Wrangling Big Data for DH

Félix-Antoine Fortin
Welcome to DHSI 2018!

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.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2018”.

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 ....
Regional Map of Greater Victoria

Legend:
- Direction of Travel
- Route Name
- Transit Exchange
- Park & Ride Lot
  - (no overnight parking)
- Major Stop

Average Frequency:
- **Regional Route**: 15–60 minute service with limited stops
- **Frequent Route**: 15 minute or better service, 7am-7pm, Mon-Fri
- **Local Route**: 20–120 minute service

Legend:
- Vic General
- Colwood
- Uptown
- Regional Route
- Frequent Route
- Local Route

Service Frequencies:
- **Regional Route**: Average Frequency
- **Frequent Route**: 20 minute or better service
- **Local Route**: 30 minute or better service

Service Times:
- **Regional Route**: Average Frequency
- **Frequent Route**: 20 minute or better service
- **Local Route**: 30 minute or better service

Service Areas:
- **Westshore**: Map of the region
- **Saanich**: Details of the area
- **Victoria**: Central part of the region
- **Colwood**: Southern part of the region
- **Royal Oak**: Northern part of the region

Transportation:
- **Bus Services**: Route numbers and stops
- **Transit Exchanges**: Map of the transit exchange locations
- **Park & Ride Lots**: Locations of park and ride lots
- **Major Stops**: Major stops along the routes

The 2018 schedule is just about ready! A very few things to confirm, add, etc, but this is the place to be to find out what is happening when / where ...

Psst: Some 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, Saltspring 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 Saltspring 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 Christie’s Carriage House, a hangry 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 (Hickman 120, 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, 4 June 2018

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Dan Sondheim.

7:45 to 8:15

Last-minute Registration (MacLaurin Building, Room A100)

8:30 to 10:00

Welcome, Orientation, and Instructor Overview (MacLaurin A144)
Classes in Session (click for details and locations)

- 1. [Foundations] Text Encoding Fundamentals and their Application (Cornett A128, Classroom)
- 3. [Foundations] Making Choices About Your Data (MacLaurin D109, Classroom)
- 4. [Foundations] DH For Department Chairs and Deans (Hickman 120, Classroom)
- 5. [Foundations] Introduction to Javascript and Data Visualization (Clearihue D132, Classroom)
- 6. [Foundations] Introduction to Computation for Literary Criticism (Clearihue A195, Lab)
- 7. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab)
- 8. Sounds and Digital Humanities (MacLaurin D111, Classroom)
- 9. Digital Humanities Pedagogy: Integration in the Curriculum (MacLaurin D016, Classroom)
- 10. Text Processing - Techniques & Traditions (McPherson Library A003, Classroom)
- 11. 3D Modelling for the Digital Humanities and Social Sciences (MacLaurin D010, Classroom)
- 12. Conceptualising and Creating a Digital Edition (MacLaurin D103, Classroom)
- 13. Visualizing Information: Where Data Meets Design (MacLaurin D107, Classroom)
- 14. Introduction to Electronic Literature in DH: Research and Practice (MacLaurin D115, Classroom)
- 15. Ethical Data Visualization: Taming Treacherous Data (MacLaurin D101, Classroom)
- 16. Race, Social Justice, and DH: Applied Theories and Methods (MacLaurin D105, Classroom)
- 17. XML Applications for Historical and Literary Research (Clearihue A103, Lab)
- 18. Processing Humanities Multimedia (Human and Social Development A150, Lab)
- 19. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (MacLaurin D110, Classroom)
- 20. Web APIs with Python (Human and Social Development A170, Lab)
- 21. Digital Publishing in the Humanities (Clearihue D131, Classroom)
- 22. Linked Open Data and the Semantic Web (Clearihue D130, Classroom)
- 23. Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images (MacLaurin D114, Classroom)
- 24. Feminist Digital Humanities: Theoretical, Social, and Material Engagements (Cornett A229, Classroom)
- 25. The Frontend: Modern JavaScript & CSS Development (Clearihue A030, Classroom)
- 26. The Frontend: Modern JavaScript & CSS Development (Clearihue A030, Classroom)

10:15 to Noon

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

Undergraduate Meet-up, Brown-Bag (details via email)

1:30 to 4:00

Institute Panel: Perspectives on DH (or, #myDHis …)
Chair: Alyssa Arbuckle (U Victoria)
(MacLaurin A144)

- Milena Radzikowska (Mt Royal C): "Release the Kraken: Story-Driven Prototyping for the Digital Humanities."
  Abstract: I have spent the last 15 years of my career designing text analysis tools for use by humanities scholars. In this brief presentation, I propose to share a concept-based approach to interface design for DH.

- Emily Murphy (U Victoria): "#MyDHis Edgy."
  Abstract: I will build upon—or, possibly, perform a misprision of—a tweet by Polina Vinogradova; "#myDHis messy, dusty, edgy, and radically inclusive!" Vinogradova evokes the mess and dust of the archives, the edges that connect nodes of a network, and the political impetus to think of cultural history and community together. I argue that these aspects of DH have a renewed importance as we head into a moment of feminist historiography.

- Margaret Konkol (Old Dominion U): "Prototyping Mina Loy’s Alphabet with a 3D Printer."
  Abstract: This talk discusses the interpretive and methodological implications of using 3D printing technologies to prototype the archival diagrams of a proposed but never constructed plastic segmental alphabet letter kit—a game designed by modernist poet Mina Loy for F.A.O Schwarz. Although intended as a toy for young children, "The Alphabet that Builds Itself," as a work of "object typography" articulates a theory of language as kinetic, geometric, recombinant, and open to mutation. Alphabetic segments extend into the x, y, and z coordinates in exponential iterations and conjoin with magnets. Combining elements of contemporaneous typefaces like Futura and Gill Sans, which represented modernity’s functional ideals and democratic principles of simplicity, these recombinant letters represent, as this talk argues, Loy’s unpublished modernist poem, an articulation of Loy’s concept of language as a physical fact in which substance, not just form, is semantic.

- Lee Zickel (Case Western Reserve U): "Comfortably Trepid."
  Abstract: #myDHis found outside the well-established, DH-friendly institutions, at an institution that is devoted predominantly to Medicine and Engineering. I, and with increasing frequency other DH practitioners and instructors, am not positioned in a DH Lab or Humanities Center, but in ITS. Part teacher, part technologist, part translator, I will briefly discuss my work supporting humanists and social scientists, particularly those who are new to or less comfortable with computational methodologies.

4:10 to 5:00

- Dorothy Kim (Vassar C): "#MyDHis Antifascist."
  Abstract: I’ve spent a lot of time in the last 12 months thinking about fascism, digital humanities, its long histories, and what it means to do DH work that centers social justice particularly in this global rise of late fascism. I will speak briefly about DH’s history, including the medieval history related to Busa but how that history really connects to data systems that created the Holocaust and also participated in the Cold War nuclear military complex.
Randa El Khatib (U Victoria): "Learning from the Iterative Process."
Abstract: #MyDHIs Iterative. In addition to the improvements that come with iterative projects, the iterative process itself is a fruitful area for scholarly inquiry. Within this iterative context, the various teams that I work with and I have been reflecting on and rethinking central DH practices, such as what it means to collaborate, prototype, remix, and implement DH values in our work. In this talk, I will present the various lessons learnt along the way.

Sarah Melton (Boston C): "#MyDHIs... People."
Abstract: Taking seriously Miriam Posner’s exhortation to “commit to DH people, not DH projects,” I invite us to reflect on how people are the core of DH. In this brief talk, I will explore the intersections between DH, labor, and infrastructure.

5:00 to 6:00
Opening Reception (University Club)
We are grateful to Gale Cengage for its sponsorship.

Tuesday, 5 June 2018

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
"Mystery" Lunches

DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)
- 73. Introduction to ORCID (Digital Scholarship Commons, Classroom).

1:30 to 4:00
Classes in Session

DHSI Colloquium Lightning Talk Session 1 (MacLaurin A144)
Chair: James O’Sullivan
- New Modes of DH and Archival Skills Acquisition in a Graduate Public History Course. Paulina Rousseau (Ryerson U)
- Walking a Transect: Exploring a Soundscape. John Barber (Washington State U)
- Centering the Edge Case: Designing Services for Humanities Data Research. Grace Afsari-Mamagani (New York U)
- Orwellian Vocabulary and the 21st-Century Politics. Ilgin Kizilgunesler (U Manitoba)
- Making Open Data from a Gray Archive. Sara Palmer (Emory U)

6:00 to 8:00
DHSI Newcomer’s Beer-B-Q (Felicitas, Student Union Building)

Wednesday, 6 June 2018

9:00 to Noon
Classes in Session

Lunch break / Unconference
"Mystery" Lunches

Brown Bag Lecture: Alexandra Branzan Albu (U Victoria): "Visual Recognition of Symbolic and Natural Patterns" (Digital Scholarship Commons, 3rd Floor McPherson Library)

Abstract: Image-based object recognition is a visual pattern recognition problem; one may characterize visual patterns as either symbolic or natural. Symbolic patterns evolved for human communication; they include but are not limited to text, forms, tables, graphics, engineering drawings etc. Symbolic patterns vary widely in terms of size, style, language, alphabet and fonts; however, literate humans can easily compensate for this variability and instantly recognize most symbolic patterns. On the other hand, natural patterns characterize images of physical structures; they often lack the intrinsic discriminability and structure of symbolic patterns, and vary widely in terms of pose, perspective, and lighting.

This lecture will explore similarities and differences in approaches designed for recognizing visual and symbolic patterns, and will address the following questions via examples.
- What are the distinctive characteristics of natural patterns? What dimensions of variability can we infer?
- What are the distinctive characteristics of symbolic patterns? What dimensions of variability can we infer?

Alexandra Branzan Albu is an Associate Professor with the Department of Electrical and Computer Engineering and cross-listed with Computer Science. Her research interests are related to image analysis, computer vision, and visual computing. She is actively pursuing outreach activities dedicated to increasing the women's presence in electrical engineering and computer science.

1:30 to 4:00
Classes in Session
Thursday, 7 June 2018

9:00 to Noon Classes in Session

12:15 to 1:15 UVIC Library/ETCL lunchtime talk: “A Humanities Application of 3D printing and Machine Translation in the ChessBard and Loss Sets” by Dr. Aaron Tucker

1:30 to 4:00 Classes in Session

4:15 to 5:15 DHSI Colloquium Lightning Talk Session 3 (MacLaurin A144) Chair: James O’Sullivan

- Documenting Deportation: A Collaborative Digital Collection. Paulina Rousseau (Ryerson U)
- Unleashing the Power of Texts as Networks: Visualizing the Scholastic Commentaries and Texts Archive. Jeffrey Witt (Loyola U Maryland) and Drew Winget (Stanford U)
- #haunteDH: Punching holes in the International Busa Machine Narrative. Arun Jacob (McMaster U)
- Text in World: Computational Analysis of Trauma in Genocide Narratives. Nanditha Narayanamoorthy (U York) and Krish Perumal (U Toronto)

7:30 to 9:30 (Groovy?) Movie Night (MacLaurin A144)

Friday, 8 June 2018 [DHSI; DLFxDHSI Opening]

9:00 to Noon DHSI Classes in Session

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

1:00 to 2:00 DLFxDHSI Registration (MacLaurin A100)

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

2:00 to 3:00 Joint Institute Lecture (DHSI and DLFxDHSI): Bethany Nowviskie (CLIR DLF and U Virginia): “Reconstitute the World: Machine-reading Archives of Mass Extinction” Chair: Lisa Goddard (U Victoria) (MacLaurin A144)

Abstract: The basic constitution of our digital collections becomes vastly more important in the face of two understandings: first, that archives of modernity are archives of the sixth great mass extinction of life on our planet; and next, that we no longer steward cultural heritage for human readers alone. In the same way that we people are shaped by what we read, hear, and see, the machine readers that follow us into and perhaps beyond the Anthropocene have begun to learn from "unsupervised" encounters with our digital libraries. What will we preserve for the living generations and artificial intelligences that will come? What do we neglect, or even choose to extinguish? And from an elegiac archive, a library of endings, can we create forward-looking, speculative collections--collections from which to deep-dream new futures? The most extra/ordinary power we possess is the power to make poetry from records of the past. Could it be called on, one day, to reconstitute the world?
Joint Reception: DHSI and DLFxDHSI (University Club)
DLFxDHSI Poster/Demo Session

* Mediaris as a Colonialist Artifact in Menzies’ Journal. Paula Johanson (U Victoria)
* Camp Edit: the Institute for the Editing of Historical Documents. Nikolaus Wasmoen (Association for Documentary Editing, U Buffalo), Jennifer Stertzter (Association for Documentary Editing, U Virginia), and Cathy Moran Hajo (Association for Documentary Editing, Ramapo C)
* A Digital Archaeology of Life in Cleveland’s Depression-Era Slums. Charlie Harper (Case Western Reserve U) and Jared Bendis (Case Western Reserve U)
* Feminist Pest Control: controlling and not controlling nonhuman pests. Lindsay Garcia (C of William and Mary)
* Legends of the Buddhist Saints. Jonathan S. Walters (Whitman C) and Dana Johnson (Freelance Web Developer)
* Accessibility in Digital Environments Via TEI-Encoded Uncontracted Braille. Gia Alexander (Texas A&M U)
* Translation3point0: Why Literary Translation Data Matters. Katie King (U Washington)
* PoéticaSonora: A Digital Audio Repository Prototype for Latin American Sound Art and Poetry. Aurelio Meza (Concordia U)
* Beauty and the Book: Pre-Raphaelite Artistic Practice Contained. Josie Greenhill (U Victoria)
* Poetic Procedures/Digital Deformances. Corey Sparks (California State U, Chico)
* Miranda, the Folger Shakespeare Library’s new Digital Asset Platform. Meaghan Brown (Folger Shakespeare Library)
* Living Song Project. Quinn Patrick Ankrum (U Cincinnati) and Elizabeth Avery (U Oklahoma)
* Digital Frankenstein Variorum. Rikk Mulligan (Carnegie Mellon U)

**Saturday, 9 June 2018 [DLFxDHSI + DHSI Conference and Colloquium]**

8:30 to 9:00  DLFxDHSI Registration (MacLaurin A100)

9:00 to 5:30  DLFxDHSI UnConference Sessions

9:00 to 4:00  DHSI All Day Workshop Session (click for workshop details and free registration for DHSI participants)

9:00 to 4:00  53. Building Your Academic Digital Identity (MacLaurin D105, Classroom)

9:00 to 5:00  DHSI Colloquium Day Conference (MacLaurin A144)

**Welcome**

People I: Documenting Online Lives. Chair: Molly Nebiolo (University of New York)

- Examining Gendered Harassment Online and in Silicon Valley. Andrea Flores (Utica College)
- This is Just to Say I Have &lt;X&gt; the &lt;Y&gt; in your &lt;Z&gt;: Modernist Memes in an Era of Public Apology. Shawna Ross (Texas A&M University)

Break

People II: Documenting Lives Online. Chair: Dheepa Sundaram (College of Wooster)

- Youtube Yoga and Ritual on Demand: The Virtual Economics of Hindu Soteriology. Dheepa Sundaram (College of Wooster)
- The Resemblage Project: Creativity and Digital Health Humanities in Canada. Andrea Charise (University of Toronto) and Stefan Krecsy (University of Toronto)

Lunch

Projects I: Building and Analyzing. Chair: Yannis Rammos (New York University)

- Building the ARTECHNE Database: New directions in Digital Art History. Marieke Hendriksen (Old Dominion University)
- The Ineffective Inquisition: The Holy Office’s Sphere of Influence in Early Modern New Spain. Kira Homo (Pennsylvania State University)

Break

Projects II: Mapping and Visualizing. Chair: Innocent Opara (Qumet Institute)

- Mapping Sarah Sophia Bank’s Numismatic Collection. Erica Hayes (North Carolina State University) and Kacie Wills (University of California, Riverside)
- Text Mining and Visualizing 18th Century American Correspondence. Ashley Sanders Garcia (University of California, Los Angeles)

Break

Practices: Digital Scholarship on Campus and in the Classroom. Chair: Alyssa Arhuckle (University of Victoria)
Concluding Remarks

Sunday, 10 June 2018 [SINM + DHSI Registration, Workshops]

8:30 to 9:00
Symposium on Indigenous New Media Registration (MacLaurin A100)

9:00 to 5:00
DHSI Registration (MacLaurin A100)

9:00 to 4:00
▼ SINM Sessions
- 63. Symposium on Indigenous New Media: Reading Group (Hickman 105, Classroom)
- 72. Symposium on Indigenous New Media: Indigitization (Hickman 120, Classroom)

9:00 to 4:00
▼ DHSI All Day Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 53. Building Your Academic Digital Identity (MacLaurin D105, Classroom)
- 54. An Introduction to the Archaeology of 1980s Computing (MacLaurin D114, Classroom)

9:00 to Noon
▼ DHSI AM Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 55. Regular Expressions (MacLaurin D111, Classroom)
- 56. 3D Visualization for the Humanities (MacLaurin D100, Classroom)
- 58. DH Fieldwork Methods (MacLaurin D016, Classroom)
- 60. Pedagogy of the Digitally Oppressed: Inculcating De-/Anti-/Post-Colonial Digital Humanities (MacLaurin D107, Classroom)
- 61. Introduction to #GraphPoem. Digital Tools for Poetry Computational Analysis and Graph Theory Apps in Poetry (MacLaurin D101, Classroom)
- 62. Creating a CV for Digital Humanities Makers (MacLaurin D115, Classroom)

1:00 to 4:00
▼ DHSI PM Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 64. Agent-Based Modelling in the Humanities (MacLaurin D111, Classroom)
- 65. Unleash Linux on MacOS (MacLaurin D010, Classroom)
- 66. DHSI Knits: History of Textiles and Technology (MacLaurin D016, Classroom)
- 67. Crowdsourcing as a Tool for Research and Public Engagement (MacLaurin D109, Classroom)
- 69. Web Annotation as Critical Humanities Practice (MacLaurin D103, Classroom)
- 70. Dynamic Ontologies for the Humanities (MacLaurin D107, Classroom)
- 71. Social Media Research in the Humanities (MacLaurin D101, Classroom)

4:10 to 5:00
▼ Joint Institute Lecture (DHSI and SINM):
David Gaertner (U British Columbia): "A Landless Territory?: CyberPowWow and the Politics of Indigenous New Media.”
Chair: Deanna Reder (Simon Fraser U) (MacLaurin A144)
Abstract: Following the 1997 launch of Skawennati’s (Mohawk) CyberPowWow, digital space has become a vital new territory for the resurgence of Indigenous storytelling and cultural practice: "We have signed a new treaty," Cree artist Archer Pechawis wrote of this period, "and it is good. We have the right to hunt, fish, dance and make art at www.CyberPowWow.net, .org and .com for as long as the grass grows and the rivers flow.” This talk will critically explore the theoretical, cultural, political-economic, and gendered dynamics underwriting the histories and futures of Indigenous new media. Particular attention will be given in examining the ways in which new media and digital storytelling connect to and support key issues in the field of Indigenous studies, such as sovereignty, self-determination, decolonization, and land rights.

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.

Monday, 11 June 2018 [DHSI + SINM]
Your hosts for the week are Ray Siemens and Dan Sondheim.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:45 to 8:15</td>
<td>DHSI Last-minute Registration [MacLaurin A100]</td>
<td></td>
</tr>
<tr>
<td>8:30 to 10:00</td>
<td>DHSI Welcome, Orientation, and Instructor Overview [MacLaurin A144]</td>
<td></td>
</tr>
<tr>
<td>9:00 to 4:00</td>
<td>SINM Sessions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DHSI Classes in Session (click for details and locations)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 28. [Foundations] Developing a Digital Project (With Omeka) [Clearihue D132, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 29. [Foundations] Models for DH at Liberal Arts Colleges (&amp; 4 yr Institutions) [MacLaurin D109, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 32. Stylistometry with R: Computer-Assisted Analysis of Literary Texts [Clearihue A102, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 33. Digital Storytelling [MacLaurin D111, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 34. Text Mapping as Modelling [Clearihue D131, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 35. Geographical Information Systems in the Digital Humanities [Clearihue A105, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 36. Open Access and Open Social Scholarship [MacLaurin D114, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 37. Introduction to Machine Learning in the Digital Humanities [Cornett A229, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 38. Queer Digital Humanities: Intersections, Interrogations, Iterations [MacLaurin D110, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 41. Using Fedora Commons / Islandora [Human and Social Development A160, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 42. Documenting Born Digital Creative and Scholarly Works for Access and Preservation [MacLaurin D115, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 43. Games for Digital Humanists [MacLaurin D106, Classroom &amp; Human and Social Development A170, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 44. XPath for Document Archeology and Project Management [Cornett A128, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 45. Surveillance and the Digital Humanities [MacLaurin D103, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 47. Text Analysis with Python and the Natural Language ToolKit [Clearihue A103, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 48. Information Security for Digital Researchers [Clearihue D130, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 49. Wrangling Big Data for DH [Human and Social Development A150, Lab]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 50. Accessibility &amp; Digital Environments [MacLaurin D101, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 51. Critical Pedagogy and Digital Praxis in the Humanities [MacLaurin D105, Classroom]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 52. Drupal for Digital Humanities Projects [MacLaurin D107, Classroom]</td>
<td></td>
</tr>
<tr>
<td>10:15 to Noon</td>
<td>Lunch break / Unconference Coordination Session [MacLaurin A144]</td>
<td>(Grab a sandwich and come on down!) DHSI Undergraduate Meet-up, Brown-Bag (details via email)</td>
</tr>
<tr>
<td></td>
<td>DHSI Classes in Session</td>
<td></td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference Coordination Session [MacLaurin A144]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Grab a sandwich and come on down!) DHSI Undergraduate Meet-up, Brown-Bag (details via email)</td>
<td></td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>DHSI Classes in Session</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ Joint Institute Lecture (DHSI and SINM): Jordan Abel (Simon Fraser U): &quot;Indigeneity, Conceptualism, and the Borders of DH.&quot; Chair: Michelle Brown (U Hawaii) [MacLaurin A144]</td>
<td></td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Joint Reception: DHSI and SINM [University Club]</td>
<td></td>
</tr>
<tr>
<td>5:00 to 6:00</td>
<td>Joint Reception: DHSI and SINM [University Club]</td>
<td></td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lunch break / Unconference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ DHSI Lunchtime Workshop Session [click for workshop details and free registration for DHSI participants]</td>
<td></td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▼ 73. Introduction to ORCID [Digital Scholarship Commons, Classroom]</td>
<td></td>
</tr>
</tbody>
</table>
**Wednesday, 13 June 2018**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Colloquium Lightning Talk Session 4 (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>Chair: Lindsey Seatter</td>
<td></td>
</tr>
<tr>
<td>- Mapping Indigenous and Chicana/o Environmental Imaginaries using GIS. Stevie Ruiz (California State U, Northridge), Quetzalli Enrique (California State U, Northridge), Enrique Ramirez (California State U, Northridge), and Tomas Figueura (California State U, Northridge)</td>
<td></td>
</tr>
<tr>
<td>- &quot;But is it any good?&quot;: A quantitative approach to the popularity of digital fanfiction. Suzanne Black (U Edinburgh)</td>
<td></td>
</tr>
<tr>
<td>- The American Prison Writing Archive (APWA). Doran Larson (Hamilton C), Janet Simons (Digital Humanities Initiative, Hamilton C), and William Rasenberger (Hamilton C)</td>
<td></td>
</tr>
<tr>
<td>6:00 to 8:00</td>
<td>DHSI Newcomer's Beer-B-Q (<a href="#">Felicitas, Student Union Building</a>)</td>
</tr>
</tbody>
</table>

**Thursday, 14 June 2018**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Colloquium Lightning Talk Session 5 (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>Chair: Lindsey Seatter</td>
<td></td>
</tr>
<tr>
<td>- Faraway, so close: Has the political environment really changed in Ecuador?. Luis Meneses (Electronic Textual Cultures Lab, U Victoria)</td>
<td></td>
</tr>
<tr>
<td>- Re-mixing Melville’s Reading: Text Analysis of Marginalia with R and XSLT. Christopher Ohge (U London, School of Advanced Study) and Steven Olsen-Smith (Boise State U)</td>
<td></td>
</tr>
<tr>
<td>- Developing Interactive and Open-Source OER: Inquiry-Based Music Theory. Evan Williamson (U Idaho)</td>
<td></td>
</tr>
<tr>
<td>- Spatial Humanities and the Web of Everywhere. Ken Cooper (SUNY Geneseo)</td>
<td></td>
</tr>
<tr>
<td>6:00 to 7:00</td>
<td>&quot;Half Way There (yet again)!&quot; [An Informal, Self-Organized Birds of a Feather Get-Together] (<a href="#">Felicitas, Student Union Building</a>)</td>
</tr>
<tr>
<td>Bring your DHSI nametag and enjoy your first tipple on us!</td>
<td></td>
</tr>
</tbody>
</table>

**Friday, 15 June 2018**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Colloquium Lightning Talk Session 6 (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>Chair: Lindsey Seatter</td>
<td></td>
</tr>
<tr>
<td>- Composition not Inheritance: Imagining a Functional Digital Humanities. Andrew Pilsch (Texas A&amp;M U)</td>
<td></td>
</tr>
<tr>
<td>- Plotting Our Trajectories: Navigating, Situating, and Re-Inventing Research Topoi with R. Sean McCullough (Texas Christian University) and Jongkeyong Kim (Texas Christian U)</td>
<td></td>
</tr>
<tr>
<td>- Herb Simon and His Books. Avery Wiscomb (Carnegie Mellon U) and Daniel Evans (Carnegie Mellon U)</td>
<td></td>
</tr>
<tr>
<td>- (De/Re)Defining &quot;The Digital&quot;: A Decolonial Approach to Digital Humanities. Ashley Caranto Morford (U Toronto) and Arun Jacob (McMaster U)</td>
<td></td>
</tr>
<tr>
<td>7:30 to 9:30</td>
<td>(Groovier?) Movie(r) Night (<a href="#">MacLaurin A144</a>)</td>
</tr>
</tbody>
</table>

**Classes in Session**

9:00 to Noon

Lunch Reception / Course E-Exhibits ([MacLaurin A100](#))
1:30 to 2:30

(MacLaurin A144)

Abstract: Much has changed and continues to change in digital humanities since the formal establishment of Iter in the Fall of 1997. However, the mandate of the not-for-profit partnership to support “the advancement of learning in the study and teaching of Middle Ages and Renaissance (400–1700) through the development and distribution of online resources” continues to have relevance. This presentation explores the striking challenges faced by Iter and presents our current thinking on the realization of this mandate for the future through a platform with a focus on facilitating the discovery of the academic resources necessary to our work; creating an environment for collaboration, sharing and developing projects; and on enabling the distribution and publication of our scholarship.

2:40 to 3:00

Awards and Bursaries Recognition
Closing, DHSI in Review (MacLaurin A144)

Contact info:
institut@uvic.ca P: 250-472-5401 F: 250-472-5681
Wrangling Big Data for DH

DHSI 2018, June 11-15, 2018

Rough Schedule

(Note that the schedule and content may change but the overall scope should remain fixed)

Day 1/Monday: What Big Data is and a quick hands-on running Spark. This day will lay down the fundamental concepts in the morning and then spend the afternoon showing how those concepts come into play by creating a process to automatically scrape twitter content.

Day 2/Tuesday. A day working with the basics of Spark. Having received the basic concepts and walked through a pre-created example on Monday participants will now be walked through the process of creating a Big Data analysis pipeline from scratch. Additional concepts will be introduced. The morning will mostly focus on instruction and the afternoon on working through a series of challenges/exercises.

Day 3/Wednesday. Hands-on with more real-world use cases and examples. We will focus on OCR data provided by Gale Cengage that covers the London Times. Participants will be given more independence in their approach to solving the relevant tasks.

Day 4/Thursday. Independent Hands-on. Participants are encouraged to bring their own Big Data problem to the course and today is the day that we will focus on crunching those problems or at least designing the analysis pipeline. For those without a problem at hand the instructors will provide a use case that will likely involve locating text in collections of old photos and/or another text analysis example.

Day 5/Friday. Clean-up. We only have the morning here so we’ll be focused on adding any final components to the independent projects from Thursday, making sure that all participants have understood the relevant concepts, and pointing to resources for continuing Big Data analysis in the future.
Some initial course reading suggestions.

Note that all content will be covered during the course and it is not necessary to be familiar with the content of these readings in advance (although it will certainly help).


Big Data for the Digital Humanities. Original Source: Produced by Belaid Moa (bmoa@uvic.ca) for this course.


IBM - Big Data, Chapter 4, All About Hadoop: The Big Data Lingo Chapter 4.

Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.2.23
Preface

The strange history of this book

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize jargon and define each term at first use.
- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.


In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.
Since then I’ve continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises.

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.
- I added more exercises, ranging from short tests of understanding to a few substantial projects. Most exercises include a link to my solution.
- I added a series of case studies—longer examples with exercises, solutions, and discussion.
- I expanded the discussion of program development plans and basic design patterns.
- I added appendices about debugging and analysis of algorithms.

The second edition of *Think Python* has these new features:

- The book and all supporting code have been updated to Python 3.
- I added a few sections, and more details on the web, to help beginners get started running Python in a browser, so you don’t have to deal with installing Python until you want to.
- For Chapter 4.1 I switched from my own turtle graphics package, called Swampy, to a more standard Python module, `turtle`, which is easier to install and more powerful.
- I added a new chapter called “The Goodies”, which introduces some additional Python features that are not strictly necessary, but sometimes handy.

I hope you enjoy working with this book, and that it helps you learn to program and think like a computer scientist, at least a little bit.

Allen B. Downey

Olin College

**Acknowledgments**

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

Thanks also to Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and Creative Commons for the license I am using now.
Thanks to the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

Thanks to the editors at O’Reilly Media who worked on *Think Python*.

Thanks to all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

**Contributor List**

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to feedback@thinkpython.com. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word “unconsciously” in Chapter 1 needed to be changed to “subconsciously”.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
• Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.

• John Ouzts corrected the definition of “return value” in Chapter 3.

• Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.

• David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.

• Michael Schmitt sent in a correction to the chapter on files and exceptions.

• Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.

• Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen’s Perl script that generates HTML from LaTeX.

• Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.

• Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.

• Keith Verheyden sent in a correction in Chapter 3.

• Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.

• Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.

• Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.

• Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between `gleich` and `selbe`.

• James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.

• Hayden McAfee caught a potentially confusing inconsistency between two examples.

• Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.

• Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.

• Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.

• Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.

• Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.

• Christopher P. Smith caught several typos and helped us update the book for Python 2.2.

• David Hutchins caught a typo in the Foreword.

• Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
• Julie Peters caught a typo in the Preface.
• Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
• D. J. Webre suggested a clarification in Chapter 3.
• Ken found a fistful of errors in Chapters 8, 9 and 11.
• Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
• Curtis Yanko suggested a clarification in Chapter 2.
• Ben Logan sent in a number of typos and problems with translating the book into HTML.
• Jason Armstrong saw the missing word in Chapter 2.
• Louis Cordier noticed a spot in Chapter 16 where the code didn’t match the text.
• Brian Cain suggested several clarifications in Chapters 2 and 3.
• Rob Black sent in a passel of corrections, including some changes for Python 2.2.
• Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
• Jason Mader at George Washington University made a number of useful suggestions and corrections.
• Jan Gundtofte-Bruun reminded us that “a error” is an error.
• Abel David and Alexis Dinno reminded us that the plural of “matrix” is “matrices”, not “matrixes”. This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
• Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of “argument” and “parameter”.
• Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
• Sam Bull pointed out a confusing paragraph in Chapter 2.
• Andrew Cheung pointed out two instances of “use before def”.
• C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
• Alessandra helped clear up some Turtle confusion.
• Wim Champagne found a brain-o in a dictionary example.
• Douglas Wright pointed out a problem with floor division in `arc`.
• Jared Spindor found some jetsam at the end of a sentence.
• Lin Peiheng sent a number of very helpful suggestions.
• Ray Hagtvedt sent in two errors and a not-quite-error.
• Torsten Hübsch pointed out an inconsistency in Swampy.
• Inga Petuhhov corrected an example in Chapter 14.
• Arne Babenhauserheide sent several helpful corrections.
• Mark E. Casida is good at spotting repeated words.
• Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
• Gordon Shephard sent in several corrections, all in separate emails.
• Andrew Turner spotted an error in Chapter 8.
• Adam Hobart fixed a problem with floor division in arc.
• Daryl Hammond and Sarah Zimmerman pointed out that I served up \texttt{math.pi} too early. And Zim spotted a typo.
• George Sass found a bug in a Debugging section.
• Brian Bingham suggested Exercise \ref{sec:11.5}.
• Leah Engelbert-Fenton pointed out that I used \texttt{tuple} as a variable name, contrary to my own advice. And then found a bunch of typos and a “use before def”.
• Joe Funke spotted a typo.
• Chao-chao Chen found an inconsistency in the Fibonacci example.
• Jeff Paine knows the difference between space and spam.
• Lubos Pintes sent in a typo.
• Gregg Lind and Abigail Heithoff suggested Exercise \ref{sec:14.3}.
• Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary \textit{Concrete Abstractions}, which you might want to read when you are done with this book.
• Chotipat Pornavalai found an error in an error message.
• Stanislaw Antol sent a list of very helpful suggestions.
• Eric Pashman sent a number of corrections for Chapters 4–11.
• Miguel Azevedo found some typos.
• Jianhua Liu sent in a long list of corrections.
• Nick King found a missing word.
• Martin Zuther sent a long list of suggestions.
• Adam Zimmerman found an inconsistency in my instance of an “instance” and several other errors.
• Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
• Anurag Goel suggested another solution for \texttt{is_abecedarian} and sent some additional corrections. And he knows how to spell Jane Austen.
• Kelli Kratzter spotted one of the typos.
• Mark Griffiths pointed out a confusing example in Chapter 3.
• Roydan Ongie found an error in my Newton’s method.
• Patryk Wolowiec helped me with a problem in the HTML version.
Mark Chonofsky told me about a new keyword in Python 3.

Russell Coleman helped me with my geometry.

Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn’t mention it by name.

Stéphane Morin sent in several corrections and suggestions.

Paul Stoop corrected a typo in `uses_only`.

Eric Bronner pointed out a confusion in the discussion of the order of operations.

Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!

Gray Thomas knows his right from his left.

Giovanni Escobar Sosa sent a long list of corrections and suggestions.

Daniel Neilson corrected an error about the order of operations.

Will McGinnis pointed out that `polyline` was defined differently in two places.

Frank Hecker pointed out an exercise that was under-specified, and some broken links.

Animesh B helped me clean up a confusing example.

Martin Caspersen found two round-off errors.

Gregor Ulm sent several corrections and suggestions.

Dimitrios Tsirigkas suggested I clarify an exercise.

Carlos Tafur sent a page of corrections and suggestions.

Martin Nordsletten found a bug in an exercise solution.

Sven Hoexter pointed out that a variable named `input` shadows a build-in function.

Stephen Gregory pointed out the problem with `cmp` in Python 3.

Ishwar Bhat corrected my statement of Fermat’s last theorem.

Andrea Zanella translated the book into Italian, and sent a number of corrections along the way.

Many, many thanks to Melissa Lewis and Luciano Ramalho for excellent comments and suggestions on the second edition.

Thanks to Harry Percival from PythonAnywhere for his help getting people started running Python in a browser.

William Murray corrected my definition of floor division.

Per Starbäck brought me up to date on universal newlines in Python 3.

In addition, people who spotted typos or made corrections include Czeslaw Czapla, Dale Wilson, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, and Eric Ransom.
## Contents

1. **Preface**                                    v

1. **The way of the program**                   1
   1.1 What is a program?                        1
   1.2 Running Python                            2
   1.3 The first program                         3
   1.4 Arithmetic operators                      3
   1.5 Values and types                          4
   1.6 Formal and natural languages              4
   1.7 Debugging                                 6
   1.8 Glossary                                  6
   1.9 Exercises                                 7

2. **Variables, expressions and statements**    9
   2.1 Assignment statements                    9
   2.2 Variable names                            9
   2.3 Expressions and statements                10
   2.4 Script mode                               11
   2.5 Order of operations                       12
   2.6 String operations                         12
   2.7 Comments                                  13
   2.8 Debugging                                 13
   2.9 Glossary                                  14
   2.10 Exercises                                15
# Contents

## 5 Conditionals and recursion  
5.1 Floor division and modulus ................................. 39
5.2 Boolean expressions ........................................ 40
5.3 Logical operators ................................ .......... 40
5.4 Conditional execution ....................................... 41
5.5 Alternative execution ....................................... 41
5.6 Chained conditionals ....................................... 41
5.7 Nested conditionals ....................................... 42
5.8 Recursion .................................................... 43
5.9 Stack diagrams for recursive functions .................... 44
5.10 Infinite recursion .......................................... 44
5.11 Keyboard input ............................................ 45
5.12 Debugging .................................................. 46
5.13 Glossary ................................................... 47
5.14 Exercises .................................................. 47

## 6 Fruitful functions  
6.1 Return values .............................................. 51
6.2 Incremental development .................................... 52
6.3 Composition ................................................ 54
6.4 Boolean functions ......................................... 54
6.5 More recursion ............................................ 55
6.6 Leap of faith ............................................... 57
6.7 One more example ......................................... 57
6.8 Checking types ............................................. 58
6.9 Debugging .................................................. 59
6.10 Glossary ................................................... 60
6.11 Exercises .................................................. 60
7 Iteration

7.1 Reassignment .................................................. 63
7.2 Updating variables ............................................. 64
7.3 The while statement .......................................... 64
7.4 break .................................................................. 66
7.5 Square roots ...................................................... 66
7.6 Algorithms .......................................................... 67
7.7 Debugging ............................................................ 68
7.8 Glossary .............................................................. 68
7.9 Exercises ............................................................. 69

8 Strings ................................................................ 71

8.1 A string is a sequence .......................................... 71
8.2 len ..................................................................... 72
8.3 Traversal with a for loop ...................................... 72
8.4 String slices ....................................................... 73
8.5 Strings are immutable ......................................... 74
8.6 Searching ............................................................ 74
8.7 Looping and counting .......................................... 75
8.8 String methods .................................................... 75
8.9 The in operator .................................................... 76
8.10 String comparison ............................................... 77
8.11 Debugging .......................................................... 77
8.12 Glossary ............................................................. 79
8.13 Exercises ............................................................ 79

9 Case study: word play .......................................... 83

9.1 Reading word lists ............................................... 83
9.2 Exercises ............................................................. 84
9.3 Search ................................................................ 85
9.4 Looping with indices .......................................... 86
9.5 Debugging ............................................................ 87
9.6 Glossary .............................................................. 87
9.7 Exercises ............................................................. 88
## 12 Tuples

12.1 Tuples are immutable  
115
12.2 Tuple assignment  
116
12.3 Tuples as return values  
117
12.4 Variable-length argument tuples  
118
12.5 Lists and tuples  
118
12.6 Dictionaries and tuples  
120
12.7 Sequences of sequences  
121
12.8 Debugging  
122
12.9 Glossary  
122
12.10 Exercises  
123

## 13 Case study: data structure selection

13.1 Word frequency analysis  
125
13.2 Random numbers  
126
13.3 Word histogram  
127
13.4 Most common words  
128
13.5 Optional parameters  
129
13.6 Dictionary subtraction  
129
13.7 Random words  
130
13.8 Markov analysis  
130
13.9 Data structures  
132
13.10 Debugging  
133
13.11 Glossary  
134
13.12 Exercises  
134

## 14 Files

14.1 Persistence  
137
14.2 Reading and writing  
137
14.3 Format operator  
138
14.4 Filenames and paths  
139
14.5 Catching exceptions  
140
## Contents

### 14.6 Databases

### 14.7 Pickling

### 14.8 Pipes

### 14.9 Writing modules

### 14.10 Debugging

### 14.11 Glossary

### 14.12 Exercises

### 15 Classes and objects

#### 15.1 Programmer-defined types

#### 15.2 Attributes

#### 15.3 Rectangles

#### 15.4 Instances as return values

#### 15.5 Objects are mutable

#### 15.6 Copying

#### 15.7 Debugging

#### 15.8 Glossary

#### 15.9 Exercises

### 16 Classes and functions

#### 16.1 Time

#### 16.2 Pure functions

#### 16.3 Modifiers

#### 16.4 Prototyping versus planning

#### 16.5 Debugging

#### 16.6 Glossary

#### 16.7 Exercises

### 17 Classes and methods

#### 17.1 Object-oriented features

#### 17.2 Printing objects

#### 17.3 Another example
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.4</td>
<td>A more complicated example</td>
<td>164</td>
</tr>
<tr>
<td>17.5</td>
<td>The <code>init</code> method</td>
<td>164</td>
</tr>
<tr>
<td>17.6</td>
<td>The <code>str</code> method</td>
<td>165</td>
</tr>
<tr>
<td>17.7</td>
<td>Operator overloading</td>
<td>165</td>
</tr>
<tr>
<td>17.8</td>
<td>Type-based dispatch</td>
<td>166</td>
</tr>
<tr>
<td>17.9</td>
<td>Polymorphism</td>
<td>167</td>
</tr>
<tr>
<td>17.10</td>
<td>Debugging</td>
<td>168</td>
</tr>
<tr>
<td>17.11</td>
<td>Interface and implementation</td>
<td>169</td>
</tr>
<tr>
<td>17.12</td>
<td>Glossary</td>
<td>169</td>
</tr>
<tr>
<td>17.13</td>
<td>Exercises</td>
<td>170</td>
</tr>
<tr>
<td>18</td>
<td>Inheritance</td>
<td>171</td>
</tr>
<tr>
<td>18.1</td>
<td>Card objects</td>
<td>171</td>
</tr>
<tr>
<td>18.2</td>
<td>Class attributes</td>
<td>172</td>
</tr>
<tr>
<td>18.3</td>
<td>Comparing cards</td>
<td>173</td>
</tr>
<tr>
<td>18.4</td>
<td>Decks</td>
<td>174</td>
</tr>
<tr>
<td>18.5</td>
<td>Printing the deck</td>
<td>174</td>
</tr>
<tr>
<td>18.6</td>
<td>Add, remove, shuffle and sort</td>
<td>175</td>
</tr>
<tr>
<td>18.7</td>
<td>Inheritance</td>
<td>176</td>
</tr>
<tr>
<td>18.8</td>
<td>Class diagrams</td>
<td>177</td>
</tr>
<tr>
<td>18.9</td>
<td>Debugging</td>
<td>178</td>
</tr>
<tr>
<td>18.10</td>
<td>Data encapsulation</td>
<td>179</td>
</tr>
<tr>
<td>18.11</td>
<td>Glossary</td>
<td>180</td>
</tr>
<tr>
<td>18.12</td>
<td>Exercises</td>
<td>181</td>
</tr>
<tr>
<td>19</td>
<td>The Goodies</td>
<td>183</td>
</tr>
<tr>
<td>19.1</td>
<td>Conditional expressions</td>
<td>183</td>
</tr>
<tr>
<td>19.2</td>
<td>List comprehensions</td>
<td>184</td>
</tr>
<tr>
<td>19.3</td>
<td>Generator expressions</td>
<td>185</td>
</tr>
<tr>
<td>19.4</td>
<td><code>any</code> and <code>all</code></td>
<td>185</td>
</tr>
<tr>
<td>19.5</td>
<td>Sets</td>
<td>186</td>
</tr>
<tr>
<td>19.6</td>
<td>Counters</td>
<td>187</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>19.7</td>
<td>defaultdict</td>
<td>188</td>
</tr>
<tr>
<td>19.8</td>
<td>Named tuples</td>
<td>189</td>
</tr>
<tr>
<td>19.9</td>
<td>Gathering keyword args</td>
<td>190</td>
</tr>
<tr>
<td>19.10</td>
<td>Glossary</td>
<td>191</td>
</tr>
<tr>
<td>19.11</td>
<td>Exercises</td>
<td>192</td>
</tr>
<tr>
<td>A</td>
<td>Debugging</td>
<td>193</td>
</tr>
<tr>
<td>A.1</td>
<td>Syntax errors</td>
<td>193</td>
</tr>
<tr>
<td>A.2</td>
<td>Runtime errors</td>
<td>195</td>
</tr>
<tr>
<td>A.3</td>
<td>Semantic errors</td>
<td>198</td>
</tr>
<tr>
<td>B</td>
<td>Analysis of Algorithms</td>
<td>201</td>
</tr>
<tr>
<td>B.1</td>
<td>Order of growth</td>
<td>202</td>
</tr>
<tr>
<td>B.2</td>
<td>Analysis of basic Python operations</td>
<td>204</td>
</tr>
<tr>
<td>B.3</td>
<td>Analysis of search algorithms</td>
<td>205</td>
</tr>
<tr>
<td>B.4</td>
<td>Hashtables</td>
<td>206</td>
</tr>
<tr>
<td>B.5</td>
<td>Glossary</td>
<td>209</td>
</tr>
</tbody>
</table>
Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That’s why this chapter is called, “The way of the program”.

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, the network, or some other device.
output: Display data on the screen, save it in a file, send it over the network, etc.
math: Perform basic mathematical operations like addition and multiplication.
conditional execution: Check for certain conditions and run the appropriate code.
repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that’s pretty much all there is to it. Every program you’ve ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

1.2 Running Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I’ll make suggestions for installing Python on your computer.

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started at http://tinyurl.com/thinkpython2e

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

The Python interpreter is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing python on a command line. When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The first three lines contain information about the interpreter and the operating system it’s running on, so it might be different for you. But you should check that the version number, which is 3.4.0 in this example, begins with 3, which indicates that you are running Python 3. If it begins with 2, you are running (you guessed it) Python 2.

The last line is a prompt that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

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

Now you’re ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.
1.3 The first program

Traditionally, the first program you write in a new language is called “Hello, World!” because all it does is display the words “Hello, World!”. In Python, it looks like this:

```python
>>> print('Hello, World!')
```

This is an example of a print statement, although it doesn’t actually print anything on paper. It displays a result on the screen. In this case, the result is the words Hello, World!

The quotation marks in the program mark the beginning and end of the text to be displayed; they don’t appear in the result.

The parentheses indicate that print is a function. We’ll get to functions in Chapter 3.

In Python 2, the print statement is slightly different; it is not a function, so it doesn’t use parentheses.

```python
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that’s enough to get started.

1.4 Arithmetic operators

After “Hello, World”, the next step is arithmetic. Python provides operators, which are special symbols that represent computations like addition and multiplication.

The operators +, -, and * perform addition, subtraction, and multiplication, as in the following examples:

```python
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

The operator / performs division:

```python
>>> 84 / 2
42.0
```

You might wonder why the result is 42.0 instead of 42. I’ll explain in the next section.

Finally, the operator ** performs exponentiation; that is, it raises a number to a power:

```python
>>> 6**2 + 6
42
```

In some other languages, ^ is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

```python
>>> 6 ^ 2
4
```

I won’t cover bitwise operators in this book, but you can read about them at [http://wiki.python.org/moin/BitwiseOperators](http://wiki.python.org/moin/BitwiseOperators).
1.5 Values and types

A value is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are 2, 42.0, and 'Hello, World!'.

These values belong to different types: 2 is an integer, 42.0 is a floating-point number, and 'Hello, World!' is a string, so-called because the letters it contains are strung together.

If you are not sure what type a value has, the interpreter can tell you:

```python
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word “class” is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type int, strings belong to str and floating-point numbers belong to float.

What about values like '2' and '42.0'? They look like numbers, but they are in quotation marks like strings.

```python
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

They’re strings.

When you type a large integer, you might be tempted to use commas between groups of digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```python
>>> 1,000,000
(1, 0, 0)
```

That’s not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We’ll learn more about this kind of sequence later.

1.6 Formal and natural languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.
Formal languages tend to have strict syntax rules that govern the structure of statements. For example, in mathematics the statement $3 + 3 = 6$ has correct syntax, but $3+ = 3$ is not. In chemistry $H_2O$ is a syntactically correct formula, but $Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3+ = 3\$/ is that $\$ is not a legal token in mathematics (at least as far as I know). Similarly, $Zz$ is not legal because there is no element with the abbreviation $Zz$.

The second type of syntax rule pertains to the way tokens are combined. The equation $3+ = 3$ is illegal because even though $+$ and $=$ are legal tokens, you can’t have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is a well-structured English sentence with invalid tokens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called parsing.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

- **ambiguity**: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

- **redundancy**: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

- **literalness**: Natural languages are full of idiom and metaphor. If I say, “The penny dropped”, there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

- **Poetry**: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

- **Prose**: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

- **Programs**: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.
Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.7 Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

1.8 Glossary

**problem solving:** The process of formulating a problem, finding a solution, and expressing it.

**high-level language:** A programming language like Python that is designed to be easy for humans to read and write.

**low-level language:** A programming language that is designed to be easy for a computer to run; also called “machine language” or “assembly language”.

**portability:** A property of a program that can run on more than one kind of computer.

**interpreter:** A program that reads another program and executes it

**prompt:** Characters displayed by the interpreter to indicate that it is ready to take input from the user.

**program:** A set of instructions that specifies a computation.
print statement: An instruction that causes the Python interpreter to display a value on the screen.

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

value: One of the basic units of data, like a number or string, that a program manipulates.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

string: A type that represents sequences of characters.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

syntax: The rules that govern the structure of a program.

parse: To examine a program and analyze the syntactic structure.

bug: An error in a program.

debugging: The process of finding and correcting bugs.

1.9 Exercises

Exercise 1.1. It is a good idea to read this book in front of a computer so you can try out the examples as you go.

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

1. In a print statement, what happens if you leave out one of the parentheses, or both?

2. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?

3. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2+2`?
4. In math notation, leading zeros are ok, as in 02. What happens if you try this in Python?
5. What happens if you have two values with no operator between them?

Exercise 1.2. Start the Python interpreter and use it as a calculator.

1. How many seconds are there in 42 minutes 42 seconds?

2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.

3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?
Chapter 2

Variables, expressions and statements

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

2.1 Assignment statements

An assignment statement creates a new variable and gives it a value:

```python
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer 17 to `n`; the third assigns the (approximate) value of π to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a state diagram because it shows what state each of the variables is in (think of it as the variable’s state of mind). Figure 2.1 shows the result of the previous example.

2.2 Variable names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

```
message  --->  'And now for something completely different'
            n  --->  17
            pi  --->  3.141592653589793
```

Figure 2.1: State diagram.
Variable names can be as long as you like. They can contain both letters and numbers, but they can’t begin with a number. It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.

The underscore character, _, can appear in a name. It is often used in names with multiple words, such as your name or airspeed_of_unladen_swallow.

If you give a variable an illegal name, you get a syntax error:

```python
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what’s wrong with class?

It turns out that class is one of Python’s keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python 3 has these keywords:

```plaintext
False    class    finally    is     return
None     continue for    lambda    try
True     def     from     nonlocal    while
and      del     global    not     with
as       elif    if       or     yield
assert   else    import    pass    raise
break    except  in       raise
```

You don’t have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you’ll know.

### 2.3 Expressions and statements

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```python
>>> 42
42
>>> n
17
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this example, n has the value 17 and n + 25 has the value 42.

A statement is a unit of code that has an effect, like creating a variable or displaying a value.

```python
>>> n = 17
>>> print(n)
```
2.4 Script mode

The first line is an assignment statement that gives a value to \( n \). The second line is a print statement that displays the value of \( n \).

When you type a statement, the interpreter **executes** it, which means that it does whatever the statement says. In general, statements don’t have values.

### 2.4 Script mode

So far we have run Python in **interactive mode**, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a **script** and then run the interpreter in **script mode** to execute the script. By convention, Python scripts have names that end with `.py`.

If you know how to create and run a script on your computer, you are ready to go. Otherwise I recommend using PythonAnywhere again. I have posted instructions for running in script mode at [http://tinyurl.com/thinkpython2e](http://tinyurl.com/thinkpython2e).

Because Python provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Python as a calculator, you might type

```python
>>> miles = 26.2
>>> miles * 1.61
42.182
```

The first line assigns a value to `miles`, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python actually evaluates the expression, but it doesn’t display the value unless you tell it to:

```python
miles = 26.2
print(miles * 1.61)
```

This behavior can be confusing at first.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```python
print(1)
x = 2
print(x)
```

produces the output

1
2
The assignment statement produces no output.

To check your understanding, type the following statements in the Python interpreter and see what they do:

```python
5
x = 5
x + 1
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

### 2.5 Order of operations

When an expression contains more than one operator, the order of evaluation depends on the **order of operations**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, \(2 \times (3-1)\) is 4, and \((1+1)\times(5-2)\) is 8. You can also use parentheses to make an expression easier to read, as in \((\text{minute} \times 100) / 60\), even if it doesn’t change the result.

- Exponentiation has the next highest precedence, so \(1 + 2 \times 3 \times 3\) is 18, not 36.

- Multiplication and Division have higher precedence than Addition and Subtraction. So \(2 \times 3 - 1\) is 5, not 4, and \(6+4/2\) is 8, not 5.

- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression \(\text{degrees} / 2 \times \pi\), the division happens first and the result is multiplied by \(\pi\). To divide by \(2\pi\), you can use parentheses or write \(\text{degrees} / 2 / \pi\).

I don’t work very hard to remember the precedence of operators. If I can’t tell by looking at the expression, I use parentheses to make it obvious.

### 2.6 String operations

In general, you can’t perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```python
'2'-'1'  'eggs'/'easy'  'third'+'a charm'
```

But there are two exceptions, + and *. The + operator performs **string concatenation**, which means it joins the strings by linking them end-to-end. For example:

```python
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```
2.7. Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60  # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is more useful to explain why.

This comment is redundant with the code and useless:

```
v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

2.8 Debugging

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

**Syntax error:** “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so \((1 + 2)\) is legal, but \(1 + 2\) is a syntax error.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few
weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

**Runtime error:** The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

**Semantic error:** The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

### 2.9 Glossary

**variable:** A name that refers to a value.

**assignment:** A statement that assigns a value to a variable.

**state diagram:** A graphical representation of a set of variables and the values they refer to.

**keyword:** A reserved word that is used to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

**operand:** One of the values on which an operator operates.

**expression:** A combination of variables, operators, and values that represents a single result.

**evaluate:** To simplify an expression by performing the operations in order to yield a single value.

**statement:** A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

**execute:** To run a statement and do what it says.

**interactive mode:** A way of using the Python interpreter by typing code at the prompt.

**script mode:** A way of using the Python interpreter to read code from a script and run it.

**script:** A program stored in a file.

**order of operations:** Rules governing the order in which expressions involving multiple operators and operands are evaluated.

**concatenate:** To join two operands end-to-end.
comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

exception: An error that is detected while the program is running.

semantics: The meaning of a program.

semantic error: An error in a program that makes it do something other than what the programmer intended.

2.10 Exercises

Exercise 2.1. Repeating my advice from the previous chapter, whenever you learn a new feature, you should try it out in interactive mode and make errors on purpose to see what goes wrong.

• We’ve seen that \( n = 42 \) is legal. What about \( 42 = n \)?
• How about \( x = y = 1 \)?
• In some languages every statement ends with a semi-colon, ;. What happens if you put a semi-colon at the end of a Python statement?
• What if you put a period at the end of a statement?
• In math notation you can multiply \( x \) and \( y \) like this: \( xy \). What happens if you try that in Python?

Exercise 2.2. Practice using the Python interpreter as a calculator:

1. The volume of a sphere with radius \( r \) is \( \frac{4}{3} \pi r^3 \). What is the volume of a sphere with radius 5?

2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?

3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?
Chapter 3

Functions

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name.

3.1 Function calls

We have already seen one example of a function call:

```python
>>> type(42)
<class 'int'>
```

The name of the function is type. The expression in parentheses is called the argument of the function. The result, for this function, is the type of the argument.

It is common to say that a function “takes” an argument and “returns” a result. The result is also called the return value.

Python provides functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```python
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

```python
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

```python
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```
Finally, `str` converts its argument to a string:

```python
g>>> str(32)  
'32'
```

```python
g>>> str(3.14159)  
'3.14159'
```

### 3.2 Math functions

Python has a math module that provides most of the familiar mathematical functions. A `module` is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an `import` statement:

```python
g>>> import math
```

This statement creates a `module object` named `math`. If you display the module object, you get some information about it:

```python
g>>> math
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called `dot notation`.

```python
g>>> ratio = signal_power / noise_power
```

```python
g>>> decibels = 10 * math.log10(ratio)
```

```python
g>>> radians = 0.7
```

```python
g>>> height = math.sin(radians)
```

The first example uses `math.log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The math module also provides `log`, which computes logarithms base `e`.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 180 and multiply by $\pi$:

```python
g>>> degrees = 45
```

```python
g>>> radians = degrees / 180.0 * math.pi
```

```python
g>>> math.sin(radians)
0.70710678118
```

The expression `math.pi` gets the variable `pi` from the math module. Its value is a floating-point approximation of $\pi$, accurate to about 15 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```python
g>>> math.sqrt(2) / 2.0
```

```python
0.70710678118
``
3.3 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

\[ x = \text{math.sin}(\text{degrees} / 360.0 * 2 * \text{math.pi}) \]

And even function calls:

\[ x = \text{math.exp}(\text{math.log}(x+1)) \]

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

```python
>>> minutes = hours * 60  # right
>>> hours * 60 = minutes   # wrong!
SyntaxError: can't assign to operator
```

3.4 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that run when the function is called.

Here is an example:

```python
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can’t be a number. You can’t use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn’t take any arguments.

The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

All quotation marks (single and double) must be “straight quotes”, usually located next to Enter on the keyboard. “Curly quotes”, like the ones in this sentence, are not legal in Python.

If you type a function definition in interactive mode, the interpreter prints dots (\ldots) to let you know that the definition isn’t complete:
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...

To end the function, you have to enter an empty line.

Defining a function creates a function object, which has type function:

```python
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

The syntax for calling the new function is the same as for built-in functions:

```python
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```python
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call repeat_lyrics:

```python
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

### 3.5 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```python
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

This program contains two function definitions: print_lyrics and repeat_lyrics. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.
3.6 Flow of execution

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Now move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

3.6 Flow of execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don’t run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don’t always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

3.7 Parameters and arguments

Some of the functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

```python
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359

The same rules of composition that apply to built-in functions also apply to programmer-defined functions, so we can use any kind of expression as an argument for `print_twice`:

>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0

The argument is evaluated before the function is called, so in the examples the expressions 'Spam '*4 and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody bruce.

### 3.8 Variables and parameters are local

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```python
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```python
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:
3.9 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1.

The frames are arranged in a stack that indicates which function called which, and so on. In this example, print_twice was called by cat_twice, and cat_twice was called by __main__, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to __main__.

Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as line1, part2 has the same value as line2, and bruce has the same value as cat.

If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called that, all the way back to __main__.

For example, if you try to access cat from within print_twice, you get a NameError.

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside print_twice, there is no such thing as bruce.
This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

### 3.10 Fruitful functions and void functions

Some of the functions we have used, such as the math functions, return results; for lack of a better name, I call them fruitful functions. Other functions, like print_twice, perform an action but don’t return a value. They are called void functions.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```python
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```plaintext
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

```python
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn’t store or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don’t have a return value. If you assign the result to a variable, you get a special value called None.

```plaintext
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value None is not the same as the string 'None'. It is a special value that has its own type:

```plaintext
>>> type(None)
<class 'NoneType'>
```

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

### 3.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:
3.12 Debugging

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

3.12 Debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a working program and make small modifications, debugging them as you go.

For example, Linux is an operating system that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux.” (*The Linux Users’ Guide* Beta Version 1).

3.13 Glossary

**function:** A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

**function definition:** A statement that creates a new function, specifying its name, parameters, and the statements it contains.

**function object:** A value created by a function definition. The name of the function is a variable that refers to a function object.

**header:** The first line of a function definition.
body: The sequence of statements inside a function definition.

parameter: A name used inside a function to refer to the value passed as an argument.

function call: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

fruitful function: A function that returns a value.

void function: A function that always returns None. None A special value returned by void functions.

module: A file that contains a collection of related functions and other definitions.

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an import statement that provides access to the values defined in a module.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution: The order statements run in.

stack diagram: A graphical representation of a stack of functions, their variables, and the values they refer to.

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback: A list of the functions that are executing, printed when an exception occurs.

3.14 Exercises

Exercise 3.1. Write a function named right_justify that takes a string named s as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

>>> right_justify('monty')

monty

Hint: Use string concatenation and repetition. Also, Python provides a built-in function called len that returns the length of a string, so the value of len('monty') is 5.
Exercise 3.2. A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

```python
def do_twice(f):
    f()
    f()
```

Here's an example that uses `do_twice` to call a function named `print_spam` twice.

```python
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. Type this example into a script and test it.
2. Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `print_twice` from earlier in this chapter to your script.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing 'spam' as an argument.
5. Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Solution: [http://thinkpython2.com/code/do_four.py](http://thinkpython2.com/code/do_four.py)

Exercise 3.3. Note: This exercise should be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```
+ - - - - + - - - - +
|       |       |
|       |       |
|       |       |
+ - - - - + - - - - +
|       |       |
|       |       |
+ - - - - + - - - - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

```python
print('+', ', '-'
```

By default, `print` advances to the next line, but you can override that behavior and put a space at the end, like this:

```python
print('+', end=' ')
print('-')
```
The output of these statements is ‘+ -’ on the same line. The output from the next print statement would begin on the next line.

2. Write a function that draws a similar grid with four rows and four columns.

The Unreasonable Effectiveness of Data

Alon Halevy, Peter Norvig, and Fernando Pereira, Google

Eugene Wigner’s article “The Unreasonable Effectiveness of Mathematics in the Natural Sciences” examines why so much of physics can be neatly explained with simple mathematical formulas such as $f = ma$ or $e = mc^2$. Meanwhile, sciences that involve human beings rather than elementary particles have proven more resistant to elegant mathematics. Economists suffer from physics envy over their inability to neatly model human behavior. An informal, incomplete grammar of the English language runs over 1,700 pages. Perhaps when it comes to natural language processing and related fields, we’re doomed to complex theories that will never have the elegance of physics equations. But if that’s so, we should stop acting as if our goal is to author extremely elegant theories, and instead embrace complexity and make use of the best ally we have: the unreasonable effectiveness of data.

One of us, as an undergraduate at Brown University, remembers the excitement of having access to the Brown Corpus, containing one million English words. Since then, our field has seen several notable corpora that are about 100 times larger, and in 2006, Google released a trillion-word corpus with frequency counts for all sequences up to five words long. In some ways this corpus is a step backwards from the Brown Corpus: it’s taken from unfiltered Web pages and thus contains incomplete sentences, spelling errors, grammatical errors, and all sorts of other errors. It’s not annotated with carefully hand-corrected part-of-speech tags. But the fact that it’s a million times larger than the Brown Corpus outweighs these drawbacks. A trillion-word corpus—along with other Web-derived corpora of millions, billions, or trillions of links, videos, images, tables, and user interactions—captures even very rare aspects of human behavior. So, this corpus could serve as the basis of a complete model for certain tasks—if only we knew how to extract the model from the data.

Learning from Text at Web Scale

The biggest successes in natural-language-related machine learning have been statistical speech recognition and statistical machine translation. The reason for these successes is not that these tasks are easier than other tasks; they are in fact much harder than tasks such as document classification that extract just a few bits of information from each document. The reason is that translation is a natural task routinely done every day for a real human need (think of the operations of the European Union or of news agencies). The same is true of speech transcription (think of closed-caption broadcasts). In other words, a large training set of the input-output behavior that we seek to automate is available to us in the wild. In contrast, traditional natural language processing problems such as document classification, part-of-speech tagging, named-entity recognition, or parsing are not routine tasks, so they have no large corpus available in the wild. Instead, a corpus for these tasks requires skilled human annotation. Such annotation is not only slow and expensive to acquire but also difficult for experts to agree on, being bedeviled by many of the difficulties we discuss later in relation to the Semantic Web. The first lesson of Web-scale learning is to use available large-scale data rather than hoping for annotated data that isn’t available. For instance, we find that useful semantic relationships can be automatically learned from the statistics of search queries and the corresponding results or from the accumulated evidence of Web-based text patterns and formatted tables, in both cases without needing any manually annotated data.
Another important lesson from statistical methods in speech recognition and machine translation is that memorization is a good policy if you have a lot of training data. The statistical language models that are used in both tasks consist primarily of a huge database of probabilities of short sequences of consecutive words (n-grams). These models are built by counting the number of occurrences of each n-gram sequence from a corpus of billions or trillions of words. Researchers have done a lot of work in estimating the probabilities of new n-grams from the frequencies of observed n-grams (using, for example, Good-Turing or Kneser-Ney smoothing), leading to elaborate probabilistic models. But invariably, simple models and a lot of data trump more elaborate models based on less data. Similarly, early work on machine translation relied on elaborate rules for the relationships between syntactic and semantic patterns in the source and target languages. Currently, statistical translation models consist mostly of large memorized phrase tables that give candidate mappings between specific source- and target-language phrases.

Instead of assuming that general patterns are more effective than memorizing specific phrases, today’s translation models introduce general rules only when they improve translation over just memorizing particular phrases (for instance, in rules for dates and numbers). Similar observations have been made in every other application of machine learning to Web data: simple n-gram models or linear classifiers based on millions of specific features perform better than elaborate models that try to discover general rules. In many cases there appears to be a threshold of sufficient data. For example, James Hays and Alexei A. Efros addressed the task of scene completion: removing an unwanted, unsightly automobile or ex-spouse from a photograph and filling in the background with pixels taken from a large corpus of other photos. With a corpus of thousands of photos, the results were poor. But once they accumulated millions of photos, the same algorithm performed quite well.

Human language has evolved over millennia to have words for the important concepts; let’s use them. Abstract representations (such as clusters from latent analysis) that lack linguistic counterparts are hard to learn or validate and tend to lose information. Relying on overt statistics of words and word co-occurrences has the further advantage that we can estimate models in an amount of time proportional to available data and can often parallelize them easily. So, learning from the Web becomes naturally scalable.

The success of n-gram models has unfortunately led to a false dichotomy. Many people now believe there are only two approaches to natural language processing:

- a deep approach that relies on hand-coded grammars and ontologies, represented as complex networks of relations; and
- a statistical approach that relies on learning n-gram statistics from large corpora.

In reality, three orthogonal problems arise:

- choosing a representation language,
- encoding a model in that language, and
- performing inference on the model.

Each problem can be addressed in several ways, resulting in dozens of approaches. The deep approach that was popular in the 1980s used first-order logic (or something similar) as the representation language, encoded a model with the labor of a team of graduate students, and did inference with complex inference rules appropriate to the representation language. In the 1980s and 90s, it became fashionable to...
use finite state machines as the representation language, use counting and smoothing over a large corpus to encode a model, and use simple Bayesian statistics as the inference method.

But many other combinations are possible, and in the 2000s, many are being tried. For example, Lise Getoor and Ben Taskar collect work on statistical relational learning—that is, representation languages that are powerful enough to represent relations between objects (such as first-order logic) but that have a sound, probabilistic definition that allows models to be built by statistical learning.9 Taskar and his colleagues show how the same kind of maximum-margin classifier used in support vector machines can improve traditional parsing.9 Stefan Schoenmackers, Oren Etzioni, and Daniel S. Weld show how a relational logic and a 100-million-page corpus can answer questions such as “what vegetables help prevent osteoporosis?” by isolating and combining the relational assertions that “kale is high in calcium” and “calcium helps prevent osteoporosis.”10

**Semantic Web versus Semantic Interpretation**

The Semantic Web is a convention for formal representation languages that lets software services interact with each other “without needing artificial intelligence.”11 A software service that enables us to make a hotel reservation is transformed into a Semantic Web service by agreeing to use one of several standards for representing dates, prices, and locations. The service can then interoperate with other services that use either the same standard or a different one with a known translation into the chosen standard. As Tim Berners-Lee, James Hendler, and Ora Lassila write, “The Semantic Web will enable machines to comprehend semantic documents and data, not human speech and writings.”11 The problem of understanding human speech and writing—the *semantic interpretation problem*—is quite different from the problem of software service interoperability. Semantic interpretation deals with imprecise, ambiguous natural languages, whereas service interoperability deals with making data precise enough that the programs operating on the data will function effectively. Unfortunately, the fact that the word “semantic” appears in both “Semantic Web” and “semantic interpretation” means that the two problems have often been conflated, causing needless and endless consternation and confusion. The “semantics” in Semantic Web services is embodied in the code that implements those services in accordance with the specifications expressed by the relevant ontologies and attached informal documentation. The “semantics” in semantic interpretation of natural languages is instead embodied in human cognitive and cultural processes whereby linguistic expression elicits expected responses and expected changes in cognitive state. Because of a huge shared cognitive and cultural context, linguistic expression can be highly ambiguous and still often be understood correctly.

**Because of a huge shared cognitive and cultural context, linguistic expression can be highly ambiguous and still often be understood correctly.**

Given these challenges, building Semantic Web services is an engineering and sociological challenge. So, even though we understand the required technology, we must deal with significant hurdles:

- **Ontology writing.** The important easy cases have been done. For example, the Dublin Core defines dates, locations, publishers, and other concepts that are sufficient for card catalog entries. Bioformats.org defines chromosomes, species, and gene sequences. Other organizations provide ontologies for their specific fields. But there’s a long tail of rarely used concepts that are too expensive to formalize with current technology. Project Halo did an excellent job of encoding and reasoning with knowledge from a chemistry textbook, but the cost was US$10,000 per page.12 Obviously we can’t afford that cost for a trillion Web pages.

- **Difficulty of implementation.** Publishing a static Web page written in natural language is easy; anyone with a keyboard and Web connection can do it. Creating a database-backed Web service is substantially harder, requiring specialized skills. Making that service compliant with Semantic Web protocols is harder still. Major sites with competent technology experts will find the extra effort worthwhile, but the vast majority of small sites and individuals will find it too difficult, at least with current tools.

- **Competition.** In some domains, competing factions each want to promote their own ontology. In other domains, the entrenched leaders of the field oppose any ontology because it would level the playing field for their competitors. This is a problem in diplomacy, not technology. As Tom Gruber says, “Every ontology is a treaty—a social agreement—among people with some common motive in sharing.”13 When a motive for sharing is lacking, so are common ontologies.

- **Inaccuracy and deception.** We
we're left with a scientific problem of aggregating and indexing all this content. We've solved the technological problem of building a network best where an honest, self-correcting group of cooperative users exists and not as well where competition and deception exist.

The challenges for achieving accurate semantic interpretation are different. We've already solved the sociological problem of building a network infrastructure that has encouraged hundreds of millions of authors to share a trillion pages of content. We've solved the technological problem of aggregating and indexing all this content. But we're left with a scientific problem of interpreting the content, which is mainly that of learning as much as possible about the context of the content to correctly disambiguate it. The semantic interpretation problem remains regardless of whether or not we're using a Semantic Web framework. The same meaning can be expressed in many different ways, and the same expression can express many different meanings. For example, a table of company information might be expressed in ad hoc HTML with column headers called “Company,” “Location,” and so on. Or it could be expressed in a Semantic Web format, with standard identifiers for “Company Name” and “Location,” using the Dublin Core Metadata Initiative point-encoding scheme. But even if we have a formal Semantic Web “Company Name” attribute, we can’t expect to have an ontology for every possible value of this attribute. For example, we can’t know for sure what company the string “Joe’s Pizza” refers to because hundreds of businesses have that name and new ones are being added all the time. We also can’t always tell which business is meant by the string “HP.” It could refer to Helmerich & Payne Corp. when the column is populated by stock ticker symbols but probably refers to Hewlett-Packard when the column is populated by names of large technology companies. The problem of semantic interpretation remains; using a Semantic Web formalism just means that semantic interpretation must be done on shorter strings that fall between angle brackets.

### The same meaning can be expressed in many different ways, and the same expression can express many different meanings.

What we need are methods to infer relationships between column headers or mentions of entities in the world. These inferences may be incorrect at times, but if they’re done well enough we can connect disparate data collections and thereby substantially enhance our interaction with Web data. Interestingly, here too Web-scale data might be an important part of the solution. The Web contains hundreds of millions of independently created tables and possibly a similar number of lists that can be transformed into tables. These tables represent structured data in myriad domains. They also represent how different people organize data—the choices they make for which columns to include and the names given to the columns. The tables also provide a rich collection of column values, and values that they decided belong in the same column of a table. We've never before had such a vast collection of tables (and their schemata) at our disposal to help us resolve semantic heterogeneity. Using such a corpus, we hope to be able to accomplish tasks such as deciding when “Company” and “Company Name” are synonyms, deciding when “HP” means Helmerich & Payne or Hewlett-Packard, and determining that an object with attributes “passengers” and “cruising altitude” is probably an aircraft.

### Examples

How can we use such a corpus of tables? Suppose we want to find synonyms for attribute names—for example, when “Company Name” could be equivalent to “Company” and “price” could be equivalent to “discount”). Such synonyms differ from those in a thesaurus because here, they are highly context dependent (both in tables and in natural language). Given the corpus, we can extract a set of schemata from the tables’ column labels; for example, researchers reliably extracted 2.5 million distinct schemata from a collection of 150 million tables, not all of which had schema. We can now examine the co-occurrences of attribute names in these schemata. If we see a pair of attributes A and B that rarely occur together but always occur with the same other attribute names, this might mean that A and B are synonyms. We can further justify this hypothesis if we see that data elements have a significant overlap or are of the same data type. Similarly, we can also offer a schema autocomplete feature for database designers. For example, by analyzing such a large corpus of schemata, we can discover that schemata that have the attributes Make and Model also tend to have the attributes Year, Color, and Mileage. Providing such feedback to schemata creators can save them time but can also help them use more common attribute names, thereby decreasing a possible
source of heterogeneity in Web-based data. Of course, we’ll find immense opportunities to create interesting data sets if we can automatically combine data from multiple tables in this collection. This is an area of active research.

Another opportunity is to combine data from multiple tables with data from other sources, such as unstructured Web pages or Web search queries. For example, Marius Paşca also considered the task of identifying attributes of classes. That is, his system first identifies classes such as “Company,” then finds examples such as “Adobe Systems,” “Macromedia,” “Apple Computer,” “Target,” and so on, and finally identifies class attributes such as “location,” “CEO,” “headquarters,” “stock price,” and “company profile.” Michael Cafarella and his colleagues showed this can be gleaned from tables, but Paşca showed it can also be extracted from plain text on Web pages and from user queries in search logs. That is, from the user query “Apple Computer stock price” and from the other information we know about existing classes and attributes, we can confirm that “stock price” is an attribute of the “Company” class. Moreover, the technique works not just for a few dozen of the most popular classes but for thousands of classes and tens of thousands of attributes, including classes like “Aircraft Model,” which has attributes “weight,” “length,” “fuel consumption,” “interior photos,” “specifications,” and “seating arrangement.” Paşca shows that including query logs can lead to excellent performance, with 90 percent precision over the top 10 attributes per class.

So, follow the data. Choose a representation that can use unsupervised learning on unlabeled data, which is so much more plentiful than labeled data. Represent all the data with a nonparametric model rather than trying to summarize it with a parametric model, because with very large data sources, the data holds a lot of detail. For natural language applications, trust that human language has already evolved words for the important concepts. See how far you can go by tying together the words that are already there, rather than by inventing new concepts with clusters of words. Now go out and gather some data, and see what it can do.

Choose a representation that can use unsupervised learning on unlabeled data, which is so much more plentiful than labeled data.

References


Alon Halevy is a research scientist at Google. Contact him at halevy@google.com.

Peter Norvig is a research director at Google. Contact him at pnorvig@google.com.

Fernando Pereira is a research director at Google. Contact him at pereira@google.com.
Big Data Now: 2012 Edition

O’Reilly Media, Inc.
Big Data Is Our Generation’s Civil Rights Issue, and We Don’t Know It

By Alistair Croll

Data doesn’t invade people’s lives. Lack of control over how it’s used does.

What’s really driving so-called big data isn’t the volume of information. It turns out big data doesn’t have to be all that big. Rather, it’s about a reconsideration of the fundamental economics of analyzing data.

For decades, there’s been a fundamental tension between three attributes of databases. You can have the data fast; you can have it big; or you can have it varied. The catch is, you can’t have all three at once.
I first heard this as the “three V’s of data”: Volume, Variety, and Velocity. Traditionally, getting two was easy but getting three was very, very, very expensive.

The advent of clouds, platforms like Hadoop, and the inexorable march of Moore’s Law means that now, analyzing data is trivially inexpensive. And when things become so cheap that they’re practically free, big changes happen — just look at the advent of steam power, or the copying of digital music, or the rise of home printing. Abundance replaces scarcity, and we invent new business models.

In the old, data-is-scarce model, companies had to decide what to collect first, and then collect it. A traditional enterprise data warehouse might have tracked sales of widgets by color, region, and size. This act of deciding what to store and how to store it is called designing the schema, and in many ways, it’s the moment where someone decides what the data is about. It’s the instant of context.

That needs repeating:

**You decide what data is about the moment you define its schema.**

With the new, data-is-abundant model, we collect first and ask questions later. The schema comes after the collection. Indeed, big data success stories like Splunk, Palantir, and others are prized because of their ability to make sense of content well after it has been collected — sometimes called a schema-less query. This means we collect information long before we decide what it’s for.

And this is a dangerous thing.

When bank managers tried to restrict loans to residents of certain areas (known as redlining), Congress stepped in to stop it (with the Fair Housing Act of 1968). They were able to legislate against discrimination, making it illegal to change loan policy based on someone’s race.

“Personalization” is another word for discrimination. We’re not discriminating if we tailor things to you based on what we know about you — right? That’s just better service.

In one case, American Express used purchase history to adjust credit limits based on where a customer shopped, despite his excellent credit limit:

Johnson says his jaw dropped when he read one of the reasons American Express gave for lowering his credit limit: “Other customers who have used their card at establishments where you recently shopped have a poor repayment history with American Express.”

We’re seeing the start of this slippery slope everywhere from tailored credit-card limits like this one to car insurance based on driver profiles. In this regard, big data is a civil rights issue, but it’s one that society in general is ill-equipped to deal with.

We’re great at using taste to predict things about people. OKcupid’s 2010 blog post “The Real Stuff White People Like” showed just how easily we can use information to guess at race. It’s a real eye-opener
(and the guys who wrote it didn’t include everything they learned — some of it was a bit too controversial). They simply looked at the words one group used which others didn’t often use. The result was a list of “trigger” words for a particular race or gender.

Now run this backwards. If I know you like these things, or see you mention them in blog posts, on Facebook, or in tweets, then there’s a good chance I know your gender and your race, and maybe even your religion and your sexual orientation. And that I can personalize my marketing efforts towards you.

That makes it a civil rights issue.

If I collect information on the music you listen to, you might assume I will use that data in order to suggest new songs, or share it with your friends. But instead, I could use it to guess at your racial background. And then I could use that data to deny you a loan.

Want another example? Check out Private Data In Public Ways, something I wrote a few months ago after seeing a talk at Big Data London, which discusses how publicly available last name information can be used to generate racial boundary maps:

This TED talk by Malte Spitz does a great job of explaining the challenges of tracking citizens today, and he speculates about whether the Berlin Wall would ever have come down if the Stasi had access to phone records in the way today’s governments do.
So how do we regulate the way data is used?

The only way to deal with this properly is to somehow link *what the data is* with *how it can be used*. I might, for example, say that my musical tastes should be used for song recommendation, but not for banking decisions.

Tying data to permissions can be done through encryption, which is slow, riddled with DRM, burdensome, hard to implement, and bad for innovation. Or it can be done through legislation, which has about as much chance of success as regulating spam: it feels great, but it’s damned hard to enforce.

There are brilliant examples of how a quantified society can improve the way we live, love, work, and play. Big data helps detect disease outbreaks, improve how students learn, reveal political partisanship, and save hundreds of millions of dollars for commuters — to pick just four examples. These are benefits we simply can’t ignore as we try to survive on a planet bursting with people and shaken by climate and energy crises.

But governments need to balance reliance on data with checks and balances about how this reliance erodes privacy and creates civil and moral issues we haven’t thought through. It’s something that most of the electorate isn’t thinking about, and yet it affects every purchase they make.

This should be fun.
Embracing Failure and Learning from the Impostor Syndrome

by Alice Zheng

You can read this post on oreilly.com here.

Lately, there has been a slew of media coverage about the impostor syndrome. Many columnists, bloggers, and public speakers have spoken or written about their own struggles with the impostor syndrome. And original psychological research on the impostor syndrome has found that out of every five successful people, two consider themselves a fraud.

I’m certainly no stranger to the sinking feeling of being out of place. During college and graduate school, it often seemed like everyone else around me was sailing through to the finish line, while I alone lumbered with the weight of programming projects and mathematical proofs. This led to an ongoing self-debate about my choice of a major and profession. One day, I noticed myself reading the same sentence over and over again in a textbook; my eyes were looking at the text, but my mind was saying Why aren’t you getting this yet? It’s so simple. Everybody else gets it. What’s wrong with you?

When I look back on those years, I have two thoughts: first, That was hard, and second, What a waste of perfectly good brain cells! I could have done so many cool things if I had not spent all that time doubting myself.

But one can’t simply snap out of the impostor syndrome. It has a variety of causes, and it’s sticky. I was brought up with the idea of holding myself to a high standard, to measure my own progress against others’ achievements. Falling short of expectations is supposed to be a great motivator for action…or is it?

In practice, measuring one’s own worth against someone else’s achievements can hinder progress more than it helps. It is a flawed method. I have a mathematical analogy for this: When we compare our position against others, we are comparing the static value of functions. But what determines the global optimum of a function are its derivatives. The first derivative measures the speed of change, the second derivative measures how much the speed picks up over time, and so on. How much we can achieve tomorrow is not just determined by where we are today, but how fast we are learning,
changing, and adapting. The rate of change is much more important than a static snapshot of the current position. And yet, we fall into the trap of letting the static snapshots define us.

Computer science is a discipline where the rate of change is particularly important. For one thing, it’s a fast-moving and relatively young field. New things are always being invented. Everyone in the field is continually learning new skills in order to keep up. What’s important today may become obsolete tomorrow. Those who stop learning, stop being relevant.

Even more fundamentally, software programming is about tinkering, and tinkering involves failures. This is why the hacker mentality is so prevalent. We learn by doing, and failing, and re-doing. We learn about good designs by iterating over initial bad designs. We work on pet projects where we have no idea what we are doing, but that teach us new skills. Eventually, we take on bigger, real projects.

Perhaps this is the crux of my position: I’ve noticed a cautiousness and an aversion to failure in myself and many others. I find myself wanting to wrap my mind around a project and perfectly understand its ins and outs before I feel comfortable diving in. I want to get it right the first time. Few things make me feel more powerless and incompetent than a screen full of cryptic build errors and stack traces, and part of me wants to avoid it as much as I can.

The thing is, everything about computers is imperfect, from software to hardware, from design to implementation. Everything up and down the stack breaks. The ecosystem is complicated. Components interact with each other in weird ways. When something breaks, fixing it sometimes requires knowing how different components interact with each other; other times it requires superior Googling skills. The only way to learn the system is to break it and fix it. It is impossible to wrap your mind around the stack in one day: application, compiler, network, operating system, client, server, hardware, and so on. And one certainly can’t grok it by standing on the outside as an observer.

Further, many computer science programs try to teach their students computing concepts on the first go: recursion, references, data structures, semaphores, locks, and so on. These are beautiful, important concepts. But they are also very abstract and inaccessible by themselves. They also don’t instruct students on how to succeed in real software engineering projects. In the courses I took, program-
ming projects constituted a large part, but they were included as a way of illustrating abstract concepts. You still needed to parse through the concepts to pass the course. In my view, the ordering should be reversed, especially for beginners. Hands-on practice with programming projects should be the primary mode of teaching; concepts and theory should play a secondary, supporting role. It should be made clear to students that mastering all the concepts is not a prerequisite for writing a kick-ass program.

In some ways, all of us in this field are impostors. No one knows everything. The only way to progress is to dive in and start doing. Let us not measure ourselves against others, or focus on how much we don’t yet know. Let us measure ourselves by how much we’ve learned since last week, and how far we’ve come. Let us learn through playing and failing. The impostor syndrome can be a great teacher. It teaches us to love our failures and keep going.

*O’Reilly’s 2015 Edition of Women in Data reveals inspiring success stories from four women working in data across the European Union, and features interviews with 19 women who are central to data businesses.*

**The Key to Agile Data Science: Experimentation**

*by Jerry Overton*

You can read this post on oreilly.com here.

I lead a research team of data scientists responsible for discovering insights that generate market and competitive intelligence for our company, Computer Sciences Corporation (CSC). We are a busy group. We get questions from all different areas of the company and it’s important to be agile.

The nature of data science is experimental. You don’t know the answer to the question asked of you—or even if an answer exists. You don’t know how long it will take to produce a result or how much data you need. The easiest approach is to just come up with an idea and work on it until you have something. But for those of us with deadlines and expectations, that approach doesn’t fly. Companies that issue you regular paychecks usually want insight into your progress.
This is where being agile matters. An agile data scientist works in small iterations, pivots based on results, and learns along the way. Being agile doesn’t guarantee that an idea will succeed, but it does decrease the amount of time it takes to spot a dead end. Agile data science lets you deliver results on a regular basis and it keeps stakeholders engaged.

The key to agile data science is delivering data products in defined time boxes—say, two- to three-week sprints. Short delivery cycles force us to be creative and break our research into small chunks that can be tested using minimum viable experiments. We deliver something tangible after almost every sprint for our stakeholders to review and give us feedback. Our stakeholders get better visibility into our work, and we learn early on if we are on track.

This approach might sound obvious, but it isn’t always natural for the team. We have to get used to working on just enough to meet stakeholders’ needs and resist the urge to make solutions perfect before moving on. After we make something work in one sprint, we make it better in the next only if we can find a really good reason to do so.

**An Example Using the Stack Overflow Data Explorer**

Being an agile data scientist sounds good, but it’s not always obvious how to put the theory into everyday practice. In business, we are used to thinking about things in terms of tasks, but the agile data scientist has to be able to convert a task-oriented approach into an experiment-oriented approach. Here’s a recent example from my personal experience.

Our CTO is responsible for making sure the company has the next-generation skills we need to stay competitive—that takes data. We have to know what skills are hot and how difficult they are to attract and retain. Our team was given the task of categorizing key skills by how important they are, and by how rare they are (see Figure 1-1).
We already developed the ability to categorize key skills as important or not. By mining years of CIO survey results, social media sites, job boards, and internal HR records, we could produce a list of the skills most needed to support any of CSC’s IT priorities. For example, the following is a list of programming language skills with the highest utility across all areas of the company:

<table>
<thead>
<tr>
<th>Programming language</th>
<th>Importance (0–1 scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>1</td>
</tr>
<tr>
<td>SQL</td>
<td>0.4</td>
</tr>
<tr>
<td>Python</td>
<td>0.3</td>
</tr>
<tr>
<td>C#</td>
<td>0.2</td>
</tr>
<tr>
<td>C++</td>
<td>0.1</td>
</tr>
<tr>
<td>Perl</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Note that this is a composite score for all the different technology domains we considered. The importance of Python, for example, varies a lot depending on whether or not you are hiring for a data scientist or a mainframe specialist.

For our top skills, we had the “importance” dimension, but we still needed the “abundance” dimension. We considered purchasing IT survey data that could tell us how many IT professionals had a
particular skill, but we couldn’t find a source with enough breadth and detail. We considered conducting a survey of our own, but that would be expensive and time consuming. Instead, we decided to take a step back and perform an agile experiment.

Our goal was to find the relative number of technical professionals with a certain skill. Perhaps we could estimate that number based on activity within a technical community. It seemed reasonable to assume that the more people who have a skill, the more you will see helpful posts in communities like Stack Overflow. For example, if there are twice as many Java programmers as Python programmers, you should see about twice as many helpful Java programmer posts as Python programmer posts. Which led us to a hypothesis:

*You can predict the relative number of technical professionals with a certain IT skill based on the relative number of helpful contributors in a technical community.*

We looked for the fastest, cheapest way to test the hypothesis. We took a handful of important programming skills and counted the number of unique contributors with posts rated above a certain threshold. We ran this query in the Stack Overflow Data Explorer:

```sql
SELECT
  Count(DISTINCT Users.Id),
  Tags.TagName as Tag_Name
FROM
  Users, Posts, PostTags, Tags
WHERE
  Posts.OwnerUserId = Users.Id AND
  PostTags.PostId = Posts.Id AND
  Tags.Id = PostTags.TagId AND
  Posts.Score > 15 AND
  Posts.CreationDate BETWEEN '1/1/2012' AND '1/1/2015' AND
  Tags.TagName IN ('python', 'r', 'java', 'perl', 'sql', 'c#', 'c++')
GROUP BY
  Tags.TagName
```
Which gave us these results:

<table>
<thead>
<tr>
<th>Programming language</th>
<th>Unique contributors</th>
<th>Scaled value (0–1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>2,276</td>
<td>1.00</td>
</tr>
<tr>
<td>C#</td>
<td>1,868</td>
<td>0.82</td>
</tr>
<tr>
<td>C++</td>
<td>1,529</td>
<td>0.67</td>
</tr>
<tr>
<td>Python</td>
<td>1,380</td>
<td>0.61</td>
</tr>
<tr>
<td>SQL</td>
<td>314</td>
<td>0.14</td>
</tr>
<tr>
<td>Perl</td>
<td>70</td>
<td>0.03</td>
</tr>
</tbody>
</table>

We converted the scores according to a linear scale with the top score mapped to 1 and the lowest score being 0. Considering a skill to be “plentiful” is a relative thing. We decided to use the skill with the highest population score as the standard. At first glance, these results seemed to match our intuition, but we needed a simple, objective way of cross-validating the results. We considered looking for a targeted IT professional survey, but decided to perform a simple LinkedIn people search instead. We went into LinkedIn, typed a programming language into the search box, and recorded the number of people with that skill:

<table>
<thead>
<tr>
<th>Programming language</th>
<th>LinkedIn population (M)</th>
<th>Scaled value (0–1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>5.2</td>
<td>1.00</td>
</tr>
<tr>
<td>C#</td>
<td>4.6</td>
<td>0.88</td>
</tr>
<tr>
<td>C++</td>
<td>3</td>
<td>0.58</td>
</tr>
<tr>
<td>Python</td>
<td>1.7</td>
<td>0.33</td>
</tr>
<tr>
<td>SQL</td>
<td>1</td>
<td>0.19</td>
</tr>
<tr>
<td>Perl</td>
<td>0.5</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Some of the experiment’s results matched the cross-validation, but some were way off. The Java and C++ population scores predicted by the experiment matched pretty closely with the validation. But the experiment predicted that SQL would be one of the rarest skills, while the LinkedIn search told us that it is the most plentiful. This discrepancy makes sense. Foundational skills, such as SQL, that have been around a while will have a lot of practitioners, but are unlikely to be a hot topic of discussion. By the way, adjusting the allowable post creation dates made little difference to the relative outcome.
We couldn’t confirm the hypothesis, but we learned something valuable. Why not just use the number of people that show up in the LinkedIn search as the measure of our population with the particular skill? We have to build the population list by hand, but that kind of grunt work is the cost of doing business in data science. Combining the results of LinkedIn searches with our previous analysis of skills importance, we can categorize programming language skills for the company, as shown in Figure 1-2.

Figure 1-2. Programming language skill categorization (image courtesy of Jerry Overton)

Lessons Learned from a Minimum Viable Experiment

The entire experiment, from hypothesis to conclusion, took just three hours to complete. Along the way, there were concerns about which Stack Overflow contributors to include, how to define a helpful post, and the allowable sizes of technical communities—the list of possible pitfalls went on and on. But we were able to slice through the noise and stay focused on what mattered by sticking to a basic hypothesis and a minimum viable experiment.

Using simple tests and minimum viable experiments, we learned enough to deliver real value to our stakeholders in a very short amount of time. No one is getting hired or fired based on these results, but we can now recommend to our stakeholders strategies for getting the most out of our skills. We can recommend targets for recruiting and strategies for prioritizing talent development efforts.
Best of all, I think, we can tell our stakeholders how these priorities should change depending on the technology domain.
particular the de-identifying of data and the prevention of re-identification, as well as data minimization, which are both cousin to unlinkability.

Certainly, the FTC and the automakers’ associations are to be applauded for taking privacy seriously as qualitative and quantitative changes occur in the software and hardware landscapes. Given the IoT’s global character, there is room for global thinking on these matters. The best of European and American thought can be brought into the same conversation for the betterment of all. As hardware companies become software companies, they can delve into a broader set of privacy discussions to select design strategies that reflect a range of corporate goals, customer preference, regulatory imperative, and commercial priorities.

Five Principles for Applying Data Science for Social Good

by Jake Porway

You can read this post on oreilly.com here.

“We’re making the world a better place.” That line echoes from the parody of the Disrupt conference in the opening episode of HBO’s Silicon Valley. It’s a satirical take on our sector’s occasional tendency to equate narrow tech solutions like “software-designed data centers for cloud computing” with historical improvements to the human condition.

Whether you take it as parody or not, there is a very real swell in organizations hoping to use “data for good.” Every week, a data or technology company declares that it wants to “do good” and there are countless workshops hosted by major foundations musing on what “big data can do for society.” Add to that a growing number of data-for-good programs from Data Science for Social Good’s fantastic summer program to Bayes Impact’s data science fellowships to DrivenData’s data-science-for-good competitions, and you can see how quickly this idea of “data for good” is growing.

Yes, it’s an exciting time to be exploring the ways new data sets, new techniques, and new scientists could be deployed to “make the world a better place.” We’ve already seen deep learning applied to ocean health, satellite imagery used to estimate poverty levels,
and cellphone data used to elucidate Nairobi’s hidden public transportation routes. And yet, for all this excitement about the potential of this “data for good movement,” we are still desperately far from creating lasting impact. Many efforts will not only fall short of lasting impact—they will make no change at all.

At DataKind, we’ve spent the last three years teaming data scientists with social change organizations, to bring the same algorithms that companies use to boost profits to mission-driven organizations in order to boost their impact. It has become clear that using data science in the service of humanity requires much more than free software, free labor, and good intentions.

So how can these well-intentioned efforts reach their full potential for real impact? Embracing the following five principles can drastically accelerate a world in which we truly use data to serve humanity.

“Statistics” Is So Much More Than “Percentages”

We must convey what constitutes data, what it can be used for, and why it’s valuable.

There was a packed house for the March 2015 release of the No Ceilings Full Participation Report. Hillary Clinton, Melinda Gates, and Chelsea Clinton stood on stage and lauded the report, the culmination of a year-long effort to aggregate and analyze new and existing global data, as the biggest, most comprehensive data collection effort about women and gender ever attempted. One of the most trumpeted parts of the effort was the release of the data in an open and easily accessible way.

I ran home and excitedly pulled up the data from the No Ceilings GitHub, giddy to use it for our DataKind projects. As I downloaded each file, my heart sunk. The 6 MB size of the entire global data set told me what I would find inside before I even opened the first file. Like a familiar ache, the first row of the spreadsheet said it all: “USA, 2009, 84.4%.”

What I’d encountered was a common situation when it comes to data in the social sector: the prevalence of inert, aggregate data. Huge tomes of indicators, averages, and percentages fill the landscape of international development data. These data sets are sometimes cutely referred to as “massive passive” data, because they are
large, backward-looking, exceedingly coarse, and nearly impossible to make decisions from, much less actually perform any real statistical analysis upon.

The promise of a data-driven society lies in the sudden availability of more real-time, granular data, accessible as a resource for looking forward, not just a fossil record to look back upon. Mobile phone data, satellite data, even simple social media data or digitized documents can yield mountains of rich, insightful data from which we can build statistical models, create smarter systems, and adjust course to provide the most successful social interventions.

To affect social change, we must spread the idea beyond technologists that data is more than “spreadsheets” or “indicators.” We must consider any digital information, of any kind, as a potential data source that could yield new information.

**Finding Problems Can Be Harder Than Finding Solutions**

We must scale the process of problem discovery through deeper collaboration between the problem holders, the data holders, and the skills holders.

In the immortal words of Henry Ford, “If I’d asked people what they wanted, they would have said a faster horse.” Right now, the field of data science is in a similar position. Framing data solutions for organizations that don’t realize how much is now possible can be a frustrating search for faster horses. If data cleaning is 80% of the hard work in data science, then problem discovery makes up nearly the remaining 20% when doing data science for good.

The plague here is one of education. Without a clear understanding that it is even possible to predict something from data, how can we expect someone to be able to articulate that need? Moreover, knowing what to optimize for is a crucial first step before even addressing how prediction could help you optimize it. This means that the organizations that can most easily take advantage of the data science fellowship programs and project-based work are those that are already fairly data savvy—they already understand what is possible, but may not have the skill set or resources to do the work on their own. As Nancy Lublin, founder of the very data savvy DoSomething.org and Crisis Text Line, put it so well at Data on Purpose—“data science is not overhead.”
But there are many organizations doing tremendous work that still think of data science as overhead or don’t think of it at all, yet their expertise is critical to moving the entire field forward. As data scientists, we need to find ways of illustrating the power and potential of data science to address social sector issues, so that organizations and their funders see this untapped powerful resource for what it is. Similarly, social actors need to find ways to expose themselves to this new technology so that they can become familiar with it.

We also need to create more opportunities for good old-fashioned conversation between issue area and data experts. It’s in the very human process of rubbing elbows and getting to know one another that our individual expertise and skills can collide, uncovering the data challenges with the potential to create real impact in the world.

**Communication Is More Important Than Technology**

*We must foster environments in which people can speak openly, honestly, and without judgment. We must be constantly curious about one another.*

At the conclusion of one of our recent DataKind events, one of our partner nonprofit organizations lined up to hear the results from their volunteer team of data scientists. Everyone was all smiles—the nonprofit leaders had loved the project experience, the data scientists were excited with their results. The presentations began. “We used Amazon RedShift to store the data, which allowed us to quickly build a multinomial regression. The p-value of 0.002 shows ...” Eyes glazed over. The nonprofit leaders furrowed their brows in telegraphed concentration. *The jargon was standing in the way of understanding the true utility of the project’s findings.* It was clear that, like so many other well-intentioned efforts, the project was at risk of gathering dust on a shelf if the team of volunteers couldn’t help the organization understand what they had learned and how it could be integrated into the organization’s ongoing work.

In many of our projects, we’ve seen telltale signs that people are talking past one another. Social change representatives may be afraid to speak up if they don’t understand something, either because they feel intimidated by the volunteers or because they don’t feel comfortable asking for things of volunteers who are so generously donating their time. Similarly, we often find volunteers who are excited to try out the most cutting-edge algorithms they can on
these new data sets, either because they’ve fallen in love with a certain model of Recurrent Neural Nets or because they want a data set to learn them with. This excitement can cloud their efforts and get lost in translation. It may be that a simple bar chart is all that is needed to spur action.

Lastly, some volunteers assume nonprofits have the resources to operate like the for-profit sector. Nonprofits are, more often than not, resource-constrained, understaffed, under appreciated, and trying to tackle the world’s problems on a shoestring budget. Moreover, “free” technology and “pro bono” services often require an immense time investment on the nonprofit professionals’ part to manage and be responsive to these projects. They may not have a monetary cost, but they are hardly free.

Socially minded data science competitions and fellowship models will continue to thrive, but we must build empathy—strong communication through which diverse parties gain a greater understanding of and respect for each other—into those frameworks. Otherwise we’ll forever be “hacking” social change problems, creating tools that are “fun,” but not “functional.”

**We Need Diverse Viewpoints**

*To tackle sector-wide challenges, we need a range of voices involved.*

One of the most challenging aspects to making change at the sector level is the range of diverse viewpoints necessary to understand a problem in its entirety. In the business world, profit, revenue, or output can be valid metrics of success. Rarely, if ever, are metrics for social change so cleanly defined.

Moreover, any substantial social, political, or environmental problem quickly expands beyond its bounds. Take, for example, a seemingly innocuous challenge like “providing healthier school lunches.” What initially appears to be a straightforward opportunity to improve the nutritional offerings available to schools quickly involves the complex educational budgeting system, which in turn is determined through even more politically fraught processes. As with most major humanitarian challenges, the central issue is like a string in a hairball wound around a nest of other related problems, and no single strand can be removed without tightening the whole mess. Oh, and halfway through you find out that the strings are actually snakes.
Challenging this paradigm requires diverse, or “collective impact,” approaches to problem solving. The idea has been around for a while (h/t Chris Diehl), but has not yet been widely implemented due to the challenges in successful collective impact. Moreover, while there are many diverse collectives committed to social change, few have the voice of expert data scientists involved. DataKind is piloting a collective impact model called DataKind Labs, that seeks to bring together diverse problem holders, data holders, and data science experts to co-create solutions that can be applied across an entire sector-wide challenge. We just launched our first project with Microsoft to increase traffic safety and are hopeful that this effort will demonstrate how vital a role data science can play in a collective impact approach.

**We Must Design for People**

Data is not truth, and tech is not an answer in and of itself. Without designing for the humans on the other end, our work is in vain.

So many of the data projects making headlines—a new app for finding public services, a new probabilistic model for predicting weather patterns for subsistence farmers, a visualization of government spending—are great and interesting accomplishments, but don’t seem to have an end user in mind. The current approach appears to be “get the tech geeks to hack on this problem, and we’ll have cool new solutions!” I’ve opined that, though there are many benefits to hackathons, you can’t just hack your way to social change.

A big part of that argument centers on the fact that the “data for good” solutions we build must be co-created with the people at the other end. We need to embrace human-centered design, to begin with the questions, not the data. We have to build with the end in mind. When we tap into the social issue expertise that already exists in many mission-driven organizations, there is a powerful opportunity to create solutions to make real change. However, we must make sure those solutions are sustainable given resource and data literacy constraints that social sector organizations face.

That means that we must design with people in mind, accounting for their habits, their data literacy level, and, most importantly, for what drives them. At DataKind, we start with the questions before we ever touch the data and strive to use human-centered design to create solutions that we feel confident our partners are going to use.
before we even begin. In addition, we build all of our projects off of deep collaboration that takes the organization's needs into account, first and foremost.

These problems are daunting, but not insurmountable. Data science is new, exciting, and largely misunderstood, but we have an opportunity to align our efforts and proceed forward together. If we incorporate these five principles into our efforts, I believe data science will truly play a key role in making the world a better place for all of humanity.

**What's Next**

Almost three years ago, DataKind launched on the stage of Strata + Hadoop World NYC as Data Without Borders. True to its motto to “work on stuff that matters,” O'Reilly has not only been a huge supporter of our work, but arguably one of the main reasons that our organization can carry on its mission today.

That’s why we could think of no place more fitting to make our announcement that DataKind and O'Reilly are formally partnering to expand the ways we use data science in the service of humanity. Under this media partnership, we will be regularly contributing our findings to O'Reilly, bringing new and inspirational examples of data science across the social sector to our community, and giving you new opportunities to get involved with the cause, from volunteering on world-changing projects to simply lending your voice. We couldn’t be more excited to be sharing this partnership with an organization that so closely embodies our values of community, social change, and ethical uses of technology.

We’ll see you on the front lines!
INTRODUCTION

Big data is a reality we are experiencing and facing every day. It is not a buzz word anymore; it is a fact and is here to stay. As opposed to decades ago in which finding the data was the major obstacle, handling and processing the available data is now the main bottleneck in most research fields. Big data is not only involved in science and engineering fields but also in digital humanities. Classifying all available English novels, topic modelling all the news articles, studying the personality types of the characters in all novels, comparing novels according to their plot spatial distributions, etc. are all examples of big data and big data analytics in digital humanities. Imagine building a novel recommendation system for users based on the existing reviews, and/or on the user’s preferences and old history. Imagine doing so based also on his email, twitter, and Facebook messages as well as his blogs.

Most of us are bombarded with data every second, from different sources and sensors, and of different types: news, articles, twitters, youtube videos, images, ads, etc. Some of us may have large data sets and wish to just serve it to others in real or semi-real time. Others may have large data sets but want to analyze it and serve the results to their users within a decent time frame. Others may not experience that much data but, using the traditional tools, it is challenging for them to extract knowledge and value from the modest data they have. In all these cases, big data is here to help.

The aim of this course is to provide the digital humanists with the necessary big data tools to embark on their big data analytics journey with ease and confidence. In addition to a thorough study of a special use case, the attendees will go through hands-on exercises and, along the way, learn how to setup their own big data platform and ecosystem.

BIG DATA REALM

In their paper “The unreasonable effectiveness of data”, Alon et al. argued that, as opposed to the natural sciences where the elegant mathematical formula can explain the physical phenomena neatly (as Wigner’s showed in his seminal article “The unreasonable effectiveness of mathematics in the natural sciences”), sciences that involve humans are rather difficult to understand using an elegant mathematics. In fact, the English grammar spans over 1700 pages and fails any attempt to frame it in a single neat mathematical theory. So, instead of trying to frame those complex problems using complex mathematical
theories, we should instead embrace data and using it to extract knowledge. Such approach proved that data-driven models can outperform the most advanced theoretical/statistical model for natural languages. Moreover, this approach is invading even natural sciences and is also common in most physics fields these days including particle physics (CERN/ATLAS experiments is an excellent example).

“So, follow the data. Choose a representation that can use unsupervised learning on unlabeled data, which is so much more plentiful than labeled data. Represent all the data with a nonparametric model rather than trying to summarize it with a parametric model, because with very large data sources, the data holds a lot of detail.” (from the “The unreasonable effectiveness of data” paper).

“Big data” term can lead to big confusion. In fact, it is very hard to find a universal definition of the “big data” term. However, we will adopt what we think is the most adequate definition:

“Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value” (copied from “What is big data? A consensual definition and a review of key research topics” paper).

In our terms, if you are dealing with data that cannot be handled properly and efficiently using the current existing technologies and analytical methods, then you are automatically in the big data realm.

Big data does not necessarily mean big in size. The criteria for characterizing big data are usually best described by the five or even seven Vs. Keep in mind that although we did not include anything related to security and privacy, big data realm embraces those also.

**VOLUME**

The scale at which data is produced is mind blowing and is doubling every two years. It is estimated that in 2020, the data created and copied will reach 44 ZB. We are not any more in the TB or PB scales. Here is some information about amounts of data involved from aci.info:

In every minute,
- Facebook users share nearly 2.5 million pieces of content.
- Twitter users tweet nearly 300,000 times.
- Instagram users post nearly 220,000 new photos.
- YouTube users upload 72 hours of new video content.
- Apple users download nearly 50,000 apps.
- Email users send over 200 million messages.
- Amazon generates over $80,000 in online sales.
The 2014 infographic from DOMO below summaries is it all in Figure 1. Keep in mind that these statistics does not include the petabytes of data that researchers and small companies are gathering every minute. Even in natural sciences in which the elegance of mathematics can compress their data, we have seen simulations that produced more than 5TB in a single snapshot in time.

![Image of infographic]

FIGURE 1. 2014 DATA NEVER SLEEPS 2.0 INFOGRAPHIC.

**VARIETY**

As the infographic above shows, not only that the size of the data is becoming huge but also its types are increasing in number and complexity. In the past, we relied on the structured and relational database types to store, manage and mine our data. Most of the forms we used to fill up are stored in a relational database and then used to do any analysis on the
data. Unfortunately this paradigm of the data type is too simple to accommodate all kind of
data. In fact, around 80% of the current data is unstructured and does not adhere to any
schema. Videos, photos, images, audio, text messages, blogs, emails, chats, etc. moved the
complexity of the data to a higher scale. How to store, manage and mine such data? How to
extract value out of these unstructured data? All data formats, xml, json, pdf, xsl, etc. are
part of the big data variety that we, as big data analytics, need to embrace and deal with.

VELOCITY

The speed at which data is produced and moved is mind blowing. The number of clicks per
second and the number of messages, videos and images exchanged every minute on the net
are huge (see again the infographic for actual numbers). The data from different satellites,
weather stations, cameras, sensors, TV shows, etc. are continuously flowing. Data velocity
is a major challenge as the analytics need to take that into consideration and be able to
adapt to the streaming nature of the data.

VERACITY

To add to the complexity of big data, the quality, the fuzziness and the trustworthiness of
data are becoming import with growing amount of data. How to deal with data if the source
is not known or reliable? How process an OCR-based text with low quality? The uncertainty
of data should be taken into consideration when designing big data analytics tools and
algorithms.

To the above four Vs, we can without doubt add variability and visualization. Variability
captures the non-constant meaning of the data. This indeed is the case for natural text
processing where the meaning of word depends on the context in which the word occurs.
Visualization comes in handy to explore and present the processed data. It can even be a
must to identify patterns that machines and learning algorithms cannot. In fact, under big
data realm, visualization analytics is now a flourished branch in the visualization field.

All the Vs listed so far are important but their importance is conditioned upon the next V.

VALUE

Huge amount of data are not useful if they do not have any value (Imagine trying to extract
value from TBs or even PBs of randomly generated numbers). Turning data into value
should be the driving force of any big data endeavour. This is equally important for
businesses and research. Before embarking on collecting and leveraging big data, the
stakeholders need to make sure that they have a targeted value in mind. Serving the
customers better, helping the patients to make the right choice, reducing the costs by
optimizing the processes, boosting performances, helping readers to find their future best
novels, are all example of big data values. It is worth emphasizing that although the data
may be worthless or even harmful in its own, the value, the information and the insight that one can produce from leveraging and analyzing it can be priceless.

**COMPUTE CANADA AND BIG DATA PLATFORM ARCHITECTURE**

Compute Canada is a national platform for Advance Research Computing. It is composed of 4 main regions covering the whole Canada and each representing one or more consortia. Compute Canada does not only offer computational resources but also experts in advanced research computing who supports all researchers in Canada as well as their collaborators. Compute Canada has many clusters of different nature and distributed all over its consortia. Currently we have more than 120K cores on our HPC clusters. In addition we have modest GPU and PHI clusters as well as storage facilities, and visualization and collaboration rooms. In addition to serial jobs, most of our clusters support parallel jobs. At least two of our clusters support dynamic Hadoop that runs on our HPC infrastructure. It is dynamic in the sense that the hadoop ecosystem is controlled by the user and is launched when the job starts and terminates when it ends.

We also offer two openstack clouds: one in the east (University of Victoria) and the other in the west (University of Sherbroke). The two clouds are available to any Compute Canada user. We will be using the cloud as well as the dynamic hadoop for our big data infrastructure and platform. A dedicated image is created for this course so that you can instantiate it on our openstack cloud but feel free to download it and use it on other clouds. For the dynamic hadoop, 6 nodes are reserved for the course so that we can together go through the examples and the use case study. The 6 nodes have 12 cores and 24GB of memory each and are located in the hermes cluster hosted at the University of Victoria. Thanks to one of our researchers, these 6 nodes have 3x2 TB disk each. The nodes are connected via two networks: the management network (1 GigE) and the QDR infiniband network.

The big data platform has the following hadoop ecosystem: Zeppelin (0.6.0), Jupyter (4.6.0), Phoenix (4.3.0), Spark (1.5.2), Drill (1.3.0), HBase (0.98.16.1), zookeeper (3.4.6), and Hadoop (2.6.2). The hadoop ecosystem stack is shown below.

![FIGURE 2. BIG DATA SOFTWARE STACK.](image)
HADOOP (HDFS, YARN AND MAPREDUCE)

At the bottom of the stack lies HDFS. It is the distributed file system of Hadoop and is known to scale and perform well in the data space. Yarn is the resource manager of Hadoop and serves in scheduling and running the Hadoop jobs. Mapreduce is the hadoop implementation of the distributed map reduce programming paradigm.

SPARK

Currently, Spark is at the heart of the big data platform. It is a distributed and parallel processing engine engineered for big data and built for speed, simplicity and intensive analytics. It covers a decent data space as it can read data from HDFS, HBase, Hive, etc. Its speed stems from its judicious use of memory to cache the data. Spark is best viewed as a processing pipeline combining two types of operators: transformations and actions. These operators are applied to an immutable, resilient distributed dataset structure known as RDD for short. Each transformation produces an RDD that be cached in memory and/or persisted to disk depending on user’s choice. In Spark, a transformation is a lazy operator as it is not executed when it is called. Instead a direct acyclic graph (DAG) of the RDD transformations is built, optimized and only executed when an action is applied. Spark relies on a master to drive the execution of the DAG on set of executors (running on the Hadoop data nodes).

On top of the Spark core we found a rich set tool space: Spark SQL and dataframes, Machine learning library (MLLib), Graph library (GraphX) and many third-party packages. Spark also offers support to three languages: Scala, Python and R. These can be run interactively or via batch jobs. Since we are interested in user interactivity, we elaborate on the interactivity interfaces further.

SPARK TERMINAL

Users can interact with spark via it is terminal. Similar to the usual command line terminals, everything is displayed as text For scala, the user should call spark-shell to get access the scala terminal. For python and R, the user should run pyspark and sparkR, respectively. In this course, we recommend this mode for advanced users only.

JUPYTER

As a successor of ipython, Jupyter is a successful interactive and development tool for data science and scientific computing. It accommodates over 50 programming languages and offers, via its notebook web application, the necessary infrastructure to create and share documents as well as collaborate, report and publish them. A notebook is a rich document that can contain code, equations, text (especially markup text), and visualizations. This mixture of code, equations, text, and visualization allows the users to share their work products in a single and self-contained document.

ZEPPELIN
Another nice interface to interact with Spark is Zeppelin. Zeppelin is a web-based notebook that’s not limited to Spark. It supports a large number of back-ends through its interpreter mechanism. In zeppelin, any other back-end can be supported once the proper interpreter is implemented. SQL queries are supported via the SparkSQL interpreter and can be executed by preceding the query with %sql magic. What makes Zeppelin interesting is the fact that it has built-in visualizations and user can, after running an SQL query, just click on the chart to generate. This kind of at-the-finger-tip visualizations is essential for the zero-day analytics that digital humanists are interested in.

**DRILL**

Inspired by Google’s big query engine Dremel, Drill offers a distributed execution environment for large-scale SQL queries. It supports a wide range of data sources, including CSV, JSON, HBase, etc. By (re)compiling and optimizing the query and interacting with the distributed data sets via the so-called drillbit service, Drill offers a low latency SQL query engine. As opposed to the master/slave architecture of Spark in which a driver is handling the execution of the DAG on a given set of executors, the drillbits are loosely coupled and each can accept a query from the client. The receiving drillbit becomes the driver for the query, parses it, optimizes it, generates an efficient, distributed and multiphase execution plan for it, schedules the execution of each phase on the proper drillbit, and gathers the results back when the scheduled execution is done.

To run Drill in a distributed mode, one has to have a zookeeper cluster up and running. Drill 1.3.0 and Zookeeper 3.4.6 were installed and configured on our platform. In addition, in-house scripts are provided to allow dynamic hadoop ecosystem to run on top of the existing HPC clusters via module environments package.

**HANDS-ON EXERCISES SUMMARY**

Big data realm is less than a decade old but it already involves a lot of tools and packages. Hadoop ecosystem itself has more than 10 large software packages: Hadoop, Spark, Drill, HBase, Phoenix, Hive, PIG, Storm, Kafka, Hue, Zeppelin, etc. To do justice to all these packages is out of the scope of this course. In addition, the number of DH applications and examples in which these packages can be used is so huge that we limit our hands-on to specific but rewarding examples.

To get started, the attendees are expected to familiarize themselves with Hadoop, especially its HDFS and Yarn component, and Spark, as well as HBase, Drill, Jupyter and Zeppelin. As we advance through the course, they are also expected to write and solve their big data analytics problems using Spark, visualize and share their Jupyter and Zeppelin notebooks. The following tasks are just few examples of the hands-on the users will go through:

- Find out the version of Hadoop used, the live data nodes and their available spaces
• Create folders under HDFS and get used to HDFS commands
• Run HBase-shell and create your first table
• Run pyspark under Jupyter (or Zeppelin) and print the version of spark used
• Warm up
  o Get the spark context based on findSpark
  o Sum the numbers from 1,1000.
  o Compute the sum of their squares
  o Get 200 samples and compute their mean and std
  o Sum the odd numbers of the sampled RDD
  o Sum numbers: 0; 0,1; 0,1,2; 0,1,2,3; ...
  o Estimate the value of pi using a Monte Carlo approach
• Document processing use case
  o Get your favorite text file
  o Ingest it into HDFS
  o Count the number of lines
  o Count the number of words
  o Print the word counts; sort them and display the first 10
  o Print only the lines containing your favorite word
  o Plot the word counts for the first 10 words
  o Use your favorite stop word list to only count important words
  o Plot the word counts of the first 10 words
• Dive in Further
  o Get your favorite csv files
  o Ingest them to hdfs
  o Bulk load them to HBase
    ▪ You may want to use Phoenix
  o Query them via HBase-shell and spark
    ▪ You may want to use phoenix sqlline instead
  o Optional try with phoenix and Drill
  o Plot few charts using Jupyter (with python/pyspark) or Zeppelin
• HTML and big data realm
  o Get your favorite html novel and ingest it to hdfs
  o Run few spark transformations and actions on them:
    ▪ Count the number of words, plot their distributions.
    ▪ Count the number of paragraphs (text between <p>), plot some histograms,
    ▪ Do some frequency analysis on some specific words.

USE CASE STUDY SUMMARY

Similar what we mentioned in the previous section, big data DH use case studies are enormous but we restrict ourselves to the exploring, classifying and extracting knowledge from the Adelaide HTML ebooks. The same analytics can be carried out for any other
corpus (Only the URLs and the parsing of HTML docs may differ). The following are the activities involved in this use case study:

- Get the full Adelaide works in parallel using pyspark
- Use beautiful soup and urllib2 for parsing and getting the works by alphabet
- Repartition the links of works between the executors
- Get the html (optional the zip) for each work
- Plot some charts such as:
  - Number of works per author
  - Number of works per year when available
  - Number of novels per paragraph bins
  - Number of images per novel
- **Frequency Analysis**
  - Generate the word count for each novel
  - Compute the correlation between novels based on that
  - Compute the distribution of a given word in each paragraph; plot the distribution
- **Topic Modelling**
  - Choose a stop-word list
  - Topic model the novels based on the Spark MLLib
- **Clustering**
  - Cluster the novels based on that Spark MLLib
  - Build a recommendation system based on the clustering
    - Based on the first 5 words of each topic, propose a novel depending on the user choice of words.
    - Based on the first 5 words of each topic, propose a novel depending on the user choice of words.
  - Build a recommendation system based on the rating instead
    - Use ALS from Spark’s MLLib to do so.
- Optional/advanced activities
  - Images extracted from the Adelaide works and classify/cluster them.
  - Fill up the missing metadata and file it for Adelaide.

The number of books will be treating is limited by the number of Adelaide HTML ebooks available. As such the data size is will range from 40 to 1500 HTML ebooks.

**BIG DATA WORKFLOW**

Big data workflow is best viewed as a feedback iterative process involving many steps. Each step may be an iterative process in itself.
Before embarking on a big data initiative make sure that you have a well-defined and understood value in mind. The data out there is competing with the number of stars, so be careful to dive into the sea space and drawn yourself without getting any value. You may risk losing your precious time and resources before getting any reward from your big data analytics. A risk plan should also help to get there sound and safe. Once the value is understood and the risks are quantified, you can start your big data initiative with a value plan with a clear hypothesis. This hypothesis is important to drive the rest of the big data process, especially the data to deal with and the kind of analytics to look for.

**ACQUISITION, INGESTION AND STREAMING**

As surprising as it may seem, one of the challenging tasks in big data realm is what data to gather and how to get it from the existing environments into the big data ecosystem, such as Hadoop. Finding the data sources can be very time consuming, especially if it is copyrighted or requires special preprocessing if it is in the wild. The way the data is produced from the sources influences the way it should be handled.

Once the data sources and datasets are decided upon, they should be gathered and collected as necessary. The tools for doing so depend on the flow of the data from the source and the analytics to perform on it. To gather and collect data, simple script containing the usual tools such as gridftp, scp, sftp, wget, and curl are usually efficient and sufficient for most use cases; the big data cases for which the velocity is not very high and
the data content does not change that much. For data with high velocity and dynamic content, advanced bigdata tools such Kafka, Sqoop, Flume, Storm, etc. are necessary to aggregate, ingest, and stream data into the bigdata ecosystem.

HANDS-ON

In this hands-on part, the users will play with few tools including wget, curl, and python urllib2 and beautiful soup 4 (bs4) modules in addition to HDFS, HBase and Phoenix. The users will start with launching a Jupyter notebook and using it to run wget and curl to get their data of interest, then ingest it to HDFS and getting the first 10 lines of it. Then they are required to use urllib2 and bs4 to get an html page, preprocess it and ingest it into HDFS. They are then asked to either generate a CSV file from the preprocessed text or download existing CSV files and ingest them into HBase. They are then asked to query the HBase data using simple get statement. After that they should the repeat the same exercise but using Phoenix in addition to HBase.

ANALYTICS AND DATA EXPLORATION

Depending on the value intended from the big data project at hand, the user may first want to explore data by first sampling the data to reduce its size, and then filtering it from the noise. This kind of preprocessing and rough exploration may be required before hand to convince the stakeholders of the value of the big data project at hand. Under Spark, filtering and sampling are both transformation and the user should use them and avoid re-implementing them. The main analytics pipeline is then applied on the sampled data to get the intended outcome. The analytics algorithms to run will need to be implemented based on the programming paradigm with which the user is intended to work. If the paradigm is the Hadoop MapReduce, for example, the user will need to rewrite his algorithm to match that paradigm using, mainly, Java. If on the other hand, the user is interested in running Spark's transformations and actions, then she should do so based on Spark's paradigm. Once the implementation is done (which can be a single Scala or pyspark line in Spark), he will need to test, verify, debug and validate the algorithm on the sampled data. Fake/synthetic data can also be necessary for these tasks as the user may not know what to expect from the actual data.

HANDS-ON

To familiarize ourselves with the data at hand, the users are asked to first explore data. Using the HBase/Phoenix tables we prepared before, run few SQL queries. For example, run

```
select count(*) from TABLE;
```

to see the number of lines in your Phoenix table. Also run

```
select * from TABLE limit 20;
```
To see the content of the table limited to 20 records.

If Phoenix was not used, you can run something like

```
scan 'TABLE'
```

...to scan the whole HBase table. If the table is very huge, then you should first get some information about the table by running describe, list, etc. to view the tables and their schemas. Then run get to query a specify row key in your HBase table.

Querying the data, especially under HBase, is usually an exploration step and not enough to for big data analytics (although you can ran sophisticated queries using joins, avg, sum, count, etc. under Phoenix and Drill). For that, you will need advanced computing paradigms such as MapReduce and Spark transformations and actions. Since Spark's paradigm includes both and much easier to use and run, we will focus on Spark and leave the Hadoop's Java mapreduce aside.

The users are asked to play with Spark:

1. Get used to different ways to talking to Spark: scala, python, R, etc. Choose your favorite.
2. Print the version of spark you are using
3. Play with Spark's parallelize function and get used to RDDs.
4. Play with map and reduce on RDD and get used to Mapreduce transformation/action.
5. Play with advance transformations such as filters.
6. Used the above to count lines, words, images, etc., in document(s)
7. Run few stats and plot few charts using matplotlib, Jupyter based on the output of your Spark transformations.
8. Based on the exploration above, decide on a research question that maximizes the value and minimizes the risk.
9. Address the research question by running suitable transformations and algorithms from Spark/MLLib/SparkSQL/GraphX.

---

**VERIFICATION AND VALIDATION**

**PRESENTATION, VISUALIZATION AND SHARING**
HADOOP

A distributed file system – HDFS.

The main HDFS components:
- Name node, secondary name node
- Data nodes

The main HDFS options:
- Replication
- Socket transfer
- Compression

The main command line:
- `hdfs dfsadmin ...
- `hdfs dfs ...

A resource manager – Yarn

Schedule the jobs

Manage the resources

Main components
- Master
- Nodes
- Application Master
Old paradigm that steam for two main function:

Map: takes a collection and produces another by applying a function on each element.
Example: map each element in a given list to its square.
Reduce: takes a list of collections, reduces them according to an operation.
Example: sum the element of a given list.

Hadoop mapreduce paradigm –
A Mapreduce by key framework
Distributed mapreduce.

Map:
Takes (key,value) collection and maps them to new (key, value)-based collection
Reduce:
Takes the pairs with the same key and reduces them according to a given operation.

SPARK

Spark Core
RDDs
A pipeline of transformations and actions
DAGs
Spark Modules
Spark SQL
MLLIB
GraphX
Streaming
Spark Transformations and Actions Paradigm
Spark SQL
Spark Graph
Spark MLLib
Spark Streaming
HBASE – SCHEMALESS DB

Column-based DB
Semi-realtime access
No schema required
Resides on top of hadoop
Main components:
Zookeeper
Master, Secondary Master
Region servers
Row key and Column families
Region
HStore
Memstore
HFile
Phoenix can be added to offer SQL-like behavior to HBase

HBase – Basics
Interacting with HBase via HBase-shell or sqlline if Phoenix is used
HBase shell can be used to manipulate tables and their content
sqlline can be used to run SQL commands

HBase workflow
Manipulate tables
Create a table, Drop table, Etc.
Manipulate the content of the tables
Put, get, scan, delete, etc.

HBase – Example
./bin/HBase shell

version
MACHINE LEARNING FOR BIG DATA

Classification of machine learning algorithms

Supervised Learning

Unsupervised Learning

Reinforcement Learning

Focus on two:

Clustering

Topic modeling

OVERVIEW

What's machine learning.

CLASSIFICATION OF THE MACHINE LEARNING ALGORITHMS

Supervised Learning

Unsupervised Learning

Reinforcement Learning

Semi-supervised Learning

SUPERVISED LEARNING

Data with known labels/outcomes is assumed to be available

Given model M with parameters m1,...,m4, the task is find m1,...,m4 based on the known data (training step) and gives good labeling results for the unknown data (prediction step)
Classification, regression, Neural networks, etc. belong to this category

Examples:

Predict the price of the houses depending on their location given a list of houses and their prices
Predict the user decision based on their profiles given a list of users their profiles and decisions.

UNSUPERVISED LEARNING

Data without known labels or outcomes
Develop algorithms that cluster/group the data with similar features together.
Clustering algorithms, topic modeling, etc. belong to this kind of learning.

Examples:

Given a set of images, cluster them and find similar ones
Given a set of documents, cluster them according to their topics.

REINFORCEMENT LEARNING

Learn by trial and error: interact with the environment and reward/punish depending on the actions
The goal in RL is maximize rewards/minimize loses.
V- and Q-Learning algorithms are just for that!

Examples:

 Autonomous system for driving cars/helicopters.
Adjust the recommended options depending on user’s present action.

CLUSTERING

Powerful technique for exploring and organizing data into groups
Given a set of data, the clustering task is to then group “similar” data and assign them to the same cluster
A similarity measure is required to use clustering
Many clustering algorithms exist
K-means
Affinity propagation
FIGURE 4. CLUSTERING BY GROUPING SIMILAR DATA TOGETHER.

TOPIC MODELLING

Used to discover the topics in a set of documents
Relies on statistical/probabilistic models
The task is then to estimate the model parameters given the documents
Latent Dirichlet Allocation (LDA) is one of the most used topic modeling techniques
A generative model
Each document is a distribution over topics
Each topic is a distribution over words
The task is to estimate those two distributions
FIGURE 5. DOCUMENTS AS DISTRIBUTIONS OVER TOPICS.
For each topic $T_p$

a) Choose a distribution $P_{T_p}$ over words

For each document $L_i$:

a) Choose a distribution $p_{Li}$ over $K$ topics

b) for each word in $L_i$:

i. Draw one of the $K$ topics, $T_j$ according $p_{Li}$

ii. Draw one of the $V$ words from $P_{T_j}$
FIGURE 8. THE PROBABLISTIC GRAPHICAL MODEL FOR LDA.

CONCLUSIONS
Big Data Now

Current Perspectives from O'Reilly Media
or robot applications will require the system to recognize and understand its environment from scratch, and adapt to novel challenges in real time. What about autonomous vehicles exploring new territory for the first time (think about an independent Mars rover, at one extreme), or that face rapidly-shifting or even adversarial situations in which a static map, however detailed, simply can’t capture the essential aspects of the situation? The bottom line is that there are many environments that can’t be measured or instrumented sufficiently to be rendered legible to Google-style machines.

Other candidates include the interpretation and prediction of company performance from financial and other public data (properties 1 and 2); understanding manufacturing performance and other business processes directly from sensor data, and suggesting improvements thereon (2 and 3); and mapping and optimizing the real information and decision-making flows within organizations, an area that’s seen far more promise than delivery (1, 2, and 3).

This is a long way from coherent advice, but it’s in areas like these where I see the opportunities. It’s not that the large Internet companies can’t go after these applications; it’s that these kinds of problems fit poorly with their ingrained assumptions, modes of organization, existing skill sets, and internal consensus about the right way to go about things. Maybe that’s not much daylight, but it’s all you’re going to get.

What is Deep Learning, and Why Should You Care?

by Pete Warden

When I first ran across the results in the Kaggle image-recognition competitions, I didn’t believe them. I’ve spent years working with machine vision, and the reported accuracy on tricky tasks like distinguishing dogs from cats was beyond anything I’d seen, or imagined I’d see anytime soon. To understand more, I reached out to one of the competitors, Daniel Nouri, and he demonstrated how he used the Decaf open-source project to do so well. Even better, he showed me how he was quickly able to apply it to a whole bunch of other image-recognition problems we had at Jetpac, and produce much better results than my conventional methods.
I’ve never encountered such a big improvement from a technique that was largely unheard of just a couple of years before, so I became obsessed with understanding more. To be able to use it commercially across hundreds of millions of photos, I built my own specialized library to efficiently run prediction on clusters of low-end machines and embedded devices, and I also spent months learning the dark arts of training neural networks. Now I’m keen to share some of what I’ve found, so if you’re curious about what on earth deep learning is, and how it might help you, I’ll be covering the basics in a series of blog posts here on Radar, and in a short upcoming ebook.

So, What is Deep Learning?

It’s a term that covers a particular approach to building and training neural networks. Neural networks have been around since the 1950s, and like nuclear fusion, they’ve been an incredibly promising laboratory idea whose practical deployment has been beset by constant delays. I’ll go into the details of how neural networks work a bit later, but for now you can think of them as decision-making black boxes. They take an array of numbers (that can represent pixels, audio waveforms, or words), run a series of functions on that array, and output one or more numbers as outputs. The outputs are usually a prediction of some properties you’re trying to guess from the input, for example whether or not an image is a picture of a cat.

The functions that are run inside the black box are controlled by the memory of the neural network, arrays of numbers known as weights that define how the inputs are combined and recombined to produce the results. Dealing with real-world problems like cat-detection requires very complex functions, which mean these arrays are very large, containing around 60 million numbers in the case of one of the recent computer vision networks. The biggest obstacle to using neural networks has been figuring out how to set all these massive arrays to values that will do a good job transforming the input signals into output predictions.

Training

One of the theoretical properties of neural networks that has kept researchers working on them is that they should be teachable. It’s pretty simple to show on a small scale how you can supply a series of example inputs and expected outputs, and go through a mechanical
process to take the weights from initial random values to progressively better numbers that produce more accurate predictions (I’ll give a practical demonstration of that later). The problem has always been how to do the same thing on much more complex problems like speech recognition or computer vision with far larger numbers of weights.

That was the real breakthrough in the 2012 Imagenet paper sparking the current renaissance in neural networks. Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton brought together a whole bunch of different ways of accelerating the learning process, including convolutional networks, clever use of GPUs, and some novel mathematical tricks like ReLU and dropout, and showed that in a few weeks they could train a very complex network to a level that outperformed conventional approaches to computer vision.

This isn’t an aberration, similar approaches have been used very successfully in natural language processing and speech recognition. This is the heart of deep learning—the new techniques that have been discovered that allow us to build and train neural networks to handle previously unsolved problems.

**How is it Different from Other Approaches?**

With most machine learning, the hard part is identifying the features in the raw input data, for example SIFT or SURF in images. Deep learning removes that manual step, instead relying on the training process to discover the most useful patterns across the input examples. You still have to make choices about the internal layout of the networks before you start training, but the automatic feature discovery makes life a lot easier. In other ways, too, neural networks are more general than most other machine-learning techniques. I’ve successfully used the original Imagenet network to recognize classes of objects it was never trained on, and even do other image tasks like scene-type analysis. The same underlying techniques for architecting and training networks are useful across all kinds of natural data, from audio to seismic sensors or natural language. No other approach is nearly as flexible.

**Why Should You Dig In Deeper?**

The bottom line is that deep learning works really well, and if you ever deal with messy data from the real world, it’s going to be an essential element in your toolbox over the next few years. Until
recently, it’s been an obscure and daunting area to learn about, but its success has brought a lot of great resources and projects that make it easier than ever to get started. I’m looking forward to taking you through some of those, delving deeper into the inner workings of the networks, and generally have some fun exploring what we can all do with this new technology!

**Artificial Intelligence: Summoning the Demon**

*We need to understand our own intelligence is competition for our artificial, not-quite intelligences*

by Mike Loukides

In October, Elon Musk likened artificial intelligence (AI) to “summoning the demon”. As I’m sure you know, there are many stories in which someone summons a demon. As Musk said, they rarely turn out well.

There’s no question that Musk is an astute student of technology. But his reaction is misplaced. There are certainly reasons for concern, but they’re not Musk’s.

The problem with AI right now is that its achievements are greatly over-hyped. That’s not to say those achievements aren’t real, but they don’t mean what people think they mean. Researchers in deep learning are happy if they can recognize human faces with 80% accuracy. (I’m skeptical about claims that deep learning systems can reach 97.5% accuracy; I suspect that the problem has been constrained some way that makes it much easier. For example, asking “is there a face in this picture?” or “where is the face in this picture?” is much different from asking “what is in this picture?”) That’s a hard problem, a really hard problem. But humans recognize faces with nearly 100% accuracy. For a deep learning system, that’s an almost inconceivable goal. And 100% accuracy is orders of magnitude harder than 80% accuracy, or even 97.5%.

What kinds of applications can you build from technologies that are only accurate 80% of the time, or even 97.5% of the time? Quite a few. You might build an application that created dynamic travel guides from online photos. Or you might build an application that measures how long diners stay in a restaurant, how long it takes them to be served, whether they’re smiling, and other statistics. You
Understanding Big Data

Analytics for Enterprise Class Hadoop and Streaming Data

- Learn how IBM hardens Hadoop for enterprise-class scalability and reliability
- Gain insight into IBM's unique in-motion and at-rest Big Data analytics platform
- Learn tips and tricks for Big Data use cases and solutions
- Get a quick Hadoop primer

CHRIS EATON  DIRK DEROOS
TOM DEUTSCH  GEORGE LAPIS
PAUL ZIKOPOULOS
It should be evident now that you’ve read Part I, but we have a hunch you already figured this out before picking up this book—there are mountains of untapped potential in our information. Until now, it’s been too cost prohibitive to analyze these massive volumes. Of course, there’s also been a staggering opportunity cost associated with not tapping into this information, as the potential of this yet-to-be-analyzed information is near-limitless. And we’re not just talking the ubiquitous “competitive differentiation” marketing slogan here; we’re talking innovation, discovery, association, and pretty much anything else that could make the way you work tomorrow very different, with even more tangible results and insight, from the way you work today.

People and organizations have attempted to tackle this problem from many different angles. Of course, the angle that is currently leading the pack in terms of popularity for massive data analysis is an open source project called Hadoop that is shipped as part of the IBM InfoSphere BigInsights (BigInsights) platform. Quite simply, BigInsights embraces, hardens, and extends the Hadoop open source framework with enterprise-grade security, governance, availability, integration into existing data stores, tooling that simplifies and improves developer productivity, scalability, analytic toolkits, and more.

When we wrote this book, we thought it would be beneficial to include a chapter about Hadoop itself, since BigInsights is (and will always be) based on the nonforked core Hadoop distribution, and backwards compatibility
with the Apache Hadoop project will always be maintained. In short, applications written for Hadoop will always run on BigInsights. This chapter isn’t going to make you a Hadoop expert by any means, but after reading it, you’ll understand the basic concepts behind the core Hadoop technology, and you might even sound really smart with the nontechies at the water cooler. If you’re new to Hadoop, this chapter’s for you.

Just the Facts: The History of Hadoop

Hadoop (http://hadoop.apache.org/) is a top-level Apache project in the Apache Software Foundation that’s written in Java. For all intents and purposes, you can think of Hadoop as a computing environment built on top of a distributed clustered file system that was designed specifically for very large-scale data operations.

Hadoop was inspired by Google’s work on its Google (distributed) File System (GFS) and the MapReduce programming paradigm, in which work is broken down into mapper and reducer tasks to manipulate data that is stored across a cluster of servers for massive parallelism. MapReduce is not a new concept (IBM teamed up with Google in October 2007 to do some joint university research on MapReduce and GFS for large-scale Internet problems); however, Hadoop has made it practical to be applied to a much wider set of use cases. Unlike transactional systems, Hadoop is designed to scan through large data sets to produce its results through a highly scalable, distributed batch processing system. Hadoop is not about speed-of-thought response times, real-time warehousing, or blazing transactional speeds; it is about discovery and making the once near-impossible possible from a scalability and analysis perspective. The Hadoop methodology is built around a function-to-data model as opposed to data-to-function; in this model, because there is so much data, the analysis programs are sent to the data (we’ll detail this later in this chapter).

Hadoop is quite the odd name (and you’ll find a lot of odd names in the Hadoop world). Read any book on Hadoop today and it pretty much starts with the name that serves as this project’s mascot, so let’s start there too. Hadoop is actually the name that creator Doug Cutting’s son gave to his stuffed toy elephant. In thinking up a name for his project, Cutting was apparently looking for something that was easy to say and stands for nothing in particular, so the name of his son’s toy seemed to make perfect sense.
Cutting’s naming approach has kicked off a wild collection of names (as you will soon find out), but to be honest, we like it. (We reflected among ourselves about some of the names associated with our kids’ toys while we wrote this book, and we’re glad Cutting dubbed this technology and not us; Pinky and Squiggles don’t sound like good choices.)

Hadoop is generally seen as having two parts: a file system (the *Hadoop Distributed File System*) and a programming paradigm (*MapReduce*)—more on these in a bit. One of the key components of Hadoop is the redundancy built into the environment. Not only is the data redundantly stored in multiple places across the cluster, but the programming model is such that failures are expected and are resolved automatically by running portions of the program on various servers in the cluster. Due to this redundancy, it’s possible to distribute the data and its associated programming across a very large cluster of commodity components. It is well known that commodity hardware components will fail (especially when you have very large numbers of them), but this redundancy provides fault tolerance and a capability for the Hadoop cluster to heal itself. This allows Hadoop to scale out workloads across large clusters of inexpensive machines to work on Big Data problems.

There are a number of Hadoop-related projects, and some of these we cover in this book (and some we don’t, due to its size). Some of the more notable Hadoop-related projects include: Apache Avro (for data serialization), Cassandra and HBase (databases), Chukwa (a monitoring system specifically designed with large distributed systems in mind), Hive (provides ad hoc SQL-like queries for data aggregation and summarization), Mahout (a machine learning library), Pig (a high-level Hadoop programming language that provides a data-flow language and execution framework for parallel computation), ZooKeeper (provides coordination services for distributed applications), and more.

## Components of Hadoop

The Hadoop project is comprised of three pieces: *Hadoop Distributed File System (HDFS)*, the *Hadoop MapReduce* model, and *Hadoop Common*. To understand Hadoop, you must understand the underlying infrastructure of the file system and the MapReduce programming model. Let’s first talk about Hadoop’s file system, which allows applications to be run across multiple servers.
The Hadoop Distributed File System

To understand how it’s possible to scale a Hadoop cluster to hundreds (and even thousands) of nodes, you have to start with HDFS. Data in a Hadoop cluster is broken down into smaller pieces (called blocks) and distributed throughout the cluster. In this way, the map and reduce functions can be executed on smaller subsets of your larger data sets, and this provides the scalability that is needed for Big Data processing.

The goal of Hadoop is to use commonly available servers in a very large cluster, where each server has a set of inexpensive internal disk drives. For higher performance, MapReduce tries to assign workloads to these servers where the data to be processed is stored. This is known as data locality. (It’s because of this principle that using a storage area network (SAN), or network attached storage (NAS), in a Hadoop environment is not recommended. For Hadoop deployments using a SAN or NAS, the extra network communication overhead can cause performance bottlenecks, especially for larger clusters.) Now take a moment and think of a 1000-machine cluster, where each machine has three internal disk drives; then consider the failure rate of a cluster composed of 3000 inexpensive drives + 1000 inexpensive servers!

We’re likely already on the same page here: The component mean time to failure (MTTF) you’re going to experience in a Hadoop cluster is likely analogous to a zipper on your kid’s jacket: it’s going to fail (and poetically enough, zippers seem to fail only when you really need them). The cool thing about Hadoop is that the reality of the MTTF rates associated with inexpensive hardware is actually well understood (a design point if you will), and part of the strength of Hadoop is that it has built-in fault tolerance and fault compensation capabilities. This is the same for HDFS, in that data is divided into blocks, and copies of these blocks are stored on other servers in the Hadoop cluster. That is, an individual file is actually stored as smaller blocks that are replicated across multiple servers in the entire cluster.

Think of a file that contains the phone numbers for everyone in the United States; the people with a last name starting with A might be stored on server 1, B on server 2, and so on. In a Hadoop world, pieces of this phonebook would be stored across the cluster, and to reconstruct the entire phonebook, your program would need the blocks from every server in the cluster. To achieve availability as components fail, HDFS replicates these smaller pieces (see Figure 4-1) onto two additional servers by default. (This redundancy can
be increased or decreased on a per-file basis or for a whole environment; for example, a development Hadoop cluster typically doesn’t need any data redundancy.) This redundancy offers multiple benefits, the most obvious being higher availability. In addition, this redundancy allows the Hadoop cluster to break work up into smaller chunks and run those jobs on all the servers in the cluster for better scalability. Finally, you get the benefit of data locality, which is critical when working with large data sets. We detail these important benefits later in this chapter.

A data file in HDFS is divided into blocks, and the default size of these blocks for Apache Hadoop is 64 MB. For larger files, a higher block size is a good idea, as this will greatly reduce the amount of metadata required by the NameNode. The expected workload is another consideration, as nonsequential access patterns (random reads) will perform more optimally with a smaller block size. In BigInsights, the default block size is 128 MB, because in the experience of IBM Hadoop practitioners, the most common deployments involve larger files and workloads with sequential reads. This is a much larger block size than is used with other environments—for example, typical file
Understanding Big Data systems have an on-disk block size of 512 bytes, whereas relational databases typically store data blocks in sizes ranging from 4 KB to 32 KB. Remember that Hadoop was designed to scan through very large data sets, so it makes sense for it to use a very large block size so that each server can work on a larger chunk of data at the same time. Coordination across a cluster has significant overhead, so the ability to process large chunks of work locally without sending data to other nodes helps improve both performance and the overhead to real work ratio. Recall that each data block is stored by default on three different servers; in Hadoop, this is implemented by HDFS working behind the scenes to make sure at least two blocks are stored on a separate server rack to improve reliability in the event you lose an entire rack of servers.

All of Hadoop’s data placement logic is managed by a special server called NameNode. This NameNode server keeps track of all the data files in HDFS, such as where the blocks are stored, and more. All of the NameNode’s information is stored in memory, which allows it to provide quick response times to storage manipulation or read requests. Now, we know what you’re thinking: If there is only one NameNode for your entire Hadoop cluster, you need to be aware that storing this information in memory creates a single point of failure (SPOF). For this reason, we strongly recommend that the server components you choose for the NameNode be much more robust than the rest of the servers in your Hadoop cluster to minimize the possibility of failures. In addition, we also strongly recommend that you have a regular backup process for the cluster metadata stored in the NameNode. Any data loss in this metadata will result in a permanent loss of corresponding data in the cluster. When this book was written, the next version of Hadoop (version 0.21) was to include the capability to define a BackupNode, which can act as a cold standby for the NameNode.

Figure 4-1 represents a file that is made up of three data blocks, where a data block (denoted as block_n) is replicated on two additional servers (denoted by block_n' and block_n''). The second and third replicas are stored on a separate physical rack, on separate nodes for additional protection.

We’re detailing how HDFS stores data blocks to give you a brief introduction to this Hadoop component. The great thing about the Hadoop MapReduce application framework is that, unlike prior grid technologies, the developer doesn’t have to deal with the concepts of the NameNode and where data is stored—Hadoop does that for you. When you fire off a Hadoop job and the
application has to read data and starts to work on the programmed MapReduce tasks, Hadoop will contact the NameNode, find the servers that hold the parts of the data that need to be accessed to carry out the job, and then send your application to run locally on those nodes. (We cover the details of MapReduce in the next section.) Similarly, when you create a file, HDFS will automatically communicate with the NameNode to allocate storage on specific servers and perform the data replication. It’s important to note that when you’re working with data, there’s no need for your MapReduce code to directly reference the NameNode. Interaction with the NameNode is mostly done when the jobs are scheduled on various servers in the Hadoop cluster. This greatly reduces communications to the NameNode during job execution, which helps to improve scalability of the solution. In summary, the NameNode deals with cluster metadata describing where files are stored; actual data being processed by MapReduce jobs never flows through the NameNode.

In this book, we talk about how IBM brings enterprise capability to Hadoop, and this is one specific area where IBM uses its decades of experience and research to leverage its ubiquitous enterprise IBM General Parallel File System (GPFS) to alleviate these concerns. GPFS initially only ran on SAN technologies. In 2009, GPFS was extended to run on a shared nothing cluster (known as GPFS-SNC) and is intended for use cases like Hadoop. GFPS-SNC provides many advantages over HDFS, and one of them addresses the aforementioned NameNode issue. A Hadoop runtime implemented within GPFS-SNC does not have to contend with this particular SPOF issue. GPFS-SNC allows you to build a more reliable Hadoop cluster (among other benefits such as easier administration and performance).

In addition to the concerns expressed about a single NameNode, some clients have noted that HDFS is not a Portable Operating System Interface for UNIX (POSIX)–compliant file system. What this means is that almost all of the familiar commands you might use in interacting with files (copying files, deleting files, writing to files, moving files, and so on) are available in a different form with HDFS (there are syntactical differences and, in some cases, limitations in functionality). To work around this, you either have to write your own Java applications to perform some of the functions, or train your IT staff to learn the different HDFS commands to manage and manipulate files in the file system. We’ll go into more detail on this topic later in the chapter, but here
we want you to note that this is yet another “Enterprise-rounding” that BigInsights offers to Hadoop environments for Big Data processing. GPFS-SNC is fully compliant with the IEEE-defined POSIX standard that defines an API, shell, and utility interfaces that provide compatibility across different flavors of UNIX (such as AIX, Apple OSX, and HP-UX).

The Basics of MapReduce

MapReduce is the heart of Hadoop. It is this programming paradigm that allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster. The MapReduce concept is fairly simple to understand for those who are familiar with clustered scale-out data processing solutions. For people new to this topic, it can be somewhat difficult to grasp, because it’s not typically something people have been exposed to previously. If you’re new to Hadoop’s MapReduce jobs, don’t worry: we’re going to describe it in a way that gets you up to speed quickly.

The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job.

Let’s look at a simple example. Assume you have five files, and each file contains two columns (a key and a value in Hadoop terms) that represent a city and the corresponding temperature recorded in that city for the various measurement days. Of course we’ve made this example very simple so it’s easy to follow. You can imagine that a real application won’t be quite so simple, as it’s likely to contain millions or even billions of rows, and they might not be neatly formatted rows at all; in fact, no matter how big or small the amount of data you need to analyze, the key principles we’re covering here remain the same. Either way, in this example, city is the key and temperature is the value.

The following snippet shows a sample of the data from one of our test files (incidentally, in case the temperatures have you reaching for a hat and gloves, they are in Celsius):
Out of all the data we have collected, we want to find the maximum temperature for *each* city across all of the data files (note that each file might have the same city represented multiple times). Using the MapReduce framework, we can break this down into five map tasks, where each mapper works on one of the five files and the mapper task goes through the data and returns the maximum temperature for each city. For example, the results produced from one mapper task for the data above would look like this:

(Toronto, 20) (Whitby, 25) (New York, 22) (Rome, 33)

Let’s assume the other four mapper tasks (working on the other four files not shown here) produced the following intermediate results:

(Toronto, 18) (Whitby, 27) (New York, 32) (Rome, 37)
(Toronto, 32) (Whitby, 20) (New York, 33) (Rome, 38)
(Toronto, 22) (Whitby, 19) (New York, 20) (Rome, 31)
(Toronto, 31) (Whitby, 22) (New York, 19) (Rome, 30)

All five of these output streams would be fed into the reduce tasks, which combine the input results and output a single value for each city, producing a final result set as follows:

(Toronto, 32) (Whitby, 27) (New York, 33) (Rome, 38)

As an analogy, you can think of map and reduce tasks as the way a census was conducted in Roman times, where the census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital city. There, the results from each city would be reduced to a single count (sum of all cities) to determine the overall population of the empire. This *mapping* of people to cities, in parallel, and then combining the results (*reducing*) is much more efficient than sending a single person to count every person in the empire in a serial fashion.

In a Hadoop cluster, a MapReduce program is referred to as a *job*. A job is executed by subsequently breaking it down into pieces called *tasks*. 
An application submits a job to a specific node in a Hadoop cluster, which is running a daemon called the *JobTracker*. The JobTracker communicates with the NameNode to find out where all of the data required for this job exists across the cluster, and then breaks the job down into map and reduce tasks for each node to work on in the cluster. These tasks are scheduled on the nodes in the cluster where the data exists. Note that a node might be given a task for which the data needed by that task is not local to that node. In such a case, the node would have to ask for the data to be sent across the network interconnect to perform its task. Of course, this isn’t very efficient, so the JobTracker tries to avoid this and attempts to schedule tasks where the data is stored. This is the concept of data locality we introduced earlier, and it is critical when working with large volumes of data. In a Hadoop cluster, a set of continually running daemons, referred to as *TaskTracker* agents, monitor the status of each task. If a task fails to complete, the status of that failure is reported back to the JobTracker, which will then reschedule that task on another node in the cluster. (You can dictate how many times the task will be attempted before the entire job gets cancelled.)

Figure 4-2 shows an example of a MapReduce flow. You can see that multiple reduce tasks can serve to increase the parallelism and improve the overall performance of the job. In the case of Figure 4-2, the output of the map tasks must be directed (by key value) to the appropriate reduce task. If we apply our maximum temperature example to this figure, all of the records that have a key value of *Toronto* must be sent to the same reduce task to

![Figure 4-2](image-url)
produce an accurate result (one reducer must be able to see all of the temperatures for Toronto to determine the maximum for that city). This directing of records to reduce tasks is known as a Shuffle, which takes input from the map tasks and directs the output to a specific reduce task. Hadoop gives you the option to perform local aggregation on the output of each map task before sending the results off to a reduce task through a local aggregation called a Combiner (but it’s not shown in Figure 4-2). Clearly more work and overhead are involved in running multiple reduce tasks, but for very large datasets, having many reducers can improve overall performance.

All MapReduce programs that run natively under Hadoop are written in Java, and it is the Java Archive file (jar) that’s distributed by the JobTracker to the various Hadoop cluster nodes to execute the map and reduce tasks. For further details on MapReduce, you can review the Apache Hadoop documentation’s tutorial that leverages the ubiquitous Hello World programming language equivalent for Hadoop: WordCount. WordCount is a simple to understand example with all of the Java code needed to run the samples.

Of course, if you’re looking for the fastest and easiest way to get up and running with Hadoop, check out BigDataUniversity.com and download InfoSphere BigInsights Basic Edition (www.ibm.com/software/data/infosphere/biginsights/basic.html). It’s got some of the great IBM add-on capabilities (for example, the whole up and running experience is completely streamlined for you, so you get a running Hadoop cluster in the same manner that you’d see in any commercial software) and more. Most importantly, it’s 100 percent free, and you can optionally buy a support contract for BigInsights’ Basic Edition. Of course, by the time you are finished reading this book, you’ll have a complete grasp as to how IBM InfoSphere BigInsights Enterprise Edition embraces and extends the Hadoop stack to provide the same capabilities expected from other enterprise systems.

Hadoop Common Components

The Hadoop Common Components are a set of libraries that support the various Hadoop subprojects. Earlier in this chapter, we mentioned some of these components in passing. In this section, we want to spend time discussing the file system shell. As mentioned (and this is a really important point, which is why we are making note of it again), HDFS is not a POSIX-compliant file system, which means you can’t interact with it as you would a Linux- or UNIX-
based file system. To interact with files in HDFS, you need to use the 
/bin/hdfs dfs <args> file system shell command interface, where args 
represents the command arguments you want to use on files in the file 
system.

Here are some examples of HDFS shell commands:

- **cat**: Copies the file to standard output (stdout).
- **chmod**: Changes the permissions for reading and writing to a 
given file or set of files.
- **chown**: Changes the owner of a given file or set of files.
- **copyFromLocal**: Copies a file from the local file system into HDFS.
- **copyToLocal**: Copies a file from HDFS to the local file system.
- **cp**: Copies HDFS files from one directory to another.
- **expunge**: Empties all of the files that are in the trash. When you 
delete an HDFS file, the data is not actually gone (think 
of your MAC or Windows-based home computers, and 
you’ll get the point). Deleted HDFS files can be found in 
the trash, which is automatically cleaned at some later 
point in time. If you want to empty the trash 
immediately, you can use the expunge argument.
- **ls**: Displays a listing of files in a given directory.
- **mkdir**: Creates a directory in HDFS.
- **mv**: Moves files from one directory to another.
- **rm**: Deletes a file and sends it to the trash. If you want to skip 
the trash process and delete the file from HDFS on the 
spot, you can use the –skiptrash option of the rm 
command.

**Application Development in Hadoop**

As you probably inferred from the preceding section, the Hadoop platform 
can be a powerful tool for manipulating extremely large data sets. However, 
the core Hadoop MapReduce APIs are primarily called from Java, which 
requires skilled programmers. In addition, it is even more complex for
programmers to develop and maintain MapReduce applications for business applications that require long and pipelined processing.

If you’ve been around programming long enough, you’ll find history has a way of repeating itself. For example, we often cite XML as “The Revenge of IMS” due to its hierarchal nature and retrieval system. In the area of computer language development, just as assembler gave way to structured programming languages and then to the development of 3GL and 4GL languages, so too goes the world of Hadoop application development languages. To abstract some of the complexity of the Hadoop programming model, several application development languages have emerged that run on top of Hadoop. In this section, we cover three of the more popular ones, which admittedly sound like we’re at a zoo: Pig, Hive, and Jaql (by the way, we’ll cover ZooKeeper in this chapter, too).

Pig and PigLatin

Pig was initially developed at Yahoo! to allow people using Hadoop to focus more on analyzing large data sets and spend less time having to write mapper and reducer programs. Like actual pigs, who eat almost anything, the Pig programming language is designed to handle any kind of data—hence the name! Pig is made up of two components: the first is the language itself, which is called PigLatin (yes, people naming various Hadoop projects do tend to have a sense of humor associated with their naming conventions), and the second is a runtime environment where PigLatin programs are executed. Think of the relationship between a Java Virtual Machine (JVM) and a Java application. In this section, we’ll just refer to the whole entity as Pig.

Let’s first look at the programming language itself so that you can see how it’s significantly easier than having to write mapper and reducer programs. The first step in a Pig program is to LOAD the data you want to manipulate from HDFS. Then you run the data through a set of transformations (which, under the covers, are translated into a set of mapper and reducer tasks). Finally, you DUMP the data to the screen or you STORE the results in a file somewhere.

LOAD

As is the case with all the Hadoop features, the objects that are being worked on by Hadoop are stored in HDFS. In order for a Pig program to access this data, the program must first tell Pig what file (or files) it will use, and that’s
done through the `LOAD 'data_file'` command (where `'data_file'` specifies either an HDFS file or directory). If a directory is specified, all the files in that directory will be loaded into the program. If the data is stored in a file format that is not natively accessible to Pig, you can optionally add the `USING` function to the `LOAD` statement to specify a user-defined function that can read in and interpret the data.

**TRANSFORM**

The transformation logic is where all the data manipulation happens. Here you can `FILTER` out rows that are not of interest, `JOIN` two sets of data files, `GROUP` data to build aggregations, `ORDER` results, and much more. The following is an example of a Pig program that takes a file composed of Twitter feeds, selects only those tweets that are using the `en` (English) `iso_language_code`, then groups them by the user who is tweeting, and displays the sum of the number of retweets of that user’s tweets.

```pig
L  = LOAD 'hdfs//node/tweet_data';
FL = FILTER L BY iso_language_code  EQ 'en';
G  = GROUP FL BY from_user;
RT = FOREACH G GENERATE group, SUM(retweets);
```

**DUMP and STORE**

If you don’t specify the `DUMP` or `STORE` command, the results of a Pig program are not generated. You would typically use the `DUMP` command, which sends the output to the screen, when you are debugging your Pig programs. When you go into production, you simply change the `DUMP` call to a `STORE` call so that any results from running your programs are stored in a file for further processing or analysis. Note that you can use the `DUMP` command anywhere in your program to dump intermediate result sets to the screen, which is very useful for debugging purposes.

Now that we’ve got a Pig program, we need to have it run in the Hadoop environment. Here is where the Pig runtime comes in. There are three ways to run a Pig program: embedded in a script, embedded in a Java program, or from the Pig command line, called `Grunt` (which is of course the sound a pig makes—we told you that the Hadoop community has a lighter side).

No matter which of the three ways you run the program, the Pig runtime environment translates the program into a set of `map` and `reduce` tasks and runs them under the covers on your behalf. This greatly simplifies the work
associated with the analysis of large amounts of data and lets the developer focus on the analysis of the data rather than on the individual map and reduce tasks.

Hive

Although Pig can be quite a powerful and simple language to use, the downside is that it’s something new to learn and master. Some folks at Facebook developed a runtime Hadoop support structure that allows anyone who is already fluent with SQL (which is commonplace for relational database developers) to leverage the Hadoop platform right out of the gate. Their creation, called Hive, allows SQL developers to write Hive Query Language (HQL) statements that are similar to standard SQL statements; now you should be aware that HQL is limited in the commands it understands, but it is still pretty useful. HQL statements are broken down by the Hive service into MapReduce jobs and executed across a Hadoop cluster.

For anyone with a SQL or relational database background, this section will look very familiar to you. As with any database management system (DBMS), you can run your Hive queries in many ways. You can run them from a command line interface (known as the Hive shell), from a Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC) application leveraging the Hive JDBC/ODBC drivers, or from what is called a Hive Thrift Client. The Hive Thrift Client is much like any database client that gets installed on a user’s client machine (or in a middle tier of a three-tier architecture): it communicates with the Hive services running on the server. You can use the Hive Thrift Client within applications written in C++, Java, PHP, Python, or Ruby (much like you can use these client-side languages with embedded SQL to access a database such as DB2 or Informix). The following shows an example of creating a table, populating it, and then querying that table using Hive:

```
CREATE TABLE Tweets(from_user STRING, userid BIGINT, tweettext STRING, retweets INT)
  COMMENT 'This is the Twitter feed table'
STORED AS SEQUENCEFILE;
LOAD DATA INPATH 'hdfs://node/tweetdata' INTO TABLE TWEETS;
SELECT from_user, SUM(retweets)
FROM TWEETS
GROUP BY from_user;
```
As you can see, Hive looks very much like traditional database code with SQL access. However, because Hive is based on Hadoop and MapReduce operations, there are several key differences. The first is that Hadoop is intended for long sequential scans, and because Hive is based on Hadoop, you can expect queries to have a very high latency (many minutes). This means that Hive would not be appropriate for applications that need very fast response times, as you would expect with a database such as DB2. Finally, Hive is read-based and therefore not appropriate for transaction processing that typically involves a high percentage of write operations.

**Jaql**

Jaql is primarily a query language for JavaScript Object Notation (JSON), but it supports more than just JSON. It allows you to process both structured and nontraditional data and was donated by IBM to the open source community (just one of many contributions IBM has made to open source). Specifically, Jaql allows you to select, join, group, and filter data that is stored in HDFS, much like a blend of Pig and Hive. Jaql’s query language was inspired by many programming and query languages, including Lisp, SQL, XQuery, and Pig. Jaql is a functional, declarative query language that is designed to process large data sets. For parallelism, Jaql rewrites high-level queries, when appropriate, into “low-level” queries consisting of MapReduce jobs.

Before we get into the Jaql language, let’s first look at the popular data interchange format known as JSON, so that we can build our Jaql examples on top of it. Application developers are moving in large numbers towards JSON as their choice for a data interchange format, because it’s easy for humans to read, and because of its structure, it’s easy for applications to parse or generate.

JSON is built on top of two types of structures. The first is a collection of name/value pairs (which, as you learned earlier in the “The Basics of MapReduce” section, makes it ideal for data manipulation in Hadoop, which works on key/value pairs). These name/value pairs can represent anything since they are simply text strings (and subsequently fit well into existing models) that could represent a record in a database, an object, an associative array, and more. The second JSON structure is the ability to create an ordered list of values much like an array, list, or sequence you might have in your existing applications.
An object in JSON is represented as { string : value }, where an array can be simply represented by [ value, value, ... ], where value can be a string, number, another JSON object, or another JSON array. The following shows an example of a JSON representation of a Twitter feed (we’ve removed many of the fields that are found in the tweet syntax to enhance readability):

```json
results: [
  {
    created_at: "Thurs, 14 Jul 2011 09:47:45 +0000",
    from_user: "eatonchris",
    geo: {
      coordinates: [43.866667, 78.933333],
      type: "Point"
    },
    iso_language_code: "en",
    text: "Reliance Life Insurance migrates from #Oracle to #DB2 and cuts costs in half. Read what they say about their migration http://bit.ly/pP7vaT",
    retweet: 3,
    to_user_id: null,
    to_user_id_str: null
  }
]
```

Both Jaql and JSON are record-oriented models, and thus fit together perfectly. Note that JSON is not the only format that Jaql supports—in fact, Jaql is extremely flexible and can support many semistructured data sources such as XML, CSV, flat files, and more. However, in consideration of the space we have, we’ll use the JSON example above in the following Jaql queries. As you will see from this section, Jaql looks very similar to Pig but also has some similarity to SQL.

**Jaql Operators**

Jaql is built on a set of core operators. Let’s look at some of the most popular operators found in Jaql, how they work, and then go through some simple examples that will allow us to query the Twitter feed represented earlier.

**FILTER**  
The FILTER operator takes an array as input and filters out the elements of interest based on a specified predicate. For those familiar with SQL, think of the FILTER operator as a WHERE clause. For example, if you
want to look only at the input records from the Twitter feed that were created by user eatonchris, you’d put something similar to the following in your query:

```javascript
filter $.from_user == "eatonchris"
```

If you wanted to see only the tweets that have been retweeted more than twice, you would include a Jaql query such as this:

```javascript
filter $.retweet > 2
```

**TRANSFORM**  The **TRANSFORM** operator takes an array as input and outputs another array where the elements of the first array have been transformed in some way. For SQL addicts, you’ll find this similar to the **SELECT** clause. For example, if an input array has two numbers denoted by $N1$ and $N2$, the **TRANSFORM** operator could produce the sum of these two numbers using the following:

```javascript
transform { sum: $.N1 + $.N2 }
```

**GROUP**  The **GROUP** operator works much like the **GROUP BY** clause in SQL, where a set of data is aggregated for output. For example, if you wanted to count the total number of tweets in this section’s working example, you could use this:

```javascript
group into count($)
```

Likewise, if you wanted to determine the sum of all retweets by user, you would use a Jaql query such as this:

```javascript
group by u = $.from_user into { total: sum($.retweet) }
```

**JOIN**  The **JOIN** operator takes two input arrays and produces an output array based on the join condition specified in the **WHERE** clause—similar to a join operation in SQL. Let’s assume you have an array of tweets (such as the JSON tweet example) and you also have a set of interesting data that comes from a group of the people whom you follow on Twitter. Such an array may look like this:

```javascript
following = { from_user: "eatonchris" },
            { from_user: "paulzikopoulos" }
```
In this example, you could use the `JOIN` operator to join the Twitter `feed` data with the Twitter `following` data to produce results for only the tweets from people you follow, like so:

```
join feed, follow
where feed.from_user = following.from_user
into {feed.*}
```

**EXPAND**  The `EXPAND` operator takes a nested array as input and produces a single array as output. Let’s assume you have a nested array of geographic locations (denoted with latitude and longitude coordinates) as shown here:

```
geolocations = [[[93.456, 123.222],[21.324, 90.456]]
```

In this case, the `geolocations -> expand;` command would return results in a single array as follows:

```
[93.456, 123.222, 21.324, 90.456]
```

**SORT**  As you might expect, the `SORT` operator takes an array as input and produces an array as output, where the elements are in a sorted order. The default Jaql sort order is ascending. You can sort Jaql results in a descending order using the `sort by desc` keyword.

**TOP**  The `TOP` operator returns the first `n` elements of the input array, where `n` is an `<integer>` that follows the `TOP` keyword.

**Built-in Jaql Functions**

In addition to the core operators, Jaql also has a large set of built-in functions that allow you to read in, manipulate, and write out data, as well as call external functions such as HDFS calls, and more. You can add your own custom-built functions, which can, in turn, invoke other functions. The more than 100 built-in functions are obviously too many to cover in this book; however, they are well documented in the base Jaql documentation.

**A Jaql Query**

Much like a MapReduce job is a flow of data, Jaql can be thought of as a pipeline of data flowing from a source, through a set of various operators, and out into a sink (a destination). The operand used to signify flow from one
operand to another is an arrow: \(-\rightarrow\). Unlike SQL, where the output comes first (for example, the SELECT list), in Jaql, the operations listed are in natural order, where you specify the source, followed by the various operators you want to use to manipulate the data, and finally the sink.

Let’s wrap up this Jaql section and put it all together with a simple Jaql example that counts the number of tweets written in English by user:

\[
\begin{align*}
\$tweets & = \text{read(hdfs("tweet_log"))}; \\
\$tweets & \rightarrow \text{filter \(\$.iso\_language\_code = "en"\)} \\
& \rightarrow \text{group by} \ u = \$.from\_user \\
& \quad \into \{ \text{user: \$.from\_user, total: sum(\$.retweet)} \};
\end{align*}
\]

The first line simply opens up the file containing the data (with the intent to read it), which resides in HDFS, and assigns it a name, which in this case is \$tweets. Next, the Jaql query reads \$tweets and passes the data to the FILTER operator. The filter only passes on tweets that have an iso_language_code = en. These records are subsequently passed to the GROUP BY operator that adds the retweet values for each user together to get a sum for each given user.

Internally, the Jaql engine transforms the query into map and reduce tasks that can significantly reduce the application development time associated with analyzing massive amounts of data in Hadoop. Note that we’ve shown only the relationship between Jaql and JSON in this chapter; it’s important to realize that this is not the only data format with which Jaql works. In fact, quite the contrary is true: Jaql is a flexible infrastructure for managing and analyzing many kinds of semistructured data such as XML, CSV data, flat files, relational data, and so on. In addition, from a development perspective, don’t forget that the Jaql infrastructure is extremely flexible and extensible, and allows for the passing of data between the query interface and the application language of your choice (for example, Java, JavaScript, Python, Perl, Ruby, and so on).

**Hadoop Streaming**

In addition to Java, you can write map and reduce functions in other languages and invoke them using an API known as Hadoop Streaming (Streaming, for short). Streaming is based on the concept of UNIX streaming, where
input is read from standard input, and output is written to standard output. These data streams represent the interface between Hadoop and your applications.

The Streaming interface lends itself best to short and simple applications you would typically develop using a scripting language such as Python or Ruby. A major reason for this is the text-based nature of the data flow, where each line of text represents a single record.

The following example shows the execution of map and reduce functions (written in Python) using Streaming:

For example:

```bash
hadoop jar contrib/streaming/hadoop-streaming.jar \
  -input input/dataset.txt \
  -output output \
  -mapper text_processor_map.py \
  -reducer text_processor_reduce.py
```

---

**Getting Your Data into Hadoop**

One of the challenges with HDFS is that it’s not a POSIX-compliant file system. This means that all the things you are accustomed to when it comes to interacting with a typical file system (copying, creating, moving, deleting, or accessing a file, and more) don’t automatically apply to HDFS. To do anything with a file in HDFS, you must use the HDFS interfaces or APIs directly. That is yet another advantage of using the GPFS-SNC file system; with GPFS-SNC, you interact with your Big Data files in the same manner that you would any other file system, and, therefore, file manipulation tasks with Hadoop running on GPFS-SNC are greatly reduced. In this section, we discuss the basics of getting your data into HDFS and cover **Flume**, which is a distributed data collection service for flowing data into a Hadoop cluster.

**Basic Copy Data**

As you’ll recall from the “Hadoop Common Components” section earlier in the chapter, you must use specific commands to move files into HDFS either through APIs or using the command shell. The most common way to move files from a local file system into HDFS is through the copyFromLocal
command. To get files out of HDFS to the local file system, you’ll typically use the `copyToLocal` command. An example of each of these commands is shown here:

```
    hdfs dfs -copyFromLocal /user/dir/file hdfs://s1.n1.com/dir/hdfsfile
    hdfs dfs -copyToLocal hdfs://s1.n1.com/dir/hdfsfile /user/dir/file
```

These commands are run through the HDFS shell program, which is simply a Java application. The shell uses the Java APIs for getting data into and out of HDFS. These APIs can be called from any Java application.

**NOTE** HDFS commands can also be issued through the Hadoop shell, which is invoked by the command `hadoop fs`.

The problem with this method is that you must have Java application developers write the logic and programs to read and write data from HDFS. Other methods are available (such as C++ APIs, or via the Thrift framework for cross-language services), but these are merely wrappers for the base Java APIs. If you need to access HDFS files from your Java applications, you would use the methods in the `org.apache.hadoop.fs` package. This allows you to incorporate read and write operations directly, to and from HDFS, from within your MapReduce applications. Note, however, that HDFS is designed for sequential read and write. This means when you write data to an HDFS file, you can write only to the end of the file (it’s referred to as an APPEND in the database world). Herein lies yet another advantage to using GPFS-SNC as the file system backbone for your Hadoop cluster, because this specialized file system has the inherent ability to seek and write within a file, not just at the end of a file.

**Flume**

A flume is a channel that directs water from a source to some other location where water is needed. As its clever name implies, Flume was created (as of the time this book was published, it was an incubator Apache project) to allow you to flow data from a source into your Hadoop environment. In Flume, the entities you work with are called sources, decorators, and sinks. A source can be any data source, and Flume has many predefined source adapters, which we’ll discuss in this section. A sink is the target of a specific operation (and in Flume, among other paradigms that use this term, the sink of one operation
can be the source for the next downstream operation). A *decorator* is an operation on the stream that can transform the stream in some manner, which could be to compress or uncompress data, modify data by adding or removing pieces of information, and more.

A number of predefined source adapters are built into Flume. For example, some adapters allow the flow of anything coming off a TCP port to enter the flow, or anything coming to standard input (*stdin*). A number of text file source adapters give you the granular control to grab a specific file and feed it into a data flow or even take the tail of a file and continuously feed the flow with whatever new data is written to that file. The latter is very useful for feeding diagnostic or web logs into a data flow, since they are constantly being appended to, and the **TAIL** operator will continuously grab the latest entries from the file and put them into the flow. A number of other predefined source adapters, as well as a command exit, allow you to use any executable command to feed the flow of data.

There are three types of sinks in Flume. One sink is basically the final flow destination and is known as a *Collector Tier Event* sink. This is where you would land a flow (or possibly multiple flows joined together) into an HDFS-formatted file system. Another sink type used in Flume is called an *Agent Tier Event*; this sink is used when you want the sink to be the input source for another operation. When you use these sinks, Flume will also ensure the integrity of the flow by sending back acknowledgments that data has actually arrived at the sink. The final sink type is known as a *Basic* sink, which can be a text file, the console display, a simple HDFS path, or a null bucket where the data is simply deleted.

Look to Flume when you want to flow data from many sources (it was designed for log data, but it can be used for other kinds of data too), manipulate it, and then drop it into your Hadoop environment. Of course, when you want to perform very complex transformations and cleansing of your data, you should be looking at an enterprise-class data quality toolset such as IBM Information Server, which provides services for transformation, extraction, discovery, quality, remediation, and more. IBM Information Server can handle large-scale data manipulations prior to working on the data in a Hadoop cluster, and integration points are provided (with more coming) between the technologies (for instance the ability to see data lineage).
Other Hadoop Components

Many other open source projects fall under the Hadoop umbrella, either as Hadoop subprojects or as top-level Apache projects, with more popping up as time goes on (and as you may have guessed, their names are just as interesting: ZooKeeper, HBase, Oozie, Lucene, and more). In this section, we cover four additional Hadoop-related projects that you might encounter (all of which are shipped as part of any InfoSphere BigInsights edition).

ZooKeeper

*ZooKeeper* is an open source Apache project that provides a centralized infrastructure and services that enable synchronization across a cluster. ZooKeeper maintains common objects needed in large cluster environments. Examples of these objects include configuration information, hierarchical naming space, and so on. Applications can leverage these services to coordinate distributed processing across large clusters.

Imagine a Hadoop cluster spanning 500 or more commodity servers. If you’ve ever managed a database cluster with just 10 servers, you know there’s a need for centralized management of the entire cluster in terms of name services, group services, synchronization services, configuration management, and more. In addition, many other open source projects that leverage Hadoop clusters require these types of cross-cluster services, and having them available in ZooKeeper means that each of these projects can embed ZooKeeper without having to build synchronization services from scratch into each project. Interaction with ZooKeeper occurs via Java or C interfaces at this time (our guess is that in the future the Open Source community will add additional development languages that interact with ZooKeeper).

ZooKeeper provides an infrastructure for cross-node synchronization and can be used by applications to ensure that tasks across the cluster are serialized or synchronized. It does this by maintaining status type information in memory on ZooKeeper servers. A ZooKeeper server is a machine that keeps a copy of the state of the entire system and persists this information in local log files. A very large Hadoop cluster can be supported by multiple ZooKeeper servers (in this case, a master server synchronizes the top-level servers). Each client machine communicates with one of the ZooKeeper servers to retrieve and update its synchronization information.
Within ZooKeeper, an application can create what is called a znode (a file that persists in memory on the ZooKeeper servers). The znode can be updated by any node in the cluster, and any node in the cluster can register to be informed of changes to that znode (in ZooKeeper parlance, a server can be set up to “watch” a specific znode). Using this znode infrastructure (and there is much more to this such that we can’t even begin to do it justice in this section), applications can synchronize their tasks across the distributed cluster by updating their status in a ZooKeeper znode, which would then inform the rest of the cluster of a specific node’s status change. This cluster-wide status centralization service is essential for management and serialization tasks across a large distributed set of servers.

**HBase**

*HBase* is a column-oriented database management system that runs on top of HDFS. It is well suited for sparse data sets, which are common in many Big Data use cases. Unlike relational database systems, HBase does not support a structured query language like SQL; in fact, HBase isn’t a relational data store at all. HBase applications are written in Java much like a typical MapReduce application. HBase does support writing applications in Avro, REST, and Thrift. (We briefly cover Avro at the end of this chapter, and the other two aren’t covered in this book, but you can find details about them easily with a simple Google search.)

An HBase system comprises a set of tables. Each table contains rows and columns, much like a traditional database. Each table must have an element defined as a Primary Key, and all access attempts to HBase tables must use this Primary Key. An HBase column represents an attribute of an object; for example, if the table is storing diagnostic logs from servers in your environment, where each row might be a log record, a typical column in such a table would be the timestamp of when the log record was written, or perhaps the servername where the record originated. In fact, HBase allows for many attributes to be grouped together into what are known as *column families*, such that the elements of a column family are all stored together. This is different from a row-oriented relational database, where all the columns of a given row are stored together. With HBase you must predefine the table schema and specify the column families. However, it’s very flexible in that
new columns can be added to families at any time, making the schema flexible and therefore able to adapt to changing application requirements.

Just as HDFS has a NameNode and slave nodes, and MapReduce has JobTracker and TaskTracker slaves, HBase is built on similar concepts. In HBase a master node manages the cluster and region servers store portions of the tables and perform the work on the data. In the same way HDFS has some enterprise concerns due to the availability of the NameNode (among other areas that can be “hardened” for true enterprise deployments by BigInsights), HBase is also sensitive to the loss of its master node.

Oozie

As you have probably noticed in our discussion on MapReduce capabilities, many jobs might need to be chained together to satisfy a complex application. Oozie is an open source project that simplifies workflow and coordination between jobs. It provides users with the ability to define actions and dependencies between actions. Oozie will then schedule actions to execute when the required dependencies have been met.

A workflow in Oozie is defined in what is called a Directed Acyclical Graph (DAG). Acyclical means there are no loops in the graph (in other words, there’s a starting point and an ending point to the graph), and all tasks and dependencies point from start to end without going back. A DAG is made up of action nodes and dependency nodes. An action node can be a MapReduce job, a Pig application, a file system task, or a Java application. Flow control in the graph is represented by node elements that provide logic based on the input from the preceding task in the graph. Examples of flow control nodes are decisions, forks, and join nodes.

A workflow can be scheduled to begin based on a given time or based on the arrival of some specific data in the file system. After inception, further workflow actions are executed based on the completion of the previous actions in the graph. Figure 4-3 is an example of an Oozie workflow, where the nodes represent the actions and control flow operations.

Lucene

Lucene is an extremely popular open source Apache project for text search and is included in many open source projects. Lucene predates Hadoop and
has been a top-level Apache project since 2005. Lucene provides full text indexing and searching libraries for use within your Java application. (Note that Lucene has been ported to C++, Python, Perl, and more.) If you’ve searched on the Internet, it’s likely that you’ve interacted with Lucene (although you probably didn’t know it).

The Lucene concept is fairly simple, yet the use of these search libraries can be very powerful. In a nutshell, let’s say you need to search within a collection of text, or a set of documents. Lucene breaks down these documents into text fields and builds an index on these fields. The index is the key component of Lucene, as it forms the basis for rapid text search capabilities. You then use the searching methods within the Lucene libraries to find the text components. This indexing and search platform is shipped with BigInsights and is integrated into Jaql, providing the ability to build, scan, and query Lucene indexes within Jaql.

BigInsights adds even greater capabilities by shipping a very robust text extraction library to glean structure out of unstructured text, which natively runs on BigInsights and leverages MapReduce. There’s even a development framework to extend and customize the library with a complete tooling environment to make it relatively easy to use. By adding these text extractors to the text indexing capability, BigInsights provides one of the most feature-rich and powerful text analytics platforms for Hadoop available on the market today. What’s more, you can’t store a Lucene index in HDFS; however, you can store it with your other Hadoop data in GPFS-SNC.
Avro

Avro is an Apache project that provides data serialization services. When writing Avro data to a file, the schema that defines that data is always written to the file. This makes it easy for any application to read the data at a later time, because the schema defining the data is stored within the file. There’s an added benefit to the Avro process: Data can be versioned by the fact that a schema change in an application can be easily handled because the schema for the older data remains stored within the data file. An Avro schema is defined using JSON, which we briefly discussed earlier in the “Jaql” section.

A schema defines the data types contained within a file and is validated as the data is written to the file using the Avro APIs. Similarly, the data can be formatted based on the schema definition as the data is read back from the file. The schema allows you to define two types of data. The first are the primitive data types such as STRING, INT[eger], LONG, FLOAT, DOUBLE, BYTE, NULL, and BOOLEAN. The second are complex type definitions. A complex type can be a record, an array, an enum (which defines an enumerated list of possible values for a type), a map, a union (which defines a type to be one of several types), or a fixed type.

APIs for Avro are available in C, C++, C#, Java, Python, Ruby, and PHP, making it available to most application development environments that are common around Hadoop.

Wrapping It Up

As you can see, Hadoop is more than just a single project, but rather an ecosystem of projects all targeted at simplifying, managing, coordinating, and analyzing large sets of data. IBM InfoSphere BigInsights fully embraces this ecosystem with code committers, contributions, and a no-fork backwards compatibility commitment. In the next chapter, we’ll specifically look at the things IBM does to extend Hadoop and its related technologies into an analytics platform enriched with the enterprise-class experience IBM brings to this partnership.
Mining of Massive Datasets

Jure Leskovec
Stanford Univ.

Anand Rajaraman
Milliway Labs

Jeffrey D. Ullman
Stanford Univ.

Chapter 1

Data Mining

In this introductory chapter we begin with the essence of data mining and a discussion of how data mining is treated by the various disciplines that contribute to this field. We cover “Bonferroni’s Principle,” which is really a warning about overusing the ability to mine data. This chapter is also the place where we summarize a few useful ideas that are not data mining but are useful in understanding some important data-mining concepts. These include the TF.IDF measure of word importance, behavior of hash functions and indexes, and identities involving $e$, the base of natural logarithms. Finally, we give an outline of the topics covered in the balance of the book.

1.1 What is Data Mining?

The most commonly accepted definition of “data mining” is the discovery of “models” for data. A “model,” however, can be one of several things. We mention below the most important directions in modeling.

1.1.1 Statistical Modeling

Statisticians were the first to use the term “data mining.” Originally, “data mining” or “data dredging” was a derogatory term referring to attempts to extract information that was not supported by the data. Section 1.2 illustrates the sort of errors one can make by trying to extract what really isn’t in the data. Today, “data mining” has taken on a positive meaning. Now, statisticians view data mining as the construction of a statistical model, that is, an underlying distribution from which the visible data is drawn.

Example 1.1: Suppose our data is a set of numbers. This data is much simpler than data that would be data-mined, but it will serve as an example. A statistician might decide that the data comes from a Gaussian distribution and use a formula to compute the most likely parameters of this Gaussian. The mean
and standard deviation of this Gaussian distribution completely characterize the
distribution and would become the model of the data.

1.1.2 Machine Learning

There are some who regard data mining as synonymous with machine learning. There is no question that some data mining appropriately uses algorithms from machine learning. Machine-learning practitioners use the data as a training set, to train an algorithm of one of the many types used by machine-learning practitioners, such as Bayes nets, support-vector machines, decision trees, hidden Markov models, and many others.

There are situations where using data in this way makes sense. The typical case where machine learning is a good approach is when we have little idea of what we are looking for in the data. For example, it is rather unclear what it is about movies that makes certain movie-goers like or dislike it. Thus, in answering the “Netflix challenge” to devise an algorithm that predicts the ratings of movies by users, based on a sample of their responses, machine-learning algorithms have proved quite successful. We shall discuss a simple form of this type of algorithm in Section 9.4.

On the other hand, machine learning has not proved successful in situations where we can describe the goals of the mining more directly. An interesting case in point is the attempt by WhizBang! Labs\footnote{This startup attempted to use machine learning to mine large-scale data, and hired many of the top machine-learning people to do so. Unfortunately, it was not able to survive.} to use machine learning to locate people’s resumes on the Web. It was not able to do better than algorithms designed by hand to look for some of the obvious words and phrases that appear in the typical resume. Since everyone who has looked at or written a resume has a pretty good idea of what resumes contain, there was no mystery about what makes a Web page a resume. Thus, there was no advantage to machine-learning over the direct design of an algorithm to discover resumes.

1.1.3 Computational Approaches to Modeling

More recently, computer scientists have looked at data mining as an algorithmic problem. In this case, the model of the data is simply the answer to a complex query about it. For instance, given the set of numbers of Example 1.1, we might compute their average and standard deviation. Note that these values might not be the parameters of the Gaussian that best fits the data, although they will almost certainly be very close if the size of the data is large.

There are many different approaches to modeling data. We have already mentioned the possibility of constructing a statistical process whereby the data could have been generated. Most other approaches to modeling can be described as either

1. Summarizing the data succinctly and approximately, or
1.1. WHAT IS DATA MINING?

2. Extracting the most prominent features of the data and ignoring the rest.

We shall explore these two approaches in the following sections.

1.1.4 Summarization

One of the most interesting forms of summarization is the PageRank idea, which made Google successful and which we shall cover in Chapter 5. In this form of Web mining, the entire complex structure of the Web is summarized by a single number for each page. This number, the “PageRank” of the page, is (oversimplifying somewhat) the probability that a random walker on the graph would be at that page at any given time. The remarkable property this ranking has is that it reflects very well the “importance” of the page – the degree to which typical searchers would like that page returned as an answer to their search query.

Another important form of summary – clustering – will be covered in Chapter 7. Here, data is viewed as points in a multidimensional space. Points that are “close” in this space are assigned to the same cluster. The clusters themselves are summarized, perhaps by giving the centroid of the cluster and the average distance from the centroid of points in the cluster. These cluster summaries become the summary of the entire data set.

Example 1.2: A famous instance of clustering to solve a problem took place long ago in London, and it was done entirely without computers. The physician John Snow, dealing with a Cholera outbreak plotted the cases on a map of the city. A small illustration suggesting the process is shown in Fig. 1.1.

![Figure 1.1: Plotting cholera cases on a map of London](http://en.wikipedia.org/wiki/1854_Broad_Street_cholera_outbreak)
The cases clustered around some of the intersections of roads. These inter-
sections were the locations of wells that had become contaminated; people who
lived nearest these wells got sick, while people who lived nearer to wells that
had not been contaminated did not get sick. Without the ability to cluster the
data, the cause of Cholera would not have been discovered.

1.1.5 Feature Extraction

The typical feature-based model looks for the most extreme examples of a phe-
nomenon and represents the data by these examples. If you are familiar with
Bayes nets, a branch of machine learning and a topic we do not cover in this
book, you know how a complex relationship between objects is represented by
finding the strongest statistical dependencies among these objects and using
only those in representing all statistical connections. Some of the important
kinds of feature extraction from large-scale data that we shall study are:

1. **Frequent Itemsets.** This model makes sense for data that consists of “bas-
kets” of small sets of items, as in the market-basket problem that we shall
discuss in Chapter 6. We look for small sets of items that appear together
in many baskets, and these “frequent itemsets” are the characterization of
the data that we seek. The original application of this sort of mining was
true market baskets: the sets of items, such as hamburger and ketchup,
that people tend to buy together when checking out at the cash register
of a store or super market.

2. **Similar Items.** Often, your data looks like a collection of sets, and the
objective is to find pairs of sets that have a relatively large fraction of
their elements in common. An example is treating customers at an on-
line store like Amazon as the set of items they have bought. In order
for Amazon to recommend something else they might like, Amazon can
look for “similar” customers and recommend something many of these
customers have bought. This process is called “collaborative filtering.”
If customers were single-minded, that is, they bought only one kind of
thing, then clustering customers might work. However, since customers
tend to have interests in many different things, it is more useful to find,
for each customer, a small number of other customers who are similar
in their tastes, and represent the data by these connections. We discuss
similarity in Chapter 3.

1.2 Statistical Limits on Data Mining

A common sort of data-mining problem involves discovering unusual events
hidden within massive amounts of data. This section is a discussion of the
problem, including “Bonferroni’s Principle,” a warning against overzealous use
of data mining.
1.2.1 Total Information Awareness

In 2002, the Bush administration put forward a plan to mine all the data it could find, including credit-card receipts, hotel records, travel data, and many other kinds of information in order to track terrorist activity. This idea naturally caused great concern among privacy advocates, and the project, called TIA, or Total Information Awareness, was eventually killed by Congress, although it is unclear whether the project in fact exists under another name. It is not the purpose of this book to discuss the difficult issue of the privacy-security tradeoff. However, the prospect of TIA or a system like it does raise technical questions about its feasibility and the realism of its assumptions.

The concern raised by many is that if you look at so much data, and you try to find within it activities that look like terrorist behavior, are you not going to find many innocent activities – or even illicit activities that are not terrorism – that will result in visits from the police and maybe worse than just a visit? The answer is that it all depends on how narrowly you define the activities that you look for. Statisticians have seen this problem in many guises and have a theory, which we introduce in the next section.

1.2.2 Bonferroni’s Principle

Suppose you have a certain amount of data, and you look for events of a certain type within that data. You can expect events of this type to occur, even if the data is completely random, and the number of occurrences of these events will grow as the size of the data grows. These occurrences are “bogus,” in the sense that they have no cause other than that random data will always have some number of unusual features that look significant but aren’t. A theorem of statistics, known as the Bonferroni correction gives a statistically sound way to avoid most of these bogus positive responses to a search through the data. Without going into the statistical details, we offer an informal version, Bonferroni’s principle, that helps us avoid treating random occurrences as if they were real. Calculate the expected number of occurrences of the events you are looking for, on the assumption that data is random. If this number is significantly larger than the number of real instances you hope to find, then you must expect almost anything you find to be bogus, i.e., a statistical artifact rather than evidence of what you are looking for. This observation is the informal statement of Bonferroni’s principle.

In a situation like searching for terrorists, where we expect that there are few terrorists operating at any one time, Bonferroni’s principle says that we may only detect terrorists by looking for events that are so rare that they are unlikely to occur in random data. We shall give an extended example in the next section.
1.2.3 An Example of Bonferroni’s Principle

Suppose there are believed to be some “evil-doers” out there, and we want to detect them. Suppose further that we have reason to believe that periodically, evil-doers gather at a hotel to plot their evil. Let us make the following assumptions about the size of the problem:

1. There are one billion people who might be evil-doers.
2. Everyone goes to a hotel one day in 100.
3. A hotel holds 100 people. Hence, there are 100,000 hotels – enough to hold the 1% of a billion people who visit a hotel on any given day.
4. We shall examine hotel records for 1000 days.

To find evil-doers in this data, we shall look for people who, on two different days, were both at the same hotel. Suppose, however, that there really are no evil-doers. That is, everyone behaves at random, deciding with probability 0.01 to visit a hotel on any given day, and if so, choosing one of the \(10^5\) hotels at random. Would we find any pairs of people who appear to be evil-doers?

We can do a simple approximate calculation as follows. The probability of any two people both deciding to visit a hotel on any given day is .0001. The chance that they will visit the same hotel is this probability divided by \(10^5\), the number of hotels. Thus, the chance that they will visit the same hotel on one given day is \(10^{-9}\). The chance that they will visit the same hotel on two different given days is the square of this number, \(10^{-18}\). Note that the hotels can be different on the two days.

Now, we must consider how many events will indicate evil-doing. An “event” in this sense is a pair of people and a pair of days, such that the two people were at the same hotel on each of the two days. To simplify the arithmetic, note that for large \(n\), \(\binom{n}{2}\) is about \(n^2/2\). We shall use this approximation in what follows. Thus, the number of pairs of people is \(\binom{10^9}{2} = 5 \times 10^{17}\). The number of pairs of days is \(\binom{1000}{2} = 5 \times 10^5\). The expected number of events that look like evil-doing is the product of the number of pairs of people, the number of pairs of days, and the probability that any one pair of people and pair of days is an instance of the behavior we are looking for. That number is

\[
5 \times 10^{17} \times 5 \times 10^5 \times 10^{-18} = 250,000
\]

That is, there will be a quarter of a million pairs of people who look like evil-doers, even though they are not.

Now, suppose there really are 10 pairs of evil-doers out there. The police will need to investigate a quarter of a million other pairs in order to find the real evil-doers. In addition to the intrusion on the lives of half a million innocent people, the work involved is sufficiently great that this approach to finding evil-doers is probably not feasible.
1.3. THINGS USEFUL TO KNOW

1.2.4 Exercises for Section 1.2

Exercise 1.2.1: Using the information from Section 1.2.3, what would be the number of suspected pairs if the following changes were made to the data (and all other numbers remained as they were in that section)?

(a) The number of days of observation was raised to 2000.

(b) The number of people observed was raised to 2 billion (and there were therefore 200,000 hotels).

(c) We only reported a pair as suspect if they were at the same hotel at the same time on three different days.

Exercise 1.2.2: Suppose we have information about the supermarket purchases of 100 million people. Each person goes to the supermarket 100 times in a year and buys 10 of the 1000 items that the supermarket sells. We believe that a pair of terrorists will buy exactly the same set of 10 items (perhaps the ingredients for a bomb?) at some time during the year. If we search for pairs of people who have bought the same set of items, would we expect that any such people found were truly terrorists?3

1.3 Things Useful to Know

In this section, we offer brief introductions to subjects that you may or may not have seen in your study of other courses. Each will be useful in the study of data mining. They include:

1. The TF.IDF measure of word importance.

2. Hash functions and their use.

3. Secondary storage (disk) and its effect on running time of algorithms.

4. The base e of natural logarithms and identities involving that constant.

5. Power laws.

1.3.1 Importance of Words in Documents

In several applications of data mining, we shall be faced with the problem of categorizing documents (sequences of words) by their topic. Typically, topics are identified by finding the special words that characterize documents about that topic. For instance, articles about baseball would tend to have many occurrences of words like “ball,” “bat,” “pitch,” “run,” and so on. Once we

3That is, assume our hypothesis that terrorists will surely buy a set of 10 items in common at some time during the year. We don’t want to address the matter of whether or not terrorists would necessarily do so.
have classified documents to determine they are about baseball, it is not hard
to notice that words such as these appear unusually frequently. However, until
we have made the classification, it is not possible to identify these words as characteristic.

Thus, classification often starts by looking at documents, and finding the
significant words in those documents. Our first guess might be that the words
appearing most frequently in a document are the most significant. However,
that intuition is exactly opposite of the truth. The most frequent words will
most surely be the common words such as “the” or “and,” which help build
ideas but do not carry any significance themselves. In fact, the several hundred
most common words in English (called stop words) are often removed from
documents before any attempt to classify them.

In fact, the indicators of the topic are relatively rare words. However, not
all rare words are equally useful as indicators. There are certain words, for
example “notwithstanding” or “albeit,” that appear rarely in a collection of
documents, yet do not tell us anything useful. On the other hand, a word like
“chukker” is probably equally rare, but tips us off that the document is about
the sport of polo. The difference between rare words that tell us something and
those that do not has to do with the concentration of the useful words in just a
few documents. That is, the presence of a word like “albeit” in a document does
not make it terribly more likely that it will appear multiple times. However,
if an article mentions “chukker” once, it is likely to tell us what happened in
the “first chukker,” then the “second chukker,” and so on. That is, the word is
likely to be repeated if it appears at all.

The formal measure of how concentrated into relatively few documents are
the occurrences of a given word is called TF.IDF (Term Frequency times Inverse
Document Frequency). It is normally computed as follows. Suppose we
have a collection of \( N \) documents. Define \( f_{ij} \) to be the frequency (number of
occurrences) of term (word) \( i \) in document \( j \). Then, define the term frequency
\( TF_{ij} \) to be:

\[
TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}
\]

That is, the term frequency of term \( i \) in document \( j \) is \( f_{ij} \) normalized by dividing
it by the maximum number of occurrences of any term (perhaps excluding stop
words) in the same document. Thus, the most frequent term in document \( j \)
gets a TF of 1, and other terms get fractions as their term frequency for this
document.

The IDF for a term is defined as follows. Suppose term \( i \) appears in \( n_i \)
of the \( N \) documents in the collection. Then \( IDF_i = \log_2(N/n_i) \). The TF.IDF
score for term \( i \) in document \( j \) is then defined to be \( TF_{ij} \times IDF_i \). The terms
with the highest TF.IDF score are often the terms that best characterize the
topic of the document.

**Example 1.3:** Suppose our repository consists of \( 2^{20} = 1,048,576 \) documents.
Suppose word \( w \) appears in \( 2^{10} = 1024 \) of these documents. Then \( IDF_w = \)
log₂(2²₀ / 2¹⁰) = log₂(2¹⁰) = 10. Consider a document \( j \) in which \( w \) appears 20 times, and that is the maximum number of times in which any word appears (perhaps after eliminating stop words). Then \( TF_{wj} = 1 \), and the TF.IDF score for \( w \) in document \( j \) is 10.

Suppose that in document \( k \), word \( w \) appears once, while the maximum number of occurrences of any word in this document is 20. Then \( TF_{wk} = 1/20 \), and the TF.IDF score for \( w \) in document \( k \) is \( 1/2 \). □

1.3.2 Hash Functions

The reader has probably heard of hash tables, and perhaps used them in Java classes or similar packages. The hash functions that make hash tables feasible are also essential components in a number of data-mining algorithms, where the hash table takes an unfamiliar form. We shall review the basics here.

First, a hash function \( h \) takes a hash-key value as an argument and produces a bucket number as a result. The bucket number is an integer, normally in the range 0 to \( B - 1 \), where \( B \) is the number of buckets. Hash-keys can be of any type. There is an intuitive property of hash functions that they “randomize” hash-keys. To be precise, if hash-keys are drawn randomly from a reasonable population of possible hash-keys, then \( h \) will send approximately equal numbers of hash-keys to each of the \( B \) buckets. It would be impossible to do so if, for example, the population of possible hash-keys were smaller than \( B \). Such a population would not be “reasonable.” However, there can be more subtle reasons why a hash function fails to achieve an approximately uniform distribution into buckets.

**Example 1.4:** Suppose hash-keys are positive integers. A common and simple hash function is to pick \( h(x) = x \mod B \), that is, the remainder when \( x \) is divided by \( B \). That choice works fine if our population of hash-keys is all positive integers. \( 1/B \)th of the integers will be assigned to each of the buckets. However, suppose our population is the even integers, and \( B = 10 \). Then only buckets 0, 2, 4, 6, and 8 can be the value of \( h(x) \), and the hash function is distinctly nonrandom in its behavior. On the other hand, if we picked \( B = 11 \), then we would find that \( 1/11 \)th of the even integers get sent to each of the 11 buckets, so the hash function would work very well. □

The generalization of Example 1.4 is that when hash-keys are integers, choosing \( B \) to have any common factor with all (or even most of) the possible hash-keys will result in nonrandom distribution into buckets. Thus, it is normally preferred that we choose \( B \) to be a prime. That choice reduces the chance of nonrandom behavior, although we still have to consider the possibility that all hash-keys have \( B \) as a factor. Of course there are many other types of hash functions not based on modular arithmetic. We shall not try to summarize the options here, but some sources of information will be mentioned in the bibliographic notes.
What if hash-keys are not integers? In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers. However, there are some simple rules that enable us to convert common types to integers. For example, if hash-keys are strings, convert each character to its ASCII or Unicode equivalent, which can be interpreted as a small integer. Sum the integers before dividing by $B$. As long as $B$ is smaller than the typical sum of character codes for the population of strings, the distribution into buckets will be relatively uniform. If $B$ is larger, then we can partition the characters of a string into groups of several characters each. Treat the concatenation of the codes for the characters of a group as a single integer. Sum the integers associated with all the groups of a string, and divide by $B$ as before. For instance, if $B$ is around a billion, or $2^{30}$, then grouping characters four at a time will give us 32-bit integers. The sum of several of these will distribute fairly evenly into a billion buckets.

For more complex data types, we can extend the idea used for converting strings to integers, recursively.

- For a type that is a record, each of whose components has its own type, recursively convert the value of each component to an integer, using the algorithm appropriate for the type of that component. Sum the integers for the components, and convert the integer sum to buckets by dividing by $B$.

- For a type that is an array, set, or bag of elements of some one type, convert the values of the elements’ type to integers, sum the integers, and divide by $B$.

### 1.3.3 Indexes

An index is a data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects. The most common situation is one where the objects are records, and the index is on one of the fields of that record. Given a value $v$ for that field, the index lets us retrieve all the records with value $v$ in that field. For example, we could have a file of (name, address, phone) triples, and an index on the phone field. Given a phone number, the index allows us to find quickly the record or records with that phone number.

There are many ways to implement indexes, and we shall not attempt to survey the matter here. The bibliographic notes give suggestions for further reading. However, a hash table is one simple way to build an index. The field or fields on which the index is based form the hash-key for a hash function. Records have the hash function applied to value of the hash-key, and the record itself is placed in the bucket whose number is determined by the hash function. The bucket could be a list of records in main-memory, or a disk block, for example.
Then, given a hash-key value, we can hash it, find the bucket, and need to search only that bucket to find the records with that value for the hash-key. If we choose the number of buckets $B$ to be comparable to the number of records in the file, then there will be relatively few records in any bucket, and the search of a bucket takes little time.

Figure 1.2: A hash table used as an index; phone numbers are hashed to buckets, and the entire record is placed in the bucket whose number is the hash value of the phone

**Example 1.5:** Figure 1.2 suggests what a main-memory index of records with name, address, and phone fields might look like. Here, the index is on the phone field, and buckets are linked lists. We show the phone 800-555-1212 hashed to bucket number 17. There is an array of *bucket headers*, whose $i$th element is the head of a linked list for the bucket numbered $i$. We show expanded one of the elements of the linked list. It contains a record with name, address, and phone fields. This record is in fact one with the phone number 800-555-1212. Other records in that bucket may or may not have this phone number. We only know that whatever phone number they have is a phone that hashes to 17.

1.3.4 Secondary Storage

It is important, when dealing with large-scale data, that we have a good understanding of the difference in time taken to perform computations when the data is initially on disk, as opposed to the time needed if the data is initially in main memory. The physical characteristics of disks is another subject on which we could say much, but shall say only a little and leave the interested reader to follow the bibliographic notes.

Disks are organized into *blocks*, which are the minimum units that the operating system uses to move data between main memory and disk. For example,
the Windows operating system uses blocks of 64K bytes (i.e., $2^{16} = 65,536$ bytes to be exact). It takes approximately ten milliseconds to access (move the disk head to the track of the block and wait for the block to rotate under the head) and read a disk block. That delay is at least five orders of magnitude (a factor of $10^5$) slower than the time taken to read a word from main memory, so if all we want to do is access a few bytes, there is an overwhelming benefit to having data in main memory. In fact, if we want to do something simple to every byte of a disk block, e.g., treat the block as a bucket of a hash table and search for a particular value of the hash-key among all the records in that bucket, then the time taken to move the block from disk to main memory will be far larger than the time taken to do the computation.

By organizing our data so that related data is on a single cylinder (the collection of blocks reachable at a fixed radius from the center of the disk, and therefore accessible without moving the disk head), we can read all the blocks on the cylinder into main memory in considerably less than 10 milliseconds per block. You can assume that a disk cannot transfer data to main memory at more than a hundred million bytes per second, no matter how that data is organized. That is not a problem when your dataset is a megabyte. But a dataset of a hundred gigabytes or a terabyte presents problems just accessing it, let alone doing anything useful with it.

1.3.5 The Base of Natural Logarithms

The constant $e = 2.7182818\cdots$ has a number of useful special properties. In particular, $e$ is the limit of $(1 + \frac{1}{x})^x$ as $x$ goes to infinity. The values of this expression for $x = 1, 2, 3, 4$ are approximately 2.25, 2.37, 2.44, so you should find it easy to believe that the limit of this series is around 2.72.

Some algebra lets us obtain approximations to many seemingly complex expressions. Consider $(1 + a)^b$, where $a$ is small. We can rewrite the expression as $(1 + a)^{(1/a)(ab)}$. Then substitute $a = 1/x$ and $1/a = x$, so we have $(1 + \frac{1}{x})^{x(ab)}$, which is

$\left(1 + \frac{1}{x}\right)^{ab}$

Since $a$ is assumed small, $x$ is large, so the subexpression $(1 + \frac{1}{x})^x$ will be close to the limiting value of $e$. We can thus approximate $(1 + a)^b$ as $e^{ab}$.

Similar identities hold when $a$ is negative. That is, the limit as $x$ goes to infinity of $(1 - \frac{1}{x})^x$ is $1/e$. It follows that the approximation $(1 + a)^b = e^{ab}$ holds even when $a$ is a small negative number. Put another way, $(1 - a)^b$ is approximately $e^{-ab}$ when $a$ is small and $b$ is large.

Some other useful approximations follow from the Taylor expansion of $e^x$. That is, $e^x = \sum_{i=0}^{\infty} x^i/i!$, or $e^x = 1 + x + x^2/2! + x^3/6 + x^4/24 + \cdots$. When $x$ is large, the above series converges slowly, although it does converge because $n!$ grows faster than $x^n$ for any constant $x$. However, when $x$ is small, either positive or negative, the series converges rapidly, and only a few terms are necessary to get a good approximation.
Example 1.6: Let $x = 1/2$. Then

$$e^{1/2} = 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{48} + \frac{1}{384} + \cdots$$

or approximately $e^{1/2} = 1.64844$.

Let $x = -1$. Then

$$e^{-1} = 1 - 1 + \frac{1}{2} - \frac{1}{6} + \frac{1}{24} - \frac{1}{120} + \frac{1}{720} - \frac{1}{5040} + \cdots$$

or approximately $e^{-1} = 0.36786$. □

1.3.6 Power Laws

There are many phenomena that relate two variables by a power law, that is, a linear relationship between the logarithms of the variables. Figure 1.3 suggests such a relationship. If $x$ is the horizontal axis and $y$ is the vertical axis, then the relationship is $\log_{10} y = 6 - 2 \log_{10} x$.

![Figure 1.3: A power law with a slope of −2](image)

Example 1.7: We might examine book sales at Amazon.com, and let $x$ represent the rank of books by sales. Then $y$ is the number of sales of the $x$th best-selling book over some period. The implication of the graph of Fig. 1.3 would be that the best-selling book sold 1,000,000 copies, the 10th best-selling book sold 10,000 copies, the 100th best-selling book sold 100 copies, and so on for all ranks between these numbers and beyond. The implication that above
The Matthew Effect

Often, the existence of power laws with values of the exponent higher than 1 are explained by the Matthew effect. In the biblical Book of Matthew, there is a verse about “the rich get richer.” Many phenomena exhibit this behavior, where getting a high value of some property causes that very property to increase. For example, if a Web page has many links in, then people are more likely to find the page and may choose to link to it from one of their pages as well. As another example, if a book is selling well on Amazon, then it is likely to be advertised when customers go to the Amazon site. Some of these people will choose to buy the book as well, thus increasing the sales of this book.

rank 1000 the sales are a fraction of a book is too extreme, and we would in fact expect the line to flatten out for ranks much higher than 1000. □

The general form of a power law relating \( x \) and \( y \) is \( \log y = b + a \log x \). If we raise the base of the logarithm (which doesn’t actually matter), say \( e \), to the values on both sides of this equation, we get \( y = e^b e^{a \log x} = e^b x^a \). Since \( e^b \) is just “some constant,” let us replace it by constant \( c \). Thus, a power law can be written as \( y = cx^a \) for some constants \( a \) and \( c \).

**Example 1.8:** In Fig. 1.3 we see that when \( x = 1 \), \( y = 10^6 \), and when \( x = 1000 \), \( y = 1 \). Making the first substitution, we see \( 10^6 = c \). The second substitution gives us \( 1 = c(1000)^a \). Since we now know \( c = 10^6 \), the second equation gives us \( 1 = 10^6(1000)^a \), from which we see \( a = -2 \). That is, the law expressed by Fig. 1.3 is \( y = 10^6 x^{-2} \), or \( y = 10^6/x^2 \). □

We shall meet in this book many ways that power laws govern phenomena. Here are some examples:

1. **Node Degrees in the Web Graph:** Order all pages by the number of in-links to that page. Let \( x \) be the position of a page in this ordering, and let \( y \) be the number of in-links to the \( x \)th page. Then \( y \) as a function of \( x \) looks very much like Fig. 1.3. The exponent \( a \) is slightly larger than the \(-2 \) shown there; it has been found closer to 2.1.

2. **Sales of Products:** Order products, say books at Amazon.com, by their sales over the past year. Let \( y \) be the number of sales of the \( x \)th most popular book. Again, the function \( y(x) \) will look something like Fig. 1.3. We shall discuss the consequences of this distribution of sales in Section 9.1.2, where we take up the matter of the “long tail.”

3. **Sizes of Web Sites:** Count the number of pages at Web sites, and order sites by the number of their pages. Let \( y \) be the number of pages at the \( x \)th site. Again, the function \( y(x) \) follows a power law.
4. Zipf’s Law: This power law originally referred to the frequency of words in a collection of documents. If you order words by frequency, and let \( y \) be the number of times the \( x \)th word in the order appears, then you get a power law, although with a much shallower slope than that of Fig. 1.3. Zipf’s observation was that \( y = cx^{-1/2} \). Interestingly, a number of other kinds of data follow this particular power law. For example, if we order states in the US by population and let \( y \) be the population of the \( x \)th most populous state, then \( x \) and \( y \) obey Zipf’s law approximately.

1.3.7 Exercises for Section 1.3

Exercise 1.3.1: Suppose there is a repository of ten million documents. What (to the nearest integer) is the IDF for a word that appears in (a) 40 documents (b) 10,000 documents?

Exercise 1.3.2: Suppose there is a repository of ten million documents, and word \( w \) appears in 320 of them. In a particular document \( d \), the maximum number of occurrences of a word is 15. Approximately what is the TF.IDF score for \( w \) if that word appears (a) once (b) five times?

Exercise 1.3.3: Suppose hash-keys are drawn from the population of all non-negative integers that are multiples of some constant \( c \), and hash function \( h(x) \) is \( x \mod 15 \). For what values of \( c \) will \( h \) be a suitable hash function, i.e., a large random choice of hash-keys will be divided roughly equally into buckets?

Exercise 1.3.4: In terms of \( e \), give approximations to

(a) \((1.01)^{500}\) (b) \((1.05)^{1000}\) (c) \((0.9)^{40}\)

Exercise 1.3.5: Use the Taylor expansion of \( e^x \) to compute, to three decimal places: (a) \( e^{1/10} \) (b) \( e^{-1/10} \) (c) \( e^2 \).

1.4 Outline of the Book

This section gives brief summaries of the remaining chapters of the book.

Chapter 2 is not about data mining per se. Rather, it introduces us to the MapReduce methodology for exploiting parallelism in computing clouds (racks of interconnected processors). There is reason to believe that cloud computing, and MapReduce in particular, will become the normal way to compute when analysis of very large amounts of data is involved. A pervasive issue in later chapters will be the exploitation of the MapReduce methodology to implement the algorithms we cover.

Chapter 3 is about finding similar items. Our starting point is that items can be represented by sets of elements, and similar sets are those that have a large fraction of their elements in common. The key techniques of minhashing and locality-sensitive hashing are explained. These techniques have numerous
applications and often give surprisingly efficient solutions to problems that appear impossible for massive data sets.

In Chapter 4, we consider data in the form of a stream. The difference between a stream and a database is that the data in a stream is lost if you do not do something about it immediately. Important examples of streams are the streams of search queries at a search engine or clicks at a popular Web site. In this chapter, we see several of the surprising applications of hashing that make management of stream data feasible.

Chapter 5 is devoted to a single application: the computation of PageRank. This computation is the idea that made Google stand out from other search engines, and it is still an essential part of how search engines know what pages the user is likely to want to see. Extensions of PageRank are also essential in the fight against spam (euphemistically called “search engine optimization”), and we shall examine the latest extensions of the idea for the purpose of combating spam.

Then, Chapter 6 introduces the market-basket model of data, and its canonical problems of association rules and finding frequent itemsets. In the market-basket model, data consists of a large collection of baskets, each of which contains a small set of items. We give a sequence of algorithms capable of finding all frequent pairs of items, that is pairs of items that appear together in many baskets. Another sequence of algorithms are useful for finding most of the frequent itemsets larger than pairs, with high efficiency.

Chapter 7 examines the problem of clustering. We assume a set of items with a distance measure defining how close or far one item is from another. The goal is to examine a large amount of data and partition it into subsets (clusters), each cluster consisting of items that are all close to one another, yet far from items in the other clusters.

Chapter 8 is devoted to on-line advertising and the computational problems it engenders. We introduce the notion of an on-line algorithm – one where a good response must be given immediately, rather than waiting until we have seen the entire dataset. The idea of competitive ratio is another important concept covered in this chapter; it is the ratio of the guaranteed performance of an on-line algorithm compared with the performance of the optimal algorithm that is allowed to see all the data before making any decisions. These ideas are used to give good algorithms that match bids by advertisers for the right to display their ad in response to a query against the search queries arriving at a search engine.

Chapter 9 is devoted to recommendation systems. Many Web applications involve advising users on what they might like. The Netflix challenge is one example, where it is desired to predict what movies a user would like, or Amazon’s problem of pitching a product to a customer based on information about what they might be interested in buying. There are two basic approaches to recommendation. We can characterize items by features, e.g., the stars of a movie, and recommend items with the same features as those the user is known to like. Or, we can look at other users with preferences similar to that of the
1.5 Summary of Chapter 1

User in question, and see what they liked (a technique known as collaborative filtering).

In Chapter 10, we study social networks and algorithms for their analysis. The canonical example of a social network is the graph of Facebook friends, where the nodes are people, and edges connect two people if they are friends. Directed graphs, such as followers on Twitter, can also be viewed as social networks. A common example of a problem to be addressed is identifying “communities,” that is, small sets of nodes with an unusually large number of edges among them. Other questions about social networks are general questions about graphs, such as computing the transitive closure or diameter of a graph, but are made more difficult by the size of typical networks.

Chapter 11 looks at dimensionality reduction. We are given a very large matrix, typically sparse. Think of the matrix as representing a relationship between two kinds of entities, e.g., ratings of movies by viewers. Intuitively, there are a small number of concepts, many fewer concepts than there are movies or viewers, that explain why certain viewers like certain movies. We offer several algorithms that simplify matrices by decomposing them into a product of matrices that are much smaller in one of the two dimensions. One matrix relates entities of one kind to the small number of concepts and another relates the concepts to the other kind of entity. If done correctly, the product of the smaller matrices will be very close to the original matrix.

Finally, Chapter 12 discusses algorithms for machine learning from very large datasets. Techniques covered include perceptrons, support-vector machines, finding models by gradient descent, nearest-neighbor models, and decision trees.

1.5 Summary of Chapter 1

✦ Data Mining: This term refers to the process of extracting useful models of data. Sometimes, a model can be a summary of the data, or it can be the set of most extreme features of the data.

✦ Bonferroni’s Principle: If we are willing to view as an interesting feature of data something of which many instances can be expected to exist in random data, then we cannot rely on such features being significant. This observation limits our ability to mine data for features that are not sufficiently rare in practice.

✦ TF.IDF: The measure called TF.IDF lets us identify words in a collection of documents that are useful for determining the topic of each document. A word has high TF.IDF score in a document if it appears in relatively few documents, but appears in this one, and when it appears in a document it tends to appear many times.

✦ Hash Functions: A hash function maps hash-keys of some data type to integer bucket numbers. A good hash function distributes the possible
hash-key values approximately evenly among buckets. Any data type can be the domain of a hash function.

✧ **Indexes**: An index is a data structure that allows us to store and retrieve data records efficiently, given the value in one or more of the fields of the record. Hashing is one way to build an index.

✧ **Storage on Disk**: When data must be stored on disk (secondary memory), it takes very much more time to access a desired data item than if the same data were stored in main memory. When data is large, it is important that algorithms strive to keep needed data in main memory.

✧ **Power Laws**: Many phenomena obey a law that can be expressed as \( y = cx^a \) for some power \( a \), often around \(-2\). Such phenomena include the sales of the \(x\)th most popular book, or the number of in-links to the \(x\)th most popular page.

### 1.6 References for Chapter 1

[7] is a clear introduction to the basics of data mining. [2] covers data mining principally from the point of view of machine learning and statistics.

For construction of hash functions and hash tables, see [4]. Details of the TF.IDF measure and other matters regarding document processing can be found in [5]. See [3] for more on managing indexes, hash tables, and data on disk.

Power laws pertaining to the Web were explored by [1]. The Matthew effect was first observed in [6].
