
<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 six possible values for `formatter`.

The default is `formatter="minimal"`. Strings will only be processed enough to ensure that Beautiful Soup generates valid HTML/XML:

```python
french = "<p>Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;&lt;/p>"
soup = BeautifulSoup(french)
(soup.prettify(formatter="minimal"))
# &lt;html&gt;
# &lt;body&gt;
# &lt;p&gt;
#   Il a dit &lt;&lt;Sacré bleu!&gt;&gt;&lt;/p&gt;
# &lt;/p&gt;
# &lt;/body&gt;
# &lt;/html&gt;
```

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;Sacré bleu!&gt;&gt;
# </p>
# </body>
# </html>

If you pass in `"formatter="html5"``, it's the same as

`formatter="html5"`, but Beautiful Soup will omit the closing slash in HTML
void tags like “br”:

sou = BeautifulSoup("<br>")

(soup.encode(formatter="html"))
# <html><body><br></body></html>

(soup.encode(formatter="html5"))
# <html><body><br></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:

(soup.prettify(formatter=None))
# <html>
# <body>
# <p>
#   Il a dit <<Sacré bleu!>>
# </p>
# </body>
# </html>

link_sou = BeautifulSoup('<a href=http://example.com/?foo=val1&bar=val2">A
link</a>')
(link_sou.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()

(soup.prettify(formatter=uppercase))
# <html>
# <body>
# <p>
#   IL A DIT &lt;&lt;SACRÉ BLEU!&gt;&gt;
# </p>
# </body>
# </html>

(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=\text{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())

(soup.prettify(formatter=uppercase_and_substitute_html_entities))
# <html>
# <body>
# <p>
#   IL A DIT &lt;&amp;SACRÉ; BLEU!&gt;&gt;
# </p>
# </body>
# </html>

One last caveat: if you create a CData object, the text inside that object is
always presented \textit{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")
(soup.a.prettify(formatter="xml"))
# <a>
# <![[CDATA[one < three]]>
# </a>

```python
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:

```python
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:

```python
# 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:

```python
# 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:

```python
[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:

```python
BeautifulSoup("<a><b /></a>")
# <html><head></head><body><a><b></b></a></body></html>
```

Since an empty `<b />` tag is not valid HTML, the parser turns it into a `<b></b>` 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:

```python
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>"", "xml")
# <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>

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>” 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:

```python
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 UnicodeDammit 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:
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 lock on it, and misidentifies it as ISO-8859-7:

```python
markup = b"<h1>\xed\xe5\xec\xf9</h1>"
soup = BeautifulSoup(markup)
soup.h1
<h1>εμω</h1>
soup.original_encoding
'ISO-8859-7'
```

We can fix this by passing in the correct `from_encoding`:

```python
soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
soup.h1
<h1>εμω</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`:

```python
soup = BeautifulSoup(markup, exclude_encodings=["ISO-8859-7"])
soup.h1
<h1>εμω</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:

```python
markup = b''
<html>
<head>
    <meta content="text/html; charset=ISO-Latin-1" http-equiv="Content-type"/>
</head>
<body>
    <p>Sacré bleu!</p>
</body>
</html>
```

soup = BeautifulSoup(markup)
(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()`:

```python
(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:

```python
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:

### 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:

```python
from bs4 import UnicodeDammit
dammit = UnicodeDammit("Sacr\xc3\xa9 bleu!")
(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](https://github.com/chardet/chardet) or [ccchardet](https://github.com/maxim-kaz/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:

```python
dammit = UnicodeDammit("Sacr\xe9 bleu!", ["latin-1", "iso-8859-1"])
(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'\x93I just &ldquo;love&rdquo; Microsoft Word&rsquo;s smart quotes</p>'

UnicodeDammit(markup, ["windows-1252"],
smart_quotes_to="xml").unicode_markup
# u'I just &lt;I love&lt; Microsoft Word&lt;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'\x201cI love\x201d Microsoft Word\x2019s 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'\u201cI love\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:

```python
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE QUOTATION MARK}"
```

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:

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:

```python
new_doc = UnicodeDammit.detwingle(doc)
(new_doc.decode("utf8"))
# ÜÜÜ“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 “å¨få¨få¨f“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 `<b>` tags are treated as equal, even though they live in different parts of the object tree, because they both look like “<b>pizza</b>”:

```python
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')
first_b == second_b
# True

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`:

Copying Beautiful Soup objects

You can use `copy.copy()` to create a copy of any `Tag` or `NavigableString`:

```python
import copy
p_copy = copy.copy(soup.p)
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:

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:

This is because two different `Tag` objects can’t occupy the same space at the same time.
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.)

```python
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:
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.

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:

```python
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']
```
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:

```python
from bs4.diagnose import diagnose
with open("bad.html") as fp:
    data = fp.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.HTMLParserError`. 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.HTMLParserError: malformed
start tag` and `HTMLParser.HTMLParserError: 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 `chardet` 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:

```bash
$ apt-get install python-beautifulsoup
```

It’s also published through PyPi as `BeautifulSoup`:

```bash
$ easy_install BeautifulSoup
$ pip install BeautifulSoup
```

You can also [download a tarball of Beautiful Soup 3.2.0](https://www.crummy.com/software/BeautifulSoup/bs3/dists/BeautifulSoup-3.2.0.tar.gz).

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:

```python
from BeautifulSoup import BeautifulSoup
```

becomes this:

```python
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](https://www.python.org/dev/peps/pep-0008/). 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:

```python
Tag.has_key() -> Tag.has_attr()
```

I renamed one attribute to use more accurate terminology:

```python
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:

```python
for parent in tag.parentGenerator():
    ...
```

You can write this:

```python
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.
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. (a 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.