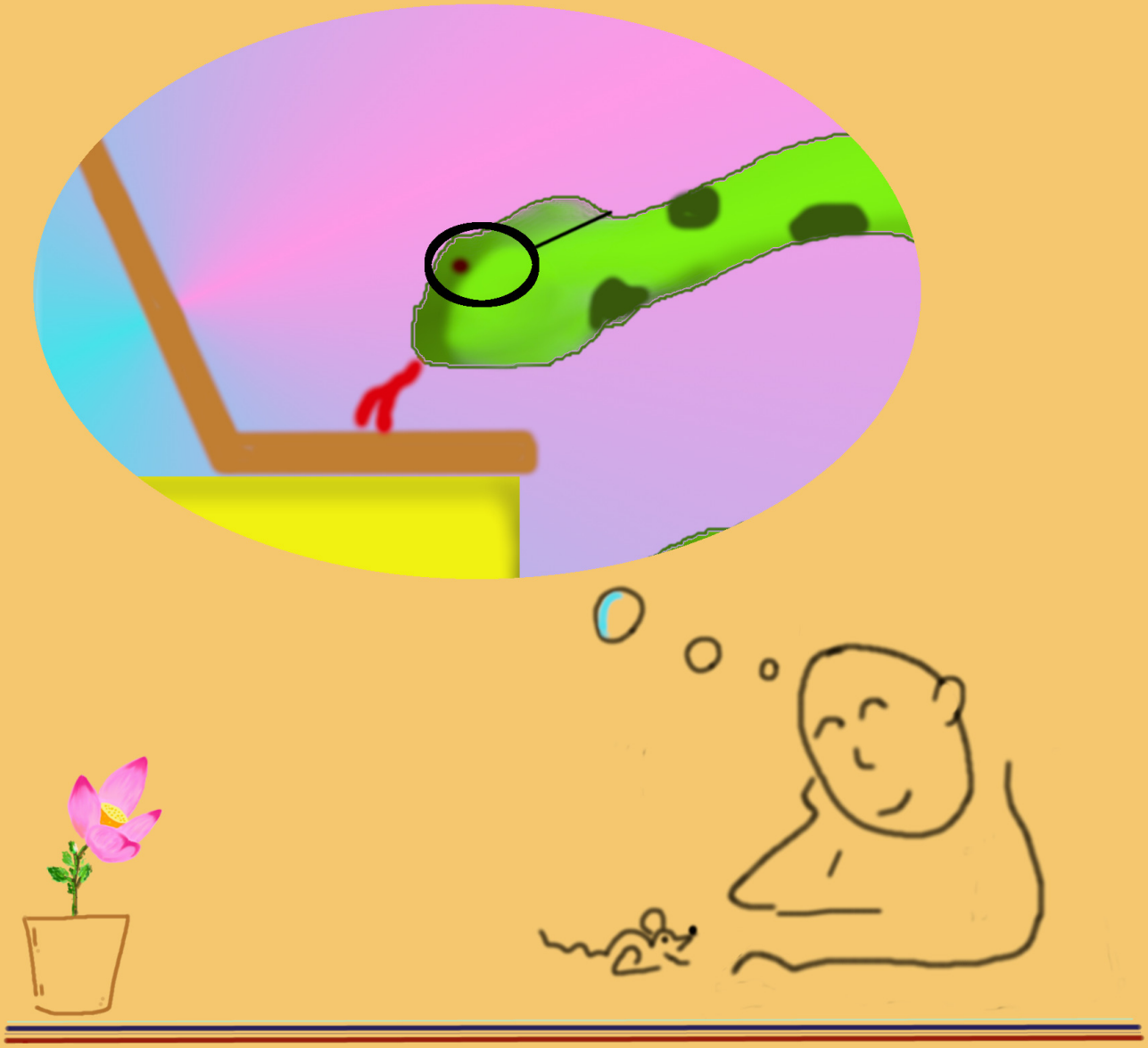


คิดไพธอน วิธีคิดแบบวิศวกรคอมพิวเตอร์

คิดไพธอน

วิธีคิดแบบวิศวกรคอมพิวเตอร์

อัลเลน ดาวนี่

แปลโดย

ธัชพงศ์ กตัญญูกุล

จิระเดช พลสวัสดิ์

กรชวัล ชายผา

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from `http://www.thinkpython2.com`

คำนำ

ประวัติแปลก ๆ ของหนังสือเล่มนี้

เดือนมกราคม ปีค.ศ. 1999 ตอนนั้น ผมกำลังเตรียมสอนวิชาการเขียนโปรแกรมเบื้องต้น ที่เป็นภาษาจาวา. ผมสอนวิชานี้มาแล้วสามครั้ง จนผมเองเริ่มกลุ้มใจ อัตราการตกสูงมาก และแม้กับนักศึกษาที่ผ่าน ระดับของความสำเร็จโดยรวมก็ยังต่ำมาก.

ปัญหาหนึ่งที่ผมเห็นก็คือ หนังสือ. แต่ละเล่มหนามาก มีรายละเอียดภาษาจาวาที่เกินความจำเป็นเยอะมาก แต่มีการสอนเกี่ยวกับการเขียนโปรแกรมน้อยเกินไป. และหนังสือพวกนี้ ทั้งหมดก็มีปัญหา*ประตูลับ* เหมือนกันหมดเลย คือ มันจะเริ่มจากง่าย ๆ และค่อย ๆ ดำเนินไปช้า ๆ จนประมาณบทที่ 5 อยู่ดี ๆ เหมือนพื้นร่วงไปเลย ทุกอย่างเปลี่ยน. เรื่องใหม่ ๆ มาเยอะมาก เร็วมาก จนผมต้องใช้เวลาที่เหลือในเทอม มาแก้ มาเก็บซากหักพังต่าง ๆ.

สองสัปดาห์ก่อนจะเริ่มสอนเทอมใหม่ ผมเลยตัดสินใจว่าจะเขียนหนังสือของตัวเอง. เป้าหมายคือ

- ทำให้สั้นกระชับ. ให้นักศึกษาอ่านหนังสือ 10 หน้า ดีกว่าให้อ่าน 50 หน้า.
- ระวังเรื่องคำศัพท์. ผมพยายามจะลดการใช้*ศัพท์เฉพาะ* และจะนิยามก่อนที่จะใช้ทุกครั้ง.
- ดำเนินเรื่องค่อยเป็นค่อยไป. หลีกเลี่ยง*ประตูลับ* ผมแตกเรื่องที่ยาก ๆ ออกเป็นขั้นเล็ก ๆ หลาย ๆ ขั้น.
- เน้นที่การเขียนโปรแกรม ไม่ใช่ภาษาโปรแกรมคอมพิวเตอร์. นั่นรวมถึง เน้นการใช้คำสั่งที่จำเป็นจำนวนน้อย ๆ และตัดพวกคำสั่งจำนวนมาก ที่ไม่ค่อยจำเป็นออกไป.

ผมต้องการชื่อหนังสือ และก็ลังเล ๆ ผมก็เลือก*วิธีคิดแบบวิศวกรคอมพิวเตอร์*.

เวอร์ชั่นแรกของหนังสือค่อนข้างหยาบ แต่มันก็ได้ผล. นักศึกษาอ่านหนังสือ และก็เข้าใจเนื้อหาพอที่ผมใช้เวลาไปสอนเนื้อหาที่ยาก ๆ และน่าสนใจได้ และ (สำคัญที่สุด) ผมได้มีเวลาให้นักศึกษาได้ฝึกปฏิบัติ.

ผมออกหนังสือเล่มนี้ภายใต้*ลิขสิทธิ์เอกสารอิสระจีเอนยู* (*GNU Free Documentation License*) ที่อนุญาตให้ผู้ใช้ สำเนา แก้ไข หรือแจกจ่าย หนังสือได้.

เรื่องที่เกิดต่อมา เป็นสิ่งที่เจ๋งมาก. เจฟ เอลค์เนอร์ ครูมัธยมในรัฐเวอร์จิเนีย ใช้หนังสือของผม และแปลมันไปเป็นภาษาไพธอน. เขาส่งสำเนาการแปลของเขาให้ผม ผมก็เลยได้ประสบการณ์ที่ประหลาดของการเรียนไพธอน จากการอ่านหนังสือของผมเอง. ในฐานของ*สำนักพิมพ์กรีนที* ผมก็เผยแพร่หนังสือไพธอนเวอร์ชั่นแรกในปี ค.ศ. 2001.

ปี ค.ศ. 2003 ผมเริ่มสอนที่*วิทยาลัยโอลิน* และผมก็ได้สอนไพธอนเป็นครั้งแรก. มันต่างกับจาวาชัดมาก. นักศึกษามีปัญหาลดลง เรียนได้มากขึ้น ทำโปรเจคที่น่าสนใจได้มากขึ้น และโดยส่วนใหญ่ ก็สนุกขึ้นมาก.

ตั้งแต่นั้นมา ผมก็พัฒนาหนังสือมาเรื่อย ๆ แก้ข้อผิด ปรับปรุงตัวอย่าง เพิ่มเนื้อหา โดยเฉพาะแบบฝึกหัดต่าง ๆ. ผลลัพธ์ก็คือ หนังสือเล่มนี้ ที่ปัจจุบันชื่ออหังการน้อยลง เป็น*คิดไพธอน*. การเปลี่ยนแปลงส่วนหนึ่ง ได้แก่

- ผมเพิ่มหัวข้อการดีบัก ที่ท้ายบทแต่ละบท. หัวข้อการดีบักเหล่านี้ สอนวิธีทั่ว ๆ ไปในการหาและหลีกเลี่ยงข้อผิดพลาดในการเขียนโปรแกรม รวมถึงเตือนเกี่ยวกับปัญหาของไพธอนที่อาจเกิดขึ้นได้.
- ผมเพิ่มแบบฝึกหัด ที่มีตั้งแต่ ทดสอบความเข้าใจเล็ก ๆ น้อย ๆ ไปจนถึง โปรเจคใหญ่ ๆ. แบบฝึกหัดเกือบทั้งหมด จะมีลิงค์ไปที่เฉลยที่ผมเตรียมไว้.
- ผมเพิ่มกรณีศึกษาต่าง ๆ ซึ่งเป็นตัวอย่างยาว ๆ กับแบบฝึกหัด เฉลย และบทอภิปราย.
- ผมขยายการอภิปรายแผนการพัฒนาโปรแกรม และ*ดีไซน์แพตเทิร์น*พื้นฐาน
- ผมเพิ่มภาคผนวก เรื่องการดีบัก และการวิเคราะห์อัลกอริธึม

ฉบับพิมพ์ที่สองของ*คิดไพธอน*มีการปรับปรุง ดังนี้

- หนังสือ และโปรแกรมที่เกี่ยวข้องทั้งหมด ปรับเปลี่ยนเป็น*ไพธอนสาม* (*Python 3*)
- ผมเพิ่มหัวข้อ พร้อมรายละเอียดในเว็บ เพื่อช่วยให้ผู้เรียนใหม่ได้ลองรันไพธอนใน*เบราว์เซอร์* (*browser*) ดังนั้นผู้เรียนใหม่ยังไม่ต้องยุ่งกับการติดตั้งไพธอน จนกว่าจะอยากทำ.
- บท 4.1 ผมเปลี่ยนจากโมดูลกราฟฟิก `Swampy` ที่เขียนเอง ไปเป็นโมดูลมาตราฐานของไพธอน `turtle` ที่ติดตั้งง่ายกว่า แล้วก็มีความสามารถมากกว่า.
- ผมเพิ่มบทใหม่ “ของดี ๆ” ที่แนะนำความสามารถใหม่ ๆ ของไพธอน ที่อาจจะไม่ได้จำเป็น แต่มีประโยชน์.

ผมหวังว่า ผู้อ่าน จะสนุกที่เรียนรู้จากหนังสือเล่มนี้ และว่ามันจะได้ช่วยผู้อ่านเรียนเขียนโปรแกรม และฝึกที่จะคิดแบบวิศวกรคอมพิวเตอร์ (อย่างน้อยก็สักนิดหนึ่ง).

อัลเลน บี ดาวนี่ (Allen B. Downey)

วิทยาลัยโอลิน (Olin College)

กิตติกรรมประกาศ

ขอบคุณมาก ๆ กับเจฟเอลค์เนอร์ ผู้ที่แปลหนังสือจาวาของผมมาเป็นไพธอน ที่ทำให้โปรเจคนี้เริ่ม และทำให้ผมได้รู้จักสิ่งที่ได้กลายมาเป็นภาษาโปรดของผม

ขอบคุณเช่นกันกับคริสเมเยอส์ ที่ช่วยหลาย ๆ หัวข้อของ *วิธีคิดแบบวิศวกรคอมพิวเตอร์*.

ขอบคุณ*มูลนิธิซอฟต์แวร์อิสระ* ที่ได้ทำ*ลิขสิทธิ์เอกสารอิสระจีเอนยู*ขึ้น ซึ่งได้ช่วยให้เกิดความร่วมมือกันกับเจฟและคริสได้ รวมถึง*ครีเอทิฟคอมมอนส์* (*Creative Commons*) ที่เป็นลิขสิทธิ์ที่ผมใช้อยู่ตอนนี้.

ขอบคุณเหล่าบรรณาธิการที่*ลูลู* ที่ช่วยทำ*วิธีคิดแบบวิศวกรคอมพิวเตอร์*.

ขอบคุณเหล่าบรรณาธิการที่*โอไรลี่มีเดีย* ที่ช่วยทำ *คิดไพธอน*.

ขอบคุณนักเรียนนักศึกษาทุกคน ที่ใช้เวอร์ชั่นแรก ๆ ของหนังสือเล่มนี้ และก็ผู้ช่วยเหลือทุก ๆ ท่าน (ดังรายนามข้างล่าง) ที่ช่วยส่งการแก้ไขจุดผิด และคำแนะนำต่าง ๆ.

รายนามผู้ช่วยเหลือ

[รายนามนี้เป็นเสมือนการสื่อสารระหว่างผู้เขียน, อัลเลน ดาวนี่, กับผู้ช่วยเหลือ คณะผู้แปลเห็นควรปล่อยไว้ในรูปของภาษาต้นฉบับ]

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 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 `spot`ted 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 `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise 11.5.
- Leah Engelbert-Fenton pointed out that I used `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 14.3.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *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 `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer 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.
- Xavier Van Aubel made several useful corrections in the second edition.
- 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, 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.

คำนำจากผู้แปล

เป็นเรื่องที่แปลกมาก ที่คณะผู้แปลเองก็มีประสบการณ์คล้าย ๆ กับผู้เขียน นั่นคือ จากการสอนการเขียนโปรแกรมคอมพิวเตอร์พื้นฐาน ซึ่งเป็นวิชาเขียนโปรแกรมวิชาแรก ของนักศึกษาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยขอนแก่น คณะผู้แปลพบว่า การสอนการเขียนโปรแกรมให้กับนักศึกษา เป็นหนึ่งในงานที่ท้าทายที่สุด โดยเฉพาะสำหรับนักศึกษาที่ไม่เคยเขียนโปรแกรมมาก่อน. นอกจากนั้น คณะผู้แปลเองก็มองว่า สิ่งที่สำคัญที่สุดสำหรับวิชาการเขียนโปรแกรมนั้น น่าจะเป็น*ทักษะการแก้ปัญหา* ซึ่งก็ตรงกับความเห็นของผู้เขียนเช่นกัน. ดังนั้น เมื่อคณะผู้แปลได้พบหนังสือเล่มนี้ (ฉบับภาษาอังกฤษ) ก็รู้สึกว่า หนังสือเล่มนี้เขียนขึ้นมา ด้วยปรัชญาที่ตรงกับใจของคณะผู้แปล (กระชับ เรียบง่าย ค่อย ๆ ดำเนินเนื้อหาทีละขั้นเล็ก ๆ และที่สำคัญคือ เน้นที่การเขียนโปรแกรม มากกว่าตัวภาษา และมีกลุ่มเป้าหมายหลักคือ นักศึกษาที่เพิ่งเรียนเขียนโปรแกรมเป็นครั้งแรก) คณะผู้แปลจึงสนใจที่จะแปลมาเป็นภาษาไทย เพื่อให้นักศึกษา รวมถึงผู้สนใจทั่วไป ได้มีตำราที่เหมาะสำหรับผู้เริ่มเรียนเขียนโปรแกรมใหม่ และเข้าถึงได้ง่ายขึ้นจากการที่เป็นภาษาแม่ที่คุ้นเคย คณะผู้แปล รู้สึกขอบคุณที่ผู้เขียน อัลเลน ดาวนี่ ที่ได้เผยแพร่หนังสือเล่มนี้ภายใต้ลิขสิทธิ์ที่ช่วยให้ผู้อ่านเข้าถึงได้ง่าย และคณะผู้แปลจึงได้ขออนุญาตผู้เขียน เพื่อทำการแปลหนังสือเล่มนี้ออกมาเป็นตำราภาษาไทย

อย่างไรก็ตาม ในการแปลอาจมีการปรับคำศัพท์บางอย่าง เพื่อให้กระชับ ตรงตามความหมาย และเข้ากับบริบทของไทย รวมถึงสะท้อนจุดประสงค์และกลุ่มเป้าหมายหลักของผู้แปล การปรับคำศัพท์นี้ คณะผู้แปลได้กระทำอย่างระมัดระวัง เพื่อไม่ให้กระทบกับจุดประสงค์ และเนื้อหาหลัก ๆ ของตำรา

นอกจากนั้น เนื่องจากตำรานี้เป็นการแปลจากภาษาอังกฤษที่สื่อสารด้วยทำนองที่ค่อนข้างเป็นกันเอง และจุดประสงค์หลักคือ มุ่งเน้นให้ผู้อ่านเข้าถึงได้ง่าย ดังนั้น คณะผู้แปลจึงขออนุญาตแปล โดยใช้ภาษาไทยในรูปแบบที่เป็นทางการน้อยกว่าตำราวิชาการทั่วไป คณะผู้แปลก็ขออภัยมา ณ ที่นี้ด้วย

สารบัญ

1. วิถีของโปรแกรม

จุดประสงค์ของตำรานี้ คือ สอนให้เราคิดอย่างวิศวกรคอมพิวเตอร์ วิธีคิดนี้ รวมเอาส่วนของแง่มุมที่ดีที่สุดของคณิตศาสตร์ วิศวกรรม และวิทยาศาสตร์ธรรมชาติ ในลักษณะเดียวกับนักคณิตศาสตร์ วิศวกรคอมพิวเตอร์ใช้ ภาษารูปนัย (formal language) เพื่อสื่อถึงความคิด (โดยเฉพาะ การคำนวณ การประมวลข้อมูล) ในลักษณะเดียวกับวิศวกรอื่น ๆ วิศวกรคอมพิวเตอร์ออกแบบสิ่งต่าง ๆ ประกอบส่วนประกอบต่าง ๆ เข้าด้วยกันเป็นระบบที่ใหญ่ขึ้น และประเมินข้อดี-ข้อเสีย (tradeoffs) ของทางเลือกต่าง ๆ ในลักษณะเดียวกับนักวิทยาศาสตร์ วิศวกรคอมพิวเตอร์สังเกตพฤติกรรมของระบบที่ซับซ้อน ตั้งสมมติฐาน และทดสอบสมมติฐาน

ทักษะที่สำคัญที่สุดของวิศวกรคอมพิวเตอร์ คือ **ทักษะการแก้ปัญหา** (**problem solving**) ทักษะการแก้ปัญหา หมายถึง ความสามารถในการอ่าน-ตีความปัญหา (formulate problems) คิดอย่างสร้างสรรค์เกี่ยวกับวิธีการแก้ปัญหา (think creatively about solutions) และ การนำเสนอวิธีแก้ปัญหาได้อย่างกระจ่างและตรงประเด็น (express a solution clearly and accurately) นั่นเท่ากับว่า กระบวนการเรียนการเขียนโปรแกรม เป็น โอกาสที่ดีในการฝึกฝนทักษะการแก้ปัญหา นั่นเป็นเหตุผลว่า บทนี้เรียกว่า “วิถีของโปรแกรม” (the way of the program)

หากมองในระดับผิวเผิน เราจะได้เรียนรู้วิธีการเขียนโปรแกรม ซึ่งในตัววิธีการเอง ก็เป็นทักษะที่มีประโยชน์. แต่เมื่อมองลึกลงไป เราจะใช้ การเขียนโปรแกรมเป็นวิธีหนึ่ง เพื่อไปสู่จุดหมายที่แท้จริง ในขณะที่เราเรียนเขียนโปรแกรมไปเรื่อย ๆ จุดหมายที่แท้จริงจะค่อย ๆ ชัดเจนขึ้น

1.1. อะไรคือโปรแกรม

โปรแกรม เป็นลำดับของคำสั่งต่าง ๆ ที่บอกคอมพิวเตอร์ว่าจะคำนวณ จะประมวลข้อมูลอย่างไร การคำนวณ อาจจะเป็นแบบคณิตศาสตร์ เช่น แก้ปัญหาระบบสมการ หรือ หารากของพหุนาม แต่ก็อาจจะ

หมายรวมถึง การคำนวณเชิงสัญลักษณ์ เช่น การค้นหา และแทนข้อความ ในเอกสาร หรืออาจจะหมายถึง การทำงานเกี่ยวกับภาพ เช่น การประมวลผลภาพ หรือการเปิดดูวิดีโอ

รายละเอียดของโปรแกรม แตกต่างกันไปบ้างตามแต่ละภาษา แต่คำสั่งพื้นฐานต่าง ๆ ก็มีอยู่ในทุกๆภาษา ได้แก่

อินพุต (input): รับข้อมูลเข้าจากแป้นพิมพ์ ไฟล์ เครือข่าย หรือ อุปกรณ์อื่น ๆ

เอาต์พุต (output): แสดงข้อมูลออกหน้าจอ บันทึกข้อมูลลงไฟล์ ส่งข้อมูลผ่านเครือข่าย หรือ ส่งข้อมูลไปอุปกรณ์อื่น ๆ

คณิตศาสตร์ (math): ทำการคิดคณิตศาสตร์พื้นฐาน เช่น การบวก และการคูณ

การทำเงื่อนไข (conditional execution): ตรวจสอบเงื่อนไขที่ระบุ และเลือกทำคำสั่งที่เหมาะสม

การทำซ้ำ (repetition): ทำคำสั่งซ้ำ ๆ โดยแต่ละรอบของการทำซ้ำอาจมีความต่างบ้าง

จริง ๆ แล้ว นี่ก็เกือบทั้งหมดของคำสั่งต่าง ๆ คอมพิวเตอร์แล้ว ทุก ๆ โปรแกรมที่เราเคยใช้ ไม่ว่าจะซับซ้อนแค่ไหน เขียนขึ้นจากคำสั่งต่าง ๆ ข้างต้นนี้ ดังนั้น เราอาจจะมองว่า การเขียนโปรแกรม เป็น กระบวนการที่แตกงานใหญ่ที่ซับซ้อน ออกเป็นงานย่อย ๆ จนกระทั่งงานย่อยๆเล็กลง จนสามารถที่จะทำให้สำเร็จได้ด้วยคำสั่งพื้นฐานต่าง ๆ ข้างต้นนี้

1.2. การรันโปรแกรมไพธอน

เรื่องหนึ่งที่ยากที่สุดของการเริ่มใช้ไพธอน (Python) คือ เราอาจจะต้องติดตั้งไพธอน รวมถึงซอฟต์แวร์ที่เกี่ยวข้อง ถ้าคุณคุ้นเคยกับระบบปฏิบัติการ โดยเฉพาะถ้าคุณสะดวกที่จะใช้คอมมานด์ไลน์ (command-line interface) คุณไม่น่ามีปัญหาในการติดตั้งไพธอน แต่สำหรับมือใหม่ ก็อาจจะลำบากบ้างที่จะต้องเรียนการจัดการระบบไปพร้อม ๆ กับการเรียนเขียนโปรแกรม

เพื่อไม่ให้เกิดปัญหาข้างต้น แนะนำว่าให้คุณเริ่มรันโปรแกรมไพธอนในเบราว์เซอร์ดูก่อน จากนั้น ถ้าพอคุ้นเคยกับไพธอนบ้างแล้ว ค่อยติดตั้งไพธอนลงเครื่อง มีเว็บต่าง ๆ มากมายที่สามารถรันไพธอนได้ ถ้าคุณมีเว็บที่ชอบอยู่แล้ว ก็ใช้ได้เลย แต่ถ้าไม่มี ก็แนะนำ `PythonAnywhere` โดยมีคำแนะนำสำหรับมือใหม่ที่ `http://tinyurl.com/thinkpython2e`

ไพธอนมีสองเวอร์ชั่น คือ ไพธอน-2 และไพธอน-3 ทั้งสองเวอร์ชั่นคล้ายกันมาก ถ้าเรียนเวอร์ชั่นหนึ่ง ก็ไม่ยากที่จะเปลี่ยนเป็นอีกเวอร์ชั่น จริง ๆ แล้ว มีความต่างแค่นิดหน่อยเท่านั้น สำหรับการเขียนโปรแกรมเบื้องต้น ตำราเล่มนี้เขียนสำหรับไพธอน-3 แต่ก็แนะนำข้อสังเกตบางอย่างเกี่ยวกับไพธอน-2

ไพธอน **อินเตอร์พรีเตอร์** (**interpreter**) เป็นโปรแกรมที่อ่านและรันคำสั่งภาษาไพธอน ขึ้นกับ*เอนไวรอนเมนต์*[1] ของเรา บางเอนไวรอนเมนต์ อาจจะเปิดไพธอนอินเตอร์พรีเตอร์ ด้วยการคลิกไอคอน หรือ บางเอนไวรอนเมนต์ อาจจะด้วยการพิมพ์คำสั่ง `python` *ที่หน้ารับคำสั่ง* (*command console*) หลังจากที่ไพธอนอินเตอร์พรีเตอร์เปิดขึ้นมา เราอาจจะเห็น

```
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.
>>>
```

สามบรรทัดแรก บอกข้อมูลเกี่ยวกับตัวอินเตอร์พรีเตอร์ และระบบปฏิบัติการ ดังนั้นข้อมูลนี้อาจจะต่างจากที่คุณได้ แต่คุณควรจะตรวจสอบเวอร์ชั่นดูด้วย เวอร์ชั่นที่ในตัวอย่างนี้เป็น `3.4.0` ตัวเลขแรกบอกว่าเป็นไพธอน-3 ถ้าตัวเลขแรกเป็น 2 แปลว่า คุณกำลังรัน ไพธอน-2 อยู่

บรรทัดสุดท้ายเป็น **ข้อความพร้อมรับ** (**prompt**) ที่เป็นตัวบอกว่า อินเตอร์พรีเตอร์พร้อมที่จะรับคำสั่งจากเรา ถ้าคุณพิมพ์คำสั่งและกด "`Enter`" อินเตอร์พรีเตอร์จะแสดงผลออกมา

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

ตอนนี้ เรารู้แล้วว่า การเปิดไพธอนอินเตอร์พรีเตอร์ และการรันคำสั่ง ทำอย่างไร ตอนนี้เราก็พร้อมแล้วที่จะเริ่มเขียนโปรแกรม

1.3. โปรแกรมแรก

ตามธรรมเนียมแล้ว โปรแกรมแรกที่เราจะเขียนในภาษาใหม่ จะเรียกว่า โปรแกรม "Hello, World!" เพราะว่า มันจะทำแค่แสดงคำว่า "Hello, World!" สำหรับไพธอน หน้าตาของโปรแกรมจะเป็นดังนี้

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

นี่เป็นตัวอย่างของ **คำสั่งพิมพ์** (**print statement**) ที่จริง ๆ แล้ว จะไม่ได้พิมพ์อะไรลงกระดาษ แต่เป็นแค่แสดงข้อความออกหน้าจอ ซึ่งในกรณีนี้ ผลลัพธ์คือ

[1] เอนไวรอนเมนต์ (operating environment) ในที่นี้ หมายถึง โครงสร้างพื้นฐานที่ใช้รันไพธอนอินเตอร์พรีเตอร์ ซึ่งอาจหมายถึง ระบบปฏิบัติการ เช่น ไมโครซอฟต์วินโดว์ ลีนุกซ์ รวมไปถึง *ไอดีอี* (*IDE*, ย่อจาก Integrated development environment) ซึ่งเป็นระบบอำนวยความสะดวก สำหรับการพัฒนา เช่น `Anaconda` (`https://anaconda.org/anaconda/python`) เป็นต้น

```
Hello, World!
```

เครื่องหมายคำพูด (`'...'`) ในโปรแกรม ใช้ระบุจุดเริ่มต้นและจุดจบ ของข้อความที่จะแสดง ผลลัพธ์จะไม่แสดงเครื่องหมายคำพูดนี้ออกมา

วงเล็บใช้บอกว่า `print` เป็นฟังก์ชัน (function). เราจะเรียนเรื่องฟังก์ชันกัน ในบทที่ 3

ในไพธอน-2 คำสั่งพิมพ์จะต่างออกไปเล็กน้อย นั่นคือ จะไม่ใช่ฟังก์ชัน ดังนั้นจึงไม่ต้องการวงเล็บ เช่น

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

จุดต่างนี้จะค่อย ๆ กระจ่างขึ้นทีหลัง

1.4. ตัวดำเนินการพีชคณิต

จาก “Hello, World!” ขั้นต่อไปคือ การคำนวณพีชคณิต (arithmetic) ไพธอนมี **ตัวดำเนินการ** (**operators**) ที่ใช้สัญลักษณ์พิเศษ เพื่อแสดงการคำนวณ เช่น บวกและคูณ

ตัวดำเนินการ `+`, `-`, และ `*` ทำการบวก, ทำการลบ, และทำการคูณ ดังแสดงในตัวอย่าง เช่น

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

ตัวดำเนินการ `/` ทำการหาร เช่น

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

คุณอาจจะสงสัยว่า ทำไมผลลัพธ์ถึงเป็น `42.0` แทนที่จะเป็น `42` เราจะอธิบายเรื่องนี้ในหัวข้อถัดไป

ส่วน ตัวดำเนินการ `**` ทำการยกกำลัง นั่นคือ เช่น

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

บางภาษาคอมพิวเตอร์จะใช้ ^ สำหรับการยกกำลัง แต่ไพธอนใช้ ^ สำหรับ*ตัวดำเนินการตามบิต* (*bitwise operator*) ที่เรียกว่า เอ็กซ์ออร์ (XOR) ถ้าคุณยังไม่คุ้นเคยกับ*ตัวดำเนินการตามบิต* ผลลัพธ์อาจจะดูแปลก เช่น

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

เนื้อหาของตำรานี้ไม่ครอบคลุม*ตัวดำเนินการตามบิต* แต่ถ้าสนใจ ก็สามารถค้นคว้าเพิ่มเติมได้จาก `http://wiki.python.org/moin/BitwiseOperators`.

1.5. ค่าและชนิดข้อมูล

ค่าข้อมูล (**value**) เป็นสิ่งพื้นฐานที่โปรแกรมทำงานด้วย ค่าที่พูดถึง ได้แก่ ตัวอักษร และ ตัวเลข บางค่าที่เราดูกันไปแล้ว เช่น `2`, `42.0`, และ `'Hello, World!'`

ค่าต่าง ๆ เหล่านี้ เป็น**ชนิดข้อมูล** (**types**) ต่าง ๆ กัน เช่น ค่า `2` เป็นชนิด**จำนวนเต็ม** (**integer**) ค่า `42.0` เป็นชนิดเลขทศนิยม หรือ **จำนวนโฟลตติ้งพอยต์** (**floating-point number**) และ `'Hello, World!'` เป็นชนิด**สายอักขระ** (**string**) ที่เรียกเช่นนี้ ก็เพราะตัวอักษรถูกนำมาเรียงร้อยต่อกัน

ถ้าไม่แน่ใจว่าค่าข้อมูลเป็นชนิดไหน เราก็ใช้อินเตอร์พรีเตอร์บอกเราได้ เช่น

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

ตัวอย่างข้างต้น คำว่า “class” ในที่นี้ ถูกใช้ ในแง่ที่หมายถึง ชนิดหมวดหมู่ *ชนิดข้อมูล* ก็คือ หมวดหมู่ของ*ค่าข้อมูล* ผลที่ได้ก็ไม่น่าแปลกใจว่า จำนวนเต็ม เป็นชนิดข้อมูล `int` สายอักขระ เป็นชนิดข้อมูล `str` และเลขทศนิยม (จำนวนโฟลตติ้งพอยต์) เป็นชนิดข้อมูล `float`

แล้วถ้าเป็นค่าแบบ `'2'` และ `'42.0'` จะนับเป็นชนิดข้อมูลแบบไหน ค่าเหล่านี้ดูเหมือนตัวเลข แต่ค่าเหล่านี้อยู่ในอัญประกาศ (quotation marks) เหมือนสายอักขระ

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

มันเป็น สายอักขระ

เวลาที่เราพิมพ์เลขจำนวนเต็มค่าใหญ่ ๆ เราอาจจะอยากที่จะใช้เครื่องหมายจุลภาคคั่นกลุ่มของตัวเลข แบบ **1,000,000** นี่ไม่ใช่รูปแบบ**จำนวนเต็ม**ที่ถูกต้องในไพธอน

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

ไม่เหมือนที่เราคิดเลย. ไพธอนมอง **1,000,000** เป็นลำดับของจำนวนเต็มที่คั่นด้วยจุลภาค เราจะเรียนชนิดของข้อมูลแบบลำดับแบบนี้ทีหลัง (บทที่ 12)

1.6. ภาษารูปนัยและภาษาธรรมชาติ

ภาษาธรรมชาติ เป็นภาษาที่คนใช้พูด เช่น ภาษาไทย ภาษาจีน ภาษาอังกฤษ ภาษาธรรมชาติไม่ได้ถูกออกแบบ โดยนักออกแบบภาษา (แม้ว่าจะมีคนพยายามจะใส่กฎเกณฑ์เข้าไป) ภาษาธรรมชาติพัฒนาและมีวิวัฒนาการตามธรรมชาติที่ผู้คนใช้

ภาษารูปนัย เป็นภาษาที่ถูกออกแบบ โดยนักออกแบบสำหรับจุดประสงค์เฉพาะ ตัวอย่างเช่น สัญกรณ์ (notation) ที่นักคณิตศาสตร์ใช้ ก็เป็นภาษารูปนัยที่ออกแบบมาเฉพาะ สำหรับแสดงถึงความสัมพันธ์ระหว่างตัวเลขและสัญลักษณ์ต่าง ๆ นักเคมีก็ใช้ภาษารูปนัย แสดงโครงสร้างทางเคมีของโมเลกุล และที่สำคัญ

ภาษาคอมพิวเตอร์ เป็นภาษารูปนัยที่ออกแบบมาเพื่อระบุวิธีการประมวลผล

ภาษารูปนัยมักจะมีกฎ**วากยสัมพันธ์** (**syntax**) ที่เข้มงวด ซึ่งกฎวากยสัมพันธ์ ใช้เพื่อควบคุมโครงสร้างของข้อความ (หรือโครงสร้างของคำสั่ง) ตัวอย่างเช่น ในคณิตศาสตร์ ข้อความ $3 + 3 = 6$ มีวากยสัมพันธ์ที่ถูกต้อง แต่ $3+ = 3\$6$ ไม่ถูก ในเคมี H_2O เป็นสูตรที่ถูกต้องตามวากยสัมพันธ์ แต่สูตร ${}_2Zz$ ไม่ถูกต้อง

กฎวากยสัมพันธ์มาในสองรูปแบบ ซึ่งคือเกี่ยวกับ **โทเค็น**ต่าง ๆ (**tokens**) และโครงสร้าง (structure) โทเค็น เป็นส่วนประกอบพื้นฐานของภาษา เช่น คำ (ในภาษาธรรมชาติ) ตัวเลข (ในคณิตศาสตร์) ธาตุทางเคมี (ในเคมี)

หนึ่งในปัญหาของ $3+ = 3\$6$ คือ \$ ไม่ใช่โทเค็นที่ถูกต้องทางคณิตศาสตร์ (อย่างน้อยก็เท่าที่เรารู้) ในทำนองเดียวกัน ${}_2Zz$ ไม่ถูกต้อง เพราะว่า ไม่มีธาตุทางเคมีที่ใช้ตัวย่อ Zz

กฎวากยสัมพันธ์แบบที่สอง จะเกี่ยวกับ วิธีที่รวมโทเค็นเข้าด้วยกัน สมการ $3+ = 3$ ไม่ถูกวากยสัมพันธ์ทางคณิตศาสตร์ เพราะว่า แม้ $+$ และ $=$ จะเป็นโทเค็นที่ถูกต้อง แต่เราไม่สามารถเอาโทเค็นสองตัวนี้มาวางต่อกันแบบนี้ได้ ทำนองเดียวกัน ในสูตรเคมี ตัวห้อยจะต้องมาหลังจากธาตุ ไม่ใช่มาก่อนธาตุ

ตัวอย่าง ข้อความภาษาอังกฤษที่ถูกวากยสัมพันธ์ด้านโครงสร้าง แต่ผิดด้านโทเค็น ได้แก่ “This is @ well-structured Engli$h sentence with invalid t*kens in it.” เปรียบเทียบกับ ข้อความภาษาอังกฤษที่ถูกวากยสัมพันธ์ด้านโทเค็น แต่ผิดด้านโครงสร้าง ได้แก่

This sentence all valid tokens has, but invalid structure with.

ประโยคนี้ถูก$\int$ครงสร้7งภ7ษ7ไทϵ 11ต่ผิดด้าน$\int$ทเค็น ส่วนทั้งหมดโทเค็นถูกนี้ประโยค แต่ไม่โครงสร้างถูก

เวลาที่เราอ่านประโยคภาษาไทย หรือข้อความในภาษารูปนัย เราจะมองหาโครงสร้างของภาษา (แม้ว่า เรามักจะทำด้วยจิตใต้สำนึก สำหรับภาษาธรรมชาติ) กระบวนการนี้ (จัดโทเค็นเข้าโครงสร้าง) จะเรียกว่า **การแจงส่วน** (**parsing**)

แม้ว่า ภาษารูปนัย และภาษาธรรมชาติ จะมีลักษณะหลาย ๆ อย่างคล้าย ๆ กัน ไม่ว่าจะเป็น โทเค็น โครงสร้าง และวากยสัมพันธ์ แต่ก็มีความแตกต่างกันอยู่มาก ได้แก่

ความกำกวม (ambiguity): ภาษาธรรมชาติจะเต็มไปด้วยความกำกวม ที่คนมักจะใช้บริบทหรือข้อมูลอื่น ๆ ช่วย ภาษารูปนัยถูกออกแบบมาให้ปราศจาก (หรือเกือบปราศจาก) ความกำกวมเลย นั่นคือ ข้อความในภาษารูปนัย จะมีความหมายอย่างเดียวแน่นอน โดยไม่เกี่ยวกับบริบท

ความซ้ำซ้อน (redundancy): เพื่อชดเชยความกำกวม และลดการเข้าใจผิด ภาษาธรรมชาติมักใช้ความซ้ำซ้อนเข้ามาช่วย ผลคือ ภาษาธรรมชาติมักจะใช้ถ้อยคำมากเกินไป ภาษารูปนัยไม่ค่อยซ้ำซ้อนและกระชับกว่า

ความตรงตามตัวอักษร (literalness): ภาษาธรรมชาติเต็มไปด้วยสำนวนและการอุปมา ภาษาอังกฤษ มีสำนวน “The penny dropped” ที่ไม่ได้หมายถึง เศษเงิน หรือแม้แต่มีอะไรร่วงหล่น (สำนวนนี้หมายถึง การที่ใครสักคนได้เข้าใจเรื่องราวสักที หลังจากงงมานาน[2]) ภาษาไทย ก็มีสำนวน เช่น “ตีเหล็กเมื่อแดง กินแกงเมื่อร้อน” ซึ่งไม่ได้หมายถึง ทั้งเหล็ก หรือว่าแกง แต่หมายถึง การทำสิ่งที่มีประโยชน์ เมื่อยังมีโอกาส ภาษารูปนัยหมายความตรงตามที่เขียน

[2] สำนวน เป็นการอุปมากับ เครื่องขายของอัตโนมัติ ที่เหรียญค้างไม่ยอมทำงาน จนพักใหญ่ ๆ กว่าที่เหรียญจะหลุดเข้าไปในเครื่อง และเครื่องทำงานได้ จากคณะผู้แปล

เพราะว่า เราใช้ภาษาธรรมชาติมาตลอด ทำให้บางครั้ง มันอาจจะยากหน่อยที่จะปรับตัว เพื่อใช้ภาษารูปนัย ความต่างของภาษารูปนัยกับภาษาธรรมชาติ ก็ต่างกันแบบที่ ร้อยกรอง(บทกวี)ต่างจากร้อยแก้ว(ข้อความธรรมดา) แต่ภาษารูปนัยกับภาษาธรรมชาติต่างกันมากกว่า

ร้อยกรอง หรือ บทกวี: คำจะถูกเลือกใช้ จากเสียงของคำพอ ๆ กับจากความหมายของมัน ร้อยกรองทั้งบทจะสร้างผลเชิงอารมณ์ ความกำกวมไม่ใช่แค่มาตามปกติ แต่หลาย ๆ ครั้งเป็นความตั้งใจของผู้แต่ง

ร้อยแก้ว หรือ ข้อความธรรมดา: คำที่ใช้ถูกเลือกตามความหมายของคำมากกว่าเสียงของคำ และโครงสร้างประโยคจะบอกความหมาย ร้อยแก้วจะเข้าใจได้ง่ายกว่าร้อยกรอง แต่ก็ยังกำกวมอยู่

โปรแกรม (Programs): ความหมายของโปรแกรมคอมพิวเตอร์ชัดเจน ไม่กำกวม และตรงตามตัวอักษร และความหมายสามารถเข้าใจได้อย่างครบถ้วน โดยการวิเคราะห์โทเค็นและโครงสร้าง

ภาษารูปนัยจะมีข้อความแน่นกว่าภาษาธรรมชาติ ดังนั้นภาษารูปนัยจึงใช้เวลาอ่านมากกว่าภาษาธรรมชาติ นอกจากนั้น โครงสร้างก็สำคัญ ดังนั้น การอ่านจากบนลงล่าง ซ้ายไปขวา จึงไม่ใช่วิธีที่ดีที่สุดเสมอไป เราจึงควรฝึก*แจงส่วน*โปรแกรมในหัว นั่นคือ ลองระบุโทเค็น ลองตีความโครงสร้าง ในหัว นอกจากนั้นแล้ว รายละเอียดสำคัญมาก ความผิดพลาดเล็ก ๆ น้อย ๆ ในการสะกดคำ หรือการใช้เครื่องหมายวรรคตอน ที่อาจจะไม่มีผลมากในภาษาธรรมชาติ ความผิดพลาดเล็ก ๆ น้อย ๆ ในลักษณะเดียวกัน จะมีผลต่างกันมากในภาษารูปนัย

1.7. การดีบัก

โปรแกรมเมอร์ทำผิดพลาดได้ ด้วยเหตุผลแปลก ๆ ความผิดพลาดในโปรแกรม (programming errors) ถูกเรียกว่า **บัก** (**bugs**) และกระบวนการตรวจหาข้อผิดพลาดนี้ เรียกว่า การตรวจแก้จุดบกพร่อง หรือการ**ดีบัก** (**debugging**)

บางครั้ง การเขียนโปรแกรม โดยเฉพาะการดีบัก อาจจะทำให้อารมณ์เสียได้ บางครั้ง ที่เรามีปัญหากับบัคที่ยาก ๆ เราอาจจะรู้สึกโมโห รู้สึกท้อ หรือรู้สึกอาย

มีหลักฐานว่า คนมักจะปฏิสัมพันธ์กับคอมพิวเตอร์ ราวกับว่าคอมพิวเตอร์เป็นคน เวลาที่คอมพิวเตอร์ทำงานได้ดี เรามักมองคอมพิวเตอร์เหมือนเป็นเพื่อนร่วมทีม แต่เวลาที่คอมพิวเตอร์ดื้อหรือหยาบคาย เราก็มักจะทำกับคอมพิวเตอร์ เหมือนกับที่เราทำกับคนดื้อ หรือคนหยาบคาย (จาก Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*)

เตรียมใจไว้สำหรับปฏิกิริยาแบบนี้ จะช่วยให้เราจัดการกับมันได้ดีขึ้น แนวทางหนึ่งก็คือให้มองคอมพิวเตอร์เป็นเสมือนลูกน้องที่เก่งเฉพาะเรื่อง เช่น ความเร็ว ความแม่นยำ แต่มีจุดอ่อน คือ ไม่มีความรู้สึก และไม่สามารถที่จะเข้าใจภาพรวมของงานได้

งานของเราก็คือ เป็นลูกพี่ที่ดี นั่นคือ หาวิธีที่จะใช้ประโยชน์จากจุดแข็ง แก้จุดอ่อน และใช้พลังงานและอารมณ์ของเรา กับงานที่ทำ โดยไม่ให้อารมณ์เข้าไปกวนความสามารถในการทำงานของเรา

การเรียนรู้วิธีดีบัก อาจจะทำให้สับสน แต่มันก็จะเป็นทักษะที่มีค่ามาก เป็นประโยชน์กับเรื่องอื่น ๆ ด้วย นอกเหนือจากการเขียนโปรแกรม ส่วนท้ายของแต่ละบท จะมีหัวข้อคล้าย ๆ หัวข้อนี้ (การดีบัก) ที่แนะนำวิธีดีบัก และหวังว่า คำแนะนำเหล่านี้จะช่วยได้บ้าง

1.8. อภิธานศัพท์

การแก้ปัญหา (problem solving): กระบวนการที่รวม การอ่านปัญหา การตีความปัญหา การหาวิธีแก้ปัญหา และการอธิบายวิธีแก้ปัญหาที่ได้

ภาษาระดับสูง (high-level language): ภาษาโปรแกรม เช่น ไพธอน ที่ออกแบบมาให้คนอ่านและเขียนได้ง่าย

ภาษาระดับล่าง (low-level language): ภาษาโปรแกรม ที่ออกแบบมาให้คอมพิวเตอร์ทำงานได้ง่าย บางครั้ง ภาษาระดับล่าง เรียกว่า “ภาษาเครื่อง” (machine language) หรือ “ภาษาแอสเซมบลี” (assembly language)

ความเคลื่อนย้ายง่าย (portability): คุณสมบัติของโปรแกรม ที่สามารถทำงานบนคอมพิวเตอร์ได้หลายประเภท

อินเตอร์พรีเตอร์ (interpreter): โปรแกรมที่อ่าน โปรแกรมอื่น และดำเนินการตามโปรแกรมที่อ่าน

พร้อม (prompt): ตัวอักขระที่แสดงโดยอินเตอร์พรีเตอร์ เพื่อจะบอกว่า อินเตอร์พรีเตอร์พร้อมที่จะรับอินพุตจากผู้ใช้

โปรแกรม (program): ชุดของคำสั่งที่ระบุวิธีการประมวลผล

ข้อความพิมพ์ (print statement): คำสั่ง ที่ทำให้ไพธอนอินเตอร์พรีเตอร์แสดงค่าออกหน้าจอ

ตัวดำเนินการ (operator): สัญลักษณ์พิเศษ ที่แทนการประมวลผลพื้นฐาน เช่น การบวก การคูณ หรือการต่อสายอักขระ

ค่า (value): หนึ่งในหน่วยพื้นฐานของข้อมูลที่โปรแกรมใช้ เช่น ตัวเลข หรือ สายอักขระ

ชนิด (type): ประเภทของ*ค่า*ข้อมูล ชนิดข้อมูลที่เราได้เห็นกันไปแล้ว ได้แก่ จำนวนเต็ม (ชนิด `int`) จำนวนโฟลตติ้งพอยต์ (ชนิด `float`) และ สายอักขระ (ชนิด `str`)

จำนวนเต็ม (integer): ชนิดข้อมูลที่แทนจำนวนเต็ม

โฟลตติ้งพอยต์ (floating-point): ชนิดข้อมูลที่แทนเลขที่มีเศษส่วน

สายอักขระ (string): ชนิดข้อมูลที่แทนชุดลำดับของตัวอักษร

ภาษาธรรมชาติ (natural language): ภาษาใด ๆ ที่ผู้คนใช้พูดกัน และมีวิวัฒนาการของภาษาไปตามธรรมชาติ

ภาษารูปนัย (formal language): ภาษาใด ๆ ที่คนได้ออกแบบมาสำหรับจุดประสงค์เฉพาะ เช่น เพื่อแสดงความคิดเชิงคณิตศาสตร์ หรือโปรแกรมคอมพิวเตอร์ ภาษาคอมพิวเตอร์ทุกภาษาเป็นภาษารูปนัย

โทเค็น (token): หน่วยพื้นฐาน ของโครงสร้างไวยากรณ์ของโปรแกรม ในลักษณะเดียวกับ คำ ของภาษาธรรมชาติ

วากยสัมพันธ์ (syntax): กฎเกณฑ์ต่าง ๆ ที่กำหนดโครงสร้างของโปรแกรม

แจงส่วน (parse): ตรวจสอบโปรแกรม และวิเคราะห์โครงสร้างตามวากยสัมพันธ์

บัก (bug): ความผิดพลาดในโปรแกรม

การดีบัก (debugging): กระบวนการหาและแก้ไขบัก

1.9. แบบฝึกหัด

***แบบฝึกหัด* 1.1.** *ถ้าเราอ่านหนังสือนี้หน้าคอมพิวเตอร์ ก็จะดีที่ว่า เราสามารถทดลองตัวอย่างต่าง ๆ ตามที่อ่านไปได้เลย*

เมื่อเราทดลองความรู้ใหม่ ๆ ที่ได้เรียน เราควรจะลองที่จะทำอะไรผิดด้วย ตัวอย่างเช่น ในโปรแกรม “Hello, world!” มันจะเกิดอะไรขึ้น ถ้าเราใส่อัญประกาศ (`'`) แค่ตัวเดียว? มันจะเกิดอะไรขึ้น ถ้าเราไม่ใส่อัญประกาศเลย? มันจะเกิดอะไรขึ้น ถ้าเราสะกด `print` ผิด?

การทดลองในลักษณะนี้ จะช่วยให้เราจำสิ่งที่เราอ่านได้ และก็จะช่วยตอนที่เราเขียนโปรแกรมด้วย เพราะเราจะรู้ว่าข้อความแจ้งข้อผิดพลาด (error messages) หมายความว่าอย่างไร

มันดีกว่าที่ เราจะทำพลาดตอนนี้ โดยตั้งใจ ดีกว่าที่ เราไปพลาดตอนหลัง โดยไม่ตั้งใจ.

1. *ในคำสั่ง* `print` *มันจะเกิดอะไรขึ้น ถ้าเราใส่วงเล็บแค่อันเดียว หรือไม่ใส่เลย?*
2. *ถ้าเราลองพิมพ์สายอักขระ (string) ออกมา มันจะเกิดอะไรขึ้น ถ้าเราใส่อัญประกาศ (*`'`*) อันเดียว หรือไม่ใส่เลย?*
3. *เราสามารถใช้เครื่องหมายลบ เพื่อสร้างเลขลบออกมาได้ เช่น* `-2`*. มันจะเกิดอะไรขึ้น ถ้าเราใส่เครื่องหมายบวกไปหน้าตัวเลข? ถ้าอันนี้จะได้อะไรออกมา* `2++2`*?*
4. *ในคณิตศาสตร์ การใส่ศูนย์ข้างหน้าตัวเลขไม่มีปัญหาอะไร เช่น* `02` *มันจะเกิดอะไรขึ้น ถ้าทำแบบนี้ในไพธอน?*
5. *มันจะเกิดอะไรขึ้น ถ้าเราลองสองตัวเลข โดยไม่มีตัวดำเนินการ (operator) ระหว่างตัวเลขเลย? ลอง* `4 8`

***แบบฝึกหัด* 1.2.** *เปิดไพธอนอินเตอร์พรีเตอร์มา และใช้มันเป็นเหมือนเครื่องคิดเลข*

1. *42 นาที 42 วินาที คิดเป็นกี่วินาที?*
2. *10 กิโลเมตร คิดเป็นกี่ไมล์? คำใบ้: 1.61 กิโลเมตร เท่ากับ 1 ไมล์*
3. *ถ้าเราใช้เวลา 42 นาที 42 วินาทีวิ่งได้ 10 กิโลเมตร เราใช้เวลากี่นาทีในการวิ่งต่อไมล์? เราวิ่งได้เร็วกี่ไมล์ต่อชั่วโมง?*

2. ตัวแปร นิพจน์ และคำสั่ง

สิ่งที่ทรงพลังมากที่สุดสิ่งหนึ่งในภาษาของการเขียนโปรแกรม คือ การที่เราสามารถจัดการกับ **ตัวแปร (variable)** ตัวแปร คือ ชื่อที่ถูกอ้างถึงค่าหนึ่ง ๆ

(หมายเหตุผู้แปล: ค่าที่ถูกเก็บอยู่ในหน่วยความจำ)

2.1. คำสั่งที่ใช้ในการกำหนดค่า

คำสั่งกำหนดค่า จะสร้างตัวแปรตัวใหม่ขึ้นมา พร้อมให้ค่ากับตัวแปรนั้น:

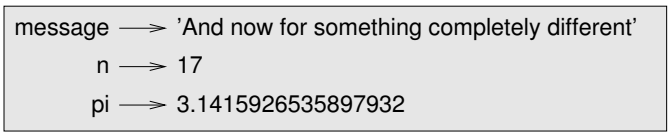
```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
```

ตัวอย่างข้างบนนี้ได้ทำการกำหนดค่า 3 ค่า คำสั่งแรกกำหนดสายอักขระ (string) ในตัวแปรใหม่ที่ชื่อว่า `message`; คำสั่งที่สอง กำหนดค่าจำนวนเต็ม (integer) `17` ให้กับตัวแปร `n`; คำสั่งที่สาม กำหนดค่าประมาณของค่า π ให้กับตัวแปรที่ชื่อว่า `pi`

โดยปกติแล้ว เราแสดงตัวแปรบนหน้ากระดาษโดยใช้การเขียนชื่อตัวแปรแล้วชี้ไปที่ค่าของมัน การเขียนแบบนี้เรียกว่า **แผนภาพสถานะ (state diagram)** เพราะว่ามันแสดงสถานะว่าค่าของตัวแปร นั้นเป็นอย่างไรในตอนนี้ (ให้ลองนึกถึงสถานะต่าง ๆ ที่เปลี่ยนแปลงไปของจิตของเรา) รูปที่ 2.1 แสดงให้เห็นถึงผลการให้ค่าของตัวอย่างที่แล้ว

2.2. ชื่อตัวแปร

โดยทั่วไปแล้ว โปรแกรมเมอร์เลือกชื่อสำหรับตัวแปรที่มีความหมาย—พวกเขาทำการบันทึก ว่าตัวแปรหนึ่ง ๆ เอาไว้ทำอะไรหรือเก็บค่าอะไร



รูปที่ 2.1.: แผนภาพสถานะ

ชื่อของตัวแปรสามารถยาวเท่าที่เราต้องการได้ ชื่อของตัวแปรสามารถมีทั้งตัวอักษรและตัวเลข แต่ชื่อนั้นไม่สามารถขึ้นต้นด้วยตัวเลขได้ จริง ๆ แล้วเราสามารถขึ้นต้นชื่อตัวแปรด้วยอักษร ตัวใหญ่ได้ แต่เรานิยมเริ่มต้นด้วยอักษรตัวเล็กซะมากกว่า

ในชื่อตัวแปรนั้นสามารถมีเครื่องหมายขีดล่าง (underscore), `_`, ส่วนใหญ่แล้วเราจะ ใช้ชื่อตัวแปรที่มีเครื่องหมายขีดล่างในชื่อที่ประกอบด้วยคำหลายคำ เช่น `your_name` หรือ `airspeed_of_unladen_swallow`

ถ้าเราตั้งชื่อตัวแปรผิดกฎ เราจะได้ข้อผิดพลาดทางวากยสัมพันธ์ (syntax error) หรือข้อผิดพลาด เชิงกฎเกณฑ์ของภาษา

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

ชื่อตัวแปร `76trombones` นั้นผิดกฎเพราะว่ามันเริ่มด้วยตัวเลข ส่วนชื่อตัวแปร `more@` นั้นผิดกฎเพราะว่ามันมีอักขระต้องห้าม `@` แล้วสำหรับชื่อตัวแปร `class` มันผิดอย่างไรกันนะ

ปรากฏว่า ชื่อตัวแปร `class` นั้นเป็น **คำสำคัญ (keywords)** ของภาษาไพธอน ตัวแปลภาษาใช้คำสำคัญเหล่านี้ในการรู้จำโครงสร้างของโปรแกรม เราจึงไม่สามารถใช้คำสำคัญเหล่านี้ ในการตั้งชื่อตัวแปรได้

ไพธอนเวอร์ชัน 3 มีคำสำคัญดังนี้:

```
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
break     except    in        raise
```

เราไม่จำเป็นต้องจำคำสำคัญเหล่านี้ทั้งหมด ในโปรแกรมที่มีสภาพแวดล้อมสำหรับการพัฒนา (development environment) ส่วนใหญ่แล้วจะแสดงคำสำคัญเหล่านี้ โดยใช้สีที่แตกต่างจากโค้ดธรรมดา ถ้าเราพยายามจะใช้คำเหล่านี้ในโปรแกรม เราก็จะรู้ได้โดยสังเกตจากสีที่แตกต่างออกไป

2.3. นิพจน์และคำสั่ง

นิพจน์ (expression) เป็นการรวมกันของค่า ตัวแปร และตัวดำเนินการต่าง ๆ ค่า (value) เดี่ยว ๆ ก็ถือเป็นนิพจน์ เช่นเดียวกับ ตัวแปรเดี่ยว ๆ ด้วย ดังนั้น สิ่งต่าง ๆ ต่อไปนี้ถือเป็นนิพจน์ที่ถูกต้องตามวากยสัมพันธ์

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

เมื่อเราพิมพ์นิพจน์ที่ข้อความพร้อมรับ (prompt) ตัวแปลภาษาจะทำการ **ประเมินค่า (evaluates)** ซึ่งหมายความว่า จะมีการหาค่าของนิพจน์นั้นออกมา ในตัวอย่างนี้ **n** มีค่า 17 และ **n + 25** มีค่าเท่ากับ 42

คำสั่ง (statement) เป็นหน่วยของโค้ดที่ทำให้เกิดการทำงานต่าง ๆ เช่น สร้างตัวแปร หรือแสดงค่า

```
>>> n = 17
>>> print(n)
```

บรรทัดแรกเป็นคำสั่งที่ให้ค่ากับตัวแปร **n** บรรทัดที่สองเป็นคำสั่งพิมพ์ ที่ใช้แสดงค่าของตัวแปร **n**

เมื่อเราพิมพ์คำสั่ง ตัวแปลภาษาจะ **ดำเนินงาน (execute)** ตามคำสั่งนั้น ซึ่งหมายความว่า จะมีการทำงานตามที่คำสั่งนั้นบอกให้ทำ โดยทั่วไปแล้ว คำสั่ง (statement) จะไม่มีค่าใด ๆ (ไม่ได้ให้ค่า หรือเก็บค่า)

2.4. โหมดสคริปต์

จนถึงตอนนี้ เราได้รันไพธอนใน **โหมดโต้ตอบ (interactive mode)** ซึ่งหมายความว่า เราพิมพ์โค้ดเพื่อโต้ตอบโดยตรงกับตัวแปลภาษา โหมดโต้ตอบเป็นโหมดที่ดีสำหรับการเริ่มต้นเขียนโปรแกรม แต่หากเราต้องการเขียนมากกว่าสองสามบรรทัด มันอาจจะดูงุ่มง่ามหน่อย

อีกทางเลือกในการเขียนโปรแกรม คือ การเขียนและบันทึก (save) โค้ดลงในไฟล์ที่เรียกว่า **สคริปต์ (script)** และให้ตัวแปลภาษาทำงานใน **โหมดสคริปต์ (script mode)** เพื่อที่จะปฏิบัติตามสคริปต์นั้น เรานิยมตั้งชื่อไฟล์สคริปต์โดยใช้นามสกุล `.py`

ถ้าเกิดเรารู้ว่าจะสร้างและรันสคริปต์บนเครื่องคอมพิวเตอร์ของเราได้อย่างไร เราก็พร้อมที่จะไปต่อแล้ว แต่หากยังไม่รู้ ผมแนะนำให้ใช้ระบบ PythonAnywhere อีกครั้ง ผมได้เขียนขั้นตอนการรัน โหมดสคริปต์ในระบบที่ `http://tinyurl.com/thinkpython2e`.

เนื่องจากภาษาไพธอนสามารถทำงานได้ในทั้งสองโหมด เราสามารถทดสอบบางส่วนของโค้ดในโหมดโต้ตอบ ก่อนที่จะเอาโค้ดไปวางในสคริปต์ได้ แต่มันก็มีความแตกต่างระหว่างสองโหมดนี้บางประการ ที่ทำให้โปรแกรมเมอร์สับสนได้เช่นกัน

ตัวอย่างเช่น ถ้าเราเอาไพธอนเป็นเครื่องคิดเลข เราจะพิมพ์ว่า

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

บรรทัดแรกกำหนดค่าให้ตัวแปร `miles` แต่มันไม่ได้มีผลที่แสดงให้เห็นได้ บรรทัดที่สองเป็นนิพจน์ ดังนั้นตัวแปลภาษาจึงประเมินค่าและแสดงผลออกมาให้เราดู นั่นคือ ระยะทางของการวิ่งมาราธอน คือ ประมาณ 42 กิโลเมตร

แต่ถ้าเราพิมพ์โค้ดดังกล่าวในไฟล์สคริปต์และรันสคริปต์นั้น เราจะไม่ได้เห็นการแสดงค่าเลย ในโหมดสคริปต์นั้น ตัวของนิพจน์เองจะไม่มีผลที่แสดงให้เห็นได้ ไพธอนจะมีการประเมินนิพจน์ นั้นอยู่ แต่มันจะไม่แสดงผลออกมาให้เห็น นอกจากเราจะบอกให้แสดง (โดยใช้คำสั่ง แสดงผล):

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

พฤติกรรมนี้อาจจะทำให้โปรแกรมเมอร์งงได้ในตอนแรก ๆ

โดยปกติแล้ว ไฟล์สคริปต์จะมีลำดับของคำสั่ง ถ้ามีคำสั่งในสคริปต์มากกว่าหนึ่งคำสั่ง คำสั่งจะปฏิบัติและแสดงผลออกมาทีละคำสั่งตามลำดับ (จากบนลงล่าง)

ตัวอย่างเช่น ในสคริปต์ต่อไปนี้

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

ทำให้เกิดการแสดงผล คือ

```
1
2
```

คำสั่งกำหนดค่า (assignment statement) ไม่ได้ทำให้เกิดการแสดงผลใด ๆ

เพื่อที่จะตรวจสอบความเข้าใจของเรา ให้พิมพ์คำสั่งต่อไปนี้ในตัวแปลภาษาไพธอน (โหมดโต้ตอบ) และลองดูว่ามันทำอะไรบ้าง:

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

คราวนี้ ให้เอาคำสั่งข้างบนเดียวกันนี้ไปใส่ในไฟล์สคริปต์ แล้วลองรันดู มันให้ผลอย่างไรบ้าง?

จากนั้น ให้แก้ไขแต่ละคำสั่งให้เป็นคำสั่งพิมพ์ (print) และลองรันดูอีกที

2.5. ลำดับการดำเนินการ

เมื่อนิพจน์ประกอบด้วยตัวดำเนินการ (operator) มากกว่าหนึ่งตัว ลำดับของการประเมินผล ค่านั้นจะขึ้นอยู่กับ **ลำดับของตัวดำเนินการ (order of operations)** สำหรับ ตัวดำเนินการทางคณิตศาสตร์ (mathematical operator) ไพธอนจะทำตามลำดับที่เป็น ที่นิยมกันทั่วไป โดยให้เราจำตัวย่อ **PEMDAS** เพื่อที่จะจำได้ง่ายขึ้น ดังนี้:

- **P**arentheses หรือ วงเล็บ มีลำดับความสำคัญสูงที่สุด และสามารถใช้เพื่อ บังคับให้นิพจน์ถูกประเมินค่าตามลำดับที่เราต้องการได้ เนื่องจากนิพจน์ในวงเล็บจะถูกประเมินค่า ก่อน ดังนั้น `2 * (3-1)` จะมีค่าเท่ากับ 4 และ `(1+1)**(5-2)` มีค่าเท่ากับ 8 เราสามารถใช้วงเล็บเขียนให้นิพจน์อ่านได้ง่ายขึ้นด้วย เช่น ในนิพจน์ `(minute * 100) / 60` ถึงแม้ว่าลำดับการประเมินค่าจะไม่เปลี่ยนเลยก็ตาม
- **E**xponentiation หรือการยกกำลัง มีลำดับความสำคัญมากที่สุดเป็นลำดับรองลงมา ดังนั้น `1 + 2**3` มีค่า 9 ไม่ใช่ 27 และ `2 * 3**2` มีค่าเป็น 18 ไม่ใช่ 36

- Multiplication (การคูณ) และ Division (การหาร) มีลำดับ ความสำคัญมากกว่า Addition (การบวก) และ Subtraction การลบ ดังนั้น `2*3-1` มีค่าเป็น 5 ไม่ใช่ 4และ `6+4/2` มีค่าเป็น 8 ไม่ใช่ 5

- ตัวดำเนินการที่มีลำดับความสำคัญเท่ากันจะถูกประเมินค่าจากซ้ายไปขวา (นอกจากการยกกำลัง) ดังนั้นในนิพจน์ `degrees / 2 * pi` การหารจะถูกทำก่อน และผลของการหารจะถูกคูณด้วยค่า `pi` ถ้าเราต้องการจะหารด้วย 2π เราสามารถใช้วงเล็บครอบ 2π ไว้ หรือเขียนว่า `degrees / 2 / pi`

ผมเองไม่ค่อยจะได้ใส่ใจที่จะจำลำดับความสำคัญพวกนี้เท่าไร แต่จะใช้วิธีว่า ถ้าเกิดผมมองนิพจน์ แล้วประเมินค่าไม่ได้ทันที ก็จะใช้วิธีเขียนวงเล็บเพื่อจะทำให้นิพจน์นั้นชัดเจนขึ้นแทน

2.6. การดำเนินการกับสายอักขระ

โดยทั่วไปแล้ว เราไม่สามารถดำเนินการทางคณิตศาสตร์กับสายอักขระได้ แม้ว่าสายอักขระจะดูเหมือนเป็นตัวเลขก็ตาม ดังนั้น การทำดังต่อไปนี้จะผิดพลาดเชิงวากยสัมพันธ์:

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

แต่ก็มีข้อยกเว้นสำหรับ 2 อย่าง คือ เครื่องหมาย **+** และ *

ตัวดำเนินการ **+** จะทำ **การเชื่อมสายอักขระ (string concatenation)** ซึ่งหมายความว่ามันจะเชื่อมสายอักขระสองตัวเข้าด้วยกันทางด้านท้าย เช่น:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

ตัวดำเนินการ * ยังสามารถทำงานกับสายอักขระได้ด้วย มันทำให้เกิดการทำซ้ำสายอักขระ เช่น `'Spam'*3` ให้ผลว่า `'SpamSpamSpam'` ถ้าตัวถูกดำเนินการตัวใดตัวหนึ่งเป็นสายอักขระ อีกตัวหนึ่งจะต้องเป็นจำนวนเต็มเสมอ (ไม่เช่นนั้นจะเกิดข้อผิดพลาดขึ้น)

การใช้งาน **+** และ * ในลักษณะนี้ มันก็สอดคล้องกับการบวกและการคูณ เปรียบเทียบกับการทำ `4*3` จะให้ค่าเท่ากับ `4+4+4` เราจึงคาดว่า `'Spam'*3` ก็น่าจะให้ค่าเท่ากับ `'Spam'+'Spam'+'Spam'` และมันก็เป็นเช่นนั้น ในทางกลับกัน การต่อสายอักขระและการทำซ้ำสายอักขระ ก็มีความแตกต่างอย่าง

มากกับการบวกและคูณจำนวนเต็ม เราลองคิดดูสิว่า มีคุณสมบัติอะไรบ้างที่การบวกจำนวนเต็มนั้นแตกต่างจากการต่อสายอักขระ

2.7. คอมเมนต์

เมื่อโปรแกรมเริ่มที่จะใหญ่ขึ้นและซับซ้อนมากขึ้น นั่นก็จะทำให้อ่านยากมากขึ้นด้วย พวกภาษาเชิงรูปแบบมักจะมีโค้ดหนาแน่นและยากที่จะดูส่วนต่าง ๆ ของโค้ด ว่าแต่ละส่วนนั้นทำอะไรบ้าง และทำไมจึงถูกเขียนขึ้นมาแบบนั้น

จากเหตุผลดังกล่าว มันเลยเป็นการดีที่จะเพิ่มคำอธิบายโค้ดสั้น ๆ เข้าไปในโปรแกรมด้วย เพื่ออธิบายให้เป็นภาษามนุษย์ ว่าโปรแกรมทำงานอะไรบ้าง คำอธิบายเหล่านี้เรียกว่า **คอมเมนต์ (comments)** และส่วนของคอมเมนต์จะเริ่มด้วยสัญลักษณ์ **#**:

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

ในกรณีนี้ คอมเมนต์อยู่ในบรรทัดเดี่ยว ๆ ของมันเอง นอกจากนี้เรายังสามารถใส่คอมเมนต์ เข้าไปในท้ายบรรทัดของคำสั่งต่าง ๆ ได้ด้วย:

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

ทุกอย่างตั้งแต่สัญลักษณ์ **#** ไปจนถึงจบบรรทัดนั้นจะถูกเมินโดยโปรแกรม— มันจะไม่มีผลกับการทำงานของโปรแกรมเลย

คอมเมนต์จะมีประโยชน์มากที่สุดเมื่อใช้อธิบายลักษณะเฉพาะของโปรแกรมที่เข้าใจได้ยาก หรือไม่ชัดเจน มันเหมาะสมที่เราจะสมมติว่าผู้ที่อ่านโค้ดนั้นเข้าใจว่าโค้ด *ทำอะไร* ดังนั้น จะมีประโยชน์มากกว่าที่จะอธิบายว่าโค้ดนั้นมีไว้ *ทำไม*

คอมเมนต์ข้างล่างนี้ซ้ำซ้อนกับโค้ดและไม่มีประโยชน์:

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

ส่วนคอมเมนต์ต่อไปนี้ มีข้อมูลที่เป็นประโยชน์ที่ไม่ได้อยู่ในโค้ด:

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

การตั้งชื่อตัวแปรให้ดีมีความหมายจะช่วยลดความจำเป็นในการมีคอมเมนต์ แต่ถ้าชื่อตัวแปร ยาวเกินไปก็อาจจะทำให้นิพจน์ที่ซับซ้อนนั้นอ่านยาก เราต้องลองชั่งน้ำหนักดู

2.8. การดีบัก

ข้อผิดพลาดในโปรแกรมสามารถเกิดได้ 3 แบบ: ข้อผิดพลาดเชิงวากยสัมพันธ์ (syntax error), ข้อผิดพลาดเวลาดำเนินการ (runtime error) และ ข้อผิดพลาดเชิงความหมาย (semantic error) เวลาเกิดข้อผิดพลาดมันจะเป็นประโยชน์ ถ้าเราสามารถแยกแยะได้ว่า เป็นความผิดพลาดประเภทใด เพื่อที่จะหาสาเหตุได้เร็วขึ้น

ข้อผิดพลาดเชิงวากยสัมพันธ์ (Syntax error): "syntax" หมายความถึงโครงสร้างของ โปรแกรม และกฎเกณฑ์ต่าง ๆ ที่เกี่ยวกับโครงสร้างนั้น เช่น วงเล็บจะต้องมีคู่เปิดปิดให้ครบ ดังนั้น `(1 + 2)` จะถูกกฎ แต่ `8)` จะเป็นข้อผิดพลาดเชิงวากยสัมพันธ์

หากเกิดข้อผิดพลาดเชิงวากยสัมพันธ์สักแห่งในโปรแกรม ไพธอนจะแสดงข้อความระบุข้อผิดพลาด (error message) และออกจากโปรแกรมไป และเราก็จะไม่สามารถรันโปรแกรมนั้นได้ ในช่วงเวลาที่เราเริ่มฝึกเขียน 2-3 สัปดาห์แรก เราอาจจะต้องเวลามากในการหาสาเหตุของข้อผิดพลาดเชิงวากยสัมพันธ์ แต่เมื่อเรามีประสบการณ์ในการเขียนโปรแกรมมากขึ้นแล้ว เราอาจจะมีข้อผิดพลาดเหล่านี้น้อยลงและหาต้นตอได้เร็วขึ้น

ข้อผิดพลาดตอนดำเนินการ (Runtime error): ข้อผิดพลาดแบบที่สอง คือ ข้อผิดพลาดตอนดำเนินการ (เวลารันโปรแกรม) เราเรียกแบบนั้น เพราะว่าข้อผิดพลาดแบบนี้จะไม่เกิดขึ้นจนกว่าโปรแกรมจะเริ่มรัน เรายังเรียกข้อผิดพลาดแบบนี้ได้ว่า **เอ็กเซ็ปชั่น (exceptions)** เพราะโดยปกติแล้วจะหมายความว่า มันมีความผิดปกติในการทำงานเกิดขึ้น (และแย่ด้วย)

ข้อผิดพลาดตอนดำเนินการนั้น เกิดได้ยากในโปรแกรมพื้นฐานที่เราจะได้เห็นใน 2-3 บทแรกนี้ ดังนั้น อีกสักพักเลยเราถึงจะเจอข้อผิดพลาดแบบนี้

ข้อผิดพลาดเชิงความหมาย (Semantic error): ข้อผิดพลาดอย่างที่สาม คือ "semantic" ซึ่งเกี่ยวกับความหมายของคำสั่ง ถ้าหากโปรแกรมของเราเกิดข้อผิดพลาดเชิงความหมาย โปรแกรมจะยังคงทำงาน ได้อยู่โดยไม่มีข้อความระบุข้อผิดพลาดใด ๆ แต่มันจะทำให้โปรแกรมนั้นทำงานไม่ถูกต้องตามที่เราต้องการ มันจะทำอย่างอื่นแทน นั่นคือ มันจะทำอะไรที่เราบอกให้มันทำ

(หมายเหตุผู้แปล: อาจจะไม่ใช่สิ่งที่เราต้องการทำจริง ๆ คือ เราบอกมันผิดจากที่เราต้องการนั่นเอง)

การระบุข้อผิดพลาดเชิงความหมายอาจจะลำบากหน่อย เพราะเราจะต้องกลับไปไล่ดูผลการทำงานของโปรแกรม และพยายามที่จะหาว่าโปรแกรมมันทำอะไรกันแน่ (และทำพลาดตรงไหน)

2.9. อภิธานศัพท์

ตัวแปร (variable): ชื่อที่อ้างถึงค่าค่าหนึ่ง (ในหน่วยความจำ)

การกำหนดค่า (assignment): คำสั่งที่กำหนดค่าให้ตัวแปร

แผนภาพสถานะ (state diagram): รูปที่แสดงตัวแปรและค่าที่มันอ้างถึง

คำสำคัญ (keyword): คำที่สงวนไว้สำหรับแยกวิเคราะห์ส่วนต่าง ๆ ของโปรแกรม เราไม่สามารถใช้คำสำคัญ เช่น `if`, `def`, และ `while` เพื่อเป็นชื่อของตัวแปรได้

ตัวถูกดำเนินการ (operand): ค่าที่ตัวดำเนินการ (operator) ใช้ทำงาน

นิพจน์ (expression): การประกอบกันของตัวแปร ตัวดำเนินการ และค่าต่าง ๆ ที่ทำให้เกิดผลอย่างใดอย่างหนึ่ง

ประเมินค่า (evaluate): การทำให้นิพจน์นั้นง่ายขึ้น ด้วยการดำเนินการต่าง ๆ เพื่อให้เหลือแค่ค่าเดียว

คำสั่ง (statement): ส่วนของโค้ดที่เป็นการสั่งงาน หรือ การกระทำ จนถึงตอนนี้ คำสั่งที่เราเจอมีแค่คำสั่งกำหนดค่าและคำสั่งพิมพ์

ดำเนินงาน (execute): รันคำสั่งและทำตามคำสั่ง

โหมดโต้ตอบ (interactive mode): วิธีการใช้ตัวแปลภาษาไพธอนโดยการพิมพ์โค้ดที่ prompt เลย

โหมดสคริปต์ (script mode): วิธีการใช้ตัวแปลภาษาไพธอนโดยการให้อ่านโค้ดจากสคริปต์ และรันสคริปต์

สคริปต์ (script): โปรแกรมที่ถูกเก็บในไฟล์

ลำดับการดำเนินการ (order of operations): กฎที่กำหนดลำดับการประเมินค่าในนิพจน์ที่มีตัวดำเนินการและตัวถูกดำเนินการหลายตัว

เชื่อม (concatenate): เชื่อมตัวถูกดำเนินการสองตัวเข้าด้วยกันทางด้านท้าย

คอมเมนต์ (comment): ข้อมูลในโปรแกรมที่เขียนขึ้นสำหรับโปรแกรมเมอร์คนอื่น (หรือ ใครก็ตามที่อ่านโค้ด) ให้เข้าใจโค้ด และมันไม่มีผลต่อการดำเนินการหรือการทำงานของโปรแกรม

ข้อผิดพลาดเชิงวากยสัมพันธ์ (syntax error): ข้อผิดพลาดในโปรแกรมที่ทำให้โปรแกรมแยกแยะโครงสร้างไม่ได้ (และทำให้แปลความหมายของโปรแกรมไม่ได้)

เอ็กเซ็ปชั่น (exception): ข้อผิดพลาดที่ถูกพบตอนที่โปรแกรมกำลังทำงานอยู่

อรรถศาสตร์ (semantics): ความหมายของโปรแกรม

ข้อผิดพลาดเชิงความหมาย (semantic error): ข้อผิดพลาดในโปรแกรมที่ทำให้โปรแกรมทำงานอย่างอื่น นอกเหนือจากที่โปรแกรมเมอร์ตั้งใจให้เป็น

2.10. แบบฝึกหัด

***แบบฝึกหัด* 2.1.** *ขอย้ำคำแนะนำจากบทที่แล้ว เมื่อไหร่ก็ตามที่เราได้เรียนรู้ลักษณะเฉพาะใหม่ ๆ ของภาษา เราควรที่จะต้องทดลองในโหมดโต้ตอบ และลองทำให้เกิดข้อผิดพลาดต่าง ๆ เพื่อที่จะดูว่าจะเกิดอะไรขึ้นได้บ้าง*

- *เราเห็นแล้วว่า* `n = 42` *มันถูกต้องตามวากยสัมพันธ์ แล้วถ้าเป็น* `42 = n` *ล่ะ?*
- *แล้วถ้าเป็น* `x = y = 1` *ล่ะ?*
- *ในบางภาษา ทุกคำสั่งจะจบด้วยเครื่องหมายอัฒภาค (จุดครึ่ง หรือ เซมิโคลอน),* `;` *มันจะเกิดอะไรขึ้นถ้าเราใส่เครื่องหมาย* `;` *ในตอนท้ายของคำสั่งไพธอน?*
- *แล้วถ้าใส่จุดในท้ายคำสั่งไพธอนล่ะ?*
- *ในสัญกรณ์ทางคณิตศาสตร์ (math notation) เราสามารถคูณ* x *กับ* y *แบบนี้ได้:* xy *จะเกิดอะไรขึ้นถ้าเราลองทำแบบนี้บ้างในไพธอน?*

***แบบฝึกหัด* 2.2.** *ให้ฝึกการใช้ตัวแปลภาษาไพธอนให้เป็นเครื่องคิดเลข:*

1. *ปริมาตรของทรงกลมที่มีรัศมี* r *คือ* $\frac{4}{3}\pi r^3$ *แล้วปริมาตรของทรงกลมที่มีรัศมีเท่ากับ 5 เป็นเท่าไร*
2. *สมมติว่า ราคาหน้าปกหนังสือ คือ $24.95 แต่ร้านหนังสือลดราคาให้ 40% ค่าส่งหนังสือ คือ $3 สำหรับเล่มแรก และ 75 เซ็นต์ ($0.75) สำหรับแต่ละเล่มที่เพิ่มขึ้นมา ราคาของการซื้อหนังสือ จำนวน 60 เล่มจะเป็นเท่าไร*
3. *ถ้าเราออกจากบ้านในเวลา 6:52 และวิ่ง 1 ไมล์แบบเบา ๆ (8:15 นาทีต่อไมล์) จากนั้นวิ่งทำเวลาเป็นจำนวน 3 ไมล์ (7:12 นาทีต่อไมล์) และจบด้วยวิ่งเบา ๆ อีก 1 ไมล์ แล้วเราจะได้กลับบ้านไปกินข้าวเช้าในเวลาใด*

3. ฟังก์ชัน

ในบริบทของการเขียนโปรแกรม **ฟังก์ชัน (function)** เป็นลำดับของคำสั่งต่าง ๆ ที่ทำงานต่อเนื่องกันที่ถูกกำหนดชื่อให้ เมื่อเรานิยามฟังก์ชัน เราต้องระบุชื่อและลำดับของคำสั่งต่าง ๆ หลังจากนั้น เราสามารถ “เรียก” ฟังก์ชันโดยใช้ชื่อที่ระบุไว้ได้

3.1. การเรียกฟังก์ชัน

เราได้เห็นตัวอย่างหนึ่งของการเรียกฟังก์ชันไปแล้ว:

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

ชื่อของฟังก์ชันนี้คือ `type` นิพจน์ในวงเล็บ เรียกว่า **อาร์กิวเมนต์ (argument)** ของฟังก์ชัน ผลลัพธ์ของฟังก์ชันนี้ คือ ชนิดของฟังก์ชัน (ในกรณีนี้ คือ ชนิดจำนวนเต็ม (int))

โดยปกติแล้วเราพูดได้ว่า ฟังก์ชันจะ “รับ” *อาร์กิวเมนต์* (*argument)* และ “**ส่งคืน**” (**return**) ค่าผลลัพธ์ของฟังก์ชัน ผลลัพธ์นี้ยังสามารถเรียกว่า “**ค่าคืนกลับ**” (return value) ได้อีกด้วย

ไพธอนได้เตรียมฟังก์ชันที่แปลงค่าจากชนิดหนึ่งไปยังอีกชนิดหนึ่งไว้ให้ ฟังก์ชัน `int` รับค่าอะไรก็ได้ และแปลงค่าที่รับมานั้นไปเป็นจำนวนเต็มถ้ามันสามารถทำได้ ถ้าทำไม่ได้มันก็จะบ่นใส่เรา:

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

ฟังก์ชัน `int` สามารถแปลงเลขทศนิยม หรือโฟลตติ้งพอยต์ (floating-point) ไปเป็นจำนวนเต็มได้ แต่มันไม่ได้ใช้การปัดเศษ (ขึ้นหรือลง) มันจะตัดจุดทศนิยมออกไปเลย:

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

ฟังก์ชัน `float` แปลงจำนวนเต็มและสายอักขระไปเป็นเลขทศนิยม:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

อย่างสุดท้าย ฟังก์ชัน `str` แปลงอาร์กิวเมนต์ของมันไปเป็นสายอักขระ:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2. ฟังก์ชันทางคณิตศาสตร์

ไพธอนมีมอดูลสำหรับฟังก์ชันทางคณิตศาสตร์ที่เราคุ้นเคยกัน **มอดูล (module)** เป็นไฟล์ที่บรรจุกลุ่มของฟังก์ชันที่เกี่ยวข้องกันไว้ด้วยกัน

ก่อนที่เราจะใช้ฟังก์ชันจากมอดูลใด ๆ เราจะต้องนำเข้า (import) มอดูลนั้นก่อน ด้วย **คำสั่งนำเข้า (import statement)**

```
>>> import math
```

คำสั่งนี้สร้าง **อ็อบเจกต์มอดูล (module object)** ที่ชื่อว่า math ถ้าเราสั่งให้แสดงอ็อบเจกต์มอดูลอันนี้ เราจะได้ข้อมูลเกี่ยวกับตัวอ็อบเจกต์ออกมา:

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

อ็อบเจกต์มอดูลบรรจุฟังก์ชันต่าง ๆ และตัวแปรต่าง ๆ ที่ถูกนิยามขึ้นในมอดูล ในการเข้าถึงฟังก์ชันหนึ่ง ๆ ที่อยู่ในมอดูลนี้ เราจะต้องระบุชื่อของมอดูลและชื่อของฟังก์ชันที่เราต้องการ โดยการคั่นด้วยจุด รูปแบบดังกล่าว เรียกว่า **สัญกรณ์จุด (dot notation)**

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

ตัวอย่างแรกใช้ฟังก์ชัน `math.log10` ในการคำนวณอัตราส่วนของสัญญาณต่อการรบกวน ในหน่วยเดซิเบล (สมมติว่า `signal_power` และ `noise_power` ได้ถูกนิยามไปแล้ว) มอดูล math ได้จัดเตรียมฟังก์ชัน `log` ซึ่งจะคำนวณค่า log ฐาน `e` ไว้ให้แล้ว

ตัวอย่างที่สองเป็นการหาค่า sin ของตัวแปร `radians` ซึ่งชื่อของตัวแปรนี้ใบ้ให้เราทราบว่า ฟังก์ชัน `sin` และฟังก์ชันอื่นในตรีโกณมิติ (`cos`, `tan`, และอื่น ๆ) นั้น รับอาร์กิวเมนต์ในหน่วยเรเดียน (radian) การแปลงจากหน่วยองศา (degree) มาเป็นหน่วยเรเดียนนั้น เราจะหารด้วย 180 และคูณด้วย π:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

นิพจน์ `math.pi` นำตัวแปร `pi` มาจากมอดูล math ค่าของมันเป็นค่าโฟลตติ้งพอยต์ ที่ประมาณค่าของ π มาอีกที โดยมีจุดทศนิยมประมาณ 15 ตำแหน่ง

ถ้าเรารู้จักตรีโกณมิติ เราจะสามารถตรวจสอบผลการทำงานก่อนหน้านี้ โดยเปรียบเทียบกับค่ารากที่สองของ 2 แล้วหารด้วย 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3. การประกอบ

ถึงตอนนี้ เราได้ดูส่วนประกอบต่าง ๆ ของโปรแกรม—ตัวแปร นิพจน์ คำสั่ง—แบบแยกกัน โดยที่ไม่ได้พูดถึงว่า เราจะประกอบมันเข้าด้วยกันได้อย่างไร

ลักษณะเฉพาะที่ทรงพลังที่สุดอย่างหนึ่งของภาษาการเขียนโปรแกรม นั่นคือ การที่พวกมันสามารถ เอาบล็อกก่อสร้างเล็ก ๆ มา **ประกอบ** เข้าด้วยกัน เช่น การที่อาร์กิวเมนต์ของฟังก์ชัน สามารถเป็น นิพจน์แบบไหนก็ได้ รวมถึงตัวดำเนินการทางคณิตศาสตร์ด้วย:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

และแม้แต่การเรียกฟังก์ชัน:

```
x = math.exp(math.log(x+1))
```

ในเกือบทุกที่ เราสามารถใส่ค่านิพจน์ใด ๆ ได้ ยกเว้นหนึ่งที่: ทางซ้ายของการกำหนดค่า จะต้องเป็นชื่อตัวแปร การวางนิพจน์แบบอื่นทางซ้ายนั้น จะเป็นข้อผิดพลาดเชิงวากยสัมพันธ์ (syntax error) (เราจะได้เห็นข้อยกเว้นของกฎข้อนี้ในภายหลัง)

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

3.4. การเพิ่มฟังก์ชันใหม่

ถึงตอนนี้ เราได้ใช้แค่ฟังก์ชันที่มากับไพธอน แต่มันเป็นไปได้ที่เราจะเพิ่มฟังก์ชันต่าง ๆ เข้าไปใหม่ **นิยามของฟังก์ชัน (function definition)** ระบุชื่อของฟังก์ชัน และกลุ่มของคำสั่ง ที่ต่อเนื่องกันในฟังก์ชัน ซึ่งคำสั่งเหล่านี้จะทำงานเมื่อฟังก์ชันถูกเรียกใช้

นี่คือตัวอย่าง:

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

`def` เป็นคำสำคัญที่ระบุว่า นี่คือ นิยามของฟังก์ชัน ชื่อของฟังก์ชัน คือ `print_lyrics` กฎการตั้งชื่อฟังก์ชันนั้นเหมือนกับกฎการตั้งชื่อตัวแปร: ตัวอักษร ตัวเลข และขีดล่างนั้นใช้ได้ แต่อักขระตัวแรกต้องไม่เป็นตัวเลข เราไม่สามารถใช้คำสำคัญเป็นชื่อของฟังก์ชัน และเราต้องหลีกเลี่ยง การใช้ชื่อตัวแปรและชื่อฟังก์ชันที่เหมือนกัน

วงเล็บว่างหลังชื่อฟังก์ชัน ระบุว่าฟังก์ชันนี้ไม่รับอาร์กิวเมนต์ใด ๆ เข้ามาในฟังก์ชัน

บรรทัดแรกของนิยามฟังก์ชันเรียกว่า **ส่วนหัว (header)**; ที่เหลือเรียกว่า **ส่วนตัว (body)** ส่วนหัวจะต้องลงท้ายด้วยเครื่องหมายทวิภาค (จุดคู่ หรือ โคลอน) (:) และส่วนตัวของฟังก์ชันจะต้อง ย่อหน้าเข้าไป การย่อหน้าเข้าไปนิยมใช้ช่องว่าง (space) 4 ตำแหน่ง ตัวของฟังก์ชันสามารถบรรจุคำสั่ง กี่คำสั่งก็ได้

(หมายเหตุผู้แปล: การย่อหน้าสามารถใช้ tab ได้ แต่ต้องเลือกการย่อหน้าให้เป็นรูปแบบเดียวกันทั้งไฟล์)

สายอักขระที่อยู่ในคำสั่งพิมพ์ถูกใส่อยู่ในเครื่องหมายอัญประกาศ อัญประกาศเดี่ยว (single quotes) และอัญประกาศ (double quotes) ทำงานเหมือนกัน คนส่วนใหญ่ใช้อัญประกาศเดี่ยว ยกเว้นในกรณีอย่างเช่นการมีเครื่องหมายอัญประกาศเดี่ยวตัวเดียว (apostrophe) ในสายอักขระ

อัญประกาศทุกแบบจะต้องเป็นเครื่องหมายที่เป็น “เส้นตรง ๆ” ซึ่งโดยปกติแล้วจะอยู่ถัดจากปุ่ม Enter บนคีย์บอร์ด ส่วนเครื่องหมายอัญประกาศ “แบบโค้ง” เช่นในประโยคนี้ จะผิดกฎของไพธอน

ถ้าเราพิมพ์นิยามของฟังก์ชันในโหมดโต้ตอบ ตัวแปลภาษาไพธอนจะขึ้นจุดหลาย ๆ จุด (`...`) มาให้ เพื่อที่เราจะได้รู้ว่านิยามฟังก์ชันนั้นยังไม่สมบูรณ์ดี

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

ถ้าต้องการจบฟังก์ชัน ให้เรา enter บรรทัดเปล่า ๆ เข้าไปหนึ่งบรรทัด

การนิยามฟังก์ชันจะสร้าง **อ็อบเจกต์ฟังก์ชัน (function object)** ที่มีชนิดเป็น `function`:

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

กฎวากยสัมพันธ์ในการเรียกฟังก์ชันใหม่นั้นเหมือนกับการเรียกฟังก์ชันสำเร็จรูป (built-in function):

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

เมื่อเราได้นิยามฟังก์ชันแล้ว เราสามารถใช้มันในฟังก์ชันอื่นได้ด้วย เช่น ถ้าเราต้องการจะพิมพ์เนื้อเพลงบทที่แล้วอีกรอบ เราสามารถเขียนฟังก์ชันที่ชื่อว่า `repeat_lyrics`:

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

จากนั้น ให้เรียก `repeat_lyrics`:

```
>>> 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.
```

แต่จริง ๆ แล้วเนื้อเพลงมันไม่ได้เป็นแบบนั้นหรอกนะ

3.5. นิยามและการใช้งาน

เมื่อทำการรวบรวมโค้ดชิ้นต่าง ๆ จากเนื้อหาในส่วนที่ผ่านมา โปรแกรมที่สมบูรณ์ก็จะหน้าตาเป็นแบบนี้:

```
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()
```

โปรแกรมนี้ประกอบด้วยนิยามฟังก์ชัน 2 ตัว: `print_lyrics` และ `repeat_lyrics` นิยามฟังก์ชันทำงานเหมือนคำสั่งต่าง ๆ ทั่วไป แต่จะส่งผลให้เกิดการสร้างอ็อบเจกต์ฟังก์ชัน คำสั่งต่าง ๆ ที่อยู่ในฟังก์ชันจะไม่ทำงานจนกว่าฟังก์ชันนั้นจะถูกเรียก และตัวนิยามของฟังก์ชันเองจะไม่ทำให้เกิดผลลัพธ์ใด ๆ

(หมายเหตุผู้แปล: นั่นคือ จะต้องมีการเรียกฟังก์ชันก่อน ถึงจะมีการทำงานใด ๆ นั่นเอง)

เราอาจจะพอคาดได้ว่า เราจะต้องสร้างฟังก์ชันก่อนที่เราจะเรียกมันใช้งาน อีกนัยหนึ่งคือ นิยามของฟังก์ชันจะต้องถูกรันก่อนที่ฟังก์ชันนั้นจะถูกเรียกใช้งาน

เพื่อเป็นการฝึก ให้ลองย้ายบรรทัดสุดท้ายของโปรแกรมไปไว้ข้างบนสุด ซึ่งทำให้การเรียกฟังก์ชันนั้นเกิดก่อน การนิยามฟังก์ชัน ให้รันโปรแกรมและดูว่าไพธอนจะให้ข้อความแจ้งข้อผิดพลาดอะไรออกมา

คราวนี้ ให้ย้ายการเรียกฟังก์ชันกลับมาข้างล่างสุดเหมือนเดิม และย้ายนิยามของ `print_lyrics` ให้ไปอยู่หลังนิยามของ `repeat_lyrics` จะเกิดอะไรขึ้นเมื่อเรารันโปรแกรมนี้?

3.6. กระแสการดำเนินการ

เพื่อที่ให้แน่ใจว่าฟังก์ชันนั้นถูกนิยามก่อนการใช้งานครั้งแรก เราต้องรู้ลำดับการทำงานของคำสั่งก่อน ซึ่งเรียกว่า **กระแสการดำเนินการ (flow of execution)**

การดำเนินการของคำสั่งเริ่มจากคำสั่งแรกสุดของโปรแกรมเสมอ คำสั่งจะทำงานทีละคำสั่ง เรียงลำดับจากบนลงล่าง

การนิยามฟังก์ชันไม่ได้เปลี่ยนแปลงกระแสการดำเนินการ แต่ให้จำไว้ว่า คำสั่งที่อยู่ในฟังก์ชันนั้นจะไม่ทำงานจนกว่าฟังก์ชันนั้นจะถูกเรียกใช้

การเรียกใช้ฟังก์ชันเหมือนกับการเบี่ยงกระแสการดำเนินการ แทนที่โปรแกรมจะทำคำสั่งถัดไป การทำงานจะกระโดดไปที่ตัวของฟังก์ชัน ทำงานตามคำสั่งในฟังก์ชันนั้น และจากนั้นก็กลับมาทำต่อจากตรงที่มันกระโดดออกไป

ก็ฟังดูง่ายนะ จนกระทั่งเราจำได้ว่าฟังก์ชันหนึ่งสามารถเรียกอีกฟังก์ชันหนึ่งได้ด้วย เมื่ออยู่ระหว่าง การทำงานในฟังก์ชันหนึ่ง ๆ โปรแกรมอาจจะทำคำสั่งที่ต้องเรียกฟังก์ชันอื่น จากนั้นในขณะที่ทำงานใน ฟังก์ชันใหม่นั้น โปรแกรมก็อาจจะเรียกทำงานฟังก์ชันอันอื่นอีก!

ยังโชคดีที่ไพธอนนั้นเก่งในการติดตามว่าตอนนี้มันทำงานอยู่ที่ไหนแล้ว ทำให้แต่ละครั้งที่ฟังก์ชันทำงานเสร็จ โปรแกรมจะกลับไปทำงานค้างในฟังก์ชันที่เรียกฟังก์ชันที่เพิ่งทำงานเสร็จนี้ เมื่อไพธอนทำทุกคำสั่งในโปรแกรมเสร็จแล้ว มันก็จะหยุดทำงาน

โดยสรุปแล้ว เมื่อเราอ่านโปรแกรม เราไม่จำเป็นจะต้องอ่านจากบนลงล่างเสมอไป บางครั้งมันเข้าท่าที่จะอ่านตามกระแสการดำเนินการ (flow of execution)

3.7. พารามิเตอร์และอาร์กิวเมนต์

ฟังก์ชันบางอันที่เราได้เห็นมานั้นต้องการอาร์กิวเมนต์ เช่น เมื่อเราเรียกฟังก์ชัน `math.sin` เราจะต้องผ่านเลขเข้าไปเป็นอาร์กิวเมนต์ ฟังก์ชันบางตัวรับค่ามากกว่าหนึ่งอาร์กิวเมนต์: ฟังก์ชัน `math.pow` รับค่าเข้ามาสองตัว คือ เลขฐานและเลขยกกำลัง

ในฟังก์ชัน อาร์กิวเมนต์จะถูกกำหนดตัวแปรให้ เรียกว่า **พารามิเตอร์ (parameters)** นี่คือนิยามของฟังก์ชันที่รับอาร์กิวเมนต์เข้ามา

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

ฟังก์ชันนี้กำหนดให้อาร์กิวเมนต์ที่รับเข้ามาเป็นพารามิเตอร์ที่ชื่อว่า `bruce` เมื่อฟังก์ชันถูกเรียก มันจะพิมพ์ค่าของพารามิเตอร์ตัวนี้ (ไม่ว่ามันจะมีค่าอะไรก็ตาม) สองครั้ง

ฟังก์ชันนี้จะทำงานได้กับค่าอะไรก็ตามที่สามารถพิมพ์ออกมาได้

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

กฎของการประกอบ (rules of composition) อันเดียวกันกับที่ใช้กับฟังก์ชันสำเร็จรูป สามารถใช้ได้กับฟังก์ชันที่โปรแกรมเมอร์นิยามขึ้นมาเอง ดังนั้น เราสามารถใช้นิพจน์ใด ๆ เป็นอาร์กิวเมนต์สำหรับ ฟังก์ชัน `print_twice`:

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

อาร์กิวเมนต์จะถูกประเมินค่าก่อนที่ฟังก์ชันจะถูกเรียก ดังนั้น ในตัวอย่างนี้ นิพจน์ `'Spam '*4` และ `math.cos(math.pi)` จะถูกประเมินค่าแค่ครั้งเดียว

(หมายเหตุผู้แปล: หลังจากประเมินค่า โปรแกรมจะส่งค่าที่ประเมินแล้วเข้าไปทำงานในฟังก์ชัน)

เราสามารถใช้ตัวแปรเป็นอาร์กิวเมนต์ได้ด้วย:

```
>>> michael = 'Eric, the half a bee.'
```

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

ชื่อของตัวแปรที่เราผ่านเข้าไปเป็นอาร์กิวเมนต์ (`michael`) ไม่มีอะไรเกี่ยวข้องกับชื่อของพารามิเตอร์ (`bruce`) มันไม่เกี่ยวว่าค่าหนึ่ง ๆ ถูกเรียกว่าอะไรในที่เดิม (ในส่วนที่เรียกฟังก์ชัน); ณ ที่นี่ ในฟังก์ชัน `print_twice` เราเรียกทุกตัว (ที่ถูกส่งเข้ามา) ว่า `bruce`

3.8. ตัวแปรและพารามิเตอร์เป็นค่าเฉพาะที่

เมื่อเราสร้างตัวแปรข้างในฟังก์ชัน มันจะมีค่า **เฉพาะที่ (local)** ซึ่งหมายความว่ามันจะมีตัวตนแค่ในฟังก์ชันนี้เท่านั้น เช่น:

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

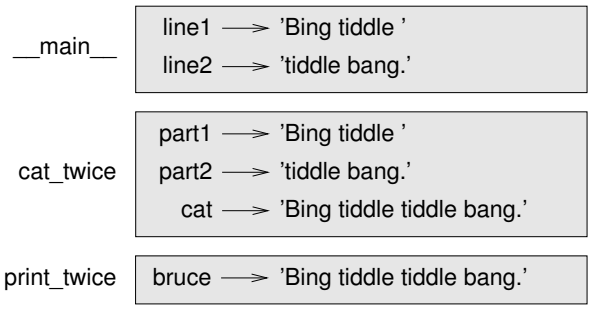
ฟังก์ชันนี้รับอาร์กิวเมนต์เข้ามาสองตัว เชื่อมต่อมันเข้าด้วยกัน และพิมพ์ผลลัพธ์สองครั้ง นี่คือตัวอย่างที่ใช้ฟังก์ชันนี้:

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

เมื่อฟังก์ชัน `cat_twice` จบการทำงาน ตัวแปร `cat` จะถูกทำลาย ถ้าเราพยายามที่จะพิมพ์มันออกมา เราจะได้เอ็กเซ็ปชั่น (ข้อผิดพลาดตอนโปรแกรมทำงาน):

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

พารามิเตอร์ก็เป็นค่าเฉพาะที่เหมือนกัน เช่น ภายนอกฟังก์ชัน `print_twice` มันไม่มีสิ่งที่เรียกว่า `bruce`



รูปที่ 3.1.: แผนภาพแบบกองซ้อน (Stack diagram)

3.9. แผนภาพแบบกองซ้อน

เพื่อที่จะติดตามว่าตัวแปรไหนใช้ที่ไหนได้บ้าง บางครั้งมันก็เป็นประโยชน์ที่จะวาด **แผนภาพแบบกองซ้อน (stack diagram)** เช่นเดียวกับแผนภาพสถานะ แผนภาพแบบกองซ้อนแสดงค่าของแต่ละตัวแปร แต่มันยังแสดงว่าตัวแปรนั้นเป็นของฟังก์ชันไหน

แต่ละฟังก์ชันถูกแสดงด้วย **กรอบ (frame)** กรอบเป็นกล่องที่มีชื่อฟังก์ชันอยู่ข้าง ๆ และมีพารามิเตอร์และตัวแปรของฟังก์ชันนั้นอยู่ข้างใน แผนภาพแบบกองซ้อนสำหรับตัวอย่างที่แล้วแสดงอยู่ในรูปที่ 3.1

กรอบต่าง ๆ ถูกจัดวางเป็นกอง ๆ ซ้อนกัน โดยระบุว่าฟังก์ชันไหนเรียกฟังก์ชันไหน ไปเรื่อย ๆ ในตัวอย่างนี้ ฟังก์ชัน `print_twice` ถูกเรียกโดยฟังก์ชัน `cat_twice` และ ฟังก์ชัน `cat_twice` ถูกเรียกโดยฟังก์ชัน `__main__` ซึ่งเป็นชื่อพิเศษสำหรับเรียกกรอบที่อยู่บนสุดของโปรแกรม เมื่อเราสร้างตัวแปรภายนอกฟังก์ชันอะไรก็ตาม มันจะถือว่าเป็นตัวแปรของส่วน `__main__`

พารามิเตอร์แต่ละตัวอ้างอิงถึงค่าเดียวกันกับค่าของอาร์กิวเมนต์ที่ผ่านเข้ามา ณ ตำแหน่งเดียวกัน ดังนั้น `part1` มีค่าเดียวกันกับ `line1`, `part2` มีค่าเดียวกันกับ `line2`, และ `bruce` มีค่าเดียวกันกับ `cat`

ถ้ามีข้อผิดพลาดเกิดขึ้นระหว่างที่มีการเรียกฟังก์ชัน ไพธอนจะพิมพ์ชื่อฟังก์ชันที่มีปัญหา ชื่อฟังก์ชันที่เรียกฟังก์ชันที่มีปัญหา และชื่อฟังก์ชันที่เรียกฟังก์ชัน *เมื่อกี้นี้* และย้อนกลับไปเรื่อย ๆ จนถึง `__main__`

ตัวอย่างเช่น ถ้าเราพยายามที่จะเข้าถึง `cat` จากในฟังก์ชัน `print_twice` เราจะได้ ข้อผิดพลาดแบบทางชื่อ หรือ `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
```

```
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

รายชื่อฟังก์ชันเหล่านี้เรียกว่า **การย้อนรอย (traceback)** มันบอกเราว่าโปรแกรมผิดพลาดที่ไฟล์ใด บรรทัดใด และฟังก์ชันอะไรที่ทำงานอยู่ตอนนั้น มันยังแสดงเลขบรรทัดของโค้ดที่เป็นเหตุของข้อผิดพลาดอีกด้วย

ลำดับของฟังก์ชันต่าง ๆ ในการย้อนรอยเป็นลำดับเดียวกับลำดับของกรอบในแผนภาพแบบกองซ้อน ฟังก์ชันที่กำลังทำงานอยู่จะอยู่ข้างล่างสุด

3.10. ฟังก์ชันที่ให้ผลและฟังก์ชันที่ไม่ให้ผล

บางฟังก์ชันที่เราใช้กันมา เช่น ฟังก์ชันทางคณิตศาสตร์ มีการคืนผลลัพธ์กลับมาให้เรา เนื่องจากไม่มีชื่อที่ดีกว่านี้ ผมจะเรียกพวกมันว่า **ฟังก์ชันที่ให้ผล (fruitful functions)** ส่วนฟังก์ชันอื่น ๆ จำพวกฟังก์ชัน `print_twice` จะทำงานต่าง ๆ โดยไม่มีการคืนค่ากลับมา พวกมันเรียกว่า **ฟังก์ชันที่ไม่ให้ผล หรือ ฟังก์ชันวอยด์ (void functions)**

เมื่อเราเรียกฟังก์ชันที่ให้ผล เราต้องทำอะไรสักอย่างกับผลที่ได้คืนมานั้นเกือบจะเสมอ เช่น เราอาจจะกำหนดค่าของมันให้กับตัวแปร หรือใช้เป็นส่วนหนึ่งของนิพจน์:

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

เมื่อเราเรียกฟังก์ชันในโหมดโต้ตอบ ไพธอนจะแสดงผลออกมา

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

แต่ในโหมดสคริปต์ ถ้าเราเรียกฟังก์ชันที่มีผลแบบลอย ๆ ค่าผลลัพธ์ที่คืนกลับมาจะหายไปเลยชั่วนิรันดร์!

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

สคริปต์นี้คำนวณรากที่สองของ 5 แต่เนื่องจากมันไม่ได้เก็บค่าที่คืนกลับมาหรือแสดงค่าออกมา มันก็ไม่มีประโยชน์อะไรเท่าไร

ฟังก์ชันวอยด์อาจจะแสดงอะไรสักอย่างบนหน้าจอ หรือมีผลอย่างอื่น แต่มันไม่*ส่งคืน*ค่าใด ๆ กลับออกมา ถ้าเรากำหนดค่าที่คืนกลับมาจากฟังก์ชันวอยด์ให้กับตัวแปร เราจะได้ค่าพิเศษที่เรียกว่า None

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

ค่า None ไม่ใช่ค่าเดียวกับสายอักขระ 'None' แต่มันเป็นค่าพิเศษที่มีชนิดของข้อมูลเป็นของมันเอง:

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

ฟังก์ชันที่เราเขียนกันมาถึงตอนนี้เป็นฟังก์ชันวอยด์ทั้งหมด เราจะเริ่มเขียนฟังก์ชันที่ให้ผลให้อีกสองสามบทถัดไป

3.11. ทำไมต้องใช้ฟังก์ชัน

มันอาจจะไม่ชัดเจนเท่าไรว่าทำไมมันถึงคุ้มค่ากับการปวดหัวเพื่อที่จะแบ่งโปรแกรมเป็นฟังก์ชันต่าง ๆ มันมีเหตุผลหลายประการ ดังนี้:

- การสร้างฟังก์ชันใหม่สร้างโอกาสให้เราตั้งชื่อให้กลุ่มของคำสั่งได้ ทำให้โปรแกรมของเรานั้นอ่านและดีบักง่าย
- ฟังก์ชันต่าง ๆ ทำให้โปรแกรมเล็กลงโดยการกำจัดโค้ดที่ซ้ำซ้อน ในคราวหลัง ถ้าเราเปลี่ยนแปลงโค้ดพวกนั้น เราก็เพียงเปลี่ยนในที่เดียวเท่านั้น
- การแบ่งโปรแกรมยาว ๆ ให้เป็นฟังก์ชันต่าง ๆ ทำให้เราดีบักได้ทีละส่วน จากนั้นเราก็ รวมส่วนต่าง ๆ เข้าด้วยกันให้มันทำงานได้แบบเต็มโปรแกรม
- ฟังก์ชันที่ถูกออกแบบมาเป็นอย่างดีนั้นบ่อยครั้งจะมีประโยชน์กับหลายโปรแกรม เมื่อเราเขียนและดีบักครั้งหนึ่งแล้ว เราสามารถใช้มันซ้ำได้ (ในโปรแกรมอื่น ๆ ด้วย)

3.12. การดีบัก

ทักษะที่สำคัญมากที่สุดอย่างหนึ่งที่เราจะได้เรียนรู้ คือ การตรวจแก้จุดบกพร่อง หรือการดีบัก (debugging) ถึงแม้ว่ามันจะน่าท้อแท้อยู่สักหน่อย แต่การดีบักเป็นสิ่งที่ประเทืองปัญญา ท้าทาย และเป็นส่วนที่น่าสนใจของการเขียนโปรแกรม

ในบางแง่มุม การดีบักเหมือนกับงานสืบสวน เราจะเจอเบาะแสและเราจะต้องวินิจฉัยกระบวนการ และเหตุการณ์ที่ทำให้เกิดผลลัพธ์แบบที่เราเห็นอยู่นี้

(หมายเหตุผู้แปล: คิดว่าผลนี้มันเกิดมาจากไหน ได้อย่างไร)

การดีบักยังเป็นเหมือนกับการทดลองทางวิทยาศาสตร์ เมื่อเรามีความคิดว่าอะไรที่น่าจะผิดพลาด เราจะแก้โปรแกรม และลองดูอีก ถ้าสมมติฐานของเราถูก เราจะสามารถทำนายผลของการแก้ไขนั้นได้ และเราก็ได้ก้าวเข้าใกล้ โปรแกรมที่สามารถทำงานได้อีกก้าวหนึ่งแล้ว ถ้าสมมติฐานของเราผิด เราก็ต้องคิดสมมติฐานขึ้นมาใหม่ เป็นอย่างที่เชอร์ล็อค โฮล์มส์ชี้ให้เห็นว่า “เมื่อเรากำจัดความเป็นไปไม่ได้แล้ว, อะไรก็ตามที่เหลือ, ไม่ว่ามันจะ ดูเป็นไปไม่ได้ขนาดไหน, ก็จะต้องเป็นความจริง“ (A. Conan Doyle, *The Sign of Four*)

สำหรับบางคนแล้ว การเขียนโปรแกรมและการดีบักเป็นอย่างเดียวกัน นั่นคือ การเขียนโปรแกรมเป็นกระบวนการของการค่อย ๆ ดีบักโปรแกรมจนกว่ามันจะทำสิ่งที่เราต้องการได้ หลักการคือ เราควรจะเริ่มจากโปรแกรมเล็ก ๆ ที่ทำงานได้ และแก้ไขไปทีละน้อย นั่นคือการดีบักไปพร้อมกับการเขียนโปรแกรม

(หมายเหตุผู้แปล: เขียนเพิ่มและดีบักเพิ่มทีละเล็กทีละน้อย)

ตัวอย่างเช่น ลินุกซ์ (Linux) เป็นระบบปฏิบัติการที่มีโค้ดเป็นล้าน ๆ บรรทัด แต่มันเริ่มมาจากโปรแกรมเล็ก ๆ ที่ ลินัส ทอร์วัลด์ส (Linus Torvalds) ใช้สำรวจชิป Intel 80386 จากคำบอกเล่าของ ลาร์รี กรีนฟิลด์ (Larry Greenfield) “โครงงานแรก ๆ ของลินัส คือ โปรแกรมที่จะสับเปลี่ยนระหว่างการพิมพ์ AAAA และ BBBB โปรแกรมนี้ได้วิวัฒนาการมาเป็นลินุกซ์” (*The Linux Users' Guide* Beta Version 1)

3.13. อภิธานศัพท์

ฟังก์ชัน (function): ลำดับของคำสั่งที่มีชื่อเรียก ซึ่งทำงานอะไรสักอย่างที่มีประโยชน์ ฟังก์ชันอาจจะรับหรือไม่รับอาร์กิวเมนต์ และอาจจะมีผลลัพธ์หรือไม่ก็ได้

นิยามฟังก์ชัน (function definition): คำสั่งที่ใช้สร้างฟังก์ชันใหม่ โดยระบุชื่อ พารามิเตอร์ และคำสั่งที่อยู่ในฟังก์ชัน

อ็อบเจกต์ฟังก์ชัน (function object): ค่าที่ถูกสร้างโดยนิยามฟังก์ชัน ชื่อฟังก์ชันคือตัวแปรที่อ้างอิงถึงอ็อบเจกต์ฟังก์ชัน

ส่วนหัว (header): บรรทัดแรกของนิยามฟังก์ชัน

ส่วนตัว (body): กลุ่มลำดับของคำสั่งที่อยู่ในนิยามฟังก์ชัน

พารามิเตอร์ (parameter): ชื่อที่ถูกใช้ในฟังก์ชันเพื่ออ้างอิงถึงค่าที่ถูกผ่านเข้ามาเป็นอาร์กิวเมนต์

การเรียกฟังก์ชัน (function call): คำสั่งที่ทำให้ฟังก์ชันทำงาน ประกอบด้วย ชื่อฟังก์ชัน ตามด้วยรายการ อาร์กิวเมนต์ในวงเล็บ

อาร์กิวเมนต์ (argument): ค่าที่ส่งผ่านไปให้ฟังก์ชันเมื่อฟังก์ชันถูกเรียกใช้งาน ค่านี้ถูกกำหนดให้กับพารามิเตอร์ที่ ตรงตำแหน่งกันในฟังก์ชัน

ตัวแปรเฉพาะที่ (local variable): ตัวแปรที่ถูกนิยามขึ้นภายในฟังก์ชัน ตัวแปรเฉพาะที่สามารถถูกใช้งานได้ ภายในฟังก์ชันของมันเท่านั้น

ค่าคืนกลับ (return value): ผลลัพธ์ของฟังก์ชัน ถ้าการเรียกฟังก์ชันนั้นถูกใช้เป็นนิพจน์ ค่าคืนกลับจะกลายมาเป็นค่าของนิพจน์นั้นเลย

ฟังก์ชันที่ให้ผล (fruitful function): ฟังก์ชันที่คืนค่ากลับมา

ฟังก์ชันที่ไม่ให้ผล หรือ ฟังก์ชันวอยด์ (void function): ฟังก์ชันที่คืนค่ากลับมาเป็น `None` เสมอ

None: ค่าพิเศษที่ถูกคืนกลับมาโดยฟังก์ชันวอยด์

มอดูล (module): ไฟล์ที่บรรจุฟังก์ชันและนิยามต่าง ๆ ที่เกี่ยวข้องกัน

คำสั่งนำเข้า (import statement): คำสั่งที่อ่านไฟล์มอดูลและสร้างอ็อบเจกต์มอดูล

อ็อบเจกต์มอดูล (module object): ค่าที่ถูกสร้างขึ้นโดยคำสั่ง `import` ที่เอื้อให้สามารถเข้าถึงค่าที่ถูกนิยามในมอดูล

สัญกรณ์จุด (dot notation): กฎวากยสัมพันธ์สำหรับเรียกฟังก์ชันในมอดูลอื่น ซึ่งให้ระบุชื่อของมอดูลตามด้วยเครื่องหมายจุดและชื่อฟังก์ชันที่ต้องการเรียก

การประกอบ (composition): การใช้นิพจน์เป็นส่วนประกอบของนิพจน์อีกอันที่ใหญ่กว่า หรือคำสั่งที่เป็นส่วนประกอบของอีกคำสั่งที่ใหญ่กว่า

กระแสการดำเนินการ (flow of execution): ลำดับการทำงานของคำสั่งต่าง ๆ

แผนภาพแบบกองซ้อน (stack diagram): การใช้รูปภาพแสดงกองของฟังก์ชัน ตัวแปรของแต่ละฟังก์ชัน และค่าที่ถูกอ้างถึง

กรอบ (frame): กล่องในแผนภาพแบบกองซ้อนที่แสดงการเรียกฟังก์ชันหนึ่ง ๆ ซึ่งประกอบด้วยตัวแปรเฉพาะที่ และพารามิเตอร์ของฟังก์ชัน

การย้อนรอย (traceback): รายการของฟังก์ชันต่าง ๆ ที่ทำงานอยู่ โดยจะถูกพิมพ์ออกมาเมื่อเกิดเอ็กเซ็ปชั่น (ความผิดพลาดตอนโปรแกรมทำงาน)

3.14. แบบฝึกหัด

แบบฝึกหัด **3.1.** *ให้เขียนฟังก์ชันที่ชื่อว่า* `right_justify` *ซึ่งรับสายอักขระที่ชื่อว่า* `s` *เป็นพารามิเตอร์ และพิมพ์สายอักขระดังกล่าวโดยมีช่องว่างนำหน้า แล้วทำให้อักษรตัวสุดท้ายของสายอักขระอยู่ในหลักที่ 70 ของการแสดงผล*

```
>>> right_justify('monty')
                                                                monty
```

คำใบ้: ใช้การเชื่อมต่อสายอักขระและการทำซ้ำ นอกจากนี้ ไพธอนได้เตรียมฟังก์ชันที่ชื่อว่า `len` *ซึ่งจะคืนค่าความยาวของสายอักขระ ดังนั้น ค่าของ* `len('monty')` *คือ 5*

แบบฝึกหัด **3.2.** *อ็อบเจกต์ฟังก์ชัน คือ ค่าที่เราสามารถกำหนดให้กับตัวแปร หรือผ่านเข้าไปเป็นอาร์กิวเมนต์ เช่น ฟังก์ชัน* `do_twice` *เป็นฟังก์ชันที่รับอ็อบเจกต์ฟังก์ชันเป็นอาร์กิวเมนต์ และเรียกฟังก์ชันนั้นสองรอบ (ฟังก์ชัน f() จะทำงาน 2 ครั้ง)*

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

นี่คือตัวอย่างที่ใช้ฟังก์ชัน `do_twice` *เพื่อเรียกฟังก์ชัน* `print_spam` *ให้ทำงานสองครั้ง*

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


do_twice(print_spam)
```

1. *ให้เขียนตัวอย่างนี้ลงในสคริปต์ และทดสอบโปรแกรมดู*
2. *ให้แก้ไขฟังก์ชัน* `do_twice` *ให้รับอาร์กิวเมนต์เข้ามา 2 ตัว เป็นอ็อบเจกต์ฟังก์ชัน 1 ตัว และค่า 1 ค่า ให้เรียกฟังก์ชันที่รับเข้ามา 2 รอบ โดยผ่านอีกค่าที่รับเข้ามาเป็นอาร์กิวเมนต์ของฟังก์ชันดังกล่าว*
3. *ให้คัดลอกนิยามของฟังก์ชัน* `print_twice` *จากก่อนหน้านี้ในบทนี้แล้วใส่ในสคริปต์ของเรา*
4. *ให้ใช้ฟังก์ชัน* `do_twice` *ที่แก้ไขไปก่อนหน้านี้เพื่อเรียกฟังก์ชัน* `print_twice` *สองครั้ง โดยให้ผ่าน* `'spam'` *เป็นอาร์กิวเมนต์*
5. *ให้นิยามฟังก์ชันใหม่ที่ชื่อว่า* `do_four` *ซึ่งรับอ็อบเจกต์ฟังก์ชัน 1 ตัว และค่า 1 ค่า และเรียกฟังก์ชันที่รับเข้ามา 4 รอบโดยผ่านอีกค่าที่รับเข้ามาเป็นพารามิเตอร์ จะต้องมีคำสั่งแค่ 2 คำสั่งเท่านั้นในตัวฟังก์ชันนี้ ไม่ใช่ 4 คำสั่ง*

เฉลย: `http://thinkpython2.com/code/do_four.py`.

แบบฝึกหัด 3.3. *หมายเหตุ: ข้อนี้จะต้องใช้คำสั่งและลักษณะเฉพาะที่เราเรียนมาจนถึงตอนนี้เท่านั้น*

1. *ให้เขียนฟังก์ชันที่วาดรูปตารางดังต่อไปนี้*

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

คำใบ้: ในการพิมพ์ค่ามากกว่า 1 ค่าบนหนึ่งบรรทัด เราสามารถพิมพ์ค่าหลายค่าต่อกันโดยคั่นด้วยเครื่องหมายจุลภาค (,):

```
print('+', '-')
```

คำสั่ง `print` *ขึ้นบรรทัดใหม่โดยปริยาย แต่เราสามารถยกเลิกพฤติกรรมนี้ และใส่ช่องว่างในตอนท้ายของการพิมพ์ได้ โดยทำแบบนี้:*

```
print('+', end=' ')
print('-')
```

ผลของคำสั่งเหล่านี้ คือ `'+ -'`

คำสั่ง `print` *ที่ไม่มีอาร์กิวเมนต์จะจบบรรทัดปัจจุบันและขึ้นบรรทัดใหม่*

2. *ให้เขียนฟังก์ชันที่วาดรูปตารางแบบรูปที่แล้ว แต่ให้มี 4 แถวและ 4 หลัก*

เฉลย: `http://thinkpython2.com/code/grid.py`. *ที่มา: แบบฝึกหัดข้อนี้ดัดแปลงมาจากแบบฝึกหัดใน Oualline,* Practical C Programming, Third Edition*, O'Reilly Media, 1997.*

4. กรณีศึกษา การออกแบบส่วนต่อประสานงาน

บทนี้นำเสนอกรณีศึกษาที่สาธิตขั้นตอนการออกแบบฟังก์ชันต่าง ๆ ที่ทำงานร่วมกัน

บทนี้แนะนำมอดูล `turtle` ที่ทำให้เราสามารถสร้างรูปภาพโดยใช้กราฟิกเต่า (turtle graphics) มอดูล `turtle` มากับการติดตั้งภาษาไพธอนส่วนมากอยู่แล้ว แต่ถ้าเราใช้ไพธอนของ PythonAnywhere เราจะไม่สามารถรันตัวอย่างในมอดูล turtle ได้ (อย่างน้อยก็ยังไม่ได้ในตอนที่ผมเขียนหนังสือเล่มนี้)

ถ้าเราลงไพธอนบนคอมพิวเตอร์ของเราแล้ว เราจะสามารถรันตัวอย่างในมอดูล turtle ได้ ไม่เช่นนั้น ตอนนี้ก็เป็นเวลาที่ดีนะที่จะติดตั้งไพธอน ผมได้โพสต์ขั้นตอนการติดตั้งไว้ที่ `http://tinyurl.com/thinkpython2e`

ตัวอย่างของโค้ดที่ใช้ในบทนี้ อยู่ที่ `http://thinkpython2.com/code/polygon.py`

หมายเหตุผู้แปล: ในบทนี้จะใช้คำว่า turtle แทนคำว่า เต่า เพราะเป็นชื่อเฉพาะ

4.1. มอดูล turtle

ในการตรวจสอบว่าเรามีมอดูล `turtle` ในเครื่องหรือเปล่า ให้เปิดตัวแปลภาษาไพธอนขึ้นมาแล้วพิมพ์

```
>>> import turtle
>>> bob = turtle.Turtle()
```

เมื่อเรารันโค้ดนี้แล้ว ไพธอนควรจะสร้างหน้าต่างใหม่ขึ้นมาพร้อมกับลูกศรเล็ก ๆ ที่เป็นตัวแทนของ turtle ถ้าได้แล้วก็ปิดหน้าต่างได้

ให้สร้างไฟล์ชื่อว่า `mypolygon.py` และพิมพ์โค้ดต่อไปนี้:

```
import turtle
bob = turtle.Turtle()
```

```
print(bob)
turtle.mainloop()
```

มอดูล `turtle` (เขียนด้วย 't' ตัวเล็ก) ได้เตรียมฟังก์ชันชื่อว่า `Turtle` (เขียนด้วย 'T' ตัวใหญ่) ซึ่งสร้างอ็อบเจกต์ Turtle ที่เรากำหนดให้กับตัวแปรที่ชื่อว่า `bob` การพิมพ์ `bob` ออกมาจะแสดงผลประมาณนี้:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

นี่หมายความว่า `bob` อ้างอิงถึงอ็อบเจกต์ชนิด `Turtle` อย่างที่ถูกนิยามในมอดูล `turtle` นั่นเอง

ฟังก์ชัน `mainloop` สั่งให้หน้าต่างที่เปิดขึ้นมารอให้ผู้ใช้ทำอะไรสักอย่าง แม้ว่าในกรณีนี้ มันไม่มีอะไร ให้ผู้ใช้ทำมากนัก ยกเว้นการปิดหน้าต่าง

เมื่อเราสร้างอ็อบเจกต์ชนิด Turtle แล้ว เราสามารถเรียก **เมธอด (method)** เพื่อจะเลื่อนมันไปรอบ ๆ หน้าต่างได้ เมธอดเป็นเหมือนฟังก์ชัน แต่มันใช้กฎวากยสัมพันธ์ในการเขียนที่ต่างไปนิดหน่อย เช่น เมื่อเราต้องการจะเลื่อนเต่าไปข้างหน้า:

```
bob.fd(100)
```

เมธอด `fd` มีความเชื่อมโยงกับอ็อบเจกต์ turtle ที่เรียกว่า `bob` การเรียกใช้งานเมธอดเหมือนกับการขอร้อง: เราขอให้ `bob` เคลื่อนที่ไปข้างหน้า

อาร์กิวเมนต์ของเมธอด `fd` คือ ระยะทางในหน่วยพิกเซล (pixel) ดังนั้น ขนาดของการเคลื่อนที่จะขึ้นอยู่กับจอของแต่ละคน

เมธอดอื่นที่เราสามารถเรียกใช้กับ Turtle ได้ คือ `bk` เพื่อที่จะเคลื่อนไปข้างหลัง, `lt` เพื่อเลี้ยวซ้าย, และ `rt` เพื่อเลี้ยวขวา อาร์กิวเมนต์ของ `lt` และ `rt` เป็นขนาดของมุมในหน่วยองศา (degree)

นอกจากนี้ Turtle แต่ละตัวจะถือปากกาของมันเอง ซึ่งจะจรดลง (down) หรือ ยกขึ้น (up); ถ้า Turtle จรดปากกาลงมันจะทิ้งรอยขีดไว้เมื่อมันเคลื่อนที่ เมธอด `pu` และ `pd` เป็นตัวย่อของ "pen up' (ยกปากกาขึ้น) และ "pen down" (จรดปากกาลง)

เพื่อที่จะวาดมุมที่ถูกต้อง ให้เพิ่มบรรทัดเหล่านี้เข้าไปในโปรแกรม หลังจากสร้าง `bob` แล้ว แต่ใส่ก่อนที่จะเรียกฟังก์ชัน `mainloop`:

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

เมื่อเรารันโปรแกรมนี้ เราควรจะเห็น bob เคลื่อนที่ไปทางทิศตะวันออก และจากนั้นไปทางทิศเหนือ โดยที่ทิ้งเส้นไว้สองส่วน

คราวนี้ ให้แก้ไขโปรแกรมเพื่อจะวาดรูปสี่เหลี่ยมจตุรัส อย่าเพิ่งทำอย่างอื่นต่อจนกว่าจะวาดได้!

4.2. การทำซ้ำแบบง่าย ๆ

เราอาจจะเขียนโปรแกรมเมื่อกี้ออกมาประมาณนี้:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

เราสามารถทำสิ่งที่เหมือนกันแต่กระชับกว่า โดยการใช้คำสั่ง `for` ให้เพิ่มตัวอย่างนี้เข้าไปในไฟล์ `mypolygon.py` แล้วลองรันดูอีกที

```
for i in range(4):
    print('Hello!')
```

แล้วเราควรจะเห็นผลประมาณนี้:

```
Hello!
Hello!
Hello!
Hello!
```

นี่เป็นการใช้งานที่ง่ายที่สุดของคำสั่ง `for`; เราจะเห็นตัวอย่างอีกมากในภายหลัง แต่ตัวอย่างนี้ก็น่าจะเพียงพอสำหรับการให้เราเขียนโปรแกรมวาดรูปสี่เหลี่ยมใหม่ อย่าเพิ่งทำอย่างอื่นต่อจนกว่าจะทำได้นะ

นี่เป็นคำสั่ง `for` ที่วาดรูปสี่เหลี่ยมจตุรัส:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

กฎวากยสัมพันธ์ของคำสั่ง `for` นั้นเหมือนกับของฟังก์ชัน มันจะมีส่วนหัวที่ลงท้ายด้วยทวิภาค (:) และส่วนตัวที่ต้องย่อหน้าเข้าไป ส่วนตัวของ for สามารถมีคำสั่งอยู่ภายในได้กี่คำสั่งก็ได้

คำสั่ง `for` นี้สามารถเรียกว่า **ลูป (loop)** ได้ด้วย เนื่องจากกระแสการดำเนินการ (flow of execution) ทำงานผ่านไปตามส่วนตัวของคำสั่ง จากนั้นมันจะวนกลับไปทำคำสั่งบนสุดของคำสั่ง for ในกรณีนี้ โปรแกรมจะรันคำสั่งในส่วนตัวเป็นจำนวน 4 ครั้ง

โค้ดเวอร์ชันนี้แตกต่างจากโค้ดวาดรูปสี่เหลี่ยมจตุรัสอันที่แล้วนิดหน่อย เพราะว่ามันหันข้างหลังจากวาดเส้นสุดท้ายของสี่เหลี่ยมจตุรัสแล้ว การหันข้างที่เพิ่มขึ้นมานี้ใช้เวลามากขึ้น แต่ว่ามันทำให้โค้ดง่ายขึ้นถ้าเราจะทำ สิ่งที่เหมือน ๆ กันทุกครั้งที่ทำลูป โค้ดเวอร์ชันนี้ยังทำให้ turtle มันกลับไปที่จุดเริ่มต้น และหันหน้ากลับไปในทางเดียวกับตอนเริ่มต้นด้วย

4.3. แบบฝึกหัด

ต่อไปนี้จะเป็นชุดแบบฝึกหัดที่ใช้ TurtleWorld มันน่าจะสนุกแต่ก็มีประเด็นความรู้ด้วย ในระหว่างที่เราทำแบบฝึกหัดนี้ ให้คิดว่าประเด็นความรู้ที่ได้คืออะไร

หัวข้อต่อไปนี้มีเฉลยของแบบฝึกหัดด้วย ดังนั้น ห้ามไปดูเฉลยจนกว่าเราจะทำเสร็จ (หรืออย่างน้อยก็พยายามทำก่อน)

1. ให้เขียนฟังก์ชันที่ชื่อว่า `square` ซึ่งรับพารามิเตอร์ชื่อว่า `t` ที่เป็น turtle ฟังก์ชันนี้ควรจะใช้ turtle ในการวาดสี่เหลี่ยมจัตุรัส

   ให้เขียนการเรียกฟังก์ชันที่ผ่านค่า `bob` เข้าไปเป็นอาร์กิวเมนต์สำหรับฟังก์ชัน `square` จากนั้นให้รันโปรแกรมอีกทีหนึ่ง

2. ให้เพิ่มพารามิเตอร์อีกหนึ่งตัว ชื่อว่า `length` เข้าไปยัง `square` ทำการแก้ไขส่วนตัวของฟังก์ชันโดยให้ความยาวของด้านคือ `length` จากนั้นให้แก้ไขการเรียกฟังก์ชัน โดยให้ผ่านอาร์กิวเมนต์ตัวที่สองเข้าไปด้วย รันโปรแกรมอีกครั้งหนึ่ง ทดสอบโปรแกรมกับค่า `length` ในช่วงค่าต่าง ๆ

3. ให้คัดลอกฟังก์ชัน `square` แล้วเปลี่ยนชื่อเป็น `polygon` เพิ่มพารามิเตอร์อีกตัวชื่อว่า `n` ทำการแก้ไขส่วนตัวของฟังก์ชันให้วาดรูป n เหลี่ยมด้านเท่า (รูปหลายเหลี่ยมด้านเท่าที่มี n ด้าน) คำใบ้: มุมภายนอกของรูป n เหลี่ยมด้านเท่า คือ $360/n$ องศา

4. ให้เขียนฟังก์ชันที่ชื่อว่า `circle` ซึ่งรับ turtle `t`, และรัศมี `r` เป็นพารามิเตอร์ และวาดวงกลมกลาย ๆ โดยการเรียกฟังก์ชัน `polygon` ด้วยความยาวและจำนวนด้านที่เหมาะสม ทดสอบฟังก์ชันของเราด้วยช่วงค่า `r` ต่าง ๆ

   คำใบ้: ลองหาเส้นรอบวง (circumference) ของวงกลม และทำให้ `length * n = circumference`

5. ให้สร้างฟังก์ชัน `circle` ที่จะสามารถใช้ครอบคลุมกรณีอื่น ๆ ได้ โดยให้ชื่อว่า `arc` ซึ่งรับพารามิเตอร์อีกตัวหนึ่ง คือ `angle` ซึ่งกำหนดสัดส่วนของวงกลมที่เราจะวาด `angle` มีหน่วยเป็นองศา (degree) ดังนั้น เมื่อ `angle=360` ฟังก์ชัน `arc` ควรจะวาดวงกลมที่สมบูรณ์

4.4. การห่อหุ้ม

แบบฝึกหัดแรกให้เรานำโค้ดวาดรูปสี่เหลี่ยมจัตุรัสมาใส่ในนิยามฟังก์ชัน และจากนั้นเรียกฟังก์ชัน โดยผ่าน turtle เป็นพารามิเตอร์ นี่คือเฉลย:

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

คำสั่งชุดข้างในสุด (ย่อหน้าในสุด), `fd` และ `lt`, ถูกย่อหน้าเข้าไปสองรอบ เพื่อแสดงให้เห็นว่าพวกมันอยู่ใน `for` ลูป ซึ่งอยู่ในนิยามของฟังก์ชันอีกที บรรทัดถัดมา, `square(bob)`, ถูกล้างขอบซ้าย (เว้นบรรทัดและกลับเข้าไปชิดขอบซ้ายสุด) ซึ่งระบุถึงการสิ้นสุดของทั้งลูป `for` และนิยามฟังก์ชัน

ในฟังก์ชันนี้ ตัวแปร `t` อ้างอิงถึง turtle ตัวเดียวกับ `bob` ดังนั้น คำสั่ง `t.lt(90)` จะให้ผลเหมือนกับคำสั่ง `bob.lt(90)` ในกรณีนี้ แล้วทำไมเราไม่เรียกพารามิเตอร์ว่า `bob` ล่ะ? หลักการคือ `t` สามารถเป็น turtle ตัวไหนก็ได้ ไม่เฉพาะแค่ `bob` เท่านั้น ดังนั้น เราสามารถ สร้าง turtle ตัวที่สอง และผ่านมันเป็นอาร์กิวเมนต์เข้าไปยังฟังก์ชัน `square` ได้:

```
alice = turtle.Turtle()
square(alice)
```

การห่อชิ้นส่วนของโค้ดในฟังก์ชันนั้นเรียกว่า **การห่อหุ้ม (encapsulation)** ประโยชน์ของการห่อหุ้มอย่างหนึ่ง คือ มันจะผูกโค้ดเหล่านั้นไว้กับชื่อชื่อหนึ่ง ซึ่งทำหน้าที่คล้ายการบันทึกเอกสารด้วย ประโยชน์อีกอย่าง คือ การที่เราสามารถใช้โค้ดนี้ซ้ำได้ มันกระชับมากกว่าเมื่อเราเรียกฟังก์ชันซ้ำสองครั้ง แทนที่จะคัดลอกโค้ดและเอามาวาง (copy and paste)

4.5. การทำให้ครอบคลุม

ขั้นตอนถัดไป คือ การเพิ่มพารามิเตอร์ `length` ให้กับฟังก์ชัน `square` นี่คือเฉลย:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

การเพิ่มพารามิเตอร์ให้ฟังก์ชันนั้นเรียกว่า **การทำให้ครอบคลุม (generalization)** เพราะว่ามันทำให้ฟังก์ชันนั้นทำงานแบบครอบคลุมกรณีทั่วไปมากยิ่งขึ้น: ในเวอร์ชันก่อนหน้า สี่เหลี่ยมจตุรัสนั้นมีขนาดเดิมเสมอ; แต่ในเวอร์ชันนี้ มันสามารถเป็นขนาดอะไรก็ได้

ขั้นตอนถัดไปก็เป็นการทำให้ครอบคลุมเช่นกัน แทนที่เราจะวาดสี่เหลี่ยมจตุรัส ฟังก์ชัน `polygon` วาดรูปหลายเหลี่ยมด้านเท่า โดยมีจำนวนด้านเท่าไรก็ได้ นี่คือเฉลย:

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

ตัวอย่างนี้วาดรูป 7 เหลี่ยมที่มีด้านยาว 70 (หมายเหตุผู้แปล: หน่วยเป็นพิกเซล)

ถ้าเราใช้ไพธอน-2 ค่าของ `angle` อาจจะเพี้ยน ๆ หน่อย เพราะเป็นการหารของจำนวนเต็ม ทางแก้ง่าย ๆ คือการคำนวณเป็น `angle = 360.0 / n` เพราะว่าตัวเศษเป็นเลขโฟลตติ้งพอยต์ ค่าผลลัพธ์ที่ได้ก็จะเป็นเลขทศนิยม

(หมายเหตุผู้แปล: ในการทำ integer division นั้นจำนวนเต็มหารจำนวนเต็มจะได้ค่าผลลัพธ์เป็นจำนวนเต็ม)

เมื่อฟังก์ชันมีจำนวนอาร์กิวเมนต์มากกว่า 2-3 ตัว มันง่ายที่จะลืมว่าแต่ละตัวคืออะไร หรือลำดับในการวางควรเป็นอย่างไร ในกรณีดังกล่าว มันเป็นความคิดที่ดีที่จะใส่ชื่อของพารามิเตอร์ในลิสต์ของอาร์กิวเมนต์:

```
polygon(bob, n=7, length=70)
```

อาร์กิวเมนต์แบบนี้เรียกว่า **อาร์กิวเมนต์คำสำคัญ (keyword arguments)** เพราะว่ามันใส่ชื่อของพารามิเตอร์ลงไปเป็น "คำสำคัญ (keywords)" (อย่าสับสนกับคำสำคัญของไพธอน เช่น `while` และ `def`)

กฎวากยสัมพันธ์นี้ทำให้เราสามารถอ่านโปรแกรมได้ง่ายขึ้น มันยังเตือนให้เรารู้ว่าอาร์กิวเมนต์และพารามิเตอร์ทำงานอย่างไร: เมื่อเราเรียกฟังก์ชัน อาร์กิวเมนต์จะถูกกำหนดค่าให้กับพารามิเตอร์

4.6. การออกแบบส่วนต่อประสานงาน

ขั้นต่อไปคือการเขียนฟังก์ชัน `circle` ซึ่งรับรัศมี (radius) `r` เป็นพารามิเตอร์ นี่คือเฉลยง่าย ๆ ที่ใช้ฟังก์ชัน `polygon` ในการวาดรูป 50 เหลี่ยมด้านเท่า:

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

บรรทัดแรกคำนวณเส้นรอบวงของวงกลมที่มีรัศมี `r` โดยใช้สูตร $2\pi r$ เนื่องจากเราใช้ `math.pi` เราต้องนำเข้ามอดูล `math` ด้วย โดยปกติแล้ว คำสั่ง `import` จะอยู่ในตอนเริ่มต้นของไฟล์สคริปต์

`n` คือ จำนวนเส้นในการวาดวงกลมโดยประมาณของเรา และ `length` คือ ความยาวของแต่ละเส้นนั้น ดังนั้น ฟังก์ชัน `polygon` วาดรูปหลายเหลี่ยมจำนวน 50 เหลี่ยม เพื่อจะประมาณการเป็นวงกลมที่มีรัศมี `r`

ข้อจำกัดของเฉลยแบบนี้คือ n เป็นค่าคงที่ หมายถึงว่าสำหรับวงกลมอันใหญ่ ๆ แล้ว เส้นที่วาดนี้มันจะยาวเกินไป และสำหรับวงกลมเล็ก ๆ นี้ เราจะเสียเวลาวาดเส้นน้อย ๆ หลาย ๆ เส้น ทางแก้ทางหนึ่งคือการทำฟังก์ชันให้ครอบคลุม (generalize) โดยการเอา n มาเป็นพารามิเตอร์ มันจะทำให้ผู้ใช้ (หรือใครก็ตามที่เรียกฟังก์ชัน circle) สามารถควบคุม การใช้งานได้ดีขึ้น แต่ส่วนต่อประสานงาน (interface) จะเรียบร้อยน้อยลง

ส่วนต่อประสานงาน (interface) ของฟังก์ชัน คือ การสรุปว่าฟังก์ชันมันใช้งานอย่างไร: พารามิเตอร์คืออะไร? ฟังก์ชันทำงานอะไรบ้าง? และค่าที่ถูกส่งคืนกลับไปคืออะไร? ส่วนต่อประสานงานจะดู “เรียบร้อย” ถ้ามันทำให้โปรแกรมที่เรียกใช้ฟังก์ชันสามารถใช้งานที่ต้องการได้โดยไม่ต้องมายุ่งกับรายละเอียดที่ไม่จำเป็น

ในตัวอย่างนี้ r เป็นส่วนหนึ่งของส่วนต่อประสานงาน เพราะมันระบุคุณลักษณะของวงกลมที่จะต้องวาด ส่วน n ไม่ค่อยเหมาะเท่าไรเพราะมันเกี่ยวกับรายละเอียดของการที่จะแสดงผลว่าวงกลมจะเป็น *อย่างไร*

แทนที่จะทำให้ส่วนต่อประสานงานดูรก มันดีกว่าจะเลือกค่าที่เหมาะสมของ n โดยขึ้นอยู่กับค่าของ circumference

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 3
    length = circumference / n
    polygon(t, n, length)
```

คราวนี้ จำนวนของเส้น (segment) คือ จำนวนเต็มที่ใกล้เคียงกับ circumference/3 ทำให้ความยาวของเส้นมีค่าประมาณ 3 ซึ่งก็เล็กเพียงพอที่จะให้วงกลมนั้นดูดี แต่ใหญ่พอที่จะมีประสิทธิภาพ และเป็นที่ยอมรับได้สำหรับวงกลมขนาดใด ๆ

การเพิ่ม 3 ให้ n เป็นการรับรองว่ารูปหลายเหลี่ยมนี้จะมีอย่างน้อย 3 ด้าน

4.7. การปรับโครงสร้าง

ตอนที่ผมเขียนฟังก์ชัน circle ผมสามารถนำฟังก์ชัน polygon กลับมาใช้ได้ เนื่องจากรูปหลายเหลี่ยมนั้นเป็นการร่างรูปวงกลมได้ดี แต่ถ้านำมาใช้ในฟังก์ชัน arc มันไม่ค่อย ให้ความร่วมมือเท่าไร; เราไม่สามารถใช้ polygon หรือ circle มาวาดเส้นโค้งได้

ทางเลือกหนึ่งคือเริ่มด้วยสำเนาของฟังก์ชัน polygon และแปลงร่างให้มันเป็นฟังก์ชัน arc ผลที่ได้จะหน้าตาเป็นประมาณนี้:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

ครึ่งที่สองของฟังก์ชันนี้ดูเหมือนกับฟังก์ชัน polygon แต่เราไม่สามารถนำ polygon มาใช้ได้โดยไม่เปลี่ยนส่วนต่อประสานงาน เราอาจจะทำ polygon ให้ครอบคลุม โดยการรับขนาดของมุมมาเป็นอาร์กิวเมนต์ตัวที่สาม แต่ polygon มันจะไม่ใช่ชื่อที่เหมาะสมแล้ว! ถ้าอย่างนั้น เราลองเรียกแบบกว้าง ๆ ว่าเป็นฟังก์ชัน polyline ก็แล้วกัน:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

คราวนี้ เราสามารถเขียนฟังก์ชัน polygon และ ฟังก์ชัน arc ใหม่โดยใช้ polyline:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

ในที่สุด เราก็สามารถเขียน `circle` ใหม่โดยใช้ `arc`:

```
def circle(t, r):
    arc(t, r, 360)
```

กระบวนการเช่นนี้—การเรียงโปรแกรมใหม่เพื่อพัฒนาส่วนต่อประสานงานและอำนวยความสะดวกให้โค้ดถูกใช้งานซ้ำ ๆ ได้ เรียกว่า **การปรับโครงสร้าง (refactoring)** ในกรณีนี้ เราสังเกตว่า มีโค้ดที่เหมือนกันในฟังก์ชัน `arc` และ `polygon` ดังนั้นเราจึง "แยกส่วน" มันออกมา ใส่ในฟังก์ชัน `polyline`

ถ้าเราวางแผนล่วงหน้าสักหน่อย เราอาจจะเขียน `polyline` ก่อน และหลีกเลี่ยงการปรับโครงสร้าง แต่บ่อยครั้งที่เราก็ไม่มีข้อมูลเพียงพอในตอนเริ่มต้นของโปรเจ็กต์ ที่จะออกแบบส่วนต่อประสานงานทุกอย่าง เมื่อเราเริ่มต้นเขียนโค้ดแล้ว เราจะเข้าใจปัญหาต่าง ๆ ดีขึ้น ในบางครั้งการปรับโครงสร้างเป็นสัญญาณบอกว่า เราก็ได้เรียนรู้อะไรบ้างล่ะนะ

4.8. แผนการพัฒนา

แผนการพัฒนา (development plan) เป็นกระบวนการเขียนโปรแกรมอย่างหนึ่ง กระบวนการที่เราใช้ใน กรณีศึกษานี้ คือ "การห่อหุ้มและการทำให้ครอบคลุม (encapsulation and generalization)" ขั้นตอนของกระบวนการนี้คือ:

1. เริ่มด้วยการเขียนโปรแกรมเล็ก ๆ ที่ไม่มีนิยามฟังก์ชันก่อน
2. เมื่อเราได้โปรแกรมที่ทำงานได้แล้ว ให้ระบุชิ้นส่วนที่สอดคล้องกัน ห่อหุ้มส่วนนั้นเข้าเป็นฟังก์ชันและตั้งชื่อให้มัน
3. ทำฟังก์ชันให้ครอบคลุมการใช้งานโดยเพิ่มพารามิเตอร์ที่เหมาะสม
4. ทำขั้นตอนที่ 1–3 ซ้ำจนกว่าจะได้ชุดของฟังก์ชันที่ทำงานได้ คัดลอกและวางโค้ดที่ทำงานได้แล้วเพื่อที่จะหลีกเลี่ยงการพิมพ์โค้ดซ้ำอีก (และต้องมานั่งดีบักโปรแกรมซ้ำอีก)
5. หาโอกาสปรับปรุงโปรแกรมโดยการปรับโครงสร้าง เช่น ถ้าเรามีโค้ดที่เหมือนกันในหลาย ๆ ที่ ให้พิจารณาปรับมันเป็นฟังก์ชันที่ครอบคลุมการทำงานที่เหมาะสม

กระบวนการข้างต้นนี้มีข้อเสียบางอย่าง—เราจะเห็นทางเลือกอื่นทีหลัง—แต่มันก็เป็นประโยชน์ถ้าเราไม่รู้ล่วงหน้า ว่าจะแบ่งโปรแกรมเป็นฟังก์ชันต่าง ๆ ได้อย่างไร กระบวนการแบบนี้ทำให้เราออกแบบโปรแกรมควบคู่ไปกับการเขียนได้

4.9. ด็อกสตริง

A **ด็อกสตริง (docstring)** คือ สายอักขระที่อยู่ตอนเริ่มต้นของฟังก์ชัน ซึ่งอธิบายส่วนต่อประสานงานของฟังก์ชัน (“doc” เป็นคำเรียกสั้น ๆ ของ “documentation” หรือเอกสาร) นี่คือตัวอย่าง:

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

โดยปกตินิยมแล้ว ด็อกสตริงทุกอันจะเป็นสายอักขระที่อยู่ในอัญประกาศสามอัน (triple quotes) ซึ่งรู้จักกันในนาม สายอักขระหลายบรรทัด เพราะว่าอัญประกาศสามอันจะทำให้สายอักขระสามารถขยายความยาวได้มากกว่าหนึ่งบรรทัด

(หมายเหตุผู้แปล: string แปลเป็นภาษาไทยว่า สายอักขระ แต่เพื่อไม่ให้สับสนมาก เราคิดว่ามันเป็นข้อมูลที่เป็นข้อความที่ประกอบด้วยคำอาจจะหลายคำก็พอได้)

ด็อกสตริงเป็นสิ่งที่สั้น แต่มันก็มีข้อมูลที่สำคัญที่บางคนจำเป็นต้องรู้ในการใช้ฟังก์ชัน มันอธิบายแบบกระชับว่าฟังก์ชันทำอะไร (โดยที่เราไม่ต้องรู้รายละเอียดว่ามันทำอย่างไร) มันยังอธิบายว่าพารามิเตอร์แต่ละตัวมีผลกับฟังก์ชันอย่างไร และชนิดของพารามิเตอร์แต่ละตัวควรจะเป็นชนิดอะไร (ถ้ามันไม่ชัดเจนอยู่แล้ว)

การเขียนเอกสารแบบนี้เป็นส่วนสำคัญของการออกแบบส่วนต่อประสานงาน ส่วนต่อประสานงานที่ถูกออกแบบมาอย่างดี ควรจะง่ายต่อการอธิบาย; ถ้าเรามีความลำบากในการอธิบายฟังก์ชันหนึ่ง ๆ ของเราแล้ว บางทีส่วนต่อประสานงานนั้น อาจจะต้องได้รับการปรับปรุง

4.10. การดีบัก

ส่วนต่อประสานงานเหมือนกับสัญญาระหว่างฟังก์ชันกับตัวที่เรียกฟังก์ชัน ตัวเรียกฟังก์ชันตกลงที่จะผ่านค่าชนิดที่ตกลงกันไว้อย่างแน่นอนเข้ามาเป็นพารามิเตอร์ และฟังก์ชันก็ตกลงที่จะทำงานบางอย่างที่ตกลงกันไว้อย่างแน่นอนเช่นกัน

ตัวอย่างเช่น ฟังก์ชัน `polyline` ต้องการใช้อาร์กิวเมนต์ 4 ตัว: `t` จะต้องเป็นชนิด Turtle; `n` จะต้องเป็นจำนวนเต็ม; `length` จะต้องเป็นค่าบวก; และ `angle` จะต้องเป็นเลขใด ๆ ที่หน่วยเป็นองศา (degree)

สิ่งที่ฟังก์ชันต้องการเหล่านี้เรียกว่า **เงื่อนไขก่อน (preconditions)** เพราะว่าพวกมันจะต้องเป็นจริงก่อนที่ฟังก์ชันจะเริ่มทำงาน ในทางกลับกัน เงื่อนไขต่าง ๆ ในตอนจบของฟังก์ชันจะต้องเป็น **เงื่อนไขลงท้าย (postconditions)** เงื่อนไขลงท้ายรวมไปถึงผลการทำงานของฟังก์ชัน (เช่น การวาดรูปเส้นต่าง ๆ) และผลข้างเคียงต่าง ๆ (เช่น การเคลื่อนที่ของ Turtle และการเปลี่ยนแปลงค่าอย่างอื่น)

เงื่อนไขก่อนอยู่ในความรับผิดชอบของโปรแกรมที่เรียกฟังก์ชัน (caller) ถ้าโปรแกรมที่เรียกฟังก์ชันไม่สามารถทำตามเงื่อนไขก่อน (ที่ถูกอธิบายไว้อย่างเหมาะสม!) ได้ และทำให้ฟังก์ชันทำงานไม่ได้ ความผิดพลาดจะอยู่ที่ตัวโปรแกรมที่เรียกฟังก์ชัน ไม่ใช่ที่ตัวฟังก์ชัน

ถ้าเงื่อนไขก่อนนั้นถูกต้องแล้ว และเงื่อนไขลงท้ายไม่ถูกต้อง ความผิดพลาดนั้นจะอยู่ในฟังก์ชัน ถ้าเงื่อนไขทั้งเบื้องต้นและลงท้ายนั้นชัดเจน มันจะช่วยในการดีบักโปรแกรม

4.11. อภิธานศัพท์

เมธอด (method): ฟังก์ชันที่สัมพันธ์กับอ็อบเจกต์ และถูกเรียกโดยใช้สัญกรจุด

ลูป (loop): ส่วนของโปรแกรมที่สามารถทำงานซ้ำไปซ้ำมา

การห่อหุ้ม (encapsulation): กระบวนการแปลงลำดับของคำสั่งให้ไปอยู่ในนิยามของฟังก์ชัน

การทำให้ครอบคลุม (generalization): กระบวนการแทนที่บางอย่างที่เฉพาะเจาะจงโดยไม่จำเป็น (เช่น ตัวเลข) โดยบางอย่างที่ครอบคลุมมากกว่า (เช่น ตัวแปร หรือ พารามิเตอร์)

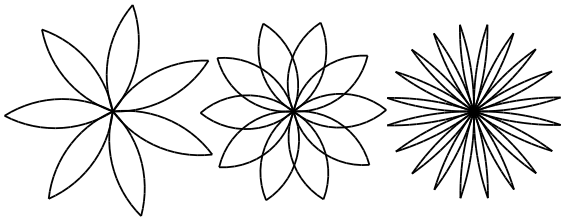
อาร์กิวเมนต์คำสำคัญ (keyword argument): อาร์กิวเมนต์ที่มีชื่อของพารามิเตอร์เป็นคำสำคัญ

ส่วนต่อประสานงาน (interface): คำอธิบายถึงการใช้งานฟังก์ชัน รวมถึงชื่อและการอธิบายอาร์กิวเมนต์ และค่าที่ถูกคืนกลับไป

การปรับโครงสร้าง (refactoring): กระบวนการแก้ไขโปรแกรมที่ทำงานได้แล้ว เพื่อพัฒนาส่วนต่อประสานงาน ของฟังก์ชันหรือคุณสมบัติอื่นของโค้ดให้ดีขึ้น

แผนการพัฒนา (development plan): กระบวนการเขียนโปรแกรมอย่างหนึ่ง

ด็อกสตริง (docstring): สายอักขระที่ปรากฏอยู่ส่วนบนสุดของนิยามฟังก์ชันเพื่อที่จะอธิบายเกี่ยวกับส่วนต่อ ประสานงานของฟังก์ชัน



รูปที่ 4.1.: ดอกไม้ของ Turtle (Turtle flowers.)

เงื่อนไขก่อน (precondition): โปรแกรมที่เรียกฟังก์ชัน (caller) จะต้องทำให้เป็นจริงก่อนที่ฟังก์ชันจะเริ่มทำงาน

เงื่อนไขลงท้าย (postcondition): สิ่งที่ฟังก์ชันจะต้องทำให้เป็นจริงก่อนที่ฟังก์ชันจะทำงานเสร็จ

4.12. แบบฝึกหัด

***แบบฝึกหัด* 4.1.** *ดาวน์โหลดโค้ดที่ใช้ในบทนี้ได้จาก* `http://thinkpython2.com/code/polygon.py`

1. *ให้วาดแผนภาพแบบกองซ้อนที่แสดงสถานะของโปรแกรมในขณะที่ทำงานในฟังก์ชัน* `circle(bob, radius)` *เราสามารถคำนวณโดยมือ หรือเพิ่มคำสั่ง* `print` *ในโค้ดก็ได้*

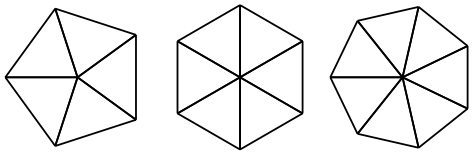
2. *เวอร์ชันของฟังก์ชัน* `arc` *ในหัวข้อที่ 4.7 ไม่ค่อยถูกเท่าไร เพราะการประมาณการ เชิงเส้นของวงกลมนั้นจะทำให้ประมาณไปเกินขอบนอกของวงกลมจริง ๆ เสมอ นั่นส่งผลให้ Turtle ไปจบที่ 2-3 พิกเซล เลยไปจากจุดหมายปลายทางที่ถูกต้อง เฉลยของผมได้แสดงให้เห็นถึงวิธีลดผลกระทบของความผิดพลาดนี้ ให้อ่านโค้ด และดูว่ามันดูเข้าท่าหรือไม่ ถ้าเราเขียนแผนภาพดู เราอาจจะเห็นว่ามันทำงานอย่างไร*

***แบบฝึกหัด* 4.2.** *ให้เขียนชุดของฟังก์ชันต่าง ๆ ให้ครอบคลุมการวาดดอกไม้ในรูป 4.1*

เฉลย: `http://thinkpython2.com/code/flower.py`*, ไฟล์ที่จำเป็นต้องใช้:* `http://thinkpython2.com/code/polygon.py`

***แบบฝึกหัด* 4.3.** *ให้เขียนชุดของฟังก์ชันให้ครอบคลุมการวาดรูปทรงในรูปที่ 4.2*

เฉลย: `http://thinkpython2.com/code/pie.py`.



รูปที่ 4.2.: พายของ Turtle (Turtle pies)

***แบบฝึกหัด* 4.4.** *ตัวอักษรของพยัญชนะสามารถสร้างได้จากส่วนประกอบเล็ก ๆ เช่น เส้นตั้ง เส้นนอน และเส้นโค้งจำนวนไม่กี่เส้น ให้ออกแบบพยัญชนะที่สามารถวาดจากส่วนประกอบเล็ก ๆ เหล่านี้ในจำนวนที่น้อยที่สุด จากนั้นให้เขียนฟังก์ชันที่วาดอักษรเหล่านี้*

เราควรจะเขียนหนึ่งฟังก์ชันสำหรับอักษรแต่ละตัว แล้วตั้งชื่อ เช่น `draw_a`*,* `draw_b`*, และอื่น ๆ และให้ใส่ฟังก์ชันเหล่านี้ลงในไฟล์ชื่อ* `letters.py` *เราสามารถดาวน์โหลด "โปรแกรมพิมพ์ดีด turtle (turtle typewriter)" จาก* `http://thinkpython2.com/code/typewriter.py` *เพื่อที่จะทดสอบ โค้ดของเรา*

เราสามารถดูเฉลยได้จาก `http://thinkpython2.com/code/letters.py`*; มันต้องการไฟล์นี้ในการรัน* `http://thinkpython2.com/code/polygon.py`*.*

***แบบฝึกหัด* 4.5.** *อ่านเกี่ยวกับเกลียว (spiral) ได้ที่* `http://en.wikipedia.org/wiki/Spiral`*; จากนั้นเขียนโปรแกรมที่วาดรูปเกลียว Archimedian (หรือเกลียวแบบอื่น ๆ) เฉลยอยู่ที่* `http://thinkpython2.com/code/spiral.py`

5. เงื่อนไขและการย้อนเรียกใช้

หัวข้อหลักของบทนี้ คือ คำสั่ง **if** ซึ่งดำเนินการกับโค้ดต่างกันขึ้นอยู่กับสถานะของโปรแกรม แต่ก่อนอื่น ผมอยากจะแนะนำตัวดำเนินการใหม่สองตัว: การหารปัดเศษลง (floor division) และโมดูลัส (modulus)

5.1. การหารปัดเศษลง และโมดูลัส

เครื่องหมาย **การหารปัดเศษลง (floor division)**, **//**, หารเลขสองตัวแล้วปัดเศษลงให้เป็นเลขจำนวนเต็ม ตัวอย่างเช่น สมมติว่าเวลาฉายหนัง คือ 105 นาที เราอาจจะอยากทราบว่ามันนานกี่ชั่วโมง การหารแบบปกตินิยมจะให้ค่าเป็นเลขทศนิยม:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

แต่โดยปกติแล้วเราไม่เขียนเลขชั่วโมงแบบมีจุดทศนิยม การหารปัดเศษลงจะให้ค่าจำนวนเต็มของเลขชั่วโมง โดยปัดเศษลง:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

ในการหาเศษของชั่วโมง เราสามารถลบจำนวนนาทีในหนึ่งชั่วโมงออก:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

อีกทางหนึ่ง คือ การใช้ **ตัวดำเนินการมอดูลัส (modulus operator)**, %, ซึ่งจะหารเลขสองตัว และให้ค่าเป็นเศษของการหาร

```
>>> remainder = minutes % 60
>>> remainder
45
```

ตัวดำเนินการมอดูลัสนั้นมีประโยชน์มากกว่าที่คิด เช่น เราสามารถตรวจสอบได้ว่าเลขตัวหนึ่งถูกหารลงตัวจากเลขอีกตัว ได้หรือไม่—ถ้า `x % y` เป็นศูนย์ แล้ว `x` จะถูกหารด้วย `y` ลงตัว

นอกจากนี้ เราสามารถดึงตัวเลขหลักทางขวาสุดออกมาได้ด้วย (หลักเดียวหรือหลายหลัก) เช่น `x % 10` จะได้เลขตัวขวาสุด (หลักหน่วย) ของ `x` (ในฐาน 10) เช่นเดียวกันกับ `x % 100` จะให้เลขสองหลักสุดท้ายออกมา

แต่ถ้าเราใช้ไพธอน-2 การหารเลขจะต่างออกไป ตัวดำเนินการหาร, `/`, จะทำการหารแบบปัดเศษถ้าเลขทั้งสอง เป็นจำนวนเต็ม และจะทำการหารแบบโฟลตติ้งพอยต์หากเลขตัวใดตัวหนึ่งเป็น `float`

5.2. นิพจน์บูลีน

นิพจน์บูลีน (boolean expression) คือนิพจน์ที่มีค่าความจริงเป็น จริง หรือ เท็จ ตัวอย่างต่อไปนี้ใช้ตัวดำเนินการ == ซึ่งเปรียบเทียบตัวถูกดำเนินการสองตัว และให้ค่าเป็น จริง (`True`) ถ้าสองค่านั้นเท่ากัน และให้ค่าเป็น เท็จ (`False`) หากไม่เท่ากัน:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` และ `False` เป็นค่าพิเศษที่มีชนิดเป็น บูล (`bool`); มันไม่ใช่สายอักขระ:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

ตัวดำเนินการ == เป็นหนึ่งใน **ตัวดำเนินการเชิงสัมพันธ์ (relational operators)**; ตัวอื่น ๆ คือ:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

ถึงแม้ว่าเราน่าจะคุ้นเคยกับการดำเนินการเหล่านี้ สัญลักษณ์ที่ใช้ในไพธอนจะต่างกับสัญลักษณ์ที่ใช้ในคณิตศาสตร์ ข้อผิดพลาดที่เกิดขึ้นบ่อยคือการใช้เครื่องหมายเท่ากับ (=) แทนที่จะใช้เครื่องหมายเท่ากับคู่ (==) ให้จำไว้ว่า = เป็นตัวดำเนินการสำหรับการกำหนดค่า และ == เป็นตัวดำเนินการเชิงสัมพันธ์ ไม่มีการใช้เครื่องหมาย =< หรือ =>

5.3. ตัวดำเนินการทางตรรกะ

ตัวดำเนินการทางตรรกะ (logical operators) มี 3 ตัว: `and`, `or`, และ `not`. ความหมายของตัวดำเนินการเหล่านี้ตรงกับความหมายในภาษาอังกฤษเลย เช่น `x > 0 and x < 10` จะเป็นจริงถ้า `x` มากกว่า 0 *และ* น้อยกว่า 10 เท่านั้น

`n%2 == 0 or n%3 == 0` เป็นจริงถ้า *ตัวใดตัวหนึ่ง หรือ ทั้งสองตัว* ของเงื่อนไขเป็นจริง นั่นคือ ถ้าเลขนั้นสามารถหารด้วย 2 *หรือ* 3 ลงตัว

สุดท้ายนี้ ตัวดำเนินการ `not` จะทำนิพจน์บูลีนให้เป็นนิเสธ (การกลับค่าความจริง) ดังนั้น `not (x > y)` จะเป็นจริง ถ้า `x > y` เป็นเท็จ นั่นคือ ถ้า `x` น้อยกว่าหรือเท่ากับ `y`

จริง ๆ แล้ว ตัวถูกดำเนินการของตัวดำเนินการทางตรรกะควรจะเป็นนิพจน์บูลีน แต่ไพธอนนั้นไม่ค่อยเคร่งครัดเท่าไร เลขที่ไม่เท่ากับศูนย์ใด ๆ จะถูกแปลให้มีค่าเป็น จริง `True`:

```
>>> 42 and True
True
```

ความยืดหยุ่นนี้สามารถเป็นประโยชน์ได้ แต่ก็มีรายละเอียดปลีกย่อยที่จะทำให้สับสนได้ เราควรจะหลีกเลี่ยงการทำอะไรแบบนี้ (ยกเว้นว่า เรารู้ตัวว่ากำลังทำอะไรอยู่)

5.4. การดำเนินการตามเงื่อนไข

ในการที่จะเขียนโปรแกรมที่เป็นประโยชน์ เราต้องการความสามารถในการตรวจสอบเงื่อนไขเกือบจะตลอดเวลา และสามารถเปลี่ยนพฤติกรรมของโปรแกรมตามเงื่อนไขนั้น **คำสั่งเงื่อนไข (Conditional statements)** มอบความสามารถนี้ให้เรา รูปแบบที่ง่ายที่สุด คือ คำสั่ง `if`:

```
if x > 0:
    print('x is positive')
```

นิพจน์บูลีนที่อยู่หลังจาก `if` เรียกว่า **เงื่อนไข (condition)** ถ้าเงื่อนไขเป็นจริง คำสั่งที่ย่อหน้าเข้าไปนั้นจะถูกรัน ไม่เช่นนั้น ก็ไม่มีอะไรเกิดขึ้น

คำสั่ง `if` มีโครงสร้างเหมือนนิยามของฟังก์ชัน: มีส่วนหัว ตามด้วยส่วนตัวที่ถูกย่อหน้าเข้าไป คำสั่งแบบนี้เรียกว่า **คำสั่งประกอบ (compound statements)**

มันไม่มีข้อจำกัดสำหรับจำนวนคำสั่งที่ปรากฏในส่วนตัวของคำสั่ง if แต่จะต้องมีอย่างน้อยหนึ่งคำสั่ง ในบางครั้ง มันก็มีประโยชน์ที่จะมีส่วนตัวที่ไม่มีคำสั่ง (โดยปกติแล้วใช้เป็นที่สำรองสำหรับโค้ดที่ยังไม่ได้เขียน) ในกรณีดังกล่าว เราสามารถใส่คำสั่ง `pass` ซึ่งเป็นคำสั่งที่ไม่ทำอะไรเลย

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

5.5. การดำเนินการทางเลือก

รูปแบบที่สองของคำสั่ง `if` คือ “การดำเนินการทางเลือก (alternative execution)” ซึ่งมีทางเลือกทำสองทางและมีเงื่อนไขที่จะกำหนดว่าคำสั่งชุดไหนจะถูกรัน กฎวากยสัมพันธ์ของคำสั่งเป็นแบบนี้:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

ถ้าเศษของการหารเมื่อเราหาร `x` ด้วย 2 มีค่าเป็น 0 แล้ว เรารู้ว่า `x` เป็นจำนวนคู่ และโปรแกรมจะแสดงข้อความตามนั้น แต่ถ้าเงื่อนไขเป็นเท็จ คำสั่งชุดที่สองจะทำงาน เนื่องจากเงื่อนไขจะต้องเป็นจริงหรือเท็จเท่านั้น จึงมีแค่ทางเลือกหนึ่งอย่างเท่านั้นที่ทำงาน ทางเลือกเหล่านี้เรียกว่า **แขนง (branches)** เพราะว่ามันเป็นกิ่งที่แตกแยกออกไปของกระแสการดำเนินการ (flow of execution)

5.6. เงื่อนไขลูกโซ่

บางครั้งมันก็มีมากกว่าสองทางเลือก และเราต้องการมากกว่าสองแขนงที่แตกออกไป ทางหนึ่งในการแสดงการคำนวณแบบนี้ คือ **เงื่อนไขลูกโซ่ (chained conditional)**

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` เป็นคำย่อของ “else if” (ไม่เช่นนั้น ถ้า) ทบทวนอีกทีว่าแขนงแค่หนึ่งอันเท่านั้นที่จะทำงาน มันไม่มีการจำกัดจำนวนของคำสั่ง `elif` ถ้าจะมีข้อย่อย (clause) `else` ด้วย ข้อย่อยนี้จะต้องอยู่ท้ายสุด แต่ไม่จำเป็นต้องมี

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

เงื่อนไขแต่ละอันจะถูกตรวจสอบตามลำดับ ถ้าอันแรกเป็นเท็จ อันถัดไปจะถูกตรวจสอบ และเป็นแบบนี้ไปเรื่อย ๆ ถ้าเงื่อนไขใดเป็นจริง คำสั่งในแขนงนั้นจะทำงาน และชุดคำสั่งนี้จะจบการทำงาน แม้ว่ามีเงื่อนไขมากกว่าหนึ่งที่เป็นจริง แต่แค่คำสั่งในแขนงของเงื่อนไขแรกที่เป็นจริงเท่านั้นที่จะทำงาน

5.7. เงื่อนไขซ้อน

เงื่อนไขหนึ่ง ๆ สามารถซ้อนในเงื่อนไขอื่นได้ เราสามารถเขียนตัวอย่างในหัวข้อที่แล้วให้เป็นแบบนี้ได้:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
```

```
        print('x is less than y')
    else:
        print('x is greater than y')
```

เงื่อนไขด้านนอกมีสองแขนง แขนงแรกมีคำสั่งง่าย ๆ คำสั่งเดียว แขนงที่สองมีคำสั่ง `if` อีกอันบรรจุอยู่ ซึ่งก็มีเงื่อนไขอีกสองแขนงย่อยลงไปอีก ทั้งสองแขนงข้างในนั้นมีคำสั่งแบบง่าย ๆ อยู่ แม้ว่าข้างในสามารถเป็นเงื่อนไขอีกชั้นหนึ่งก็ได้

แม้ว่าการย่อหน้าของคำสั่งต่าง ๆ ทำให้โปรแกรมมีโครงสร้างที่ชัดเจน แต่ **เงื่อนไขซ้อน (nested conditionals)** มันทำให้อ่านยากเวลาอ่านเร็ว ๆ จึงเป็นความคิดที่ดีที่จะหลีกเลี่ยงถ้าทำได้

ตัวดำเนินการทางตรรกะสามารถทำให้เราเขียนคำสั่งเงื่อนไขซ้อนให้ง่ายขึ้น เช่น เราสามารถเขียน โค้ดต่อไปนี้อีกแบบหนึ่ง โดยใช้เงื่อนไขแบบเดี่ยว:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

คำสั่ง `print` จะทำงานถ้าเราผ่านเงื่อนไขทั้งสองนี้ได้ ดังนั้น เราสามารถทำให้เกิดผลอย่างเดียวกัน โดยการใช้ตัวดำเนินการ `and`:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

สำหรับเงื่อนไขประเภทนี้ ไพธอนมีทางเลือกให้ทำแบบกระชับ:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8. การย้อนเรียกใช้

ฟังก์ชันหนึ่ง ๆ สามารถเรียกฟังก์ชันอื่นได้; ฟังก์ชันหนึ่ง ๆ ยังสามารถเรียกตัวเองได้ด้วย มันอาจจะไม่ชัดเจนว่าทำไมมันเป็นสิ่งที่ดี แต่ปรากฏว่ามันเป็นสิ่งมหัศจรรย์อย่างหนึ่งเลย ที่โปรแกรมสามารถทำได้ เช่น ให้ดูฟังก์ชันต่อไปนี้:

```
def countdown(n):
    if n <= 0:
```

```
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

ถ้า **n** เป็น 0 หรือมีค่าลบ มันจะแสดงคำว่า “Blastoff!” ออกมา ไม่เช่นนั้น มันจะแสดงค่า **n** ออกมาและเรียกฟังก์ชันที่ชื่อว่า **countdown**—ตัวมันเอง—โดยผ่านค่า **n-1** เป็นอาร์กิวเมนต์

จะเกิดอะไรขึ้นถ้าเราเรียกฟังก์ชันนี้ในลักษณะนี้?

```
>>> countdown(3)
```

การดำเนินการของฟังก์ชัน **countdown** เริ่มด้วย **n=3** และเนื่องจาก **n** มีค่ามากกว่า 0 ฟังก์ชันจะแสดงค่า 3 ออกมาและจากนั้นจึงเรียกตัวมันเอง...

> การทำงานของฟังก์ชัน **countdown** เริ่มด้วย **n=2** และเนื่องจาก **n** มีค่ามากกว่า 0 ฟังก์ชันจะแสดงค่า 2 ออกมาและจากนั้นจึงเรียกตัวมันเอง...
>
> > การทำงานของฟังก์ชัน **countdown** เริ่มด้วย **n=1** และเนื่องจาก **n** มีค่ามากกว่า 0 ฟังก์ชันจะแสดงค่า 1 ออกมาและจากนั้นจึงเรียกตัวมันเอง...
> >
> > > การทำงานของฟังก์ชัน **countdown** เริ่มด้วย **n=0** และเนื่องจาก **n** มีค่าไม่มากกว่า 0 ฟังก์ชันจะแสดงคำว่า “Blastoff!” ออกมา จบ และกลับออกไป
> >
> > ฟังก์ชัน **countdown** ที่ได้รับค่า **n=1** มาก็จบ และกลับออกไป
>
> ฟังก์ชัน **countdown** ที่ได้รับค่า **n=2** มาก็จบ และกลับออกไป

ฟังก์ชัน **countdown** ที่ได้รับค่า **n=3** มาก็จบ และกลับออกไป

และจากนั้นเราก็กลับมายัง **__main__** ดังนั้น เอาต์พุตทั้งหมดจะหน้าตาเป็นแบบนี้:

```
3
2
1
Blastoff!
```

ฟังก์ชันที่เรียกตัวมันเอง คือ **ฟังก์ชันย้อนเรียกใช้ (recursive function)** หรือ ฟังก์ชันเวียนเกิด (ในหนังสือเล่มนี้ จะเรียกว่า ฟังก์ชันย้อนเรียกใช้); กระบวนการทำงานของมันเรียกว่า **การย้อนเรียกใช้ (recursion)**

อีกตัวอย่างหนึ่ง เราสามารถเขียนฟังก์ชันที่พิมพ์สายอักขระจำนวน n ครั้ง

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

ถ้า `n <= 0` คำสั่ง **return** จะทำให้จบฟังก์ชัน กระแสการดำเนินการจะกลับไปยังโปรแกรมที่เรียกฟังก์ชัน (caller) ทันที และบรรทัดที่เหลือในฟังก์ชันนั้นจะไม่ถูกรัน

ส่วนที่เหลือของฟังก์ชันนั้นเหมือนกับฟังก์ชัน `countdown`: มันแสดง `s` และจากนั้นเรียกตัวเองเพื่อแสดง `s` ไปอีก $n-1$ ครั้ง ดังนั้น จำนวนบรรทัดของเอาต์พุตจะเป็น `1 + (n - 1)` ซึ่งรวมกันแล้วได้ `n` บรรทัด

สำหรับตัวอย่างง่าย ๆ แบบนี้ มันอาจจะง่ายกว่าที่จะใช้ลูป `for` แต่เราจะเห็นตัวอย่างอีกมากในภายหลังที่ยากที่จะเขียนด้วย ลูป `for` และง่ายที่จะเขียนด้วยการย้อนเรียกใช้ (recursion) ดังนั้น จึงเป็นเรื่องที่ดีที่จะเริ่มเข้าใจหัวข้อนี้ก่อน

5.9. แผนภาพแบบกองซ้อนสำหรับฟังก์ชันเรียกซ้ำ

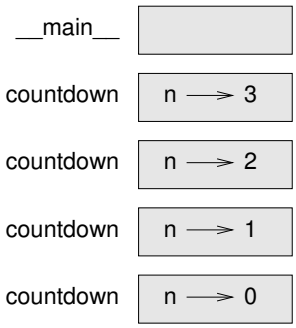
ในหัวข้อที่ 3.9 เราใช้แผนภาพแบบกองซ้อนในการแสดงสถานะของโปรแกรม ในขณะที่มีการเรียกฟังก์ชัน แผนภาพชนิดเดียวกันนี้สามารถช่วยให้เข้าใจฟังก์ชันเรียกซ้ำ (recursive function) ได้ด้วย

ทุกครั้งที่ฟังก์ชันถูกเรียก ไพธอนจะสร้างกรอบที่บรรจุ*ตัวแปรเฉพาะที่*และพารามิเตอร์ของฟังก์ชัน สำหรับฟังก์ชันเรียกซ้ำ มันอาจจะมีกรอบมากกว่าหนึ่งกรอบบนกอง ณ ขณะหนึ่ง

รูปที่ 5.1 แสดงแผนภาพแบบกองซ้อนสำหรับฟังก์ชัน `countdown` ที่ถูกเรียกด้วยค่า `n = 3`.

เช่นเดียวกับโปรแกรมทั่ว ๆ ไป บนสุดของกองคือกรอบของ `__main__` มันว่างเปล่าเพราะว่าเราไม่ได้สร้างตัวแปรใด ๆ ใน `__main__` หรือไม่ได้ผ่านอาร์กิวเมนต์ใด ๆ เข้าไป

กรอบ `countdown` ทั้ง 4 กรอบ มีค่าพารามิเตอร์ `n` ที่ต่างกัน ล่างสุดของกองซึ่ง `n=0` เรียกว่า **กรณีฐาน (base case)** มันไม่ได้เรียกตัวเองซ้ำ ดังนั้น จึงไม่มีกรอบเพิ่มไปอีก



รูปที่ 5.1.: แผนภาพแบบกองซ้อน (Stack diagram)

เพื่อเป็นการฝึกทำ ให้วาดแผนภาพแบบกองซ้อนสำหรับฟังก์ชัน `print_n` ที่ถูกเรียกด้วยค่า `s = 'Hello'` และ `n=2` จากนั้นให้เขียนฟังก์ชันชื่อว่า `do_n` ที่รับอ็อบเจกต์ฟังก์ชันและจำนวน `n` เข้าเป็นอาร์กิวเมนต์ และมันจะเรียกฟังก์ชันที่กำหนดให้เป็นจำนวน `n` ครั้ง

5.10. การย้อนเรียกใช้ไม่รู้จบ

ถ้าการย้อนเรียกใช้ไปไม่ถึงกรณีฐาน (base case) เสียที มันจะทำให้เกิดการย้อนเรียกใช้ไปตลอดกาล และโปรแกรมก็จะไม่จบ นี่เรียกว่า **การย้อนเรียกใช้ไม่รู้จบ (infinite recursion)** และมันก็ไม่ได้เป็นดีเลย นี่คือ โปรแกรมแบบสั้นที่สุดที่จะทำให้เกิดการย้อนเรียกใช้แบบไม่สิ้นสุด:

```
def recurse():
    recurse()
```

ในสภาพแวดล้อมของการเขียนโปรแกรมส่วนใหญ่ โปรแกรมจะไม่ทำงานไปตลอดกาลแบบนั้น ไพธอนจะรายงานข้อความความผิดพลาดเมื่อมีการย้อนเรียกใช้จนถึงความลึกที่มากที่สุด (ที่อนุญาตให้รันได้):

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

การย้อนรอยแบบนี้มันจะเยอะกว่าที่เราเคยทำในบทก่อนหน้านี้นิดหน่อย เมื่อเกิดข้อผิดพลาดในตัวอย่างนี้ขึ้นมา มันมี 1000 `recurse` frames (กรอบการเรียกซ้ำ) บนกองนี้!

ถ้าเราเจอการย้อนเรียกใช้ไม่รู้จบโดยบังเอิญ ให้ทบทวนฟังก์ชันของเราเพื่อให้แน่ใจว่ามันมีกรณีฐาน (base case) ที่ไม่เรียกตัวเองซ้ำ และถ้ามีกรณีฐานแล้ว ให้ตรวจสอบว่าเราจะไปถึงมันจริง ๆ

5.11. การนำเข้าข้อมูลผ่านคีย์บอร์ด

โปรแกรมที่เราเขียนมาถึงตอนนี้ไม่ได้รับอินพุต (input) มาจากผู้ใช้ มันแค่ทำอะไรซ้ำ ๆ ตลอดเวลา

ไพธอนได้เตรียมฟังก์ชันพร้อมใช้เรียกว่า `input` ที่หยุดโปรแกรมและรอให้ผู้ใช้พิมพ์อะไรสักอย่างเข้ามา เมื่อผู้ใช้กดปุ่ม **Return** หรือ **Enter** โปรแกรมจะทำงานต่อ และฟังก์ชัน `input` จะคืนค่าสิ่งที่ผู้ใช้พิมพ์เข้ามาเป็นชนิดสายอักขระ ในไพธอน-2 ฟังก์ชันที่ทำงานแบบเดียวกันนี้เรียกว่า `raw_input`

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

ก่อนที่จะรับอินพุตมาจากผู้ใช้ มันเป็นความคิดที่ดีที่จะพิมพ์ข้อความไปบอกผู้ใช้ว่าให้พิมพ์อะไรเข้ามา ฟังก์ชัน `input` สามารถรับข้อความพร้อมรับ (prompt) เป็นอาร์กิวเมนต์ได้:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

ลำดับอักขระ `\n` ในตอนท้ายของข้อความพร้อมรับเป็นตัวแทนของ **บรรทัดใหม่ (newline)** ซึ่งเป็นอักขระพิเศษที่ทำให้เกิดการขึ้นบรรทัดใหม่ นั่นคือเหตุผลที่อินพุตของผู้ใช้ปรากฏอยู่ข้างใต้ข้อความพร้อมรับ

ถ้าเราคาดหวังว่าผู้ใช้จะพิมพ์จำนวนเต็มเข้ามา เราสามารถลองแปลงค่าที่คืนกลับมาเป็น `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?
\n'
```

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

แต่ถ้าผู้ใช้พิมพ์อย่างอื่นที่ไม่ใช่สายอักขระของตัวเลขแล้วล่ะก็ เราก็จะได้ข้อผิดพลาด:

(หมายเหตุผู้แปล: ได้ข้อผิดพลาดตอนที่พยายามแปลงให้เป็น int)

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

เราจะรู้ว่าจะจัดการกับข้อผิดพลาดประเภทนี้อย่างไรในภายหลัง

5.12. การดีบัก

เมื่อเกิดข้อผิดพลาดเชิงวากยสัมพันธ์ (syntax error) หรือข้อผิดพลาดตอนดำเนินการ (runtime error) ข้อความแจ้งข้อผิดพลาด (error message) มีข้อมูลจำนวนมากให้เรา แต่มันจะทำให้เรารู้สึกท่วมท้นมาก ส่วนที่เป็นประโยชน์โดยปกติแล้วจะเป็น:

- ข้อผิดพลาดที่เกิดขึ้นเป็นชนิดใด และ
- มันเกิดขึ้นที่ไหน

ข้อผิดพลาดเชิงวากยสัมพันธ์นั้นง่ายที่จะหา แต่มันก็มีข้อต้องระวังนิดหน่อย ข้อผิดพลาดที่เกี่ยวกับการเว้นวรรค (whitespace error) อาจจะทำให้เราปวดหัวได้ เพราะว่าช่องว่างและย่อหน้านั้นมันมองไม่เห็น และเราก็ชินกับการไม่สนใจมัน

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
```

```
IndentationError: unexpected indent
```

ในตัวอย่างนี้ ปัญหา คือ บรรทัดที่สองนั้นถูกย่อหน้าเข้าไปโดยช่องว่าง 1 ช่อง แต่ข้อผิดพลาดนั้นชี้ไปที่ตัวแปร y ซึ่งอาจจะทำให้ไขว้เขวได้ โดยทั่วไปแล้ว ข้อความแจ้งข้อผิดพลาดจะระบุตำแหน่งตรงที่เจอปัญหา แต่ข้อผิดพลาดจริง ๆ อาจจะอยู่ก่อนตำแหน่งที่ระบุก็ได้ บางทีก็อยู่ในบรรทัดก่อนหน้า

เช่นเดียวกันกับข้อผิดพลาดตอนดำเนินการ สมมติว่าเราพยายามที่จะคำนวณอัตราส่วนระหว่าง สัญญาณและสัญญาณรบกวน (signal-to-noise ratio) ในหน่วยเดซิเบล สูตรคือ $SNR_{db} = 10 \log_{10}(P_{signal} / P_{noise})$ ในไพธอน เราน่าจะเขียนประมาณนี้:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

เมื่อเรารันโปรแกรม เราจะได้เอ็กเซ็ปชั่น:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

ข้อความแจ้งข้อผิดพลาดระบุว่ามีปัญหาที่บรรทัดที่ 5 แต่ก็ไม่เห็นมีอะไรผิดนี่นา เพื่อที่จะหาข้อผิดพลาดที่แท้จริง มันอาจจะเป็นประโยชน์ที่จะพิมพ์ค่าของ `ratio` ออกมาดู ซึ่งมีค่าเป็น 0 ปัญหาจึงอยู่ที่บรรทัดที่ 4 ที่ใช้การหารแบบปัดเศษลง แทนที่จะใช้การหารแบบโฟลตติ้งพอยต์

เราควรที่จะใช้เวลาในการอ่านข้อความแจ้งข้อผิดพลาดอย่างระมัดระวัง แต่อย่าคิดเอาเองว่า ทุกสิ่งมันบอกเรานั้นถูกต้องเป๊ะตามนั้น

5.13. อภิธานศัพท์

การหารปัดเศษลง (floor division): ตัวดำเนินการที่มีเครื่องหมายเป็น `//` ซึ่งหารเลขสองตัวและปัดเศษลงให้เป็นจำนวนเต็ม

ตัวดำเนินการมอดูลัส (modulus operator): ตัวดำเนินการที่มีเครื่องหมายเป็นเปอร์เซ็นต์ (**%**) ซึ่งทำงานกับจำนวนเต็มและคืนค่าเป็นเศษของการหาร

นิพจน์บูลีน (boolean expression): นิพจน์ที่มีค่าเป็น `True` (จริง) หรือ `False` (เท็จ)

ตัวดำเนินการเชิงสัมพันธ์ (relational operator): ตัวดำเนินการที่เปรียบเทียบค่าของตัวถูกดำเนินการ: **==**, **!=**, **>**, **<**, **>=**, และ **<=**

ตัวดำเนินการเชิงตรรกะ (logical operator): ตัวดำเนินการที่เชื่อมประกอบนิพจน์บูลีน: **and**, **or**, และ **not**

คำสั่งเงื่อนไข (conditional statement): คำสั่งที่ควบคุมการทำงานของโปรแกรมให้ขึ้นอยู่กับเงื่อนไขบางอย่าง

เงื่อนไข (condition): นิพจน์บูลีนในคำสั่งเงื่อนไขที่กำหนดว่าแขนงไหนของคำสั่งจะถูกทำงาน

คำสั่งประกอบ (compound statement): คำสั่งที่ประกอบด้วยส่วนหัว (header) และส่วนตัว (body) ส่วนหัวจะลงท้ายด้วยเครื่องหมายทวิภาค หรือ โคลอน (:) ส่วนตัวจะถูกย่อหน้าเข้าไปให้สัมพันธ์กับส่วนหัว

แขนง (branch): ชุดคำสั่งที่เป็นทางเลือกอันหนึ่งของคำสั่งเงื่อนไข

เงื่อนไขลูกโซ่ (chained conditional): คำสั่งเงื่อนไขที่มีชุดของแขนงทางเลือกหลาย ๆ อัน

เงื่อนไขซ้อน (nested conditional): คำสั่งเงื่อนไขที่ปรากฏอยู่ในแขนงหนึ่งของคำสั่งเงื่อนไขอีกอันหนึ่ง

คำสั่งคืนค่า (return statement): คำสั่งที่ทำให้ฟังก์ชันจบการทำงานทันทีและคืนการทำงานให้กับโปรแกรมที่เรียกฟังก์ชัน (caller)

การย้อนเรียกใช้ (recursion): ขั้นตอนของการเรียกฟังก์ชันตัวที่กำลังรันอยู่ (หมายเหตุผู้แปล: การเรียกตัวมันเอง)

กรณีฐาน (base case): แขนงเงื่อนไขหนึ่งในฟังก์ชันเรียกซ้ำที่ไม่มีการเรียกตัวเองซ้ำ

การย้อนเรียกใช้ไม่รู้จบ (infinite recursion): การย้อนเรียกใช้ที่ไม่มีกรณีฐาน (base case) หรือไม่มีทางเรียกกรณีฐานเลย ในที่สุดแล้ว การย้อนเรียกใช้ไม่รู้จบจะทำให้เกิดข้อผิดพลาดในการดำเนินการ (runtime error)

5.14. แบบฝึกหัด

แบบฝึกหัด 5.1. *มอดูล* `time` *มีฟังก์ชันที่ชื่อว่า* `time` *ให้ใช้ ซึ่งคืนค่าเป็นเวลามาตรฐานโลกตามนาฬิกาที่กรีนิช (Greenwich Mean Time: GMT) ณ ปัจจุบัน อ้างอิงจาก "the epoch" ซึ่งเป็นเวลามาตรฐานที่ใช้อ้างอิง บนระบบยูนิกซ์ epoch คือ วันที่ 1 มกราคม ค.ศ. 1970*

(หมายเหตุผู้แปล: epoch อ่านว่า เอพ'เพิค คือเวลา 00:00:00 UTC ของวันที่ 1 มกราคม ค.ศ. 1970)

```
>>> import time
>>> time.time()
1437746094.5735958
```

ให้เขียนสคริปต์ที่อ่านเวลาปัจจุบัน และแปลงให้เป็นเวลาในหน่วยชั่วโมง นาที และวินาที และจำนวนวันตั้งแต่ the epoch

แบบฝึกหัด 5.2. *ทฤษฎีบทสุดท้ายของแฟร์มา (Fermat's Last Theorem) ระบุว่า ไม่มีจำนวนเต็มบวก a, b, และ c ใด ๆ ที่ทำให้:*

$$a^n + b^n = c^n$$

สำหรับค่าใด ๆ ของ n ที่มากกว่า 2

1. *ให้เขียนฟังก์ชันที่ชื่อว่* `check_fermat` *ซึ่งรับค่าพารามิเตอร์ 4 ค่า —*`a`*,* `b`*,* `c` *และ* `n`*— และตรวจสอบว่าทฤษฎีบทสุดท้ายของแฟร์มา นั้นจริงหรือไม่ ถ้า n มากกว่า 2 และ*

   $$a^n + b^n = c^n$$

   โปรแกรมควรจะพิมพ์ว่า "Holy smokes, Fermat was wrong!" ไม่เช่นนั้น โปรแกรมควรจะพิมพ์ว่า "No, that doesn't work."

2. *ให้เขียนฟังก์ชันที่รับค่า* `a`*,* `b`*,* `c` *และ* `n` *จากผู้ใช้ แปลงค่าเหล่านี้ เป็นจำนวนเต็ม และใช้ฟังก์ชัน* `check_fermat` *ตรวจสอบดูว่าค่าเหล่านี้ผ่าฝืนทฤษฎีบทสุดท้ายของแฟร์มาหรือไม่*

แบบฝึกหัด 5.3. *ถ้าเราได้กิ่งไม้มาสามชิ้น เราอาจจะหรืออาจจะไม่สามารถที่จะเรียงมันให้เป็นสามเหลี่ยมได้ เช่น ถ้าไม้อันหนึ่งยาว 12 นิ้ว และอีกสองอันยาวอันละ 1 นิ้ว เราจะไม่สามารถทำให้ไม้แท่งสั้นประกบกันได้ สำหรับความยาวของไม้ทั้งสามแท่ง มันมีการทดสอบแบบ ง่าย ๆ เพื่อที่จะดูว่ามันสามารถที่จะประกอบกันเป็นสามเหลี่ยมได้หรือไม่:*

ถ้าความยาวของด้านใดด้านหนึ่งนั้นยาวมากกว่าผลรวมของอีกสองด้านที่เหลือ เราจะไม่สามารถสร้างสามเหลี่ยมได้ นอกจากนี้ เราจะสามารถทำได้ (ถ้าผลรวมของสองด้าน เท่ากับด้านที่สาม มันจะประกอบกันเป็นสิ่งที่เรียกว่า สามเหลี่ยมลดรูป หรือ "degenerate" triangle)

(หมายเหตุผู้แปล: สามเหลี่ยมลดรูป หรือ degenerate triangle คือ สามเหลี่ยมที่ไม่เห็นเป็นรูปสามเหลี่ยมทั่วไป แต่เป็นคล้าย ๆ เส้นตรงแทน)

1. *ให้เขียนฟังก์ชันชื่อว่า* `is_triangle` *ซึ่งรับจำนวนเต็มมาเป็นอาร์กิวเมนต์ และพิมพ์ "Yes" หรือ "No" ขึ้นอยู่กับว่า เราสามารถจะประกอบสามเหลี่ยมแท่งไม้จากความยาวที่กำหนดให้ได้หรือไม่*

2. *ให้เขียนฟังก์ชันที่รับอินพุตมาจากผู้ใช้เป็นความยาวของไม้ 3 แท่ง แปลงให้เป็น จำนวนเต็มและใช้ฟังก์ชัน* `is_triangle` *ตรวจสอบว่าแท่งไม้ที่มีความยาวที่ใส่เข้ามาจะสามารถประกอบกันเป็นสามเหลี่ยมได้หรือไม่*

แบบฝึกหัด 5.4. *ผลลัพธ์ของโปรแกรมต่อไปนี้คืออะไร? ให้วาดรูปแผนภาพกองซ้อนที่แสดงสถานะ ของโปรแกรม เมื่อมันทำการพิมพ์ผลออกมา*

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

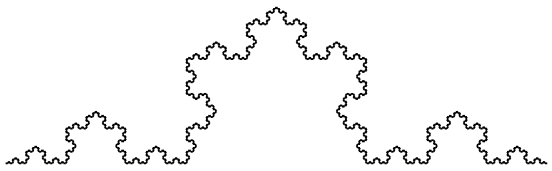
1. *จะเกิดอะไรขึ้นถ้าเราเรียกฟังก์ชันแบบนี้:* `recurse(-1, 0)`*?*

2. *ให้เขียนด็อกสตริง (docstring) ที่อธิบายทุกอย่างที่บางคนต้องรู้ เพื่อที่จะใช้ฟังก์ชันนี้ (ไม่ต้องเขียนอย่างอื่นมา)*

แบบฝึกหัดต่อไปนี้ใช้มอดูล `turtle` ในบทที่ 4:

แบบฝึกหัด 5.5. *ให้อ่านฟังก์ชันต่อไปนี้ และดูว่าเรารู้ไหมว่ามันทำอะไร (ดูตัวอย่าง ในบทที่ 4) จากนั้นให้รันฟังก์ชันและดูว่าเราคิดถูกไหม*

```
def draw(t, length, n):
    if n == 0:
```



รูปที่ 5.2.: เส้นโค้งค็อค (Koch curve)

```
    return
angle = 50
t.fd(length*n)
t.lt(angle)
draw(t, length, n-1)
t.rt(2*angle)
draw(t, length, n-1)
t.lt(angle)
t.bk(length*n)
```

แบบฝึกหัด 5.6. *เส้นโค้งค็อค (Koch curve) เป็นส่วนที่คล้ายรูปที่ 5.2 ในการจะวาดเส้นโค้งค็อคที่ ยาว x ทั้งหมดที่เราจะต้องทำคือ*

1. *วาดเส้นโค้งค็อคที่ยาว $x/3$*
2. *หันซ้าย 60 องศา*
3. *วาดเส้นโค้งค็อคที่ยาว $x/3$*
4. *หันขวา 120 องศา*
5. *วาดเส้นโค้งค็อคที่ยาว $x/3$*
6. *หันซ้าย 60 องศา*
7. *วาดเส้นโค้งค็อคที่ยาว $x/3$*

ยกเว้น ถ้า x น้อยกว่า 3: เราจะแค่วาดเส้นตรงที่ยาว x ได้เท่านั้น

1. *ให้เขียนฟังก์ชันที่ชื่อว่า* **koch** *ซึ่งรับ turtle และความยาวเข้ามา เป็นพารามิเตอร์ และใช้ turtle วาดเส้นโค้งค็อคที่ยาวตามที่กำหนดให้*

2. *ให้เขียนฟังก์ชันที่ชื่อว่า* `snowflake` *ซึ่งวาดเส้นโค้งค็อค 3 เส้น เพื่อจะร่างเส้นของเกล็ดหิมะ*

   เฉลย: `http://thinkpython2.com/code/koch.py`.

3. *เส้นโค้งค็อคสามารถถูกทำให้ครอบคลุมกรณีอื่น ๆ ในหลายทาง ดูตัวอย่างที่* `http://en.wikipedia.org/wiki/Koch_snowflake` *และทำรูปร่างที่เราชอบได้เลย*

6. ฟังก์ชันที่ให้ผล

ฟังก์ชันไพธอนหลายอันที่เราได้ใช้มา เช่น ฟังก์ชันทางคณิตศาสตร์ จะให้ค่าที่คืนกลับไปยังผู้เรียก (return value) แต่ฟังก์ชันที่เราเขียนขึ้นมาเองจนถึงตอนนี้เป็นฟังก์ชันที่ไม่คืนค่า: พวกมันมีการทำงาน เช่น พิมพ์ค่า หรือทำให้ turtle เคลื่อนที่ แต่ไม่มีค่าที่จะคืนกลับไป ในบทนี้ เราจะเรียนรู้เกี่ยวกับการเขียนฟังก์ชันที่มีค่าที่คืนกลับไป หรือ ฟังก์ชัน ที่ให้ผล (fruitful function)

6.1. ค่าคืนกลับ

การเรียกฟังก์ชันจะทำให้เกิดค่าที่ถูกส่งคืนกลับมา ซึ่งโดยปกติแล้วเรากำหนดค่านี้ให้กับตัวแปร หรือใช้เป็นส่วนหนึ่งของนิพจน์

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

ฟังก์ชันที่เราเขียนมาจนถึงตอนนี้เป็นฟังก์ชันวอยด์ (void function) ถ้าให้พูดแบบสบาย ๆ ก็คือ มันไม่คืนค่า; แต่จริง ๆ แล้วค่าที่ถูกส่งคืนกลับมาคือ `None`

ในบทนี้ (ในที่สุด) เราจะเขียนฟังก์ชันที่ให้ผลออกมา ตัวอย่างแรกคือฟังก์ชัน `area` ซึ่ง คืนค่าเป็นพื้นที่ของวงกลมที่มีรัศมีที่กำหนดให้:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

เราเจอคำสั่ง `return` มาก่อนหน้านี้แล้ว แต่ในฟังก์ชันที่ให้ผลนี้ คำสั่ง `return` จะมีนิพจน์ด้วย คำสั่งนี้หมายความว่า “ให้กลับออกไปจากฟังก์ชันนี้ทันที และใช้ค่าต่อไปนี้เป็นค่าที่ถูกคืน กลับไป” นิพจน์ในฟังก์ชันนี้อาจจะดูยุ่งยากแบบไม่มีเหตุผลหน่อย ดังนั้น เราสามารถเขียนฟังก์ชันนี้ใหม่แบบกระชับขึ้น:

```
def area(radius):
    return math.pi * radius**2
```

แต่ในอีกทางหนึ่ง การมี **ตัวแปรชั่วคราว** เช่น `a` สามารถทำให้การดีบักง่ายขึ้น

ในบางครั้ง มันก็มีประโยชน์ที่จะมีคำสั่ง return หลายอัน แต่ละอันอยู่ในแต่ละแขนงของเงื่อนไข:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

เนื่องจากคำสั่ง `return` อยู่ในเงื่อนไขทางเลือก เพราะฉะนั้นจะมีคำสั่งอันเดียวเท่านั้นที่จะทำงาน

ทันทีที่คำสั่ง return ทำงาน ฟังก์ชันจะหยุดการทำงานโดยไม่ทำคำสั่งที่เหลืออีก โค้ดที่ปรากฏหลังจากคำสั่ง `return` หรือ ณ ที่ใดก็ตามที่กระแสการดำเนินการของโปรแกรมไปไม่ถึง เรียกว่า **โค้ดตาย (dead code)**

ในฟังก์ชันที่ให้ผล มันเป็นความคิดที่ดีที่จะทำให้มั่นใจว่าทุกเส้นทางในโปรแกรมนั้นจบด้วยคำสั่ง `return` เช่น:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

ฟังก์ชันนี้ไม่ถูกต้องเพราะว่า ถ้า `x` เป็น 0 แล้ว ไม่มีเงื่อนไขใดที่เป็นจริง และ ฟังก์ชันก็จบโดยไม่ได้รันคำสั่ง `return` ถ้ากระแสการดำเนินการไปถึงตอนจบของฟังก์ชัน ค่าที่ถูกคืนกลับไปจะเป็น `None` ซึ่งไม่ใช่ค่า 0 ซะทีเดียว

```
>>> print(absolute_value(0))
None
```

อย่างไรก็ตาม ไพธอนได้เตรียมฟังก์ชันภายในเรียกว่า `abs` ที่หาค่าสัมบูรณ์ (absolute value)

เพื่อเป็นการฝึกทำ ให้เขียนฟังก์ชันที่ชื่อว่า `compare` ซึ่งรับค่าเข้ามา 2 ค่า คือ `x` และ `y` และคืนค่า `1` ถ้า `x > y`, คืนค่า `0` ถ้า `x == y`, และคืนค่า `-1` ถ้า `x < y`

6.2. การพัฒนาโปรแกรมแบบเพิ่มส่วน

เมื่อเราเขียนฟังก์ชันที่ใหญ่ขึ้น เราอาจจะเจอว่าเราใช้เวลาในการดีบักนานขึ้น

เพื่อที่จะจัดการกับโปรแกรมที่ซับซ้อนมากขึ้นเรื่อย ๆ เราอาจจะพยายามทำกระบวนการที่เรียกว่า **การพัฒนาโปรแกรมแบบเพิ่มส่วน (incremental development)** จุดประสงค์ของการพัฒนาโปรแกรมแบบเพิ่มส่วน คือ การหลีกเลี่ยงช่วงเวลาดีบักที่ยาว โดยการเติมและทดสอบโค้ดทีละน้อย

ตัวอย่างเช่น สมมติว่าเราต้องการจะหาระยะทางระหว่างจุดสองจุด โดยกำหนดพิกัด (x_1, y_1) และ (x_2, y_2) จากทฤษฎีพิธากอรัส ระยะทาง คือ:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

ขั้นตอนแรก คือ การพิจารณาว่าฟังก์ชัน `distance` จะหน้าตาเป็นอย่างไรในไพธอน อีกนัย หนึ่งคืออะไรคืออินพุต (พารามิเตอร์) และอะไรคือเอาต์พุต (ค่าคืนกลับ)

ในกรณีนี้ อินพุต คือ จุดสองจุด ซึ่งเราสามารถแทนด้วยตัวเลข 4 ตัว ค่าคืนกลับ คือ ระยะทาง ซึ่งเป็นเลขทศนิยม

เราสามารถเขียนโครงฟังก์ชันได้ในทันที:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

เป็นที่ชัดเจนว่า เวอร์ชันนี้ยังไม่ได้คำนวณระยะทาง; มันจะส่งค่า 0 กลับไปตลอด แต่มันก็ถูกต้อง ตามกฎวากยสัมพันธ์และมันก็ทำงานได้ ซึ่งหมายความว่า เราสามารถทดสอบมันก่อนที่เราจะทำให้มัน ซับซ้อนไปมากกว่านี้

ในการทดสอบฟังก์ชันใหม่นี้ เราจะเรียกมันโดยใช้อาร์กิวเมนต์ตัวอย่าง:

```
>>> distance(1, 2, 4, 6)
0.0
```

ผมเลือกค่าเหล่านี้ เพื่อทำให้ระยะทางราบเท่ากับ 3 และระยะทางดิ่งเท่ากับ 4; ซึ่งจะทำให้ผลลัพธ์เป็น 5 ซึ่งตรงกับสามเหลี่ยมพิธากอรัส 3-4-5 เมื่อทำการทดสอบฟังก์ชัน มีประโยชน์ที่รู้คำตอบที่ถูก

ถึงตรงนี้ เราได้ยืนยันว่าฟังก์ชันนี้ถูกเชิงวากยสัมพันธ์ และเราสามารถเริ่มเพิ่มโค้ดให้กับส่วนตัว ของฟังก์ชันได้ ขั้นตอนต่อไปที่เหมาะสม คือ การหาผลต่างของ $x_2 - x_1$ และ $y_2 - y_1$ เวอร์ชันต่อไปของฟังก์ชันเก็บค่าเหล่านี้ในตัวแปรชั่วคราวและพิมพ์มันออกมา

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

ถ้าฟังก์ชันทำงานได้ถูกต้อง มันควรจะแสดงผลว่า `dx is 3` และ `dy is 4` ถ้าเป็นเช่นนั้น เรารู้ว่าฟังก์ชันรับค่าอาร์กิวเมนต์เข้าไปได้ และทำการคำนวณอย่างแรกได้ถูกต้อง ไม่เช่นนั้น ก็จะมีแค่ไม่กี่บรรทัดเท่านั้นที่เราต้องตรวจสอบ

ขั้นต่อไป เราคำนวณผลรวมของกำลังที่สองของ `dx` และ `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

อีกครั้งหนึ่ง เราจะรันโปรแกรมตอนนี้และตรวจสอบเอาต์พุต (ซึ่งควรจะเป็น 25) ท้ายที่สุด เราสามารถใช้ `math.sqrt` เพื่อที่จะคำนวณและคืนค่ากลับมาได้:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

ถ้ามันทำงานได้ถูกต้อง เราก็เสร็จเรียบร้อย ไม่เช่นนั้น เราอาจจะต้องพิมพ์ค่าของ `result` ก่อนคำสั่ง return

เวอร์ชันสุดท้ายของฟังก์ชันนี้ไม่ได้แสดงอะไรเลยตอนที่ทำงาน มันแค่คืนค่ากลับมา คำสั่ง `print` ที่เราเขียนนั้นมีประโยชน์กับการดีบัก แต่เมื่อเราทำให้ฟังก์ชันทำงานได้แล้ว เราควรที่จะเอามันออกไป โค้ดพวกนี้เรียกว่า **นั่งร้าน (scaffolding)** เพราะว่ามันมี ประโยชน์กับการสร้างโปรแกรม แต่มันไม่ใช่ส่วนหนึ่งของผลลัพธ์สุดท้าย

เมื่อเราเริ่มฝึกเขียนโปรแกรม เราควรจะเพิ่มโค้ดแค่ทีละ 1-2 บรรทัด แต่เมื่อเรามีประสบการณ์ ในการเขียนโปรแกรมมากขึ้นแล้ว เราจะพบว่าเราจะเขียนและดีบักโค้ดก้อนที่ใหญ่ขึ้น ไม่ว่าจะแบบไหนก็ตาม การพัฒนาโปรแกรมแบบเพิ่มส่วนสามารถทำให้เราประหยัดเวลาดีบักได้มาก

ลักษณะสำคัญของขั้นตอนเหล่านี้คือ:

1. เริ่มด้วยโปรแกรมที่ทำงานได้และเพิ่มขึ้นทีละน้อย ณ จุดใด ๆ ถ้ามีข้อผิดพลาดเราจะรู้ดีว่ามันอยู่ตรงไหน
2. ใช้ตัวแปรในการเก็บค่าที่ใช้ระหว่างทาง เพื่อที่เราจะสามารถแสดงและตรวจสอบค่ามันได้
3. เมื่อโปรแกรมทำงานได้แล้ว เราอาจจะต้องเอาโค้ดนั่งร้านออก หรือ รวมคำสั่งหลายคำสั่งเข้าเป็นนิพจน์ประกอบ แต่ทำเฉพาะกรณีที่มันไม่ทำให้โปรแกรมนั้นอ่านยากขึ้น

เพื่อเป็นการฝึกทำ ให้ใช้การพัฒนาโปรแกรมแบบเพิ่มส่วน เขียนฟังก์ชันที่ชื่อว่า `hypotenuse` ซึ่งคืนค่าเป็นความยาวของด้านตรงข้ามของสามเหลี่ยมมุมฉาก (hypotenuse) เมื่อกำหนดความยาวของสองด้านที่เหลือให้เป็นอาร์กิวเมนต์ บันทึกแต่ละช่วงของขั้นตอนการพัฒนาควบคู่ไปกับการทำ

6.3. การประกอบ

ถึงตอนนี้เราควรจะคาดได้แล้วว่า เราสามารถเรียกฟังก์ชันหนึ่งจากข้างในฟังก์ชันอีกอันหนึ่ง เช่น เราจะเขียนฟังก์ชันที่รับจุดสองจุด คือ จุดกึ่งกลางของวงกลม และจุดที่อยู่บนเส้นรอบวง และคำนวณพื้นที่ของวงกลม

สมมติว่าจุดกึ่งกลางนั้นถูกเก็บอยู่ในตัวแปร `xc` และ `yc` และจุดบนเส้นรอบวงถูกเก็บอยู่ใน `xp` และ `yp` ขั้นตอนแรกคือการหารัศมีของวงกลม ซึ่งคือระยะทางระหว่างจุดสองจุดนี้ เราเพิ่งจะเขียนฟังก์ชัน distance ที่จะหาระยะทางนี้:

```
radius = distance(xc, yc, xp, yp)
```

ขั้นตอนต่อไป คือ การหาพื้นที่ของวงกลมด้วยรัศมีที่เราได้มา; เราก็เพิ่งเขียนมันไปด้วยเช่นกัน:

```
result = area(radius)
```

เราก็ห่อหุ้มขั้นตอนเหล่านี้ให้มาอยู่ในฟังก์ชัน ได้เป็น:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
```

```
    result = area(radius)
    return result
```

ตัวแปรชั่วคราว `radius` และ `result` มีประโยชน์ในการพัฒนาและดีบัก แต่เมื่อโปรแกรมรันได้อย่างถูกต้องแล้ว เราก็สามารถทำให้มันกระชับมากขึ้น โดยการเขียนฟังก์ชันให้เป็น:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4. ฟังก์ชันบูลีน

ฟังก์ชันสามารถคืนค่าเป็นชนิดบูลีนได้ ซึ่งหลายครั้งทำให้สะดวกสำหรับการซ่อนการทดสอบที่ซับซ้อนในฟังก์ชัน ตัวอย่างเช่น:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

มันเป็นเรื่องธรรมดาที่จะตั้งชื่อฟังก์ชันบูลีนด้วยชื่อที่ฟังคล้ายกับคำถามใช่หรือไม่ (yes/no question); ฟังก์ชัน `is_divisible` คืนค่ากลับมาเป็นถ้าไม่ `True` ก็ `False` เพื่อระบุว่า `x` ถูกหารด้วย `y` ลงตัวหรือไม่

นี่คือตัวอย่าง:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

ผลลัพธ์ของตัวดำเนินการ `==` คือค่าบูลีน ดังนั้น เราสามารถเขียนฟังก์ชันให้กระชับมากยิ่งขึ้นด้วยการคืนค่าการดำเนินการไปตรง ๆ เลย:

```
def is_divisible(x, y):
    return x % y == 0
```

ฟังก์ชันบูลีนนิยมใช้ในคำสั่งเงื่อนไข (conditional statement):

```
if is_divisible(x, y):
    print('x is divisible by y')
```

เราอาจจะอยากเขียนแบบนี้:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

แต่การเปรียบเทียบที่เพิ่มขึ้นมานั้นไม่จำเป็น

เพื่อเป็นการฝึกทำ ให้เขียนฟังก์ชันที่ชื่อว่า `is_between(x, y, z)` ที่คืนค่า `True` ถ้า $x \leq y \leq z$ หรือไม่เช่นนั้นให้คืนค่า `False`

6.5. การย้อนเรียกใช้เพิ่มเติม

เราได้เรียนรู้ไปเพียงแค่ซับเซ็ตเล็ก ๆ ของไพธอน แต่เราอาจจะสนใจที่รู้ว่าซับเซ็ตอันนี้เป็นภาษาโปรแกรมที่ *สมบูรณ์* แล้ว ซึ่งหมายความว่า อะไรก็ตามที่สามารถคำนวณได้ จะสามารถเขียนแทนได้ด้วยภาษานี้ โปรแกรมใด ๆ ที่เคยถูกเขียนมาจะสามารถถูกเขียนใหม่ได้ โดยใช้แค่คุณลักษณะที่เราได้เรียนมาจนบัดนี้ (ที่จริงแล้ว แล้วต้องใช้คำสั่งอย่างอื่นอีกหน่อย เพื่อที่จะควบคุมเม้าส์ ดิสก์ และอื่น ๆ แต่ก็เท่านั้นแหละ)

การพิสูจน์ข้อกล่าวอ้างนี้ เป็นอะไรที่ไม่ง่ายนักซึ่งทำสำเร็จเป็นครั้งแรกโดยอลัน ทัวริง (Alan Turing) หนึ่งในนักวิทยาการคอมพิวเตอร์คนแรก ๆ (บางคนอาจจะเถียงว่าเขาเป็นนักคณิตศาสตร์ แต่ว่า นักวิทยาการคอมพิวเตอร์หลายคนก็เริ่มด้วยการเป็นนักคณิตศาสตร์) ดังนั้น มันจึงเป็นที่รู้จักกัน ในชื่อ ข้อวินิจฉัยของทัวริง (Turing Thesis) ผมแนะนำให้อ่านหนังสือของไมเคิล ซิปเซอร์ (Michael Sipser) ที่ชื่อว่า *Introduction to the Theory of Computation*

เพื่อที่จะทำให้เห็นว่าเราสามารถทำอะไรกับเครื่องมือที่เราเรียนมาได้บ้าง เราจะประเมินฟังก์ชันทางคณิตศาสตร์ที่ถูกเขียนแบบย้อนเรียกใช้ (recursive) สองสามตัวอย่าง นิยามการย้อนเรียกใช้นั้นเหมือนกับ นิยามการเวียน (circular definition) ในแง่ที่ว่านิยามมีการอ้างอิงไปยังสิ่งที่มันกำลังถูกนิยามอยู่ (อ้างอิงไปถึงตัวมันเอง) นิยามการเวียนของจริงไม่ค่อยมีประโยชน์เท่าไรนัก:

vorpal: คือ คุณศัพท์ที่ใช้บรรยายบางสิ่งที่มีลักษณะ vorpal

ถ้าเราเห็นนิยามนี้ในพจนานุกรม เราอาจจะรำคาญได้ ในอีกแง่หนึ่ง ถ้าเราเปิดดูนิยามของฟังก์ชัน แฟกทอ

เรียล ที่แสดงด้วยเครื่องหมาย ! เราก็จะได้อะไรประมาณนี้

$$0! = 1$$
$$n! = n(n-1)!$$

นิยามนี้กล่าวว่า แฟกทอเรียลของ 0 คือ 1 และแฟกทอเรียลของค่าอื่น ๆ, n, คือ n คูณ ด้วยค่าแฟกทอเรียลของ $n-1$

ดังนั้น 3! คือ 3 คูณกับ 2!, ซึ่งคือ 2 คูณกับ 1!, ซึ่งคือ 1 คูณกับ 0!. พอเอามารวม ๆ กันแล้ว, 3! เท่ากับ 3 คูณกับ 2 คูณกับ 1 คูณกับ 1, ซึ่งคือ 6.

ถ้าเราสามารถเขียนนิยามการย้อนเรียกใช้ของอะไรสักอย่าง เราก็สามารถเขียนโปรแกรมไพธอนเพื่อที่จะประเมินผลมัน ขั้นตอนแรกคือการตัดสินใจว่าพารามิเตอร์ควรจะเป็นอะไร ในกรณีนี้ มันควรจะชัดเจนว่าฟังก์ชัน `factorial` รับค่าเป็นจำนวนเต็ม:

```
def factorial(n):
```

ถ้าอาร์กิวเมนต์ดันเป็น 0 สิ่งที่เราต้องทำก็คือ คืนค่า 1 กลับไป:

```
def factorial(n):
    if n == 0:
        return 1
```

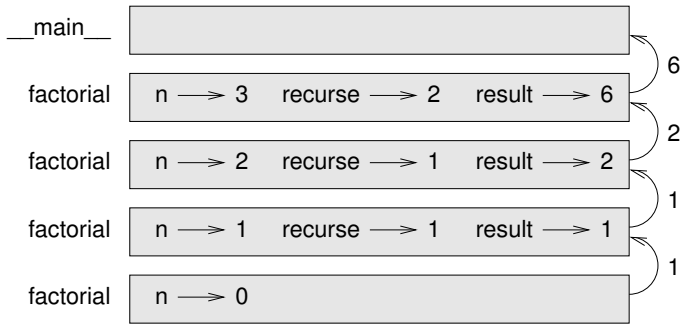
ไม่เช่นนั้น, และนี่คือส่วนที่น่าสนใจ, เราจะต้องเรียกซ้ำเพื่อหาค่าแฟกทอเรียลของ $n-1$ และคูณด้วย n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

กระแสการดำเนินการสำหรับโปรแกรมนี้จะเหมือนกับกระแสของคำสั่งใน `countdown` ในหัวข้อที่ 5.8 ถ้าเราเรียกฟังก์ชัน `factorial` ด้วยค่า 3:

เนื่องจาก 3 ไม่ใช่ 0 เราจะไปทำแขนงที่สองและคำนวณแฟกทอเรียลของ `n-1` ...

เนื่องจาก 2 ไม่ใช่ 0 เราจะไปทำแขนงที่สองและคำนวณแฟกทอเรียลของ `n-1` ...



รูปที่ 6.1.: แผนภาพแบบกองซ้อน

> เนื่องจาก 1 ไม่ใช่ 0 เราจะไปทำแขนงที่สองและคำนวณแฟกทอเรียล ของ `n-1` ...
>
> > เนื่องจาก 0 เท่ากับ 0 เราทำแขนงแรก และคืนค่า 1 โดย ไม่ต้องเรียกซ้ำอีก
> >
> > ค่าที่ถูกคืนกลับมา ซึ่งคือ 1 จะถูกคูณด้วย n ซึ่งคือ 1 และผลลัพธ์ก็จะ ถูกส่งคืนกลับไป
>
> ค่าที่ถูกคืนกลับมา ซื่อคือ 1 จะถูกคูณด้วย n ซึ่งคือ 2 และผลลัพธ์ก็จะถูกส่งคืนกลับไป

ค่าที่ถูกคืนกลับมา (2) จะถูกคูณด้วย n ซึ่งคือ 3 และผลลัพธ์ 6 ก็จะกลายเป็นค่าที่ถูกส่งคืนกลับไปยังจุดที่เรียกฟังก์ชันที่เริ่มกระบวนการทั้งหมดนี้

รูปที่ 6.1 แสดงให้เห็นว่าแผนภาพแบบกองซ้อนเป็นอย่างไร สำหรับลำดับการเรียกฟังก์ชันนี้

ในรูป *ค่าคืนกลับ*ถูกแสดงให้เห็นว่า กำลังถูกส่งคืนย้อนกลับขึ้นไปบนกอง (stack) ในแต่ละกรอบ ค่าคืนกลับคือค่าของ `result` ซึ่งเป็นผลคูณของ `n` และ `recurse`

ในกรอบสุดท้าย มันไม่มี*ตัวแปรเฉพาะที่* `recurse` และ `result` เพราะว่า แขนงที่ทำให้มันเกิดขึ้นนั้นไม่ได้ทำงาน

6.6. ก้าวแห่งศรัทธา

การไล่ตามกระแสการดำเนินการเป็นหนทางหนึ่งที่จะอ่านโปรแกรม แต่มันก็สามารถทำให้เรารู้สึกท่วมท้นได้เร็ว อีกทางเลือกหนึ่ง คือ สิ่งที่ผมเรียกว่า “ก้าวแห่งศรัทธา (Leap of faith)” เมื่อเรามาถึง การเรียก

ฟังก์ชัน แทนที่จะไล่ตามกระแสการดำเนินการ เราจะ *สมมติ* ว่าฟังก์ชันทำงานได้อย่าง ถูกต้อง และคืนค่าที่ถูกต้องกลับมาให้

ที่จริงแล้ว เราได้ฝึกก้าวแห่งศรัทธามาแล้ว เมื่อเราใช้ฟังก์ชันสำเร็จรูป (built-in) เมื่อเราเรียก `math.cos` หรือ `math.exp` เราไม่ได้สำรวจส่วนตัวของฟังก์ชันดังกล่าวเลย เราแค่สมมติว่ามันทำงานได้ เพราะว่าคนที่เขียนฟังก์ชันพวกนี้เป็นโปรแกรมเมอร์ที่เก่ง

เป็นจริงเช่นเดียวกันเมื่อเราเรียกฟังก์ชันของเราเอง เช่น ในหัวข้อ 6.4 เราเขียนฟังก์ชันที่ชื่อว่า `is_divisible` ซึ่งหาว่าเลขหนึ่งถูกเลขอีกตัวหนึ่งหารลงตัวหรือไม่ เมื่อเรามั่นใจแล้วว่าฟังก์ชันนี้น่าจะทำงานถูกต้อง—โดยการสำรวจโค้ดและทดสอบมัน—เราสามารถใช้ฟังก์ชันนี้ไปเลยโดยไม่ต้องดูส่วนของตัวฟังก์ชันอีก

เป็นจริงเช่นเดียวกันกับโปรแกรมย้อนเรียกใช้ เมื่อเราทำการย้อนเรียกใช้ แทนที่เราจะไล่ตามกระแสการดำเนินการ เราควรจะสมมติว่าการย้อนเรียกใช้นั้นทำงานได้ (คืนค่าที่ถูกต้องกลับมา) และจากนั้นให้ถามตัวเองว่า “สมมติว่าเราสามารถหาค่าแฟกทอเรียบของ $n-1$ แล้ว เราจะสามารถคำนวณค่าแฟกทอเรียลของ n ได้หรือเปล่า” มันชัดเจนว่าเราสามารถทำได้ โดยการคูณค่า n เข้าไป

แน่นอนว่ามันแปลกหน่อย ๆ ที่จะสมมติว่าฟังก์ชันมันทำงานได้ถูกต้อง เมื่อเรายังเขียนไม่เสร็จเลย แต่มันก็เป็นเหตุผลที่ว่าทำไมเราถึงเรียกมันว่า ก้าวแห่งศรัทธา ยังไงล่ะ!

6.7. อีกตัวอย่างหนึ่ง

หลังจากฟังก์ชัน `factorial` แล้ว ตัวอย่างที่นิยมใช้ของฟังก์ชันทางคณิตศาสตร์แบบเรียกซ้ำ คือ ฟังก์ชัน `fibonacci` ซึ่งมีนิยามดังนี้ (ดู `http://en.wikipedia.org/wiki/Fibonacci_number`):

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

เมื่อแปลมาเป็นไพธอนจะมีหน้าตาแบบนี้:

```
def fibonacci(n):
    if n == 0:
```

```
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

ถ้าเราพยายามที่จะไล่ตามกระแสการดำเนินการตรงนี้ แม้แต่เป็นค่าน้อย ๆ ของ n หัวของเราจะระเบิด แต่จากหลักการปล่อยให้เป็นไปตามโชคชะตาแล้ว ถ้าเราสมมติว่าการย้อนเรียกใช้ทั้งสองนั้นทำงาน ได้อย่างถูกต้องแล้ว มันก็จะชัดเจนว่าเราจะได้ผลลัพธ์ที่ถูกต้องจากการบวกค่าทั้งสองเข้าด้วยกัน

6.8. การตรวจสอบชนิดของข้อมูล

จะเกิดอะไรขึ้นถ้าเราเรียกฟังก์ชัน `factorial` และผ่านค่า 1.5 เป็นอาร์กิวเมนต์?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

ดูเหมือนว่ามันเป็นการย้อนเรียกใช้ไม่รู้จบ เป็นไปได้ยังไงกัน? ฟังก์ชันนี้มีกรณีฐาน—เมื่อ `n == 0` แต่ถ้า `n` ไม่ใช่จำนวนเต็ม เราสามารถจะ *พลาด* การเจอกรณีฐาน และเรียกซ้ำไปชั่วนิรันดร์ได้

ในการย้อนเรียกใช้ครั้งแรก ค่าของ `n` คือ 0.5 ส่วนในครั้งถัดไปคือ -0.5 จากนั้น ค่ามันก็จะน้อยลงเรื่อย ๆ (เป็นลบมากขึ้น) แต่มันจะไม่มีวันเป็น 0

เรามีสองทางเลือก เราสามารถที่จะทำให้ฟังก์ชัน `factorial` ครอบคลุมถึงเลขทศนิยม หรือเราสามารถทำให้ `factorial` ตรวจสอบชนิดของอาร์กิวเมนต์ได้ ทางเลือกแรก เรียกว่า ฟังก์ชันแกมม่า (gamma function) และมันเกินขอบเขตของหนังสือเล่มนี้ ดังนั้น เราจะทำทางเลือกที่สอง

เราสามารถใช้ฟังก์ชันพร้อมใช้ที่ชื่อว่า `isinstance` เพื่อตรวจสอบชนิดของอาร์กิวเมนต์ ในเวลาเดียวกัน เราสามารถตรวจสอบว่ามันเป็นจำนวนบวกหรือไม่ได้ด้วย

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
       print('Factorial is not defined for negative integers.')
```

```
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

กรณีฐานแรกจัดการกับเลขที่ไม่ใช่จำนวนเต็ม; กรณีฐานที่สองจัดการกับจำนวนเต็มลบ ในทั้งสองกรณี โปรแกรมพิมพ์ข้อความแจ้งและคืนค่าเป็น None เพื่อระบุว่าบางสิ่งบางอย่างนั้นไม่ถูกต้อง:

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

ถ้าเราสามารถผ่านการตรวจสอบทั้งสองอย่างได้ เรารู้ว่า n เป็นจำนวนบวก หรือ ศูนย์ ดังนั้น เราสามารถพิสูจน์ได้ว่าการย้อนเรียกใช้นั้นจะสิ้นสุด

โปรแกรมนี้สาธิตรูปแบบที่บางทีเรียกว่า **ผู้พิทักษ์ (guardian)** เงื่อนไขสองข้อแรกทำตัวเป็นผู้พิทักษ์ป้องกันโค้ดจากค่าที่ทำให้เกิดข้อผิดพลาดขึ้นได้ ผู้พิทักษ์ทำให้การพิสูจน์ความถูกต้องของโค้ดนั้นเป็นไปได้

ในหัวข้อที่ 11.4 เราจะเห็นทางเลือกที่ยืดหยุ่นกว่านี้ในการพิมพ์ข้อความแจ้งข้อผิดพลาด: การแจ้งเอ็กเซ็ปชั่น (raising an exception)

6.9. การดีบัก

การแตกโปรแกรมใหญ่ ๆ ให้เป็นฟังก์ชันเล็ก ๆ ทำให้เกิดจุดตรวจสอบโค้ด (checkpoints for debugging) โดยธรรมชาติ ถ้าฟังก์ชันใดทำงานไม่ได้ มีความเป็นไปได้สามอย่างที่จะต้องพิจารณา:

- มีอะไรบางอย่างผิดปกติเกี่ยวกับอาร์กิวเมนต์ที่ได้รับมา; เงื่อนไขก่อนผิด
- มีอะไรบางอย่างผิดปกติเกี่ยวกับฟังก์ชันเอง; เงื่อนไขลงท้ายผิด
- มีอะไรบางอย่างผิดปกติเกี่ยวกับค่าคืนกลับ หรือเกี่ยวกับวิธีที่ค่านั้นถูกใช้

เพื่อที่จะตัดความเป็นไปได้ข้อแรกออกไป เราสามารถเพิ่มคำสั่ง `print` ตอนเริ่มฟังก์ชัน และแสดงค่าของพารามิเตอร์ต่าง ๆ (และอาจจะชนิดของข้อมูลด้วย) หรือเราสามารถเขียนโค้ดที่ตรวจสอบเงื่อนไขก่อนโดยตรงเลย

ถ้าพารามิเตอร์ก็ดูใช้ได้ดี ให้เพิ่มคำสั่ง `print` ก่อนคำสั่ง return แต่ละอัน และแสดงค่าคืนกลับ ถ้าเป็นไปได้ ให้ตรวจสอบผลลัพธ์โดยมือ พิจารณาการเรียกฟังก์ชันด้วยค่าที่ง่ายต่อการตรวจสอบผลลัพธ์ที่ถูกต้อง (เหมือนในหัวข้อที่ 6.2)

ถ้าฟังก์ชันก็น่าจะทำงานถูกต้อง ให้ดูที่การเรียกฟังก์ชันเพื่อที่จะมั่นใจว่าค่าที่ถูกส่งคืนกลับมานั้น ถูกนำมาใช้ได้อย่างถูกต้อง (หรือถูกนำมาใช้จริง ๆ!)

การเพิ่มคำสั่ง print ในตอนเริ่มและจบของฟังก์ชัน ช่วยให้กระแสการดำเนินการนั้นชัดเจนขึ้น เช่น นี่เป็นเวอร์ชันอันหนึ่งของฟังก์ชัน `factorial` ที่มีคำสั่ง print อยู่:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` เป็นสายอักขระของอักขระเว้นวรรค ที่ควบคุมการย่อหน้าของเอาต์พุต นี่คือผลการทำงานของ `factorial(4)`:

```
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
```

```
    returning 1
        returning 2
            returning 6
                returning 24
```

ถ้าเรางงเกี่ยวกับกระแสการดำเนินการ เอาต์พุตลักษณะนี้จะมีประโยชน์ มันใช้เวลาระยะหนึ่งในการพัฒนานั่งร้านที่มีประสิทธิภาพ แต่การสร้างนั่งร้านขึ้นมาเล็กน้อยนั้นสามารถประหยัดการดีบักได้มาก

6.10. อภิธานศัพท์

ตัวแปรชั่วคราว (temporary variable): ตัวแปรที่ใช้เก็บค่าระหว่างทางในการคำนวณที่ซับซ้อน

โค้ดตาย (dead code): ส่วนของโปรแกรมที่ไม่มีวันทำงาน บ่อยครั้งเพราะมันอยู่หลังจากคำสั่ง `return`

การพัฒนาแบบเพิ่มส่วน (incremental development): แผนการพัฒนาโปรแกรมที่มุ่งหลีกเลี่ยงการดีบัก โดยการคูณค่า เพิ่มและทดสอบโค้ดทีละน้อย

นั่งร้าน (scaffolding): โค้ดที่ใช้ในระหว่างการพัฒนาโปรแกรม แต่ไม่ใช่ส่วนหนึ่งของโปรแกรมเวอร์ชันสุดท้าย

ผู้พิทักษ์ (guardian): รูปแบบการเขียนโปรแกรมที่ใช้คำสั่งเงื่อนไขเพื่อตรวจสอบและจัดการกับเหตุการณ์ที่อาจจะทำให้เกิดข้อผิดพลาด

6.11. แบบฝึกหัด

***แบบฝึกหัด* 6.1.** *ให้เขียนแผนภาพแบบกองซ้อนสำหรับโปรแกรมต่อไปนี้ โปรแกรมนี้พิมพ์อะไรออกมา?*

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
```

```
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

แบบฝึกหัด 6.2. *ฟังก์ชันแอคเคอร์มานน์, $A(m,n)$, นิยามว่า:*

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

ดูเพิ่มเติมที่ `http://en.wikipedia.org/wiki/Ackermann_function`.

ให้เขียนฟังก์ชันที่ชื่อว่า `ack` *ซึ่งประเมินค่าของฟังก์ชันแอคเคอร์มานน์ ใช้ฟังก์ชันของเราเพื่อหา* `ack(3, 4)` *ซึ่งควรจะมีค่าเป็น 125 เกิดอะไรขึ้นกับค่าของ* `m` *และ* `n` *ที่มากกว่านี้? เฉลย:* `http://thinkpython2.com/code/ackermann.py`.

แบบฝึกหัด 6.3. *พาลินโดรม (palindrome) คือ คำที่สะกดเหมือนกันทั้งเวลาเริ่มอ่านจากข้างหน้าและเริ่มอ่านจากข้างหลัง เช่น "noon" และ "redivider" ถ้าคิดแบบการย้อนเรียกใช้ คำใด ๆ เป็นพาลินโดรม เมื่ออักษรตัวแรกและตัวสุดท้ายเป็นตัวเดียวกัน และตรงกลางเป็นพาลินโดรม*

ต่อไปนี้ คือ ฟังก์ชันที่รับอาร์กิวเมนต์ที่เป็นสายอักขระและส่งค่าคืนกลับเป็น อักษรตัวแรก อักษรตัวสุดท้าย และกลุ่มอักษรตรงกลาง:

```
def first(word):
    return word[0]
```

```
def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

เราจะเห็นว่ามันทำงานอย่างไรในบทที่ 8

1. *ให้พิมพ์ฟังก์ชันเหล่านี้ลงในไฟล์ชื่อว่า* **palindrome.py** *และทดสอบมันดู จะเกิดอะไรขึ้นถ้าเราเรียก* **middle** *ด้วยสายอักขระที่มีสองตัวอักษร? แล้วถ้าเรียกด้วยตัวเดียวล่ะ? แล้วถ้าเป็นสายอักขระว่าง (empty string) ซึ่งเขียนว่า* **''** *และไม่มีตัวอักษรเลยล่ะ?*

2. *ให้เขียนฟังก์ชันชื่อว่า* **is_palindrome** *ซึ่งรับอาร์กิวเมนต์ที่เป็นสายอักขระ และคืนค่า* **True** *ถ้าเป็นพาลินโดรม และ* **False** *ถ้าไม่ใช่ จำไว้ว่าเราสามารถใช้ฟังก์ชันพร้อมใช้* **len** *เพื่อที่จะตรวจสอบความยาวของสายอักขระ*

เฉลย: **http://thinkpython2.com/code/palindrome_soln.py**.

***แบบฝึกหัด* 6.4.** *เลข a เป็นค่ายกกำลังของ b หากมันสามารถหารด้วย b ลงตัว และ a/b เป็นค่ายกกำลังของ b ให้เขียนการเรียกฟังก์ชัน* **is_power** *ซึ่งรับพารามิเตอร์ **a** และ **b** และคืนค่า* **True** *ถ้า **a** เป็นค่ายกกำลังของ **b**. หมายเหตุ: เราจะต้องคิดเรื่องกรณีฐานด้วย*

***แบบฝึกหัด* 6.5.** *ตัวหารร่วมมาก (ห.ร.ม.) หรือ the greatest common divisor (GCD) ของ a และ b คือเลขจำนวนที่มากที่สุดที่สามารถหารพวกมันได้โดยไม่มีเศษ*

วิธีหนึ่งที่จะหาค่า ห.ร.ม. ของเลขสองตัวมาจากข้อสังเกตที่ว่า ถ้า r เป็นเศษของการหาร เมื่อ a ถูกหารด้วย b แล้ว $gcd(a,b) = gcd(b,r)$ สำหรับกรณีฐาน เราสามารถใช้ $gcd(a,0) = a$.

ให้เขียนฟังก์ชันที่ชื่อว่า **gcd** *ซึ่งรับพารามิเตอร์ **a** และ **b** และคืนค่า ตัวหารร่วมมาก*

เครดิต: แบบฝึกหัดนี้มาจากตัวอย่างจากหนังสือของ Abelson และ Sussman ชื่อว่า Structure and Interpretation of Computer Programs

7. การวนซ้ำ

เนื้อหาในบทนี้เกี่ยวกับการวนซ้ำ ซึ่งเป็นความสามารถในการรันก้อนชุดคำสั่งซ้ำไปมา เราได้เห็นการวนซ้ำแบบหนึ่งไปแล้วโดยการใช้การย้อนเรียกใช้ ในหัวข้อที่ 5.8 เราได้เห็นอีกแบบ โดยใช้ลูป `for` ในหัวข้อที่ 4.2 ในบทนี้ เราจะเห็นอีกแบบ โดยการใช้คำสั่ง `while` แต่ก่อนอื่น ผมอยากจะพูดอะไรนิดหน่อยเกี่ยวกับการกำหนดค่าให้ตัวแปร

7.1. การกำหนดค่าให้ใหม่

เราอาจจะค้นพบแล้วว่า เราสามารถกำหนดค่าให้ตัวแปรตัวเดิมได้มากกว่าหนึ่งครั้ง การกำหนดค่าครั้งใหม่ทำให้ตัวแปรนั้นอ้างอิงไปที่ค่าใหม่ (และหยุดอ้างอิงไปที่ค่าเก่า)

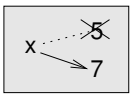
```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

ครั้งแรกที่เราแสดงค่า `x` มันเป็น 5; พอครั้งที่สอง มันเป็น 7

รูปที่ 7.1 แสดงให้เห็นว่าการกำหนดค่าให้ใหม่เป็นอย่างไรในแผนภาพสถานะ

ถึงตรงนี้ ผมอยากจะพูดถึงจุดที่สับสนกันมาก เพราะว่าไพธอนใช้เครื่องหมายเท่ากับ (=) เพื่อการกำหนดค่า มันเลยทำให้เราอยากจะแปล คำสั่งเช่น `a = b` ว่าเป็นการแสดงการเท่ากันทางตรรกะ; นั่นคือ การที่บอกว่า `a` และ `b` นั้นเท่ากัน แต่การแปลความแบบนี้มันผิด

อย่างแรก การเท่ากันเป็นความสัมพันธ์แบบสมมาตร แต่การกำหนดค่าไม่ใช่ เช่น ในคณิตศาสตร์ ถ้า $a = 7$ แล้ว $7 = a$ แต่ในไพธอน คำสั่ง `a = 7` นั้นทำได้ และ `7 = a` นั้นทำไม่ได้



รูปที่ 7.1.: แผนภาพสถานะของตัวแปร

นอกจากนี้ ในคณิตศาสตร์ การเท่ากันทางตรรกะมีค่าเป็น จริง หรือ เท็จ ตลอดเวลา ถ้า $a = b$ ในตอนนี้แล้ว a จะเท่ากับ b เสมอ ในไพธอน การกำหนดค่าอาจจะทำให้ตัวแปรสองตัวมีค่าเท่ากัน แต่มันจะไม่เป็นแบบนั้นตลอดไป:

```
>>> a = 5
>>> b = a    # a and b are now equal
>>> a = 3    # a and b are no longer equal
>>> b
5
```

บรรทัดที่สามเปลี่ยนค่าของ **a** แต่ไม่เปลี่ยนค่าของ **b** ดังนั้น มันจึงไม่เท่ากันแล้ว

การกำหนดค่าให้ใหม่นั้นบ่อยครั้งมีประโยชน์ แต่เราควรจะต้องใช้มันอย่างระมัดระวัง ถ้าค่าของตัวแปรนั้นเปลี่ยนบ่อย ๆ มันอาจจะทำให้อ่านโค้ดและดีบักได้ยากขึ้น

7.2. การปรับค่าตัวแปร

การกำหนดค่าให้ใหม่ที่ทำเป็นปกติ คือ **การปรับค่าตัวแปร (update)** โดยที่ค่าใหม่จะขึ้นอยู่กับค่าเก่า

```
>>> x = x + 1
```

มันหมายความว่า “เอาค่าปัจจุบันของ **x**, บวกหนึ่ง, แล้วปรับค่า **x** ให้เป็นค่าใหม่”

ถ้าเราพยายามที่จะปรับค่าตัวแปรที่ไม่มีอยู่จริง เราจะได้ข้อผิดพลาด เพราะว่าไพธอนจะประเมินค่าทางขวาก่อนที่จะกำหนดค่าให้ตัวแปร **x**:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

ก่อนที่เราจะสามารถปรับค่าตัวแปรได้ เราจะต้อง **ให้ค่าเริ่มต้น (initialize)** กับมันก่อน โดยปกติแล้ว จะใช้การกำหนดค่าแบบง่าย ๆ:

```
>>> x = 0
>>> x = x + 1
```

การปรับค่าตัวแปรโดยการบวก 1 เข้าไปเรียกว่า **การเพิ่มค่า (increment)**; ส่วนการลบออก 1 เรียกว่า **การลดค่า (decrement)**

7.3. คำสั่ง while

บ่อยครั้งที่คอมพิวเตอร์เคยชินกับการทำงานซ้ำ ๆ แบบอัตโนมัติ การทำงานอะไรที่เหมือน ๆ กันโดยไม่มีข้อผิดพลาด คือ สิ่งที่คอมพิวเตอร์สามารถทำงานได้ดี และคนทำได้ไม่ดีนัก ในโปรแกรมคอมพิวเตอร์ การทำงานซ้ำ ๆ เรียกอีกอย่างหนึ่งว่า **การวนซ้ำ (iteration)**

เราได้เห็นฟังก์ชันไปแล้วสองฟังก์ชัน, `countdown` และ `print_n`, ที่วนซ้ำโดยการใช้การย้อนเรียกใช้ เพราะว่าการวนซ้ำเป็นอะไรที่ปกติธรรมดามาก ไพธอนจึงเตรียมคุณลักษณะของภาษาที่ทำให้มันง่ายขึ้น หนึ่งคือคำสั่ง `for` ที่เราเห็นในหัวข้อที่ 4.2 เราจะกลับไปพูดเรื่องนั้นทีหลัง

อีกคำสั่งหนึ่ง คือ คำสั่ง `while` นี่คือเวอร์ชันของฟังก์ชัน `countdown` ที่ใช้คำสั่ง `while`:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

เราสามารถอ่านคำสั่ง `while` เหมือนกับว่าเป็นภาษาอังกฤษทั่วไปเลย มันหมายความว่า "ในขณะที่ `n` มีค่ามากกว่า 0, ให้แสดงค่าของ `n` และจากนั้นให้ลดค่า `n` ลงหนึ่ง เมื่อเราได้ค่าเป็น 0 แล้ว ให้แสดงคำว่า `Blastoff!`"

นี่คือกระแสการดำเนินการสำหรับคำสั่ง `while` อย่างเป็นทางการ:

1. ประเมินว่าเงื่อนไขเป็น จริง หรือ เท็จ
2. ถ้าเป็นเท็จ ให้ออกจากคำสั่ง `while` และทำคำสั่งถัดไปเลย
3. ถ้าเงื่อนไขเป็นจริง ให้รันส่วนตัวของคำสั่ง while ให้ครบ แล้วกลับไปที่ขั้นตอนที่ 1

กระแสการทำงานแบบนี้เรียกว่า ลูป (loop) เพราะว่าขั้นตอนที่ 3 จะทำการวนกลับ (ลูปกลับ) ย้อนไปข้างบน

ส่วนที่เป็นตัวของลูปควรจะเปลี่ยนค่าของตัวแปรอย่างน้อยหนึ่งตัว เพื่อที่จะทำให้เงื่อนไขกลายเป็น เท็จ (False) ในที่สุด และทำให้ลูปหยุดทำงาน ไม่เช่นนั้น ลูปจะทำงานไปเรื่อย ๆ ชั่วนิรันดร์ ซึ่งเรียกว่า **ลูปไม่รู้จบ (infinite loop)** แหล่งของความบันเทิงแบบไม่สิ้นสุดของนักวิทยาการคอมพิวเตอร์ คือ การสังเกตพบว่า วิธีใช้งานของยาสระผม คือ “สระให้เกิดฟอง, ล้างออก, ทำซ้ำอีกรอบ” นั้นเป็นลูปไม่รู้จบ

ในกรณีของฟังก์ชัน `countdown` เราสามารถพิสูจน์ได้ว่า ลูปจะหยุดทำงาน: ถ้า `n` เป็นศูนย์หรือลบ ลูปจะไม่ทำงานเลย ไม่เช่นนั้น ค่า `n` จะน้อยลงในแต่ละรอบของการลูป ทำให้ในที่สุด เราก็จะต้องได้ 0

สำหรับลูปอื่น ๆ บางลูป มันไม่ง่ายนักที่จะพิสูจน์แบบนั้น เช่น:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n is even
            n = n / 2
        else:                 # n is odd
            n = n*3 + 1
```

เงื่อนไขสำหรับลูปนี้ คือ `n != 1` ดังนั้น ลูปจะวนไปเรื่อย ๆ จนกว่า `n` เป็น `1` ซึ่งทำให้เงื่อนไขนั้นเป็นเท็จ

แต่ละครั้งที่ทำลูป โปรแกรมจะเอาต์พุตค่าของ `n` และจากนั้นจะตรวจสอบว่ามันเป็นจำนวนคู่หรือจำนวนคี่ ถ้าเป็นจำนวนคู่ `n` จะถูกหารด้วย 2 แต่ถ้าเป็นจำนวนคี่ ค่าของ `n` จะถูกแทนด้วย `n*3 + 1` เช่น ถ้าอาร์กิวเมนต์ที่ผ่านเข้าไปใน `sequence` คือ 3 ผลของ `n` จะเป็น 3, 10, 5, 16, 8, 4, 2, 1

เนื่องจากบางครั้ง `n` มีค่าเพิ่มขึ้นและบางครั้งมีค่าลดลง เลยพิสูจน์ไม่ได้อย่างชัดเจนว่า `n` จะมีค่าถึง 1 หรือไม่ หรือโปรแกรมจะหยุดทำงานหรือไม่ สำหรับเฉพาะบางค่าของ `n` เราสามารถพิสูจน์ได้ว่าโปรแกรมจะหยุดทำงาน เช่น ถ้าค่าเริ่มต้นเป็นค่าที่ยกกำลังสอง `n` จะเป็นจำนวนคู่ในทุก ๆ รอบของลูปจนกระทั่งถึงค่า 1 ตัวอย่างที่แล้วจบด้วยลำดับอย่างว่าโดยเริ่มที่ค่า 16

คำถามที่ยาก คือ เราสามารถพิสูจน์ได้หรือไม่ว่าโปรแกรมจะจบสำหรับค่า `n` ที่เป็นบวก *ทุกค่า* จนถึงตอนนี้ ไม่มีใครสามารถพิสูจน์ว่ามันได้ *หรือ* พิสูจน์ว่ามันไม่ได้! (ให้ดู `http://en.wikipedia.org/wiki/Collatz_conjecture`.)

เพื่อเป็นการฝึกทำ ให้เขียนฟังก์ชัน `print_n` ใหม่จากหัวข้อที่ 5.8 โดยใช้การวนซ้ำ (iteration) แทนการใช้การย้อนเรียกใช้ (recursion)

7.4. คำสั่ง break

บางครั้ง เราไม่รู้ว่าเมื่อไหร่จะต้องหยุดลูป จนกระทั่งเรามาถึงครึ่งทางของลูป ในกรณีนั้น เราสามารถใช้คำสั่ง break ในการกระโดดออกจากลูป

ตัวอย่างเช่น สมมติว่าเราต้องเอาอินพุตมาจากผู้ใช้จนกระทั่งเขาพิมพ์เข้ามาว่า done เราสามารถเขียนว่า:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

เงื่อนไขของลูปนี้ คือ True ซึ่งจะเป็นจริงเสมอ ดังนั้น ลูปจะรันจนกว่าจะเจอคำสั่ง break

ในแต่ละรอบของลูป โปรแกรมจะขอข้อมูลจากผู้ใช้โดยใช้เครื่องหมายวงเล็บสามเหลี่ยมแบบเปิด (<) ถ้าผู้ใช้พิมพ์เข้ามาว่า done คำสั่ง break จะทำให้ออกจากลูป ไม่เช่นนั้น โปรแกรมจะสะท้อนสิ่งที่ผู้ใช้พิมพ์เข้ามา แล้วก็กลับไปยังข้างบนสุดของลูป นี่คือตัวอย่างการทำงาน:

```
> not done
not done
> done
Done!
```

การเขียนลูป while แบบนี้เป็นเรื่องปกติ เพราะว่าเราสามารถตรวจสอบเงื่อนไขที่ไหนก็ได้ในลูป (ไม่เพียงแค่ในส่วนบนสุดของลูป) และเราสามารถยืนยันเงื่อนไขการหยุดได้ (“หยุดเมื่อสิ่งนี้เกิดขึ้น”) แทนที่จะบอกในทางลบ (“ทำงานไปเรื่อย ๆ จนกว่าสิ่งนั้นจะเกิด”)

7.5. รากที่สอง

ลูปมักถูกใช้ในโปรแกรมที่คำนวณผลลัพธ์เชิงตัวเลข โดยการเริ่มด้วยค่าประมาณ และวนซ้ำเพื่อทำคำตอบให้ดีขึ้น

ตัวอย่างเช่น วิธีหนึ่งของการคำนวณรากที่สอง คือ วิธีของนิวตัน (Newton's method) สมมติว่าเราต้องการจะหารากที่สองของ a ถ้าเราเริ่มด้วยค่าประมาณสักอย่าง, x, เรา สามารถคำนวณค่าประมาณที่ดีกว่าเดิมโดยใช้สูตรต่อไปนี้:

$$y = \frac{x + a/x}{2}$$

ตัวอย่างเช่น ถ้า a คือ 4 และ x คือ 3

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

ผลลัพธ์ที่ได้จะใกล้เคียงกับคำตอบที่ถูกมากขึ้น ($\sqrt{4} = 2$) ถ้าเราทำกระบวนการนี้ซ้ำ ด้วยการประมาณค่าใหม่ มันก็จะใกล้เข้าไปอีก:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

หลังจากการปรับค่าอีกสองสามครั้ง ค่าประมาณก็เกือบจะเป๊ะ:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

โดยทั่วไปแล้ว เราไม่รู้ก่อนว่าจะต้องทำกี่ขั้นตอนจนกว่าจะได้คำตอบที่ถูกต้อง แต่เรารู้ เมื่อเราได้ค่านั้นมาแล้วเพราะว่าค่าประมาณนั้นไม่เปลี่ยนแล้ว:

```
>>> x = y
```

```
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

เมื่อ `y == x` เราสามารถหยุดได้ นี่คือลูปที่เริ่มด้วยค่าประมาณการเริ่มต้น, `x`, และปรับค่าให้ดีขึ้นเรื่อย ๆ จนกระทั่งมันหยุดเปลี่ยน:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

สำหรับค่า `a` ส่วนมาก วิธีนี้ทำงานได้ดี แต่โดยทั่วไปแล้วมันอันตรายถ้าจะทดสอบ การเท่ากันของค่าเลขทศนิยม `float` ค่าเลขทศนิยมเป็นเพียงแค่ค่าประมาณ: จำนวนตรรกยะส่วนมาก เช่น $1/3$ และอตรรกยะ เช่น $\sqrt{2}$ ไม่สามารถแทนให้ตรงเป๊ะด้วยข้อมูลชนิด `float`

แทนที่จะตรวจสอบว่า `x` และ `y` เท่ากันเป๊ะ มันปลอดภัยกว่าที่จะใช้ ฟังก์ชันพร้อมใช้ `abs` เพื่อคำนวณหาค่าสัมบูรณ์, หรือขนาด, ของความต่างระหว่าง สองค่านี้:

```
    if abs(y-x) < epsilon:
        break
```

โดยที่ `epsilon` มีค่าเช่น `0.0000001` ซึ่งกำหนดว่าใกล้เท่าไร จึงจะเพียงพอ

7.6. อัลกอริธึม

วิธีของนิวตัน (Newton's method) เป็นตัวอย่างของ **อัลกอริธึม (algorithm)**: มันคือกระบวนการทางกลไกสำหรับแก้ปัญหาของหมวดหมู่หนึ่ง ๆ (ในกรณีนี้ การคำนวณรากที่สอง)

เพื่อที่จะเข้าใจว่า อัลกอริธึม คืออะไร มันอาจจะช่วยโดยการเริ่มด้วยบางสิ่งที่ไม่ใช่อัลกอริธึม เมื่อเราเรียนรู้ที่จะคูณเลขหลักเดียว เราอาจจะจำสูตรคูณ มันทำให้เราต้องท่องจำคำตอบ 100 อย่างโดยเฉพาะ ความรู้แบบนี้ไม่ใช่อัลกอริธึม

แต่ถ้าเรา "ขี้เกียจ' เราอาจจะเรียนรู้เทคนิคพิเศษสองสามอย่าง เช่น ในการที่จะหาผลคูณ ของ n และ 9 เราสามารถเขียน $n-1$ เป็นเลขหลักแรก และ $10-n$ เป็นเลขหลักที่สอง เทคนิคนี้เป็นการแก้ปัญหาแบบครอบคลุมสำหรับการคูณเลขหลักเดียวตัวไหนก็ได้กับ 9 นี่คือ อัลกอริธึม!

ในทำนองเดียวกัน เทคนิคที่เราเรียนรู้สำหรับการบวกและลบแบบทดยืม และการหารยาวล้วนเป็นอัลกอริธึม ลักษณะหนึ่งของอัลกอริธึม คือ มันไม่ต้องการความฉลาดในการทำงาน มันเป็นกลไกที่แต่ละขั้นตอนนั้นทำต่อ ๆ กันไปโดยอ้างอิงจากกฎเกณฑ์ง่าย ๆ ชุดหนึ่ง

การทำงานตามอัลกอริธึมมันน่าเบื่อ แต่การออกแบบนั้นน่าสนใจ ประเทืองปัญญา และเป็นศูนย์กลางของวิทยาการคอมพิวเตอร์

บางสิ่งที่ผู้คนทำตามธรรมชาติโดยไม่ยากหรือไม่ต้องคิด เป็นสิ่งที่ยากที่สุดในการเขียนออกมาเป็นอัลกอริธึม การทำความเข้าใจภาษาธรรมชาติ (natural language) เป็นตัวอย่างที่ดี เราทุกคนเข้าใจมัน แต่จนกระทั่งบัดนี้ ไม่มีใครสามารถจะอธิบายได้ว่าเราทำมันได้ *อย่างไร* อย่างน้อยก็ไม่ใช่ในรูปแบบของอัลกอริธึม

7.7. การดีบัก

เมื่อเราเริ่มเขียนโปรแกรมที่ใหญ่ขึ้น เราอาจจะใช้เวลาในการดีบักมากขึ้น โค้ดที่มากขึ้น หมายความว่า โอกาสที่มากขึ้นที่จะเกิดข้อผิดพลาด และมีหลายตำแหน่งมากขึ้นที่บักซ่อนอยู่

ทางหนึ่งที่จะลดเวลาการดีบัก คือ "การดีบักโดยการตัดครึ่งส่วน" เช่น ถ้าในโปรแกรมมี 100 บรรทัด แล้วเราตรวจสอบไปทีละบรรทัด มันก็จะใช้ 100 ขั้นตอน

แทนที่จะทำแบบนั้น ให้แบ่งปัญหาลงครึ่งหนึ่ง ให้ดูแถว ๆ ตรงกลางของโปรแกรม หาค่าระหว่างทางที่เราสามารถตรวจสอบได้ เพิ่มคำสั่ง `print` (หรืออะไรสักอย่างที่สามารถทำให้เราตรวจสอบความถูกต้องได้) และรันโปรแกรม

ถ้าตรงกลางโปรแกรมมันผิด จะต้องมีปัญหาที่ครึ่งแรกของโปรแกรมแน่ ๆ แต่ถ้ามันถูก ปัญหาจะอยู่ที่ครึ่งหลังของโปรแกรม

ทุกครั้งที่เราทำการตรวจสอบแบบนี้ เราจะแบ่งครึ่งจำนวนบรรทัดที่เราต้องค้นหา หลังจากผ่านไป หกขั้นตอนแล้ว (ซึ่งน้อยกว่า 100 ขั้นตอน) เราจะเหลือแค่หนึ่งหรือสองบรรทัดเท่านั้น อย่างน้อยก็ตามทฤษฎี

ในทางปฏิบัติ มันไม่ค่อยชัดเท่าไรว่า “กึ่งกลางโปรแกรม” คืออะไร และอาจจะไม่สามารถตรวจสอบมันได้ มันไม่ค่อยเข้าท่าเท่าไรที่จะนับบรรทัดและหาจุดกึ่งกลางเป๊ะ แทนที่จะทำแบบนั้น ให้คิดว่าตำแหน่งใดในโปรแกรมที่อาจจะมีข้อผิดพลาด และตรงไหนที่มันตรวจสอบได้ง่าย จากนั้นให้เลือกมาจุดหนึ่งซึ่งเราคิดว่ามีโอกาสเท่า ๆ กันที่จะเกิดบักก่อนหน้าและหลังจากจุดนั้น

7.8. อภิธานศัพท์

การกำหนดค่าใหม่ (reassignment): การกำหนดค่าใหม่ให้ตัวแปรที่มีอยู่แล้ว

การปรับค่า (update): คำสั่งที่ทำให้ค่าใหม่ของตัวแปรนั้นขึ้นอยู่กับค่าเก่า

การให้ค่าตั้งต้น (initialization): การกำหนดค่าเริ่มต้นให้ตัวแปรที่จะต้องถูกปรับค่า

การเพิ่มค่า (increment): การปรับค่าที่เพิ่มค่าตัวแปรจากเดิม (ปกติคือเพิ่มขึ้นไป 1)

การลดค่า (decrement): การปรับค่าที่ลดค่าตัวแปรจากเดิม (ปกติคือลดลงมา 1)

การวนซ้ำ (iteration): การทำชุดคำสั่งซ้ำ ๆ โดยใช้การเรียกซ้ำ หรือ ลูป

ลูปไม่รู้จบ (infinite loop): ลูปที่เงื่อนไขการหยุดลูปไม่มีวันเป็นจริง

อัลกอริธึม (algorithm): กระบวนการทั่วไปสำหรับแก้ปัญหาในหมวดหนึ่ง ๆ

7.9. แบบฝึกหัด

***แบบฝึกหัด* 7.1.** *คัดลอกลูปจากหัวข้อที่ 7.5 และหุ้มมันด้วยฟังก์ชันที่ชื่อว่า* `mysqrt` *ซึ่งรับ* `a` *มาเป็นพารามิเตอร์ และคืนค่าประมาณของรากที่สองของ* `a`

เพื่อทดสอบโค้ด ให้เขียนฟังก์ชันชื่อว่า `test_square_root` *ซึ่งพิมพ์ ตารางหน้าตาแบบนี้:*

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
```

```
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

คอลัมน์แรก เป็นเลข, a; คอลัมน์ที่สองเป็นรากที่สองของ a ที่คำนวณจากฟังก์ชัน **`mysqrt`**; *คอลัมน์ที่สาม เป็นรากที่สองที่คำนวณจาก* **`math.sqrt`**; *คอลัมน์ที่สี่เป็นค่าสัมบูรณ์ของผลต่างระหว่างค่าประมาณทั้งสองค่า*

***แบบฝึกหัด* 7.2.** *ฟังก์ชันพร้อมใช้* **`eval`** *รับสายอักขระเข้ามา และประเมินค่าโดยใช้ตัวแปลภาษาไพธอน เช่น:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

ให้เขียนฟังก์ชันที่ชื่อว่า **`eval_loop`** *ที่วนรับค่าจากผู้ใช้ และประเมินค่าที่รับเข้ามาโดยใช้ฟังก์ชัน* **`eval`** *และพิมพ์ผลลัพธ์ออกมา*

มันควรจะทำงานไปเรื่อย ๆ จนกว่าผู้ใช้จะใส่ค่า **`'done'`** *เข้ามา และจากนั้น ให้คืนค่ากลับไปเป็นค่าของนิพจน์สุดท้ายที่ฟังก์ชันนี้ประเมิน*

***แบบฝึกหัด* 7.3.** *นักคณิตศาสตร์ที่ชื่อว่า ศรีนิวาสา รามานุจัน (Srinivasa Ramanujan) พบอนุกรมไม่รู้จบ ซึ่งสามารถใช้สร้างค่าประมาณของ* $1/\pi$*:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

ให้เขียนฟังก์ชันที่ชื่อว่า **`estimate_pi`** *ซึ่งใช้สูตรนี้ในการคำนวณ และคืนค่า ประมาณของ* π *มันควรจะใช้ลูป* **`while`** *ในการคำนวณผลรวมของพจน์ต่าง ๆ จนกว่าพจน์สุดท้ายจะมีค่าน้อยกว่า* **`1e-15`** *(ซึ่ง*

เป็นสัญลักษณ์ที่ไพธอนใช้แทนค่า 10^{-15}*) เราสามารถตรวจสอบผลลัพธ์โดยการเปรียบเทียบค่าที่ได้กับค่า* `math.pi`

เฉลย: `http://thinkpython2.com/code/pi.py`.

8. สายอักขระ

สายอักขระ หรือสตริง (string) ไม่เหมือนข้อมูลชนิดจำนวนเต็ม เลขทศนิยม หรือบูลีน สายอักขระเป็น **ข้อมูลแบบลำดับ (sequence)** ซึ่งหมายความว่ามันเป็นกลุ่มของค่าที่อยู่เรียงต่อกันเป็นลำดับ ในบทนี้ เราจะได้เห็นว่า การเข้าถึง อักขระ (character) ที่ประกอบกันเป็นสายอักขระนั้นทำอย่างไร และเราจะได้เรียนรู้เกี่ยวกับเมธอดบางอันที่ใช้กับสายอักขระ

8.1. สายอักขระเป็นข้อมูลแบบลำดับ

สายอักขระ คือ ลำดับของอักขระ เราสามารถเข้าถึงอักขระทีละตัวได้โดยใช้ตัวดำเนินการวงเล็บสี่เหลี่ยม (bracket):

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

คำสั่งที่สองเลือกอักขระเบอร์ 1 จาก `fruit` และกำหนดให้เป็นค่าของ `letter`

นิพจน์ในวงเล็บสี่เหลี่ยม เรียกว่า **ดัชนี (index)** ดัชนีระบุชี้ว่าอักขระตัวไหนในลำดับคือตัวที่เราต้องการ (เป็นที่มาของชื่อ)

แต่เราอาจจะไม่ได้ค่าที่เราคาดไว้:

```
>>> letter
'a'
```

สำหรับคนส่วนใหญ่แล้ว อักษรตัวแรกของ `'banana'` คือ **b** ไม่ใช่ **a** แต่สำหรับนักวิทยาการคอมพิวเตอร์แล้ว ดัชนีคือออฟเซต (offset) จากตอนต้นของสายอักขระ ออฟเซตของอักษรตัวแรก คือ ศูนย์

```
>>> letter = fruit[0]
>>> letter
'b'
```

ดังนั้น **b** คืออักษรตัวที่ 0 ของคำว่า `'banana'`, **a** คืออักษรตัวที่ 1, และ **n** คืออักษรตัวที่ 2

สำหรับค่าของดัชนี เราสามารถใช้นิพจน์ที่มีตัวแปรและตัวดำเนินการได้:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

แต่ค่าของดัชนีจะต้องเป็นจำนวนเต็ม ไม่เช่นนั้น เราจะได้:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2. ฟังก์ชัน `len`

`len` เป็นฟังก์ชันพร้อมใช้ที่คืนค่าเป็นจำนวนของอักขระในสายอักขระ:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

ในการเข้าถึงอักษรตัวสุดท้ายของสายอักขระ เราอาจจะอยากลองอะไรแบบนี้:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

เหตุผลที่เกิด `IndexError` หรือ ข้อผิดพลาดกับดัชนี คือว่า ใน `'banana'` ไม่มีอักษรตัวที่ดัชนีเท่ากับ 6 เนื่องจากเราเริ่มนับจาก 0 อักษรทั้ง 6 ตัวในคำมีหมายเลขเป็น 0 ถึง 5 ในการที่จะเอาค่าของอักขระตัวสุดท้าย เราจะต้องลบ 1 ออกจากความยาว (`length`) ของสายอักขระ

```
>>> last = fruit[length-1]
>>> last
'a'
```

หรือ เราสามารถใช้ดัชนีค่าติดลบได้ ซึ่งจะนับกลับหลังจากจุดสิ้นสุดของสายอักขระ นิพจน์ `fruit[-1]` ให้ค่าเป็นอักษรตัวสุดท้าย `fruit[-2]` ให้ค่าเป็นอักษรตัวรองสุดท้าย เป็นแบบนี้ไปเรื่อย ๆ

8.3. การท่องสำรวจด้วยลูป for

การคำนวณหลายอย่างเกี่ยวข้องกับการดำเนินการกับสายอักขระทีละอักขระ บ่อยครั้งที่มันจะเริ่มจากตอนต้น เลือกอักขระทีละตัวต่อรอบ ทำอะไรสักอย่างกับมัน แล้วทำต่อไปเรื่อย ๆ จนจบ รูปแบบของการดำเนินการแบบนี้เรียกว่า **การท่องสำรวจ (traversal)** ทางหนึ่งที่จะเขียนการท่องสำรวจนี้ คือการใช้ลูป `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

ลูปนี้ได้แวะผ่านสายอักขระและแสดงอักษรแต่ละตัวบนบรรทัดของใครของมัน เงื่อนไขของลูป คือ `index < len(fruit)` ดังนั้น เมื่อ `index` มีค่าเท่ากับความยาวของสายอักขระ เงื่อนไขจะเป็นเท็จ และส่วนตัวของลูปจะไม่ทำงาน อักขระตัวสุดท้ายที่ถูกเข้าถึงคือตัวที่ดัชนีเป็น `len(fruit)-1` ซึ่งเป็นอักขระตัวสุดท้ายในสายอักขระนี้

เพื่อที่จะฝึกทำ ให้เขียนฟังก์ชันซึ่งรับค่าสายอักขระเป็นอาร์กิวเมนต์ และแสดงค่าของอักษรแบบย้อนหลังบรรทัดละตัว

อีกทางหนึ่งที่จะเขียนการท่องสำรวจ คือ การใช้ลูป `for`:

```
for letter in fruit:
    print(letter)
```

ในแต่ละครั้งที่ผ่านลูป อักขระตัวถัดไปในสายอักขระจะถูกกำหนดเป็นค่าให้กับตัวแปร `letter` ลูปจะทำงาน ไปจนกว่าไม่มีอักขระเหลืออีกแล้ว

ตัวอย่างต่อไปนี้แสดงให้เห็นว่าการเชื่อมต่อสายอักขระนั้นทำอย่างไร (string concatenation, string addition) และการใช้ลูป `for` ในการสร้างซีรีส์เอบีซีดาเรียน (abecedarian series) (นั่นคือ ซีรีส์ของอักษรตามลำดับอักษร) ในหนังสือของโรเบิร์ต แมคคลอสกี้ (Robert McCloskey) ที่ชื่อว่า *Make Way for Ducklings* ชื่อของลูกเป็ดคือ Jack, Kack, Lack, Mack, Nack, Ouack, Pack, และ Quack ลูปต่อไปนี้แสดงชื่อดังกล่าวตามลำดับ:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
```

```
for letter in prefixes:
    print(letter + suffix)
```

เอาต์พุต คือ:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

แน่นอนว่ามันไม่ค่อยถูก เพราะ “Ouack” และ “Quack” นั้นสะกดผิด เพื่อเป็นการหัดทำ ให้แก้โปรแกรมนี้เพื่อแก้ไขข้อผิดพลาด

8.4. ช่วงตัดของสายอักขระ

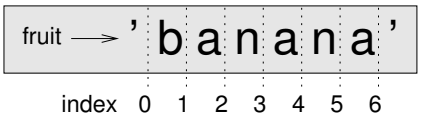
ท่อนของสายอักขระเรียกว่า **ช่วงตัด หรือ สไลซ์ (slice)** การเลือกช่วงตัดเหมือนกับการเลือกอักขระ:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

ตัวดำเนินการ `[n:m]` คืนค่ามาเป็น ส่วนของสายอักขระจากอักขระตัวที่ n ถึง m รวมตัวแรก แต่ไม่รวมตัวหลัง (รวม n ไม่รวม m) พฤติกรรมแบบนี้ไม่ตรงตามสัญชาตญาณ แต่มันอาจจะช่วยถ้าเราลองจินตนาการว่าดัชนีมันชี้อยู่ตรง *ระหว่าง* อักขระแต่ละตัว เหมือนในรูปที่ 8.1

ถ้าเราไม่ใส่ดัชนีตัวแรก (ก่อนเครื่องหมายทวิภาค หรือ โคลอน) ช่วงตัดนี้จะเริ่มที่จุดเริ่มต้นของสายอักขระ ถ้าเราไม่ใส่ดัชนีตัวที่สอง ช่วงตัดนี้จะยาวไปถึงตอนท้ายของสายอักขระ:

```
>>> fruit = 'banana'
```



รูปที่ 8.1.: ดัชนีของช่วงตัด

```
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

ถ้าดัชนีตัวแรกนั้นมากกว่าหรือเท่ากับตัวที่สอง ผลลัพธ์จะเป็น **สายอักขระว่าง (empty string)** แทนด้วยเครื่องหมายอัญประกาศสองตัว:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

สายอักขระว่างไม่มีอักขระอยู่เลย และมีความยาวเป็น 0 แต่นอกจากนี้ มันเป็นเหมือนกับสายอักขระอื่น ๆ

มาทำตัวอย่างนี้ต่อ เราคิดว่า `fruit[:]` หมายความว่าอะไร? ลองดูสิ

8.5. สายอักขระเปลี่ยนแปลงไม่ได้

มันน่าลองที่จะใช้ตัวดำเนินการ `[]` ในทางซ้ายของคำสั่ง โดยตั้งใจให้เปลี่ยนค่าอักขระในสายอักขระ เช่น:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

“object” ในกรณีนี้คือสายอักขระ และ “item” คืออักขระที่เราพยายามจะเปลี่ยนมัน สำหรับตอนนี้ อ็อบเจกต์ (object) ก็เหมือนกับค่า แต่เราจะเกลานิยามของคำนี้ทีหลัง (หัวข้อที่ 10.10)

เหตุผลของข้อผิดพลาดนี้ คือ สายอักขระนั้น **เปลี่ยนแปลงไม่ได้ (immutable)** ซึ่งหมายความว่า เราไม่สามารถเปลี่ยนค่าของสายอักขระที่มีอยู่ สิ่งที่ทำได้ดีที่สุดคือเราสามารถสร้างสายอักขระตัวใหม่ ซึ่งเปลี่ยนไปจากสายอักขระเดิม:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

ตัวอย่างนี้ทำการเชื่อมอักษรตัวแรกตัวใหม่เข้ากับสไลซ์ของ `greeting` มันไม่มีผลกับ สายอักขระตัวเดิม

8.6. การค้นหา

ฟังก์ชันต่อไปนี้ทำอะไร?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

ในมุมหนึ่ง ฟังก์ชัน `find` เป็นส่วนกลับของตัวดำเนินการ `[]` แทนที่จะรับดัชนีแล้วแยกส่วนอักขระออกมา มันกลับรับอักขระเข้ามา แล้วหาดัชนีที่อักขระตัวนั้นอยู่ ถ้าหาอักขระตัวนั้นไม่พบ ฟังก์ชันคืนค่าเป็น `-1`

นี่คือตัวอย่างแรกที่เราเจอคำสั่ง `return` ที่อยู่ในลูป ถ้า `word[index] == letter` ฟังก์ชันจะออกจากลูปและคืนค่ากลับไปทันที

ถ้าอักขระนั้นไม่มีในสายอักขระ โปรแกรมจะออกจากลูปแบบปกติ และคืนค่า `-1` กลับไป

รูปแบบของการคำนวณแบบนี้—การท่องสำรวจข้อมูลตามลำดับ และคืนค่าเมื่อเราพบสิ่งที่เราหาอยู่—เรียกว่า **การค้นหา (search)**

เพื่อเป็นการฝึกทำ ให้แก้ฟังก์ชัน `find` ให้มีพารามิเตอร์ตัวที่สาม คือ ดัชนีใน `word` ตรงที่ให้เริ่มค้นหาค่า

8.7. การทำลูปและการนับ

โปรแกรมต่อไปนี้นับว่าอักษร `a` ปรากฏในสายอักขระกี่ครั้ง:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

โปรแกรมนี้สาธิตอีกรูปแบบหนึ่งของการคำนวณที่เรียกว่า **ตัวนับ (counter)** ตัวแปร `count` มีค่าเริ่มต้นเป็น 0 และจากนั้นเพิ่มทีละหนึ่งในแต่ละครั้งที่เจอ `a` เมื่อออกจากลูป `count` ก็จะมีค่าผลลัพธ์—จำนวนทั้งหมดของ `a`

เพื่อเป็นการฝึกทำ ให้หุ้มโค้ดนี้ในฟังก์ชันที่ชื่อว่า `count` และทำให้มันครอบคลุมถึงการรับสายอักขระและอักษรเข้ามาเป็นอาร์กิวเมนต์

จากนั้นให้เขียนฟังก์ชันนี้ใหม่ โดยแทนที่จะใช้วิธีแวะผ่านสายอักขระ (string traversing) มันจะใช้ฟังก์ชัน `find` เวอร์ชันที่มีพารามิเตอร์สามตัวจากหัวข้อที่แล้ว

8.8. เมธอดของสายอักขระ

สายอักขระเตรียมเมธอดสำหรับทำงานที่มีประโยชน์ได้หลากหลาย เมธอด (method) เหมือนกับฟังก์ชัน — มันรับอาร์กิวเมนต์และคืนค่ากลับมา—แต่มีกฎวากยสัมพันธ์ที่ต่างออกไป เช่น เมธอด `upper` รับค่าสายอักขระเข้ามาและคืนค่าเป็นสายอักขระตัวใหม่ที่เป็นอักษรตัวใหญ่ทั้งหมด

แทนที่จะใช้กฎวากยสัมพันธ์ของฟังก์ชัน คือ `upper(word)` มันจะใช้กฎวากยสัมพันธ์ของเมธอด คือ `word.upper()`

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

รูปแบบของสัญกรจุด (dot notation) นี้ ระบุชื่อของเมธอด, `upper`, และชื่อของสายอักขระ ที่จะใช้เมธอดนั้น, `word` วงเล็บว่างระบุว่าเมธอดนี้ไม่รับอาร์กิวเมนต์เข้ามา

การเรียกเมธอดนั้นเรียกว่า **การเรียกใช้ (invocation)**; ในกรณีนี้ เราพูดได้ว่า เราเรียกใช้ให้ทำ `upper` กับ `word`

กลายเป็นว่า มีเมธอดของสายอักขระที่ชื่อว่า `find` ที่เหมือนกันอย่างประหลาดกับฟังก์ชันที่เราได้เขียนไป:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

ในตัวอย่างนี้ เราเรียกใช้ให้ `find` กับ `word` และผ่านอักษรที่เราต้องการหาเข้า ไปเป็นพารามิเตอร์

ที่จริงแล้ว เมธอด `find` มันครอบคลุมมากกว่าฟังก์ชันของเรา; มันสามารถหาสายอักขระย่อย (substring) ไม่ใช่แค่อักขระ (character):

```
>>> word.find('na')
2
```

โดยปริยาย `find` จะเริ่มที่จุดเริ่มต้นของสายอักขระ แต่มันสามารถรับอาร์กิวเมนต์ตัวที่สอง คือ ดัชนีที่มันควรจะเริ่มต้นค้นหา:

```
>>> word.find('na', 3)
4
```

นี่คือตัวอย่างของ **อาร์กิวเมนต์ทางเลือก (optional argument)**; `find` สามารถรับอาร์กิวเมนต์ตัวที่สามได้ด้วย คือ ดัชนีที่มันควรจะหยุดหา:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

การค้นหานี้ไม่สำเร็จ เพราะ `b` ไม่ปรากฏอยู่ในช่วงดัชนี `1` ถึง `2` โดยไม่รวม `2` การค้นหาไปจนถึง, แต่ไม่รวม, ดัชนีตัวที่สองนี้ ทำให้ `find` ทำงานสอดคล้องกันกับตัวดำเนินการตัดช่วงสายอักขระ (slice)

8.9. ตัวดำเนินการ `in`

คำว่า `in` เป็นตัวดำเนินการบูลีน ซึ่งรับสายอักขระสองตัว และคืนค่าเป็น `True` ถ้าตัวแรกเป็นสายอักขระย่อย (substring) ของตัวที่สอง:

```
>>> 'a' in 'banana'
```

```
True
>>> 'seed' in 'banana'
False
```

ตัวอย่างเช่น ฟังก์ชันต่อไปนี้พิมพ์อักษรทุกตัวจาก word1 ที่อยู่ใน word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

ถ้าเราตั้งชื่อตัวแปรดี ๆ บางครั้งไพธอนจะอ่านเหมือนเป็นภาษาอังกฤษปกติเลย เราอาจจะอ่านลูปนี้ว่า “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.” แปลว่า สำหรับ letter (อักษรแต่ละตัว) ใน word (คำแรก), ถ้า letter (อักษรตัวนั้น) (ปรากฏอยู่) ใน word (อักษรตัวที่สอง) ให้พิมพ์ letter (อักษรตัวนั้น) ออกมา

นี่คือสิ่งที่เราจะได้ถ้าเราเปรียบเทียบคำว่า apples กับ oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10. การเปรียบเทียบสายอักขระ

ตัวดำเนินการเชิงสัมพันธ์ทำงานได้กับสายอักขระ เพื่อที่จะหาว่าสายอักขระสองตัวเท่ากันหรือเปล่า:

```
if word == 'banana':
    print('All right, bananas.')
```

การดำเนินการเชิงสัมพันธ์แบบอื่นเป็นประโยชน์สำหรับเรียงคำตามลำดับตัวอักษร:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

ไพธอนไม่จัดการอักษรตัวใหญ่และตัวเล็กเหมือนกับที่มนุษย์ทำ อักษรตัวใหญ่ทั้งหมดจะอยู่ก่อนอักษรตัวเล็กทั้งหมด ดังนั้น:

```
Your word, Pineapple, comes before banana.
```

วิธีที่นิยมใช้กันเพื่อที่จะแก้ปัญหานี้ คือการแปลงสายอักขระให้เป็นรูปแบบมาตรฐาน, เช่น แปลง ให้เป็นตัวเล็กทั้งหมด, ก่อนที่จะเปรียบเทียบ ให้จำอันนี้เอาไว้ในกรณีที่เราจะต้องป้องกันตัวเองจากคนที่ติดอาวุธเป็นสับปะรด (Pineapple)

8.11. การดีบัก

เมื่อเราใช้ดัชนีในการท่องสำรวจค่าต่าง ๆ ในข้อมูลแบบลำดับ มันยุ่งยากหน่อยที่จะ เริ่มและจบแบบถูกต้อง นี่คือฟังก์ชันที่ควรจะเปรียบเทียบคำสองคำ และคืนค่า `True` ถ้าคำหนึ่งเป็นส่วนกลับของอีกคำหนึ่ง แต่มันมีข้อผิดพลาดสองจุด:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

คำสั่ง `if` อันแรกตรวจสอบว่าคำที่รับเข้ามามีความยาวเท่ากันหรือไม่ ถ้าไม่ เราสามารถคืนค่า `False` ได้เลย ไม่เช่นนั้น ในการทำงานที่เหลือของฟังก์ชันเราสามารถเชื่อได้ว่าคำสองคำมีความยาวเท่ากัน นี่คือตัวอย่างของรูปแบบผู้พิทักษ์ (guardian pattern) ที่กล่าวถึงในหัวข้อที่ 6.8

ตัวแปร `i` และ `j` คือดัชนี: `i` แวะผ่าน `word1` จากหน้าไปหลัง ในขณะที่ `j` แวะผ่าน `word2` จากหลังมาหน้า ถ้าเราเจอ อักษรสองตัวที่ไม่เหมือนกัน เราสามารถคืนค่า `False` กลับไปได้ทันที ถ้าเราทำลูปจนครบรอบแล้วและทุกอักษรเหมือนกัน เราจะคืนค่าเป็น `True`

ถ้าเราทดสอบฟังก์ชันนี้ด้วยคำว่า “pots” และ “stop” เราคาดว่าจะได้ค่าที่คืนกลับไปเป็น `True` แต่เรากลับได้ข้อผิดพลาดทางดัชนี (IndexError):

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

สำหรับการดีบักข้อผิดพลาดประเภทนี้ สิ่งที่ผมทำอย่างแรก คือ พิมพ์ค่าของดัชนี ในบรรทัดติดกันที่อยู่ก่อนบรรทัดที่เกิดข้อผิดพลาด

```
    while j > 0:
        print(i, j)        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

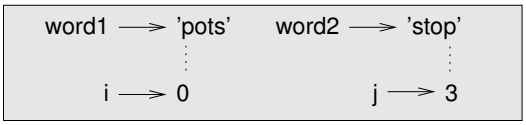
ตอนนี้ เมื่อผมรันโปรแกรมอีกครั้ง ผมได้ข้อมูลเพิ่มเติม:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

ครั้งแรกที่เข้าไปในลูป ค่าของ `j` เป็น 4 ซึ่งมันเกินช่วง (out of range) สำหรับสายอักขระ `'pots'` ดัชนีของอักขระตัวสุดท้ายคือ 3 ดังนั้น ค่าเริ่มต้นของ `j` ควรจะเป็น `len(word2)-1`

ถ้าผมแก้ไขข้อผิดพลาดนั้นแล้วรันโปรแกรมอีกที ผมก็จะได้:

```
>>> is_reverse('pots', 'stop')
0 3
```



รูปที่ 8.2.: แผนภาพสถานะ

```
1 2
2 1
True
```

ครั้งนี้เราได้คำตอบที่ถูกต้อง แต่เหมือนกับว่าลูปทำงานแค่ 3 ครั้ง ซึ่งมันน่าสงสัย เพื่อที่จะได้ มุมมองที่ดีขึ้นว่าเกิดอะไรขึ้น มันเป็นประโยชน์ที่จะวาดแผนภาพสถานะ ในการวนซ้ำรอบแรก กรอบของ `is_reverse` ถูกแสดงในรูปที่ 8.2

ผมขออนุญาตจัดเรียงตัวแปรในเฟรมใหม่ และเพิ่มเส้นประเพื่อจะแสดงว่าค่าของ `i` และ `j` ระบุอักขระใน `word1` และ `word2`

เริ่มด้วยแผนภาพอันนี้ รันโปรแกรมบนกระดาษ แล้วเปลี่ยนค่าของ `i` และ `j` ในแต่ละรอบของการวนซ้ำ ให้หาข้อผิดพลาดข้อที่สองในฟังก์ชันและแก้มันซะ

8.12. อภิธานศัพท์

อ็อบเจกต์ (object): บางสิ่งที่ตัวแปรสามารถอ้างอิงถึงได้ สำหรับตอนนี้ เราสามารถใช้ “อ็อบเจกต์ (object)” and “ค่า (value)” สลับกันไปมาได้

ข้อมูลแบบลำดับ (sequence): กลุ่มของค่าที่เรียงไว้ โดยแต่ละค่าถูกระบุได้ด้วยดัชนีที่เป็นจำนวนเต็ม

รายการ (item): หนึ่งในค่าที่อยู่ในลำดับ

ดัชนี (index): จำนวนเต็มที่ใช้เลือกรายการในลำดับ เช่น อักขระในสายอักขระ ในไพธอน ดัชนีเริ่มต้นที่ 0

ช่วงตัด (slice): ส่วนของสายอักขระที่ถูกระบุโดยช่วงของดัชนี

สายอักขระว่าง (empty string): สายอักขระที่ไม่มีอักขระอยู่และมีความยาวเป็น 0 ซึ่งเขียนแทนค่าด้วยอัญประกาศสองตัว

เปลี่ยนแปลงไม่ได้ (immutable): คุณสมบัติของลำดับที่ว่ารายการของลำดับนั้นไม่สามารถเปลี่ยนค่าได้

การท่องสำรวจ (traverse): การวนซ้ำไปในรายการต่าง ๆ ในลำดับ เพื่อดำเนินการอย่างเดียวกันในแต่ละรอบ

การค้นหา (search): รูปแบบของการท่องสำรวจที่หยุดทำงานเมื่อมันเจอสิ่งที่มันค้นหาแล้ว

ตัวนับ (counter): ตัวแปรที่ใช้นับอะไรสักอย่าง โดยปกติแล้วจะมีค่าเริ่มต้นเป็น 0 และถูกเพิ่มค่าขึ้นเรื่อย ๆ

การเรียกใช้ (invocation): คำสั่งที่ใช้เรียกเมธอด

อาร์กิวเมนต์ทางเลือก (optional argument): อาร์กิวเมนต์ของฟังก์ชันหรือเมธอดที่ไม่จำเป็นจะต้องใส่ค่าให้

8.13. แบบฝึกหัด

***แบบฝึกหัด* 8.1.** *อ่านเอกสารของเมธอดของสายอักขระที่ `http://docs.python.org/3/library/stdtypes.html#string-methods` เราอาจจะอยากทดลองกับบางเมธอดเพื่อให้แน่ใจว่าเราเข้าใจว่ามันทำงานยังไง เมธอด `strip` และ `replace` เป็นเมธอดที่มีประโยชน์เป็นพิเศษ*

เอกสารข้างต้นใช้กฎวากสัมพันธ์ที่อาจจะสับสนหน่อย เช่น ใน `find(sub[, start[, end]])` วงเล็บทำการระบุอาร์กิวเมนต์ทางเลือก (optional argument) ดังนั้น `sub` เป็นสิ่งจำเป็น แต่ `start` เป็นทางเลือก และถ้าเราใส่ `start` เข้าไปแล้ว `end` ก็เป็นทางเลือก

***แบบฝึกหัด* 8.2.** *มีเมธอดของสายอักขระที่ชื่อว่า `count` ที่ทำงานเหมือนกับฟังก์ชันในหัวข้อที่ 8.7 ให้อ่านเอกสารของเมธอดนี้และเขียนการเรียกใช้ (invocation) ที่นับจำนวนของ `a` ในคำว่า `'banana'`*

***แบบฝึกหัด* 8.3.** *ช่วงตัดของสายอักขระสามารถรับค่าดัชนีตัวที่สามซึ่งระบุ "ขนาดของขั้น (step size)" ได้; นั่นคือ จำนวนของที่ว่างระหว่างอักขระที่ติดกัน ขนาดของขั้นเท่ากับ 2 หมายความว่าอักขระเว้นอักขระ; ส่วน 3 หมายความว่าเอาอักขระมาทุก ๆ 3 ตัว, และอื่น ๆ*

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

ขนาดของขั้นเท่ากับ -1 จะทำงานกับคำนั้นแบบกลับหลัง ดังนั้น การตัดช่วง `[::-1]` *ทำให้เกิดสายอักขระแบบกลับหลัง*

ให้ใช้ลักษณะเฉพาะอันนี้เขียนฟังก์ชัน `is_palindrome` *จากแบบฝึกหัดที่ 6.3 ให้เป็นเวอร์ชันที่มี 1 บรรทัด*

แบบฝึกหัด 8.4. *ฟังก์ชันต่อไปนี้ทั้งหมด ตั้งใจ ที่จะตรวจสอบว่าสายอักขระมีอักษรตัวเล็กหรือไม่ แต่ อย่างน้อยบางฟังก์ชันก็ผิด ในแต่ละฟังก์ชัน ให้บรรยายว่าฟังก์ชันนั้นทำงานอะไรกันแน่ (สมมติว่าพารามิเตอร์ของแต่ละฟังก์ชันเป็นสายอักขระ)*

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag
```

```
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

แบบฝึกหัด 8.5. *รหัสซีซาร์ (Caesar cypher) เป็นรูปแบบการเข้ารหัสที่อ่อนแออย่างหนึ่ง ซึ่งเกี่ยวข้องกับการ "หมุน" อักษรแต่ละตัวไปเป็นจำนวนตำแหน่งที่ตายตัว การหมุนอักษรหมายความว่าการเลื่อน ไปในรายการพยัญชนะ แล้ววนกลับมาพยัญชนะเริ่มต้นหากจำเป็น ดังนั้น การหมุน 'A' ไป 3 ตำแหน่งจะได้ 'D' และการหมุน 'Z' ไป 1 ตำแหน่งจะได้ 'A'*

การหมุนคำ เราหมุนอักษรแต่ละตัวในคำในจำนวนที่เท่ากัน เช่น การหมุน "cheer" ไป 7 ตำแหน่งจะได้ "jolly" และการหมุน "melon" ไป -10 ตำแหน่งจะได้ "cubed" ในหนังสือ 2001: A Space Odyssey *คอมพิวเตอร์ของยานชื่อว่า HAL ซึ่งเป็นการหมุน คำว่า IMB ไป -1 ตำแหน่ง*

ตัวอย่างเช่น คำว่า "sleep" หมุนไป 9 ตำแหน่ง คือคำว่า "bunny" และ คำว่า "latex" หมุนไป 7 ตำแหน่ง คือคำว่า "shale"

ให้เขียนฟังก์ชันที่ชื่อว่า `rotate_word` *ซึ่งรับสายอักขระและจำนวนเต็มเป็นพารามิเตอร์ และคืนค่าเป็นสายอักขระตัวใหม่ที่มีอักษรจากสายอักขระเดิมที่ถูกหมุนเป็นจำนวนที่กำหนดให้*

เราอาจจะต้องการใช้ฟังก์ชันพร้อมใช้ `ord` *ซึ่งแปลงอักขระเป็นรหัสตัวเลข และ ฟังก์ชัน* `chr` *ซึ่งแปลงรหัสตัวเลขเป็นอักขระ อักษรต่าง ๆ ถูกเข้ารหัสโดยเรียงตามตัวอักษร ดังนั้น เช่น:*

```
>>> ord('c') - ord('a')
2
```

เพราะว่า `'c'` *คือ อักษรลำดับที่ 2 ของพยัญชนะ แต่ระวัง: รหัสตัวเลขสำหรับอักษรตัวใหญ่นั้นแตกต่างออกไป*

(หมายเหตุผู้แปล: อักษรลำดับที่ 0 คือ `'a'`*)*

ตลกที่อาจจะไม่เหมาะสมบนอินเทอร์เน็ตบางครั้งจะถูกเข้ารหัสด้วย ROT13 ซึ่งคือรหัสซีซาร์ ที่หมุนไป 13 ตัว ถ้าเราไม่ได้เป็นคนที่ขุ่นเคืองง่าย ลองหาและถอดรหัสบางอันดู

เฉลย: `http://thinkpython2.com/code/rotate.py`.

9. กรณีศึกษา เกมทายคำ

บทนี้นำเสนอกรณีศึกษากรณีที่สอง ซึ่งเกี่ยวกับปริศนาทายคำ โดยการค้นหาคำที่มี คุณสมบัติตามที่กำหนด เช่น เราจะหาพาลินโดรม (palindrome) ที่ยาวที่สุด ในภาษาอังกฤษ และหาคำที่มีอักษรในคำเรียงตามลำดับตัวอักษร และผมจะนำเสนอแผนการพัฒนาโปรแกรมอีกแบบหนึ่ง: การลดทอนให้เป็นปัญหาที่ถูกแก้ไปแล้ว

9.1. การอ่านรายการของคำ

สำหรับแบบฝึกหัดในบทนี้ เราต้องการรายการ หรือ ลิสต์ (list) ของคำภาษาอังกฤษ มันมีหลาย ชุด ให้ใช้บนอินเทอร์เน็ต แต่อันที่เหมาะกับจุดประสงค์ของเราคือรายการคำที่จัดเก็บและเผยแพร่สู่ สาธารณะ โดย แกรดี้ วอร์ด (Grady Ward) ซึ่งเป็นส่วนหนึ่งของโครงการโมบี้เลซิคอน (Moby lexicon project) (ดูเพิ่มที่ `http://wikipedia.org/wiki/Moby_Project`) มันเป็นรายการที่มี 113,809 คำที่ใช้เล่นเกมครอสเวิร์ด (crossword) อย่างเป็นทางการ; นั่นคือ คำที่ให้ใช้ได้ในครอสเวิร์ดและ เกมเกี่ยวกับคำศัพท์อื่น ๆ ในรายการคำโมบี้นี้มีชื่อไฟล์ว่า `113809of.fic`; เราสามารถดาวน์โหลด สำเนาของไฟล์โดยใช้ชื่อที่ง่ายกว่าว่า `words.txt` จาก `http://thinkpython2.com/code/words.txt`

ไฟล์นี้ถูกเก็บอยู่ในรูปแบบตัวข้อความธรรมดา ดังนั้นเราสามารถเปิดมันโดยใช้โปรแกรมบรรณาธิการข้อความ (text editor) แต่เราก็สามารถอ่านมันจากไพธอนได้ด้วย ฟังก์ชันพร้อมใช้ **open** รับชื่อของไฟล์เข้ามาเป็นพารามิเตอร์ และคืนค่าเป็น **อ็อบเจกต์ไฟล์ (file object)** ที่เราสามารถใช้ในการอ่านไฟล์ได้

```
>>> fin = open('words.txt')
```

`fin` เป็นชื่อที่ใช้โดยทั่วไปสำหรับอ็อบเจกต์ไฟล์ที่ใช้สำหรับอินพุตหรือนำเข้าข้อมูล อ็อบเจกต์ไฟล์มีหลายเมธอดสำหรับการอ่าน รวมถึงเมธอด `readline` ซึ่งอ่านอักขระต่าง ๆ จากไฟล์จนกว่าจะเจอบรรทัดใหม่ (newline) และคืนค่าเป็นสายอักขระ:

```
>>> fin.readline()
'aa\n'
```

คำแรกในลิสต์นี้ คือ “aa” ซึ่งแปลว่าลาวาชนิดหนึ่ง ลำดับอักขระ `\n` แทนอักขระ เริ่มบรรทัดใหม่ (newline character) ซึ่งแบ่งคำนี้จากคำถัดไป

อ็อบเจกต์ไฟล์จดจำว่ามันอ่านถึงไหนแล้วในไฟล์นั้น ดังนั้น ถ้าเราเรียก `readline` อีกครั้งหนึ่ง เราจะได้คำถัดไป:

```
>>> fin.readline()
'aah\n'
```

คำถัดไป คือคำว่า “aah” ซึ่งเป็นคำที่แท้จริง ดังนั้น หยุดมองผมแบบนั้นซะที หรือ ถ้ามันเป็นอักขระเริ่มบรรทัดใหม่ที่ทำให้ขัดใจ เราสามารถกำจัดมันด้วยเมธอด ของสายอักขระที่ชื่อว่า `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

เราสามารถใช้อ็อบเจกต์ไฟล์เป็นส่วนหนึ่งของลูป `for` ได้ โปรแกรมนี้อ่านไฟล์ `words.txt` และพิมพ์แต่ละคำออกมาคำละบรรทัด:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2. แบบฝึกหัด

เฉลยสำหรับแบบฝึกหัดต่อไปนี้จะอยู่ในหัวข้อถัดไป อย่างน้อยเราควรพยายามที่จะทำแต่ละข้อ ก่อนที่จะไปดูเฉลย

***แบบฝึกหัด* 9.1.** *ให้เขียนโปรแกรมซึ่งอ่านไฟล์ `words.txt` และพิมพ์คำที่มีความยาวมากกว่า 20 อักขระ (ไม่นับเว้นวรรค)*

***แบบฝึกหัด* 9.2.** *ในปี ค.ศ. 1939 เอิร์นเนิสต์ วินเซนต์ ไรต์ (Ernest Vincent Wright) ได้ตีพิมพ์ นวนิยายความยาว 50,000 คำที่ชื่อว่า* แกดสบี้ (Gadsby) *ซึ่งไม่มีอักษร “e” อยู่เลย เนื่องจาก “e” เป็นอักษรที่นิยมใช้เป็นปกติในภาษาอังกฤษ มันจึงไม่ง่ายเลยที่จะทำแบบนั้น*

ที่จริงแล้ว มันยากที่จะสร้างความคิดโดดเดี่ยวโดยปราศจากสัญลักษณ์ที่นิยมใช้อันนี้ การทำเป็นไป อย่างเชื่องช้าในตอนแรก แต่ด้วยความรอบคอบและการฝึกฝนอันยาวนาน เราสามารถค่อย ๆ เพิ่มความง่ายขึ้น

เอาเถอะ ผมจะหยุดตรงนี้

ให้เขียนฟังก์ชันที่ชื่อว่า **`has_no_e`** *ที่คืนค่า* **`True`** *ถ้าคำที่กำหนดให้ไม่มีอักษร "e" ในคำนั้น*

ให้แก้ไขโปรแกรมจากหัวข้อที่แล้วให้พิมพ์แค่คำที่ไม่มี "e" และคำนวณเปอร์เซนต์ของคำในรายการที่ไม่มี "e"

แบบฝึกหัด **9.3.** *ให้เขียนฟังก์ชันที่ชื่อว่า* **`avoids`** *ซึ่งรับคำหนึ่งคำ และสายอักขระของอักษรต้องห้าม และคืนค่า* **`True`** *ถ้าคำนั้นไม่มีอักษรต้องห้ามที่กำหนดให้เลย*

ให้แก้ไขโปรแกรมเพื่อรับสายอักขระของอักษรต้องห้ามจากผู้ใช้ และจากนั้นพิมพ์จำนวนคำที่ไม่มีอักษรต้องห้ามใด ๆ เลย

เราสามารถหาการรวมกันของอักษรต้องห้าม 5 ตัว ที่แยกคำออกไปเป็นจำนวนน้อยที่สุดออกไปได้หรือไม่?

แบบฝึกหัด **9.4.** *ให้เขียนฟังก์ชันที่ชื่อว่า* **`uses_only`** *ซึ่งรับคำหนึ่งคำ และสายอักขระของอักษร และ คืนค่าเป็น* **`True`** *ถ้าคำนั้นมีแค่อักษรในลิสต์ที่กำหนดให้ เราสามารถสร้างประโยคโดยใช้แค่อักษรใน* **`acefhlo`** *ได้หรือไม่? แล้วที่ไม่ใช่คำว่า "Hoe alfalfa" ล่ะ?*

แบบฝึกหัด **9.5.** *ให้เขียนฟังก์ชันที่ชื่อว่า* **`uses_all`** *ที่รับคำหนึ่งคำ และสายอักขระของอักษรบังคับ (ที่ต้องใช้) และให้คืนค่าเป็น* **`True`** *ถ้าคำนั้นใช้อักษรบังคับอย่างน้อยตัวละหนึ่งครั้ง มีคำกี่คำที่ใช้ สระทั้งหมดใน* **`aeiou`***? แล้วใน* **`aeiouy`** *ล่ะ?*

แบบฝึกหัด **9.6.** *ให้เขียนฟังก์ชันชื่อว่า* **`is_abecedarian`** *ซึ่งคืนค่าเป็น* **`True`** *ถ้าอักษรในคำ นั้นปรากฏตามลำดับตัวอักษร (ใช้อักษรคู่ได้) มีคำที่เป็น เอบีซีดาเรียน (abecedarian) ทั้งหมดกี่คำ?*

9.3. การค้นหา

แบบฝึกหัดทั้งหมดในหัวข้อที่แล้วมีบางอย่างที่เหมือนกัน: มันสามารถแก้ได้โดยใช้รูปแบบการค้นหาที่เราเจอในหัวข้อที่ 8.6 ตัวอย่างที่ง่ายที่สุดคือ:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

ลูป for แวะผ่านอักขระใน word ถ้าเราเจออักษร “e” เราสามารถคืนค่า False; ได้ทันที ไม่เช่นนั้น เราต้องไปที่อักษรตัวถัดไป ถ้าเราออกจากลูปแบบปกติแล้ว นั่นหมายความว่า เราไม่เจอ “e” ดังนั้น เราสามารถคืนค่า True ได้

เราอาจจะเขียนฟังก์ชันนี้ให้กระชับขึ้นโดยการใช้ตัวดำเนินการ in แต่ผมเริ่มจากเวอร์ชันนี้เพราะมันสาธิตตรรกะของรูปแบบการค้นหา

ฟังก์ชัน avoids มันครอบคลุมมากกว่าเวอร์ชันของ has_no_e แต่มัน มีโครงสร้างที่เหมือนกัน:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

เราสามารถคืนค่า False ทันทีที่เราเจออักษรต้องห้าม; ถ้าเราไปถึงการจบลูปแล้ว เราก็คืนค่า True กลับไป

ฟังก์ชัน uses_only ก็เหมือนกัน ยกเว้นว่าความหมายของเงื่อนไขก็จะกลับกัน:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

แทนที่จะใช้รายการของอักษรต้องห้าม เรามีรายการของอักษรที่ใช้ได้ ถ้าเราเจออักษรใน word ที่ไม่ได้อยู่ใน available เราสามารถคืนค่า False กลับไปได้

ฟังก์ชัน uses_all ก็เหมือนกัน ยกเว้นว่าเรากลับหน้าที่ของคำและสายอักขระของอักษร:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

แทนที่จะแวะผ่านอักษรต่าง ๆ ใน word ลูปนี้แวะผ่านอักษรบังคับเท่านั้น ถ้าอักษรบังคับตัวไหนไม่ปรากฏในคำ เราสามารถคืนค่าเป็น False ได้

ถ้าเราคิดจริงจังเหมือนกับนักวิทยาการคอมพิวเตอร์ เราจะรู้จำได้ว่า `uses_all` เป็นกรณีตัวอย่างของปัญหาที่ถูกแก้ไปแล้ว และเราก็จะเขียนว่า:

```
def uses_all(word, required):
    return uses_only(required, word)
```

นี่คือตัวอย่างของแผนการพัฒนาโปรแกรมที่เรียกว่า **การลดทอนให้เป็นปัญหาที่ถูกแก้ไปแล้ว** ซึ่งหมายความว่าเรารู้จำว่าปัญหาที่เรากำลังแก้อยู่นี้เป็นกรณีตัวอย่างของปัญหาที่เคยถูกแก้ไปแล้ว แล้วเราก็ประยุกต์ใช้ทางแก้ปัญหาที่มันมีอยู่แล้ว

9.4. ลูปด้วยดัชนี

ผมเขียนฟังก์ชันในหัวข้อที่แล้วด้วยลูป `for` เพราะผมแค่ต้องการอักขระในสายอักขระ; ผมไม่ได้ต้องการที่จะทำอะไรกับดัชนี

สำหรับฟังก์ชัน `is_abecedarian` เราต้องเปรียบเทียบอักษรที่อยู่ติดกัน ซึ่งจะยุ่งยากหน่อย เมื่อใช้ลูป `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

ทางเลือกหนึ่งคือการใช้การย้อนเรียกใช้ (recursion):

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

อีกทางหนึ่งคือการใช้ลูป `while`:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

ลูปเริ่มที่ `i=0` และจบเมื่อ `i=len(word)-1` ในแต่ละครั้งที่รันลูป มันจะเปรียบเทียบ อักขระตัวที่ *i*th (ซึ่งเราสามารถคิดว่ามันเป็นอักขระตัวปัจจุบันได้) กับอักขระตัวที่ $i+1$th (ซึ่งเราสามารถคิดว่ามันเป็นอักขระตัวถัดไป)

ถ้าอักขระตัวถัดไปน้อยกว่า (มาก่อนตามลำดับตัวอักษร) ตัวปัจจุบันแล้ว เราค้นพบตัวทำลายแนวโน้มการเป็น abecedarian และเราจะคืนค่าเป็น `False`

ถ้าเราไปถึงตอนท้ายของลูปโดยไม่เจออะไรผิดพลาดแล้ว คำนั้นก็ผ่านการทดสอบ ในการโน้มน้าว ตัวเราเองว่าลูปนั้นจบแบบถูกต้อง ให้พิจารณาตัวอย่างเช่นคำว่า `'flossy'` ความยาวของคำคือ 5 ดังนั้น ครั้งสุดท้ายที่ลูปทำงานคือเมื่อ `i` is 4 ซึ่งเป็นดัชนีของอักขระตัวรองสุดท้าย ในการวนซ้ำรอบสุดท้าย มันเปรียบเทียบอักขระตัวรองสุดท้ายนี้กับอักขระตัวสุดท้าย ซึ่งเป็นสิ่งที่เราต้องการ

นี่คือเวอร์ชันของฟังก์ชัน `is_palindrome` (ไปดูแบบฝึกหัดที่ 6.3) ซึ่ง ใช้ดัชนีสองตัว; ตัวแรกเริ่มที่จุดเริ่มต้นและนับขึ้นไปในคำ; อีกตัวเริ่มที่ตอนจบแล้วนับถอยหลังลงมา

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

หรือเราสามารถจะลดทอนให้เป็นปัญหาที่ถูกแก้ไปแล้ว และเขียนว่า:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

โดยการใช้ฟังก์ชัน `is_reverse` จากหัวข้อที่ 8.11

9.5. การดีบัก

การทดสอบโปรแกรมนั้นยาก ฟังก์ชันต่าง ๆ ในบทนี้มันค่อนข้างง่ายที่จะทดสอบ เพราะว่าเราสามารถตรวจสอบผลลัพธ์โดยมือได้ ถึงกระนั้น มันก็ยังถือว่าอยู่ระหว่างความยากและเป็นไปไม่ได้ที่จะเลือกชุดของคำเพื่อทดสอบหาข้อผิดพลาดที่สามารถเกิดขึ้นได้ทั้งหมด

เอาฟังก์ชัน `has_no_e` เป็นตัวอย่าง มีสองกรณีที่ชัดเจนที่จะตรวจสอบ: คำที่มี 'e' จะต้อง คืนค่ากลับมาเป็น `False` และคำที่ไม่มีควรจะคืนค่ากลับมาเป็น `True` เราควรจะไม่มีปัญหาในการคิดคำหนึ่งคำในแต่ละกรณี

ในแต่ละกรณี มีกรณีย่อยบางอันที่ไม่ค่อยชัดเจนเท่า เราควรจะทดสอบด้วยคำที่มี "e" ตอนขึ้นต้น, ลงท้าย และตรงกลาง ๆ คำ เราควรทดสอบด้วยคำยาว คำสั้น และคำสั้นมาก ๆ เช่น สายอักขระว่าง สายอักขระว่างเป็นตัวอย่างของ **กรณีพิเศษ** ซึ่งเป็นหนึ่งในกรณีที่ไม่ค่อยชัดเจนที่มีข้อผิดพลาดเกิดขึ้นบ่อย ๆ

นอกจากการทดสอบกรณีที่เราเป็นคนสร้างขึ้นมาแล้ว เราสามารถทดสอบโปรแกรมของเรา ด้วยรายการคำศัพท์ เช่น `words.txt` อีกด้วย ในการตรวจเอาต์พุต เราอาจจะสามารถจับข้อผิดพลาดได้ แต่ให้ระวัง: เราอาจจะจับข้อผิดพลาดประเภทหนึ่ง (คำที่ไม่ควรจะ มีในเอาต์พุต แต่ดันมี) และ ไม่สามารถจับอีกประเภทหนึ่งได้ (คำที่ควรจะมีในเอาต์พุต แต่ดันไม่มี)

โดยทั่วไปแล้ว การทดสอบสามารถช่วยเราหาบัก (bug) ได้ แต่มันไม่ง่ายที่จะสร้างชุดทดสอบที่ดี และแม้กระทั่งถึงเราสร้างได้ เราอาจจะไม่สามารถแน่ใจได้ว่าโปรแกรมเราถูกต้องอย่างหมดจดหรือไม่ ตามที่นักวิทยาการคอมพิวเตอร์ในตำนานได้กล่าวไว้:

> การทดสอบโปรแกรมสามารถใช้เพื่อแสดงว่ามีบัก แต่มันไม่สามารถแสดงได้ว่าไม่มีบัก!
>
> — เอ็ดสเกอร์ ไดก์สตรา (Edsger W. Dijkstra)

9.6. อภิธานศัพท์

อ็อบเจกต์ไฟล์ (file object): ค่าที่ใช้แทนไฟล์ที่ถูกเปิดใช้งาน

ลดทอนให้เป็นปัญหาที่ถูกแก้ไปแล้ว (reduction to a previously solved problem): วิธีการแก้ปัญหาโดยการทำให้อยู่ในรูปแบบของปัญหาที่เคยถูกแก้ไปแล้ว

กรณีพิเศษ (special case): กรณีทดสอบที่ไม่ปกติหรือไม่ชัดแจ้ง(และไม่น่าจะถูกจัดการได้อย่างถูกต้อง)

9.7. แบบฝึกหัด

***แบบฝึกหัด* 9.7.** *คำถามนี้มาจากปริศนาในวิทยุรายการ* Car Talk (`http://www.cartalk.com/content/puzzlers`)*:*

> *จงหาคำมาหนึ่งคำที่มีอักษรคู่สามคู่ติดกัน ผมจะให้สองสามคำที่เกือบจะเป็น แต่ไม่ได้เป็น เช่น คำว่า committee, c-o-m-m-i-t-t-e-e มันจะเยี่ยมเลยยกเว้นดันมี 'i' ที่แอบเข้าไปอยู่ในคำ หรือคำว่า Mississippi: M-i-s-s-i-s-s-i-p-p-i ถ้าเราสามารถเอา i ออกมาได้ ก็จะใช้ได้ แต่มีอีกคำที่มีอักษรคู่สามคู่ติดกัน และเท่าที่ผมรู้ มันอาจจะเป็นคำเดียวด้วยซ้ำ แน่นอนว่ามันอาจจะมี 500 คำอื่น แต่ผมสามารถคิดได้แค่คำเดียว คำนั้นคือคำว่าอะไร?*

จงเขียนโปรแกรมเพื่อหาคำนั้น เฉลย: `http://thinkpython2.com/code/cartalk1.py`*.*

***แบบฝึกหัด* 9.8.** *นี่คือปริศนาอีกข้อจากรายการ* Car Talk (`http://www.cartalk.com/content/puzzlers`)*:*

> *"ผมขับรถบนทางหลวงเมื่อวันก่อน และบังเอิญสังเกตเครื่องวัดระยะทาง เหมือนเครื่องวัด ระยะทางส่วนมาก มันแสดงเลขหกหลักเป็นจำนวนไมล์ถ้วน ๆ เท่านั้น ดังนั้น ถ้ารถของผมเดินทางมาแล้ว 300000 ไมล์ ผมก็จะเห็นเลข 3-0-0-0-0-0 เป็นต้น*
>
> *"คราวนี้นะ สิ่งที่ผมเห็นมันน่าสนใจมาก ผมสังเกตว่าเลขสี่ตัวสุดท้ายเป็นแบบพาลินโดรม (palindromic); นั่นคือ มันเป็นเลขตัวเดียวกันทั้งไปหน้าแล้วกลับหลัง เช่น 5-4-4-5 เป็นพาลินโดรม ดังนั้น เครื่องวัดระยะ ทางของผมอาจจะขึ้นว่า 3-1-5-4-4-5*
>
> *"อีกหนึ่งไมล์ต่อมา เลขห้าตัวสุดท้ายก็เป็นแบบพาลินโดรม เช่น มันอาจจะเป็น 3-6-5-4-5-6 และอีกหนึ่งไมล์หลังจากนั้น เลขสี่ตัวตรงกลางก็เป็นแบบพาลินโดรม คุณพร้อมแล้วใช่ไหม? อีกหนึ่งไมล์ต่อมา เลขทั้งหกตัวก็เป็นแบบพาลินโดรม!*

“คำถามคือ เลขตัวแรกที่ผมสังเกตเห็น คือเลขอะไร?”

ให้เขียนโปรแกรมไพธอนที่ทดสอบเลขที่มีหกหลักทั้งหมด และพิมพ์เลขที่ตรงกับสิ่งที่กำหนดข้างต้นออกมา

เฉลย: `http://thinkpython2.com/code/cartalk2.py`.

แบบฝึกหัด 9.9. *นี่ก็เป็นปริศนาอีกข้อจากรายการ* Car Talk *ที่เราสามารถแก้ได้ด้วยการค้นหา (*`http://www.cartalk.com/content/puzzlers`*):*

> *“เร็ว ๆ นี้ ผมได้ไปเยี่ยมแม่ของผม และเราพบว่าเลขสองหลักที่เป็นอายุของผมนั้นเมื่ออ่านย้อนกลับ แล้วจะเป็นอายุของแม่ เช่น ถ้าแม่อายุ 73 ผมจะอายุ 37 เราสงสัยว่าสิ่งนี้มันเกิดขึ้นบ่อยขนาดไหน ในปีที่ผ่าน ๆ มา แต่เราก็ออกทะเลไปคุยเรื่องอื่น และก็ไม่ได้คิดหาคำตอบ*
>
> *“เมื่อผมกลับถึงบ้าน ผมก็พบว่าเลขของอายุของพวกเรานั้นกลับกันมาหกครั้งแล้ว ผมยังพบว่าถ้าเราโชคดี มันจะเกิดขึ้นอีกครั้งในอีกไม่กี่ปีนี้ และถ้าเราโชคดีมาก ๆ มันก็จะเกิดขึ้นอีกครั้งหลักจากนั้น ดังนั้น คำถามคือ ตอนนี้ผมอายุเท่าไรกันนะ”*

ให้เขียนโปรแกรมไพธอนที่ค้นหาคำตอบแก่ปริศนานี้ คำใบ้: เราอาจจะพบว่า เมธอดของสายอักขระที่ชื่อว่า `zfill` *นั้นเป็นประโยชน์*

เฉลย: `http://thinkpython2.com/code/cartalk3.py`.

10. ลิสต์

บทนี้เป็นเรื่องของชนิดข้อมูลชนิดหนึ่งที่มีประโยชน์มากที่สุดในภาษาไพธอน นั่นคือ ลิสต์ (list) เราจะเรียนเพิ่มเติมในเรื่อง*อ็อบเจกต์* (*object*) และสิ่งที่จะเกิดขึ้น ถ้าเรามีมากกว่าหนึ่งชื่อสำหรับ*อ็อบเจกต์*เดียวกัน

10.1. ลิสต์เป็นข้อมูลแบบลำดับ

ในลักษณะเดียวกับสายอักขระ (string) ลิสต์เป็นลำดับของค่าต่าง ๆ ในสายอักขระ ค่าต่าง ๆ ที่ว่า คือ ตัวอักษร ในลิสต์ ค่าต่าง ๆ นั้นอาจจะเป็นอะไรก็ได้ ค่าต่าง ๆ ในลิสต์นั้น จะเรียกว่า **อิลิเมนต์** (**elements**) หรือบางครั้งก็เรียกว่า **ไอเท็ม** (**items**)

มีหลายวิธีที่ใช้สร้างลิสต์ได้ วิธีที่ง่ายที่สุดคือ ล้อมอิลิเมนต์ต่าง ๆ ไว้ในวงเล็บสี่เหลี่ยม (`[` และ `]`) เช่น

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

ตัวอย่างแรก เป็นลิสต์ของเลขจำนวนเต็มสี่ตัว ตัวอย่างที่สอง เป็นลิสต์ของสายอักขระสามตัว อิลิเมนต์ต่าง ๆ ในลิสต์ไม่จำเป็นต้องเป็นชนิดเดียวกัน ลิสต์ข้างล่างนี้ มีสายอักขระ เลขทศนิยม เลขจำนวนเต็ม และลิสต์อีกอัน

```
['spam', 2.0, 5, [10, 20]]
```

ลิสต์ที่อยู่ในอีกลิสต์หนึ่ง จะเรียกว่า **ลิสต์ซ้อน** (**nested list**)

ลิสต์ที่ไม่มีอิลิเมนต์อยู่เลย จะเรียกว่า ลิสต์ว่าง (empty list) เราสามารถสร้างลิสต์ว่างได้ด้วยวงเล็บสี่เหลี่ยมว่าง ๆ `[]`

ซึ่งก็น่าจะเดาได้ว่า เราสามารถกำหนดค่าที่เป็นลิสต์ให้กับตัวแปรได้

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
```

```
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10.2. ลิสต์เป็นชนิดข้อมูลที่เปลี่ยนแปลงได้

วิธีการเข้าถึงอิลิเมนต์ของลิสต์คล้ายกับวิธีการเข้าถึงตัวอักษรในสายอักขระ นั่นคือ ใช้ตัวดำเนินการวงเล็บสี่เหลี่ยม นิพจน์ภายในวงเล็บสี่เหลี่ยมจะระบุ*ดัชนี*(*index* หรือเลขลำดับ)ของอิลิเมนต์ ทบทวนว่า ดัชนีแรกเริ่มต้นที่ 0

จากตัวอย่างที่แล้ว

```
>>> cheeses[0]
'Cheddar'
```

สิ่งที่ต่างจาก*สายอักขระ*(*string*) ก็คือ ลิสต์เป็นชนิดข้อมูลที่เปลี่ยนแปลงได้ (mutable) ถ้าตัวดำเนินการวงเล็บสี่เหลี่ยมอยู่ทางซ้ายมือของ*การกำหนดค่า*(*assignment*) มันจะระบุอิลิเมนต์ของลิสต์ที่จะกำหนดค่าให้ใหม่

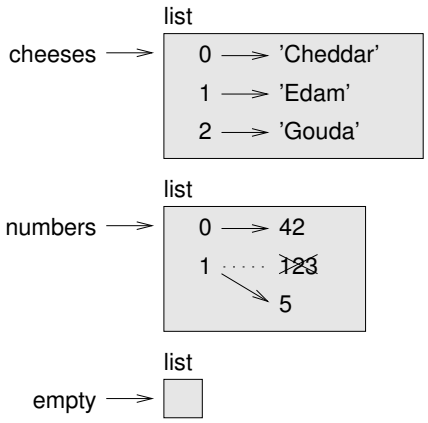
```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

อิลิเมนต์ดัชนี 1 ของตัวแปร `numbers` ที่เคยเป็น 123 ตอนนี้กลายเป็น 5

รูป 10.1 แสดง*แผนภาพสถานะ*(*state diagram*) ของตัวแปร `cheeses`, `numbers` และ `empty` ลิสต์แสดงเป็นกล่องที่มีคำว่า “list” กำกับอยู่ข้างบน และมีอิลิเมนต์ต่าง ๆ อยู่ภายใน. ตัวแปร `cheeses` อ้างถึงลิสต์ที่มีสามอิลิเมนต์ ใช้ดัชนี 0, 1, และ 2 ตัวแปร `numbers` มีสองอิลิเมนต์ แผนภาพแสดงค่าของอิลิเมนต์ที่สอง (ดัชนี 1) ถูกเปลี่ยนจาก 123 เป็น 5 ตัวแปร `empty` อ้างถึงลิสต์ที่ไม่มีอิลิเมนต์

ดัชนีของลิสต์ทำงานแบบเดียวกับดัชนีของสายอักขระ

- นิพจน์ที่ให้ผลเป็นจำนวนเต็มใด ๆ สามารถใช้เป็นดัชนีได้ เช่น



รูปที่ 10.1.: แผนภาพสถานะ

```
>>> cheeses[3 - 2]
'Edam'
```

- ถ้าเราพยายามไปอ่านหรือเขียนอิลิเมนต์ที่ไม่มีอยู่ เราจะได้ `IndexError` ออกมา เช่น

```
>>> cheeses[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- ถ้าดัชนีเป็นเลขลบ มันจะนับย้อนกลับจากอิลิเมนต์สุดท้ายในลิสต์ เช่น

```
>>> cheeses[-1]
'Gouda'
```

ตัวดำเนินการ `in` ก็ทำงานกับลิสต์ได้เหมือนกับในสายอักขระ

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3. การท่องสำรวจลิสต์

วิธีที่นิยมที่สุดในการท่องสำรวจอิลิเมนต์ต่าง ๆ ของลิสต์ คือ การใช้ `for` ลูปไวยากรณ์ก็จะคล้ายกับตอนที่ใช้กับสายอักขระ

```
for cheese in cheeses:
    print(cheese)
```

วิธีนี้ใช้ได้ดี ถ้าเราต้องการแค่อ่านอิลิเมนต์ของลิสต์ แต่ถ้าเราต้องการเขียนหรือเปลี่ยนค่าของอิลิเมนต์ เราต้องใช้ดัชนี วิธีง่าย ๆ คือ ใช้*ฟังก์ชันสำเร็จรูป* (*built-in functions*) ได้แก่ `range` และ `len`

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

ลูปนี้ท่องสำรวจลิสต์ และแก้ค่าของอิลิเมนต์แต่ละตัว ฟังก์ชัน `len` ส่งค่าจำนวนของอิลิเมนต์ในลิสต์ออกมา ฟังก์ชัน `range` ส่งค่าดัชนีจาก 0 ถึง $n-1$ ออกมา โดย n เป็นความยาว[1]ของลิสต์ แต่ละครั้งของลูป ตัวแปร `i` จะรับดัชนีของอิลิเมนต์มา การกำหนดค่าใน*ตัวลูป* (*loop body*) จะใช้ตัวแปร `i` เพื่ออ่านค่าเดิมของอิลิเมนต์ออกมา แล้วค่อยกำหนดค่าใหม่เข้าไป

ถ้าใช้ลูป `for` กับ*ลิสต์ว่าง* (*empty list*) *ตัวลูป* จะไม่ถูกดำเนินการ เช่น

```
for x in []:
    print('This never happens.')
```

ถึงแม้ลิสต์สามารถจะมีสมาชิกเป็นลิสต์อีกอันได้ แต่*ลิสต์ซ้อน*จะนับเป็นแค่หนึ่งอิลิเมนต์ของลิสต์แม่. ตัวอย่างข้างล่างความยาวของลิสต์จึงเป็นแค่สี่

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4. การดำเนินการกับลิสต์

ตัวดำเนินการ **+** ทำการต่อลิสต์เข้าด้วยกัน เช่น

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
```

[1] ความยาวของลิสต์ ก็คือ จำนวนของอิลิเมนต์ในลิสต์

```
>>> c
[1, 2, 3, 4, 5, 6]
```

ตัวดำเนินการ * ให้ลิสต์ซ้ำเท่ากับจำนวนตัวเลขที่ระบุ

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

ตัวอย่างแรกให้ลิสต์ซ้ำของ ลิสต์ `[0]` สี่ครั้ง ตัวอย่างที่สองให้ลิสต์ซ้ำของ ลิสต์ `[1, 2, 3]` สามครั้ง

10.5. การตัดช่วงลิสต์

ตัวดำเนินการตัด (slice operator) ก็ทำงานกับลิสต์ได้เช่นเดียวกับสายอักขระ เช่น

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

ถ้าเราไม่ใส่ดัชนีตัวแรก การตัดจะเริ่มที่ดัชนีเริ่มต้น ถ้าเราไม่ใส่ดัชนีตัวที่สอง การตัดจะไปจนถึงตัวสุดท้าย ถ้าไม่ใส่ดัชนีเลย การตัดจะทำสำเนาลิสต์ทั้งลิสต์ออกมา

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

เพราะว่าลิสต์เปลี่ยนแปลงแก้ไขได้ ดังนั้นส่วนใหญ่แล้ว จะมีประโยชน์ที่เราจะคัดลอกลิสต์ออกมาก่อนที่จะแก้ไขเปลี่ยนแปลงมัน

การใช้ตัวดำเนินการตัดทางด้านซ้ายของการกำหนดค่า สามารถใช้เพื่อแก้ไขค่าอิลิเมนต์หลาย ๆ ข้อความพร้อมรับ ๆ กันได้

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
```

```
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6. เมธอดต่าง ๆ ของลิสต์

ไพธอนมีเมธอดของลิสต์อยู่หลายตัว ตัวอย่างเช่น append เพิ่ม*อิลิเมนต์*ใหม่เข้าไปท้ายลิสต์

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

เมธอด extend รับลิสต์เป็นอาร์กิวเมนต์ และต่ออิลิเมนต์ทั้งหมดเข้าไป

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

ในตัวอย่างนี้ ลิสต์ t2 จะเหมือนเดิม

เมธอด sort จะเรียงอิลิเมนต์ต่าง ๆ ในลิสต์จากน้อยไปมาก

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

เมธอดของลิสต์ส่วนใหญ่ไม่ได้ให้ค่าออกมา นั่นคือ มันแก้สมาชิกของลิสต์ตามหน้าที่ และให้ค่า None ออกมา ถ้าบังเอิญไปเขียน `t = t.sort()` ก็อาจจะผิดหวังได้

10.7. การแปลง การกรอง และการยุบ

ถ้าต้องการบวกเลขต่าง ๆ ที่อยู่ในลิสต์ เราอาจทำเป็นลูปแบบนี้

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

ตัวแปร `total` มีค่าเริ่มต้นเป็น 0 แต่ละครั้งของลูป ตัวแปร `x` รับอิลิเมนต์ทีละตัวมาจากลิสต์ ตัวดำเนินการ `+=` เป็นวิธีเขียนสั้น ๆ เพื่อเปลี่ยนค่าตัวแปร **คำสั่งเสริมสำหรับกำหนดค่า** (**augmented assignment statement**) ข้างล่างนี้

```
    total += x
```

เทียบเท่ากับ

```
    total = total + x
```

ขณะที่ลูปทำงานไป ตัวแปร `total` ก็จะสะสมผลรวมของอิลิเมนต์ ตัวแปรที่ใช้งานในลักษณะนี้ บางครั้งจะเรียกว่า **ตัวสะสม** (**accumulator**)

การบวกอิลิเมนต์ทุกตัวในลิสต์เป็นสิ่งที่ใช้บ่อยมาก จนไพธอนมี*ฟังก์ชันสำเร็จรูป*ให้ คือ `sum`

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

การทำลักษณะนี้ ที่รวบเอาอิลิเมนต์ต่าง ๆ มาเป็นค่า ๆ เดียว บางครั้งจะเรียกว่า *การยุบ* (*reduce*)

บางครั้ง เราอาจต้องการท่องสำรวจลิสต์หนึ่ง เพื่อสร้างอีกลิสต์หนึ่ง ตัวอย่างเช่น ฟังก์ชันต่อไปนี้รับลิสต์ของ*สายอักขระ* (*string*) และสร้างลิสต์ใหม่ออกมา โดยลิสต์ใหม่จะเป็นลิสต์ของสายอักขระ ที่ทุกคำขึ้นต้นด้วยตัวใหญ่

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

ตัวแปร `res` ถูกกำหนดค่าเริ่มต้นเป็นลิสต์ว่าง แต่ละครั้งของลูป เราจะเติมอิลิเมนต์เข้าไปทีละอิลิเมนต์ ดังนั้น `res` ก็เป็น*ตัวสะสม*อีกแบบหนึ่ง

ลักษณะการทำแบบฟังก์ชัน `capitalize_all` บางครั้งจะเรียกว่า การ**แปลง** (**map**) เพราะว่ามัน“แปลง”แต่ละอิลิเมนต์ (ด้วย*ฟังก์ชัน*หรือในที่นี้ ด้วย*เมธอด* `capitalize`) ในลิสต์

ลักษณะงานอีกอย่างที่มักเจอคือ การเลือกบางอิลิเมนต์มาจากลิสต์ และสร้าง*ลิสต์ย่อย*ขึ้นมาใหม่ ตัวอย่างเช่น ฟังก์ชันต่อไปนี้รับลิสต์ของ*สายอักขระ*เข้าไป และให้ลิสต์ที่มีเฉพาะคำที่เป็นอักษรตัวใหญ่ออกมา

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` เป็นเมธอดของสายอักขระ ที่ให้ค่า `True` ถ้าสายอักขระมีแต่อักษรตัวใหญ่

ลักษณะการทำแบบฟังก์ชัน `only_upper` จะเรียกว่า **การกรอง** (**filter**) เพราะว่า มันเลือกเฉพาะบางอิลิเมนต์ และกรองอิลิเมนต์อื่น ๆ ออกไป

การทำงานกับลิสต์ส่วนใหญ่ มักจะสามารถแสดงอยู่ในรูปแบบผสมกันของ การแปลง การกรอง และการยุบได้

10.8. การลบอิลิเมนต์

มีหลาย ๆ วิธีที่จะลบอิลิเมนต์ออกจากลิสต์ ถ้าเรารู้ดัชนีของอิลิเมนต์ที่เราต้องการลบ เราก็สามารถใช้ `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

เมธอด `pop` แก้ไขค่าของลิสต์ และให้อิลิเมนต์ที่ถูกถอดออกมา ถ้าเราเรียกใช้ *เมธอด* `pop` โดยไม่ระบุดัชนี มันจะถอดอิลิเมนต์สุดท้ายออกมาให้

หรือถ้าเราไม่ต้องการได้อิลิเมนต์ที่ถอดออกมา เราสามารถใช้*ตัวดำเนินการ* `del` ได้:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

ถ้าเรารู้อิลิเมนต์ที่ต้องการลบ แต่ไม่รู้ดัชนี เราก็สามารถใช้ `remove` ได้:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

เมธอด `remove` ไม่ได้ให้ค่าออกมา (ให้ `None` ออกมา)

เราสามารถลบหลาย ๆ อิลิเมนต์พร้อม ๆ กันได้ โดยใช้ `del` กับ*ดัชนีตัดช่วง* (*slice index*):

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

เช่นเคย *การตัด*เลือกทุก ๆ อิลิเมนต์ตั้งแต่ดัชนีที่หนึ่งไปจนถึงดัชนีที่สี่ (ไม่รวมอิลิเมนต์ที่ดัชนีที่ห้า)

10.9. ลิสต์และสายอักขระ

สายอักขระ (*string*) เป็นลำดับของ*อักขระ* และลิสต์เป็นลำดับของค่าต่าง ๆ แต่ลิสต์ของอักขระไม่เหมือนกับสายอักขระ

เราสามารถแปลงจากสายอักขระเป็นลิสต์ของอักขระได้ โดยใช้ `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

เพราะว่า `list` เป็นชื่อของฟังก์ชันสำเร็จรูป ดังนั้นเราควรจะหลีกเลี่ยงการใช้คำว่า `list` เป็นชื่อตัวแปร นอกจากนั้น แนะนำว่าควรหลีกเลี่ยงที่จะใช้ตัวอักษรแอลเดี่ยว `l` เพราะว่ามันดูคล้ายกับเลขหนึ่งมาก `1`

ฟังก์ชัน `list` แยกสายอักขระออกมาเป็นอักขระแต่ละตัว (ดังแสดงในตัวอย่างข้างต้น) ถ้าเราต้องการแยกสายอักขระออกมาเป็นคำ ๆ เราควรจะใช้เมธอด `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

นอกจากนั้น เมธอด `split` ยังมีอาร์กิวเมนต์ทางเลือก **delimiter** ที่ใช้ระบุ ตัวอักษรที่ใช้เป็นตัวแบ่งคำได้ ตัวอย่างต่อไปนี้ใช้เครื่องหมายยัติภังค์ (hyphen) เป็นตัวแบ่งคำ:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

เมธอด `join` เป็นการทำตรงข้ามกับ `split` เมธอด `join` รับลิสต์ของสายอักขระ และเชื่อมต่ออิลิเมนต์เข้าด้วยกัน เมธอด `join` เป็นเมธอดของสายอักขระ ดังนั้น เราต้องเรียกใช้จากตัวแปร `delimiter` ที่เป็นชนิดสายอักขระ และส่งลิสต์เข้าไปเป็นพารามิเตอร์:

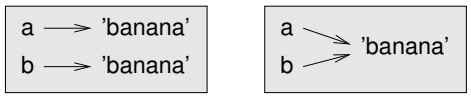
```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

กรณีนี้ ตัวแปร `delimiter` เป็นช่องว่าง ดังนั้น `join` ใส่ช่องว่างระหว่างคำ ถ้าหากต้องการต่อสายอักขระ โดยไม่มีช่องว่างระหว่างคำ เราสามารถใช้สายอักขระว่าง `''` เป็นตัวแบ่งคำได้

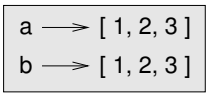
10.10. อ็อบเจกต์และค่า

ถ้าเรารันข้อความคำสั่งกำหนดค่า:

```
a = 'banana'
b = 'banana'
```



รูปที่ 10.2.: แผนภาพสถานะ



รูปที่ 10.3.: แผนภาพสถานะ

เรารู้ว่าทั้ง a และ b อ้างถึงสายอักขระ แต่เราไม่รู้ว่ามันอ้างถึงสายอักขระ*เดียวกัน*หรือไม่ มีความเป็นไปได้อยู่สองอย่าง ดังแสดงในรูป 10.2

ในกรณีที่หนึ่ง a และ b อ้างถึงอ็อบเจกต์ที่ต่างกันสองอ็อบเจกต์ที่มีค่าเหมือนกัน ในกรณีที่สอง ทั้งa และ b อ้างถึงอ็อบเจกต์เดียวกัน

เพื่อจะตรวจดูว่าตัวแปรสองตัวอ้างถึงอ็อบเจกต์เดียวกันหรือไม่ เราสามารถใช้ตัวดำเนินการ `is` ได้

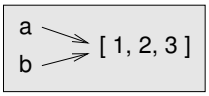
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

ในตัวอย่างนี้ ไพธอนแค่สร้างอ็อบเจกต์สายอักขระขึ้นมาแค่หนึ่งอ็อบเจกต์ และทั้งตัวแปร a และ b ก็อ้างถึงมัน แต่ถ้าเราสร้างลิสต์ออกมา เราจะได้สองอ็อบเจกต์:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

ดังนั้นแผนภาพสถานะแสดงได้ดังรูป 10.3

ในกรณีนี้ เราพูดได้ว่าลิสต์ทั้งสอง**เทียบเท่ากัน** (**equivalent**) เพราะว่า มันมีอิลิเมนต์ต่าง ๆ เหมือนกัน แต่ไม่ใช่**เป็นอันเดียวกัน** (**identical**) เพราะว่ามันไม่ใช่อ็อบเจกต์เดียวกัน ถ้าสองอ็อบเจกต์เป็นอันเดียวกันแล้ว มันจะเทียบเท่ากันด้วย แต่ถ้ามันเทียบเท่ากัน มันไม่จำเป็นต้องเป็นอันเดียวกัน



รูปที่ 10.4.: แผนภาพสถานะ

จนถึงตอนนี้ เราใช้คำว่า “อ็อบเจกต์” และ “ค่า” สลับกันไปมาได้ แต่มันจะถูกต้องมากกว่าที่จะพูดว่าอ็อบเจกต์มีค่า ถ้าเราประเมินค่า `[1, 2, 3]` เราจะได้ลิสต์ของอ็อบเจกต์ที่มีค่าเป็นลำดับของเลขจำนวนเต็มออกมา ถ้าอีกลิสต์หนึ่งมีอิลิเมนต์เหมือน ๆ กัน เราพูดได้ว่ามันมีค่าเหมือนกัน แต่มันไม่ใช่อ็อบเจกต์เดียวกัน

10.11. การทำสมนาม

ถ้าตัวแปร `a` อ้างอิงอ็อบเจกต์ และเรากำหนดให้ `b = a` แล้วตัวแปรทั้งคู่จะอ้างอิงถึงอ็อบเจกต์เดียวกัน:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

แผนภาพสถานะจะเป็นดังแสดงในรูป 10.4

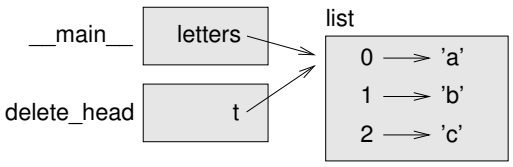
ความเกี่ยวข้องของตัวแปรกับอ็อบเจกต์จะเรียกว่า **การอ้างอิง** (**reference**) ในตัวอย่างนี้ มีสองการอ้างอิงไปที่อ็อบเจกต์เดียวกัน

อ็อบเจกต์ที่มีมากกว่าหนึ่งการอ้างอิง จะมีมากกว่าหนึ่งชื่อ ดังนั้น เราจะเรียกว่า อ็อบเจกต์ถูกทำ**สมนาม** (**aliased**)

ถ้าอ็อบเจกต์ที่ถูกทำสมนาม (อ้างถึงได้จากหลายชื่อ) สามารถเปลี่ยนแปลงค่าได้ (mutable) การเปลี่ยนแปลงที่ทำกับชื่อหนึ่ง จะมีผลไปที่ชื่ออื่น ๆ ด้วย:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

แม้ว่าพฤติกรรมนี้จะมีประโยชน์ แต่มันก็มีแนวโน้มจะสร้างปัญหาอยู่มาก โดยทั่วไปแล้ว มันจะปลอดภัยกว่าที่จะหลีกเลี่ยงการทำสมนามกับอ็อบเจกต์ที่เปลี่ยนแปลงค่าได้ (mutable objects)



รูปที่ 10.5.: แผนภาพกองซ้อน

สำหรับอ็อบเจกต์ที่เปลี่ยนแปลงค่าไม่ได้ เช่น สายอักขระ การทำสมนามจะไม่ได้เป็นปัญหาอะไรมาก ในตัวอย่างนี้:

```
a = 'banana'
b = 'banana'
```

มันแทบจะไม่ต่างเลยว่า `a` และ `b` อ้างถึงสายอักขระเดียวกันหรือไม่

10.12. อาร์กิวเมนต์ที่เป็นลิสต์

เมื่อเราส่งลิสต์เข้าไปให้กับฟังก์ชัน ฟังก์ชันจะได้การอ้างอิงถึงลิสต์ (เป็นการอ้างอิงถึงลิสต์เดิมด้วยชื่อใหม่ ไม่ใช่ได้ลิสต์ใหม่) ดังนั้นถ้าภายในฟังก์ชันมีการแก้ไขค่าของลิสต์ โปรแกรมที่เรียกฟังก์ชันจะเห็นการเปลี่ยนแปลงค่าของลิสต์นี้ด้วย ตัวอย่างเช่น `delete_head` ลบอิลิเมนต์แรกออกจากลิสต์:

```
def delete_head(t):
    del t[0]
```

และนี่คือตัวอย่างการเรียกใช้ฟังก์ชัน:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

พารามิเตอร์ `t` (ในฟังก์ชัน) กับตัวแปร `letters` (ในโปรแกรมที่เรียกฟังก์ชัน เช่น `__main__`) เป็นสมนามของอ็อบเจกต์เดียวกัน แผนภาพกองซ้อนแสดงในรูป 10.5

เนื่องจากลิสต์ถูกอ้างถึงจากทั้งสองตัวแปร ในภาพจึงวาดให้ลิสต์อยู่ระหว่างทั้งสองตัว

มันสำคัญที่จะรู้ความแตกต่างระหว่าง *การดำเนินการ*ที่แก้ไขลิสต์ และ *การดำเนินการ*ที่สร้างลิสต์ใหม่ ตัวอย่าง เมธอด append แก้ไขลิสต์ แต่ตัวดำเนินการ + สร้างลิสต์ใหม่

ตัวอย่างการใช้ append:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

ค่าที่ให้ออกมาจาก append คือ None

ตัวอย่างการใช้ตัวดำเนินการ +:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

ผลลัพธ์จากตัวดำเนินการจะเป็นลิสต์ใหม่ และลิสต์เดิมไม่ได้เปลี่ยนอะไรไป

ความต่างนี้สำคัญมาก ถ้าเราเขียนฟังก์ชันที่อาจมีการแก้ไขลิสต์ ตัวอย่าง ฟังก์ชันข้างล่างนี้*ไม่ได้* ลบหัวของลิสต์ออก:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

ตัวดำเนินการตัดลิสต์ (slice operator) สร้างลิสต์ใหม่ขึ้นมา และการกำหนดค่าได้กำหนดให้ตัวแปร t อ้างถึงลิสต์ใหม่นี้ แต่ทั้งหมดนี้ไม่ได้มีผลกับโปรแกรมที่เรียกฟังก์ชันนี้เลย

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

ในตอนเริ่มต้นฟังก์ชัน `bad_delete_head` ตัวแปร `t` (ในฟังก์ชัน) และตัวแปร `t4` (ในโปรแกรมหลัก) อ้างถึงลิสต์เดียวกัน แต่ตอนท้าย ตัวแปร `t` อ้างถึงลิสต์ใหม่ ในขณะที่ตัวแปร `t4` ยังอ้างถึงลิสต์เดิมอยู่ ลิสต์เดิมที่ไม่ได้ถูกแก้ไข

วิธีที่ดีกว่า คือ เขียนฟังก์ชันที่สร้างและให้ค่าของลิสต์ใหม่ออกมา ตัวอย่างเช่น ฟังก์ชัน `tail` ให้ค่าของลิสต์ออกมาทั้งหมด ยกเว้นอิลิเมนต์แรกสุด:

```
def tail(t):
    return t[1:]
```

ฟังก์ชันนี้ก็ไม่ได้เปลี่ยนลิสต์ดั้งเดิม วิธีเรียกใช้มันคือ:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13. การดีบัก

การใช้ลิสต์อย่างไม่ระวัง (รวมถึงอ็อบเจกต์ที่สามารถแก้ไขค่าได้ชนิดอื่น ๆ ด้วย) อาจนำไปสู่ปัญหาที่ใช้เวลานานมากในการดีบัก ต่อไปนี้เป็นตัวอย่างของความผิดพลาดที่พบบ่อย และวิธีที่จะหลีกเลี่ยง:

1. เมธอดของลิสต์เกือบทั้งหมดแก้ค่าในอาร์กิวเมนต์ และมักส่งค่า `None` ออกมา (`return None`) พฤติกรรมนี้จะต่างจากเมธอดของสายอักขระ ที่มักส่งสายอักขระใหม่ออกมา โดยไม่ไปยุ่งกับสายอักขระเดิม

   ถ้าเคยเขียนโปรแกรมแบบนี้:

   ```
   word = word.strip()
   ```

   มันอาจจะมีแนวโน้มที่จะเขียนโปรแกรมกับลิสต์แบบนี้:

   ```
   t = t.sort()              # WRONG!
   ```

   แต่เมธอด `sort` ส่งค่า `None` ออกมา หลังจากนั้น ไม่ว่าเราจะทำอะไรกับตัวแปร `t` ก็ไม่น่าจะได้เรื่องอะไร

   ก่อนจะใช้เมธอดหรือตัวดำเนินการใด ๆ ของลิสต์ ให้อ่านเอกสารให้ถี่ถ้วน และทดสอบเมธอดหรือตัวดำเนินการเหล่านั้น ในการทำงานแบบโต้ตอบ (interactive mode) ก่อน

2. เลือกรูปแบบการเขียนและยึดติดกับมัน

   ส่วนหนึ่งของปัญหาของการทำงานกับลิสต์ คือ มีวิธีที่จะทำงานหลายวิธีมาก ตัวอย่างเช่น การลบอิลิเมนต์จากลิสต์ เราสามารถใช้ pop หรือ remove หรือ del หรือแม้แต่จะใช้การกำหนดค่าและการตัดลิสต์ (slice assignment)

   การเพิ่มอิลิเมนต์เอง เราก็สามารถใช้เมธอด append หรือตัวดำเนินการ + สมมติว่า t เป็นลิสต์และ x เป็นสมาชิกของลิสต์ วิธีเพิ่มอิลิเมนต์ข้างล่างนี้ถูกต้อง:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   แต่วิธีข้างล่างนี้ผิด:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   ลองตัวอย่างแต่ละอันในการทำงานแบบโต้ตอบ เพื่อให้แน่ใจว่าเข้าใจการทำงานของมันก่อน สังเกตว่า มีเฉพาะตัวอย่างสุดท้าย (t = t + x) ที่ให้ runtime error ออกมา อีกสามตัวอย่างข้างต้น แม้ว่าไม่ได้ให้ runtime error ออกมา แต่มันทำงานผิดจากที่เราต้องการ

3. คัดลอก (copy) เพื่อเลี่ยงปัญหาจากการทำสมนาม

   เช่น ถ้าหากเราต้องการใช้เมธอดอย่าง sort ที่แก้ไขข้อมูลของลิสต์ แต่ถ้าเราต้องการเก็บข้อมูลเดิมของลิสต์ไว้ด้วย เราสามารถใช้การคัดลอกทำสำเนาไว้ได้

   ```
   >>> t = [3, 1, 2]
   >>> t2 = t[:]
   >>> t2.sort()
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

   ตัวอย่างนี้ เราสามารถใช้ฟังก์ชันสำเร็จรูป sorted ก็ได้ ซึ่งฟังก์ชันนี้ส่งคืนค่าลิสต์ใหม่ที่จัดเรียงแล้วออกมา โดยไม่ไปเปลี่ยนแปลงลิสต์เดิม

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14. อภิธานศัพท์

ลิสต์ (list): ข้อมูลค่าต่าง ๆ ในลักษณะลำดับ

อิลิเมนต์ (element): ข้อมูลค่าแต่ละค่าในลิสต์ (หรือ รวมไปถึงแต่ละค่าของข้อมูลลักษณะลำดับแบบอื่น ๆ) บางครั้งอาจเรียก ค่ารายการ (item)

ลิสต์ซ้อน (nested list): ลิสต์ที่เป็นอิลิเมนต์ของอีกลิสต์

ตัวสะสม (accumulator): ตัวแปรที่ใช้ในลูป เพื่อเพิ่มค่า หรือเพื่อเก็บสะสมผลลัพธ์

การกำหนดเสริมค่า (augmented assignment): ข้อความคำสั่งที่แก้ไขค่าของตัวแปร โดยใช้ตัวดำเนินการ เช่น **+=**

การยุบ (reduce): รูปแบบการประมวลผลที่สำรวจค่าต่าง ๆ ในลำดับ และรวมสรุปค่าอิลิเมนต์ต่าง ๆ มาเป็นค่า ๆ เดียว

การแปลง (map): รูปแบบการประมวลผลที่สำรวจค่าต่าง ๆ ในลำดับ และทำการดำเนินการกับอิลิเมนต์แต่ละตัว

การกรอง (filter): รูปแบบการประมวลผลที่สำรวจค่าต่าง ๆ ในลำดับ และเลือกเฉพาะอิลิเมนต์ที่ผ่านเงื่อนไขออกมา

อ็อบเจกต์ (object): สิ่งที่ตัวแปรอ้างถึงได้ อ็อบเจกต์มีชนิดและค่า

เทียบเท่ากัน (equivalent): มีค่าเหมือนกัน (แต่ไม่จำเป็นต้องเป็นอ็อบเจกต์อันเดียวกัน)

เป็นอันเดียวกัน (identical): เป็นอ็อบเจกต์เดียวกัน

การอ้างอิง (reference): ความเกี่ยวเนื่องเชื่อมโยงกันของตัวแปรและค่าของมัน

การทำสมนาม (aliasing): สถานการณ์ที่ตัวแปรมากกว่าสองตัวขึ้นไปอ้างถึงอ็อบเจกต์เดียวกัน

ตัวแบ่งคำ (delimiter): ตัวอักษร หรือสายอักขระ ที่ใช้เพื่อระบุว่า สายอักขระทั้งหมดควรจะถูกแบ่งที่ใด

10.15. แบบฝึกหัด

ผู้อ่านสามารถดาวน์โหลดเฉลยของแบบฝึกหัดเหล่านี้ได้จาก `http://thinkpython2.com/code/list_exercises.py`

แบบฝึกหัด **10.1.** *จงเขียนฟังก์ชัน ชื่อ* `nested_sum` *ที่รับลิสต์ของลิสต์ของเลขจำนวนเต็ม และบวกค่าอิลิเมนต์จากทุกลิสต์ซ้อนทั้งหมด ตัวอย่าง:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

แบบฝึกหัด **10.2.** *จงเขียนฟังก์ชัน ชื่อ* `cumsum` *ที่รับลิสต์ของตัวเลข และให้ค่าผลบวกสะสมออกมา นั่นคือ ลิสต์ใหม่ที่เป็นผลลัพธ์มีอิลิเมนต์ที่ i เป็นผลรวมของอิลิเมนต์ $i+1$ ตัวแรกของลิสต์ต้นฉบับ ตัวอย่าง:*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

แบบฝึกหัด **10.3.** *จงเขียนฟังก์ชัน ชื่อ* `middle` *ที่รับลิสต์ และให้ค่าลิสต์ใหม่ออกมา โดยที่ลิสต์ใหม่นั้นมีอิลิเมนต์อื่น ๆ เหมือนกับลิสต์ที่ใส่เข้าไป แต่ไม่มีอิลิเมนต์แรก ไม่มีอิลิเมนต์สุดท้าย ตัวอย่าง:*

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

แบบฝึกหัด **10.4.** *จงเขียนฟังก์ชัน ชื่อ* `chop` *ที่รับลิสต์ แล้วไปแก้ไขค่าของมัน โดยลบอิลิเมนต์แรกสุดและลบอิลิเมนต์ท้ายสุดออก ฟังก์ชันนี้ให้ค่า* `None` *ออกมา ตัวอย่าง:*

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
```

```
[2, 3]
```

แบบฝึกหัด **10.5.** *จงเขียนฟังก์ชัน ชื่อ* `is_sorted` *ที่รับลิสต์ และส่งค่า* `True` *ออกมา ถ้าลิสต์ถูกเรียงลำดับจากน้อยไปมาก และส่งค่า* `False` *ออกมา ถ้าไม่ใช่ ตัวอย่าง:*

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

แบบฝึกหัด **10.6.** *คำสองคำจะเรียกว่าเป็น อนาแกรม (anagrams) ถ้าเราสามารถเรียงตัวอักษรในคำหนึ่งให้สะกดเป็นอีกคำได้ จงเขียนฟังก์ชัน ชื่อ* `is_anagram` *ที่รับสายอักขระสองสาย และให้ค่า* `True` *ออกมา ถ้าสายอักขระทั้งสองเป็นอนาแกรม*

แบบฝึกหัด **10.7.** *จงเขียนฟังก์ชัน ชื่อ* `has_duplicates` *ที่รับลิสต์ และให้ค่า* `True` *ออกมา ถ้ามีอิลิเมนต์ในลิสต์ที่ปรากฏมากกว่าหนึ่งครั้ง โดยฟังก์ชันนี้ไม่เปลี่ยนแปลงค่าลิสต์ต้นฉบับ*

แบบฝึกหัด **10.8.** *แบบฝึกหัดนี้เกี่ยวกับ ปฏิทรรศน์วันเกิด (Birthday Paradox ซึ่งศึกษาเพิ่มเติมได้จาก* `http://en.wikipedia.org/wiki/Birthday_paradox`*).*

ถ้ามีนักเรียนในห้อง 23 คน มีโอกาสที่จะมีนักเรียนสองคนที่วันเกิดตรงกันเป็นเท่าไร จงประมาณความน่าจะเป็น โดยสร้างตัวอย่างสุ่มของวันเกิด 23 วัน และตรวจสอบว่ามีวันตรงกันหรือไม่ คำใบ้: เราสามารถสร้างตัวอย่างวันเกิดสุ่มได้ด้วยฟังก์ชัน `randint` *ในโมดูล* `random`

เฉลยดาวน์โหลดได้จาก `http://thinkpython2.com/code/birthday.py`

แบบฝึกหัด **10.9.** *จงเขียนฟังก์ชันที่อ่านไฟล์* `words.txt` *(ดาวน์โหลดไฟล์ได้จาก* `http://greenteapress.com/thinkpython2/code/words.txt`*) และสร้างลิสต์ที่แต่ละอิลิเมนต์เป็นแต่ละคำในไฟล์ ให้เขียนฟังก์ชันเป็นสองแบบ แบบแรกใช้เมธอด* `append` *และอีกแบบใช้* `t = t + [x]` *เปรียบเทียบว่าแบบไหนใช้เวลารันนานกว่า? ทำไม?*

เฉลย: `http://thinkpython2.com/code/wordlist.py`

แบบฝึกหัด **10.10.** *เพื่อตรวจสอบว่าคำอยู่ในลิสต์ของคำหรือไม่ เราควรจะใช้ตัวดำเนินการ* `in` *แต่มันจะทำงานช้า เพราะว่ามันตรวจโดยการค้นหาทีละอิลิเมนต์ตามลำดับ*

ถ้าเรารู้ว่าคำในลิสต์เรียงตามลำดับตัวอักษรระอยู่แล้ว เราสามารถทำให้การค้นหาเร็วขึ้นได้ ด้วยวิธีค้นหาแบ่งสองส่วน (bisection search หรืออีกชื่อ binary search) วิธีค้นหาแบ่งสองส่วน คล้ายกับวิธีที่เราทำ เวลาที่เราหาคำในพจนานุกรม เราเริ่มที่ตรงกลาง และดูว่าคำที่เราหามาก่อนหรือหลังคำที่ตรงกลางลิสต์ ถ้าคำที่หามาก่อน ก็ให้ไปหาในครึ่งหน้าด้วยแนวทางนี้อีก ถ้าคำที่หามาหลัง ก็ให้ไปหาในครึ่งหลัง ไม่ว่าอย่างไร เราก็ลดปริภูมิค้นหา(search space)ลงไปครึ่งหนึ่ง ถ้าในลิสต์ของคำมีคำอยู่ 113,809 คำ มันจะใช้แค่ประมาณ 17 ขั้น เพื่อหาคำให้เจอ หรือเพื่อบอกว่าคำนั้นไม่อยู่ในลิสต์

จงเขียนฟังก์ชัน `in_bisect` *ที่รับลิสต์ที่เรียงลำดับไว้แล้ว กับรับค่าที่ต้องการค้นหา แล้วส่งดัชนีของค่าที่หาออกมาถ้าลิสต์มีค่านั้นอยู่ หรือส่งค่า* `None` *ออกมา ถ้าลิสต์ไม่มีค่าที่หา*

หรือ อ่านเอกสารของโมดูล `bisect` *และเรียกใช้! เฉลย:* `http://thinkpython2.com/code/inlist.py`

แบบฝึกหัด **10.11.** *คำสองคำเป็น "คู่กลับ" (reverse pair) ถ้าคำหนึ่งเป็นคำเรียงกลับของอีกคำหนึ่ง จงเขียนโปรแกรมที่หาคู่กลับทั้งหมดในลิสต์ของคำ เฉลย:* `http://thinkpython2.com/code/reverse_pair.py`

แบบฝึกหัด **10.12.** *คำสองคำจะเรียกว่า "เกี่ยวติดกัน" (interlock) ถ้านำอักษรจากแต่ละคำมาเรียงสลับกันแล้วได้คำใหม่ ตัวอย่างเช่น "shoe" และ "cold" เกี่ยวติดกันแล้วได้คำใหม่คือ "schooled" เฉลย:* `http://thinkpython2.com/code/interlock.py` *ขอบคุณ: แบบฝึกหัดนี้ได้รับแรงบันดาลใจจากตัวอย่างใน* `http://puzzlers.org`

1. *กำหนดให้ลิสต์ของคำ (ใช้ลิสต์ของคำจากไฟล์ตัวอย่าง* `words.txt` *ได้) จงเขียนโปรแกรมที่หาทุกคู่ของคำในลิสต์ ที่เกี่ยวติดกันเป็นคำใหม่ ที่ก็อยู่ในลิสต์ คำใบ้: ไม่ต้องนับทุกคู่ (เราอาจหาคำที่เกี่ยวติดกันจากคำสองคู่ว่าอยู่ในลิสต์หรือไม่ หรืออาจดูว่าคำในลิสต์ว่าแยกออกมาเป็นคำสองคำอะไร)*

2. *คุณหาคำต่าง ๆ ที่เกี่ยวติดกันสามทางได้หรือเปล่า? คำที่เกี่ยวติดกันสามทาง (three-way interlocked) ได้มาจากสามคำโดย อักษรในคำได้จากคำต้นฉบับทั้งสามเรียงสลับกัน เช่น powered ได้จากการเกี่ยวติดกันสามทางของ ped และ or และ we เป็นต้น*

11. ดิกชันนารี

บทนี้นำเสนอ*ชนิดข้อมูลสำเร็จรูป* (*built-in type*) อีกชนิด ที่เรียกว่า ดิกชันนารี (dictionary) ดิกชันนารีเป็นลักษณะเฉพาะของไพธอนที่จัดว่าดีที่สุดอันหนึ่ง มันเป็นเสมือนส่วนประกอบพื้นฐานของอัลกอริธึมที่มีประสิทธิภาพที่แจ๋ว ๆ ต่าง ๆ

11.1. ดิกชันนารีเป็นการแปลง

ดิกชันนารีคล้ายกับลิสต์ แต่ใช้งานได้กว้างขวางกว่า สำหรับลิสต์ ดัชนีต้องเป็นเลขจำนวนเต็มเท่านั้น แต่ดิกชันนารีสามารถกำหนดใช้ดัชนีเป็นข้อมูลใดก็ได้ (เกือบทุกชนิด)

ดิกชันนารีประกอบไปด้วยกลุ่มหมู่ของดัชนีต่าง ๆ เรียกว่า **กุญแจ** (**keys**) และกลุ่มหมู่ของค่าต่าง ๆ ที่เชื่อมโยงกับดัชนี แต่ละกุญแจจะเชื่อมไปหาค่า (value) การเชื่อมโยงระหว่างกุญแจกับค่าจะเรียกว่า **คู่กุญแจกับค่า** (**key-value pair**) หรือบางครั้งอาจเรียก **รายการ** (**item**)

หากพูดในเชิงคณิตศาสตร์ ดิกชันนารีก็คือ**การแปลง** (**mapping**) จาก กุญแจ (keys) ไปเป็น ค่า (values) ซึ่งเราสามารถพูดได้ว่า แต่ละดัชนีกุญแจ “แปลงไปเป็น” ค่า ดังตัวอย่างนี้ เราจะสร้างดิกชันนารีที่แปลงจากคำในภาษาอังกฤษไปเป็นคำในภาษาสเปน ดังนั้น ทั้งกุญแจและค่า จะเป็นชนิดข้อมูลสายอักขระ

ฟังก์ชัน `dict` สร้างดิกชันนารีใหม่ขึ้นมา โดยยังไม่มีรายการใด ๆ อยู่ภายใน เนื่องจาก `dict` เป็นชื่อของฟังก์ชันสำเร็จ เราจึงต้องไม่ใช้มันไปตั้งเป็นชื่อตัวแปร

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

วงเล็บหยัก ๆ `{}` แทนดิกชันนารีที่ว่างอยู่ เพื่อจะเพิ่มรายการเข้าไปในดิกชันนารี เราสามารถใช้วงเล็บสี่เหลี่ยมดังนี้:

```
>>> eng2sp['one'] = 'uno'
```

คำสั่งบรรทัดนี้สร้างรายการขึ้นมา โดยแปลงจากกุญแจ `'one'` ไปเป็นค่า `'uno'` ถ้าเราสั่งพิมพ์ค่าดิกชันนารีออกมาดู เราจะเห็น รายการคู่กุญแจค่า ที่มีเครื่องจุดคู่ (ทวิภาค หรือ colon) คั่นระหว่างกุญแจกับค่า:

```
>>> eng2sp
{'one': 'uno'}
```

รูปแบบเอาต์พุตนี้ ก็สามารถใช้เป็นรูปแบบอินพุตได้ เช่น เราสามารถสร้างดิกชันนารีใหม่ ที่มีสามรายการได้ดังนี้:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

แต่ถ้าเราสั่งพิมพ์ `eng2sp` ออกมาดู เราอาจจะแปลกใจที่เห็น:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

ลำดับของรายการคู่กุญแจค่าอาจจะไม่เหมือนเดิม หรือแม้แต่เวลาที่คุณไปทดลองตัวอย่างนี้ในเครื่องของคุณ คุณก็อาจจะได้ผลลัพธ์ต่างออกไปก็ได้ โดยทั่วไปแล้ว ลำดับของรายการในดิกชันนารี อาจเปลี่ยนแปลงไป

แต่ลำดับของรายการในดิกชันนารีไม่ใช่ปัญหา เพราะว่ารายการต่าง ๆ ในดิกชั่นารีจะไม่ถูกอ้างจากดัชนีลำดับ สำหรับดิกชันนารีเราจะใช้ดัชนีกุญแจเพื่อหาค่าที่เป็นผูกกับมัน:

```
>>> eng2sp['two']
'dos'
```

กุญแจ `'two'` จะแปลงไปเป็นค่า `'dos'` เสมอ ดังนั้น ลำดับของรายการจึงไม่สำคัญ

ถ้าหากกุญแจไม่มีอยู่ในดิกชันนารี เราจะได้รับแจ้งข้อผิดพลาดมาแทน

```
>>> eng2sp['four']
KeyError: 'four'
```

ฟังก์ชัน `len` ก็ได้งานได้กับดิกชันนารี มันจะให้ค่าจำนวนของรายการคู่กุญแจค่าออกมา:

```
>>> len(eng2sp)
3
```

ตัวดำเนินการ `in` ก็ทำงานได้กับดิกชันนารี มันจะบอกเราว่า สิ่งที่เราสงสัยมีปรากฏเป็น *กุญแจ* (*key*) ในดิกชันนารีหรือไม่ (ปรากฏเป็นค่าไม่นับ)

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

เพื่อจะดูว่ามีอะไรเป็นค่าของกุญแจในดิกชันนารีบ้าง เราสามารถใช้เมธอด `values` ที่ให้ค่ากุญแจต่าง ๆ ในดิกชันนารีออกมา จากนั้น เราสามารถใช้ตัวดำเนินการ `in` ได้:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

ตัวดำเนินการ `in` ใช้อัลกอริธึมสำหรับดิกชันนารี ต่างกับอัลกอริธึมที่ใช้สำหรับลิสต์ สำหรับลิสต์ มันจะค้นหาอิลิเมนต์ของลิสต์ตามลำดับ เช่นในหัวข้อ 8.6 ถ้าลิสต์ยาวขึ้น เวลาในการค้นหาก็จะนานขึ้น เวลาค้นหากับความยาวลิสต์เป็นสัดส่วนกันโดยตรง

สำหรับดิกชันนารี ไพธอนใช้อัลกิริธึมที่เรียกว่า **ตารางแฮช** (**hashtable**) ซึ่งมีคุณสมบัติที่ยอดเยี่ยม นั่นคือ ตัวดำเนินการ `in` ใช้เวลาพอ ๆ กันในการค้นหา ไม่ว่าดิกชันนารีจะมีรายการอยู่มากเท่าไร รายละเอียดของ**ตารางแฮช**อธิบายอยู่ในหัวข้อ B.4 แต่คำอธิบายอาจจะเข้าใจได้ยาก หากไม่ได้ศึกษาบทต่อ ๆ ไปก่อน

11.2. ดิกชันนารีเป็นกลุ่มหมู่ของตัวนับ

สมมติว่าเราได้สายอักขระมา และเราต้องการนับจำนวนว่าตัวอักษรแต่ละตัวมีปรากฏอยู่ในสายอักขระนั้นอีกครั้ง มีหลายวิธีที่เราทำได้ เช่น:

1. วิธีหนึ่ง เราอาจจะสร้างตัวแปรขึ้นมา 26 ตัวแปร แต่ละตัวแปรสำหรับแต่ละตัวอักษร แล้วเราก็จะเข้าไปไล่ดูภายในสายอักขระ และนับจำนวนของแต่ละอักขระ ใช้ตัวแปรแต่ละตัวเก็บจำนวนนับของแต่ละตัวอักษร โดยอาจจะใช้*เงื่อนไขลูกโซ่* (คำสั่งพวก `if-elif-else`) ช่วย

2. วิธีหนึ่ง เราสามารถสร้างลิสต์ที่มี 26 อิลิเมนต์ขึ้นมา แล้วเราก็แปลงตัวอักษรแต่ละตัวเป็นตัวเลข (อาจจะใช้ฟังก์ชันสำเร็จ `ord`) และใช้ตัวเลขที่ได้เป็นดัชนีของลิสต์ และเพิ่มจำนวนนับในลิสต์

3. อีกวิธีหนึ่ง เราสามารถสร้างดิกชันนารี ที่ใช้ตัวอักษรเป็นกุญแจ และใช้ค่าของกุญแจเป็นจำนวนนับ เวลาที่เจอตัวอักษรเป็นครั้งแรก เราจะเพิ่มรายการเข้าไปในดิกต์ หลังจากนั้น เราจะเพิ่มค่าจำนวนนับของอักษรตัวนั้น

แต่ละวิธีจะทำงานเดียวกัน แต่ว่า ทำด้วยวิธีที่ต่างกัน

วิธีการสร้างโปรแกรม (**implementation**) เป็นวิธีที่จะทำงาน วิธีที่จะทำการคำนวณ วิธีที่จะเขียนโปรแกรม บางวิธีจะดีกว่าวิธีอื่น ตัวอย่าง เช่น ข้อดีของวิธีที่ใช้ดิกชันนารีคือ เราไม่จำเป็นต้องรู้ก่อนว่า ตัวอักษรมีอยู่ในสายอักขระหรือไม่ เราแค่สร้างที่เก็บตัวนับให้มัน ถ้ามันมีปรากฏอยู่ หน้าตาของโปรแกรมอาจจะเป็นดังนี้:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

ชื่อของฟังก์ชันข้างต้นคือ `histogram` ซึ่งเป็นคำศัพท์ทางสถิติที่ใช้สำหรับกลุ่มหมู่ของตัวนับ (หรือความถี่)

บรรทัดแรกของฟังก์ชันสร้างดิกชันนารีว่าง ๆ ขึ้นมา คำสั่งลูป `for` ไล่สำรวจอักษรต่าง ๆ ในสายอักขระ แต่ละรอบในลูป ถ้าตัวอักษรของตัวแปร `c` ไม่อยู่ในดิกชันนารี เราจะสร้างรายการใหม่ด้วยกุญแจ `c` และให้ค่าเริ่มต้นเป็น 1 (เนื่องจากเราเจอตัวอักษรแล้วหนึ่งครั้ง) ถ้าตัวอักษรของตัวแปร `c` มีอยู่แล้วในดิกชันนารี เราก็เพิ่มค่า `d[c]`

มันทำงานดังนี้:

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

ฮิสโตแกรมที่ได้บอกว่า อักษร `a` และอักษร `b` มีปรากฏหนึ่งครั้ง อักษร `o` มีปรากฏสองครั้ง เป็นต้น

ดิกชันนารีมีเมธอด `get` ที่รับกุญแจ และค่าโดยปริยาย (ค่าดีฟอลท์) ของมัน ถ้ากุญแจนั้นมีอยู่ในดิกชันนารีอยู่แล้ว `get` จะให้ค่าของกุญแจนั้นออกมา ไม่อย่างนั้นก็จะให้ค่าโดยปริยายออกมา ตัวอย่าง:

```
>>> h = histogram('a')
>>> h
```

```
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

เพื่อเป็นการฝึกใช้ ให้ลองใช้ `get` เพื่อเขียน `histogram` ให้กระชับยิ่งขึ้น เราน่าจะสามารถเอาคำสั่ง `if` ออกไปได้

11.3. ลูปและดิกชันนารี

ถ้าเราใช้ดิกชันนารีในคำสั่ง `for` มันจะท่องสำรวจกุญแจของดิกชันนารี ตัวอย่าง `print_hist` พิมพ์กุญแจ และค่าของกุญแจ แต่ละรายการออกมา:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

ผลลัพธ์ที่ได้จะเป็นดังนี้:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

ย้ำอีกครั้งว่า กุญแจจะไม่ได้เรียงตามลำดับใด ๆ ถ้าหากต้องการทำลูปสำรวจกุญแจตามลำดับ เราสามารถใช้ฟังก์ชันสำเร็จ `sorted`:

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
```

```
r 2
t 1
```

11.4. การเทียบค้นย้อนกลับ

ถ้าให้ดิกชันนารี **d** และกุญแจ **k** เราสามารถหาค่าของกุญแจ `v = d[k]` ได้ง่าย ๆ การดำเนินการแบบนี้เรียกว่า **การเทียบค้น** (**lookup**)

แต่หากเรามี **v** และต้องการหาค่า **k** หละ? เราเจอปัญหาสองอย่างคือ: หนึ่ง มันอาจจะมีกุญแจมากกว่าหนึ่งกุญแจที่เชื่อมไปหาค่า **v** เหมือนกัน อันนี้ก็ขึ้นกับงาน เราอาจจะเลือกสักกุญแจหนึ่ง หรือ เราอาจจะสร้างลิสต์ของกุญแจทั้งหมดของค่านั้นขึ้นมา สอง ไม่มีคำสั่งง่าย ๆ ที่จะทำ**การเทียบค้นย้อนกลับ**ให้ เราต้องค้นหาเอง

ข้างล่างนี้ คือ ฟังก์ชันที่รับค่าของกุญแจ และส่งกุญแจแรกที่เจอออกมา:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

ฟังก์ชันนี้เป็นอีกตัวอย่างหนึ่งของการค้นหารูปแบบ เพียงแต่ว่า มีการใช้คำสั่งใหม่ คำสั่ง `raise` **คำสั่ง raise** (**raise statement**) จะทำให้เกิดเอ็กเซ็ปชั่น (exception) ขึ้น ในกรณีนี้ มันจะไปเรียก `LookupError` ซึ่งเป็นเอ็กเซ็ปชั่นสำเร็จรูป (built-in exception) เพื่อแจ้งว่าการดำเนินการเทียบค้นล้มเหลว

ถ้าเราค้นหาไปจนสุดจบลูป นั่นหมายความว่า **v** ไม่ได้ปรากฏเป็นค่าในดิกชันนารี ดังนั้นเราจึงส่งเอ็กเซ็ปชั่นออกไป

ตัวอย่างนี้แสดง การเทียบค้นย้อนกลับที่สำเร็จ:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

และนี่คือตัวอย่างอันที่ไม่สำเร็จ:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

ผลลัพธ์จากที่เราส่งเอ็กเซ็ปชั่นออกมา จะเหมือนกับตอนที่ไพธอนส่งออกมา: มันจะพิมพ์ข้อความสืบย้อน (trackback message) และข้อความแสดงข้อผิดพลาดออกมา (error message)

คำสั่ง `raise` สามารถรับข้อความแสดงข้อผิดพลาด เป็นอาร์กิวเมนต์เสริมได้ ตัวอย่างเช่น:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

การเทียบค้นย้อนกลับจะช้ากว่าการเทียบค้นธรรมดามาก ถ้าเราต้องทำบ่อย ๆ หรือถ้าดิกชันนารีมีขนาดใหญ่ ประสิทธิภาพของโปรแกรมจะแย่มาก

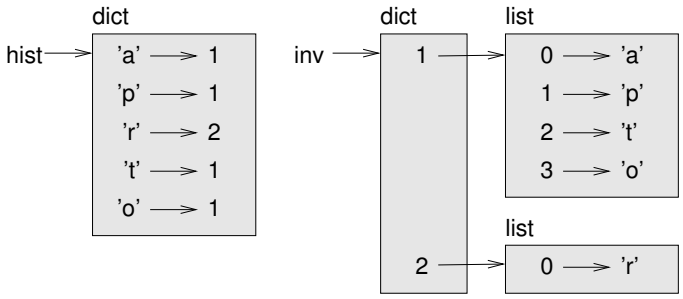
11.5. ดิกชันนารีและลิสต์

ลิสต์สามารถเป็นค่าในดิกชันนารีได้ ตัวอย่างเช่น ถ้าเราได้ดิกชันนารีที่แปลงจากตัวอักษรไปเป็นความถี่ เราอาจจะต้องการทำผกผันมัน นั่นคือ สร้างดิกชันนารีที่แปลงจากความถี่ไปเป็นตัวอักษร

เนื่องจากอาจจะมีอักษรหลายตัวที่มีความถี่เดียวกัน แต่ละค่าของดิกชันนารีผกผัน (inverted dictionary ซึ่งคือ ดิกชันนารีที่แปลงจากความถี่ไปเป็นตัวอักษร) ควรจะเป็นลิสต์ของตัวอักษร

นี่เป็นฟังก์ชันที่ทำผกผันดิกชันนารี:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
```



รูปที่ 11.1.: แผนภาพสถานะ

```
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

แต่ละรอบของลูป `key` รับกุญแจจาก `d` และ `val` รับค่าที่คู่กับกุญแจ ถ้าค่า `val` ไม่อยู่ใน `inverse` นั่นหมายความว่า เราไม่เคยเจอมันมาก่อน ดังนั้นเราจะสร้างรายการใหม่ และให้ค่าเริ่มต้นมันเป็น **เซตโทน** (singleton ซึ่งคือลิสต์ที่มีอิลิเมนต์เดียว) ถ้าไม่อย่างนั้น ก็คือเราเคยเห็นค่านี้มาก่อน ดังนั้นเราจะเพิ่มค่านี้เข้าไปในลิสต์

ตัวอย่าง:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

รูป 11.1 เป็นแผนภาพสถานะ ที่แสดง `hist` และ `inverse` ดิกชันนารี แสดงด้วยกล่องที่มีชนิด `dict` อยู่ข้างบน และมี*คู่กุญแจ-ค่า*เป็นคู่ ๆ อยู่ข้างใน ถ้าค่าเป็นเลขจำนวนเต็ม เลขทศนิยม หรือสายอักขระ มันจะวาดอยู่ในกล่อง แต่ถ้าเป็นลิสต์ อาจจะวาดอยู่นอกกล่อง เพื่อให้แผนภาพดูง่าย

ลิสต์สามารถเป็นค่าในดิกชันนารีได้ เช่นที่แสดงในตัวอย่างนี้ แต่ลิสต์ไม่สามารถใช้เป็นกุญแจได้ ข้างล่างนี้เป็นตัวอย่างว่าจะเกิดอะไรขึ้น ถ้าเราลอง:

```
>>> t = [1, 2, 3]
```

```
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

ตามที่ได้บอกไว้ตอนต้นว่า ดิกชันนารีถูกสร้างขึ้นมาด้วยตารางแฮช (hashtable) และนั่นหมายถึงว่า กุญแจต่าง ๆ ต้อง**สามารถทำแฮชได้** (**hashable**)

แฮช (**hash**) เป็นฟังก์ชันที่รับค่า (ของชนิดข้อมูลใด ๆ) และส่งคืนค่าเลขจำนวนเต็มออกมา ดิกชันนารีใช้ค่าเลขจำนวนเต็มต่าง ๆ ที่ได้นี้ ซึ่งเรียกว่า ค่าแฮช (hash values) เพื่อเก็บและค้นหา*คู่กุญแจค่า*ของดิกต์

ระบบนี้ทำงานได้อย่างดี ถ้ากุญแจต่าง ๆ ไม่สามารถถูกเปลี่ยนแปลงได้ แต่ถ้ากุญแจต่าง ๆ เปลี่ยนได้แบบเดียวกับลิสต์ ปัญหาจะเกิดขึ้น ตัวอย่างเช่น ถ้าเราสร้างคู่กุญแจค่า ไพธอนจะทำแฮชกุญแจ และเก็บค่าของกุญแจตามตำแหน่งที่แฮชได้ ถ้าเราไปเปลี่ยนกุญแจ เมื่อทำแฮช มันจะวิ่งไปหาที่ตำแหน่งอื่น ในกรณีนั้น เราอาจจะได้ข้อมูลสองที่สำหรับกุญแจเดียวกัน หรือเราอาจจะไม่สามารถหากุญแจได้เลย ไม่ว่าอย่างไร ดิกชันนารีจะทำงานผิดพลาด

นั่นจึงเป็นเหตุผลที่กุญแจต่าง ๆ ต้องสามารถทำแฮชได้ และทำไมชนิดข้อมูลที่เปลี่ยนแปลงได้ เช่น ลิสต์ถึงไม่สามารถทำแฮชได้ วิธีง่ายที่สุดที่จะก้าวข้ามข้อจำกัดนี้ คือการใช้ ทูเพิล (tuples) ที่เราจะได้เห็นในบทต่อไป

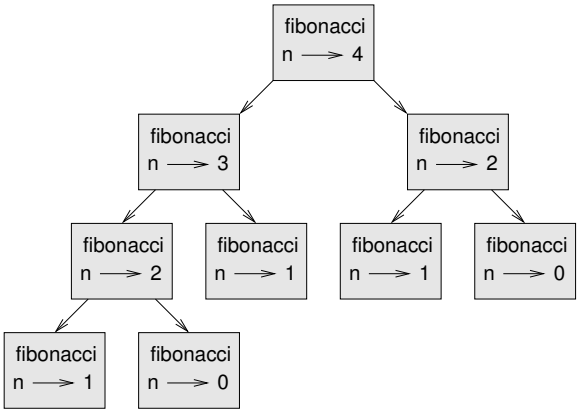
เนื่องจาก ตัวดิกชันนารีเองก็เป็นข้อมูลที่สามารถเปลี่ยนแปลงได้ มันจึงไม่สามารถใช้เป็นกุญแจได้ แต่มัน*สามารถ*ใช้เป็นค่าได้

11.6. เมโม

ตอนที่เราลองเล่นกับฟังก์ชัน `fibonacci` ในหัวข้อ 6.7 สังเกตว่า ยิ่งเราใส่ค่าอาร์กิวเมนต์ใหญ่เท่าไร ฟังก์ชันยิ่งใช้เวลารันนานเท่านั้น นอกจากนั้น สังเกตว่าเวลารันเพิ่มขึ้นเร็วมาก

เพื่อดูว่าทำไมจึงเป็นเช่นนั้น ดูรูป 11.2 ที่แสดง **กราฟการเรียกใช้** (**call graph**) สำหรับ `fibonacci` ที่ใช้ **n=4**:

กราฟการเรียกใช้แสดงฟังก์ชัน พร้อมลูกศรเชื่อมฟังก์ชันที่เรียกกับฟังก์ชันที่ถูกเรียก บนสุดของกราฟ `fibonacci` กับ **n=4** เรียก `fibonacci` กับ **n=3** และกับ **n=2** ในทำนองเดียวกัน `fibonacci` กับ **n=3** เรียก `fibonacci` กับ **n=2** และกับ **n=1** และต่อ ๆ ไปแบบเดียวกัน



รูปที่ 11.2.: กราฟการเรียกใช้ (Call graph)

ถ้านับจำนวนครั้งที่ `fibonacci(0)` และ `fibonacci(1)` ถูกเรียกใช้ จะพบว่า นี่เป็นวิธีที่ไม่มีประสิทธิภาพ และมันจะยิ่งแย่ถ้าค่าอาร์กิวเมนต์ใหญ่ขึ้น

วิธีแก้ปัญหา คือ เก็บค่าที่ได้คำนวณแล้วไว้ในดิกชันนารี ค่าที่ได้เคยคำนวณไว้แล้วและเก็บไว้ใช้ภายหลังจะเรียกว่า **เมโม** (memo) ข้างล่างนี้คือเวอร์ชั่นที่ทำเมโมของ `fibonacci`:

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

ตัวแปร `known` เป็นดิกชันนารีที่เก็บค่าของเลขฟีโบนัชชี ที่ได้คำนวณไว้แล้ว มันเริ่มจากสองรายการ: กุญแจ 0 ไปหาค่า 0 และกุญแจ 1 ไปหาค่า 1

เมื่อไรที่ `fibonacci` ถูกเรียก มันจะดูค่าใน `known` ถ้าผลที่ต้องการมีอยู่แล้วในตัวแปร `known` `fibonacci` สามารถส่งคืนค่าออกนั้นออกมาได้เลย หรือถ้าผลที่ต้องการยังไม่มี ก็คำนวณผลออกมาและเก็บไว้ในดิกชันนารี แล้วค่อยส่งคืนค่าออกนั้นออกมา

ถ้าลองรันเวอร์ชั่นนี้ของ `fibonacci` และเปรียบเทียบกับเวอร์ชั่นเดิม จะพบว่า มันเร็วกว่ามาก

11.7. ตัวแปรส่วนกลาง

ตัวอย่างที่แล้ว ตัวแปร known ถูกสร้างอยู่นอกฟังก์ชัน ดังนั้นมันจะอยู่ภายใต้ขอบเขตพิเศษ เรียกว่า __main__ ตัวแปรต่าง ๆ ที่อยู่ภายใต้ขอบเขต __main__ บางครั้งจะเรียกว่า **เป็นส่วนกลาง** (**global**) เพราะว่าตัวแปรเหล่านี้สามารถถูกเข้าถึงได้จากทุก ๆ ฟังก์ชัน ต่างจากตัวแปรเฉพาะที่ ซึ่งจะหายไปเมื่อฟังก์ชันจบ ตัวแปรส่วนกลางจะคงอยู่ได้ ผ่านการเรียกใช้แต่ละครั้ง

ตัวแปรส่วนกลางนิยมใช้สำหรับเป็น **ตัวบ่งชี้** (**flags**) นั่นคือ ตัวแปรบูลีนต่าง ๆ ที่ใช้บอกสถานะ (แบบเดียวกับ ธง) ว่าสถานะเป็นจริงหรือเปล่า ตัวอย่างเช่น บางโปรแกรมใช้ตัวบ่งชี้ ชื่อ verbose เพื่อใช้บอกระดับความละเอียดของเอาต์พุต:

```
verbose = True

def example1():
    if verbose:
        print('Running example1')
```

ถ้าลองกำหนดค่าใหม่ให้กับตัวแปรส่วนกลางดู เราจะฉงนได้ ตัวอย่างต่อไปนี้พยายามที่จะเก็บบันทึกว่าฟังก์ชันถูกเรียกใช้แล้วหรือไม่:

```
been_called = False

def example2():
    been_called = True          # WRONG
```

พอเรารันโปรแกรมนี้ดู เราจะเห็นว่าค่า been_called ไม่ได้เปลี่ยนไปเลย ปัญหาคือ example2 สร้างตัวแปรเฉพาะที่ใหม่ขึ้นมา ชื่อ been_called ตัวแปรเฉพาะที่หายไป ตอนฟังก์ชันจบ และไม่ได้มีผลอะไรกับตัวแปรส่วนกลาง

ถ้าจะกำหนดค่าใหม่ให้กับตัวแปรส่วนกลางจากภายในฟังก์ชัน เราต้อง**ประกาศ**ตัวแปรส่วนกลาง ก่อนที่จะใช้มัน:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

คำสั่ง `global` บอกอินเตอร์พรีเตอร์ ทำนองว่า “ในฟังก์ชันนี้ ถ้าเราพูดว่า `been_called` เราหมายถึงตัวแปรส่วนกลาง ไม่ต้องสร้างตัวแปรเฉพาะที่ขึ้นมาใหม่”

นี่เป็นตัวอย่างที่พยายามกำหนดค่าให้กับตัวแปรส่วนกลาง:

```
count = 0

def example3():
    count = count + 1           # WRONG
```

ถ้ารันไป เราจะได้:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

ไพธอนจะคิดว่า `count` เป็นตัวแปรเฉพาะที่ และเข้าใจว่า เราพยายามจะใช้ค่าของมัน ก่อนกำหนดค่าให้มัน วิธีแก้ไขก็แบบเดิม คือ ประกาศ `count` เป็นตัวแปรส่วนกลาง

```
def example3():
    global count
    count += 1
```

ถ้าตัวแปรส่วนกลางอ้างถึง ค่าข้อมูลที่สามารถเปลี่ยนแปลงได้ (mutable value) เราสามารถแก้ไขค่านั้นได้เลย โดยไม่ต้องประกาศตัวแปร:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

ดังนั้น เราสามารถเพิ่ม ลบ แก้ไข ค่าของลิสต์ส่วนกลาง หรือดิกชันนารีส่วนกลางได้ แต่ถ้าเราต้องการกำหนดค่าข้อมูลใหม่ให้กับตัวแปร เราต้องประกาศให้ชัดเจน:

```
def example5():
    global known
    known = dict()
```

ตัวแปรส่วนกลางนั้นนับว่ามีประโยชน์อยู่ แต่ถ้าเรามีตัวแปรส่วนกลางมากเกินไป และเราแก้ไขค่าข้อมูลของมันบ่อยเกินไป มันก็อาจจะทำให้โปรแกรมดีบักได้ยาก

11.8. การดีบัก

ตอนที่เราทำงานกับชุดข้อมูลที่ใหญ่ ๆ มันอาจจะไม่ค่อยสะดวก ที่จะดีบักด้วยการพิมพ์ออกหน้าจอ และค่อย ๆ ไล่ตรวจสอบผลลัพธ์ด้วยตา ข้างล่างนี้เป็นคำแนะนำสำหรับการดีบักชุดข้อมูลขนาดใหญ่:

ปรับขนาดของอินพุตลง: ถ้าทำได้ ให้ลดขนาดของชุดข้อมูลลง ตัวอย่างเช่น ถ้าโปรแกรมอ่านไฟล์ข้อความ ให้เริ่มจาก 10 บรรทัดแรกก่อน หรือเริ่มจากตัวอย่างบางส่วนที่เล็กที่สุดที่จะหาได้ก่อน อาจจะเข้าไปแก้ไขไฟล์โดยตรงเลย หรือ(ดีกว่า) อาจจะเข้าไปแก้ไขโปรแกรม ให้มันอ่านเฉพาะ **n** บรรทัดแรก ๆ ก่อน

ถ้ามีข้อผิดพลาดอยู่ อาจจะลองลด **n** ลงให้เป็นค่าเล็กที่สุด เท่าที่ยังมองเห็นปัญหาได้อยู่ แล้วค่อยเพิ่มมันทีละขั้น ๆ ตอนที่เจอและแก้ไขปัญหาได้

ตรวจสอบสรุปและชนิดข้อมูล: แทนที่จะพิมพ์ข้อมูลทั้งหมดออกหน้าจอ และไล่ตรวจสอบรายละเอียดทั้งหมดด้วยตา อาจจะลองพิมพ์เฉพาะสรุปข้อมูลออกหน้าจอ: ตัวอย่างเช่น จำนวนรายการในดิกชันนารี หรือจำนวนรายการของลิสต์

สาเหตุหนึ่งที่พบบ่อยมาก ๆ สำหรับข้อผิดพลาดเวลาดำเนินการ (runtime errors) คือ ค่าข้อมูลไม่ใช่ชนิดข้อมูลที่ถูกต้อง การดีบักข้อผิดพลาดแบบนี้ ก็แค่พิมพ์ชนิดของค่าข้อมูลออกมาหน้าจอเท่านั้น

เขียนโปรแกรมให้มีการตรวจสอบตัวเอง: บางครั้ง เราอาจจะเขียนโปรแกรมให้ตรวจสอบข้อผิดพลาดโดยอัตโนมัติไว้เลย ตัวอย่างเช่น ถ้าเราคำนวณค่าเฉลี่ยของลิสต์ของตัวเลข เราอาจจะตรวจสอบว่า ผลลัพธ์ที่ได้ไม่ใหญ่กว่าค่าในลิสต์ที่ใหญ่ที่สุด และผลลัพธ์ที่ได้ไม่เล็กกว่าค่าในลิสต์ที่เล็กที่สุด การทำแบบนี้ เขาเรียกว่า “เซนิตี้เช็ค” (sanity check) หรือตรวจสอบว่ายังไม่บ้า เพราะว่าโปรแกรมมันตรวจหาผลที่มันดู“บ้า ๆ” (insane)

การตรวจสอบอีกแบบหนึ่งที่เปรียบเทียบผลการคำนวณสองวิธีที่แตกต่างกัน เพื่อดูว่าผลที่ได้สอดคล้องกันหรือไม่ แบบนี้จะเรียกว่า “การตรวจสอบความสอดคล้อง” (consistency check)

จัดรูปแบบของเอาต์พุต: การจัดรูปแบบของผลแสดงจากการดีบัก จะช่วยให้มองเห็นข้อผิดพลาดได้ง่ายขึ้น เราได้เห็นตัวอย่างไปแล้วในหัวข้อ 6.9 เครื่องมืออีกตัวที่อาจจะมีประโยชน์ คือ โมดูล `pprint` ที่มีฟังก์ชัน `pprint` ซึ่งสามารถแสดงชนิดข้อมูลสำเร็จรูป (built-in types) ในรูปแบบที่ช่วยให้มนุษย์อ่านได้ง่ายขึ้น (`pprint` ย่อจาก “pretty print”)

ย้ำอีกครั้ง เวลาที่เราทำโครงโปรแกรมสามารถช่วยลดเวลาที่เราให้ดีบักโปรแกรม

11.9. อภิธานศัพท์

การแปลง (mapping): ความสัมพันธ์ ที่แต่ละรายการในเซ็ตหนึ่ง คู่กันกับรายการในอีกเซ็ตหนึ่ง

ดิกชันนารี (dictionary): การแปลงจากกุญแจ (keys) ไปหาค่าของมัน (values)

คู่กุญแจค่า (key-value pair): คู่ของการแปลงจากกุญแจหนึ่งไปหาค่าของมัน

ค่ารายการ (item): สำหรับดิกชันนารี ค่ารายการ หมายถึง คู่กุญแจค่า

กุญแจ (key): เป็นสิ่งที่อยู่ในดิกชันนารี ที่ปรากฏเป็นส่วนแรกของคู่กุญแจค่า

ค่า (value): เป็นสิ่งที่อยู่ในดิกชันนารี ที่ปรากฏเป็นส่วนหลังของคู่กุญแจค่า ความหมายนี้จะเจาะจงมากกว่าความหมายทั่วไปของคำว่า “ค่า” (value)

อิมพลิเมนเตชั่น (implementation): เป็นวิธีที่ใช้ทำการคำนวณ

ตารางแฮช (hashtable): อัลกอริธึมที่ใช้ทำการคำนวณสำหรับดิกชันนารี

ฟังก์ชันแฮช (hash function): ฟังก์ชันที่ใช้ตารางแฮช เพื่อคำนวณตำแหน่งของกุญแจ

สามารถทำแฮชได้ (hashable): ชนิดข้อมูลที่มีฟังก์ชันแฮชอยู่ ชนิดข้อมูลที่เปลี่ยนแปลงไม่ได้ (immutable types) เช่น จำนวนเต็ม (integers) เลขทศนิยม (floats) และ สายอักขระ (strings) สามารถทำแฮชได้ ชนิดข้อมูลที่เปลี่ยนแปลงได้ (mutable types) เช่น ลิสต์ (lists) และ ดิกชันนารี (dictionaries) ไม่สามารถทำแฮชได้

การเทียบค้น (lookup): ปฏิบัติการของดิกชันนารีที่รับกุญแจเข้าไป และค้นหาค่าของกุญแจนั้นออกมา

การเทียบค้นย้อนกลับ (reverse lookup): ปฏิบัติการของดิกชันนารีที่รับค่าเข้าไป และค้นหากุญแจที่เชื่อมกับค่านั้นออกมา

คำสั่งเรส (raise statement): คำสั่งที่(ตั้งใจ)ส่งสัญญาณเอ็กเซ็ปชั่นออกมา

เซตโทน (singleton): ลิสต์ (หรือข้อมูลแบบลำดับชนิดอื่น) ที่มีอิลิเมนต์เดียว

กราฟการเรียกใช้ (call graph): แผนภาพที่แสดงทุก ๆ กรอบแทนฟังก์ชัน แต่ละฟังก์ชันที่สร้างขึ้นมาระหว่างที่รันโปรแกรม โดยมีลูกศรจากโปรแกรมที่เรียกฟังก์ชัน (caller) ไปหาฟังก์ชันที่ถูกเรียก (callee)

เมโม (memo): ค่าที่ได้คำนวณได้แล้ว และเก็บไว้เพื่อใช้ภายหลัง เพื่อไม่คำนวณซ้ำอีกในอนาคต

ตัวแปรส่วนกลาง (global variable): ตัวแปรที่นิยามอยู่นอกฟังก์ชัน ตัวแปรส่วนกลางสามารถถูกเรียกใช้ได้จากทุกฟังก์ชัน

คำสั่งกำหนดตัวแปรส่วนกลาง (global statement): คำสั่งที่กำหนดให้ตัวแปรเป็นตัวแปรส่วนกลาง

ตัวบ่งชี้ (flag): ตัวแปรบูลีนที่ใช้ระบุว่าเงื่อนไขเป็นจริงหรือไม่

การประกาศ (declaration): คำสั่ง เช่นพวก `global` ที่บอกอินเตอร์พรีเตอร์บางอย่างเกี่ยวกับตัวแปร

11.10. แบบฝึกหัด

***แบบฝึกหัด* 11.1.** *เขียนฟังก์ชันที่อ่านคำใน* `words.txt` *และเก็บคำเหล่านั้นเป็นกุญแจของดิกชันนารี มันไม่สำคัญว่าค่าของกุญแจเป็นเท่าไร เราสามารถใช้ตัวปฏิบัติการ* `in` *เป็นวิธีที่รวดเร็วในการตรวจสอบว่าคำที่สนใจมีอยู่ในดิกชันนารีหรือไม่*

ถ้าทำแบบฝึกหัด 10.10มาแล้ว เราสามารถเปรียบเทียบความเร็วของอิมพลิเมนเตชั่นนี้ กับลิสต์ที่ใช้ `in` *และวิธีค้นหาแบบแบ่งสอง (bisection search)*

***แบบฝึกหัด* 11.2.** *อ่านเอกสารประกอบของดิกชันนารีเมธอด* `setdefault` *และใช้เมธอดนี้เพื่อเขียนโปรแกรมในแบบที่กระชับขึ้นของ* `invert_dict` *เฉลย:* `http://thinkpython2.com/code/invert_dict.py`

***แบบฝึกหัด* 11.3.** *ใช้เมโม เพื่อทำฟังก์ชันแอกเคอมันน์ (Ackermann function) จากแบบฝึกหัด 6.2 และดูว่า การใช้เมโม ช่วยให้สามารถรันฟังก์ชัน เมื่อใช้กับอาร์กิวเมนต์ที่ใหญ่ขึ้นได้หรือไม่ คำใบ้: ไม่ได้ เฉลย:* `http://thinkpython2.com/code/ackermann_memo.py`

***แบบฝึกหัด* 11.4.** *จากแบบฝึกหัด 10.7 เราจะมีฟังก์ชันชื่อ* `has_duplicates` *ที่รับลิสต์เป็นพารามิเตอร์ และส่งคืนค่า* `True` *ออกมา ถ้ามีรายการที่ซ้ำอยู่ในลิสต์*

ใช้ดิกชันนารีเพื่อเขียนโปรแกรมแบบที่เร็วขึ้นและง่ายขึ้นของฟังก์ชัน `has_duplicates` *เฉลย:* `http://thinkpython2.com/code/has_duplicates.py`.

***แบบฝึกหัด* 11.5.** *คำสองคำเรียกว่าเป็น "คู่หมุนเปลี่ยน" (rotate pair) เมื่อเราหมุนคำหนึ่งและผลลัพธ์ได้เป็นอีกคำหนึ่ง (ดู* `rotate_word` *ใน แบบฝึกหัด 8.5)*

เขียนโปรแกรมที่อ่านลิสต์ของคำ และหาคู่หมุนเปลี่ยนออกมาทั้งหมด เฉลย: `http://thinkpython2.com/code/rotate_pairs.py`

แบบฝึกหัด **11.6.** *นี่เป็นปริศนาจาก* Car Talk (`http://www.cartalk.com/content/puzzlers`)*:*

> *เรื่องนี้ส่งมาจากสหายชื่อว่า แดน โอเลียรี่ แดนฉงนกับคำพยางค์เดียวห้าอักษร ที่มีคุณสมบัติพิเศษ นั่นคือ เมื่อเราเอาอักษรแรกออก ตัวอักษรที่เหลือจะเป็นคำพ้องเสียง (homophone) ของคำเดิม นั่นคือคำใหม่จะออกเสียงเหมือนเดิมเลย แถมถ้าไม่เอาอักษรตัวแรกออก แต่เอาอักษรตัวที่สองออก ผลก็ยังเป็นคำพ้องเสียงเดิมอยู่ คำถามคือคำนั้นคือคำอะไร*
>
> *เดี๋ยวผมจะให้ตัวอย่างคำที่ไม่ใช่ ลองคำว่า 'wrack' W-R-A-C-K ก็แบบที่รู้ เช่น 'wrack with pain.' ถ้าผมเอาอักษรตัวแรกออก ผมจะเหลือแค่ 'R-A-C-K' เหมือนที่คำพูด 'Holy cow, did you see the rack on that buck!' เกือบได้แล้ว เพียงแต่เมื่อเราใส่ 'w' กลับเข้าไป แต่เอา 'r' ออกแทน เราจะได้ 'wack' ซึ่งไม่ได้พ้องเสียงกับอีกสองคำ ('wrack' และ 'rack')*
>
> *แต่มันมีอย่างน้อยก็คำหนึ่งหละ ที่แดนและเรารู้ว่าจะให้คำพ้องเสียงสองคำออกมา ไม่ว่าจะเอาอักษรตัวแรก หรือตัวที่สองออก คำถามคือคำนั้นคือคำว่าอะไร*

เราสามารถใช้ดิกชันนารีจากแบบฝึกหัด 11.1 เพื่อตรวจสอบว่าสายอักขระอยู่ในลิสต์ของคำหรือไม่ เพื่อตรวจสอบว่าคำสองคำเป็นคำพ้องเสียงหรือไม่ เราสามารถใช้พจนานุกรมการออกเสียงซีเอ็มยู (CMU Pronouncing Dictionary) ซึ่งสามารถดาวน์โหลดได้จาก `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` *หรือจาก* `http://thinkpython2.com/code/c06d` *และยังสามารถดาวน์โหลด* `http://thinkpython2.com/code/pronounce.py` *ที่เตรียมฟังก์ชัน* `read_dictionary` *ที่ใช้อ่านพจนานุกรมการออกเสียง แล้วส่งไพธอนดิกชันนารีที่เชื่อมคำกับการออกเสียงออกมาให้*

เขียนโปรแกรมที่หาคำทั้งหมดที่ตอบปริศนานี้ออกมา เฉลย: `http://thinkpython2.com/code/homophone.py`

12. ทูเพิล

บทนี้นำเสนอชนิดข้อมูลสำเร็จรูป อีกชนิดหนึ่ง ซึ่งคือ ทูเพิล พร้อมแสดงวิธีการใช้ ลิสต์ ดิกชันนารี และทูเพิล ทำงานด้วยกัน นอกจากนั้น เรายังอภิปรายกันถึงคุณลักษณะที่มีประโยชน์ของอาร์กิวเมนต์ความยาวแปรผัน พร้อมตัวดำเนินการ*การรวบรวม*และ*การกระจาย*

เรื่องหนึ่ง: ยังไม่มีข้อตกลงอย่างเป็นทางการว่า ควรจะอ่าน “tuple” อย่างไร บางคนอ่าน “ทับ-เพิล” ที่คล้องกับ “supple”. แต่ในบริบทของการเขียนโปรแกรม คนส่วนใหญ่อ่าน “ทู-เพิล” ที่คล้องกับ “quadruple”

12.1. ทูเพิลไม่สามารถเปลี่ยนแปลงได้

ทูเพิลเป็นลำดับของค่าต่าง ๆ ค่าต่าง ๆ เป็นข้อมูลชนิดใดก็ได้ และค่าต่าง ๆ เหล่านั้นอ้างอิงได้ด้วยดัชนีที่เป็นเลขจำนวนเต็ม ซึ่งในแง่นี้ ทูเพิลจะคล้าย ๆ กับลิสต์ ความต่างที่สำคัญเลย คือ ทูเพิลไม่สามารถเปลี่ยนแปลงแก้ไขได้

ในแง่ไวยากรณ์ ทูเพิล เป็นลำดับของค่าต่าง ๆ ที่คั่นระหว่างกันด้วยเครื่องหมายจุลภาค (‘,’) เช่น

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

นอกจากนั้น ทูเพิลมักจะอยู่ระหว่างเครื่องหมายวงเล็บ ซึ่งจริง ๆ แล้ว มันไม่ได้จำเป็นเลย ที่จะต้องใช้เครื่องหมายวงเล็บ เช่น

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

เพื่อสร้างทูเพิลที่มีอิลิเมนต์เดียว เราจำเป็นถ้าใส่เครื่องจุลภาคต่อท้ายเข้าไป เช่น

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

ค่าในวงเล็บไม่ใช่ทูเพิล:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

อีกวิธีที่จะสร้างทูเพิล คือ การใช้ฟังก์ชันสำเร็จรูป `tuple` ถ้าไม่ใส่อาร์กิวเมนต์ มันจะสร้างทูเพิลว่างให้:

```
>>> t = tuple()
>>> t
()
```

ถ้าอาร์กิวเมนต์ของฟังก์ชัน เป็นลำดับ (ไม่ว่าจะเป็น สายอักขระ ลิสต์ หรือทูเพิล) ผลที่ได้จะเป็น ทูเพิลของอิลิเมนต์ของลำดับนั้น:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

เพราะว่า `tuple` เป็นชื่อของฟังก์ชันสำเร็จรูป เราจึงไม่ควรใช้มันเป็นชื่อตัวแปร

ตัวดำเนินการต่าง ๆ ที่ใช้กับลิสต์ได้ ส่วนใหญ่ก็ใช้กับทูเพิลได้ ตัวดำเนินการวงเล็บสี่เหลี่ยมเลือกอิลิเมนต์ตามดัชนี:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

และตัวดำเนินการตัด (slice operator) เลือกขอบเขตของอิลิเมนต์ เช่น

```
>>> t[1:3]
('b', 'c')
```

แต่ถ้าเราลองแก้ไขค่าอิลิเมนต์ของทูเพิล เราจะเห็นข้อผิดพลาดแจ้งออกมา:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

เพราะว่า ทูเพิลไม่สามารถเปลี่ยนแปลงได้ (immutable) เราจึงไม่สามารถที่จะแก้ไข หรือเปลี่ยนแปลงอิลิเมนต์ของทูเพิลได้ แต่เราสามารถแทนที่ทูเพิล ด้วยทูเพิลใหม่ได้:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

คำสั่งนี้สร้างทูเพิลใหม่ขึ้นมา และกำหนดให้ตัวแปร **t** ไปอ้างถึงมัน

ตัวดำเนินการเชิงสัมพันธ์ทำงานกับทูเพิลและข้อมูลแบบลำดับชนิดอื่น ๆ ได้ ไพธอนทำงานโดยเริ่มจากเปรียบเทียบอิลิเมนต์แรกจากแต่ละทูเพิล ถ้าอิลิเมนต์จากสองทูเพิลเท่ากัน ไพธอนจะไม่สนใจ และขยับไปตรวจอิลิเมนต์ในลำดับถัดไปจาอทั้งสองทูเพิล ถ้ามันต่างกัน ไพธอนจะทำงานเปรียบเทียบ และใช้ผลการเปรียบเทียบของอิลิเมนต์คู่นี้ เป็นผลของตัวดำเนินการ โดยอิลิเมนต์ที่เหลือจะไม่นำมาคิดเลย

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> (0, 1, 1, 8, 10, 100) < (0, 1, 1, 9, 1, 0)
True
>>> (0, 1, 1, 11, 10, 100) < (0, 1, 1, 9, 1, 0)
False
```

คำสั่งแรก คู่แรกที่ต่างกันคือดัชนีที่ 1 และ อิลิเมนต์ในทูเพิลแรกมีค่าน้อยกว่าอิลิเมนต์ในทูเพิลสอง (ค่า 1 เปรียบเทียบกับ 3) จึงให้ผลออกมาเป็น **True** คำสั่งสอง ก็เช่นเดียวกัน เมื่อได้ผลจากคู่ดัชนีที่ 1 (ค่า 1 เปรียบเทียบกับ 3) แล้ว คู่ที่เหลือจะไม่ถูกคำมาคิด นั่นคือ คู่ดัชนี 2 (ค่า 2000000 เปรียบเทียบกับ 4) ไม่ได้ถูกนำมาคิด คำสั่งสาม คู่แรกที่ต่างกันคือดัชนีที่ 3 (เปรียบเทียบ ค่า 8 กับ 9) ผลของการเปรียบเทียบคู่นี้คือ ผลลัพธ์ คู่ที่เหลือจะไม่ถูกนำมาคิด คำสั่งสี่ก็เช่นเดียวกัน เมื่อได้ผลจากคู่ดัชนี 2 (ค่า 11 เปรียบเทียบกับ 9) แล้ว คู่ที่เหลือจะไม่ถูกนำมาคิด

12.2. การกำหนดค่าทูเพิล

บางครั้ง เราอาจต้องการสลับค่าระหว่างตัวแปรสองตัว ด้วยวิธีการกำหนดค่าแบบทั่ว ๆ ไป เราต้องใช้ตัวแปรชั่วคราวเข้ามาช่วย ตัวอย่างเช่น สลับค่าระหว่างตัวแปร **a** และตัวแปร **b**:

```
>>> temp = a
>>> a = b
>>> b = temp
```

วิธีนี้ค่อนข้างงุ่มง่าม วิธี**การกำหนดค่าทูเพิล** (**tuple assignment**) ดูสละสลวยกว่าเยอะ:

```
>>> a, b = b, a
```

ฝั่งซ้ายมือเป็นทูเพิลของตัวแปร ส่วนฝั่งขวามือเป็นทูเพิลของนิพจน์ แต่ละค่าจะถูกกำหนดไปให้กับตัวแปรตามลำดับ นิพจน์ทั้งหมดทางขวามือจะถูกประเมินค่า ก่อนดำเนินการกำหนดค่า

จำนวนตัวแปรทางซ้ายมือ และจำนวนของค่าต่าง ๆ ทางขวามือ จะต้องเท่ากัน:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

จริง ๆ แล้ว ทางขวามือจะเป็นข้อมูลแบบลำดับชนิดใด ๆ ก็ได้ (สายอักขระ ลิสต์ หรือทูเพิล) ตัวอย่าง เช่น เพื่อจะแยกอีเมล์แอดเดรส (email address) ออกมาเป็น ชื่อผู้ใช้ และโดเมน (domain) เราอาจจะเขียน:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

ค่าที่ส่งคืนออกมาจาก `split` เป็นลิสต์ของสองอิลิเมนต์ โดย อิลิเมนต์แรกจะถูกกำหนดให้กับ `uname` และอิลิเมนต์ที่สองให้กับ `domain`

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3. ทูเพิลที่ใช้เป็นค่าที่ส่งคืนออกมา

ถ้าพูดกันตรง ๆ ฟังก์ชันสามารถส่งค่าออกมาได้ค่าเดียว แต่ถ้าค่าที่ส่งออกมาเป็นทูเพิล ผลของมันจึงเหมือนกับการส่งค่าออกมาหลาย ๆ ค่า ตัวอย่างเช่น ถ้าเราต้องการหารเลขจำนวนเต็ม และคำนวณทั้ง ผลหาร (quotient) และเศษ (remainder) มันก็ไม่ค่อยมีประสิทธิภาพ ถ้าต้องคำนวณ `x/y` แล้วก็ `x%y`. มันดีกว่าที่จะคำนวณทั้งสองอย่างพร้อม ๆ กัน

ฟังก์ชันสำเร็จรูป `divmod` รับอาร์กิวเมนต์สองตัว และให้ทูเพิลที่มีสองค่าออกมา ซึ่งคือ ผลหารและเศษ เราสามารถเก็บค่าผลลัพธ์นี้เป็นทูเพิลได้:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

หรือ ใช้การกำหนดค่าทูเพิล เพื่อเก็บแต่ละอิลิเมนต์แยกกัน:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

นี่เป็นตัวอย่างของฟังก์ชันที่ให้ค่าทูเพิลออกมา:

```
def min_max(t):
    return min(t), max(t)
```

`max` และ `min` เป็นฟังก์ชันสำเร็จรูป ที่ใช้หาค่ามากที่สุดและน้อยที่สุด ในข้อมูลแบบลำดับ `min_max` คำนวณค่าทั้งสอง และให้ทูเพิลของค่าทั้งสองออกมา

12.4. ทูเพิลที่ใช้เป็นอาร์กิวเมนต์ความยาวแปรผัน

เราสามารถสร้างฟังก์ชันที่รับอาร์กิวเมนต์ที่มีจำนวนแปรเปลี่ยนได้ ชื่อพารามิเตอร์ที่ขึ้นต้นด้วย `*` จะ**รวบรวม** (**gather**) อาร์กิวเมนต์หลาย ๆ ตัวเข้าเป็นทูเพิล ตัวอย่างเช่น `printall` รับอาร์กิวเมนต์กี่ตัวก็ได้ และก็พิมพ์ค่าเหล่านั้นออกมา:

```
def printall(*args):
    print(args)
```

พารามิเตอร์รวม `args` สามารถจะตั้งเป็นชื่ออะไรก็ได้ แต่มักจะนิยมตั้งเป็นชื่อ `args` ฟังก์ชันทำงานดังนี้:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

ตรงข้ามกับการรวบรวมคือ **การแยกกระจาย** (**scatter**) ถ้าเรามีค่าหลาย ๆ ค่า และต้องการส่งเข้าไปให้ฟังก์ชันผ่านอาร์กิวเมนต์หลาย ๆ ตัว เราก็สามารถใช้ตัวดำเนินการ `*` ได้ ตัวอย่าง `divmod` รับอาร์กิวเมนต์สองตัว ต้องการสองตัว และสองตัวเท่านั้น ไม่ได้ใช้ทูเพิล:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

แต่เราสามารถแยกกระจายทูเพิลได้:

```
>>> divmod(*t)
(2, 1)
```

ฟังก์ชันสำเร็จรูปหลาย ๆ อันใช้ทูเพิลเป็นอาร์กิวเมนต์ความยาวแปรผัน ตัวอย่างเช่น `max` และ `min` สามารถรับอาร์กิวเมนต์กี่ตัวก็ได้:

```
>>> max(1, 2, 3)
3
```

แต่ `sum` ทำไม่ได้

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

เพื่อเป็นแบบฝึกหัด ให้ลองเขียนฟังก์ชัน ชื่อว่า `sumall` ที่รับอาร์กิวเมนต์กี่ตัวก็ได้ และส่งคืนค่าผลรวมออกมา

12.5. ลิสต์และทูเพิล

`zip` เป็นฟังก์ชันสำเร็จรูปที่รับข้อมูลแบบลำดับตั้งแต่สองชุดขึ้นไป และส่งคืนลิสต์ของทูเพิลออกมา โดยที่แต่ละทูเพิล จะมีอิลิเมนต์มาจากแต่ละชุดข้อมูลแบบลำดับ ชื่อของฟังก์ชันมาจาก ซิป(ซิปกางเกง) ที่รูดแล้วมันจะขบฟันของซิปจากสองฝั่งเข้าด้วยกัน

ตัวอย่างนี้ซิปข้อมูลสายอักขระและลิสต์เข้าด้วยกัน

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

ผลลัพธ์จากฟังก์ชัน `zip` จะเป็น **ซิปอ็อบเจกต์** (zip object) ที่สามารถวนเข้าถึงแต่ละคู่ทูเพิลได้ วิธีที่นิยมใช้กับ `zip` ที่สุดคือ ใช้กับลูป `for` เช่น:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

ซิปอ็อบเจกต์ เป็น **ตัววนซ้ำ** (iterator) ซึ่งเป็นอ็อบเจกต์ที่สามารถทำวนซ้ำกับลำดับได้ ตัววนซ้ำ จะคล้าย ๆ กับลิสต์ ในหลาย ๆ แง่ แต่ต่างจากลิสต์ คือ เราไม่สามารถใช้ดัชนีเลือกอิลิเมนต์ออกมาจากตัววนซ้ำได้

ถ้าต้องการใช้ตัวดำเนินการของลิสต์จริง ๆ เราก็สามารถแปลงซิปอ็อบเจกต์ไปเป็นลิสต์ได้:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

ผลลัพธ์จะเป็นลิสต์ของทูเพิล ในตัวอย่างนี้ แต่ละทูเพิลจะมีอักขระจากสายอักขระ `s` และอิลิเมนต์จากลิสต์ `t`

ถ้าลำดับข้อมูลมีความยาวไม่เท่ากัน ผลลัพธ์จะมีความยาวตามลำดับที่สั้นกว่า

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

เราสามารถใช้การกำหนดค่าทูเพิลกับ `for` ลูป เพื่อท่องสำรวจลิสต์ของทูเพิลได้:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

แต่ละครั้งที่ทำลูป ไพธอนเลือกทูเพิลต่อไปในลิสต์ และกำหนดค่าอิลิเมนต์ให้ `letter` และ `number` ผลลัพธ์ของลูปนี้คือ:

```
0 a
1 b
2 c
```

ถ้าเรารวม `zip`, `for`, และการกำหนดค่าทูเพิล เราจะได้เทคนิค ที่สามารถท่องสำรวจลำดับข้อมูลสองลำดับ (หรือมากกว่า) ได้พร้อม ๆ กัน ตัวอย่าง `has_match` รับสองลำดับข้อมูล `t1` และ `t2` และส่งคืนค่า `True` ถ้ามีดัชนี `i` ที่ `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

ถ้าเราอยากท่องสำรวจอิลิเมนต์ต่าง ๆ ในลำดับข้อมูล พร้อมดัชนีด้วย เราสามารถใช้ฟังก์ชันสำเร็จรูป `enumerate`:

```
for index, element in enumerate('abc'):
    print(index, element)
```

ผลลัพธ์จาก `enumerate` เป็น*อีนูเมอร์เรตอ็อบเจกต์* (*enumerate object*) ที่สามารถทำวนซ้ำลำดับคู่ โดย แต่ละคู่จะมี ดัชนี(เริ่มด้วย 0) และอิลิเมนต์จากลำดับข้อมูล ตัวอย่างนี้ จะเห็นผลลัพธ์เป็น

```
0 a
1 b
2 c
```

12.6. ดิกชันนารีและทูเพิล

ดิกชันนารีมีเมธอด ชื่อว่า `items` ที่ส่งคืนข้อมูลแบบลำดับของทูเพิลออกมา โดยที่แต่ละทูเพิล คือ คู่กุญแจ ค่า ในดิกชันนารี

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

ผลลัพธ์คือ อ็อบเจกต์ `dict_items` ที่เป็น*ตัววนซ้ำ* สามารถท่องสำรวจคู่กุญแจค่าได้ เราสามารถใช้มัน ใน `for` ลูปได้ดังนี้:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
```

```
a 0
b 1
```

แบบเดียวกับดิกชันนารี อิลิเมนต์ต่าง ๆ จะไม่ได้เรียงลำดับออกมา

มองกลับไปอีกมุมหนึ่ง เราสามารถใช้ลิสต์ของทูเพิล เพื่อกำหนดค่าเริ่มต้นให้กับดิกชันนารีใหม่ได้:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

ใช้ `dict` ร่วมกับ `zip` ช่วยให้ เรามีวิธีสร้างดิกชันนารีที่สะดวกมาก:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

เมธอด `update` ของดิกชันนารี ก็สามารถรับลิสต์ของทูเพิล และเพิ่มคู่กุญแจค่าใหม่เข้าไปในดิกชันนารีได้

เราอาจจะเห็น ทูเพิลถูกใช้เป็น*กุญแจ*ในดิกชันนารีอยู่บ่อย ๆ (เพราะว่าเราใช้ลิสต์เป็นกุญแจไม่ได้) ตัวอย่างเช่น สมุดโทรศัพท์จะใช้ ชื่อและนามสกุล ไปบอกเบอร์โทรศัพท์ สมมติว่า เรากำหนด `first`, `last` และ `number` เราอาจจะเขียน:

```
directory[first, last] = number
```

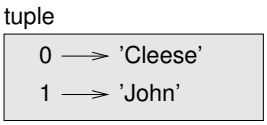
นิพจน์ในวงเล็บสี่เหลี่ยม คือ ทูเพิล เราสามารถใช้*การกำหนดค่าทูเพิล* เพื่อท่องสำรวจดิกชันนารีนี้ได้

```
for first, last in directory:
    print(first, last, directory[first, last])
```

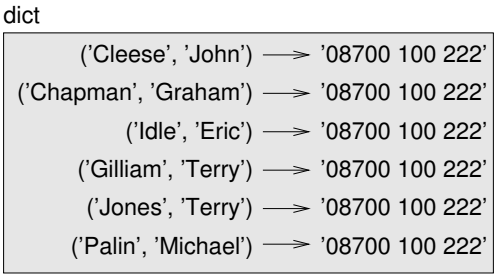
ลูปนี้ท่องสำรวจ*กุญแจ*ใน `directory` โด*ยกุญแจ*เป็นทูเพิล มันกำหนดค่าของอิลิเมนต์ในแต่ละทูเพิลให้กับ `first` กับ `last` และพิมพ์ชื่อ นามสกุล และเบอร์โทรศัพท์ออกมา

มีสองวิธี ที่จะแสดงทูเพิลในแผนภาพสถานะ วิธีที่ละเอียด จะแสดงดัชนีและอิลิเมนต์แบบเดียวกับที่แสดงลิสต์ ตัวอย่างเช่น ทูเพิล `('Cleese', 'John')` ก็จะแสดงแบบในรูปที่ 12.1

แต่ในแผนภาพที่ใหญ่ขึ้น เราไม่ต้องลงรายละเอียดขนาดนั้น ตัวอย่างเช่น แผนภาพของสมุดโทรศัพท์ อาจจะเป็นดังแสดงในรูป 12.2 ทูเพิลในแผนภาพ แสดงง่าย ๆ ด้วยไวยากรณ์ของไพธอน



รูปที่ 12.1.: แผนภาพสถานะ



รูปที่ 12.2.: แผนภาพสถานะ

12.7. ลำดับข้อมูลของลำดับข้อมูล

เราได้ดูกันเรื่อง ลิสต์ของทูเพิล แต่เกือบทั้งหมดของตัวอย่างในบทนี้ สามารถใช้กับ ลิสต์ของลิสต์ ทูเพิลของทูเพิล และทูเพิลของลิสต์ได้ เพื่อไม่ต้องสาธยาย ทุก ๆ คู่ พวกนี้ ว่าไปแล้วบางครั้ง มันก็ง่ายกว่าที่จะอ้างถึงคู่เหล่านี้เป็น ลำดับข้อมูลของลำดับข้อมูล

ในหลาย ๆ บริบท ลำดับข้อมูลหลาย ๆ ชนิด (สายอักขระ ลิสต์ และทูเพิล) สามารถที่จะเลือกใช้อันไหนก็ได้ แล้วเราจะเลือกอย่างไรว่าชนิดไหนดีกว่าชนิดอื่น ๆ

เริ่มจากเรื่องชัด ๆ ก่อน สายอักขระจะค่อนข้างจำกัดกว่าลำดับข้อมูลชนิดอื่น ๆ เพราะว่า อิลิเมนต์ของมัน ต้องเป็นอักขระเท่านั้น นอกจากนั้น สายอักขระ ยังเป็นลำดับข้อมูลชนิดที่ไม่สามารถเปลี่ยนแปลงได้ ถ้าเราต้องการลำดับข้อมูลที่สามารถเปลี่ยนอักขระในลำดับได้ (ซึ่งต่างจากการสร้างสายอักขระใหม่) เราอาจจะเลือกใช้ลิสต์ของอักขระแทน

เราจะเห็น การใช้ลิสต์บ่อยกว่าการใช้ทูเพิล ส่วนใหญ่เพราะว่า ลิสต์ เป็นลำดับข้อมูลชนิดสามารถเปลี่ยนแปลงได้ แต่ก็จะมีบางกรณีที่อาจจะเหมาะกับทูเพิลมากกว่า

1. ในบางบริบท เช่น คำสั่ง `return` มันสะดวกที่จะสร้างทูเพิล มากกว่าสร้างลิสต์ (ไวยากรณ์ของทูเพิลสะดวกกว่า เช่น `return a,b` เปรียบเทียบกับ `return [a,b]`)

2. เราต้องการลำดับข้อมูลเป็นกุญแจของดิกชันนารี เราต้องใช้ลำดับข้อมูลชนิดที่ไม่สามารถ

เปลี่ยนแปลงได้ เช่น ทูเพิล หรือ สายอักขระ

3. ถ้าเราต้องการส่งค่าของลำดับข้อมูล เป็นอาร์กิวเมนต์ให้กับฟังก์ชัน การใช้ทูเพิล จะช่วยลดความเสี่ยงของพฤติกรรมจากการทำสมนาม

เพราะว่า ทูเพิลเป็นลำดับข้อมูลชนิดที่ไม่สามารถเปลี่ยนแปลงได้ ทูเพิลจึงไม่มีพวกเมธอด เช่น `sort` และ `reverse` ซึ่งไปแก้ไขค่าของลิสต์ แต่ไพธอนก็ยังมีฟังก์ชันสำเร็จรูป `sorted` ที่รับลำดับข้อมูล แล้วส่งคืนลิสต์ของอิลิเมนต์นั้น ๆ ที่เรียงลำดับแล้วออกมาให้ และฟังก์ชันสำเร็จรูป `reversed` ที่รับลำดับข้อมูล แล้วส่งคืนตัววนซ้ำที่ใช้ท่องลิสต์ในลำดับถอยหลังออกมาให้

12.8. การดีบัก

ลิสต์ ดิกชันนารี และทูเพิล เป็นตัวอย่างของ **โครงสร้างข้อมูล** (**data structures**) ในบทนี้ เราได้เริ่มเห็น*โครงสร้างข้อมูลประกอบ* (*compound data structure*) เช่น ลิสต์ของทูเพิล หรือ ดิกชันนารีที่ใช้ทูเพิลเป็นกุญแจ และลิสต์เป็นค่า โครงสร้างข้อมูลประกอบ มีประโยชน์มาก แต่ก็เป็นความเสี่ยงให้เกิดข้อผิดพลาด แบบที่อาจจะเรียกว่า **ข้อผิดพลาดจากสัดส่วน** (**shape error**) นั่นคือ ข้อผิดพลาด ที่เกิดเมื่อใช้โครงสร้างข้อมูลผิดชนิด ผิดขนาด หรือผิดโครงสร้าง ตัวอย่าง ถ้าโปรแกรมหรือฟังก์ชันรอรับลิสต์ที่มีเลขจำนวนเต็มหนึ่งตัว แต่เราให้เลขจำนวนเต็มหนึ่งตัว (เลขจำนวนเต็มแบบดั้งเดิม ไม่ใช่อยู่ในลิสต์) มันจะไม่ทำงาน

เพื่อช่วยดีบักข้อผิดพลาดจากสัดส่วน หนังสือเล่มได้เตรียมโมดูลพิเศษไว้ให้ ชื่อว่า `structshape` ที่มีฟังก์ชัน `structshape` ที่รับโครงสร้างข้อมูลใด ๆ เป็นอาร์กิวเมนต์ แล้วส่งคืนสายอักขระที่สรุปสัดส่วนของข้อมูลออกมา โมดูลนี้สามารถดาวน์โหลดได้จาก `http://thinkpython2.com/code/structshape.py`

ตัวอย่างผลลัพธ์สำหรับลิสต์ง่าย ๆ

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

โปรแกรมที่หรูหราซับซ้อนขึ้น อาจจะสามารถรายงาน “list of 3 int**s**” ออกมาได้ แต่ว่ามันง่ายกว่า ที่จะไม่ต้องไปยุ่งกับรูปพหูพจน์ของภาษาอังกฤษ

อันนี้เป็นตัวอย่าง เมื่อใช้กับลิสต์ของลิสต์

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

ถ้าอิลิเมนต์ของลิสต์ไม่ใช่ชนิดเดียวกัน `structshape` จับมันรวมกลุ่มกัน ตามลำดับชนิด ดังเช่น

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

อันนี้ตัวอย่าง ลิสต์ของทูเพิล

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

และอันนี้ตัวอย่าง ดิกชันนารีที่มี 3 รายการที่แปลงเลขจำนวนเต็มไปเป็นสายอักขระ

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

ถ้าเรามีปัญหาการตรวจสอบโครงสร้างข้อมูล `structshape` อาจจะช่วยได้

12.9. อภิธานศัพท์

ทูเพิล (tuple): ลำดับของอิลิเมนต์ที่ไม่สามารถเปลี่ยนแปลงค่าได้

การกำหนดค่าทูเพิล (tuple assignment): การกำหนดค่าที่มีข้อมูลแบบลำดับอยู่ทางขวามือ และมีทูเพิลของตัวแปรต่าง ๆ อยู่ซ้ายมือ. ค่าต่าง ๆ ทางขวามือจะถูกประมวลผลก่อน แล้วจึงกำหนดค่าให้กับตัวแปรต่าง ๆ ทางซ้ายมือ

การรวบรวม (gather): การดำเนินการประกอบตัวแปรทูเพิลขึ้นจากตัวแปรอาร์กิวเมนต์หลาย ๆ ตัว (อาร์กิวเมนต์ความยาวแปรผัน)

การแยกกระจาย (scatter): การดำเนินการแยกกระจายค่ารายการต่าง ๆ ในตัวแปรลำดับข้อมูล ไปให้กับอาร์กิวเมนต์ต่าง ๆ แต่ละตัว

ซิปอ็อบเจกต์ (zip object): ผลลัพธ์จากการเรียกใช้ฟังก์ชันสำเร็จรูป `zip` ซึ่งเป็นอ็อบเจกต์ที่สามารถใช้ท่องสำรวจลำดับข้อมูลที่ได้ จากการประกอบสองลำดับข้อมูลเข้าด้วยกัน

ตัววนซ้ำ (iterator): อ็อบเจกต์ที่สามารถวนสำรวจลำดับของข้อมูลได้ แต่ไม่ใช่ลิสต์ และไม่สามารถใช้ตัวดำเนินการหรือเมธอดของลิสต์ได้

โครงสร้างข้อมูล (data structure): การรวบรวมค่าข้อมูลต่าง ๆ ที่เกี่ยวข้องกัน มักถูกจัดการเป็น ลิสต์ ดิกชันนารี หรือ ทูเพิล เป็นต้น

ข้อผิดพลาดจากสัดส่วน (shape error): ข้อผิดพลาดจากค่าข้อมูลที่ได้ผิดสัดส่วนที่คาดหวัง นั่นคืออาจจะผิดชนิดหรือขนาด

12.10. แบบฝึกหัด

***แบบฝึกหัด* 12.1.** *เขียนฟังก์ชันชื่อ `most_frequent` ที่รับสายอักขระ และพิมพ์ตัวอักษรออกมา โดยเรียงลำดับตามความถี่จากน้อยไปมาก. ลองหาข้อความตัวอย่างจากหลาย ๆ ภาษา และดูว่าความถี่ของแต่ละอักษรแตกต่างอย่างไรบ้างสำหรับแต่ละภาษา. ลองเปรียบเทียบผลลัพธ์ที่ได้กับตารางใน `http://en.wikipedia.org/wiki/Letter_frequencies` โปรแกรมเฉลย: `http://thinkpython2.com/code/most_frequent.py`*

***แบบฝึกหัด* 12.2.** *คำสลับอักษรอีก!*

1. *เขียนโปรแกรม ให้อ่านรายการคำ จากไฟล์ (ดูหัวข้อ 9.1) และพิมพ์กลุ่มคำทั้งหมดที่เป็นคำสลับอักษร (anagram)*

   ข้างล่างเป็นตัวอย่างของเอาต์พุต

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   คำใบ้: เราอาจจะต้องสร้างดิกชันนารี ที่แปลงจากกลุ่มหมู่ของตัวอักษร ไปเป็นลิสต์ของคำ ที่สามารถสะกดด้วยอักษรเหล่านั้น คำถามคือ เราจะใช้กลุ่มหมู่ของตัวอักษรไปเป็นกุญแจของดิกชันนารีได้อย่างไร

2. *ดัดแปลงโปรแกรมข้างต้น ให้พิมพ์ลิสต์ที่ยาวที่สุดก่อน (ลิสต์ที่มีจำนวนคำสลับอักษรมากที่สุด) ตามด้วยลิสต์ที่ยาวอันดับสอง ไปเรื่อย ๆ*

3. *ในเกมสแคร์บเบิล (Scrabble) เราจะได้"บิงโก" เมื่อเราลงตัวทั้งเจ็ดตัวของเราได้ และพอมันไปรวมกับตัวที่วางในบอร์ดแล้ว ได้เป็นคำแปดตัวอักษร คำแปดตัวอักษรใดบ้างที่น่าจะได้บิงโกมากที่สุด? คำใบ้: มีเจ็ดคำ*

   เฉลย: `http://thinkpython2.com/code/anagram_sets.py`

แบบฝึกหัด **12.3.** *คำสองคำจะเรียกว่าเป็น "คู่สลับเสียง" (metathesis pair) ถ้าเราสามารถแปลงคำหนึ่งเป็นอีกคำได้โดยการสลับสองตัวอักษร เช่น "converse" และ "conserve" เขียนโปรแกรมที่หาคู่สลับเสียงทั้งหมดออกมาจากพจนานุกรม คำใบ้: อย่าทดสอบทุก ๆ คู่ของคำ และอย่าทดสอบทุก ๆ การสลับอักษร เฉลย:* `http://thinkpython2.com/code/metathesis.py` *ขอบคุณ: แบบฝึกหัดนี้ได้รับแรงบันดาลใจจากตัวอย่างใน* `http://puzzlers.org`*.*

แบบฝึกหัด **12.4.** *อีกปริศนาจากจอมปริศนาคาร์ทอร์ค (*`http://www.cartalk.com/content/puzzlers`*):*

> *คำไหนในภาษาอังกฤษที่ยาวที่สุด ที่ถ้าดึงตัวอักษรออกทีละตัว แล้วส่วนที่เหลือยังเป็นคำในภาษาอังกฤษอยู่*
>
> *ตัวอักษรที่เอาออกอาจจะเป็นตัวที่ต้น หรือปลาย หรือกลางคำก็ได้ แต่ต้องไม่มีการเรียงตัวอักษรที่เหลือใหม่. ทุก ๆ ครั้งที่เอาตัวอักษรออก ส่วนที่เหลือต้องเป็นคำภาษาอังกฤษ. ทำไปเรื่อย ๆ สุดท้ายเราก็จะเหลือแต่ตัวอักษรตัวเดียว และตัวอักษรตัวสุดท้ายที่เหลืออยู่ก็ต้องเป็นคำในภาษาอังกฤษด้วย (มีอยู่ในพจนานุกรม). ผมอยากจะรู้ว่า คำแบบนี้ที่ยาวที่สุดคือคำว่าอะไร และมันยาวเท่าไร?*
>
> *ผมยกตัวอย่างเล็ก ๆ ให้อันหนึ่ง คำว่า sprite ถ้าเราเริ่มด้วย sprite เราเอาตัวอักษรหนึ่งออก เอาตัว r ออกจากกลางคำ เราจะเหลือ spite จากนั้นเอา e ออกจากท้าย เราเหลือ spit เอา s ออก เราเหลือ pit แล้ว it แล้ว I*

เขียนโปรแกรมที่หาคำทั้งหมดที่สามารถลดรูปได้แบบนี้ แล้วหาคำที่ยาวที่สุด

แบบฝึกหัดนี้ค่อนข้างจะยากกว่าอันอื่น ๆ คำแนะนำคือ

1. *เราอาจจะเขียนฟังก์ชันที่รับอาร์กิวเมนต์เป็นคำในภาษาอังกฤษ แล้วหาลิสต์ของคำทั้งหมด ที่เกิดจากการตัดอักษรออกตัวหนึ่ง ลิสต์ของคำนี้เป็น "ลูก" ของคำ*

2. *ทำแบบการเรียกซ้ำ คำจะลดรูปได้ ถ้าลูกของมันลดรูปได้ และกรณีฐาน เราอาจจะนับสายอักขระว่างว่าสามารถลดรูปได้*

3. *รายการคำที่ให้* `words.txt` *ไม่มีคำอักษรเดี่ยว ดังนั้นเราอาจต้องเพิ่ม "I" และ "a" และสายอักขระว่าง ๆ เข้าไป*

4. *เพื่อเพิ่มประสิทธิภาพของโปรแกรม เราอาจต้องจำคำต่าง ๆ ที่รู้แล้วว่าลดรูปได้ไว้*

เฉลย: `http://thinkpython2.com/code/reducible.py`

13. กรณีศึกษา การเลือกโครงสร้างข้อมูล

ถึงตอนนี้ เราได้เรียนโครงสร้างข้อมูลหลัก ๆ ของไพธอนไปแล้ว และเราก็ได้เห็นอัลกอริธึมที่ใช้มันไปบ้างแล้วด้วย ถ้าอยากรู้เรื่องอัลกอริธึมมากขึ้น อาจถึงเวลาแล้วที่จะไปอ่านบท B แต่ก็ไม่ได้จำเป็นว่าจะต้องอ่านบท B ก่อนจะเรียนบทนี้ จริง ๆ จะอ่านบท B ตอนไหนก็ได้ที่สนใจ

บทนี้นำเสนอกรณีตัวอย่างด้วยแบบฝึกหัด ที่จะช่วยให้คิดเรื่องเลือกโครงสร้างข้อมูล และก็จะได้ฝึกใช้มันด้วย

13.1. การวิเคราะห์ความถี่คำ

เหมือนเคย ที่อย่างน้อย เราควรจะลองทำแบบฝึกหัดด้วยตัวเองก่อนที่จะไปเปิดดูเฉลย

***แบบฝึกหัด* 13.1.** *เขียนโปรแกรมให้อ่านไฟล์ แล้วแยกแต่ละบรรทัดออกเป็นคำ ๆ เอาช่องว่างต่าง ๆ (whitespace) และเครื่องหมายวรรคตอนต่าง ๆ (punctuation) ออกจากคำ และแปลงให้เป็นอักษรตัวเล็ก (lowercase)*

คำใบ้: โมดูล `string` *มีสายอักขระชื่อ* `whitespace` *และสายอักขระชื่อ* `punctuation`*. สายอักขระ* `whitespace` *มีอักขระช่องว่าง (space) อักขระตั้งระยะ (tab) อักขระขึ้นบรรทัดใหม่ (newline) เป็นต้น สายอักขระ* `punctuation` *มีอักขระเครื่องหมายวรรคตอนต่าง ๆ ลองดู*

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

นอกจากนั้น อาจจะลองดูเมธอดของสายอักขระ เช่น เมธอด `strip` *เมธอด* `replace` *และเมธอด* `translate`

***แบบฝึกหัด* 13.2.** *ลองดูโครงการกูเทินแบร์ค (`http://gutenberg.org`) และดาวน์โหลดหนังสือที่ชอบ ที่หมดลิขสิทธิ์ไปแล้ว ดาวน์โหลดไฟล์มาในรูปข้อความธรรมดา (plain text) ดัดแปลงโปรแกรมจากแบบฝึกหัดที่แล้ว เพื่ออ่านหนังสือที่ดาวน์โหลดมา ข้ามพวกข้อมูลประกอบที่อยู่ตอนต้นของไฟล์ และประมวลผลส่วนที่เหลือ*

จากนั้น ดัดแปลงโปรแกรมให้นับจำนวนคำทั้งหมดในหนังสือ และจำนวนครั้งที่แต่ละคำปรากฏ

พิมพ์จำนวนคำต่าง ๆ ที่พบในหนังสือ เปรียบเทียบหนังสือจากนักเขียนต่าง ๆ ที่เขียนคนละยุค นักเขียนคนไหนที่ใช้คำศัพท์ได้กว้างขวางหลากหลายที่สุด?

***แบบฝึกหัด* 13.3.** *ดัดแปลงโปรแกรมจากแบบฝึกหัดที่แล้ว เพื่อพิมพ์คำที่ใช้มากที่สุด 20 คำในหนังสือ*

***แบบฝึกหัด* 13.4.** *ดัดแปลงโปรแกรมที่แล้ว เพื่อให้อ่านลิสต์ของคำ (ดูหัวข้อ 9.1) แล้วพิมพ์ทุกคำในหนังสือที่ไม่ได้อยู่ในลิสต์ของคำ มีกี่คำที่เจอที่พิมพ์ผิด? มีกี่คำที่เป็นคำทั่ว ๆ ไป ที่**ควร**จะอยู่ในลิสต์ของคำ และมีกี่คำที่คลุมเครือ?*

13.2. ตัวเลขค่าสุ่ม

ถ้าได้อินพุตเหมือน ๆ กัน คอมพิวเตอร์(ส่วนใหญ่)จะให้เอาต์พุตออกมาเหมือนกันทุกครั้ง ลักษณะแบบนี้เรียกว่า **ลักษณะชี้เฉพาะ** (deterministic) *ลักษณะการชี้เฉพาะ*โดยทั่วไปแล้วมันก็ดี เพราะว่า เรามั่นใจได้ว่า การคำนวณแบบเดียวกัน จะได้ผลออกมาแบบเดียวกัน แต่สำหรับงานบางลักษณะ เราต้องการให้คอมพิวเตอร์มีลักษณะที่เดาไม่ได้บ้าง เกมส์ก็เป็นตัวอย่างหนึ่ง แต่ยังมีตัวอย่างอื่น ๆ อีกมาก ที่ต้องลักษณะเดาไม่ได้

การที่จะเขียนโปรแกรมให้ไม่เป็น*ลักษณะชี้เฉพาะ*เลย คือเดาไม่ได้จริง ๆ ทำยากมาก แต่ก็มีหลายวิธีที่ อย่างน้อย ก็ช่วยให้โปรแกรมดูเหมือนว่า ไม่เป็น*ลักษณะชี้เฉพาะ* หนึ่งให้วิธีเหล่านั้นก็คือ ใช้อัลกอริธึมที่สร้างตัวเลข**สุ่มเทียม** (pseudorandom) ค่าของตัวเลขสุ่มเทียม ไม่ได้ถูกสุ่มขึ้นมาจริง ๆ แต่มันจะสร้างมาจากการคำนวณ*ลักษณะชี้เฉพาะ* ที่ทำให้ ตัวเลขที่ได้ ถ้าดูเผิน ๆ มันจะดูเหมือนว่าถูกสุ่มมา

โมดูล `random` มีฟังก์ชันต่าง ๆ ที่สามารถสร้างตัวเลขสุ่มเทียมได้ (ซึ่งตั้งแต่นี้ไป เราจะแค่เรียกสั้น ๆ ว่า “สุ่ม”)

ฟังก์ชัน `random` จะให้จำนวนจริงที่มีค่าสุ่มมาระหว่าง 0.0 ถึง 1.0 (รวม 0.0 แต่ไม่รวม 1.0) ทุกครั้งที่เราเรียก `random` เราจะได้เลขสุ่มใหม่ ซึ่งเป็นเลขถัดไป ในลำดับที่ยาวมาก เพื่อให้เห็นตัวอย่าง ลองรันลูปนี้:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

ฟังก์ชัน `randint` รับพารามิเตอร์ `low` และ `high` แล้วให้ค่าสุ่มเลขจำนวนเต็มระหว่าง `low` และ `high` ออกมา (รวมค่า `low` และ `high` ด้วย)

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

เพื่อเลือกอิลิเมนต์จากลำดับแบบสุ่ม เราสามารถใช้ `choice` ได้:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

โมดูล `random` ยังมีฟังก์ชันต่าง ๆ ที่สร้างค่าสุ่มจาก*การแจกแจงต่อเนื่อง* (*continuous distributions*) รวมถึง การแจกแจงแบบเกาส์เซียน (Gaussian distribution) การแจกแจงแบบเลขชี้กำลัง (exponential distribution) การแจกแจงแกมมา (gamma distribution) เป็นต้น

***แบบฝึกหัด* 13.5.** *เขียนฟังก์ชันชื่อ `choose_from_hist` ที่รับฮิสโตแกรม (histogram) แบบที่กำหนดในหัวข้อ 11.2 และส่งคืนค่าสุ่มจากฮิสโตแกรม นั่นคือ เลือกค่าสุ่มด้วยความน่าจะเป็น ให้เป็นสัดส่วนตามความถี่ในฮิสโตแกรม ตัวอย่าง สำหรับฮิสโตแกรมนี้:*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

*ฟังก์ชันควรจะให้ **a** ออกมาด้วยความน่าจะเป็น* 2/3 *และให้ **b** ออกมาด้วยความน่าจะเป็น* 1/3

13.3. ฮิสโตแกรมคำ

ควรจะลองทำแบบฝึกหัดที่แล้ว ก่อนจะอ่านเรื่องนี้ต่อ โปรแกรมเฉลยดาวน์โหลดได้จาก `http://thinkpython2.com/code/analyze_book1.py` และก็อาจจะต้องการอันนี้ด้วย `http://thinkpython2.com/code/emma.txt`

นี่เป็นโปรแกรม ที่อ่านไฟล์ และสร้างฮิสโตแกรมของคำต่าง ๆ ในไฟล์:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
      word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

โปรแกรมนี้อ่านไฟล์ `emma.txt` เป็นนิยายเรื่อง *เอ็มม่า* (*Emma*) เขียนโดน เจน ออสเทน (Jane Austen)

ฟังก์ชัน `process_file` ลูปวนทีละบรรทัดของไฟล์ ส่งแต่ละบรรทัดไปที่ `process_line` ฮิสโตแกรม `hist` ถูกใช้เป็น *ตัวสะสม* (*accumulator*) สะสมค่าความถี่จากแต่ละบรรทัด

ฟังก์ชัน `process_line` ใช้เมธอดของสายอักขระ `replace` เพื่อแทนที่ เครื่องหมายยัติภังค์ (hyphen) ด้วยช่องว่าง ก่อนที่จะใช้เมธอด `split` เพื่อแยกบรรทัด ออกไปเป็นลิสต์ของสายอักขระ มัน

ท่องสำรวจลิสต์ของคำ และใช้เมธอด `strip` และเมธอด `lower` เพื่อตัด*เครื่องหมายวรรคตอน*ต่าง ๆ และแปลงเป็นอักษรตัวเล็ก (อาจจะพูดสั้น ๆ ว่า สายอักขระถูก“แปลง” แต่ถ้าจำได้ สายอักขระเป็นข้อมูล*ที่ไม่สามารถเปลี่ยนแปลงได้* ดังนั้น เมธอดพวก `strip` และ `lower` จะให้สายอักขระใหม่ออกมา)

สุดท้าย ฟังก์ชัน `process_line` ปรับค่าของฮิสโตแกรม โดยการสร้างรายการสำหรับคำใหม่ หรือเพิ่มค่าให้กับคำที่มีอยู่แล้ว

เพื่อจะนับจำนวนคำทั้งหมดในไฟล์ เราสามารถรวมความถี่ต่าง ๆ ในฮิสโตแกรมได้:

```
def total_words(hist):
    return sum(hist.values())
```

จำนวนคำศัพท์ ก็คือจำนวนรายการในดิกชันนารี:

```
def different_words(hist):
    return len(hist)
```

นี่เป็นโปรแกรมสำหรับพิมพ์ผลลัพธ์ออกมา:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

และผลลัพธ์ที่ได้:

```
Total number of words: 161080
Number of different words: 7214
```

13.4. คำที่พบบ่อยที่สุด

เพื่อจะหาคำศัพท์ที่ใช้บ่อยที่สุด เราสามารถสร้างลิสต์ของทูเพิล ที่แต่ละทูเพิลมีคำศัพท์และความถี่ของมันแล้วก็จัดเรียงมัน

ฟังก์ชันต่อไปนี้รับฮิสโตแกรม และส่งคืนลิสต์ของทูเพิล ที่ทูเพิลเป็นคู่ของคำกับความถี่:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))
```

```
    t.sort(reverse=True)
    return t
```

ในแต่ละทูเพิล ความถี่แสดงก่อนคำ และผลลัพธ์ก็เรียงลำดับตามความถี่จากมากไปน้อย นี่เป็นลูปที่พิมพ์คำสิบคำที่พบบ่อยที่สุดออกมา:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

อาร์กิวเมนต์ `sep` บอกคำสั่ง `print` ให้ใช้อักขระตั้งระยะ เป็น “ตัวแยก” แทนช่องว่าง ทำให้คอลัมน์ที่สองเรียงกันเป็นแนว นี่เป็นผลลัพธ์ที่ได้จากนิยาย*เอ็มม่า*

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

โปรแกรมนี้จริง ๆ สามารถเขียนให้ง่ายกว่านี้ได้ โดยใช้พารามิเตอร์ `key` ของฟังก์ชัน `sort` ถ้าสนใจ ลองศึกษาดูได้จาก `https://wiki.python.org/moin/HowTo/Sorting`

13.5. พารามิเตอร์เสริม

เราได้ดูฟังก์ชันสำเร็จรูป และเมธอดต่าง ๆ ที่รับพารามิเตอร์เสริมมาแล้ว เราเองก็เขียนฟังก์ชันที่รับพารามิเตอร์เสริมได้เหมือนกัน ตัวอย่าง นี่เป็นฟังก์ชันที่พิมพ์คำที่พบบ่อยที่สุดในฮิสโตแกรมออกมา

```
def print_most_common(hist, num=10):
    t = most_common(hist)
```

```
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

พารามิเตอร์แรกต้องใส่ แต่พารามิเตอร์ที่สองเป็นพารามิเตอร์เสริม ถ้าไม่ใส่ num จะใช้ค่า*ดีฟอลท์* เป็น 10

ถ้าเรียกใช้ ด้วยหนึ่งอาร์กิวเมนต์:

```
print_most_common(hist)
```

ตัวแปร num จะใช้ค่าดีฟอลท์ แต่ถ้าเรียกใช้ ด้วยอาร์กิวเมนต์สองตัว:

```
print_most_common(hist, 20)
```

ตัวแปร num จะใช้ค่าของอาร์กิวเมนต์ พูดง่าย ๆ คือ อาร์กิวเมนต์เสริมไป**แทนที่** (override) ค่าดีฟอลท์

ถ้าฟังก์ชันมีทั้งพารามิเตอร์บังคับ และพารามิเตอร์เสริม พารามิเตอร์บังคับทุกตัวต้องมาก่อน แล้วค่อยตามด้วยพารามิเตอร์เสริม

13.6. การลบดิกชันนารี

การหาคำที่มีในหนังสือ แต่ไม่มีในลิสต์ของคำศัพท์จาก words.txt ก็เหมือนกับปัญหาการลบของเซต นั่นคือ เราต้องการหาคำทุกคำจากเซตหนึ่ง (คำต่าง ๆ ในหนังสือ) ที่ไม่ได้อยู่ในอีกเซตหนึ่ง (คำต่าง ๆ ในลิสต์)

ฟังก์ชัน subtract รับดิกชันนารีสองตัว คือ d1 และ d2 แล้วส่งคืนดิกชันนารีใหม่ออกมา โดย ดิกชันนารีใหม่มีกุญแจทั้งหมดจาก d1 ที่ไม่มีใน d2 และเพราะเราไม่สนใจค่าของมัน เราจะให้ค่าในคู่กุญแจค่าทุกรายการเป็น None

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

เพื่อหาคำในหนังสือที่ไม่มีอยู่ใน words.txt เราก็สามารถใช้ฟังก์ชัน process_file เพื่อสร้างฮิสโตแกรมสำหรับ words.txt แล้วค่อยลบออกได้:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

อันนี้เป็นผลลัพธ์ที่ได้จากนิยาย*เอ็มม่า*:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

เห็นได้ว่า บางคำเป็นชื่อและคำแสดงความเป็นเจ้าของต่าง ๆ. คำอื่น ๆ เช่น “rencontre” ก็เป็นคำที่ไม่ค่อยใช้เท่าไรแล้วในปัจจุบัน แต่ก็มีหลาย ๆ คำที่เป็นคำศัพท์ธรรมดาที่จริง ๆ แล้วควรจะมีอยู่ในลิสต์

แบบฝึกหัด **13.6.** *ไพธอนมีโครงสร้างข้อมูลที่เรียกว่า* `set` *ที่มีตัวดำเนินการของเซตอยู่หลายตัว ถ้าสนใจ ลองอ่านหัวข้อ 19.5 หรือศึกษาดูเอกสารที่* `http://docs.python.org/3/library/stdtypes.html#types-set`

เขียนโปรแกรมที่ใช้การลบเซต เพื่อหาคำต่าง ๆ ในหนังสือที่ไม่อยู่ในลิสต์ของคำ เฉลย: `http://thinkpython2.com/code/analyze_book2.py`

13.7. คำสุ่ม

เพื่อเลือกสุ่มคำตามฮิสโตแกรม อัลกอริธึมที่ง่ายที่สุด คือสร้างลิสต์ที่มีหลาย ๆ สำเนาของแต่ละคำ โดยจำนวนสำเนาเป็นไปตามความถี่ แล้วก็สุ่มเลือกคำจากลิสต์:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

นิพจน์ `[word] * freq` สร้างลิสต์ขึ้นมาด้วย สำเนาของสายอักขระ `word` เป็นจำนวน `freq` สำเนา เมธอด `extend` คล้ายกับ `append` เพียงแต่อาร์กิวเมนต์เป็นข้อมูลแบบลำดับ

อัลกอริธึมนี้ก็ทำงานได้ แต่ว่ามันไม่มีประสิทธิภาพ ทุกครั้งที่เราเลือกสุ่มคำออกมา มันก็ต้องสร้างลิสต์ขึ้นใหม่ ซึ่งลิสต์ก็ใหญ่พอ ๆ กับหนังสือเลย วิธีปรับปรุงที่เห็นได้ชัด ๆ เลย ก็คือ สร้างลิสต์ขึ้นมาทีเดียว และใช้มันหลาย ๆ ครั้ง แต่ลิสต์ก็ยังใหญ่อยู่ วิธีอื่นที่ดีกว่า ก็คือ

1. ใช้ `keys` เพื่อเอาลิสต์ของคำต่าง ๆ (คำไม่ซ้ำ) จากหนังสือออกมา

2. สร้างลิสต์ที่เก็บผลรวมสะสมของความถี่คำ (ดูแบบฝึกหัด 10.2) รายการสุดท้ายในลิสต์ (ผลรวมสะสมสุดท้าย) คือ จำนวนคำทั้งหมดในหนังสือ n (จำนวน n รวมจำนวนคำที่ซ้ำด้วย)

3. เลือกตัวเลขสุ่มขึ้นมาจากเลขระหว่าง 1 ถึง n แล้วใช้*วิธีค้นหาแบบแบ่งสอง* (See Exercise 10.10) เพื่อหาดัชนีของตัวเลขสุ่มนี้ในผลรวมสะสม (*อธิบายเพิ่มเติม เสมือนเราเอาคำทั้งหมดมาเรียงกัน คำที่ปรากฏหลายครั้ง ก็เรียงคำนั้นต่อกันซ้ำ ๆ หลายครั้ง คำแต่ละคำให้มีดัชนีเฉพาะของตัวเอง แต่คำเดียวกันปรากฏกี่ครั้งก็ตาม ให้ใช้ดัชนีเดียวกัน ดังนั้นเลขลำดับของคำที่เรียงกัน จะเรียงตั้งแต่ 1 ถึง n แต่ดัชนีจะมีแค่เท่ากับจำนวนคำที่ไม่ซ้ำกัน สุ่มหยิบเลขลำดับมาหนึ่งลำดับ แล้วไปหาดูว่าดัชนีของลำดับนี้คือเท่าไร เนื่องจากเลขลำดับสุ่มขึ้นมา ทุกคำรวมคำซ้ำ มีโอกาสเท่า ๆ กัน แต่พอไปดูดัชนีของลำดับ คำที่มีคำซ้ำมากกว่า ก็จะมีดัชนีมากกว่าและมีโอกาสถูกเลือกมากกว่า นี่เป็นเทคนิคพื้นฐานของวิชาการจำลองและโมเดลทางคอมพิวเตอร์*)

4. ใช้ดัชนีที่ได้ เพื่อหาคำจากลิสต์ของคำ

แบบฝึกหัด **13.7.** *เขียนโปรแกมที่ใช้อัลกอริธึมนี้ เพื่อสุ่มคำขึ้นมาจากหนังสือ เฉลย:* ***`http://thinkpython2.com/code/analyze_book3.py`***

13.8. การวิเคราะห์มาร์คอฟ

ถ้าเราสุ่มเลือกคำต่าง ๆ มาจากหนังสือ เราอาจจะได้เห็นคำศัพท์ต่าง ๆ แต่เราจะไม่ได้เห็นประโยค

```
this the small regard harriet which knightley's it most things
```

ลำดับของคำสุ่ม ๆ ยากที่จะทำให้รู้เรื่องอะไรได้ เพราะว่ามันไม่มีความสัมพันธ์ระหว่างคำต่าง ๆ ที่ต่อ ๆ กัน ตัวอย่างเช่น ในประโยคจริง ๆ (ในภาษาอังกฤษ) เราน่าจะเห็น*คำกำกับนาม* (*article*) เช่น “the” แล้วน่าก็ตามด้วยคำคุณศัพท์ หรือคำนาม ไม่น่าตามด้วยคำกริยา หรือคำวิเศษณ์

วิธีหนึ่งที่ใช้อธิบายความสัมพันธ์ในลักษณะแบบนี้ คือ *การวิเคราะห์มาร์คอฟ* (*Markov analysis*) ที่ช่วยประมาณความน่าจะเป็นของคำต่อไป ของลำดับของคำ ตัวอย่าง เพลง *Eric, the Half a Bee* เริ่มด้วย:

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

ในเนื้อเพลงนี้ วลี "half the" จะตามด้วยคำว่า "bee" เสมอ แต่วลี "the bee" อาจจะตามด้วย "has" หรือ "is"

ผลลัพธ์จากการวิเคราะห์มาร์คอฟเป็นการแปลงจาก *พรีฟิกซ์* (*prefix*) (เช่น "half the" และ "the bee") ไปเป็นคำต่อมา หรือ *ซัฟฟิกซ์* (*suffix*) ทุก ๆ คำที่เป็นไปได้ (เช่น "has" และ "is")

ด้วยการแปลงแบบนี้ เราสามารถสร้างข้อความสุ่มได้ เริ่มจาก พรีฟิกซ์ คำไหนก็ได้ แล้วเลือกคำต่อมาโดยสุ่มจากซัฟฟิกซ์ที่เป็นไปได้ จากนั้น เราก็สามารถใช้คำในพรีฟิกซ์กับคำต่อมาที่ได้ มาสร้างเป็นพรีฟิกซ์ใหม่ แล้วก็ทำต่อไปเรื่อย ๆ

ตัวอย่างเช่น ถ้าเราเริ่มด้วยพรีฟิกซ์ คำว่า "Half a" และคำถัดมาจะเป็น "bee" เพราะว่าพรีฟิกซ์นี้ปรากฏแค่ครั้งเดียวในเนื้อเพลง (และมันตามด้วย "bee") พรีฟิกซ์ใหม่จะเป็น "a bee" ดังนั้นซัฟฟิกซ์ต่อไปอาจจะเป็น "philosophically" หรือ "be" หรือ "due" ก็ได้

ตัวอย่างนี้ใช้พรีฟิกซ์มีความยาวเป็นสองคำเสมอ แต่จริง ๆ แล้ว เราสามารถทำการวิเคราะห์มาร์คอฟกับพรีฟิกซ์ความยาวเท่าไรก็ได้

***แบบฝึกหัด* 13.8.** *การวิเคราะห์มาร์คอฟ:*

1. *เขียนโปรแกรม เพื่ออ่านข้อความจากไฟล์ และทำการวิเคราะห์มาร์คอฟ ผลลัพธ์ควรจะเป็นดิกชันนารีที่แปลงจากพรีฟิกซ์ไปเป็นกลุ่มหมู่ของซัฟฟิกซ์ต่าง ๆ ที่เป็นไปได้. กลุ่มหมู่ที่ใช้ อาจจะทำเป็นลิสต์ หรือทูเพิล หรือดิกชันนารี ก็ได้ตามใจชอบ ตามที่เห็นสมควรเลย ทำเสร็จแล้ว ลองทดสอบโปรแกรมด้วย พรีฟิกซ์ความยาวสองก่อน และตัวโปรแกรมเองก็ควรจะเขียนแบบที่ทำให้ง่าย ถ้าต้องการจะเปลี่ยนพรีฟิกซ์เป็นความยาวอื่น ๆ*

2. *เพิ่มฟังก์ชันเข้าไปในโปรแกรมที่แล้ว เพื่อสร้างข้อความสุ่ม โดยใช้การวิเคราะห์มาร์คอฟ ตัวอย่างนี้มาจากนิยายเอ็มม่า ที่ใช้พรีฟิกซ์ยาวสองคำ:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

   ตัวอย่างนี้ ปล่อยพวกเครื่องหมายวรรคตอนติดไปกับคำ ผลที่ได้เกือบถูกตามวากยสัมพันธ์แต่ก็ยังไม่ถูก สำหรับความหมายมันก็เกือบได้แต่ก็ยังไม่ได้

   จะเกิดอะไรขึ้น ถ้าเราเพิ่มความยาวของพรีฟิกซ์? ข้อความสุ่มจะฟังมีความหมายมากขึ้นหรือเปล่า?

3. *ถ้าโปรแกรมทำงานได้แล้ว อาจจะลองผสมดู: ถ้าเราผสมข้อความจากหนังสือสองเล่มหรือมากกว่าเข้าด้วยกัน ข้อความสุ่มที่สร้างขึ้นมา จะผสมคำศัพท์หรือวลี จากแหล่งต่าง ๆ เข้าด้วยกัน*

กรณีศึกษานี้ ดัดแปลงจากตัวอย่างของ การปฏิบัติของการเขียนโปรแกรม โดย เคอนิแกนและไพค์ (Kernighan and Pike, The Practice of Programming, Addison-Wesley, 1999.)

ควรจะไปลองทำแบบฝึกหัดนี้ ก่อนที่จะศึกษาเนื้อหาต่อไป สามารถดาวน์โหลดเฉลยได้จาก `http://thinkpython2.com/code/markov.py` และอาจจะต้องไฟล์ประกอบ `http://thinkpython2.com/code/emma.txt`

13.9. โครงสร้างข้อมูล

ตอนใช้การวิเคราะห์มาร์คอฟ เพื่อสร้างข้อความสุ่มก็สนุกดี แต่จริง ๆ แล้ว มันมีประเด็นที่สำคัญของแบบฝึกหัดนี้ คือ การเลือกใช้โครงสร้างข้อมูล เวลาทำแบบฝึกหัดที่แล้ว เราต้องเลือก:

- เราจะแทนพรีฟิกซ์อย่างไร
- เราจะแทนกลุ่มหมู่ของซัฟฟิกซ์ที่เป็นไปได้อย่างไร
- เราจะแปลงจากพรีฟิกซ์แต่ละอันไปเป็นกลุ่มหมู่ของซัฟฟิกซ์ที่เป็นไปได้อย่างไร

อันสุดท้ายอาจจะง่าย ดิกชันนารีเป็นตัวเลือกที่ชัดเจน สำหรับการแปลงกุญแจต่าง ๆ ไปหาค่าต่าง ๆ ที่คู่กัน

สำหรับพรีฟิกซ์ ตัวเลือกที่ชัด ๆ ก็มี สายอักขระ หรือ ลิสต์ของสายอักขระ หรือ ทูเพิลของสายอักขระ

สำหรับซัฟฟิกซ์ ตัวเลือกหนึ่งคือ ลิสต์ อีกอันคือ ฮิสโตแกรม (ดิกชันนารี)

เราควรจะเลือกอย่างไร? ขั้นแรก คือ คิดถึงการดำเนินการต่าง ๆ ที่เราต้องเขียนสำหรับแต่ละโครงสร้างข้อมูล สำหรับซัฟฟิกซ์ เราต้องสามารถลบคำต้น ๆ ออก และเพิ่มคำเข้าทางท้าย ๆ ได้ ตัวอย่างเช่น ถ้าซัฟฟิกซ์ เป็น “Half a” และคำต่อไปเป็น “bee” เราต้องสามารถสร้างพรีฟิกซ์ต่อไปเป็น “a bee” ได้

ตัวเลือกแรก ๆ ของเรา อาจจะเป็นลิสต์ เพราะว่า มันง่ายที่จะเพิ่มหรือลบอิลิเมนต์ แต่เราก็ต้องการใช้พรีฟิกซ์เป็นกุญแจของดิกชันนารีด้วย ดังนั้นจึงใช้ลิสต์ไม่ได้ ถ้าใช้ทูเพิล เราจะเพิ่มหรือลบไม่ได้ แต่เราสามารถใช้ตัวดำเนินการบวกเพื่อสร้างทูเพิลใหม่ได้:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

ฟังก์ชัน `shift` รับทูเพิลของคำต่าง ๆ ด้วยชื่อ `prefix` และรับสายอักขระ ด้วยชื่อ `word` แล้วสร้างทูเพิลใหม่ที่มีทุกคำใน `prefix` ยกเว้นคำแรก และเพิ่มคำจาก `word` เข้าไปท้ายทูเพิลใหม่

สำหรับกลุ่มหมู่ของซัฟฟิกซ์ ตัวดำเนินการต่าง ๆ ที่เราต้องเขียน รวมถึงการเพิ่มซัฟฟิกซ์ใหม่ (หรือเพิ่มความถี่ของซัฟฟิกซ์ที่มีอยู่แล้ว) แล้วค่อยสุ่มเลือกซัฟฟิกซ์

การเพิ่มซัฟฟิกซ์ใหม่นั้น นั้นไม่ว่าใช้ลิสต์ หรือใช้ฮิสโตแกรม ก็ทำได้ง่ายพอ ๆ กัน แต่การสุ่มเลือกอิลิเมนต์จากลิสต์ทำได้ง่าย ในขณะที่การสุ่มเลือกจากฮิสโตแกรมอย่างมีประสิทธิภาพทำได้ยากทีเดียว (ดูแบบฝึกหัด 13.7)

ที่ผ่านมา เราได้พูดถึงแต่เรื่องของความง่ายในการอิมพลิเมนเตชั่น แต่มันยังมีปัจจัยอื่น ๆ ที่ควรต้องพิจารณาในการเลือกโครงสร้างข้อมูลด้วย ปัจจัยหนึ่งคือ *เวลาทำงาน* (*run time*) บางครั้ง เรามีเหตุผลในทางทฤษฎี ที่เชื่อว่า โครงสร้างข้อมูลชนิดหนึ่งจะทำงานได้เร็วกว่าชนิดอื่น ๆ ตัวอย่างเช่น เราเคยอภิปรายกันว่า ตัวดำเนินการ `in` ทำงานกับดิกชันนารี ได้เร็วกว่าทำงานกับลิสต์ (อย่างน้อยก็ในกรณีที่จำนวนอิลิเมนต์เยอะ ๆ)

แต่ส่วนใหญ่ เราไม่รู้ก่อนหรอกว่า อิมพลิเมนเตชั่นแบบไหนจะเร็วกว่า. แนวทางเลือกหนึ่งก็คือลองอิมพลิเมนเตชั่นทั้งสองแบบเลย แล้วค่อยดูว่าแบบไหนดีกว่า. แนวทางแบบนี้ เรียกว่า **การวัดเปรียบเทียบสมรรถนะ** (**benchmarking**) อีกแนวทางเลือกที่นิยมในทางปฏิบัติคือ เลือกโครงสร้างข้อมูลที่ทำอิมพลิเมนเตชั่น หรือเขียนโปรแกรมได้ง่ายที่สุด แล้วค่อยดูว่า ผลลัพธ์เร็วพอสำหรับงานที่ต้องการหรือ

เปล่า ถ้าเร็วพอแล้ว ก็ไม่มีความจำเป็นต้องทำอะไรอย่างอื่น แต่ถ้าไม่เร็วพอ มันก็มีเครื่องมือต่าง ๆ เช่น โมดูล `profile` ที่ช่วยระบุส่วนของโปรแกรมที่ใช้เวลามากที่สุด

ปัจจัยอื่นที่มักนำมาพิจารณาก็ เช่น *พื้นที่เก็บข้อมูล* (*storage space*) ตัวอย่างเช่น การใช้ฮิสโตแกรม สำหรับกลุ่มหมู่ของซัฟฟิกซ์ต่าง ๆ อาจจะใช้พื้นที่เก็บข้อมูลน้อยกว่า เพราะว่า เราเก็บแต่ละคำแค่ครั้งเดียว ไม่ว่าคำนั้นจะปรากฏในข้อความกี่ครั้งก็ตาม ในบางกรณี การประหยัดพื้นที่ อาจจะช่วยทำให้โปรแกรมรันได้เร็วขึ้นด้วย และถ้ากรณีที่สุดโต่งจริง ๆ อาจจะเจอว่า โปรแกรมจะรับไม่ได้เลย ถ้าเราไม่มีหน่วยความจำเหลือ แต่โดยทั่ว ๆ ไป พื้นที่เก็บข้อมูล เป็นเรื่องรองจากเวลาทำงาน

สุดท้าย การอภิปรายนี้ อยู่บนความคิดที่ว่า เราควรจะใช้โครงสร้างข้อมูลเดียวกัน สำหรับทั้งการวิเคราะห์ข้อความต้นฉบับ และการสร้างข้อความสุ่ม แต่จริง ๆ แล้ว มันเป็นเรื่องที่แยกกัน เราอาจจะใช้โครงสร้างข้อมูลหนึ่งสำหรับการวิเคราะห์ข้อความต้นฉบับ แล้วแปลงเป็นอีกโครงสร้างเพื่อการสร้างข้อความสุ่มก็ย่อมได้ มันอาจจะได้ผลรวม ๆ แล้วดีกว่า ถ้าเวลาที่สั้นลงตอนสร้างข้อความสุ่ม คุ้มกับเวลาที่ใช้ในการแปลงโครงสร้างข้อมูล

13.10. การดีบัก

เวลาที่เราดีบัก หรือหาจุดผิดพลาดในโปรแกรม และโดยเฉพาะอย่างยิ่ง ถ้าเรากำลังเจอจุดผิดพลาดที่ยาก มีคำแนะนำห้าอย่างที่น่าจะลอง:

อ่าน: อ่านโค้ด ตรวจสอบโปรแกรมของเรา อ่านมันออกมา และตรวจดูว่า โปรแกรมถูกเขียนแบบที่เราอยากให้มันทำงาน

รัน: ทดลอง โดยลองเปลี่ยนแปลงโปรแกรม แล้วลองรันดูหลาย ๆ เวอร์ชั่น บางครั้ง ถ้าเรามองถูกเรื่องถูกที่ ปัญหาก็จะเห็นได้ชัด แต่บางครั้ง เราอาจจะต้องลองแกะ ๆ ดู

ไตร่ตรอง: ใช้เวลาคิดไตร่ตรองดู! ข้อผิดพลาดเป็นชนิดไหน: ข้อผิดพลาดเชิงวากยสัมพันธ์ หรือ ข้อผิดพลาดเวลาดำเนินการ หรือ ข้อผิดพลาดเชิงความหมาย? อะไรบ้างที่เรารู้จากข้อความแสดงข้อผิดพลาด อะไรบ้างที่เรารู้จากเอาต์พุตของโปรแกรม? ข้อผิดพลาดชนิดไหนที่จะทำให้เกิดปัญหาแบบที่เห็นได้? อะไรที่เราทำสุดท้ายก่อนที่ปัญหาจะเกิด?

เล่าให้เป็ดยางฟัง: ถ้าเราเล่าปัญหาให้ใครสักคนฟัง บางครั้งเราจะได้คำตอบ ก่อนที่เราจะเล่าปัญหาจบด้วยซ้ำ แล้วส่วนใหญ่ เราก็ไม่ได้ต้องการคนด้วยซ้ำ เราแค่เล่าให้เป็ดยางฟังก็พอ ต้นตอของเทคนิคนี้ เรียกว่า **การดีบักเป็ดยาง** (**rubber duck debugging**) อันนี้ไม่ได้แต่งเองนะ ดู `https://en.wikipedia.org/wiki/Rubber_duck_debugging`

ถอย: ถึงจุดหนึ่งสิ่งที่ดีที่สุดที่ทำได้คือถอยออกมา แก้การเปลี่ยนแปลงล่าสุดออก จนกระทั่งได้โปรแกรมล่าสุด ที่ทำงานได้ ที่เราเข้าใจ แล้วค่อยเริ่มทำใหม่ต่อจากนั้น

โปรแกรมเมอร์มือใหม่มักจะติดอยู่กับเทคนิคหนึ่ง ในห้าเทคนิคจากคำแนะนำข้างต้น และลืมเทคนิคอื่น ๆ แต่ละเทคนิคมีจุดอ่อนของตัวเอง

ตัวอย่างเช่น การอ่านโค้ดอาจจะช่วย ถ้าปัญหาเป็น*ข้อผิดพลาดจากการพิมพ์ผิด* (*typographical error*) แต่มันจะไม่ช่วย ถ้าเป็นปัญหาจากความเข้าใจผิดเชิงแนวคิด ถ้าเราไม่เข้าใจจริง ๆ ว่า โปรแกรมทำอะไร เราอาจจะอ่านมันเป็นร้อย ๆ ครั้ง แต่ไม่เห็นข้อผิดพลาดเลยก็ได้ เพราะว่าข้อผิดพลาดมันอยู่ในหัวเราเอง

การลองเปลี่ยนโปรแกรมแล้วทดลองรัน อาจจะช่วย ถ้าเราทำการทดสอบเล็ก ๆ ง่าย ๆ แต่ถ้าเราทดลองโดยไม่ได้คิด หรือไม่ได้ตรวจสอบโค้ดดูก่อน เราอาจจะกลายเป็น แบบที่เรียกว่า “การเขียนโปรแกรมแบบเดินสุ่ม” (random walk programming) ที่แก้โปรแกรมแบบมั่ว ๆ ไปเรื่อย ๆ จนกว่าโปรแกรมจะทำงานถูก ก็คงไม่ต้องพูดมาก การเขียนโปรแกรมแบบเดินสุ่ม มันจะใช้เวลานานมาก

เราต้องใช้เวลาคิด การดีบักก็เหมือนกับวิทยาศาสตร์เชิงการทดลอง อย่างน้อย เราควรจะมีข้อสันนิษฐานบ้างว่า ปัญหาน่าจะคืออะไร ถ้ามีความเป็นไปได้หลายแบบ ลองคิดถึงการทดสอบของความเป็นไปได้แต่ละแบบ

แต่ แม้เทคนิคการดีบักที่ดีที่สุด ก็จะล้มเหลว ถ้าโปรแกรมมีข้อผิดพลาดมากเกินไป หรือ ถ้าโค้ดที่เรากำลังพยายามแก้ มันใหญ่เกินไป มันซับซ้อนเกินไป บางครั้งวิธีที่ดีที่สุดคือถอย ทำโปรแกรมให้ง่าย ๆ พื้น ๆ จนมันเริ่มทำงานได้ และเราเข้าใจการทำงานของมันก่อน

โปรแกรมเมอร์มือใหม่ มักจะลังเล ไม่อยากจะถอย เพราะว่า เขารับไม่ได้ที่จะต้องลบบรรทัดของโค้ดออก (ถึงแม้ มันจะผิดก็ตาม) ถ้ามันจะช่วยให้รู้สึกดีขึ้น อาจจะคัดลอกโปรแกรมไปอีกไฟล์หนึ่งก่อน แล้วค่อยเริ่มลบมันออก หลังจากนั้น ถ้าต้องการ ก็ค่อยสำเนาโค้ดที่เก็บไว้กลับมา โดยค่อย ๆ เอากลับมาทีละน้อย ๆ

เวลาหาบักที่ยาก ต้องทำทั้ง อ่าน รันและทดสอบ ไตร่ตรอง และบางครั้งก็ถอย ถ้าเทคนิคหนึ่งติด ให้ลองเทคนิคอื่น ๆ ที่เหลือ

13.11. อภิธานศัพท์

ลักษณะชี้เฉพาะ (deterministic): เกี่ยวกับที่ ถ้าให้อินพุตแบบเดิม โปรแกรมจะทำแบบเดิมทุกครั้งที่รัน

สุ่มเทียบ (pseudorandom): เกี่ยวกับลำดับของตัวเลขที่ดูเหมือนสุ่ม แต่จริง ๆ ถูกสร้างขึ้นจากโปรแกรมลักษณะชี้เฉพาะ

ค่าดีฟอลท์ หรือค่าโดยปริยาย (default value): ค่าที่ให้ไว้เป็นสำหรับพารามิเตอร์ ในกรณีที่อาร์กิวเมนต์ไม่ได้ให้ค่ามา

แทนที่ (override): เพื่อแทนที่ค่าดีฟอลท์ ด้วยค่าของอาร์กิวเมนต์

การวัดเปรียบเทียบสมรรถนะ (benchmarking): กระบวนการเลือกโครงสร้างข้อมูล โดยทำอิมพลิเมนเตชั่น เมื่อใช้โครงสร้างข้อมูลต่าง ๆ ที่ต้องการเลือก แล้วทดสอบกับตัวอย่างอินพุตต่าง ๆ ที่เป็นไปได้

การดีบักเป็ดยาง (rubber duck debugging): การดีบัก โดยอธิบายปัญหาที่เจอให้วัตถุ เช่น เป็ดยางฟัง. การอธิบายปัญหาให้ชัดเจนสามารถจะช่วยให้เราแก้ปัญหาได้ แม้ว่าเป็ดยางจะเขียนไพธอนไม่เป็น

13.12. แบบฝึกหัด

***แบบฝึกหัด* 13.9.** *"ลำดับที่" (rank) ของคำ คือ ตำแหน่งในลิสต์ของคำต่าง ๆ ที่เรียงลำดับตามความถี่ นั่นคือ คำที่พบบ่อยที่สุดจะเป็นลำดับที่หนึ่ง คำที่พบบ่อยที่สุดอันดับที่สอง จะเป็นลำดับที่สอง เป็นต้น*

กฎของซิปฟ์ (Zipf's law) อธิบายความสัมพันธ์ระหว่าง ลำดับที่ และความถี่ของคำในภาษาธรรมชาติ (`http://en.wikipedia.org/wiki/Zipf's_law`) นั่นคือ มันทำนายว่า ความถี่ f ของคำที่มีลำดับที่ r จะเป็น:

$$f = cr^{-s}$$

เมื่อ s และ c เป็นพารามิเตอร์ที่ขึ้นกับภาษาและข้อความ ถ้าเราใส่ลอการิทึมเข้าไปทั้งสองฝั่ง เราจะได้:

$$\log f = \log c - s \log r$$

ดังนั้น ถ้าเราวาดกราฟของ $\log f$ กับ $\log r$ เราจะได้เส้นตรง ที่มีความชันเป็น $-s$ และจุดตัดแกนเป็น $\log c$

เขียนโปรแกรมที่อ่านข้อความจากไฟล์ นับความถี่คำ พิมพ์ออกมา บรรทัดละหนึ่งคำ โดยเรียงลำดับตามความถี่ จากมากไปน้อย พร้อมค่า $\log f$ และ $\log r$ ใช้โปรแกรมวาดกราฟ ตามที่ชอบ เพื่อวาดกราฟของผลนี้ออกมา และตรวจสอบว่ามันเป็นเส้นตรงหรือไม่ จากผลลัพธ์ที่ได้ เราจะประมาณค่าของ s ได้หรือไม่?

เฉลย: `http://thinkpython2.com/code/zipf.py` *เพื่อจะรันโปรแกรมเฉลย เราต้องการโมดูลวาดกราฟ* `matplotlib` *ถ้าคุณติดตั้งไพธอนด้วย* `Anaconda` *คุณน่าจะมี* `matplotlib` *อยู่แล้ว ถ้าไม่ใช่อย่างนั้น คุณอาจจะต้องติดตั้งมันก่อน*

14. ไฟล์

บทนี้แนะนำแนวคิดของ โปรแกรม “คงอยู่” ที่เป็นข้อมูลไว้ใน*หน่วยเก็บ*ถาวร พร้อมแสดงวิธีใช้หน่วยเก็บถาวรต่าง ๆ เช่น ไฟล์ และฐานข้อมูล

14.1. ความคงอยู่

โปรแกรมเกือบทั้งหมดที่เราได้ดูกันไป เป็นลักษณะชั่วคราว คือ มันรันแล้วก็ให้เอาต์พุตออกมา แต่พอทำงานเสร็จ ข้อมูลก็หายไป ถ้าเรารันโปรแกรมใหม่ มันก็เริ่มต้นใหม่ทุกอย่าง

โปรแกรมบางอย่างเป็นลักษณะ**คงอยู่** (**persistent**) นั่นคือ มันรันนานมาก (หรืออาจจะรันตลอดเวลา) และมันเก็บข้อมูล หรืออย่างน้อยก็บางส่วนของข้อมูล ลงในหน่วยเก็บถาวร (เช่น ฮาร์ดดิสก์) แล้วถ้าปิดหรือเริ่มโปรแกรมพวกนี้ใหม่ มันจะไปเริ่มจากสถานะที่เก็บไว้ได้

ตัวอย่างของโปรแกรมคงอยู่ต่าง ๆ ก็คือ พวกระบบปฏิบัติการ ที่รันเกือบ ๆ ตลอดเวลาที่คอมพิวเตอร์เปิด และก็พวกเว็บเซิร์ฟเวอร์ที่รันตลอดเวลา คอยคำร้องที่จะมาจากเครือข่าย

วิธีหนึ่งที่ง่ายที่สุด ที่โปรแกรมจะเก็บรักษาข้อมูลของมันได้ คือ อ่านและเขียนไฟล์ข้อความ เราได้เห็นโปรแกรมที่อ่านไฟล์ข้อความไปแล้ว บทนี้ เราจะได้เห็นโปรแกรมที่เขียนไฟล์ข้อความด้วย

นอกจากไฟล์ข้อความแล้ว เราอาจจะเก็บสถานะของโปรแกรมไว้ใน*ฐานข้อมูล* (*database*) ก็ได้ บทนี้ เราจะดูฐานข้อมูลที่เรียบง่ายตัวหนึ่ง และดูโมดูล `pickle` ที่ช่วยให้เก็บข้อมูลของโปรแกรมได้ง่ายขึ้น

14.2. การอ่าน และการเขียน

ไฟล์ข้อความ (*text file*) เป็นลำดับของอักขระต่าง ๆ ที่เก็บในสื่อถาวระ เช่น ฮาร์ดดิสก์ หน่วยความจำแฟลช หรือซีดีรอม เราได้เห็นวิธีเปิดและอ่านไฟล์มาแล้ว จากหัวข้อ 9.1

เพื่อจะเขียนไฟล์ เราต้องเปิดมันด้วยโหมด `'w'` ที่ระบุด้วยพารามิเตอร์ที่สอง:

```
>>> fout = open('output.txt', 'w')
```

ถ้าไฟล์ที่เปิดมีอยู่แล้ว การไปเปิดมันในโหมดเขียน จะเท่าไปลบข้อมูลเก่าออกทั้งหมด และเริ่มต้นใหม่ เพราะฉะนั้นก็ระวังด้วย! ถ้าไฟล์ที่เปิดยังไม่มีอยู่ ไฟล์ใหม่จะถูกสร้างขึ้นมา

ฟังก์ชัน `open` ให้*ไฟล์อ็อบเจกต์* (*file object*) ออกมา โดยไฟล์อ็อบเจกต์นี้จะมีเมธอดต่าง ๆ ที่เอาไว้ใช้ทำงานกับไฟล์ เช่น เมธอด `write` ใช้เขียนข้อมูลเข้าไปในไฟล์

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

ค่าที่ส่งคืนออกมา เป็นจำนวนอักขระที่ได้เขียนเข้าไป. ไฟล์อ็อบเจกต์จะเก็บตำแหน่งว่ามันทำงานถึงตรงไหนในไฟล์ ดังนั้น ถ้าเราเรียก `write` อีกครั้ง มันจะเพิ่มข้อมูลใหม่ต่อเข้าไปที่ท้ายไฟล์

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

พอเขียนข้อมูลเข้าไปเสร็จหมดแล้ว เราก็ควรที่จะปิดไฟล์ซะ

```
>>> fout.close()
```

ถ้าเราไม่ได้ปิดไฟล์ มันก็จะปิดเองตอนที่โปรแกรมจบ

14.3. ตัวดำเนินการจัดรูปแบบ

อาร์กิวเมนต์ของ `write` ต้องเป็นข้อมูลแบบสายอักขระ ดังนั้น ถ้าเราต้องการใส่ค่าชนิดอื่น ๆ เข้าไปในไฟล์ เราต้องแปลงมันเป็นชนิดสายอักขระก่อน วิธีที่ง่ายที่สุดคือ ใช้ `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

อีกวิธีหนึ่งก็คือ การใช้ **ตัวดำเนินการจัดรูปแบบ** (format operator) คือ `%` เมื่อใช้กับตัวเลขจำนวนจริง ตัวดำเนินการ `%` ทำหน้าที่เป็นตัวดำเนินการมอดุลัส แต่ถ้า*ตัวถูกดำเนินการ*ตัวแรก เป็นสายอักขระ ตัวดำเนินการ `%` จะทำหน้าที่เป็นตัวดำเนินการจัดรูปแบบ

*ตัวถูกดำเนินการ*ตัวแรก เป็น**สายอักขระจัดรูปแบบ** (**format string**) ที่มี**ชุดจัดรูปแบบ** (**format sequence**) ตั้งแต่หนึ่งหรือมากกว่า โดย *ชุดจัดรูปแบบ* จะระบุว่า *ตัวถูกดำเนินการ*ตัวที่สอง จะถูกนำไปจัดรูปแบบเป็นสายอักขระอย่างไร ผลลัพธ์ของการดำเนินการ คือสายอักขระ

ตัวอย่าง ชุดจัดรูปแบบ `'%d'` หมายถึง *ตัวถูกดำเนินการ*ตัวที่สอง ควรจะถูกจัดรูปแบบ สายอักขระเป็นเลขจำนวนเต็มฐานสิบ:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

ผลลัพธ์คือ สายอักขระ `'42'` ที่ไม่ใช่ค่าจำนวนเต็ม `42`

ชุดจัดรูปแบบจะอยู่ตรงไหนในสายอักขระก็ได้ ดังนั้น เราสามารถฝั่งค่าเข้าไปในประโยคได้:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

ถ้ามี*ชุดจัดรูปแบบ*อยู่ในสายอักขระมากกว่าหนึ่งชุด ตัวถูกดำเนินการที่สอง ต้องเป็นทูเพิล แต่ละ*ชุดจัดรูปแบบ* จะคู่กับแต่ละอิลิเมนต์ของทูเพิล ตามลำดับ

ตัวอย่างต่อไปนี้ ใช้ชุด `'%d'` เพื่อจัดรูปแบบสายอักขระเป็นตัวเลขจำนวนเต็ม ชุด `'%g'` ใช้จัดรูปแบบเป็นเลขทศนิยม และชุด `'%s'` ใช้จัดรูปแบบ เป็นสายอักขระ:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

จำนวนอิลิเมนต์ในทูเพิลต้องเท่ากับจำนวนชุดจัดรูปแบบในสายอักขระจัดรูปแบบ และชนิดของอิลิเมนต์ ก็ยังต้องถูกชนิดกับชุดจัดรูปแบบที่คู่กันด้วย

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

ในตัวอย่างแรก จำนวนอิลิเมนต์ของทูเพิลมีไม่พอ ส่วนในตัวอย่างที่สอง อิลิเมนต์ผิดชนิด

สำหรับรายละเอียดเพิ่มเติมของตัวดำเนินการจัดรูปแบบ ดู `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting` วิธีจัดรูปแบบที่ดีกว่านี้คือ การใช้เมธอด `format`

ของสายอักขระ ซึ่งสามารถศึกษาได้จาก `https://docs.python.org/3/library/stdtypes.html#str.format`

14.4. ชื่อไฟล์และเส้นทาง

ไฟล์ต่าง ๆ ถูกจัดระเบียบเข้าไปอยู่ใน **สารบบ**ต่าง ๆ (**directories**) ซึ่งจะเรียก “โฟลเดอร์” (folder) ก็ได้ ทุก ๆ โปรแกรมที่รันอยู่ จะมี “สารบบปัจจุบัน” (current directory) ที่เป็นสารบบดีฟอลท์สำหรับการดำเนินการเกือบทั้งหมด ตัวอย่าง เวลาที่เราเปิดไฟล์เพื่ออ่าน ไพธอนจะหาไฟล์ใน*สารบบปัจจุบัน*

โมดูล `os` มีฟังก์ชันต่าง ๆ ที่ใช้ทำงานกับไฟล์และสารบบได้ (“os” ย่อมาจาก “operating system” ซึ่งหมายถึง ระบบปฏิบัติการ) ฟังก์ชัน `os.getcwd` จะให้ชื่อของสารบบปัจจุบันออกมา:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

ส่วนของชื่อ `cwd` ย่อจาก “current working directory” หมายถึง สารบบที่ทำงานอยู่ปัจจุบัน ผลลัพธ์ของตัวอย่างนี้คือ `/home/dinsdale` ที่เป็นสารบบบ้านของผู้ใช้ ชื่อ `dinsdale`

สายอักขระ เช่น `'/home/dinsdale'` ที่ระบุไฟล์หรือสารบบ จะเรียกว่า **เส้นทาง** (**path**).

ชื่อไฟล์ง่าย ๆ เช่น `memo.txt` ก็ถือว่าเป็น*เส้นทาง*ด้วยเหมือนกัน แต่อันนี้เรียกว่าเป็น **เส้นทางสัมพัทธ์** (**relative path**) เพราะว่ามันสัมพันธ์กับสารบบปัจจุบัน ถ้าสารบบปัจจุบัน คือ `/home/dinsdale` ชื่อไฟล์ `memo.txt` จะหมายถึง `/home/dinsdale/memo.txt`

สำหรับบางระบบปฏิบัติการ เช่น ยูนิกซ์ *เส้นทาง*ที่เริ่มด้วย `/` จะไม่ขึ้นกับสารบบปัจจุบัน เส้นทางแบบนี้ จะเรียกว่า **เส้นทางสัมบูรณ์** (**absolute path**) สำหรับระบบปฏิบัติการวินโดวส์ เส้นทางสัมบูรณ์ จะเริ่มต้นด้วยตัวอักษรหนึ่งตัว และตามด้วยเครื่องหมาย `:` เช่น `D:\\Users\\dinsdale\\memo.txt` เพื่อหาเส้นทางสัมบูรณ์ของไฟล์ เราสามารถใช้ `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

โมดูล `os.path` มีฟังก์ชันอื่น ๆ ที่ใช้ทำงานกับชื่อไฟล์และเส้นทางต่าง ๆ ตัวอย่าง ฟังก์ชัน `os.path.exists` ใช้ตรวจสอบว่า ไฟล์หรือสารบบมีอยู่หรือไม่:

```
>>> os.path.exists('memo.txt')
True
```

ถ้ามีอยู่ ลองใช้ `os.path.isdir` ตรวจสอบดูว่า มันเป็นสารบบหรือไม่:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

คล้าย ๆ กัน ฟังก์ชัน `os.path.isfile` ใช้ตรวจสอบว่า มันเป็นไฟล์หรือไม่

ฟังก์ชัน `os.listdir` จะให้ ลิสต์ของชื่อไฟล์ และสารบบต่าง ๆ ที่อยู่ภายใต้สารบบที่เป็นอาร์กิวเมนต์ออกมา:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

เพื่อสาธิตการใช้งานฟังก์ชันเหล่านี้ ตัวอย่างต่อไปนี้ ท่องเข้าไปในสารบบ แล้วพิมพ์ชื่อของทุกไฟล์ออกมา และเวียนเรียกตัวเองซ้ำสำหรับทุก ๆ สารบบภายใน

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

ฟังก์ชัน `os.path.join` รับชื่อสารบบ และชื่อไฟล์ แล้วนำไปต่อกัน เพื่อเป็นเส้นทางที่สมบูรณ์

โมดูล `os` มีฟังก์ชันชื่อ `walk` ที่คล้าย ๆ กับ ฟังก์ชันนี้ แต่ทำงานได้หลากหลายกว่า. เพื่อเป็นการฝึก ลองอ่านเอกสาร และใช้ฟังก์ชัน `os.walk` พิมพ์ ชื่อของไฟล์และสารบบย่อยต่าง ๆ ของสารบบที่ระบุ เฉลยสามารถดาวน์โหลดได้จาก `http://thinkpython2.com/code/walk.py`

14.5. การจับเอ็กเซ็ปชั่น

เวลาที่เราอ่านหรือเขียนไฟล์ อาจมีปัญหาเกิดขึ้นได้หลายอย่าง ถ้าเราพยายามจะเปิดไฟล์ที่ไม่มีอยู่ เราจะได้ IOError:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

ถ้าเราไม่มี*สิทธิเข้าใช้* (*permission*) หรือไม่ได้รับอนุญาตในการเข้าถึงไฟล์ เราจะเห็น

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

หรือ ถ้าเราไปเปิด*สารบบ* (*directory*) เพื่อจะอ่าน เราจะได้

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

เพื่อไม่ให้เกิดข้อผิดพลาดแบบนี้ เราควรใช้ฟังก์ชัน เช่น `os.path.exists` และ `os.path.isfile` แต่มันก็ใช้ทั้งเวลาและโค้ดจำนวนมากในการตรวจสอบทุกอย่างที่อาจจะผิดพลาดได้ (ถ้า `Errno 21` จะบอกอะไรสักอย่าง มันบอกว่า มีอย่างน้อย 21 อย่างที่อาจจะผิดพลาดได้)

มันดีกว่าที่เราจะลองทำไปเลย (และค่อยไปจัดการกับปัญหา ถ้ามันเกิด) ซึ่งนึ่งคือสิ่งที่ข้อความคำสั่ง `try` ทำ. ไวยากรณ์ของ `try` จะคล้าย ๆ กับไวยากรณ์ของ `if...else` เช่น

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

ไพธอนจะเริ่มด้วยการรันคำสั่งต่าง ๆ ในส่วนของ `try` ถ้าทุก ๆ อย่างไปได้ดี ไพธอนจะข้ามส่วนของคำสั่ง `except` ไปทำคำสั่งหลังจากนั้น แต่ถ้ามีข้อผิดพลาด หรือมี*เอ็กเซ็ปชั่น* (*exception*) เกิดขึ้น ไพธอนจะออกจากส่วนของ `try` และไปรันคำสั่งในส่วนของ `except`

การจัดการกับ*เอ็กเซ็ปชั่น* ด้วยคำสั่ง `try` จะเรียกว่า **การจับเอ็กเซ็ปชั่น** (**catching exception**). ในตัวอย่างนี้ ตัวคำสั่งในส่วน `except` ทำการพิมพ์ข้อความแจ้งข้อผิดพลาด ที่ไม่มีรายละเอียดอะไรมาก. โดยทั่วไปแล้ว การจับเอ็กเซ็ปชั่นจะให้โอกาสเราในการแก้ปัญหา หรือจะลองทำอีกครั้ง หรืออย่างน้อยก็ให้โอกาสเราได้จบโปรแกรมไปอย่างเรียบร้อย

14.6. ฐานข้อมูล

ฐานข้อมูล (**database**) เป็นไฟล์ที่ได้รับการจัดระเบียบไว้สำหรับการเก็บข้อมูล. ฐานข้อมูลหลาย ๆ ชนิดถูกจัดในลักษณะคล้าย ๆ กับดิกชันนารี ในแง่ที่ว่า มันจะแปลงจากกุญแจไปสู่ค่าที่คู่กับกุญแจ จุดต่างที่สำคัญที่สุดระหว่างฐานข้อมูลกับดิกชันนารี อยู่ที่ฮาร์ดดิสก์ (หรืออาจจะเป็นแหล่งเก็บข้อมูลถาวรอื่น ๆ) นั่นคือ ข้อมูลในฐานข้อมูลจะยังคงอยู่ได้ แม้หลังจากโปรแกรมจบไปแล้ว

โมดูล `dbm` มีฟังก์ชันสำหรับสร้างและแก้ไขไฟล์ฐานข้อมูลให้ ตัวอย่างเช่น เราจะสร้างฐานข้อมูลเพื่อเก็บคำบรรยายไฟล์รูปภาพต่าง ๆ

การเปิดฐานข้อมูล ก็คล้ายกับการเปิดไฟล์อื่น ๆ:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

อาร์กิวเมนต์ `'c'` บอก*ภาวะ* (*mode*) ว่า ฐานข้อมูลควรจะถูกสร้างขึ้นมา ถ้ามันยังไม่มีอยู่ ผลลัพธ์คืออ็อบเจกต์ฐานข้อมูล ที่สามารถจะถูกใช้ได้คล้ายกับดิกชันนารี (สำหรับการดำเนินการต่าง ๆ ส่วนใหญ่)

เมื่อเราสร้างรายการขึ้นมาใหม่ `dbm` ก็จะปรับปรุงไฟล์ฐานข้อมูลไปด้วย

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

เมื่อเราเข้าถึงรายการใดก็ตาม `dbm` ก็ต้องอ่านไฟล์:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

ผลลัพธ์จะออกมาเป็น **ไบต์อ็อบเจกต์** (**bytes object**) ที่ระบุโดยมี `b` ขึ้นต้น ไบต์อ็อบเจกต์ จะคล้ายกับสายอักขระในหลาย ๆ แง่. เวลาที่เราศึกษาไพธอนลึกลงไป ความแตกต่างระหว่างไบต์อ็อบเจกต์กับสายอักขระจะสำคัญมาก แต่ตอนนี้ เรายังไม่ต้องไปสนใจมันก่อน

ถ้าเรากำหนดค่าใหม่ให้กับกุญแจเดิม `dbm` ก็จะเอาค่าใหม่ไปเขียนทับค่าเดิม:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

เมธอดของดิกชันนารีบางเมธอด เช่น `keys` และ `items` ใช้ไม่ได้กับอ็อบเจกต์ฐานข้อมูล แต่การวนซ้ำด้วยลูป `for` ใช้ได้อยู่:

```
for key in db:
    print(key, db[key])
```

เหมือนกับไฟล์อื่น ๆ เราควรจะปิดฐานข้อมูลเมื่อเราใช้งานเสร็จ:

```
>>> db.close()
```

14.7. การทำพิกเกิล

ข้อจำกัดของ `dbm` คือ ทั้งกุญแจและค่าคู่กุญแจ จะต้องเป็นสายอักขระ หรือไบต์อ็อบเจกต์. ถ้าเราลองใช้ข้อมูลชนิดอื่น เราจะได้เห็นข้อความแจ้งข้อผิดพลาดออกมา

โมดูล `pickle` สามารถช่วยได้ มันจะแปลงข้อมูลชนิดอื่น ๆ เกือบทุกชนิด ไปเป็นสายอักขระ เพื่อให้เหมาะกับการเก็บในฐานข้อมูล (เรียกว่า เป็นการทำพิกเกิล ซึ่งมาจาก pickling ในภาษาอังกฤษที่หมายถึงการดอง) และ โมดูล `pickle` ก็สามารถช่วยแปลงจากสายอักขระเหล่านั้นกลับมาเป็นข้อมูลเดิม เวลาเรียกใช้ภายหลัง

ฟังก์ชัน `pickle.dumps` รับอ็อบเจกต์ชนิดใด ๆ เป็นพารามิเตอร์ และส่งคืนสายอักขระที่เป็นตัวแทนออกมาให้ (`dumps` ย่อมาจาก “dump string”):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

รูปแบบที่แปลงออกมา อาจจะดูยากสำหรับคนอ่าน แต่มันออกแบบมาให้ง่ายสำหรับ `pickle` ที่จะอ่านและตีความ ฟังก์ชัน `pickle.loads` (“load string”) สร้างอ็อบเจกต์ขึ้นมาใหม่:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

อ็อบเจกต์ใหม่ที่ได้ ถึงแม้จะมีค่าเหมือนกับอ็อบเจกต์เดิม แต่ มันเป็นอ็อบเจกต์คนละตัวกัน:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

พูดอีกอย่างก็คือ การทำพิกเกิล (ใช้ `pickle.dump`) แล้วทำย้อนพิกเกิล (ใช้ `pickle.load`) จะให้ผลเหมือนกับการทำสำเนาของอ็อบเจกต์

เราสามารถใช้ `pickle` เพื่อเก็บข้อมูลที่ไม่ใช่สายอักขระ เข้าไปในฐานข้อมูลได้. จริง ๆ แล้ว การใช้งาน `pickle` ร่วมกับ `dbm` นี้ เป็นที่นิยมมาก จนกระทั่งมีการรวมกันเป็นโมดูล ชื่อ `shelve`

14.8. ไปป์

ระบบปฏิบัติการส่วนใหญ่ จะมีส่วนที่ให้ผู้ใช้สามารถสั่งงานได้โดยการพิมพ์คำสั่ง (command-line interface) ซึ่งมักจะเรียกว่า **เชลล์** (**shell**) เชลล์ จะมีคำสั่งต่าง ๆ ที่สามารถใช้ท่องระบบไฟล์ และสั่งเปิดโปรแกรมต่าง ๆ ได้ ตัวอย่างเช่น ในระบบปฏิบัติการยูนิกซ์ เราสามารถเปลี่ยนสารบบได้ด้วย คำสั่ง `cd` เราสามารถแสดงสิ่งที่อยู่ในสารบบได้ด้วยคำสั่ง `ls` และเราสามารถที่จะสั่งเปิดเว็บเบราว์เซอร์โดยพิมพ์ (ตัวอย่างเช่น) `firefox`

โปรแกรมที่เราสามารถเปิดได้ด้วยเชลล์ ก็สามารถเปิดได้ด้วยไพธอน โดยใช้ **ไปป์อ็อบเจกต์** (**pipe object**)

ตัวอย่างเช่น คำสั่งยูนิกซ์ `ls -l` โดยทั่วไปจะแสดงไฟล์ต่าง ๆ ภายใต้สารบบปัจจุบัน โดยแสดงในแบบยาว (มีรายละเอียด) เราสามารถสั่ง `ls` ได้จาก `os.open`[1]

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

อาร์กิวเมนต์ เป็นสายอักขระที่มีคำสั่งของเชลล์อยู่ ค่าที่ให้กลับออกมาเป็นอ็อบเจกต์ ที่คล้าย ๆ กับการเปิดไฟล์ เราสามารถอ่านเอาต์พุตจาก `ls` แบบทีละบรรทัดได้ด้วย `readline` หรือจะอ่านทีเดียวทั้งหมดด้วย `read`:

```
>>> res = fp.read()
```

ถ้าทำงานเสร็จแล้ว ก็ปิดไปป์เหมือนกับที่ปิดไฟล์:

```
>>> stat = fp.close()
>>> print(stat)
None
```

[1] ตอนนี้ ฟังก์ชัน `popen` ถูกประกาศเป็น “deprecated” (เก่าและแนะนำให้เลิกใช้) ซึ่ง เราก็ควรจะหยุดใช้มัน และควรจะใช้โมดูล `subprocess` ที่ถูกแนะนำให้ใช้แทน แต่สำหรับกรณีง่าย ๆ ผม (อัลเลน ดาวนี่) พบว่า `subprocess` มันซับซ้อนเกินความจำเป็น ดังนั้น ผมจึงยังใช้ `popen` จนกว่าไพธอนจะไม่มีมันให้ใช้อีก

ค่าที่ส่งคืนออกมา จะบอกสถานะสุดท้ายของการทำคำสั่ง `ls` และ None หมายถึงว่า โปรแกรมจบการทำคำสั่งแบบปกติ คือไม่มีข้อผิดพลาด

ตัวอย่าง ระบบยูนิกซ์ส่วนใหญ่จะมีคำสั่ง `md5sum` มาให้ด้วย. คำสั่ง `md5sum` อ่านเนื้อหาในไฟล์ และคำนวณค่า "เช็คซัม" (checksum) แบบ *เอมดีห้า* (*MD5*) ออกมา. สามารถอ่านเรื่อง*เอมดีห้า*ได้จาก `http://en.wikipedia.org/wiki/Md5`. คำสั่งนี้เป็นวิธีที่สะดวกในการตรวจสอบว่าไฟล์สองไฟล์มีเนื้อหาเหมือนกัน ความน่าจะเป็นที่เนื้อหาที่ต่างกันจะให้ค่า*เช็คซัม*ออกมาเหมือนกันจะมีค่าน้อยมาก (นั่นคือ โอกาสที่จะซ้ำไม่น่าจะเกิดขึ้นก่อนที่เอกภพจะล่มสลาย)

เราสามารถใช้ไปป์เพื่อรัน `md5sum` จากไพธอนได้ และเราจะได้ผลลัพธ์เป็น:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9. การเขียนโมดูล

ไฟล์ที่มีโค้ดของไพธอน สามารถที่จะนำเข้ามาเป็นโมดูลได้ ตัวอย่างเช่น ถ้าเรามีไฟล์ชื่อ `wc.py` ที่มีโค้ดดังนี้:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

ถ้าเรารันโปรแกรมนี้ โปรแกรมจะอ่านเนื้อหาไฟล์ของตัวมันเอง แล้วพิมพ์จำนวนบรรทัดในไฟล์ เป็น 7 ออกมา เราสามารถนำเข้าโปรแกรมได้แบบนี้:

```
>>> import wc
7
```

ตอนนี้เรามีโมดูลอ็อบเจกต์ (module object) `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

โมดูลอ็อบเจกต์จะมี `linecount`:

```
>>> wc.linecount('wc.py')
7
```

ตอนนี้เราก็รู้วิธีเขียนโมดูลของไพธอนแล้ว

ปัญหาเดียวที่เห็นในตัวอย่างนี้คือ เมื่อเรานำเข้าโมดูล มันจะรันโค้ดทดสอบที่เขียนอยู่ไว้ด้วย เหมือนที่เราเห็น 7 พิมพ์ออกมา. โดยทั่วไปแล้ว ตอนที่เรานำเข้าโมดูล มันจะแค่นิยามฟังก์ชันใหม่ แต่ยังไม่รันมัน

โปรแกรมที่เขียนเพื่อที่จะถูกนำเข้าเป็นโมดูล ส่วนใหญ่จะเขียนด้วยรูปแบบต่อไปนี้:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

ตัวแปร `__name__` เป็นตัวแปรสำเร็จรูปที่ไพธอนจะสร้างขึ้นเมื่อเริ่มโปรแกรม ถ้าโปรแกรมถูกรันโดยตรง ตัวแปร `__name__` จะมีค่าเป็น `'__main__'` ซึ่งในกรณีนั้น โค้ดทดสอบจะถูกรัน ในกรณีที่โมดูลถูกนำเข้า ค่าของตัวแปร `__name__` จะไม่เท่ากับ `'__main__'` และโค้ดทดสอบจะไม่ถูกรัน

เพื่อเป็นแบบฝึกหัด พิมพ์ตัวอย่างนี้ในไฟล์ ชื่อ `wc.py` แล้วลองรันมันโดยตรง (รันสคริปต์ ได้แก่ สั่ง `python wc.py` ที่เชลล์). ลองนำเข้ามัน แล้วตรวจสอบค่าของตัวแปร `__name__` ของโมดูล `wc`

คำเตือน ถ้าเรานำเข้าโมดูลที่ได้นำเข้ามาก่อนแล้ว (โมดูล ชื่อเดียวกัน) ไพธอนจะไม่ทำอะไร มันจะไม่อ่านไฟล์มาใหม่ ถึงแม้ว่าเราได้แก้ไขไฟล์นั้นไปแล้ว

ถ้าเราต้องการนำเข้าโมดูลใหม่ เราสามารถใช้ฟังก์ชันสำเร็จรูป `reload` ได้ แต่มันอาจจะใช้ยากนิดหน่อย วิธีที่ง่ายที่สุด คือ ปิดแล้วเปิดอินเตอร์พรีเตอร์ใหม่ แล้วนำเข้าโมดูลอีกครั้ง

14.10. การดีบัก

เวลาที่เราอ่านหรือเขียนไฟล์ เราอาจจะเจอปัญหากับช่องว่าง ปัญหาเหล่านี้จะดีบักได้ยาก เพราะว่าเรามองตัวช่องว่างเองไม่เห็น ไม่ว่าจะเป็นช่องว่างแบบ การเว้น การย่อหน้า หรือการขึ้นบรรทัดใหม่ เรามองไม่เห็นมันโดยตรง เราเห็นมันผ่านตัวอื่นรอบ ๆ ข้าง

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

ฟังก์ชันสำเร็จรูป `repr` อาจช่วยได้ มันจะรับอ็อบเจกต์เป็นอาร์กิวเมนต์ และส่งคืนตัวแทนสายอักขระของอ็อบเจกต์นั้นออกมา. สำหรับสายอักขระ มันแทนช่องว่างต่าง ๆ ด้วย*ชุดแบคสแลช* (*backslash sequences*):

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

อันนี้อาจจะเป็นมีประโยชน์ตอนดีบัก

อีกปัญหาที่อาจจะเจอ คือ คอมพิวเตอร์ที่ต่างระบบกันอาจจะใช้อักขระที่ระบุการขึ้นบรรทัดใหม่ต่างกัน บางระบบใช้การขึ้นบรรทัดใหม่ แทนด้วย `\n`. บางระบบใช้ `\r`. บางระบบใช้ทั้งคู่ ถ้าเราย้ายไฟล์ข้ามระบบ ความต่างนี้อาจจะสร้างปัญหาให้ได้

ระบบคอมพิวเตอร์ส่วนใหญ่ มีเครื่องมือช่วยแปลงจากรูปแบบหนึ่งไปแบบอื่นได้ ลองหาเครื่องมือพวกนี้ (และอ่านเพิ่มเติมเกี่ยวกับประเด็นนี้) ที่ `http://en.wikipedia.org/wiki/Newline` หรือ แน่นอนว่า เราอาจจะลองเขียนโปรแกรมแปลงรูปแบบพวกนี้ขึ้นมาเองก็ได้

14.11. อภิธานศัพท์

คงอยู่ (persistent): เกี่ยวกับ โปรแกรมที่เก็บข้อมูล (อย่างน้อยก็บางส่วน) ไว้ในแหล่งเก็บข้อมูลถาวร

ตัวดำเนินการจัดรูปแบบ (format operator): ตัวดำเนินการ `%` ที่รับสายอักขระจัดรูปแบบและทูเพิล แล้วสร้างสายอักขระที่รวมอิลิเมนต์ต่าง ๆ ของทูเพิลเข้าไป ในรูปแบบที่ระบุด้วย*สายอักขระจัดรูปแบบ*

สายอักขระจัดรูปแบบ (format string): สายอักขระที่ใช้กับตัวดำเนินการจัดรูปแบบ และมีชุดจัดรูปแบบอยู่

ชุดจัดรูปแบบ (format sequence): ชุดลำดับของอักขระ เช่น **%d** ที่ใช้ระบุว่าค่าตัวเลขควรจะถูกจัดรูปแบบการแสดงผลอย่างไร

ไฟล์ข้อความ (text file): เป็นลำดับของตัวอักษรที่เก็บในแหล่งเก็บข้อมูลถาวร เช่น ฮาร์ดดิสก์

สารบบ (directory): ชื่อของหมวดหมู่ของไฟล์ต่าง ๆ หรืออาจเรียกว่า โฟลเดอร์ (folder)

เส้นทาง (path): สายอักขระที่ใช้ระบุไฟล์ รวมถึงสารบบต่าง ๆ ที่ไฟล์ถูกจัดอยู่

เส้นทางสัมพัทธ์ (relative path): เส้นทางที่เริ่มจากสารบบปัจจุบันที่ทำงานอยู่

เส้นทางสัมบูรณ์ (absolute path): เส้นทางที่เริ่มจากสารบบบนสุด ในระบบคอมพิวเตอร์

จับเอ็กเซ็ปชั่น (catch exceptions): การใช้คำสั่ง `try` และ `except` ช่วยเพื่อป้องกันไม่ให้เอ็กเซ็ปชั่นหรือข้อผิดพลาดปิดโปรแกรมไปเอง

ฐานข้อมูล (database): ไฟล์ที่เนื้อหาถูกจัดระเบียบในลักษณะคล้ายกับดิกชันนารี ที่มีกุญแจคู่กับค่าของกุญแจ

ไบต์อ็อบเจกต์ (bytes object): อ็อบเจกต์ที่คล้ายกับสายอักขระ

เชลล์ (shell): โปรแกรมที่ให้ผู้ใช้สามารถพิมพ์คำสั่งต่าง ๆ รวมถึงการเรียกรันโปรแกรมอื่น ๆ ผ่านการพิมพ์คำสั่ง

ไปป์อ็อบเจกต์ (pipe object): อ็อบเจกต์ที่เป็นตัวแทนโปรแกรมที่กำลังรันอยู่ และยอมให้โปรแกรมไพธอนรันคำสั่งของเชลล์และอ่านผลลัพธ์ได้

14.12. แบบฝึกหัด

***แบบฝึกหัด* 14.1.** *จงเขียนฟังก์ชัน ชื่อ `sed` ที่รับอาร์กิวเมนต์สี่ตัว คือ สายอักขระรูปแบบ สายอักขระแทนที่ และชื่อไฟล์สองชื่อ. ฟังก์ชันอ่านไฟล์แรก และเขียนเนื้อหาของไฟล์แรกลงในไฟล์ที่สอง (อาจจะสร้างไฟล์ที่สอง ถ้าจำเป็น) ถ้ามีสายอักขระรูปแบบปรากฏอยู่ในเนื้อหาของไฟล์ แทนมันด้วยสายอักขระแทนที่*

ถ้ามีข้อผิดพลาดเกิดขึ้นระหว่าง เปิดไฟล์ อ่านไฟล์ เขียนไฟล์ หรือปิดไฟล์ โปรแกรมควรจะสามารถจับเอ็กเซ็ปชั่น แล้วพิมพ์ข้อความแจ้งข้อผิดพลาดและปิดโปรแกรม เฉลย: `http://thinkpython2.com/code/sed.py`

แบบฝึกหัด **14.2.** *ถ้าคุณดาวน์โหลดเฉลยของแบบฝึกหัด 12.2 จาก* `http://thinkpython2.com/code/anagram_sets.py` *คุณจะเห็นว่ามันสร้างดิกชันนารีที่แปลงจากสายอักขระของตัวอักษรที่เรียงตามลำดับไปหาลิสต์ของคำต่าง ๆ ที่สามารถสะกดได้ด้วยตัวอักษรเหล่านั้น ตัวอย่างเช่น* `'opst'` *แปลงไปเป็นลิสต์* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`

จงเขียนโมดูลที่นำเข้า `anagram_sets` *และมีสองฟังก์ชันใหม่ ได้แก่ ฟังก์ชัน* `store_anagrams` *ควรจะมีดิกชันนารีของคำสลับอักษรเก็บไว้ และฟังก์ชัน* `read_anagrams` *เมื่อรับคำมา ควรจะค้นหาคำนั้น และส่งคืนลิสต์ของคำสลับอักษรของคำ ๆ นั้น เฉลย:* `http://thinkpython2.com/code/anagram_db.py`

แบบฝึกหัด **14.3.** *ในการเก็บไฟล์เอ็มพีสาม (MP3 files) จำนวนมาก ๆ มันอาจจะมีไฟล์ของเพลงซ้ำกันโดยเก็บที่สารบบต่างกัน หรือใช้ชื่อต่างกัน จุดประสงค์หลัก คือค้นหาเนื้อหาที่ซ้ำกัน*

1. *จงเขียนโปรแกรมที่ค้นหาสารบบ และสารบบย่อยทั้งหมด ที่มีไฟล์เอ็มพีสามอยู่ (ไฟล์ที่ลงท้ายด้วย* `.mp3`*) แล้วส่งคืนเส้นทางทั้งหมดออกมาเป็นลิสต์ คำใบ้: โมดูล* `os.path` *มีฟังก์ชันหลาย ๆ อัน ที่มีประโยชน์ในการจัดการชื่อไฟล์และเส้นทาง*

2. *เพื่อตรวจสอบว่าไฟล์ซ้ำกัน เราสามารถใช้* `md5sum` *เพื่อคำนวณหาค่า "เช็คซัม" (checksum) ของแต่ละไฟล์ ถ้าสองไฟล์มีค่าเช็คซัมเหมือนกัน สองไฟล์นั้นก็น่าจะซ้ำกัน*

3. *เพื่อตรวจสอบอีกที เราสามารถใช้คำสั่งขอยูนิกซ์* `diff` *ตรวจไฟล์ที่คิดว่าซ้ำกันอีกทีได้*

เฉลย: `http://thinkpython2.com/code/find_duplicates.py`.

15. คลาสและอ็อบเจกต์

มาถึงจุดนี้คุณได้รู้จักวิธีการใช้งานฟังก์ชันเพื่อจัดระเบียบโค้ด และรู้วิธีใช้ชนิดข้อมูลสำเร็จรูปเพื่อจัดระเบียบข้อมูล ขั้นต่อไปเป็นการเรียนรู้ “การเขียนโปรแกรมเชิงวัตถุ” ซึ่งจะใช้ชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดเองเพื่อจัดระเบียบได้ทั้งโค้ดและข้อมูล การเขียนโปรแกรมเชิงวัตถุเป็นเรื่องใหญ่ ที่เราจะต้องศึกษาเพิ่มอีกสองถึงสามบท

ตัวอย่างโปรแกรมของบทนี้สามารถดาวน์โหลดได้ที่ `http://thinkpython2.com/code/Point1.py` ผลเฉลยสำหรับแบบฝึกหัดอยู่ที่ `http://thinkpython2.com/code/Point1_soln.py`

15.1. ชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดเอง

เราได้ใช้ชนิดข้อมูลสำเร็จรูปของไพธอนมาหลายชนิดแล้ว คราวนี้เรามาลองสร้างชนิดข้อมูลใหม่ ตัวอย่างเช่น สร้างชนิดข้อมูลชื่อว่า `Point` ซึ่งใช้สำหรับแทนจุดในระนาบสองมิติ

สัญลักษณ์สำหรับแทนจุดในทางคณิตศาสตร์มักเขียนคู่ลำดับไว้ในวงเล็บและคั่นตัวเลขด้วยจุลภาค ตัวอย่างเช่น $(0, 0)$ แทนตำแหน่งต้นกำเนิด และ (x, y) ใช้แทนจุดที่ห่างไปทางขวา x หน่วย และสูงขึ้นไป y หน่วยจากจุดต้นกำเนิด

มีหลากหลายแนวทางที่อาจใช้แทนจุดในภาษาไพธอน

- เราสามารถเก็บค่าคู่ลำดับแยกกันในสองตัวแปร `x` และ `y`.
- เราสามารถเก็บค่าคู่ลำดับเป็นสมาชิกในลิสต์หรือในทูเพิล
- เราสามารถสร้างชนิดข้อมูลใหม่เพื่อใช้แทนจุดเป็นอ็อบเจกต์ (object)

การสร้างชนิดข้อมูลใหม่เป็นวิธีการที่ซับซ้อนกว่าวิธีการอื่น แต่ก็มีข้อดีอีกหลายประการดังจะได้เห็นต่อไป

ชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดขึ้นนี้มีชื่อเรียกอีกอย่างว่า **คลาส (class)** การนิยามคลาสมีลักษณะดังนี้

```
class Point:
    """Represents a point in 2-D space."""
```

ส่วนหัวบ่งชี้ว่าคลาสใหม่นี้มีชื่อว่า `Point` ส่วนเนื้อหาเป็นข้อความบรรยายที่ใช้อธิบายว่าคลาสนี้สร้างขึ้นเพื่ออะไร เราสามารถนิยามตัวแปรและเมธอดภายในส่วนนิยามคลาส แต่เราจะกลับมากล่าวถึงในภายหลัง

การนิยามคลาสชื่อ `Point` จะสร้าง **คลาสอ็อบเจกต์ (class object)** ขึ้น

```
>>> Point
<class '__main__.Point'>
```

เนื่องจาก `Point` ถูกนิยามในระดับบนสุด ดังนั้นจึงมี "ชื่อเต็ม" ว่า `__main__.Point`

คลาสอ็อบเจกต์เป็นเหมือนโรงงานสำหรับสร้างอ็อบเจกต์ เราเรียกใช้ `Point` เหมือนกับการเรียกฟังก์ชันเพื่อสร้างอ็อบเจกต์ Point

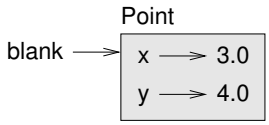
```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

ค่าที่ถูกส่งออกมาเป็นอ้างอิงไปยังอ็อบเจกต์ของ Point ซึ่งเรากำหนดค่าให้กับ `blank`.

การสร้างอ็อบเจกต์ใหม่ขึ้นมานี้จะเรียกว่า **การสร้างอินสแตนซ์ (instantiation)** และอ็อบเจกต์ที่ได้คือหนึ่ง **อินสแตนซ์ (instance)** ของคลาสนั้น

เมื่อคุณพิมพ์อินสแตนซ์ ไพธอนจะบอกว่ามันเป็นอินสแตนซ์ของคลาสใดและบอกว่าถูกเก็บไว้ที่ใดในหน่วยความจำ (ส่วนหน้า `0x` บอกให้รู้ว่า ตัวเลขที่ตามมานั้นเป็นเลขฐานสิบหก)

ทุกๆ อ็อบเจกต์เป็นหนึ่งอินสแตนซ์ของบางคลาส ดังนั้นการใช้ "อ็อบเจกต์" และ "อินสแตนซ์" สามารถใช้แทนกันได้ แต่ในบทนี้จะใช้ "อินสแตนซ์" เพื่อชี้ชัดว่ากำลังกล่าวถึงชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดขึ้น



รูปที่ 15.1.: แผนภาพอ็อบเจกต์

15.2. แอตทริบิวต์

เราสามารถกำหนดค่าให้กับอินสแตนซ์โดยการใช้สัญกรณ์จุด

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

ไวยากรณ์เช่นนี้เหมือนกับไวยากรณ์ที่ใช้ในการเลือกตัวแปรจากโมดูล ตัวอย่างเช่น `math.pi` หรือ `string.whitespace` ในกรณีนี้เราได้กำหนดค่าให้กับสมาชิกของอ็อบเจกต์ แต่อย่างไรก็ตาม สมาชิกเหล่านี้ถูกเรียกว่า **แอตทริบิวต์ (attributes)**

ออกเสียงเหมือนคำนาม “AT-trib-ute” เน้นเสียงของพยางค์แรก ตรงกันข้ามกับคำกริยา “a-TRIB-ute” ซึ่งจะเน้นเสียงของพยางค์ที่สอง

แผนภาพต่อไปนี้แสดงผลลัพธ์ของการกำหนดค่าเหล่านี้ แผนภาพสถานะที่แสดงอ็อบเจกต์และแอตทริบิวต์ของอ็อบเจกต์จะถูกเรียกว่า **แผนภาพอ็อบเจกต์ (object diagram)** ดูรูปที่ 15.1

ตัวแปร `blank` อ้างถึงอ็อบเจกต์ Point ซึ่งบรรจุสองแอตทริบิวต์ แต่ละแอตทริบิวต์อ้างอิงถึงตัวเลขทศนิยมอย่างละตัว

เราสามารถอ่านค่าของแอตทริบิวต์แต่ละตัวด้วยวิธีการเดียวกัน

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

นิพจน์ `blank.x` หมายถึง “ไปยังตำแหน่งที่อ้างถึงโดยอ็อบเจกต์ `blank` และอ่านค่าของแอตทริบิวต์ `x`” ในตัวอย่างจะเห็นว่า มีการกำหนดค่าให้กับตัวแปร `x` ซึ่งไม่ทำให้เกิดการขัดแย้งระหว่างตัวแปร `x` และแอตทริบิวต์ `x` แต่อย่างใด

คุณสามารถใช้สัญกรณ์จุดเป็นส่วนหนึ่งของนิพจน์ใด ๆ ได้ ตัวอย่างเช่น

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

เราสามารถส่งผ่านอินสแตนซ์เป็นอาร์กิวเมนต์ได้ตามปกติ ตัวอย่างเช่น

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` รับจุดหนึ่งจุดเข้ามาเป็นอาร์กิวเมนต์ แล้วแสดงค่าในรูปแบบสัญลักษณ์ทางคณิตศาสตร์ การเรียกใช้งาน เราสามารถใช้ `blank` เป็นอาร์กิวเมนต์ได้

```
>>> print_point(blank)
(3.0, 4.0)
```

ภายในฟังก์ชัน `p` เป็นสมนามของ `blank` ดังนั้นถ้าฟังก์ชันมีการแก้ไขค่าของ `p` ก็จะส่งผลต่อ `blank` เช่นกัน

เพื่อเป็นการฝึก ให้เขียนฟังก์ชันชื่อ `distance_between_points` ซึ่งรับจุดสองจุดเป็นอาร์กิวเมนต์และให้ค่าออกมาเป็นระยะห่างระหว่างสองจุดนั้น

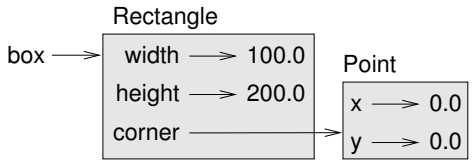
15.3. รูปสี่เหลี่ยม

บางครั้งก็มีความชัดเจนว่าอะไรควรเป็นแอตทริบิวต์ของอ็อบเจกต์ แต่ในบางคราวเราจะต้องตัดสินใจ ตัวอย่างเช่น ลองจินตนาการว่าเรากำลังออกแบบคลาสสำหรับรูปสี่เหลี่ยม อะไรเป็นแอตทริบิวต์ที่เราจะใช้เพื่อระบุตำแหน่งที่ตั้งและขนาดของรูปสี่เหลี่ยม? เราสามารถที่จะละเลยเรื่องของมุมเพื่อให้ง่ายขึ้นโดยสมมติว่ามีเฉพาะสี่เหลี่ยมในแนวตั้งและแนวนอน

มีอย่างน้อยสองแนวทางที่เป็นไปได้

- เราสามารถระบุมุมใดมุมหนึ่ง (หรือจุดศูนย์กลาง) ความกว้าง และความยาว
- เราสามารถระบุคู่จุดมุมตรงข้าม

ณ จุดนี้ ยังเป็นการยากที่จะบอกว่าวิธีการไหนจะเป็นวิธีการที่ดีกว่าวิธีอื่น ดังนั้นเราจะลองทำตามวิธีแรกเพื่อเป็นตัวอย่าง



รูปที่ 15.2.: แผนภาพอ็อบเจกต์

Here is the class definition: นี่เป็นนิยามของคลาสดังกล่าว

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

ด็อกสตริงได้ระบุรายการแอตทริบิวต์ไว้ด้วยคือ `width` และ `height` เป็นตัวเลข ส่วน `corner` เป็นอ็อบเจกต์ Point ที่ใช้ระบุตำแหน่งของมุมซ้ายล่าง

เพื่อให้ได้ตัวแทนของรูปสี่เหลี่ยมหนึ่งรูป เราจะต้องสร้างอินสแตนซ์ของคลาส Rectangle และกำหนดค่าต่าง ๆ ให้กับแอตทริบิวต์ เช่น

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

นิพจน์ `box.corner.x` หมายถึง “ไปยังตำแหน่งที่อ็อบเจกต์ `box` อ้างถึงและเลือกแอตทริบิวต์ชื่อว่า `corner` ซึ่งจะเป็นการไปยังอ็อบเจกต์นั้นและเลือกแอตทริบิวต์ชื่อ `x`”

รูปที่ 15.2 แสดงสถานะของอ็อบเจกต์ที่ได้ อ็อบเจกต์ที่ถูกกำหนดเป็นแอตทริบิวต์ของอ็อบเจกต์อื่นจะ **ฝังตัว** อยู่ภายในอีกที

15.4. อิสแสตนซ์เป็นค่าคืนกลับ

ฟังก์ชันสามารถส่งค่าออกมาเป็นอินสแตนซ์ได้ ตัวอย่างเช่น ฟังก์ชัน `find_center` รับอาร์กิวเมนต์เป็นอ็อบเจกต์ `Rectangle` แล้วให้ค่าออกมาเป็นอินสแตนซ์ของ `Point` ที่มีพิกัดตำแหน่งกึ่งกลางของรูปสี่เหลี่ยม

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

นี่เป็นตัวอย่างการส่ง `box` เป็นอาร์กิวเมนต์และกำหนดค่าผลลัพธ์ซึ่งเป็นอินสแตนซ์ Point ให้กับ `center`

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5. อ็อบเจกต์เป็นชนิดข้อมูลที่เปลี่ยนแปลงได้

เราสามารถเปลี่ยนแปลงสถานะของอ็อบเจกต์ได้ด้วยการกำหนดค่าใหม่ให้กับแอตทริบิวต์ของอ็อบเจกต์ ตัวอย่างเช่น การเปลี่ยนขนาดของรูปสี่เหลี่ยมโดยไม่ย้ายตำแหน่ง สามารถทำได้โดยการแก้ไขค่าของ `width` และ `height`

```
box.width = box.width + 50
box.height = box.height + 100
```

เราสามารถเขียนฟังก์ชันเพื่อแก้ไขอ็อบเจกต์ ตัวอย่างเช่น ฟังก์ชัน `grow_rectangle` ซึ่งรับอ็อบเจกต์ Rectangle และตัวเลขอีกสองตัวคือ `dwidth` และ `dheight` สำหรับใช้บวกเพิ่มให้กับความกว้างและความสูงของรูปสี่เหลี่ยม

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

นี่คือตัวอย่างที่แสดงให้เห็นถึงผลที่ได้

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

rect ที่อยู่ภายในฟังก์ชันเป็นสมนามของ box ดังนั้นเมื่อใดที่ฟังก์ชันมีการแก้ไขค่าของ rect ค่าของ box ก็จะเปลี่ยนตาม

เพื่อเป็นการฝึก ให้เขียนฟังก์ชันชื่อว่า move_rectangle ซึ่งรับอาร์กิวเมนต์เป็นอ็อบเจกต์ Rectangle และตัวเลขอีกสองจำนวนคือ dx และ dy ให้ฟังก์ชันทำการย้ายตำแหน่งของสี่เหลี่ยมด้วยบวกเพิ่ม dx ให้กับพิกัด x ของ corner และบวกเพิ่ม dy ให้กับพิกัด y ของ corner

15.6. การทำสำเนา

การมีสมนามทำให้การทำความเข้าใจโปรแกรมทำได้ยาก เนื่องจากการเปลี่ยนแปลงค่าในที่หนึ่งอาจส่งผลกระทบอย่างคาดไม่ถึงกับอีกที่หนึ่ง ซึ่งจะเป็นการยากที่จะติดตามค่าของทุกตัวแปรที่มีการอ้างถึงอ็อบเจกต์ที่กำหนด

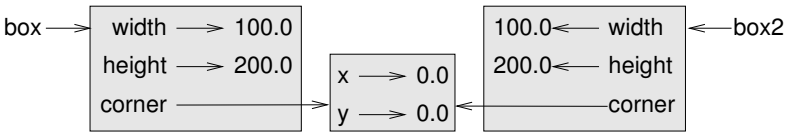
การทำสำเนาอ็อบเจกต์เป็นอีกทางเลือกแทนการทำสมนาม โมดูล copy มีฟังก์ชันชื่อว่า copy ซึ่งสามารถสร้างสำเนาของอ็อบเจกต์ใดก็ได้

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 และ p2 มีข้อมูลบรรจุภายในเหมือนกัน แต่เป็นคนละอ็อบเจกต์

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
```



รูปที่ 15.3.: แผนภาพอ็อบเจกต์

```
>>> p1 is p2
False
>>> p1 == p2
False
```

ตัวดำเนินการ `is` แสดงให้เห็นชัดว่า `p1` และ `p2` ไม่ใช่อ็อบเจกต์เดียวกันเป็นไปตามที่เราคาดหวัง แต่เราอาจจะคาดหวังว่าตัวดำเนินการ `==` ให้ผลลัพธ์เป็น `True` เนื่องจากทั้งสองจุดนี้มีข้อมูลที่เหมือนกัน ในกรณีนี้คุณอาจจะผิดหวังที่ได้รู้ว่า สำหรับอินสแตนซ์แล้วพฤติกรรมเริ่มต้นของตัวดำเนินการ `==` ให้ผลเหมือนกับตัวดำเนินการ `is` ซึ่งจะตรวจสอบว่าเป็นอ็อบเจกต์อันเดียวกันหรือไม่ ไม่ใช่ตรวจสอบว่ามีความเท่าเทียมกันหรือไม่ ทั้งนี้เป็นเพราะว่าสำหรับชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดขึ้นเองแล้วไพธอนไม่รู้ว่าควรจะตรวจสอบความเท่ากันอย่างไร อย่างน้อยก็ยัง

ถ้าเราใช้ `copy.copy` เพื่อสำเนาอ็อบเจกต์ Rectangle เราจะพบว่ามีการสำเนาเฉพาะ Rectangle แต่จะไม่สำเนา Point ที่ฝังตัวอยู่ใน Rectangle

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

รูปที่ 15.3 แสดงแผนภาพอ็อบเจกต์ให้เห็นว่ามีลักษณะอย่างไร การทำงานเช่นนี้เรียกได้ว่าเป็น **การสำเนาตื้น (shallow copy)** เนื่องจากทำการสำเนาเฉพาะตัวอ็อบเจกต์และทุกการอ้างอิง แต่ไม่สำเนาอ็อบเจกต์ที่ฝังอยู่ในอ็อบเจกต์นั้น

สำหรับแอปพลิเคชันส่วนใหญ่ไม่ได้ต้องการผลลัพธ์เช่นนี้ ในตัวอย่างนี้การเรียกใช้ `grow_rectangle` กับอ็อบเจกต์ Rectangle ใดจะไม่สร้างผลกระทบต่ออ็อบเจกต์อื่น แต่ถ้าเรียกใช้ `move_rectangle` จะส่งผลกระทบทั้งคู่ พฤติกรรมเช่นนี้สร้างความสับสนและมีข้อผิดพลาดง่าย

โชคดีที่โมดูล `copy` ได้เตรียมเมธอดชื่อ `deepcopy` ซึ่งคัดลอกไม่เฉพาะอ็อบเจกต์ที่ระบุแต่ยังคัดลอก

รวมไปถึงอ็อบเจกต์ที่ถูกอ้างถึงและอ็อบเจกต์อื่นที่ถูกอ้างถึงด้วย *อ็อบเจกต์เหล่านั้น* ไปเรื่อยๆ จึงไม่น่าแปลกใจที่กระบวนการนี้เรียกว่า **การสำเนาลึก (deep copy)**

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 และ box เป็นคนละอ็อบเจกต์ที่แยกจากกันอย่างสมบูรณ์

เพื่อเป็นการฝึก ให้เขียนฟังก์ชัน move_rectangle อีกเวอร์ชันที่สร้างอ็อบเจกต์ Rectangle และให้ค่าออกมาเป็นอ็อบเจกต์ใหม่ แทนการแก้ไขอ็อบเจกต์เดิม

15.7. การดีบัก

เมื่อเราเริ่มทำงานกับอ็อบเจกต์ เรามีแนวโน้มที่จะพบเอ็กเซ็ปชั่นใหม่ๆ ถ้าเราพยายามเข้าถึงแอตทริบิวต์ที่ไม่มีอยู่จริง เราจะพบเอ็กเซ็ปชั่น AttributeError

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

ถ้าเราไม่แน่ใจว่าอ็อบเจกต์นั้นเป็นชนิดใด เราสามารถสอบถาม

```
>>> type(p)
<class '__main__.Point'>
```

เรายังสามารถใช้ isinstance เพื่อตรวจสอบว่าอ็อบเจกต์นั้นเป็นอินสแตนซ์ของคลาสใดคลาสหนึ่งหรือไม่

```
>>> isinstance(p, Point)
True
```

ถ้าเราไม่แน่ใจว่าอ็อบเจกต์หนึ่งมีแอตทริบิวต์ที่สนใจหรือไม่ เราสามารถใช้ฟังก์ชันสำเร็จรูป hasattr ตรวจสอบได้

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

อาร์กิวเมนต์แรกคืออ็อบเจกต์ใด ๆ อาร์กิวเมนต์ที่สองเป็น *สายอักขระ* ที่มีชื่อของแอตทริบิวต์ที่ต้องการทราบ

นอกจากนี้เรายังสามารถใช้คำสั่ง `try` เพื่อหาดูว่าอ็อบเจกต์นั้นมีแอตทริบิวต์ที่เราต้องการหรือไม่

```
try:
    x = p.x
except AttributeError:
    x = 0
```

วิธีนี้สามารถช่วยทำให้การเขียนฟังก์ชันเพื่อทำงานกับหลากหลายชนิดข้อมูลได้ ดูรายละเอียดเพิ่มเติมในหัวข้อ 17.9

15.8. อภิธานศัพท์

คลาส (class): ชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดเอง การประกาศคลาสจะสร้างคลาสอ็อบเจกต์ใหม่ขึ้น

คลาสอ็อบเจกต์ (class object): อ็อบเจกต์ที่มีรายละเอียดของชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดขึ้นเอง คลาสอ็อบเจกต์สามารถใช้สร้างอินแสตนซ์ของชนิดข้อมูลนั้นได้

อินสแตนซ์ (instance): อ็อบเจกต์ที่เป็นสังกัดของคลาส

สร้างอินสแตนซ์ (instantiate): การสร้างอ็อบเจกต์ใหม่

แอตทริบิวต์ (attribute): ชื่อพร้อมกับค่าซึ่งผูกอยู่กับอ็อบเจกต์

อ็อบเจกต์ฝังตัว (embedded object): อ็อบเจกต์ที่ถูกเก็บเป็นแอตทริบิวต์ของอ็อบเจกต์อื่น

สำเนาตื้น (shallow copy): การสำเนาข้อมูลภายในอ็อบเจกต์รวมถึงแอตทริบิวต์ที่เป็นอ้างอิงไปยังอ็อบเจกต์ฝังตัว ทำได้โดยการเรียกใช้ฟังก์ชัน `copy` ที่อยู่ในโมดูล `copy`

สำเนาลึก (deep copy): การสำเนาข้อมูลภายในอ็อบเจกต์รวมทั้งข้อมูลของอ็อบเจกต์ที่เป็นอ็อบเจกต์ฝังตัวตลอดจนข้อมูลอ็อบเจกต์อื่นที่ถูกอ้างอิงด้วยอ็อบเจกต์ฝังตัวเหล่านั้นไปเรื่อยๆ ทำได้โดยการเรียกใช้ฟังก์ชัน `deepcopy` ที่อยู่ในโมดูล `copy`

แผนภาพอ็อบเจกต์ (object diagram): แผนภาพที่มีการแสดงอ็อบเจกต์กับแอตทริบิวต์ภายในและค่าของแต่ละแอตทริบิวต์

15.9. แบบฝึกหัด

***แบบฝึกหัด* 15.1.** *เขียนนิยามของคลาสชื่อ `Circle` ที่มีแอตทริบิวต์ `center` และ `radius` กำหนดให้ `center` เป็นอ็อบเจกต์ของ Point และให้ radius เป็นตัวเลข*

สร้างอินสแตนซ์ของ Circle ซึ่งเป็นตัวแทนของวงกลมซึ่งมีจุดศูนย์กลางอยู่ที่ $(150, 100)$ *รัศมี 75 หน่วย*

เขียนฟังก์ชันชื่อ `point_in_circle` ซึ่งรับอ็อบเจกต์ Circle และ Point แล้วให้ค่าคืนกลับเป็น True ถ้าจุดนั้นอยู่ภายในหรือบนขอบเขตของวงกลม

เขียนฟังก์ชันชื่อ `rect_in_circle` ซึ่งรับอ็อบเจกต์ Circle และ Rectangle แล้วให้ค่าคืนกลับเป็น True ถ้ารูปสี่เหลี่ยมนั้นอยู่ภายในหรือบนขอบเขตของวงกลม

เขียนฟังก์ชันชื่อ `rect_circle_overlap` ซึ่งรับอ็อบเจกต์ Circle และ Rectangle แล้วให้ค่าคืนกลับเป็น True ถ้ามุมใดมุมหนึ่งของสี่เหลี่ยมอยู่ภายในวงกลม หรือเขียนรุ่นที่ท้าทายมากขึ้นซึ่งจะให้ค่าออกมาเป็น True ถ้าส่วนใดส่วนหนึ่งของรูปสี่เหลี่ยมอยู่ภายในวงกลม

เฉลย: `http://thinkpython2.com/code/Circle.py`.

***แบบฝึกหัด* 15.2.** *เขียนฟังก์ชันชื่อ `draw_rect` ซึ่งรับอ็อบเจกต์ Turtle และ Rectangle แล้วใช้อ็อบเจกต์ Turtle ในการวาดรูปสี่เหลี่ยมโดยดูตัวอย่างการใช้งาน Turtle ในบทที่ 4*

เขียนฟังก์ชันชื่อ `draw_circle` ซึ่งรับอ็อบเจกต์ Turtle และ Circle แล้ววาดรูปของวงกลมนั้น

โปรแกรมเฉลย: `http://thinkpython2.com/code/draw.py`.

16. คลาสและฟังก์ชัน

ตอนนี้เราได้รู้วิธีการสร้างชนิดข้อมูลใหม่ ขั้นต่อไปเป็นการเขียนฟังก์ชันที่รับอ็อบเจกต์ชนิดผู้เขียนโปรแกรมกำหนดขึ้นเป็นพารามิเตอร์ และให้ค่าออกมาเป็นอ็อบเจกต์เหล่านั้น ในบทนี้ได้นำเสนอ “การเขียนโปรแกรมเชิงฟังก์ชัน (functional programming style)” และแผนการพัฒนาสองโปรแกรมใหม่

ตัวอย่างโปรแกรมของบทนี้สามารถดาวน์โหลดได้ที่ `http://thinkpython2.com/code/Time1.py` ผลเฉลยสำหรับแบบฝึกหัดอยู่ที่ `http://thinkpython2.com/code/Time1_soln.py`

16.1. เวลา

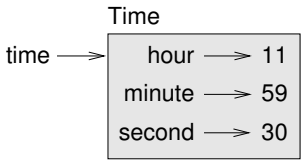
เช่นกันกับตัวอย่างอื่นของชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดเอง เราจะประกาศคลาส `Time` ซึ่งใช้บันทึกเวลาของวัน นิยามของคลาสมีลักษณะดังนี้

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

เราสามารถสร้างอ็อบเจกต์ใหม่ของ `Time` และกำหนดค่าของแอตทริบิวต์ชั่วโมง (hour) นาที (minute) และ วินาที (second)

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```



รูปที่ 16.1.: แผนภาพอ็อบเจกต์

แผนภาพสถานะของอ็อบเจกต์ `Time` มีลักษณะดังแสดงในรูปที่ 16.1

เพื่อเป็นการฝึก ให้เขียนฟังก์ชันชื่อ `print_time` ซึ่งรับอ็อบเจกต์ Time และพิมพ์ค่าเวลาในรูปแบบ `hour:minute:second` ข้อแนะนำคือ ใช้รูปแบบ `'%.2d'` สำหรับพิมพ์ตัวเลขอย่างน้อยสองตำแหน่ง โดยจะเติมศูนย์ด้านหน้ากรณีค่าน้อยกว่าสิบ

เขียนฟังก์ชันบูลีนชื่อ `is_after` ซึ่งรับอ็อบเจกต์ของ Time สองอ็อบเจกต์ `t1` และ `t2` แล้วให้ค่าออกมาเป็น `True` ถ้าเวลา `t1` ตามหลังเวลา `t2` ตามลำดับเหตุการณ์ ไม่เช่นนั้นจะให้ค่าออกมาเป็น `False`
ท้าทาย: ลองเขียนฟังก์ชันโดยไม่ใช้คำสั่ง `if`

16.2. ฟังก์ชันบริสุทธิ์

ในหัวข้อถัด ๆ ไป เราจะเขียนสองฟังก์ชันสำหรับบวกเวลาซึ่งแสดงรูปแบบการเขียนฟังก์ชันไว้สองแบบคือ แบบ*ฟังก์ชันบริสุทธิ์* (*pure function*) และแบบ*ตัวดัดแปลง* (*modifier*) และได้แสดงแผนการพัฒนาที่เรียกว่า วิธี **ต้นแบบและเติมแต่ง (prototype and patch)** ซึ่งเป็นวิธีการแก้ปัญหาที่ซับซ้อนโดยเริ่มจากต้นแบบอย่างง่าย และเพิ่มเติมการจัดการกับความซับซ้อนอย่างค่อยเป็นค่อยไป

นี่เป็นต้นแบบอย่างง่ายของฟังก์ชัน `add_time`

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

ภายในฟังก์ชันมีการสร้างอ็อบเจกต์ `Time` ขึ้นใหม่ และกำหนดค่าตั้งต้นให้กับแอตทริบิวต์แล้วให้ค่าออกมาเป็นอ้างอิงไปยังอ็อบเจกต์ใหม่นั้น การทำงานเช่นนี้เรียกว่า **ฟังก์ชันบริสุทธิ์** เนื่องจากไม่มีการแก้ไข

ค่าของอ็อบเจกต์ที่ถูกส่งเป็นอาร์กิวเมนต์และไม่ได้รับผลกระทบใดเว้นแต่ค่าที่ให้ออกมา มีลักษณะเดียวกับฟังก์ชันสำหรับแสดงค่าหรือฟังก์ชันสำหรับรับอินพุตจากผู้ใช้

เพื่อการทดสอบฟังก์ชันจึงสร้างอ็อบเจกต์ของ Time ขึ้นมาสองอ็อบเจกต์ อ็อบเจกต์ `start` ใช้เก็บค่าเวลาเริ่มต้นของภาพยนตร์ เช่นเรื่อง *มอนตี้ไพธอนกับจอกศักดิ์สิทธิ์ (Monty Python and the Holy Grail)* และอ็อบเจกต์ `duration` สำหรับเก็บค่าเวลาที่ใช้ในการฉายภาพยนตร์ ซึ่งจะเป็น 1 ชั่วโมง 35 นาที

ฟังก์ชัน `add_time` จะหาว่าภาพยนตร์ถูกฉายจบตอนไหน

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

ผลลัพธ์คือ `10:80:00` อาจจะไม่ใช่อย่างที่เราหวังไว้ ปัญหาคือ ฟังก์ชันไม่ได้จัดการกับกรณีจำนวนเวลาวินาทีและนาทีที่มีผลรวมเกิน 60 เมื่อเกิดเหตุการณ์ดังกล่าวเราจะต้อง “ทด” เวลาส่วนเกินของวินาทีไปที่นาที และยกยอดนาทีส่วนเกินไปเพิ่มให้กับชั่วโมง

นี่คือฟังก์ชันรุ่นที่ได้รับการปรับปรุง

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
```

```
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

แม้ว่าฟังก์ชันนี้จะถูกต้องแล้ว แต่ก็ค่อนข้างใหญ่ เราจะมาดูเวอร์ชันที่สั้นกว่านี้ในภายหลัง

16.3. ตัวดัดแปลง

ในบางครั้งก็ถือเป็นประโยชน์มากที่จะให้ฟังก์ชันแก้ไขค่าอ็อบเจกต์ที่รับเข้ามาเป็นพารามิเตอร์ ซึ่งในกรณีนี้โปรแกรมที่เรียกฟังก์ชัน จะเห็นการเปลี่ยนแปลงค่าของออบเจ๊คได้. ฟังก์ชันที่ทำงานในลักษณะนี้ เรียกว่า **ตัวดัดแปลง** (modifiers)

ฟังก์ชัน `increment` ทำการบวกเพิ่มตัวเลขจำนวนวินาทีให้กับอ็อบเจกต์ `Time` สามารถเขียนเป็นตัวดัดแปลงได้อย่างเป็นธรรมชาติ และนี่เป็นร่างคร่าวๆ

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

บรรทัดแรกทำหน้าที่บวกเพิ่มวินาที ส่วนที่เหลือเป็นการจัดการกับกรณีพิเศษที่เราเจอก่อนหน้านี้

ฟังก์ชันนี้ทำงานได้ถูกต้องหรือไม่? อะไรจะเกิดขึ้นถ้าค่าของ `seconds` มีค่ามากกว่า 60 มาก ๆ ?

ในกรณีนั้น การทดค่าเพียงแค่ครั้งเดียวจะยังไม่เพียงพอ เราจะต้องทำซ้ำจนกว่าค่าของ `time.second` จะน้อยกว่า 60 แนวทางหนึ่งที่ทำได้คือ แทนที่คำสั่ง `if` ด้วยคำสั่ง `while` จะทำให้การทำงานของฟังก์ชันถูกต้องแต่ก็ไม่ค่อยมีประสิทธิภาพนัก ทำเป็นแบบฝึกหัด คือ เขียนฟังก์ชัน `increment` เวอร์ชันที่ถูกต้องแต่ไม่มีการทำงานแบบวนซ้ำ

การทำงานทุกอย่างที่สามารถทำได้ด้วยตัวดัดแปลงจะสามารถทำด้วยฟังก์ชันบริสุทธิ์ได้เช่นกัน อันที่จริงภาษาเขียนโปรแกรมบางภาษาอนุญาตให้เขียนเฉพาะฟังก์ชันบริสุทธิ์ เนื่องจากมีความเชื่อว่า โปรแกรมที่ใช้เฉพาะฟังก์ชันบริสุทธิ์พัฒนาได้เร็วกว่าและมีข้อผิดพลาดน้อยกว่าโปรแกรที่ใช้ตัวดัดแปลง แต่หลายครั้งที่การใช้ตัวดัดแปลงมีความสะดวกสบายกว่า (และการใช้ฟังก์ชันมีแนวโน้มของประสิทธิภาพด้อยกว่า)

โดยทั่วไป ผมแนะนำให้คุณเขียนฟังก์ชันบริสุทธิ์เมื่อใดก็ตามที่สมเหตุสมผล และใช้ตัวดัดแปลงเฉพาะเมื่อมีข้อได้เปรียบที่น่าสนใจเท่านั้น แนวทางนี้อาจเรียกว่ารูปแบบ **การเขียนโปรแกรมเชิงฟังก์ชัน (functional programming style)**

เพื่อเป็นการฝึก ให้เขียนฟังก์ชัน `increment` ที่เป็นรุ่น “บริสุทธิ์ (pure)” ซึ่งจะสร้างอ็อบเจกต์ Time ขึ้นมาใหม่และให้ค่าคืนกลับเป็นอ็อบเจกต์นั้นแทนการแก้ไขค่าของพารามิเตอร์

16.4. การสร้างต้นแบบเทียบกับการวางแผน

แผนการพัฒนาที่ผ่านมาถูกเรียกว่า “วิธีต้นแบบและเติมแต่ง” แต่ละฟังก์ชันถูกทำเป็นต้นแบบด้วยการทำงานพื้นฐานก่อน จากนั้นทดสอบและปรับแก้ข้อผิดพลาดที่เกิดขึ้นระหว่างใช้งาน

แนวทางนี้สามารถเป็นหนทางที่มีประสิทธิภาพ โดยเฉพาะในเวลาที่เรายังไม่มีความเข้าใจปัญหาอย่างลึกซึ้ง แต่การปรับเพิ่มการแก้ไขสามารถทำให้โค้ดมีความซับซ้อน เพราะต้องจัดการกับกรณีเฉพาะหลายกรณี และทำให้ไม่น่าเชื่อถือ เนื่องจากเป็นการยากที่จะรู้ได้ว่าเราเจอข้อผิดพลาดได้ครบทุกกรณี

อีกทางเลือกหนึ่งคือ **การออกแบบแล้วพัฒนา (designed development)** ซึ่งการเข้าใจปัญหาในระดับสูงช่วยทำให้การเขียนโปรแกรมง่ายขึ้นมาก ในกรณีนี้ความเข้าใจระดับสูงคือ อ็อบเจกต์ Time เป็นเลขฐาน 60 จำนวน 3 ตำแหน่ง ! (ดูข้อมูลเพิ่มเติม `http://en.wikipedia.org/wiki/Sexagesimal`) โดยมี `second` เป็นหลักหน่วย มี `minute` เป็นหลักหกสิบ และ `hour` เป็นหลักสามพันหกร้อย

เมื่อเราเขียนฟังก์ชัน `add_time` และ `increment` เราได้ทำการบวกในฐาน 60 และนี่คือเหตุผลว่าทำไมเราจึงทดจากหลักหนึ่งไปยังหลักถัดไป

จากข้อสังเกตนี้จึงได้แนวทางใหม่ที่สามารถจัดการปัญหาได้ทั้งหมด เนื่องจากเราสามารถแปลงเวลาให้เป็นจำนวนเต็มและใช้ความสามารถของคอมพิวเตอร์ช่วยบวกจำนวนเต็มให้

นี่คือฟังก์ชันที่แปลงเวลาเป็นจำนวนเต็ม

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

และนี่คือฟังก์ชันที่แปลงจำนวนเต็มเป็นเวลา (พึงระลึกว่าฟังก์ชัน `divmod` หารอาร์กิวเมนต์แรกด้วยอาร์กิวเมนต์ที่สองแล้วให้ค่าออกมาเป็นผลหารและเศษในรูปของทูเพิล)

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

อาจจะต้องใช้เวลาคิดสักหน่อยหรือรันทดสอบเพื่อให้มั่นใจว่าฟังก์ชันนี้ทำงานได้ถูกต้อง วิธีหนึ่งที่จะช่วยทดสอบได้คือ ตรวจสอบว่า `time_to_int(int_to_time(x)) == x` สำหรับหลายๆ ค่าของ `x` นี่เป็นตัวอย่างของการตรวจสอบความสอดคล้อง

และเมื่อมั่นใจแล้วว่าการทำงานนั้นถูกต้อง เราก็สามารถใช้คำสั่งเหล่านี้มาเขียนฟังก์ชัน `add_time` ใหม่ได้ดังนี้

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

เวอร์ชันนี้สั้นกว่าเวอร์ชันก่อนหน้าและง่ายต่อการตรวจสอบ ให้ทำเป็นแบบฝึกหัด คือ เขียนฟังก์ชัน `increment` ใหม่โดยใช้ฟังก์ชัน `time_to_int` และ `int_to_time` ช่วย

บางทีการแปลงค่าจากฐาน 60 เป็นฐาน 10 ไปและกลับก็ยากกว่าการจัดการกับเวลา การแปลงฐานเข้าใจได้ยากกว่า สัญชาตญาณของเราจึงบอกว่าการจัดการกับเวลาน่าจะเป็นวิธีที่ดีกว่า

แต่ถ้าเราเข้าใจได้อย่างถ่องแท้เรื่องการจัดการกับเลขฐาน 60 และได้ลงทุนเขียนฟังก์ชันการแปลงแล้ว (`time_to_int` และ `int_to_time`) เราจะได้โปรแกรมที่สั้นกว่า ง่ายต่อการทำความเข้าใจและแก้ไข แถมยังน่าเชื่อถือกว่า

มันยังจะเป็นการง่ายกว่าที่จะเพิ่มความสามารถอย่างอื่นเข้าไป เช่นการลบเวลาเพื่อหาช่วงระยะเวลาระหว่างสองเวลา วิธีการแบบธรรมดาคือ ลบแบบมีการยืม แต่การใช้ฟังก์ชันการแปลงช่วยจะง่ายกว่าและมีโอกาสถูกต้องสูงกว่า

พูดอย่างกระทบกระเทียบ บางครั้งทำปัญหาให้ยากกว่า (มีความกว้างขวางกว่า) ทำให้ทำได้ง่ายกว่า เพราะว่ามีกรณีเฉพาะน้อยกว่าและมีโอกาสผิดพลาดน้อยกว่า

16.5. การดีบัก

อ็อบเจกต์ Time เป็นรูปแบบที่ดีถ้าค่าของ `minute` และ `second` มีค่าเป็นจำนวนเต็มระหว่าง 0 ถึง 60 (รวม 0 แต่ไม่รวม 60) และค่า `hour` เป็นเลขบวก ค่าของ `hour` และ `minute` ควรเป็นจำนวนเต็ม เราอาจจะยอมให้วินาทีมีค่าเป็นทศนิยมได้

ข้อกำหนดเหล่านี้ถูกเรียกว่า **ความคงที่ (invariants)** เพราะจะต้องเป็นจริงเสมอ ถ้ามีอะไรที่ไม่สอดคล้อง คือ ไม่ถูกต้องตามข้อกำหนด จะทำให้เกิดข้อผิดพลาดบางประการขึ้น

การเขียนคำสั่งสำหรับตรวจสอบความคงที่ของข้อมูลสามารถช่วยตรวจสอบข้อผิดพลาดและช่วยหาสาเหตุของข้อผิดพลาดได้ ตัวอย่างเช่น เราอาจจะมีฟังก์ชัน `valid_time` ที่รับอ็อบเจกต์ Time และให้ค่าออกมาเป็น `False` ถ้ามีการละเมิดข้อกำหนด

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

ในช่วงเริ่มต้นของแต่ละฟังก์ชัน เราสามารถตรวจสอบอาร์กิวเมนต์เพื่อให้มั่นใจว่ามีความถูกต้อง

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

หรือเราอาจใช้คำสั่ง assert ซึ่งจะตรวจสอบข้อกำหนดความคงที่และสร้างเอ็กเซ็ปชั่นถ้าไม่ผ่านข้อกำหนด

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

คำสั่ง `assert` เป็นคำสั่งที่มีประโยชน์ เนื่องจากสามารถแยกข้อแตกต่างของการตรวจสอบข้อผิดพลาดออกจากคำสั่งที่จัดการกับข้อมูลปกติได้

16.6. อภิธานศัพท์

ต้นแบบและเติมแต่ง (prototype and patch): แผนการพัฒนาที่ประกอบด้วยการเขียนร่างคร่าวๆ ของโปรแกรม แล้วทดสอบและแก้ไขข้อผิดพลาดที่ถูกพบ

การออกแบบแล้วพัฒนา (designed development): แผนการพัฒนาที่เกี่ยวข้องกับการเข้าใจระดับสูงในปัญหาและมีแผนการยิ่งกว่าการพัฒนาแบบต่อเติมหรือการพัฒนาต้นแบบ

ฟังก์ชันบริสุทธิ์ (pure function): ฟังก์ชันที่ไม่แก้ไขค่าของอ็อบเจกต์ซึ่งรับเป็นอาร์กิวเมนต์ ฟังก์ชันบริสุทธิ์ส่วนใหญ่เป็นฟังก์ชันที่ให้ผล

ตัวดัดแปลง (modifier): ฟังก์ชันที่แก้ไขค่าของอ็อบเจกต์ที่รับเป็นอาร์กิวเมนต์อย่างน้อยหนึ่งแห่ง ตัวดัดแปลงส่วนใหญ่จะเป็นฟังก์ชันที่ไม่ให้ค่าคืนกลับ คือคืนค่า **None**

การเขียนโปรแกรมเชิงฟังก์ชัน (functional programming style): รูปแบบการออกแบบซึ่งการทำงานส่วนใหญ่เป็นฟังก์ชันบริสุทธิ์

ความคงที่ (invariant): เงื่อนไขที่จะต้องเป็นจริงเสมอระหว่างการทำงานของโปรแกรม

คำสั่งยืนยัน (assert statement): คำสั่งที่ใช้ตรวจสอบเงื่อนไขและสร้างเอ็กเซ็ปชั่นในกรณีที่ไม่สอดคล้องกับเงื่อนไขที่กำหนด

16.7. แบบฝึกหัด

ตัวอย่างโปรแกรมของบทนี้สามารถดาวน์โหลดได้ที่ `http://thinkpython2.com/code/Time1.py` ผลเฉลยสำหรับแบบฝึกหัดอยู่ที่ `http://thinkpython2.com/code/Time1_soln.py`

แบบฝึกหัด **16.1.** *เขียนฟังก์ชันชื่อ* ***`mul_time`*** *ซึ่งรับอ็อบเจกต์ Time และตัวเลข แล้วให้ค่าออกมาเป็นอ็อบเจกต์ Time ใหม่ ที่บรรจุผลคูณของเวลาและตัวเลขนั้น*

จากนั้นใช้ฟังก์ชัน ***`mul_time`*** *เพื่อเขียนฟังก์ชันซึ่งรับอ็อบเจกต์ Time ซึ่งเป็นเวลาที่ใช้เพื่อไปถึงเส้นชัยในการแข่งขันและรับตัวเลขที่เป็นระยะทาง จากนั้นให้ค่าออกมาเป็นอ็อบเจกต์ Time ที่แสดงค่าก้าวเฉลี่ย (average pace: เวลาต่อไมล์)*

แบบฝึกหัด **16.2.** *โมดูล* ***`datetime`*** *มีอ็อบเจกต์* ***`time`*** *ที่คล้ายกับอ็อบเจกต์ Time ในบทนี้ แต่มีเมธอดและตัวดำเนินการมากมาย ศึกษาเอกสารได้ที่* ***`http://docs.python.org/3/library/datetime.html`***

1. *ใช้โมดูล* ***`datetime`*** *เพื่อเขียนโปรแกรมสำหรับอ่านเวลาปัจจุบัน และพิมพ์ค่าวันของสัปดาห์*
2. *เขียนโปรแกรมที่รับวันเกิดเป็นอินพุตแล้วพิมพ์อายุของผู้ใช้ และจำนวนวัน ชั่วโมง นาที และวินาทีที่เหลืออยู่ก่อนจะถึงวันเกิดครั้งถัดไป*
3. *สำหรับคนสองคนที่เกิดคนละวัน จะมีวันที่เขาทั้งสองมีอายุเป็นสองเท่าของอีกคน นั่นคือวันสองเท่าของพวกเขา เขียนโปรแกรมที่รับวันเกิดสองวันแล้วคำนวณหาวันสองเท่าของพวกเขา*
4. *เพื่อให้ท้าทายขึ้นอีกหน่อย จงเขียนเวอร์ชันที่ครอบคลุมกว่าคือ คำนวณวันที่คนหนึ่งมีอายุ n เท่าของอีกคน*

เฉลย: ***`http://thinkpython2.com/code/double.py`***

17. คลาสและเมธอด

แม้ว่าเราจะได้ใช้คุณสมบัติเชิงวัตถุบางอย่างของไพธอนแล้ว โปรแกรมในสองบทที่ผ่านยังไม่ใช่โปรแกรมเชิงวัตถุอย่างแท้จริง เพราะว่ายังไม่มีความสัมพันธ์ระหว่างชนิดข้อมูลที่ผู้เขียนกำหนดกับฟังก์ชันที่ทำงานกับข้อมูลเหล่านั้น ขั้นถัดไปคือการแปลงฟังก์ชันดังกล่าวให้เป็นเมธอดซึ่งจะทำให้ความสัมพันธ์ชัดเจนขึ้น

ตัวอย่างโปรแกรมของบทนี้สามารถดาวน์โหลดได้ที่ `http://thinkpython2.com/code/Time2.py` ผลเฉลยสำหรับแบบฝึกหัดอยู่ที่ `http://thinkpython2.com/code/Point2_soln.py`

17.1. คุณสมบัติเชิงวัตถุ

ไพธอนเป็น **ภาษาโปรแกรมเชิงวัตถุ (object-oriented programming language)** นั่นหมายถึง ตัวภาษามีคุณสมบัติที่รองรับการเขียนโปรแกรมเชิงวัตถุ ซึ่งมีคุณลักษณะดังนี้

- โปรแกรมมีการนิยามคลาสและเมธอด
- การประมวลผลส่วนใหญ่ถูกแสดงออกในรูปของการทำงานกับวัตถุ ที่มักเรียกว่า อ็อบเจกต์
- อ็อบเจกต์มักจะเป็นตัวแทนของสิ่งต่าง ๆ ในโลกความเป็นจริง และเมธอดมักจะสอดคล้องกับวิธีที่วัตถุในโลกจริงนั้นปฏิสัมพันธ์

ตัวอย่างเช่น คลาส `Time` ที่นิยามในบทที่ 16 สอดคล้องกับวิธีที่ผู้คนบันทึกเวลาของวัน และฟังก์ชันที่เราประกาศก็สอดคล้องกับสิ่งที่ผู้คนกระทำกับเวลา ในทำนองเดียวกัน คลาส `Point` และ `Rectangle` ในบทที่ 15 สอดคล้องกับแนวคิดทางคณิตศาสตร์ของจุดและสี่เหลี่ยม

จนถึงตอนนี้เรายังไม่ได้ใช้ประโยชน์จากคุณสมบัติที่ไพธอนมีเพื่อสนับสนุนการเขียนโปรแกรมเชิงวัตถุ คุณสมบัติเหล่านี้ไม่ได้มีความจำเป็นอย่างเคร่งครัด และคุณสมบัติส่วนใหญ่เป็นกรณีทางเลือกทางไวยากรณ์

ของสิ่งที่เราเคยทำไปแล้ว แต่ในหลายกรณีทางเลือกเหล่านั้นก็มีความรัดกุมและชัดเจนมากกว่าในการสื่อโครงสร้างของโปรแกรม

อย่างเช่นตัวอย่างใน `Time1.py` ซึ่งไม่มีความสัมพันธ์ที่ชัดเจนระหว่างส่วนนิยามคลาสกับส่วนนิยามฟังก์ชันต่าง ๆ ที่ตามมา จากการพิเคราะห์แล้วเห็นได้ชัดว่าทุกฟังก์ชันจะรับอ็อบเจกต์ `Time` อย่างน้อยหนึ่งอ็อบเจกต์เป็นอาร์กิวเมนต์

ข้อสังเกตนี้กลายเป็นแรงจูงใจสำหรับ **เมธอด** ซึ่งก็คือฟังก์ชันที่เกี่ยวข้องอยู่กับคลาสเฉพาะคลาสใดคลาสหนึ่ง เราเคยเห็นเมธอดสำหรับ สตริง ลิสต์ ดิกชันนารี และทูเพิล มาแล้ว ในบทนี้เราจะประกาศเมธอดสำหรับชนิดข้อมูลที่ผู้เขียนโปรแกรมกำหนดขึ้นเอง

เมธอดมีความหมายเหมือนกับฟังก์ชัน แต่มีไวยากรณ์ที่แตกต่างกันสองประการ

- เมธอดถูกนิยามไว้ภายในนิยามของคลาสเพื่อสร้างความสัมพันธ์ที่ชัดแจ้งระหว่างคลาสและเมธอด
- ไวยากรณ์ของการเรียกใช้เมธอดต่างจากการเรียกใช้ฟังก์ชัน

ในสองสามหัวข้อถัดไป เราจะนำฟังก์ชันจากสองบทที่แล้วมาแปลงเป็นเมธอด การแปลงนี้มีขั้นตอนตรงไปตรงมาโดยสามารถทำตามลำดับขั้นตอน ถ้าคุณคุ้นเคยกับการแปลงจากรูปแบบหนึ่งไปยังอีกรูปแบบหนึ่งแล้ว คุณจะสามารถเลือกรูปแบบที่ดีที่สุดสำหรับสิ่งที่คุณกำลังทำ

17.2. การพิมพ์อ็อบเจกต์

ในบทที่ 16 เราได้นิยามคลาสชื่อ `Time` และในหัวข้อที่ 16.1 เราได้เขียนฟังก์ชันชื่อ `print_time`

```
class Time:
    """Represents the time of day."""

def print_time(time):
   print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

เราต้องส่งอ็อบเจกต์ `Time` เป็นอาร์กิวเมนต์เพื่อเรียกใช้ฟังก์ชัน

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
```

```
>>> start.second = 00
>>> print_time(start)
09:45:00
```

เพื่อสร้างเมธอด `print_time` ทั้งหมดที่เราต้องทำคือ ย้ายส่วนนิยามฟังก์ชันไปไว้ในส่วนนิยามคลาส ให้ระวังการเปลี่ยนแปลงของการเยื้อง

```
class Time:
  def print_time(time):
    print('%.2d:%.2d:%.2d'%(time.hour,time.minute,time.second))
```

ถึงตอนนี้มีสองแนวทางที่จะเรียกใช้ `print_time` วิธีแรก (ไม่เป็นที่นิยม) คือ ใช้ไวยากรณ์ฟังก์ชัน

```
>>> Time.print_time(start)
09:45:00
```

ในการใช้สัญกรณ์จุดนี้ `Time` เป็นชื่อของคลาส และ `print_time` เป็นชื่อของเมธอด ส่วน `start` ถูกส่งเป็นพารามิเตอร์

แนวทางที่สอง (กระชับมากขึ้น) เป็นการใช้ไวยากรณ์เมธอด

```
>>> start.print_time()
09:45:00
```

ในการใช้สัญกรณ์จุดนี้ `print_time` เป็นชื่อของเมธอด (เช่นเคย) และ `start` เป็นอ็อบเจกต์ที่เมธอดถูกเรียกใช้ ซึ่งถือเป็น **ประธาน (subject)** เช่นเดียวกับประธานของประโยคเป็นสิ่งที่ประโยคเกี่ยวข้องด้วย ประธานของการเรียกใช้เมธอด ก็เป็นสิ่งที่เมธอดเกี่ยวข้องด้วย

ภายในเมธอด ประธานถูกกำหนดให้เป็นพารามิเตอร์แรก ดังนั้นในกรณีนี้ `start` ถูกกำหนดให้กับ `time`

ตามธรรมเนียมแล้วพารามิเตอร์แรกของเมธอดถูกเรียกว่า `self` ดังนั้นจึงนิยมที่จะเขียน `print_time` เป็นดังนี้

```
class Time:
  def print_time(self):
    print('%.2d:%.2d:%.2d'%(self.hour,self.minute,self.second))
```

เหตุผลของธรรมเนียมนี้อุปมาโดยปริยายได้ดังนี้

- ไวยากรณ์แบบเรียกใช้ฟังก์ชัน `print_time(start)` แสดงถึงว่าฟังก์ชันเป็นผู้กระทำ เหมือนมันบอกว่า “เฮ้ print_time! นี่คืออ็อบเจกต์สำหรับให้คุณพิมพ์”
- ในการโปรแกรมเชิงวัตถุ อ็อบเจกต์จะเป็นผู้กระทำ การเรียกใช้เมธอดในลักษณะ `start.print_time()` บ่งบอกว่า “เฮ้ start! กรุณาพิมพ์ตัวคุณเอง”

การเปลี่ยนมุมมองนี้อาจจะสุภาพกว่า แต่ก็ไม่ชัดเจนว่ามีประโยชน์ ในตัวอย่างที่ผ่านมามันอาจจะไม่ชัด แต่บางครั้งการเปลี่ยนภาระจากฟังก์ชันไปเป็นอ็อบเจกต์ทำให้สามารถที่จะเขียนได้หลากหลายฟังก์ชัน (หรือเมธอด) ช่วยให้ง่ายในการบำรุงรักษาและนำไปใช้ซ้ำ

เพื่อเป็นการฝึก ให้เขียนฟังก์ชัน `time_to_int` ใหม่ (จาก หัวข้อ 16.4) ให้เป็นเมธอด คุณอาจจะอยากเขียน `int_to_time` เป็นเมธอดด้วย แต่นั่นไม่สมเหตุสมผล เพราะว่า ไม่มีอ็อบเจกต์ที่จะเรียกใช้มัน

17.3. อีกตัวอย่าง

นี่เป็นอีกเวอร์ชันของ `increment` (จากหัวข้อ 16.3) ถูกเขียนใหม่เป็นเมธอด

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

เวอร์ชันนี้กำหนดให้ `time_to_int` ถูกเขียนเป็นเมธอด ให้สังเกตว่าเป็นฟังก์ชันบริสุทธิ์ ไม่ใช่ตัวดัดแปลง

นี่คือวิธีเรียกใช้ `increment`

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

ประธาน `start` ถูกกำหนดให้พารามิเตอร์แรกคือ `self` ส่วนอาร์กิวเมนต์ `1337` ถูกกำหนดให้กับพารามิเตอร์ที่สองคือ `seconds`

กลไกนี้อาจทำให้เกิดความสับสน โดยเฉพาะอย่างยิ่งหากคุณทำผิดพลาด ตัวอย่างเช่น ถ้าคุณเรียกใช้ `increment` ด้วยสองอาร์กิวเมนต์ คุณจะได้

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

แค่ข้อความแสดงข้อผิดพลาดก็เริ่มสับสน เนื่องจากมีแค่สองอาร์กิวเมนต์ในวงเล็บ แต่ประธานก็ถือว่าเป็นอาร์กิวเมนต์ด้วย ดังนั้นทั้งหมดจึงรวมเป็นสาม

อนึ่ง**อาร์กิวเมนต์ตำแหน่ง (positional argument)**เป็นอาร์กิวเมนต์ที่ไม่มีชื่อพารามิเตอร์ นั่นคือมันไม่ใช่อาร์กิวเมนต์คำสำคัญ ในการเรียกฟังก์ชันนี้

```
sketch(parrot, cage, dead=True)
```

`parrot` และ `cage` เป็นอาร์กิวเมนต์ตำแหน่ง ส่วน dead เป็นอาร์กิวเมนต์คำสำคัญ

17.4. ตัวอย่างที่ซับซ้อนมากขึ้น

เขียน `is_after` ใหม่ (จากหัวข้อ 16.1) จะค่อนข้างซับซ้อนกว่า เนื่องจากมันรับพารามิเตอร์เป็นอ็อบเจกต์ Time สองอ็อบเจกต์ ในกรณีนี้เป็นธรรมดาที่จะตั้งชื่อพารามิเตอร์แรกเป็น `self` และพารามิเตอร์ลำดับที่สองเป็น `other`

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

การเรียกใช้เมธอดนี้ คุณต้องเรียกมันจากอ็อบเจกต์หนึ่งและส่งอ็อบเจกต์อื่นเป็นอาร์กิวเมนต์

```
>>> end.is_after(start)
True
```

สิ่งหนึ่งที่ดีเกี่ยวกับไวยากรณ์นี้คือ มันเกือบอ่านเหมือนภาษาอังกฤษ: “end is after start?”

17.5. เมธอด init

เมธอด init (ย่อมาจาก “initialization”) เป็นเมธอดพิเศษที่ถูกเรียกใช้ขณะที่อ็อบเจกต์ถูกสร้างอินสแตนซ์ ชื่อเต็มคือ `__init__` (ตัวอักษรขีดล่างสองตัวตามด้วย `init` แล้วก็ตัวอักษรขีดล่างอีกสองตัว) เมธอด init สำหรับคลาส `Time` อาจมีลักษณะเช่นนี้

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

เป็นเรื่องปกติสำหรับพารามิเตอร์ของ `__init__` ที่ชื่อจะเหมือนกันกับแอตทริบิวต์ คำสั่งต่อไปนี้

```
        self.hour = hour
```

เก็บค่าของพารามิเตอร์ `hour` เป็นแอตทริบิวต์ของ `self`

พารามิเตอร์ต่าง ๆ เป็นแบบทางเลือก ดังนั้นถ้าคุณเรียกใช้ `Time` โดยไม่มีอาร์กิวเมนต์ เท่ากับคุณใช้ค่าดีฟอลท์ของพารามิเตอร์

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

ถ้าคุณให้อาร์กิวเมนต์ตัวเดียว ค่าที่ให้จะเข้าไปเป็นค่าของ `hour`

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

ถ้าคุณให้อาร์กิวเมนต์สองตัว มันจะเข้าไปเป็นค่าของ `hour` และ `minute` ตามลำดับ

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

และถ้าคุณให้อาร์กิวเมนต์สามตัว มันจะเข้าไปแทนที่ค่าดีฟอลท์ทั้งสามค่า

เพื่อเป็นการฝึกหัด ให้เขียนเมธอด init สำหรับคลาส `Point` ที่จะรับ `x` และ `y` เป็นอาร์กิวเมนต์แบบพารามิเตอร์ทางเลือก และกำหนดค่าให้กับแอตทริบิวต์ที่สอดคล้องกัน

17.6. เมธอด __str__

เมธอด __str__ เป็นเมธอดพิเศษ เช่นเดียวกับเมธอด __init__ ซึ่งควรจะส่งคืนข้อความบรรยายอ็อบเจกต์

นี่เป็นตัวอย่างเมธอด str สำหรับอ็อบเจกต์ Time

```
# inside class Time:

  def __str__(self):
    return '%.2d:%.2d:%.2d'%(self.hour,self.minute,self.second)
```

เมื่อคุณ print อ็อบเจกต์ ไพธอนจะเรียกใช้เมธอด str

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

เวลาผมเขียนคลาสขึ้นใหม่ ผมมักจะเริ่มต้นด้วยการเขียนเมธอด __init__ ซึ่งทำให้สร้างอินสแตนซ์อ็อบเจกต์ได้ง่ายขึ้น และเมธอด __str__ ซึ่งเป็นประโยชน์สำหรับการแก้จุดบกพร่อง

เพื่อเป็นการฝึกหัด ให้เขียนเมธอด str สำหรับคลาส Point แล้วสร้างอ็อบเจกต์ Point และพิมพ์มัน

17.7. การโอเวอร์โหลดตัวดำเนินการ

ด้วยการกำหนดเมธอดพิเศษอื่นๆ คุณสามารถระบุพฤติกรรมของตัวดำเนินการต่อชนิดข้อมูลที่กำหนดขึ้น ตัวอย่างเช่น ถ้าคุณนิยามเมธอดชื่อ __add__ สำหรับคลาส Time คุณสามารถใช้ตัวดำเนินการ + กับอ็อบเจกต์ Time

การกำหนดมีลักษณะคล้ายอย่างนี้

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

และคุณสามารถเรียกใช้ได้ดังนี้

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

เมื่อคุณใช้ตัวดำเนินการ + กับอ็อบเจกต์ Time ไพธอนเรียกใช้ เมธอด __add__ เมื่อคุณพิมพ์ผลลัพธ์ ไพธอนเรียกใช้เมธอด __str__ จึงมีอะไรเกิดขึ้นมากมายอยู่เบื้องหลัง

การเปลี่ยนพฤติกรรมของตัวดำเนินการเพื่อใช้กับชนิดข้อมูลที่กำหนดขึ้นเองเรียกว่า **การโอเวอร์โหลดตัวดำเนินการ (operator overloading)** และทุกตัวดำเนินการในไพธอนจะมีเมธอดพิเศษคล้ายกับ __add__ ดูรายละเอียดเพิ่มเติมได้ที่ http://docs.python.org/3/reference/datamodel.html#specialnames

เพื่อเป็นการฝึกหัด ให้เขียนเมธอด add สำหรับคลาส Point

17.8. การจัดการตามชนิดข้อมูล

ในส่วนก่อนหน้านี้เราได้บวกสองอ็อบเจกต์ Time แต่คุณอาจต้องการบวกจำนวนเต็มให้กับอ็อบเจกต์ Time ต่อไปนี้เป็นเวอร์ชันของ __add__ ที่ตรวจสอบ ชนิดข้อมูลของ other แล้วเรียกใช้งาน add_time หรือ increment

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
```

```
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

ฟังก์ชันสำเร็จรูป `isinstance` รับค่าและคลาสอ็อบเจกต์ และส่งคืนค่า `True` หากค่านั้นเป็นอินสแตนซ์ของคลาส

ถ้า `other` เป็นอ็อบเจกต์ Time เมธอด `__add__` จะเรียกใช้ `add_time` มิฉะนั้นจะถือว่าพารามิเตอร์เป็นตัวเลขและเรียกใช้เมธอด `increment` การดำเนินการนี้เรียกว่า **การจัดการตามชนิดข้อมูล (type-based dispatch)** เพราะมันส่งการคำนวณไปยังเมธอดต่าง ๆ ขึ้นอยู่กับประเภทของอาร์กิวเมนต์

นี่คือตัวอย่างที่ใช้ตัวดำเนินการ **+** กับชนิดข้อมูลประเภทต่าง ๆ

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

น่าเสียดายที่การใช้งานการบวกนี้ไม่อาจสลับลำดับ ถ้าจำนวนเต็มเป็นพารามิเตอร์แรกคุณจะได้

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

ปัญหาคือ แทนที่จะขอให้อ็อบเจกต์ Time บวกด้วยจำนวนเต็ม ไพธอนกลับขอให้บวกจำนวนเต็มด้วยอ็อบเจกต์ Time ทำให้ไม่รู้ว่าจะทำอย่างไร แต่มีวิธีแก้ปัญหาที่ชาญฉลาดสำหรับปัญหานี้ เมธอดพิเศษ `__radd__` ซึ่งย่อมาจาก “right-side add” เมธอดนี้ถูกเรียกใช้เมื่ออ็อบเจกต์ Time ปรากฏอยู่ด้านขวาของตัวดำเนินการ **+** นี่เป็นตัวอย่างการประกาศ

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

และนี่คือวิธีการใช้

```
>>> print(1337 + start)
10:07:17
```

เพื่อเป็นการฝึก ให้เขียนเมธอด **add** สำหรับคลาส Point ที่สามารถใช้ได้กับทั้งอ็อบเจกต์ Point และทูเพิล

- ถ้าตัวถูกดำเนินการที่สองคือ Point เมธอดควรส่งคืน Point ใหม่ โดยที่พิกัด x คือผลรวมพิกัด x ของตัวถูกดำเนินการ และในทำนองเดียวกันสำหรับพิกัด y
- ถ้าตัวถูกดำเนินการที่สองคือทูเพิล เมธอดควรเอาอิลิเมนต์แรกของทูเพิลไปใส่พิกัด x และอิลิเมนต์ที่สองไปใส่พิกัด y และส่งคืนอ็อบเจกต์ Point ใหม่พร้อมผลลัพธ์

17.9. ภาวะพหุสัณฐาน

การจัดการตามชนิดข้อมูลจะมีประโยชน์เมื่อจำเป็น แต่ (โชคดี) ที่ไม่จำเป็นเสมอไป บ่อยครั้งที่คุณสามารถหลีกเลี่ยงได้โดยการเขียนฟังก์ชันที่ทำงานได้อย่างถูกต้องสำหรับอาร์กิวเมนต์ต่างชนิดข้อมูลกัน

ฟังก์ชันหลายอย่างที่เราเขียนสำหรับสายอักขระสามารถใช้ได้กับข้อมูลแบบลำดับประเภทอื่น ๆ ตัวอย่างเช่นในหัวข้อ 11.2 เราใช้ **histogram** เพื่อนับจำนวนครั้งที่ตัวอักษรแต่ละตัวปรากฏในคำ

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

ฟังก์ชันนี้ยังใช้ได้กับลิสต์ ทูเพิล และแม้แต่ดิกชันนารีด้วย ตราบใดที่อิลิเมนต์ของ **s** สามารถแฮชได้ ซึ่งจะสามารถใช้เป็นกุญแจ (key) ใน **d** ได้

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

ฟังก์ชันที่ทำงานได้กับหลาย ๆ ชนิดข้อมูลเรียกว่า **มีพหุสัณฐาน (polymorphic)** การมีพหุสัณฐาน หรือเรียกอย่างเป็นทางการว่า ภาวะพหุสัณฐาน สามารถอำนวยความสะดวกในการใช้โค้ดซ้ำได้ ตัวอย่างเช่น ฟังก์ชันสำเร็จรูป **sum** ซึ่งหาผลรวมของสมาชิกในข้อมูลแบบลำดับ ทำงานได้เสมอ ถ้าอิลิเมนต์ของข้อมูลแบบลำดับรับปฏิบัติการบวกได้

เนื่องจากอ็อบเจกต์ Time มีเมธอด add จึงใช้งานได้กับ sum

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

โดยทั่วไป ถ้าการดำเนินการทั้งหมดภายในฟังก์ชันใช้งานได้กับชนิดข้อมูลที่กำหนด ฟังก์ชันนั้นจะใช้งานได้กับชนิดข้อมูลนั้น

ภาวะพหุสัณฐานที่ดีที่สุดคือชนิดที่ไม่ได้ตั้งใจ ซึ่งคุณจะค้นพบว่าฟังก์ชันที่คุณเขียนไปแล้วสามารถนำไปใช้กับชนิดข้อมูลที่คุณไม่เคยวางแผนไว้ได้

17.10. การดีบัก

การเพิ่มแอตทริบิวต์ให้กับอ็อบเจกต์ ณ จุดใด ๆ ในการทำงานของโปรแกรมนั้นสามารถทำได้ แต่ถ้าคุณมีอ็อบเจกต์ประเภทเดียวกันที่มีแอตทริบิวต์ไม่เหมือนกันจะทำให้เกิดข้อผิดพลาดได้ง่าย ถือเป็นความคิดที่ดีที่จะกำหนดค่าตั้งต้นแอตทริบิวต์ทั้งหมดของอ็อบเจกต์ในเมธอด init

หากคุณไม่แน่ใจว่าอ็อบเจกต์มีแอตทริบิวต์อย่างใดอย่างหนึ่งหรือไม่ คุณสามารถตรวจสอบด้วยฟังก์ชันสำเร็จรูป hasattr (ดูหัวข้อ 15.7)

อีกวิธีหนึ่งในการเข้าถึงแอตทริบิวต์คือใช้ฟังก์ชันสำเร็จรูป vars ซึ่งรับอ็อบเจกต์ และส่งคืนดิกชันนารี ที่จับคู่ชื่อแอ็ตทริบิวต์ (เป็นสายอักขระ) กับค่าของมัน

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

เพื่อจุดประสงค์ในการดีบัก คุณอาจพบว่ามีประโยชน์ในการพกพาฟังก์ชันนี้ไว้ใช้งาน

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

ฟังก์ชัน `print_attributes` สำรวจดิกชันนารีและพิมพ์ชื่อแอตทริบิวต์แต่ละรายการกับค่าของมัน

ฟังก์ชันสำเร็จรูป `getattr` รับอ็อบเจกต์และชื่อของแอ็ตทริบิวต์ (รับเป็นสายอักขระ) และจะส่งคืนค่าของแอ็ตทริบิวต์นั้นออกมา

17.11. ส่วนต่อประสานและการพัฒนา

เป้าหมายหนึ่งของการออกแบบเชิงวัตถุคือการทำให้ซอฟต์แวร์สามารถบำรุงรักษาได้สะดวกขึ้น ซึ่งหมายความว่าคุณสามารถให้โปรแกรมทำงานต่อไปได้แม้ส่วนอื่น ๆ ของระบบเปลี่ยนไปและสามารถปรับเปลี่ยนโปรแกรมให้ตรงตามข้อกำหนดใหม่ได้

หลักการออกแบบที่ช่วยให้บรรลุเป้าหมายนั้นคือการแยกส่วนต่อประสานออกจากอิมพลิเมนเตชั่น สำหรับอ็อบเจกต์นั่นหมายถึงเมธอดของคลาสไม่ควรขึ้นกับชนิดตัวแปรที่ใช้เก็บค่าแอตทริบิวต์

ตัวอย่างเช่น ในบทนี้เราได้พัฒนาคลาสที่แสดงถึงช่วงเวลาของวัน เมธอดที่คลาสนี้มีให้ ได้แก่ `time_to_int`, `is_after` และ `add_time`

รายละเอียดของการพัฒนาขึ้นอยู่กับวิธีที่เราแทนค่าเวลา ในบทนี้แอตทริบิวต์ของอ็อบเจกต์ `Time` ได้แก่ `hour` `minute` และ `second`

อีกทางเลือกหนึ่งคือ เราสามารถแทนที่แอตทริบิวต์เหล่านี้ด้วยจำนวนเต็มเดียวที่แทนจำนวนวินาทีตั้งแต่เที่ยงคืน การทำเช่นนี้จะทำให้วิธีการบางอย่าง เช่น `is_after` เขียนง่ายขึ้น แต่มันทำให้วิธีการอื่น ๆ ยากขึ้น

หลังจากที่คุณปรับใช้คลาสใหม่ คุณอาจพบอิมพลิเมนเตชั่นที่ดีขึ้น หากส่วนอื่น ๆ ของโปรแกรมใช้คลาสของคุณ มันอาจใช้เวลานานในการเปลี่ยนส่วนต่อประสานและก็มีโอกาสพลาดสูงด้วย

แต่ถ้าคุณออกแบบส่วนต่อประสานอย่างระมัดระวัง คุณสามารถเปลี่ยนการทำงานโดยไม่ต้องเปลี่ยนส่วนต่อประสาน ซึ่งหมายความว่าส่วนอื่นๆ ของโปรแกรมไม่จำเป็นต้องเปลี่ยน

17.12. อภิธานศัพท์

ภาษาเชิงวัตถุ (object-oriented language): ภาษาที่มีคุณสมบัติซึ่งอำนวยความสะดวกในการเขียนโปรแกรมเชิงวัตถุ เช่น เมธอดและชนิดข้อมูลที่กำหนดโดยโปรแกรมเมอร์

การโปรแกรมเชิงวัตถุ (object-oriented programming): รูปแบบของการเขียนโปรแกรมที่ข้อมูลและวิธีการจัดการข้อมูล จะถูกจัดเป็นคลาสและเมธอด

เมธอด (method): ฟังก์ชันที่กำหนดไว้ภายในนิยามคลาสและถูกเรียกใช้โดยอินสแตนซ์ของคลาสนั้น

ประธาน (subject): อ็อบเจกต์ที่เมธอดถูกเรียกใช้

อาร์กิวเมนต์ตำแหน่ง (positional argument): อาร์กิวเมนต์ที่ไม่มีชื่อพารามิเตอร์ ดังนั้นจึงไม่ใช่อาร์กิวเมนต์คำสำคัญ

การโอเวอร์โหลดโอเปอเรเตอร์ (operator overloading): การเปลี่ยนพฤติกรรมของตัวดำเนินการเช่นตัวดำเนินการ **+** เพื่อให้ทำงานได้กับชนิดข้อมูลที่กำหนดโดยโปรแกรมเมอร์

การจัดการตามชนิดข้อมูล (type-based dispatch): รูปแบบการเขียนโปรแกรมที่ตรวจสอบชนิดของตัวถูกดำเนินการและเรียกใช้ฟังก์ชันต่าง ๆ สำหรับชนิดข้อมูลที่ต่างกัน

พหุสัณฐาน (polymorphic): เกี่ยวกับฟังก์ชันที่ทำงานกับข้อมูลได้มากกว่าหนึ่งชนิด

การซ่อนข้อมูล (information hiding): หลักการที่ว่าส่วนต่อประสานที่จัดเตรียมโดยอ็อบเจกต์ไม่ควรขึ้นอยู่กับวิธีการพัฒนา โดยเฉพาะการแทนค่าแอตทริบิวต์

17.13. แบบฝึกหัด

***แบบฝึกหัด* 17.1.** *ดาวน์โหลดโค้ดของบทนี้จาก* `http://thinkpython2.com/code/Time2.py` *เปลี่ยนแอตทริบิวต์ของ* `Time` *ให้เป็นจำนวนเต็มเดียวที่แทนวินาทีตั้งแต่เที่ยงคืน จากนั้นแก้ไขเมธอด (และฟังก์ชัน* `int_to_time`*) เพื่อให้ทำงานได้กับการพัฒนาใหม่ คุณไม่ควรแก้ไขโค้ดทดสอบใน* `main` *หลังจากทำเสร็จแล้ว ผลลัพธ์ที่ได้ควรจะเหมือนกับผลลัพธ์ก่อนหน้านี้ เฉลย:* `http://thinkpython2.com/code/Time2_soln.py`

***แบบฝึกหัด* 17.2.** *แบบฝึกหัดนี้เป็นอุทาหรณ์เกี่ยวกับเรื่องที่พบบ่อยที่สุดเรื่องหนึ่งและหาข้อผิดพลาดได้ยากของไพธอน เขียนนิยามความของคลาสชื่อ* `Kangaroo` *พร้อมกับเมธอดต่อไปนี้*

1. `__init__` *เมธอดที่กำหนดค่าตั้งต้นให้กับแอตทริบิวต์ ชื่อ* `pouch_contents` *เป็นลิสต์ว่าง*

2. `put_in_pouch` *เมธอดที่รับอ็อบเจกต์ทุกชนิดและเพิ่มเข้าไปใน* `pouch_contents`

3. *เมธอด* `__str__` *ที่ส่งคืนค่าสายอักขระอธิบายอ็อบเจกต์ Kangaroo และรายละเอียดในกระเป๋า*

ทดสอบโค้ดของคุณโดยสร้างอ็อบเจกต์ **Kangaroo** *สองอัน และกำหนดให้กับตัวแปรที่ชื่อ* **kanga** *และ* **roo** *จากนั้นเพิ่ม* **roo** *ลงในกระเป๋าของ* **kanga**

ดาวน์โหลด `http://thinkpython2.com/code/BadKangaroo.py` *ซึ่งมีวิธีแก้ปัญหาก่อนหน้านี้ด้วยจุดบกพร่องขนาดใหญ่ที่น่ารังเกียจตัวหนึ่ง ค้นหาและแก้ไขจุดบกพร่องนั้น*

ถ้าติดขัดคุณสามารถดาวน์โหลด `http://thinkpython2.com/code/GoodKangaroo.py` *ซึ่งอธิบายปัญหาและสาธิตวิธีแก้ไข*

18. การสืบทอด

คุณลักษณะภาษาที่มักเกี่ยวข้องกับการเขียนโปรแกรมเชิงวัตถุคือ **การสืบทอด (inheritance)** การสืบทอดเป็นความสามารถในการกำหนดคลาสใหม่ที่เป็นรุ่นที่แก้ไขของคลาสที่มีอยู่เดิม ในบทนี้ผมสาธิตการสืบทอดโดยใช้คลาสที่แสดงถึงการเล่นไพ่ สำรับไพ่ และไพโป๊กเกอร์

ถ้าคุณไม่เล่นโป๊กเกอร์ สามารถอ่านได้ที่ `http://en.wikipedia.org/wiki/Poker` แต่คุณไม่จำเป็นต้องอ่าน ผมจะบอกคุณถึงสิ่งที่คุณต้องรู้สำหรับการฝึกหัด

ตัวอย่างโค้ดจากบทนี้หาได้จาก `http://thinkpython2.com/code/Card.py`

18.1. อ็อบเจกต์ไพ่

ในสำรับมีไพ่ 52 ใบ แบ่งออกเป็นสี่ชุด แต่ละชุดมี 13 อันดับ สี่ชุดนั้นได้แก่ โพธิ์ดำ โพธิ์แดง ข้าวหลามตัด และดอกจิก (เรียงจากมากไปน้อยในบริดจ์) เรียงตามลำดับดังนี้ เอซ, 2, 3, 4, 5, 6, 7, 8, 9, 10, แจ็ค, ควีน, และคิง เอซอาจสูงกว่าคิงหรือต่ำกว่า 2 ขึ้นอยู่กับเกมที่คุณกำลังเล่น

หากเราต้องการประกาศอ็อบเจกต์ใหม่เพื่อใช้แทนไพ่หนึ่งใบ เป็นที่ชัดเจนว่าแอตทริบิวต์ควรเป็นอันดับ `rank` และชุด `suit` แต่ไม่ชัดเจนว่าแอตทริบิวต์ควรจะเป็นข้อมูลชนิดใด ความเป็นไปได้อย่างหนึ่งคือการใช้สายอักขระที่มีคำเช่น `'Spade'` สำหรับชุด และควีน `'Queen'` สำหรับอันดับ ปัญหาหนึ่งของวิธีนี้คือ มันไม่ง่ายที่จะเปรียบเทียบไพ่เพื่อดูว่าใบใดมีอันดับหรือชุดที่สูงกว่า

อีกทางเลือกหนึ่งคือการใช้จำนวนเต็มเพื่อ**เข้ารหัส**อันดับและชุด ในบริบทนี้ “เข้ารหัส” (encode) หมายความว่าเราจะกำหนดการจับคู่ระหว่างตัวเลขกับชุด หรือระหว่างตัวเลขกับอันดับ การเข้ารหัสแบบนี้ไม่ได้ต้องการให้เป็นความลับ (ถ้าแบบนั้นจะเรียก “การเข้ารหัสลับ” หรือ “encryption”)

ตัวอย่างเช่น ตารางต่อไปนี้แสดงชุดและรหัสจำนวนเต็มที่เข้าคู่กัน

Spades ⟼ 3
Hearts ⟼ 2
Diamonds ⟼ 1
Clubs ⟼ 0

รหัสนี้ช่วยให้การเปรียบเทียบการ์ดทำได้ง่าย เนื่องจากชุดที่สูงกว่าจับคู่กับตัวเลขที่สูงกว่า เราสามารถเปรียบเทียบชุดโดยเปรียบเทียบรหัสของชุด

การจับคู่สำหรับอันดับนั้นค่อนข้างชัดเจน แต่ละอันดับของตัวเลขจะจับคู่กับจำนวนเต็มที่ตรงกัน และสำหรับไพ่รูปหน้า

Jack ⟼ 11
Queen ⟼ 12
King ⟼ 13

ผมใช้สัญลักษณ์ ⟼เพื่อให้ชัดเจนว่าการจับคู่เหล่านี้ไม่ได้เป็นส่วนหนึ่งของโปรแกรมไพธอน แต่เป็นส่วนหนึ่งของการออกแบบโปรแกรม และไม่ปรากฏอย่างชัดเจนในโค้ด

นิยามของคลาส Card มีลักษณะดังนี้

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

ตามปกติเมธอด init จะใช้พารามิเตอร์ทางเลือกสำหรับแต่ละแอตทริบิวต์ มีค่าเริ่มต้นเป็นอันดับ 2 ของชุดดอกจิก

ในการสร้าง Card คุณเรียกใช้ `Card` ด้วยชุดและอันดับของไพ่ที่คุณต้องการ

```
queen_of_diamonds = Card(1, 12)
```

18.2. แอตทริบิวต์ของคลาส

ในการพิมพ์อ็อบเจกต์ของ Card ในแบบที่ผู้คนสามารถอ่านได้ง่าย เราจำเป็นต้องมีการแปลงจากรหัสจำนวนเต็มไปยังอันดับและชุดที่ตรงกัน วิธีธรรมชาติในการทำเช่นนั้นคือใช้ลิสต์ของสายอักขระ เรากำหนด

รายการเหล่านี้ให้กับคลาสแอตทริบิวต์

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

ตัวแปรในลักษณะเดียวกับ `suit_names` และ `rank_names` ซึ่งถูกกำหนดไว้ภายในคลาสแต่อยู่นอกเมธอดใด ๆ จะถูกเรียกว่าคลาสแอตทริบิวต์เนื่องจากถูกผูกติดกับคลาสอ็อบเจ็คต์ Card

คลาสแอตทริบิวต์แยกความแตกต่างจากตัวแปร เช่น `suit` และ `rank` ซึ่งเรียกว่า **อินสแตนซ์แอตทริบิวต์ (instance attributes)** เนื่องจากพวกมันถูกผูกกับอินสแตนซ์เฉพาะราย

แอตทริบิวต์ทั้งสองชนิดสามารถเข้าถึงได้โดยใช้สัญกรณ์จุด ตัวอย่างเช่น ในเมธอด `__str__` `self` คืออ็อบเจกต์ Card และ `self.rank` คืออันดับ ในทำนองเดียวกัน `Card` เป็นคลาสอ็อบเจกต์ และ `Card.rank_names` คือลิสต์ที่ผูกติดอยู่กับคลาส

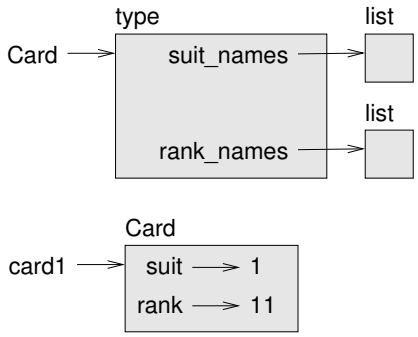
ไพ่ทุกใบมี `suit` และ `rank` ของตัวเอง แต่มี `suit_names` และ `rank_names` เพียงชุดเดียวเท่านั้น

เมื่อรวมทั้งหมดเข้าด้วยกัน นิพจน์ `Card.rank_names[self.rank]` หมายถึง “ใช้แอตทริบิวต์ `rank` จากอ็อบเจกต์ `self` เป็นดัชนีในลิสต์ `rank_names` จากคลาส `Card` และเลือกสายอักขระที่เหมาะสม”

อิลิเมนต์แรกของ `rank_names` คือ `None` เนื่องจากไม่มีไพ่ที่มีอันดับเป็นศูนย์ เมื่อรวม `None` ไว้เป็นตัวกันที่ เราจะได้การจับคู่ที่ดีที่ดัชนี 2 จับคู่กับข้อความ `'2'` และดัชนีอื่นๆ ก็เช่นกัน หากต้องการหลีกเลี่ยงการปรับแต่งนี้ เราสามารถใช้ดิกชันนารีแทนลิสต์ได้

ด้วยเมธอดที่เรามีก่อนหน้านี้ เราสามารถสร้างและพิมพ์ไพ่ได้

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```



รูปที่ 18.1.: แผนภาพอ็อบเจกต์

รูปที่ 18.1 เป็นแผนภาพของคลาสอ็อบเจกต์ `Card` และหนึ่งอินสแตนซ์ของ Card `Card` เป็นคลาสอ็อบเจกต์ จึงมีชนิดเป็น `type` ส่วน card1 เป็นอินสแตนซ์ของ `Card` จึงมีชนิดเป็น `Card` เพื่อประหยัดพื้นที่ ผมไม่ได้วาดเนื้อหาของ `suit_names` และ `rank_names`

18.3. การเปรียบเทียบไพ่

สำหรับชนิดข้อมูลสำเร็จรูป มีตัวดำเนินการเชิงสัมพันธ์ (เช่น <, >, ==) ที่เปรียบเทียบค่าและกำหนดว่าค่าใดค่าหนึ่งมากกว่า น้อยกว่า หรือเท่ากับค่าอื่น สำหรับชนิดข้อมูลที่กำหนดโดยโปรแกรมเมอร์ เราสามารถแทนที่พฤติกรรมของตัวดำเนินการสำเร็จรูปได้โดยการจัดเตรียมเมธอดที่ชื่อ `__lt__` ซึ่งย่อมาจาก “less than”

`__lt__` รับพารามิเตอร์สองตัวคือ `self` และ `other` และคืนค่า `True` หาก `self` มีค่าน้อยกว่า `other` อย่างแน่นอน

ลำดับที่ถูกต้องสำหรับการ์ดไม่ชัดเจน ตัวอย่างเช่น อันไหนดีกว่าระหว่าง 3 ของดอกจิก หรือ 2 ของข้าวหลามตัด? ใบหนึ่งมีอันดับสูงกว่า แต่อีกใบมีชุดสูงกว่า เพื่อเปรียบเทียบไพ่ คุณต้องตัดสินใจว่าอันดับหรือชุดมีความสำคัญมากกว่า

คำตอบอาจขึ้นอยู่กับว่าคุณกำลังเล่นเกมอะไรอยู่ แต่เพื่อให้ง่ายขึ้น เราจะทำการเลือกตามอำเภอใจซึ่งชุดนั้นสำคัญกว่า ดังนั้นโพดำทั้งหมดจึงมีอันดับเหนือกว่าข้าวหลามตัดทั้งหมด และอื่นๆ

ด้วยการตัดสินใจนั้น เราสามารถเขียน `__lt__` ดังนี้

```
# inside class Card:
```

```
    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

คุณสามารถเขียนให้กระชับยิ่งขึ้นได้โดยใช้การเปรียบเทียบทูเพิล

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

เพื่อเป็นการฝึก ให้เขียนเมธอด `__lt__` สำหรับอ็อบเจกต์ Time คุณสามารถใช้การเปรียบเทียบทูเพิล แต่คุณอาจพิจารณาเปรียบเทียบจำนวนเต็มด้วย

18.4. สำรับ

ตอนนี้เรามีไพ่แล้ว ขั้นตอนต่อไปคือการประกาศ สำรับ (Deck) เนื่องจากสำรับประกอบด้วยไพ่ จึงเป็นเรื่องธรรมดาที่แต่ละสำรับจะมีรายการไพ่เป็นแอตทริบิวต์

ต่อไปนี้เป็นนิยามสำหรับคลาส Deck เมธอด init สร้างแอตทริบิวต์ cards และสร้างชุดมาตรฐานของไพ่ห้าสิบสองใบ

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
```

```
                card = Card(suit, rank)
                self.cards.append(card)
```

วิธีที่ง่ายที่สุดในการเติมข้อมูลสำรับคือการใช้*ลูปซ้อน* (*nested loop*) วงนอกระบุชุดจาก 0 ถึง 3 วงในระบุอันดับจาก 1 ถึง 13 การวนซ้ำแต่ละครั้งจะสร้างการ์ดใหม่ด้วยชุดและอันดับปัจจุบัน และผนวกเข้ากับ `self.cards`

18.5. การพิมพ์สำรับ

นี่คือเมธอด `__str__` สำหรับ `Deck`

```
#inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

เมธอดนี้แสดงให้เห็นถึงวิธีที่มีประสิทธิภาพในการเก็บสายอักขระขนาดใหญ่ โดยสร้างลิสต์ของสายอักขระแล้วใช้เมธอด `join` ของสายอักขระ ฟังก์ชันสำเร็จรูป `str` เรียกใช้เมธอด `__str__` ของไพ่แต่ละใบและส่งคืนการแสดงผลของสายอักขระ

เนื่องจากเราเรียก `join` บนอักขระขึ้นบรรทัดใหม่ การ์ดแต่ละใบจะถูกคั่นด้วยการขึ้นบรรทัดใหม่

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

แม้ว่าผลลัพธ์จะปรากฏใน 52 บรรทัด แต่เป็นเพียงหนึ่งสายอักขระยาวที่มีการขึ้นบรรทัดใหม่แทรกอยู่

18.6. เพิ่ม ลบ สับเปลี่ยน และจัดเรียง

ในการแจกไพ่ เราต้องการวิธีที่จะนำไพ่ออกจากสำรับและส่งคืนค่านั้น เมธอด pop ของลิสต์เป็นวิธีที่สะดวกในการทำเช่นนั้น

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

เนื่องจาก pop ดึงเอาไพ่*ใบสุดท้าย*ในลิสต์ออก เราจึงแจกไพ่จากด้านล่างของสำรับ

ในการเพิ่มการ์ด เราสามารถใช้เมธอด append ของ list

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

วิธีการแบบนี้ที่ใช้เมธอดอื่นโดยไม่ต้องทำอะไรมาก บางครั้งเรียกว่า **วีเนียร์ (veneer)** คำอุปมานี้มาจากงานไม้ โดยที่แผ่นไม้อัดเป็นชั้นไม้คุณภาพดีบางๆ ที่ติดกาวบนพื้นผิวของแผ่นไม้ที่ราคาถูกกว่าเพื่อปรับปรุงรูปลักษณ์

ในกรณีนี้ add_card เป็น เมธอด “บาง” ที่ใช้การดำเนินการของลิสต์ในแง่ที่เหมาะสมสำหรับสำรับ เพื่อปรับปรุงลักษณะที่ปรากฏหรือส่วนต่อประสานของการใช้งาน

อีกตัวอย่างหนึ่ง เราสามารถเขียนเมธอดของ Deck ชื่อ shuffle โดยใช้ฟังก์ชัน shuffle จากโมดูล random:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

(อย่าลืม import random)

เพื่อเป็นการฝึกหัด ให้เขียนเมธอดของ Deck ชื่อ `sort` ซึ่งใช้เมธอด `sort` ของลิสต์ เพื่อจัดเรียงไพ่ในสำรับ โดยให้ sort ใช้เมธอด `__lt__` ที่เราได้นิยามไว้เพื่อกำหนดลำดับก่อนหลัง

18.7. การสืบทอด

การสืบทอด (inheritance) คือความสามารถในการสร้างคลาสใหม่ที่เป็นรุ่นที่แก้ไขของคลาสเดิม ตัวอย่างเช่น สมมติว่าเราต้องการให้คลาสเป็นตัวแทนของ “มือไพ่” หรือมือ (ภาษาอังกฤษใช้ hand ภาษาไทยทั่วไปอาจเรียก ขา) ที่เป็นไพ่ต่าง ๆ ที่ผู้เล่นคนหนึ่งถือไว้ในมือ คลาส Hand นี้ก็จะคล้ายกันกับคลาส Deck คือทั้งสองประกอบด้วยชุดไพ่ และทั้งคู่ต้องมีการดำเนินการ เช่น การเพิ่มและนำไพ่ออก

คลาส Hand ก็แตกต่างจากคลาส Deck คือมีการดำเนินการที่เราต้องการสำหรับคลาส Hand ที่ไม่สมเหตุสมผลสำหรับคลาส Deck ตัวอย่างเช่น ในโป๊กเกอร์ เราอาจเปรียบเทียบสองมือเพื่อดูว่าใครชนะ ในบริดจ์เราอาจคำนวณคะแนนของมือเพื่อต่อรอง

ความสัมพันธ์ระหว่างคลาสนี้คล้ายกัน แต่ไม่เหมือนกันซะทีเดียว ลักษณะเช่นนี้เหมาะแก่การทำการสืบทอด ในการประกาศคลาสใหม่ที่สืบทอดมาจากคลาสเดิมที่มีอยู่ เราแค่ใส่ชื่อของคลาสเดิมในวงเล็บ:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

ในการประกาศนี้บอกว่าคลาส **Hand** สืบทอดมาจากคลาส **Deck** นั่นหมายความว่าเราสามารถใช้เมธอดต่าง ๆ เช่น `pop_card` และ `add_card` สำหรับ Hands ได้เช่นเดียวกับ Decks

เมื่อคลาสใหม่สืบทอดมาจากคลาสเดิม คลาสเดิมจะเรียกว่า **พ่อแม่ (parent)** และคลาสใหม่จะถูกเรียกว่า **ลูก (child)**

ในตัวอย่างนี้ คลาส **Hand** สืบทอด `__init__` มาจากคลาส **Deck** แต่มันไม่ได้ทำในสิ่งที่เราต้องการ นั่นคือ แทนที่จะเติมไพ่ใหม่ 52 ใบให้ Hand เมธอด init สำหรับคลาส Hand ควรกำหนดค่าเริ่มต้นให้ตัวแปร `cards` ด้วยลิสต์ว่าง

ถ้าเราจัดเตรียมเมธอด init ในคลาส **Hand** มันจะไปแทนที่เมธอดในคลาส **Deck**

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

เมื่อคุณสร้างอินสแตนซ์ของ Hand ไพธอนจะเรียกใช้เมธอด init นี้ ไม่ใช่เมธอด init ใน Deck

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

เมธอดอื่น ๆ นั้นได้รับสืบทอดมาจากคลาส Deck ดังนั้นเราจึงสามารถใช้ pop_card และ add_card เพื่อแจกไพ่ได้

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

ขั้นตอนต่อไปคือการห่อหุ้มโค้ดนี้ด้วยเมธอด move_cards

```
#inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

เมธอด move_cards รับสองอาร์กิวเมนต์ ซึ่งคือ อ็อบเจกต์ hand และจำนวนไพ่ที่จะแจกให้ มันเปลี่ยนค่าของ self และ hand แล้วส่งคืนค่า None ออกมา

ในบางเกม ไพ่จะถูกย้ายจากมือหนึ่งไปอีกมือหนึ่ง หรือจากมือกลับไปที่สำรับ คุณสามารถใช้ move_cards สำหรับการดำเนินการใด ๆ เหล่านี้ เพราะ self สามารถเป็น Deck หรือ Hand ก็ได้ ไม่ว่าจะเป็นชนิดใด ต่างก็สามารถใช้ในนาม Deck ได้

การสืบทอดมีประโยชน์มาก ถ้าไม่มีการสืบทอด หลาย ๆ โปรแกรมจะเขียนซ้ำ ๆ กัน แต่หากใช้การสืบทอด โปรแกรมเหล่านี้จะสามารถเขียนได้อย่างสละสลวย การสืบทอดช่วยอำนวยความสะดวกในการนำโค้ดมาใช้ซ้ำ เพราะว่า เราสามารถปรับแต่งพฤติกรรมของคลาสพ่อแม่โดยไม่ต้องไปแก้ไขพวกมัน ในบางกรณี โครงสร้างการสืบทอดจะสะท้อนถึงโครงสร้างตามธรรมชาติของปัญหา ซึ่งทำให้การออกแบบโปรแกรมดูเข้าใจง่ายขึ้น

ในทางกลับกัน การสืบทอดอาจทำให้โปรแกรมอ่านยาก เมื่อมีการเรียกใช้เมธอด บางครั้งก็ไม่ชัดเจนว่าจะหานิยามได้จากที่ใด รหัสที่เกี่ยวข้องอาจกระจายไปทั่วหลายโมดูล นอกจากนี้ หลายๆ อย่างที่สามารถทำได้โดยใช้การสืบทอด ก็สามารถทำได้เช่นกันหรือทำได้ดีกว่าโดยไม่ใช้การสืบทอด

18.8. แผนภาพคลาส

จนถึงตอนนี้ เราได้เห็นแผนภาพแบบกองซ้อน ซึ่งแสดงสถานะของโปรแกรม และแผนภาพของอ็อบเจกต์ ซึ่งแสดงแอตทริบิวต์ของอ็อบเจกต์และค่าของมัน แผนภาพเหล่านี้เป็นเสมือนภาพนิ่งในการทำงานของโปรแกรม ดังนั้นแผนภาพเหล่านี้(ซึ่งสะท้อนสถานะกับค่าแอตทริบิวต์)จึงเปลี่ยนแปลงไปขณะโปรแกรมทำงาน

นอกจากนี้ยังมีรายละเอียดมาก สำหรับจุดประสงค์บางอย่าง แผนภาพเหล่านี้ละเอียดเกินไป แผนภาพคลาสเป็นตัวแทนของโครงสร้างของโปรแกรมที่เป็นนามธรรมมากขึ้น แทนที่จะแสดงแต่ละอ็อบเจกต์ แผนภาพคลาสจะแสดงคลาสและความสัมพันธ์ระหว่างคลาสต่าง ๆ

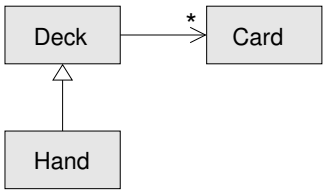
มีความสัมพันธ์ระหว่างคลาสหลายประเภท

- อ็อบเจกต์ในคลาสหนึ่งอาจมีอ็อบเจกต์คลาสอื่นอยู่[1] ตัวอย่างเช่น อ็อบเจกต์สี่เหลี่ยม Rectangle แต่ละอันมีอ็อบเจกต์จุด Point และแต่ละสำรับ Deck มีไพ่ Card อยู่หลายอ็อบเจกต์ ความสัมพันธ์แบบนี้เรียกว่า **มี (HAS-A)** เช่น “สี่เหลี่ยมมีจุด” (a Rectangle has a Point)
- คลาสหนึ่งอาจสืบทอดมาจากอีกคลาสหนึ่ง ความสัมพันธ์นี้เรียกว่า **เป็น (IS-A)** เช่น “Hand เป็น Deck ชนิดหนึ่ง” (a Hand is a kind of a Deck)
- คลาสหนึ่งอาจขึ้นอยู่กับอีกคลาสหนึ่งในแง่ที่ว่าอ็อบเจ็คต์ในคลาสหนึ่งใช้อ็อบเจ็คต์ในคลาสที่สองเป็นพารามิเตอร์ หรือใช้อ็อบเจ็คต์ในคลาสที่สองเป็นส่วนหนึ่งของการคำนวณ ความสัมพันธ์แบบนี้เรียกว่า **การขึ้นต่อกัน (dependency)**

แผนภาพคลาสเป็นการแสดงกราฟิกของความสัมพันธ์เหล่านี้ ตัวอย่างเช่น รูปที่ 18.2 แสดงความสัมพันธ์ระหว่าง **Card**, **Deck** และ **Hand**

ลูกศรที่มีหัวสามเหลี่ยมกลวงแสดงถึงความสัมพันธ์แบบ เป็น (IS-A) ในกรณีนี้แสดงว่า Hand นั้นสืบทอดมาจาก Deck

[1] การที่อ็อบเจกต์หนึ่งมีอ็อบเจกต์อื่นอยู่ จริง ๆ แล้วก็คือ อ็อบเจกต์นั้นมีจุดอ้างอิงถึงอ็อบเจกต์อื่น



รูปที่ 18.2.: แผนภาพคลาส

หัวลูกศรมาตรฐานแสดงถึงความสัมพันธ์แบบ มี (HAS-A) ในกรณีนี้ Deck มีการอ้างอิงถึงอ็อบเจ็คต์ Card หลาย ๆ อ็อบเจ็คต์

ดาว (*) ใกล้หัวลูกศรเป็น**ความหลากหลาย (multiplicity)** มันบ่งบอกว่า Deck มี Card กี่อ็อบเจ็คต์ ความหลายหลากอาจเป็นตัวเลขธรรมดา เช่น `52` หรือเป็นช่วง เช่น `5..7` หรือเป็นดาว ซึ่งบ่งชี้ว่า Deck สามารถมี Card เป็นจำนวนเท่าใดก็ได้

ไม่มีการพึ่งพาแสดงในแผนภาพนี้ โดยปกติแล้ว การพึ่งพาจะแสดงด้วยลูกศรประ หรือหากมีการพึ่งพากันมากบางครั้งก็ละเว้น

แผนภาพที่มีรายละเอียดมากขึ้นอาจแสดงว่า Deck มี*ลิสต์*ของ Card อยู่ แต่ชนิดข้อมูลสำเร็จ เช่น ลิสต์และดิกต์มักจะไม่นิยมเขียนในแผนภาพคลาส

18.9. การดีบัก

การสืบทอดอาจทำให้การดีบักทำได้ยาก เนื่องจากเมื่อเราเรียกใช้เมธอดของอ็อบเจกต์ อาจจะยากที่จะรู้ว่าเมธอดไหนจะถูกเรียกใช้

สมมติว่าคุณกำลังเขียนฟังก์ชันที่ทำงานกับอ็อบเจกต์ Hand คุณต้องการให้มันทำงานกับทุกประเภทของ Hand เช่น PokerHands, BridgeHands เป็นต้น หากคุณเรียกใช้เมธอดเช่น `shuffle` คุณอาจได้เมธอดที่กำหนดไว้ใน `Deck` แต่ถ้าคลาสย่อยใดมีเมธอดแทนที่เมธอดจากคลาสพ่อแม่ คุณจะได้เมธอดนั้นแทน พฤติกรรมแบบนี้ส่วนใหญ่แล้วดีช่วยให้เขียนโปรแกรมง่าย แต่เวลาดีบักก็อาจทำให้สับสนได้

ทุกครั้งที่คุณไม่แน่ใจเกี่ยวกับขั้นตอนการทำงานภายในโปรแกรมของคุณ วิธีที่ง่ายที่สุดคือการเพิ่มคำสั่งการพิมพ์ที่จุดเริ่มต้นของเมธอดที่เกี่ยวข้อง ถ้าเป็น `Deck.shuffle` ก็พิมพ์ข้อความที่เขียนว่า `Running Deck.shuffle` เมื่อโปรแกรมรันจะแสดงร่องรอยของขั้นตอนการทำงานภายในโปรแกรมออกมา

อีกทางเลือกหนึ่ง คุณสามารถใช้ฟังก์ชันต่อไปนี้ ซึ่งรับชื่ออ็อบเจกต์และเมธอด (เป็นสายอักขระ) แล้วส่งคืนคลาสที่มีนิยามของเมธอด

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

นี่เป็นตัวอย่างการใช้

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

ดังนั้นเมธอด `shuffle` ของ Hand นี้เป็นเมธอดหนึ่งในคลาส `Deck`

ฟังก์ชัน `find_defining_class` ใช้เมธอด `mro` เพื่อรับลิสต์ของคลาสต่าง ๆ แล้วเข้าไปค้นหาเมธอดที่สงสัยในแต่ละคลาส แล้วส่งคืนคลาสที่มีเมธอดที่สงสัยออกมา ชื่อ “MRO” ย่อมาจาก “method resolution order” ซึ่งเป็นลำดับของคลาสที่ไพธอนค้นเพื่อหาอิมพลิเมนเตชั่นของเมธอด (“resolve” a method name)

นี่คือคำแนะนำการออกแบบ: เมื่อคุณเขียนเมธอดแทนที่เมธอดเดิม ส่วนต่อประสานของเมธอดใหม่ควรเหมือนกับเมธอดเดิม ควรใช้พารามิเตอร์เดียวกัน ส่งคืนค่าประเภทเดียวกัน และปฏิบัติตามเงื่อนไขก่อนและเงื่อนไขภายหลังเดียวกัน หากคุณทำตามกฎนี้ คุณจะพบว่าฟังก์ชันใด ๆ ที่ออกแบบมาเพื่อทำงานกับอินสแตนซ์ของคลาสหลัก เช่น Deck จะทำงานกับอินสแตนซ์ของคลาสย่อย เช่น Hand และ PokerHand ได้

หากคุณละเมิดกฎนี้ ซึ่งเรียกว่า “หลักการทดแทนของลิสคอฟ” (Liskov substitution principle) โค้ดของของคุณจะพังแบบเดียวกับชีวิตนักพนัน

18.10. การห่อหุ้มข้อมูล

บทก่อนหน้านี้แสดงให้เห็นถึงแผนการพัฒนาที่เราอาจเรียกว่า “การออกแบบเชิงวัตถุ” เราระบุอ็อบเจกต์ที่เราต้องการ เช่น `Point`, `Rectangle` และ `Time` และกำหนดคลาสเพื่อเป็นตัวแทนของเหล่านี้ แต่ละกรณี มีความสอดคล้องกันอย่างชัดเจนระหว่างอ็อบเจกต์ในโปรแกรมกับวัตถุในโลกแห่งความเป็นจริง (หรืออย่างน้อยก็โลกทางคณิตศาสตร์)

แต่บางครั้งมันก็ไม่ชัดเจนว่าเราต้องการอ็อบเจกต์อะไรบ้างและอ็อบเจกต์ต่าง ๆ ควรมีปฏิสัมพันธ์ต่อกันอย่างไร ถ้าเป็นแบบนั้น เราต้องมีแผนการพัฒนาที่แตกต่างออกไป ในลักษณะเดียวกับที่เราค้นพบส่วนต่อประสานของฟังก์ชันโดยการห่อหุ้มและการวางนัยทั่วไป เราสามารถค้นพบส่วนต่อประสานของคลาสได้โดย**การห่อหุ้มข้อมูล (data encapsulation)**

การวิเคราะห์ Markov จากส่วนที่ 13.8 เป็นตัวอย่างที่ดี หากคุณดาวน์โหลดโค้ดของผมจาก `http://thinkpython2.com/code/markov.py` คุณจะเห็นว่ามีการใช้ตัวแปรส่วนกลางสองตัว `suffix_map` และ `prefix` ที่ถูกอ่านและเขียนจากหลายฟังก์ชัน

```
suffix_map = {}
prefix = ()
```

เนื่องจากตัวแปรเหล่านี้เป็นตัวแปรส่วนกลาง เราจึงสามารถเรียกใช้การวิเคราะห์ได้ครั้งละหนึ่งรายการเท่านั้น ถ้าเราอ่านสองข้อความ คำนำหน้าและส่วนต่อท้ายจะถูกเพิ่มในโครงสร้างข้อมูลเดียวกัน (ซึ่งทำให้บางข้อความที่สร้างขึ้นน่าสนใจ)

เพื่อที่จะเรียกใช้การวิเคราะห์หลายรายการและวิเคราะห์แยกกัน เราสามารถห่อหุ้มสถานะของการวิเคราะห์แต่ละรายการในอ็อบเจกต์ได้ มีหน้าตาประมาณนี้

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

ถัดไปเราแปลงฟังก์ชันเป็นเมธอด ตัวอย่างเช่น `process_word` ดังต่อไปนี้

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
```

```
self.suffix_map[self.prefix] = [word]
```

```
self.prefix = shift(self.prefix, word)
```

การแปลงโปรแกรมในลักษณะนี้ คือการเปลี่ยนการออกแบบโดยไม่เปลี่ยนลักษณะการทำงาน เป็นอีกตัวอย่างหนึ่งของการปรับโครงสร้างใหม่ (ดูส่วนที่ 4.7)

ตัวอย่างนี้แนะนำแผนการพัฒนาสำหรับการออกแบบอ็อบเจกต์และเมธอด

1. เริ่มต้นด้วยการเขียนฟังก์ชันที่อ่านและเขียนตัวแปรส่วนกลาง (เมื่อจำเป็น)
2. เมื่อคุณทำให้โปรแกรมทำงานได้ ให้มองหาความสัมพันธ์ระหว่างตัวแปรส่วนกลางและฟังก์ชันที่ใช้ตัวแปรเหล่านั้น
3. ห่อหุ้มตัวแปรที่เกี่ยวข้องเป็นแอตทริบิวต์ของอ็อบเจกต์
4. แปลงฟังก์ชันที่เกี่ยวข้องเป็นเมธอดของคลาสใหม่

เพื่อเป็นการฝึกหัด ให้ดาวน์โหลดโค้ด Markov ของผมจาก `http://thinkpython2.com/code/markov.py` และทำตามขั้นตอนที่อธิบายไว้ข้างต้นเพื่อห่อหุ้มตัวแปรส่วนกลางเป็นแอตทริบิวต์ของคลาสใหม่ที่เรียกว่า `Markov` เฉลย: `http://thinkpython2.com/code/Markov.py` (สังเกตตัวพิมพ์ใหญ่ M)

18.11. อภิธานศัพท์

เข้ารหัส (encode): การแทนค่าชุดหนึ่งด้วยชุดค่าอื่นโดยสร้างการแปลงระหว่างค่าเหล่านี้

คลาสแอตทริบิวต์ (class attribute): แอตทริบิวต์ที่เกี่ยวข้องกับคลาสอ็อบเจกต์ ซึ่งถูกประกาศไว้ภายในนิยามคลาส แต่อยู่นอกเมธอดใด ๆ

อินสแตนซ์แอตทริบิวต์ (instance attribute): แอตทริบิวต์ที่เกี่ยวข้องกับอินสแตนซ์ของคลาส

วีเนียร์ (veneer): เมธอดหรือฟังก์ชันที่ให้ส่วนต่อประสานที่แตกต่างกันไปยังฟังก์ชันอื่นโดยไม่ต้องคำนวณมาก

การสืบทอด(inheritance): ความสามารถในการนิยามคลาสใหม่ที่เป็นรุ่นที่แก้ไขของคลาสที่นิยามไว้ก่อนหน้านี้

คลาสพ่อแม่ (parent class): คลาสที่คลาสลูกสืบทอดมา

คลาสลูก (child class): คลาสใหม่ที่สร้างขึ้นโดยสืบทอดจากคลาสที่มีอยู่ เรียกอีกอย่างว่า “คลาสย่อย”

ความสัมพันธ์แบบเป็น (IS-A relationship): ความสัมพันธ์ระหว่างคลาสลูกกับคลาสพ่อแม่

ความสัมพันธ์แบบมี (HAS-A relationship): ความสัมพันธ์ระหว่างสองคลาสโดยที่อินสแตนซ์ของคลาสหนึ่งมีการอ้างอิงถึงอินสแตนซ์ของอีกคลาสหนึ่ง

การขึ้นต่อกัน (dependency): ความสัมพันธ์ระหว่างสองคลาสโดยที่อินสแตนซ์ของคลาสหนึ่งใช้อินสแตนซ์ของคลาสอื่น แต่ไม่เก็บไว้เป็นแอตทริบิวต์

แผนภาพคลาส (class diagram): แผนภาพที่แสดงคลาสในโปรแกรมและความสัมพันธ์ระหว่างคลาสต่าง ๆ

ความหลายหลาก (multiplicity): สัญกรณ์ในแผนภาพคลาสที่แสดงสำหรับความสัมพันธ์แบบมี ว่ามีการอ้างอิงถึงอินสแตนซ์ของคลาสอื่นกี่รายการ

การห่อหุ้มข้อมูล (data encapsulation): แผนการพัฒนาโปรแกรมที่เกี่ยวข้องกับต้นแบบโดยใช้ตัวแปรส่วนกลางและเวอร์ชันสุดท้ายที่ทำให้ตัวแปรส่วนกลางเป็นแอตทริบิวต์ของอินสแตนซ์

18.12. แบบฝึกหัด

***แบบฝึกหัด* 18.1.** *สำหรับโปรแกรมต่อไปนี้ ให้วาดแผนภาพคลาส UML ที่แสดงคลาสเหล่านี้และความสัมพันธ์ระหว่างคลาสเหล่านี้*

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong
```

```
class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

แบบฝึกหัด 18.2. *เขียนเมธอดสำหรับ Deck ที่เรียกว่า* `deal_hands` *ซึ่งใช้พารามิเตอร์สองตัว คือ จำนวนมือและจำนวนไพ่ต่อมือ มันควรสร้างอ็อบเจกต์ Hand ออกมาตามจำนวนที่เหมาะสม แจกไพ่ตามจำนวนที่เหมาะสมต่อแต่ละมือ และส่งคืนลิสต์ของอ็อบเจกต์ Hand ต่าง ๆ*

แบบฝึกหัด 18.3. *ต่อไปนี้เป็นมือที่เป็นไปได้ในโป๊กเกอร์ โดยเรียงตามมูลค่าที่เพิ่มขึ้นและลำดับความน่าจะเป็นที่ลดลง*

คู่ (pair): *ไพ่สองใบที่มีอันดับเดียวกัน*

สองคู่ (two pair): *ไพ่สองคู่ที่มีอันดับเดียวกัน*

ตอง (three of a kind): *ไพ่สามใบที่มีอันดับเดียวกัน*

สเตรท (straight): *ไพ่ห้าใบที่มีอันดับตามลำดับ (เอซสามารถสูงหรือต่ำได้ ดังนั้น* `Ace-2-3-4-5` *จึงเป็นไพ่สเตรท และ* `10-Jack-Queen-King-Ace` *ก็เช่นกัน แต่* `Queen-King-Ace-2-3` *ไม่ใช่.)*

ฟลัช (flush): *ไพ่ห้าใบที่เป็นชุดเดียวกัน*

ฟูลเฮ้าส์ (full house): *มี 1 ตอง และ 1 คู่*

โฟร์การ์ด (four of a kind): *ไพ่ 4 ใบแต้มเหมือนกัน*

สเตรทฟลัช (straight flush): *ไพ่ห้าใบเรียงตามลำดับ (ตามที่กำหนดไว้ข้างต้น) และเป็นชุดเดียวกัน*

เป้าหมายของแบบฝึกหัดเหล่านี้คือการประมาณความน่าจะเป็นของการจั่วไพ่แบบต่าง ๆ เหล่านี้

1. ดาวน์โหลดไฟล์ต่อไปนี้จาก `http://thinkpython2.com/code`

   `Card.py` เวอร์ชันที่สมบูรณ์ของคลาส `Card`, `Deck` และ `Hand` ในบทนี้

   `PokerHand.py` การพัฒนามือของโป๊กเกอร์ที่ยังไม่สมบูรณ์ และโค้ดบางส่วนที่ใช้ทดสอบ

2. หากคุณใช้ **`PokerHand.py`** จะแจกไพ่โป๊กเกอร์ 7 ใบ และตรวจดูว่ามีไพ่ในมือที่เป็นฟลัชหรือไม่ อ่านรหัสนี้อย่างละเอียดก่อนดำเนินการต่อ

3. เพิ่มเมธอดที่ชื่อว่า **`has_pair`**, **`has_twopair`** เป็นต้น ไปยัง **`PokerHand.py`** ซึ่งคืนค่าเป็น True หรือ False โดยขึ้นอยู่กับว่ามือนั้นตรงตามเกณฑ์ที่เกี่ยวข้องหรือไม่ โค้ดของคุณควรทำงานอย่างถูกต้องสำหรับ "มือ" ที่มีการ์ดจำนวนเท่าใดก็ได้ (แม้ว่า 5 และ 7 จะเป็นขนาดทั่วไป)

4. เขียนเมธอดที่ชื่อว่า `classify` ซึ่งคำนวณการจำแนกประเภทที่มีมูลค่าสูงสุดสำหรับมือและตั้งค่าแอตทริบิวต์ `label` ตามนั้น ตัวอย่างเช่น ไพ่ 7 ใบอาจมีฟลัชและคู่ ควรมีข้อความว่า "flush"

5. เมื่อคุณมั่นใจว่าวิธีการจำแนกของคุณได้ผล ขั้นตอนต่อไปคือการประมาณความน่าจะเป็นของมือต่าง ๆ เขียนฟังก์ชันใน **`PokerHand.py`** ที่สับไพ่สำรับ แบ่งออกเป็นมือ จำแนกมือ และนับจำนวนครั้งที่การจำแนกประเภทต่าง ๆ ปรากฏขึ้น

6. พิมพ์ตารางการจำแนกประเภทและความน่าจะเป็น รันโปรแกรมของคุณด้วยจำนวนมือที่มากขึ้นเรื่อยๆ จนกว่าค่าเอาต์พุตจะบรรจบกันในระดับความแม่นยำที่เหมาะสม เปรียบเทียบผลลัพธ์ของคุณกับค่าต่าง ๆ ที่ `http://en.wikipedia.org/wiki/Hand_rankings`

เฉลย: http://thinkpython2.com/code/PokerHandSoln.py

19. ของดี ๆ

เป้าหมายอย่างหนึ่งของผมสำหรับหนังสือเล่มนี้คือการสอนไพธอนให้น้อยที่สุด เมื่อมีสองวิธีในการทำบางสิ่ง ผมเลือกวิธีหนึ่งและหลีกเลี่ยงการพูดถึงอีกวิธีหนึ่ง หรือบางครั้งผมก็เอาอันที่สองไปเป็นแบบฝึกหัด

ตอนนี้ผมอยากกลับไปยังสิ่งดี ๆ ที่ทิ้งไว้เบื้องหลัง ไพธอนมีคุณสมบัติหลายอย่างที่ไม่จำเป็นจริงๆ คุณสามารถเขียนโค้ดที่ดีได้โดยไม่ต้องใช้มัน แต่ด้วยคุณสมบัติเหล่านี้ คุณสามารถเขียนโค้ดที่กระชับ อ่านได้ หรือมีประสิทธิภาพมากกว่า และบางครั้งก็มีทั้งสามอย่าง

19.1. นิพจน์เงื่อนไข

เราเห็นคำสั่งแบบมีเงื่อนไขในหัวข้อ 5.4 คำสั่งแบบมีเงื่อนไขมักใช้เพื่อเลือกค่าใดค่าหนึ่งจากสองค่า ตัวอย่างเช่น

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

คำสั่งนี้ตรวจสอบว่า `x` เป็นบวกหรือไม่ ถ้าใช่ มันจะคำนวณ `math.log` ถ้าไม่เช่นนั้น `math.log` จะเพิ่ม ValueError เพื่อหลีกเลี่ยงไม่ให้โปรแกรมหยุดทำงาน เราจึงสร้าง “NaN” ซึ่งเป็นค่าทศนิยมพิเศษที่แทนค่า “ไม่ใช่ตัวเลข (Not a Number)”

เราสามารถเขียนคำสั่งนี้ให้กระชับยิ่งขึ้นโดยใช้นิพจน์เงื่อนไข

```
y = math.log(x) if x > 0 else float('nan')
```

คุณเกือบจะอ่านบรรทัดนี้ได้เหมือนภาษาอังกฤษ: “`y` gets log-`x` if `x` is more than 0; otherwise it gets NaN” นั่นคือ y ได้รับ log-x ถ้า x มากกว่า 0; มิฉะนั้นจะได้รับ NaN

บางครั้งฟังก์ชันแบบเรียกซ้ำสามารถเขียนใหม่ได้โดยใช้นิพจน์เงื่อนไข ตัวอย่างนี้คือเวอร์ชันแบบเรียกซ้ำของ factorial

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

เราสามารถเขียนใหม่ได้ดังนี้

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

การใช้นิพจน์เงื่อนไขอีกวิธีหนึ่งคือการจัดการอาร์กิวเมนต์ที่เป็นทางเลือก ตัวอย่างนี้คือเมธอด init จาก GoodKangaroo (ดูแบบฝึกหัดที่ 17.2)

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

เราสามารถเขียนใหม่ได้ดังนี้

```
    def __init__(self, name, contents=None):
        self.name = name
      self.pouch_contents = [] if contents == None else contents
```

โดยทั่วไป คุณสามารถแทนที่คำสั่งแบบมีเงื่อนไขด้วยนิพจน์เงื่อนไข ถ้าทั้งสองทางเลือกมีนิพจน์ทั่วไปที่ส่งคืนหรือกำหนดให้กับตัวแปรเดียวกัน

19.2. การสรุปความลิสต์ (List comprehensions)

ในหัวข้อ 10.7 เราเห็นการแปลงและรูปแบบการกรอง ตัวอย่างเช่น ฟังก์ชันนี้รับลิสต์ของสายอักขระ แปลงแต่ละอีลีเมนต์ให้เป็นcapitalize ด้วยเมธอดของสายอักขระและส่งคืนลิสต์ใหม่ของสายอักขระ:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

เราสามารถเขียนให้กระชับยิ่งขึ้นโดยใช้**การสรุปความลิสต์** (list comprehension)

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

ตัวดำเนินการวงเล็บบ่งบอกว่าเรากำลังสร้างลิสต์ใหม่ นิพจน์ภายในวงเล็บจะระบุอิลิเมนต์ของลิสต์ และส่วนคำสั่ง `for` ระบุลำดับที่เรากำลังข้ามผ่าน

ไวยากรณ์ของการสรุปความลิสต์ นั้นดูอึดอัดเล็กน้อย เนื่องจากตัวแปรของลูป `s` ในตัวอย่างนี้ ปรากฏในนิพจน์ก่อนที่เราจะเจอส่วนของนิยาม

การสรุปความลิสต์ยังสามารถใช้สำหรับการกรอง ตัวอย่างเช่น ฟังก์ชันนี้จะเลือกเฉพาะอิลิเมนต์ของ `t` ที่เป็นตัวพิมพ์ใหญ่ และส่งคืนลิสต์ใหม่ออกมา

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

เราสามารถเขียนใหม่ได้โดยใช้การสรุปความลิสต์

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

การสรุปความลิสต์มีความกระชับและอ่านง่าย อย่างน้อยก็สำหรับนิพจน์ง่าย ๆ และมักจะเร็วกว่าลูปที่เทียบเท่ากัน บางครั้งเร็วกว่ามาก

อย่างไรก็ตาม การสรุปความลิสต์นั้นยากต่อการดีบัก เพราะเราไม่สามารถใส่คำสั่งพิมพ์ในลูปได้ ผมแนะนำให้คุณใช้มันก็ต่อเมื่อการคำนวณง่ายพอที่คุณจะทำได้ถูกในทีเดียว และสำหรับผู้เริ่มต้น นั่นหมายถึงอย่าใช้

19.3. นิพจน์ตัวสร้าง

นิพจน์ตัวสร้าง (**generator expressions**) คล้ายกับการสรุปความลิสต์ แต่มีวงเล็บแทนการใช้วงเล็บเหลี่ยม

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

ผลลัพธ์คือได้ออบเจ็กต์ตัวสร้างที่รู้วิธีวนซ้ำตามลำดับของค่า แต่ต่างจากการสรุปความลิสต์คือ จะไม่คำนวณค่าทั้งหมดพร้อมกัน มันรอที่จะให้ถาม ฟังก์ชันสำเร็จรูป `next` ใช้รับค่าถัดไปจากตัวสร้าง

```
>>> next(g)
0
>>> next(g)
1
```

เมื่อคุณไปถึงจุดสิ้นสุดของลำดับ `next` จะทำให้เกิดเอ็กเซ็ปชัน StopIteration คุณยังสามารถใช้ลูป `for` เพื่อย้ำผ่านค่า

```
>>> for val in g:
...     print(val)
4
9
16
```

อ็อบเจกต์ตัวสร้างจะติดตามตำแหน่งที่มันอยู่ในลำดับ ดังนั้นลูป `for` จะเลือกตำแหน่งที่ `next` ค้างไว้ต่อไป เมื่อตัวสร้างหมด มันจะเกิด `StopException` ต่อไป

```
>>> next(g)
StopIteration
```

นิพจน์ตัวสร้างมักใช้กับฟังก์ชันเช่น `sum` `max` และ `min`

```
>>> sum(x**2 for x in range(5))
30
```

19.4. ฟังก์ชัน any และฟังก์ชัน all

ไพธอนมีฟังก์ชันสำเร็จรูป any ที่รับข้อมูลแบบลำดับของค่าบูลีนและส่งคืนค่า True ออกมา หากค่าใดค่าหนึ่งเป็น True มันทำงานได้กับลิสต์

```
>>> any([False, False, True])
True
```

แต่มักใช้กับนิพจน์ตัวสร้าง

```
>>> any(letter == 't' for letter in 'monty')
True
```

ตัวอย่างนั้นมีประโยชน์ไม่มากนักเพราะมันทำสิ่งเดียวกับตัวดำเนินการ in แต่เราสามารถใช้ฟังก์ชัน any เพื่อเขียนใหม่บางฟังก์ชันการค้นหาที่เราเขียนไว้ในหัวข้อ 9.3 ตัวอย่างเช่น เราสามารถเขียน avoids ได้ดังนี้

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

ฟังก์ชันนี้เกือบจะอ่านเหมือนภาษาอังกฤษ “word avoids forbidden if there are not any forbidden letters in word.” “คำที่หลีกเลี่ยงสิ่งต้องห้ามหากไม่มีตัวอักษรต้องห้ามในคำ”

การใช้ any กับนิพจน์ตัวสร้างจะมีประสิทธิภาพ เนื่องจากจะหยุดทันทีหากพบค่า True ดังนั้นจึงไม่ต้องประเมินข้อมูลแบบลำดับทั้งหมด

ไพธอนมีฟังก์ชันสำเร็จรูป all ที่คืนค่า True หากทุกอิลิเมนต์ของข้อมูลแบบลำดับเป็น True เพื่อเป็นการฝึกหัด ให้ใช้ all เพื่อเขียนใหม่ uses_all จากส่วนที่ 9.3

19.5. เซต

ในหัวข้อ 13.6 เราใช้ดิกชันนารีเพื่อค้นหาคำต่าง ๆ ที่ปรากฏในเอกสารแต่ไม่อยู่ในลิสต์ของคำศัพท์ ฟังก์ชันนั้นรับดิกต์ d1 ซึ่งมีคำต่าง ๆ จากเอกสารเป็นกุญแจ และดิกต์ d2 ซึ่งมีลิสต์คำ ฟังก์ชันส่งคืนดิกชันนารีที่มีกุญแจจาก d1 ที่ไม่ได้อยู่ใน d2 ออกมา

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
```

```
        if key not in d2:
            res[key] = None
    return res
```

ในดิกชันนารีทั้งหมดเหล่านี้ ค่าต่าง ๆ คือ None เนื่องจากเราไม่เคยใช้ค่าเหล่านี้ ส่งผลให้เราเสียพื้นที่จัดเก็บไปโดยเปล่า

ไพธอนยังมีชนิดข้อมูลสำเร็จรูปที่เรียกว่า เซต set ซึ่งทำหน้าที่เหมือนชุดสะสมของกุญแจดิกชันนารีที่ไม่มีค่า การเพิ่มอิลิเมนต์ในเซตทำได้รวดเร็ว การตรวจสอบสมาชิกภาพก็เช่นกัน และเซตได้จัดเตรียมเมธอดและตัวดำเนินการในการคำนวณสำหรับเซตทั่วไป

ตัวอย่างเช่น การลบเซตสามารถใช้ได้เป็นเมธอดที่เรียกว่า difference หรือเป็นตัวดำเนินการ - เราก็เขียน subtract ใหม่ได้ดังนี้

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

ผลลัพธ์ที่ได้เป็นเซตแทนที่จะเป็นดิกชันนารี แต่สำหรับการดำเนินการ เช่น การวนซ้ำ ลักษณะการทำงานจะเหมือนกัน

แบบฝึกหัดบางส่วนในหนังสือเล่มนี้สามารถทำได้อย่างกระชับและมีประสิทธิภาพด้วยเซต ตัวอย่างเช่น วิธีแก้ปัญหาสำหรับ has_duplicates จากแบบฝึกหัด 10.7 ที่ใช้ดิกชันนารี:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

เมื่ออิลิเมนต์ปรากฏขึ้นเป็นครั้งแรก อิลิเมนต์นั้นจะถูกเพิ่มลงในดิกชันนารี หากอิลิเมนต์เดิมปรากฏขึ้นอีกครั้ง ฟังก์ชันจะคืนค่า True

เมื่อใช้เซตเราสามารถเขียนฟังก์ชันเดียวกันได้ดังนี้

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

อิลิเมนต์สามารถปรากฏในเซตได้เพียงครั้งเดียว ดังนั้นหากอิลิเมนต์ใน `t` ปรากฏมากกว่าหนึ่งครั้ง เซตจะเล็กกว่า `t` ถ้าไม่มีซ้ำ เซตจะมีขนาดเท่ากับ `t`

เรายังสามารถใช้เซตเพื่อทำแบบฝึกหัดในบทที่ 9 ได้อีกด้วย ตัวอย่างเช่น เวอร์ชันของ `uses_only` ที่มีลูป

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` ตรวจสอบว่ามีตัวอักษรทั้งหมดใน `word` อยู่ใน `available` หรือไม่ เราสามารถเขียนใหม่ได้ดังนี้

```
def uses_only(word, available):
    return set(word) <= set(available)
```

ตัวดำเนินการ **<=** จะตรวจสอบว่าเซตใดเซตหนึ่งเป็นเซตย่อยหรือไม่ ซึ่งรวมถึงความเป็นไปได้ที่เซตดังกล่าวจะเท่ากัน ซึ่งจะเป็นจริงหากตัวอักษรทั้งหมดใน `word` ปรากฏใน `available`

เพื่อเป็นการฝึกหัด ให้เขียน `avoids` ใหม่โดยใช้เซต

19.6. ตัวนับ

เคาน์เตอร์เป็นเหมือนเซต ยกเว้นว่าหากอิลิเมนต์ปรากฏขึ้นมากกว่าหนึ่งครั้ง เคาน์เตอร์จะติดตามจำนวนครั้งที่ปรากฏขึ้น หากคุณคุ้นเคยกับแนวคิดทางคณิตศาสตร์ของ **มัลติเซ็ต (multiset)** เคาน์เตอร์จะเป็นวิธีที่เป็นธรรมชาติในการแสดงมัลติเซ็ต

เคาน์เตอร์ถูกกำหนดในโมดูลมาตรฐานที่เรียกว่า `collections` ดังนั้นคุณต้องนำเข้าก่อน คุณสามารถเริ่มต้นเคาน์เตอร์ด้วยสายอักขระ ลิสต์ หรืออะไรก็ได้ที่สนับสนุนการวนซ้ำ

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

เคาน์เตอร์ทำตัวเหมือนดิกชันนารีในหลาย ๆ ด้าน; โดยจะจับคู่จากแต่ละกุญแจกับจำนวนครั้งที่ปรากฏ เช่นเดียวกับในดิกชันนารี กุญแจต่าง ๆ จะต้องสามารถแฮชได้

ต่างจากดิกชันนารีตรงที่ เคาน์เตอร์จะไม่สร้างเอ็กเซ็ปชั่นหากคุณเข้าถึงอิลิเมนต์ที่ไม่ปรากฏ แต่จะคืนค่า 0

```
>>> count['d']
0
```

เราสามารถใช้เคาน์เตอร์ เพื่อเขียน `is_anagram` จากแบบฝึกหัด 10.6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

หากคำสองคำเป็นอนาแกรม คำเหล่านั้นมีตัวอักษรเดียวกันโดยมีจำนวนเท่ากัน ดังนั้นเคาน์เตอร์จึงเท่ากัน

เคาน์เตอร์มีเมธอดและตัวดำเนินการเพื่อดำเนินการเหมือนเซต รวมถึงการบวก การลบ การยูเนี่ยน และการตัดกัน และเคาน์เตอร์ยังมีเมธอดที่มักจะเป็นประโยชน์ `most_common` ซึ่งส่งคืนลิสต์ของคู่ของค่ากับความถี่ เรียงลำดับจากความถี่มากที่สุดไปน้อยที่สุด

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7. คลาส `defaultdict`

โมดูล `collections` ยังมี `defaultdict` ซึ่งเหมือนกับดิกชันนารี ยกเว้นว่าหากคุณเข้าถึงกุญแจที่ไม่มีอยู่ ก็จะสามารถสร้างค่าใหม่ได้ทันที

เมื่อคุณสร้าง `defaultdict` คุณจัดเตรียมฟังก์ชันที่ใช้เพื่อสร้างค่าใหม่ ฟังก์ชันที่ใช้สร้างอ็อบเจกต์บางครั้งเรียกว่า **แฟคทอรี่ (factory)** ฟังก์ชันสำเร็จรูปที่สร้างลิสต์ เซต และชนิดข้อมูลอื่น ๆ สามารถใช้เป็นแฟคทอรี่ได้

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

โปรดสังเกตว่าอาร์กิวเมนต์คือ `list` ซึ่งเป็นคลาสอ็อบเจกต์ ไม่ใช่ `list()` ซึ่งเป็นลิสต์ใหม่ ฟังก์ชันที่คุณระบุจะไม่ถูกเรียก เว้นแต่คุณจะเข้าถึงกุญแจที่ไม่มีอยู่จริง

```
>>> t = d['new key']
>>> t
[]
```

ลิสต์ใหม่ที่เราเรียกว่า `t` ถูกเพิ่มลงในดิกชันนารีด้วย ดังนั้นหากเราแก้ไข `t` การเปลี่ยนแปลงจะปรากฏใน `d`

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

หากคุณกำลังสร้างดิกชันนารีของลิสต์ คุณมักจะสามารถเขียนโค้ดที่ง่ายกว่าโดยใช้ `defaultdict` ในเฉลยของแบบฝึกหัด 12.2 ซึ่งสามารถดูได้จาก `http://thinkpython2.com/code/anagram_sets.py` ผมสร้างดิกชันนารีที่จับคู่จากสตริงตัวอักษรที่เรียงลำดับไปยังลิสต์คำที่สามารถสะกดด้วยตัวอักษรเหล่านั้นได้ ตัวอย่างเช่น `'opst'` จะจับคู่กับลิสต์ `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`

นี่เป็นโค้ดดั้งเดิม

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

โค้ดนี้สามารถเขียนให้ง่ายขึ้นได้โดยใช้ `setdefault` ซึ่งคุณก็อาจใช้ในแบบฝึกหัดที่ 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
```

```
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

การแก้ปัญหานี้มีข้อเสียคือสร้างลิสต์ใหม่ทุกครั้งไม่ว่าจะจำเป็นหรือไม่ สำหรับลิสต์นั้นไม่ใช่เรื่องใหญ่ แต่ถ้าฟังก์ชั่นของแฟคทอรี่ซับซ้อนก็อาจจะเป็นเช่นนั้น

เราสามารถหลีกเลี่ยงปัญหานี้และทำให้โค้ดง่ายขึ้นโดยใช้ `defaultdict`

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

วิธีแก้ปัญหาของผมสำหรับแบบฝึกหัด 18.3 ซึ่งคุณสามารถดาวน์โหลดได้จาก `http://thinkpython2.com/code/PokerHandSoln.py` ใช้ `setdefault` ในฟังก์ชัน `has_straightflush` โซลูชันนี้มีข้อเสียเปรียบในการสร้างอ็อบเจกต์ `Hand` ทุกครั้งที่วนซ้ำไม่ว่าจะจำเป็นหรือไม่ก็ตาม เพื่อเป็นการฝึกหัด ให้เขียนใหม่โดยใช้ `defaultdict`

19.8. เนมทูเพิล (Named tuples)

อ็อบเจกต์อย่างง่ายจำนวนมากนั้นเป็นชุดของค่าที่เกี่ยวข้องกัน ตัวอย่างเช่น อ็อบเจกต์ Point ที่กำหนดไว้ในบทที่ 15 ประกอบด้วยตัวเลขสองตัวคือ **x** และ **y** เมื่อคุณกำหนดคลาสเช่นนี้ คุณมักจะเริ่มต้นด้วยเมธอด init และเมธอด str

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

```
    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

นี่ดูเหมือนเป็นโค้ดจำนวนมากในการถ่ายทอดข้อมูลจำนวนเล็กน้อย ไพธอนให้วิธีที่กระชับยิ่งขึ้นในการพูดสิ่งเดียวกัน:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

อาร์กิวเมนต์แรกคือชื่อของคลาสที่คุณต้องการสร้าง อาร์กิวเมนต์ที่สองคือรายการของแอตทริบิวต์ที่อ็อบเจกต์ Point ควรมีในรูปสายอักขระ ค่าคืนกลับจากเนมทูเพิล `namedtuple` เป็นคลาสอ็อบเจกต์

```
>>> Point
<class '__main__.Point'>
```

`Point` จะจัดเตรียมเมธอดเช่น `__init__` และ `__str__` โดยอัตโนมัติ คุณจึงไม่ต้องเขียน

ในการสร้างอ็อบเจกต์ Point คุณใช้คลาส Point เป็นฟังก์ชัน

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

เมธอด init กำหนดอาร์กิวเมนต์ให้กับแอตทริบิวต์โดยใช้ชื่อที่คุณระบุ เมธอด str พิมพ์คำบรรยายของอ็อบเจกต์ Point และแอตทริบิวต์

คุณสามารถเข้าถึงอิลิเมนต์ของเนมทูเพิลตามชื่อ

```
>>> p.x, p.y
(1, 2)
```

แต่คุณยังสามารถถือว่าเนมทูเพิลเป็นทูเพิลได้

```
>>> p[0], p[1]
(1, 2)
```

```
>>> x, y = p
>>> x, y
(1, 2)
```

เนมทูเพิลเป็นวิธีที่รวดเร็วในการกำหนดคลาสอย่างง่าย ข้อเสียคือคลาสธรรมดาไม่ได้เรียบง่ายเสมอไป คุณอาจตัดสินใจในภายหลังว่าคุณต้องการเพิ่มเมธอดให้กับ เนมทูเพิล ในกรณีนั้นคุณสามารถกำหนดคลาสใหม่ที่สืบทอดมาจากเนมทูเพิล

```
class Pointier(Point):
    # add more methods here
```

หรือคุณอาจเปลี่ยนไปใช้การประกาศคลาสแบบธรรมดาก็ได้

19.9. การรวบรวมพารามิเตอร์ด้วยคำสำคัญ args

ในหัวข้อ 12.4 เราได้เห็นวิธีเขียนฟังก์ชันที่รวบรวมอาร์กิวเมนต์เป็นทูเพิล

```
def printall(*args):
    print(args)
```

คุณสามารถเรียกใช้ฟังก์ชันนี้ด้วยอาร์กิวเมนต์ตำแหน่งจำนวนเท่าใดก็ได้ (นั่นคือ อาร์กิวเมนต์ที่ไม่มีคำสำคัญ)

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

แต่ตัวดำเนินการ * ไม่ได้รวบรวมอาร์กิวเมนต์คำสำคัญ

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

ในการรวบรวมอาร์กิวเมนต์คำสำคัญ คุณสามารถใช้ตัวดำเนินการ **

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

คุณสามารถรวบรวมพารามิเตอร์คำสำคัญอะไรก็ได้ที่คุณต้องการ แต่ kwargs เป็นตัวเลือกทั่วไป ผลลัพธ์คือดิกชันนารีที่จับคู่คำสำคัญกับค่า

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

หากคุณมีดิกชันนารีของคำสำคัญและค่า คุณสามารถใช้ตัวดำเนินการกระจาย ** เพื่อเรียกใช้ฟังก์ชันดังเช่น

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

หากไม่มีตัวดำเนินการกระจาย ฟังก์ชันจะถือว่า `d` เป็นอาร์กิวเมนต์ตำแหน่งเดียว ดังนั้นจะกำหนด `d` ให้กับ `x` และแจ้งข้อผิดพลาดเพราะไม่มีอะไรจะกำหนดค่าให้กับ `y`

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

เมื่อคุณทำงานกับฟังก์ชันที่มีพารามิเตอร์จำนวนมาก การสร้างและส่งผ่านดิกชันนารีที่ระบุตัวเลือกที่ใช้บ่อยมักจะเป็นประโยชน์

19.10. อภิธานศัพท์

นิพจน์เงื่อนไข (conditional expression): นิพจน์ที่มีค่าหนึ่งในสองค่า ขึ้นอยู่กับเงื่อนไข

การสรุปความลิสต์ (list comprehension): นิพจน์ที่มี `for` วนซ้ำในวงเล็บเหลี่ยมที่ให้ลิสต์ใหม่

นิพจน์ตัวสร้าง (generator expression): นิพจน์ที่มี `for` วนซ้ำในวงเล็บที่ให้ผลลัพธ์เป็นอ็อบเจกต์ตัวสร้าง

between the elements of a set and the number of times they appear.

มัลติเซต (multiset): เอกลักษณ์ทางคณิตศาสตร์ที่แสดงถึงการจับคู่ระหว่างอีลีเมนต์ของเซตและจำนวนครั้งที่ปรากฏขึ้น

แฟคทอรี่ (factory): ฟังก์ชันซึ่งมักจะถูกส่งผ่านเป็นพารามิเตอร์ เพื่อใช้สร้างอ็อบเจกต์

19.11. แบบฝึกหัด

***แบบฝึกหัด* 19.1.** *ต่อไปนี้เป็นฟังก์ชันคำนวณสัมประสิทธิ์ทวินามแบบเรียกซ้ำ*

```
def binomial_coeff(n, k):
    """
    Compute the binomial coefficient "n choose k".
    n: number of trials
    k: number of successes
    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

เขียนเนื้อความของฟังก์ชันใหม่โดยใช้นิพจน์เงื่อนไขที่ซ้อนกัน

หมายเหตุหนึ่ง: ฟังก์ชันนี้ไม่มีประสิทธิภาพมากนักเพราะจะลงเอยด้วยการคำนวณค่าเดียวกันซ้ำแล้วซ้ำอีก เราสามารถทำให้มีประสิทธิภาพมากขึ้นโดยการจำค่าที่คำนวณไว้แล้ว (ดูหัวข้อ 11.6) แต่เราก็จะพบว่ามันยากกว่าที่จะจำค่าถ้าเราเขียนโค้ดโดยใช้นิพจน์เงื่อนไข

A. การดีบัก

เวลาที่เราดีบัก เราควรจะแยกชนิดของข้อผิดพลาดให้ออก เพื่อจะได้หาสาเหตุมันได้เร็วขึ้น:

- **ข้อผิดพลาดเชิงวากยสัมพันธ์** (**syntax errors**) จะเห็นชัด จากที่ไพธอนอินเตอร์พรีเตอร์แจ้งข้อผิดพลาดออกมา ตอนที่มันแปลโค้ดภาษาไพธอนไปเป็นไบต์โค้ด ข้อผิดพลาดเชิงวากยสัมพันธ์จะบอกว่าโปรแกรมมีโครงสร้างของภาษาที่เขียนไม่ถูก ตัวอย่างเช่น การลืมเครื่องหมาย `:` ที่คำสั่ง `def` ก็จะทำให้ไพธอนอินเตอร์พรีเตอร์แจ้งข้อความ `SyntaxError: invalid syntax` ออกมา
- **ข้อผิดพลาดเวลาดำเนินการ** (**runtime errors**) จะถูกไพธอนอินเตอร์พรีเตอร์แจ้งออกมาตอนที่รันโปรแกรมแล้วมีปัญหาเกิดขึ้น ข้อความแจ้งข้อผิดพลาดเวลาดำเนินการส่วนใหญ่จะบอกรายละเอียดเกี่ยวกับว่า ข้อผิดพลาดเกิดขึ้นที่ไหน ฟังก์ชันอะไรที่กำลังรันอยู่ตอนที่เกิดข้อผิดพลาด ตัวอย่างเช่น การย้อนเรียกใช้ (recursion) ที่เรียกต่อ ๆ ไปไม่มีที่สิ้นสุด สุดท้ายแล้ว ก็จะทำให้เกิดข้อผิดพลาดเวลาดำเนินการ `RecursionError: maximum recursion depth exceeded` ออกมา
- **ข้อผิดพลาดเชิงความหมาย** (**semantic errors**) เป็นปัญหาที่โปรแกรมรันได้ โดยไม่มีข้อความแจ้งข้อผิดพลาดออกมาเลย แต่โปรแกรมทำงานไม่ถูกต้อง ตัวอย่างเช่น นิพจน์อาจจะไม่ได้ถูกประเมินตามลำดับที่เราคาด และให้ผลลัพธ์ที่ผิดออกมา.

ขั้นตอนแรกในการดีบักคือหาว่าข้อผิดพลาดที่เราเจอเป็นชนิดไหน หัวข้อย่อยต่อไปนี้ แม้จะเรียบเรียงตามชนิดของข้อผิดพลาด แต่บางเทคนิคอาจจะใช้ได้กับข้อผิดพลาดชนิดอื่นได้ด้วย

A.1. ข้อผิดพลาดเชิงวากยสัมพันธ์

ข้อผิดพลาดเชิงวากยสัมพันธ์ (*syntax errors*) มักจะแก้ได้ง่าย ถ้าเรารู้แล้วว่ามันคืออะไร แต่หลายครั้ง ข้อความแจ้งข้อผิดพลาดก็ไม่ได้บอกอะไรมาก ข้อความแจ้งข้อผิดพลาดที่พบบ่อย ๆ คือ `SyntaxError:`

`invalid syntax` และ `SyntaxError: invalid token` ซึ่งไม่ได้บอกอะไรมาก

อีกมุมหนึ่ง ข้อความแจ้งข้อผิดพลาดบอกเราว่าที่ไหนในโปรแกรมที่มีปัญหา แต่จริง ๆ แล้ว มันบอกเราว่าตำแหน่งไพธอนเจอปัญหา ซึ่งอาจไม่ใช่ตำแหน่งที่มีข้อผิดพลาดอยู่ บางครั้งข้อผิดพลาดก็จะอยู่ก่อนตำแหน่งที่ข้อความแจ้งออกมา บ่อย ๆ เลยที่ข้อผิดพลาดจริง ๆ อยู่บรรทัดก่อนตำแหน่งที่แจ้ง

ถ้าเราเขียนโปรแกรมแบบค่อย ๆ เพิ่ม ค่อย ๆ เติมคำสั่งเข้าไป เราน่าจะเห็นข้อผิดพลาดได้ง่าย ๆ กว่า เพราะว่า มันจะเป็นบรรทัดคำสั่งที่พึ่งเพิ่มเข้าไป

ถ้าเราเอาโค้ดมาจากหนังสือ เราอาจหาข้อผิดพลาด โดยเปรียบเทียบโค้ดที่เขียนกับโค้ดในหนังสือ ตรวจดูทุก ๆ อักขระ และให้นึกในใจว่า หนังสือก็อาจจะผิดได้ ดังนั้น ถ้าเห็นอะไรที่ดูน่าจะผิดวากยสัมพันธ์ มันก็อาจจะผิดจริง ๆ

รายการต่อไปนี้แสดงแนวทางปฏิบัติที่จะหลีกเลี่ยง ข้อผิดพลาดเชิงวากยสัมพันธ์ที่พบบ่อย ๆ

1. อย่าใช้คำสำคัญ (keyword) ของไพธอนในการตั้งชื่อตัวแปร
2. ตรวจดูว่า มีเครื่องหมายจุดคู่ (colon) ที่ท้ายคำสั่งประกอบ เช่น คำสั่ง `for` คำสั่ง `while` คำสั่ง `if` และคำสั่ง `def`
3. ตรวจสอบให้แน่ใจว่า สายอักขระทุก ๆ อันในโปรแกรม ใช้เครื่องหมายคำพูดที่เข้ากัน (ถ้าเปิดด้วย `'` ปิดด้วย `'` ถ้าเปิดด้วย `"` ปิดด้วย `"`) ตรวจสอบว่าทุก ๆ เครื่องหมายคำพูด เป็นเครื่องหมายแบบตรง ได้แก่ `'` หรือ `"` ไม่ใช่แบบโค้ง ซึ่งได้แก่ “ หรือ ” หรือ ‘ หรือ ’
4. ถ้ามีการใช้สายอักขระหลายบรรทัด ที่ใช้เครื่องหมายคำพูดสามครั้ง (ไม่ว่าจะเป็นแบบเดี่ยว `'''` หรือแบบคู่ `"""`) ตรวจสอบให้แน่ใจว่า สายอักขระหลายบรรทัดนั้นถูกปิดถูกต้องดีแล้ว สายอักขระที่ไม่ได้ถูกปิด (หรือปิดไม่ถูก) จะทำให้เกิดข้อผิดพลาด `invalid token` ออกมาที่ท้ายโปรแกรม หรือ ไพธอนอาจจะเหมารวมเอาส่วนของโปรแกรมที่ตามมาเป็นสายอักขระไปด้วย จนกว่ามันจะเจอสายอักขระใหม่ ในกรณีที่สองนี้ ไพธอนจะไม่ให้ข้อความแจ้งข้อผิดพลาดออกมาเลย
5. ตัวดำเนินการที่เปิดแล้ว แต่ยังไม่ปิด ได้แก่ ตัวดำเนินการ `(` หรือ ตัวดำเนินการ `{` หรือ ตัวดำเนินการ `[` จะทำให้ไพธอนคิดว่าบรรทัดถัดไปเป็นส่วนหนึ่งของคำสั่งที่ยังไม่ปิด ส่วนใหญ่แล้วข้อความแจ้งข้อผิดพลาด จะออกมาที่บรรทัดถัดไป
6. ตรวจสอบว่ามีการใช้ `=` แทน `==` ในการตรวจสอบเงื่อนไขหรือไม่ เช่น การตรวจสอบเงื่อนไข ของคำสั่ง `if`

7. ตรวจสอบการย่อหน้า ให้แน่ใจว่ามันถูกต้อง ไพธอนสามารถรับได้ทั้ง ช่องว่าง (space) หรือ การตั้งระยะ (tab) แต่ถ้าเราใช้มันผสมกัน มันจะมีปัญหา วิธีที่ดีที่สุด ที่จะหลีกเลี่ยงปัญหาแบบนี้คือ ใช้โปรแกรมบรรณาธิกรข้อความ (text editor) ที่เหมาะกับภาษาไพธอน และสร้างการย่อหน้าด้วยอักขระแบบเดียวกัน (เช่น เมื่อเราพิมพ์การตั้งระยะ มันจะเปลี่ยนเป็น ช่องว่างสี่ช่องให้แทน)

8. ถ้ามีอักขระที่ไม่ใช่รหัสแอสกี (non-ASCII characters) อยู่ในโค้ด (รวมถึง อยู่ในสายอักขระ หรืออยู่ในส่วนข้อคิดเห็นด้วย) มันจะมีปัญหาได้ ถึงแม้ว่าไพธอน-3 ทั่วไปแล้ว จะสามารถรับอักขระที่ไม่ใช่รหัสแอสกีได้ แต่ให้ระวัง เวลาที่เราเอาข้อความมาจากเว็บไซต์ หรือจากแหล่งอื่น ๆ

ถ้าลองตรวจดูตามนี้แล้ว ก็ยังแก้ปัญหาไม่ได้ ลองดูหัวข้อถัดไป

A.1.1. ลองทำตั้งหลายอย่างแล้ว แต่ไม่เห็นมีอะไรเปลี่ยนเลย

ถ้าอินเตอร์พรีเตอร์บอกว่ามีข้อผิดพลาดอยู่แต่เราไม่เห็นมีอะไรผิดในโค้ด อาจจะเป็นเพราะว่าเรากับอินเตอร์พรีเตอร์กำลังดูโค้ดคนละโค้ดกันก็ได้ ตรวจสอบไอดีอีที่กำลังใช้อยู่ ว่าโค้ดไพธอนที่เรากำลังเขียนอยู่ เป็นอันเดียวกับที่อินเตอร์พรีเตอร์รัน

ถ้ายังไม่แน่ใจ ให้ลองใส่โค้ดที่เป็นข้อผิดพลาดเชิงวากยสัมพันธ์ชัด ๆ เด่น ๆ สักอัน ไปที่ตอนต้น ๆ ของโปรแกรมเลย แล้วลองรันดู ถ้าอินเตอร์พรีเตอร์ไม่เจอข้อผิดพลาดใหม่ที่เราตั้งใจใส่เข้าไป เราอาจจะไม่ได้รันโค้ดที่กำลังเขียนอยู่

มีตัวการเด่น ๆ อยู่บ้าง เช่น

- เราแก้ไฟล์ไปแล้ว แต่ลืมเซฟ ก่อนจะรัน ไอดีอีบางตัว จะช่วยเซฟให้ แต่บางตัวก็ไม่ทำให้
- เราเปลี่ยนชื่อไฟล์ไป แต่ยังไม่รันชื่อไฟล์เก่า
- ไอดีอีมีค่าที่ปรับตั้งไว้ไม่ถูกอยู่
- ถ้ากำลังเขียนโมดูล และใช้ `import` อยู่ ให้ตรวจสอบว่า ไม่ได้ตั้งชื่อโมดูล ไปเหมือนกับโมดูลมาตราฐานของไพธอน
- ถ้าใช้ `import` เพื่ออ่านโมดูล ให้รู้ว่า เราต้องเริ่มอินเตอร์พรีเตอร์ขึ้นมาใหม่ หรือไม่ก็ใช้ `reload` เพื่ออ่านโมดูลที่แก้ไขใหม่ ถ้าแค่ใช้คำสั่ง `import` โมดูลที่เคยนำเข้ามาแล้ว ไพธอนจะไม่ทำอะไรเลย

ถ้ายังติดอีก และก็ยังหาไม่เจอ ให้ลองเปิดไฟล์ขึ้นมาเขียนโปรแกรมใหม่เลย เช่น “Hello, World!” และลองให้เห็นว่าโปรแกรมนี้ถูกรันได้ แล้วค่อย ๆ ใส่ส่วนต่าง ๆ ของโปรแกรมเดิมเข้าไปในโปรแกรมใหม่นี้

A.2. ข้อผิดพลาดเวลาดำเนินการ

ถ้าโปรแกรมมีวากยสัมพันธ์ที่ถูกต้อง ไพธอนจะอ่านมันได้ และอย่างน้อยก็สามารถเริ่มรันมันได้ แล้วจะมีอะไรที่จะผิดได้อีก?

A.2.1. ไม่เห็นโปรแกรมมันทำอะไรเลย

ปัญหานี้เจอบ่อยที่สุด เวลาที่โค้ดมีฟังก์ชัน มีคลาสต่าง ๆ อยู่ แต่ไม่ได้เรียกฟังก์ชันขึ้นมาทำงานจริง ๆ บางทีเราอาจจะอยากให้เป็นแบบนี้ เวลาเขียนโมดูลที่มีคลาสและฟังก์ชันต่าง ๆ เพื่อให้ไฟล์อื่นเรียกใช้

แต่ถ้าเราไม่ได้อยากให้เป็นแบบนี้ ตรวจสอบว่ามีการเรียกใช้ฟังก์ชันในโปรแกรม และตรวจสอบดูลำดับของการรันคำสั่ง ว่า การเรียกฟังก์ชันนั้นจะถูกรัน (ดูเรื่อง “ลำดับการรันคำสั่ง” ข้างล่าง)

A.2.2. โปรแกรมแฮง

ถ้าโปรแกรมค้าง และดูเหมือนไม่ได้ทำอะไร มันเรียกว่าโปรแกรม*แฮง* (*hang*) ซึ่งบ่อย ๆ เลย ที่โปรแกรม*แฮง* มาจากโปรแกรมติดอยู่ใน*ลูปไม่สิ้นสุด* (*infinite loop*) หรือติดอยู่ใน*การย้อนเรียกใช้ไม่รู้จบ* (*infinite recursion*)

- ถ้ามีลูปที่สงสัยว่าอาจจะเป็นตัวสร้างปัญหา ให้ใส่คำสั่ง `print` เข้าไปก่อนเข้าลูป โดยให้พิมพ์ออกมาว่า “entering the loop” และอีกอันใส่เข้าไปหลังจากเพิ่งออกจากลูป พิมพ์ว่า “exiting the loop”

  รันโปรแกรม แล้วถ้าเห็นแต่ข้อความแรก ไม่เห็นข้อความที่สอง เราเจอลูปไม่สิ้นสุดแล้ว ลองดูหัวข้อ “ลูปไม่สิ้นสุด” ข้างล่าง

- ส่วนใหญ่ *การย้อนเรียกใช้ไม่รู้จบ* จะทำให้โปรแกรนรันไปพักหนึ่ง ก่อนที่จะให้ ``` ``RuntimeError: Maximum recursion depth exceeded'' ``` ออกมา ถ้าเป็นแบบนี้ ลองดูหัวข้อ “การเรียกซ้ำไม่รู้จบ” ข้างล่าง

  ถ้าไม่เห็นข้อความแจ้งข้อผิดพลาดแบบนี้ แต่สงสัยว่าจะมีปัญหากับ*การวนซ้ำ* หรือว่าฟังก์ชัน ก็ยังสามารถใช้เทคนิคที่อภิปรายในหัวข้อ “การเรียกซ้ำไม่รู้จบ” ได้

- ถ้าไม่มีวิธีไหนที่ได้ผลเลย ลองตรวจสอบลูปอื่นและการวนซ้ำอื่นดู

- ถ้ายังไม่ได้ผลอีก เป็นไปได้ว่า เราอาจจะยังไม่เข้าใจลำดับขั้นตอนการทำงานของโปรแกรม ลองดูหัวข้อ “ลำดับขั้นตอนการทำงาน” ข้างล่าง

ลูปไม่สิ้นสุด

ถ้าคิดว่า กำลังเจอกับ*ลูปไม่สิ้นสุด*อยู่ และสงสัยว่า ลูปไหนที่เป็นตัวปัญหา ให้ลองใส่คำสั่ง `print` ที่ท้ายของลูป โดยพิมพ์ค่าของตัวแปรต่าง ๆ ในเงื่อนไข และค่าของเงื่อนไขออกมา

ตัวอย่างเช่น:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

ตอนนี้ถ้าเรารันโปรแกรม เราจะเห็น เอาต์พุตออกมาสามบรรทัด สำหรับการทำงานแต่ละครั้งในลูป การทำงานครั้งสุดท้ายในลูป เงื่อนไขควรจะเป็น `False` ถ้าลูปทำงานไปเรื่อย ๆ เราน่าจะเห็นค่าของ `x` และ `y` และก็น่าจะพอสืบต่อได้ว่า ทำไมค่าตัวแปรต่าง ๆ ถึงไม่ได้ถูกเปลี่ยนค่าอย่างถูกต้อง

การเรียกซ้ำไม่รู้จบ

ส่วนใหญ่ การเรียกซ้ำไม่รู้จบ จะทำให้โปรแกรมรันไปได้พักหนึ่ง ก่อนจะให้ข้อผิดพลาด `Maximum recursion depth exceeded` ออกมา

ถ้าสงสัยว่า ฟังก์ชันเป็นตัวการทำให้เกิดการเรียกซ้ำไม่รู้จบ ลองดูให้แน่ใจว่า มี*กรณีฐาน* (*base case*) มันจะต้องมีเงื่อนไข ที่ให้ฟังก์ชันส่งคืนค่ากลับออกมาได้ (ได้รันคำสั่ง `return`) โดยไม่มีการย้อนเรียกใช้อีก. ถ้าไม่มี ลองทนทวนอัลกอริธึมใหม่ และกำหนด*กรณีฐาน*นี้

ถ้ามี*กรณีฐาน* แต่โปรแกรมดูเหมือนไปไม่ถึงมัน ลองใส่คำสั่ง `print` เข้าไปที่ต้นของฟังก์ชัน และพิมพ์ค่าพารามิเตอร์ต่าง ๆ ออกมา ตอนนี้ ถ้ารันโปรแกรม เราจะเห็นบรรทัดที่พิมพ์ค่าต่าง ๆ ออกมาทุกครั้งที่ฟังก์ชันถูกเรียกใช้งาน และเห็นค่าพารามิเตอร์ต่าง ๆ เหล่านั้น ถ้าค่าของพารามิเตอร์ไม่ได้เปลี่ยนแปลงไปหา*กรณีฐาน* เราก็น่าจะพอรู้อะไรบ้างแล้วว่าทำไม

ลำดับขั้นตอนการทำงาน

ถ้าไม่แน่ใจว่าลำดับขั้นตอนการทำงานของโปรแกรมเป็นอย่างไร ลองใส่คำสั่ง `print` เข้าไปตอนเริ่มของแต่ละฟังก์ชัน โดยพิมพ์ข้อความ เช่น "`entering function foo`" เมื่อ `foo` เป็นชื่อของฟังก์ชัน

ตอนนี้ถ้ารันโปรแกรม มันจะพิมพ์ข้อความต่าง ๆ ซึ่งเหมือนร่องรอยบอกการเรียกใช้ของแต่ละฟังก์ชัน

A.2.3. เวลารันโปรแกรมแล้วได้เอ็กเซ็ปชั่นมา

ถ้ามีอะไรผิดพลาดระหว่างการรันโปรแกรม ไพธอนจะพิมพ์ข้อความ รวมถึงชื่อของเอ็กเซ็ปชั่น และบรรทัดที่โปรแกรมเจอปัญหา และ*การสืบย้อน* (*traceback*) ออกมา.

การสืบย้อน ระบุฟังก์ชันที่กำลังรันอยู่ ฟังก์ชันที่เรียกใช้มัน และฟังก์ชันที่เรียกใช้ฟังก์ชันที่เรียกใช้มัน ต่อขึ้นไปเป็นทอด ๆ พูดอีกอย่างก็คือ มันสืบย้อนลำดับของการเรียกใช้ฟังก์ชัน จนไปถึงที่ที่เราอยู่ รวมถึงบรรทัดในโปรแกรมที่มีการเรียกแต่ละครั้ง

ขั้นตอนแรก เป็นการตรวจสอบตำแหน่งที่โปรแกรมเจอข้อผิดพลาดอยู่ และลองหาดูว่าอะไรเป็นสาเหตุ รายการต่อไปนี้แสดง*ข้อผิดพลาดเวลาดำเนินการ* ที่พบได้บ่อยที่สุด:

NameError: เกิดจากการที่พยายามจะใช้ตัวแปรที่ยังไม่ได้กำหนด ลองดูว่า สะกดชื่อถูกหรือเปล่า หรืออย่างน้อยก็สะกดแบบเดียวกันตลอด จำไว้ว่า ตัวแปรเฉพาะที่ ใช้งานได้เฉพาะที่ เราไม่สามารถใช้งานมันได้นอกฟังก์ชันที่กำหนดค่าของมัน

TypeError: อาจเกิดได้จากหลายสาเหตุ:

- อาจเกิดจากการพยายามไปใช้ค่าแบบไม่ถูกต้อง ตัวอย่างเช่น การอ้างดัชนีของ*สายอักขระ* ลิสต์ หรือทูเพิล ด้วยค่าอื่นที่ไม่ใช่เลขจำนวนเต็ม
- อาจเกิดจาก*สายอักขระจัดรูปแบบ*กับตัวแปรที่ผ่านเข้าไปจัดรูปแบบ ไม่เข้ากัน หรืออาจเป็นกรณีจำนวนไม่เท่ากัน หรืออาจเป็นชนิดของค่าไม่ตรงกัน ตัวอย่างเช่น `print("%d, %d"%(5, 6, 7))` และ `print("%d"%'5')`.
- อาจเกิดจากจำนวนอาร์กิวเมนต์ที่ส่งให้ฟังก์ชัน ไม่ตรงตามจำนวนอาร์กิวเมนต์ที่ฟังก์ชันกำหนด. สำหรับเมธอด ตรวจสอบคำนิยามของเมธอด และตรวจสอบว่าพารามิเตอร์แรกเป็น `self` แล้วตรวจดูการเรียกใช้เมธอด ว่าเรียกใช้เมธอดจากอ็อบเจกต์ถูกชนิด และส่งอาร์กิวเมนต์อื่น ๆ ไปให้ถูก

KeyError: อาจเกิดจากการพยายามอ้างถึงรายการภายในของดิกชันนารี โดยใช้กุญแจที่ดิกชันนารีไม่มี ถ้ากุญแจเป็นสายอักขระ ตรวจสอบดูว่าการสะกด รวมถึงการใช้ตัวพิมพ์เล็กพิมพ์ใหญ่

AttributeError: อาจเกิดจากการพยายามอ้างถึง*ลักษณะประจำ* หรือ*เมธอด* ที่ไม่มีอยู่ ลองตรวจสอบการสะกด เราสามารถใช้ฟังก์ชันสำเร็จรูป `dir` หรือ `vars` เพื่อตรวจดู*ลักษณะประจำ*ต่าง ๆ ที่มีอยู่ได้

ถ้า `AttributeError` บอกว่า อ็อบเจกต์มี`NoneType` นั่นหมายถึงว่า ตัวอ็อบเจกต์มีค่าเป็น `None` ดังนั้นปัญหาไม่ใช่ชื่อของ*ลักษณะประจำ* แต่เป็นตัวอ็อบเจกต์เอง

สาเหตุที่อ็อบเจกต์เป็น `None` อาจจะมาจาก เราลืม `return` ค่าออกมาจากฟังก์ชัน อย่างเช่น ถ้ามีวิธีที่จะรันฟังก์ชันจนจบได้ โดยไม่ต้องทำคำสั่ง `return` ฟังก์ชันจะให้ค่า `None` ออกมาโดยอัตโนมัติ อีกสาเหตุที่พบบ่อย ๆ ก็คือ การใช้ผลลัพธ์จากเมธอดของลิสต์ เช่น `sort` ที่ให้ค่าออกมาเป็น `None`

IndexError: อาจเกิดจากดัชนีที่ใช้ในการอ้างถึงรายการในลิสต์ สายอักขระ หรือทูเพิล มีค่าใหญ่กว่าความยาวลบหนึ่ง ลองใส่คำสั่ง `print` เข้าไปก่อนตำแหน่งที่แจ้งข้อผิดพลาดออกมา โดยให้พิมพ์ค่าของดัชนี และความยาวของลิสต์ สายอักขระ หรือทูเพิลออกมา ตรวจสอบดูความยาว ว่าถูกต้องตามที่คาดหมายหรือไม่ และค่าดัชนีเป็นตามที่คาดหมายหรือไม่

ไพธอนดีบักเกอร์ (`pdb`) มีประโยชน์มากในการช่วยสืบหาสาเหตุของเอ็กเซ็ปชั่น เพราะว่า มันจะช่วยให้เราสามารถดูสถานะของโปรแกรมได้ทันทีก่อนข้อผิดพลาดจะเกิดขึ้น ศึกษาเรื่อง `pdb` เพิ่มเติมได้จาก `https://docs.python.org/3/library/pdb.html`

A.2.4. เราใส่คำสั่ง `print` เข้าไปเยอะ จนเราเริ่มมึนและท่วมท้นจากเอาต์พุตที่เห็น

ปัญหาหนึ่งจากการใช้คำสั่ง `print` เพื่อดีบัก คือสุดท้ายเราอาจจะท่วมท้นกับเอาต์พุตจำนวนมากที่ได้ มีสองวิธี คือ สะสางเอาต์พุตให้ดูง่ายขึ้น หรือสะสางโปรแกรมให้เรียบง่ายขึ้น

เพื่อสะสางเอาต์พุต เราอาจจะเอาคำสั่ง `print` ที่ไม่ได้ช่วยเท่าไรออก ซึ่งอาจจะลบออกเลย หรืออาจจะคอมเมนต์มันออก หรือเราอาจจะปรับให้คำสั่ง `print` พิมพ์เอาต์พุตที่ดูง่ายขึ้นออกมาแทน

เพื่อสะสางโปรแกรมให้เรียบง่าย มีหลาย ๆ อย่างที่ทำได้ ได้แก่ หนึ่งสะสางโดยจำกัดโจทย์ที่โปรแกรมทำลง ตัวอย่างเช่น ถ้าโปรแกรมทำการค้นหาในลิสต์ ลองค้นหาในลิสต์ขนาดเล็กดู ถ้าโปรแกรมรับอินพุตจากผู้ใช้ ลองใส่อินพุตที่ง่ายที่สุดที่สร้างปัญหาดู

สองสะสางโปรแกรมจริง ๆ จัง ๆ ไปเลย ลบส่วนของโปรแกรมที่ไม่ได้ทำงานอะไรออก และจัดเรียบเรียงโปรแกรมให้อ่านได้ง่ายที่สุดเท่าที่ทำได้ ตัวอย่างเช่น ถ้าสงสัยว่า ปัญหาอยู่ในส่วนที่ซ้อนอยู่ลึก ๆ ของโปรแกรม ลองเขียนส่วนนั้นของโปรแกรมใหม่ ด้วยโครงสร้างที่เรียบง่ายขึ้น ถ้าสงสัยการทำงานของฟังก์ชันใหญ่ ลองแตกมันออกเป็นฟังก์ชันเล็ก ๆ และทดสอบแต่ละฟังก์ชันแยกกันดู

บ่อย ๆ เลยที่ระหว่างที่กำลังคิดหากรณีทดสอบส่วนย่อย ๆ จะช่วยให้เราเจอบัก ถ้าพบว่าโปรแกรมทำงานได้ในสถานการณ์หนึ่ง แต่ทำงานไม่ได้ในอีกสถานการณ์ อันนี้จะใช้เป็นเงื่อนงำในการหาสาเหตุของปัญหา

ทำนองเดียวกัน การเขียนส่วนของโปรแกรมใหม่ จะช่วยให้เราหาบักที่ซ่อนอยู่ได้ ถ้าเราเปลี่ยนส่วนของโปรแกรมที่คิดว่าไม่น่ามีผล แต่พบว่ามันมีผล นี่เป็นสัญญาณที่สำคัญของสาเหตุของปัญหา

A.3. ข้อผิดพลาดเชิงความหมาย

จากหลาย ๆ แง่มุม *ข้อผิดพลาดเชิงความหมาย* (*semantic errors*) เป็นเรื่องที่ดีบักยากที่สุด เพราะว่าไพธอนอินเตอร์พรีเตอร์จะไม่บอกว่ามีอะไรผิดเลย มีแต่เราเท่านั้นที่รู้ว่าโปรแกรมควรจะทำงานอย่างไร

ขั้นตอนแรก คือหาความเชื่อมโยงระหว่างข้อความที่เห็นจากโปรแกรม กับพฤติกรรมของโปรแกรม เราต้องประเมินว่าจริง ๆ แล้วว่าโปรแกรมทำอะไรลงไป สิ่งหนึ่งที่ทำให้มันยากก็คือคอมพิวเตอร์รันเร็วมาก

บางที เราก็อยากจะให้โปรแกรมรันช้าลงพอที่จะเห็นการทำงานได้ และดีบักเกอร์ก็ช่วยทำให้โปรแกรมรันช้าลงได้ เพียงแต่ การใช้คำสั่ง `print` มันจะสะดวกกว่าการใช้ดีบักเกอร์ ที่รวมการใส่*จุดหยุด* (*breakpoint*) และ*การรันทีละขั้น* (*single stepping*) ไปจนพบตำแหน่งของปัญหา

A.3.1. โปรแกรมไม่ทำงาน

เราควรจะลองถามตัวเองดูว่า:

- มีอะไรไหมที่โปรแกรมควรจะทำ แต่ดูเหมือนมันไม่ได้ทำ? ลองหาส่วนของโปรแกรม ที่ทำหน้าที่นั้น และตรวจสอบว่ามันถูกรันตอนที่เราคิดว่ามันควรจะถูกรัน
- มีอะไรที่เกิดขึ้น ทั้ง ๆ ที่มันไม่ควรจะเกิดหรือไม่? ลองหาส่วนของโปรแกรมที่ทำหน้าที่นั้น และตรวจดูว่ามันถูกรันตอนที่ไม่ควรจะถูกรันหรือไม่
- มีส่วนของโปรแกรมที่ให้ผลลัพธ์ออกมา ต่างจากที่คิดหรือไม่? ตรวจสอบว่า เราเข้าใจโปรแกรมที่มีปัญหาดีแล้ว โดยเฉพาะถ้ามันเรียกใช้ฟังก์ชัน หรือเมธอดจากไพธอนมอดูลอื่น ๆ อ่าน

เอกสารที่เกี่ยวข้องกับฟังก์ชันที่เรียกใช้ ลองใช้ฟังก์ชันต่าง ๆ ดูก่อนกับโปรแกรมง่าย ๆ ในสถานการณ์ง่าย ๆ

เพื่อจะเขียนโปรแกรม เราต้องมีแนวคิดในหัวก่อน ว่าโปรแกรมจะทำงานอย่างไร ถ้าเราเขียนโปรแกรม แล้วโปรแกรมไม่ทำงานตามที่เราคิด บ่อย ๆ ครั้งเลยที่ปัญหาไม่ได้อยู่ที่โปรแกรม แต่อยู่ในหัวเราเอง

วิธีดีที่สุดในการปรับแนวคิดในหัว คือ แตกโปรแกรมออกเป็นส่วนย่อย ๆ (โดยทั่วไป คือ แตกเป็นฟังก์ชันและเมธอดต่าง ๆ) แล้วทดสอบการทำงานของแต่ละส่วนอย่างอิสระ ถ้าเราเจอว่าอะไรที่แนวคิดในหัวกับความเป็นจริงไม่ตรงกัน เราก็จะเจอคำตอบของปัญหา

แน่นอนว่า เราควรจะสร้างและทดสอบส่วนประกอบย่อยต่าง ๆ ไปเรื่อย ๆ ระหว่างเขียนโปรแกรม ถ้าตรวจสอบเรื่อย ๆ เมื่อเจอปัญหา เราก็จะพอรู้ว่า ปัญหาก็น่าจะอยู่ในส่วนของโปรแกรมที่เพิ่งเพิ่มเข้าไปใหม่

A.3.2. มีนิพจน์ที่ใหญ่แล้วก็ดูยากมาก แล้วมันก็ไม่ทำงานแบบที่คิด

การเขียนนิพจน์ที่ซับซ้อน ก็ไม่ได้ผิดอะไร ถ้ามันยังอ่านรู้เรื่องอยู่ แต่มันอาจจะทำให้ดีบักได้ยาก จริง ๆ แล้วมันดีกว่าที่จะแตกนิพจน์ที่ซับซ้อน ออกเป็นนิพจน์ย่อย ๆ ที่กำหนดค่าให้กับตัวแปรชั่วคราวต่าง ๆ

ตัวอย่างเช่น:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

อันนี้อาจจะเขียนใหม่เป็น:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

รูปแบบหลังนี้ อ่านได้ง่ายกว่า เพราะว่า ชื่อตัวแปรบอกความหมาย และมันก็ง่ายที่จะดีบักด้วย เพราะว่าเราสามารถดูชนิดของค่าต่าง ๆ ในกระบวนการคำนวณ ผ่านตัวแปรชั่วคราวได้

ปัญหาอีกอย่าง ที่อาจเกิดกับนิพจน์ที่ซับซ้อน คือ ลำดับของการคำนวณอาจจะตรงตามที่เราคิด ตัวอย่างเช่น ถ้าเราตีความนิพจน์ $\frac{x}{2\pi}$ เป็นโปรแกรมไพธอน เราอาจจะเขียน:

```
y = x / 2 * math.pi
```

ซึ่งมันไม่ถูก เพราะว่าการคูณและการหารมีลำดับการทำก่อนหลังเท่ากัน และจะถูกทำจากซ้ายไปขวา ดังนั้นนิพจน์ของโปรแกรมไพธอนนี้ คือ $x\pi/2$.

วิธีแก้ปัญหาที่ดีคือการใส่วงเล็บเข้าไป เพื่อกำหนดลำดับการทำให้ชัดเจน:

```
y = x / (2 * math.pi)
```

ถ้าเราไม่แน่ใจลำดับการทำก่อนหลังให้ใช้วงเล็บช่วย การใช้วงเล็บช่วย นอกจากจะช่วยให้โปรแกรมทำงานถูกต้อง (ในแง่ที่ว่าทำงานตรงตามที่เราตั้งใจ) แล้วมันยังช่วยให้โปรแกรมอ่านได้ง่ายขึ้นด้วย โดยเฉพาะสำหรับคนที่จำลำดับการทำปฏิบัติการไม่ได้

A.3.3. มีฟังก์ชันที่มันให้ค่าออกมาไม่เหมือนที่คิด

ถ้ามีคำสั่ง `return` กับนิพจน์ที่ซับซ้อน เราจะพิมพ์ผลลัพธ์ออกมาดูได้ยาก แต่ก็เหมือนกับเทคนิคที่อภิปรายไป เราสามารถใช้ตัวแปรชั่วคราวมาช่วยปรับให้นิพจน์อ่านง่ายขึ้น ตัวอย่างเช่น แทนที่:

```
return self.hands[i].removeMatches()
```

เราอาจจะเขียนโปรแกรมเป็น:

```
count = self.hands[i].removeMatches()
return count
```

ตอนนี้ เราสามารถพิมพ์ค่า `count` ออกมาดูก่อนได้ง่ายขึ้น

A.3.4. ติด ช่วยหน่อย

อันดับแรก ลองพักจากคอมพิวเตอร์ไปทำอย่างอื่นสักประเดี๋ยว คอมพิวเตอร์ปล่อยคลื่นที่ส่งผลกับสมอง ที่อาจจะทำให้เกิดอาการ:

- สับสนและหงุดหงิด
- งมงาย (“คอมพิวเตอร์มันเกลียดเรา” “โปรแกรมจะทำงานได้ เฉพาะตอนที่เราใส่หมวกกลับหลัง”).
- เขียนโปรแกรมแบบเดินสุ่ม (ลองมั่วเขียนโปรแกรมมันทุก ๆ แบบ และเลือกเอาแบบที่ทำงานได้)

ถ้ารู้สึกตัวเองว่ากำลังมีอาการเหล่านี้ พักและออกไปเดินเล่นก่อน พอสงบแล้วค่อยกลับมาคิดโปรแกรมต่อ โปรแกรมมันทำอะไรกันแน่? อะไรที่มันจะทำให้เกิดพฤติกรรมแบบนั้นได้? ครั้งหลังสุดตอนไหนที่โปรแกรมมันทำงานได้ และเราทำอะไรไปหลังจากนั้น?

บางครั้งมันก็แค่ต้องใช้เวลาบ้างเพื่อจะแก้ปัญหา บ่อยครั้งเลยที่เราจะดีบักได้ตอนที่เราพักจากคอมพิวเตอร์ และใจเราผ่อนคลายออก ในจังหวะที่ที่ดีบักได้ดีที่สุด คือ กำลังอาบน้ำ และก็กำลังจะหลับ

A.3.5. ไม่ได้จริง ๆ มาช่วยดูให้หน่อย

บางทีมันก็แก้เองไม่ได้จริง ๆ แม้แต่โปรแกรมเมอร์ที่เก่งที่สุด บางครั้งก็ติดเหมือนกัน บางครั้งเราอยู่กับโปรแกรมนานเกินไป จนทำให้เรามองไม่เห็นข้อผิดพลาด เราต้องการใจเบา ๆ กับสายตาสด ๆ คู่ใหม่

ก่อนที่เราจะไปตามคนมาช่วยให้เตรียมตัวก่อน โปรแกรมเราอาจจะเรียบง่ายที่สุด และเราก็ควรจะลองกับอินพุตที่เล็กที่สุด แล้วที่จะเห็นข้อผิดพลาด เราควรจะวางคำสั่ง `print` ไว้ที่ที่เหมาะสมเรียบร้อยแล้ว (และเอาต์พุตก็ควรจะเข้าใจได้ง่าย) เราควรจะต้องเข้าใจโปรแกรมดีพอที่จะอธิบายได้อย่างชัดเจน และกระชับ

เวลาที่เอาคนมาช่วย ให้แน่ใจว่า เราให้ข้อมูลต่าง ๆ ที่เขาต้องการแล้ว:

- ถ้ามีข้อความแจ้งข้อผิดพลาด มันบอกว่าอะไร และส่วนไหนของโปรแกรมที่มันแจ้ง?
- เราเพิ่งทำอะไรไป ก่อนที่จะเกิดปัญหาขึ้น? บรรทัดไหนบ้างของโปรแกรมที่เพิ่งเขียนเข้าไป หรือกรณีทดสอบไหนที่เพิ่งใส่เข้าไปแล้วโปรแกรมไม่ผ่าน?
- เราได้ลองอะไรไปแล้วบ้าง และเรารู้อะไรแล้วบ้าง?

เวลาที่เจอสาเหตุของปัญหาแล้ว ให้สละเวลา ลองคิดดูว่า เราน่าจะทำอะไรบ้าง เพื่อจะหามันเจอได้ง่ายขึ้นคราวหน้าถ้าเราเจออะไรคล้าย ๆ กัน เราจะได้ดีบักได้เร็วขึ้น

จำไว้ว่า เป้าหมายไม่ใช่แค่ให้โปรแกรมทำงานได้ เป้าหมายคือเรียนรู้ว่าจะทำอย่างไรให้โปรแกรมทำงานได้

B. การวิเคราะห์อัลกอริธึม

> ภาคผนวกนี้เป็นเนื้อหาที่ตัดตอนมาจาก *Think Complexity* โดย Allen B. Downey ซึ่งจัดพิมพ์โดย O'Reilly Media (2012) ซึ่งถ้าคุณอ่านหนังสือเล่มนี้จบแล้ว คุณอาจจะอยากไปอ่านหนังสือเล่มนั้นต่อ

การวิเคราะห์อัลกอริธึมเป็นสาขาหนึ่งของวิทยาการคอมพิวเตอร์ที่ศึกษาประสิทธิภาพของอัลกอริธึม โดยเฉพาะเวลาในการทำงานและพื้นที่ที่ต้องการ ดูเพิ่มที่ `http://en.wikipedia.org/wiki/Analysis_of_algorithms`

เป้าหมายในทางปฏิบัติของการวิเคราะห์อัลกอริธึมคือการทำนายประสิทธิภาพของอัลกอริธึมต่าง ๆ เพื่อเป็นแนวทางในการตัดสินใจออกแบบ

ระหว่างการรณรงค์หาเสียงเลือกตั้งประธานาธิบดีสหรัฐฯ ปี 2008 ผู้สมัครรับเลือกตั้ง บารัค โอบามา ถูกขอให้ทำการวิเคราะห์อย่างกะทันหันเมื่อเขาไปที่บริษัทกูเกิ้ล (Google) หัวหน้าผู้บริหาร เอริค ชมิดต์ (Eric Schmidt) หยอกโอบามา ถามเขาถึง "วิธีที่มีประสิทธิภาพที่สุดในการจัดเรียงจำนวนเต็ม 32 บิตจำนวน 1 ล้านจำนวน" โอบามาเล่นด้วย เขาตอบอย่างรวดเร็วว่า "ผมคิดว่าไม่น่าใช่วิธีบับเบิลสอร์ท" (I think the bubble sort would be the wrong way to go.) ดู `http://www.youtube.com/watch?v=k4RRi_ntQc8`

นี่เป็นความจริง บับเบิลสอร์ท (bubble sort) ตามแนวคิดนั้นเรียบง่าย แต่ช้าสำหรับชุดข้อมูลขนาดใหญ่ คำตอบที่ชมิดต์กำลังมองหาคือ เรดิซสอร์ท (radix sort ดู `http://en.wikipedia.org/wiki/Radix_sort`)[1]

[1] แต่ถ้าคุณได้รับคำถามแบบนี้ในการสัมภาษณ์ ผมคิดว่าคำตอบที่ดีกว่าคือ "วิธีที่เร็วที่สุดในการจัดเรียงจำนวนเต็มหนึ่งล้านตัวคือการใช้ฟังก์ชันการจัดเรียงใด ๆ ก็ตามที่มีให้โดยภาษาที่ผมใช้ ประสิทธิภาพดีพอสำหรับแอปพลิเคชันส่วนใหญ่ แต่ถ้าปรากฏว่าแอปพลิเคชันของผมช้าเกินไป ผมจะใช้ตัวสร้างโปรไฟล์เพื่อดูว่าเวลาที่ใช้ไปอยู่ที่ใด หากดูเหมือนว่าอัลกอริธึมการจัดเรียงที่เร็วกว่าจะมีผลมากต่อประสิทธิภาพ ผมก็คงจะมองหาอิมพลิเมนเตชั่นที่ดี ๆ ของเรดิซสอร์ทมาใช้"

เป้าหมายของการวิเคราะห์อัลกอริธึมคือการเปรียบเทียบอย่างมีความหมายระหว่างอัลกอริธึม แต่มีปัญหาบางประการ

- ประสิทธิภาพสัมพัทธ์ของอัลกอริธึมอาจขึ้นอยู่กับคุณลักษณะของฮาร์ดแวร์ ดังนั้นอัลกอริธึมหนึ่งอาจเร็วกว่าในเครื่อง A แต่อีกอันเร็วกว่าบนเครื่อง B วิธีแก้ปัญหาโดยทั่วไปคือการระบุ **โมเดลของเครื่อง**และวิเคราะห์จำนวนขั้นตอนหรือการดำเนินการที่อัลกอริธึมต้องการภายใต้โมเดลที่กำหนด

- ประสิทธิภาพสัมพัทธ์อาจขึ้นอยู่กับรายละเอียดของชุดข้อมูล ตัวอย่างเช่น อัลกอริธึมการเรียงลำดับบางอย่างจะทำงานเร็วขึ้นหากข้อมูลถูกจัดเรียงบางส่วนแล้ว อัลกอริธึมอื่นทำงานช้าลง ในกรณีนี้ วิธีทั่วไปในการหลีกเลี่ยงปัญหานี้คือการวิเคราะห์สถานการณ์ **กรณีที่เลวร้ายที่สุด (worst case)** บางครั้งการวิเคราะห์ประสิทธิภาพของกรณีโดยเฉลี่ยก็มีประโยชน์แต่มักจะยากกว่า และอาจไม่ชัดเจนว่าควรจะเฉลี่ยบนชุดข้อมูลใด

- ประสิทธิภาพสัมพัทธ์ยังขึ้นอยู่กับขนาดของปัญหาด้วย อัลกอริธึมการจัดเรียงที่รวดเร็วสำหรับลิสต์ขนาดเล็กอาจช้าสำหรับลิสต์ที่ยาว วิธีแก้ไขปัญหานี้ตามปกติคือการแสดงเวลาทำงาน (หรือจำนวนการดำเนินการ) เป็นฟังก์ชันของขนาดปัญหา และจัดกลุ่มฟังก์ชันเป็นหมวดหมู่ ที่ขึ้นกับเวลาทำงานเพิ่มเร็วขนาดไหนเมื่อขนาดปัญหาเพิ่มขึ้น

ข้อดีของการเปรียบเทียบประเภทนี้คือสามารถจำแนกอัลกอริธึมอย่างง่ายได้ ตัวอย่างเช่น ถ้าผมรู้ว่าเวลาการทำงานของอัลกอริธึม A มีแนวโน้มที่จะเป็นสัดส่วนกับขนาดของอินพุต n และอัลกอริธึม B มีแนวโน้มที่จะเป็นสัดส่วนกับ n^2 ผมคาดว่า A จะเร็วกว่า B อย่างน้อยก็สำหรับค่าขนาดใหญ่ของ n

การวิเคราะห์ประเภทนี้มีข้อควรระวังบางประการ แต่เราจะพูดถึงเรื่องนี้ในภายหลัง

B.1. ลำดับการเติบโต

สมมติว่าคุณได้วิเคราะห์สองอัลกอริธึมและแสดงเวลาทำงานในพจน์ของขนาดของอินพุต ซึ่งคือ อัลกอริธึม A ใช้ $100n + 1$ ขั้นตอนในการแก้ปัญหาขนาด n อัลกอริธึม B ใช้ $n^2 + n + 1$ ขั้นตอน

ตารางต่อไปนี้แสดงเวลาทำงานของอัลกอริธึมเหล่านี้สำหรับปัญหาขนาดต่าง ๆ

ขนาด อินพุต	เวลาการทำงาน ของอัลกอริธึม A	เวลาการทำงาน ของอัลกอริธึม B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

ที่ $n = 10$ อัลกอริธึม A ค่อนข้างแย่ ใช้เวลานานกว่าอัลกอริธึม B เกือบ 10 เท่า แต่สำหรับ $n = 100$ จะใกล้เคียงกัน และสำหรับค่า n ที่มากขึ้น A จะดีกว่า B มาก

เหตุผลพื้นฐานคือสำหรับค่าขนาดใหญ่ของ n ฟังก์ชันใด ๆ ที่มีพจน์ n^2 จะเติบโตเร็วกว่าฟังก์ชันที่มีพจน์นำเป็น n โดย **พจน์นำ (leading term)**คือพจน์ที่มีเลขชี้กำลังสูงสุด

อัลกอริธึม A มีค่าสัมประสิทธิ์ที่พจน์นำเป็น 100 ซึ่งอธิบายว่าทำไม B ทำงานดีกว่า A สำหรับ n ขนาดเล็ก แต่ไม่ว่าค่าสัมประสิทธิ์เป็นเท่าไร จะมีค่าของ n ค่าหนึ่งเสมอที่ทำให้ $an^2 > bn$ สำหรับค่าใด ๆ ของสัมประสิทธิ์ a และ b

เหตุผลเดียวกันนี้ใช้กับเงื่อนไขที่ไม่ใช่พจน์นำ แม้ว่าเวลาการทำงานของอัลกอริธึม A จะเป็น $n + 1000000$ แต่ก็ยังดีกว่าอัลกอริธึม B ถ้า n มีขนาดใหญ่พอ

โดยทั่วไป เราถือว่าอัลกอริธึมที่มีพจน์นำขนาดเล็กกว่า (มีเลขชี้กำลังเล็กกว่า) จะเป็นอัลกอริธึมที่ดีกว่าสำหรับปัญหาขนาดใหญ่ แต่สำหรับปัญหาขนาดเล็ก ๆ มันจะมี **จุดเปลี่ยน (crossover point)** ที่จะเปลี่ยนอัลกอริธึมที่ทำงานดีกว่า (ถ้าค่า n ขนาดเล็กลง) จุดเปลี่ยนนี้อยู่ตรงไหนก็ขึ้นกับรายละเอียดของอัลกอริธึม อินพุต และก็ฮาร์ดแวร์ ซึ่งมันจะไม่ใช่จุดประสงค์หลักของการวิเคราะห์อัลกอริธึม (แต่ก็ไม่ได้แปลว่าเราจะไม่ได้ตระหนักถึงมัน)

หากอัลกอริธึมสองชุดมีชั้นพจน์นำเหมือนกัน ก็ยากที่จะบอกว่าอันไหนดีกว่ากัน เพราะว่าคำตอบขึ้นอยู่กับรายละเอียด ดังนั้นสำหรับการวิเคราะห์อัลกอริธึม ฟังก์ชันที่มีพจน์นำเหมือนกันจะถือว่าเทียบเท่ากัน แม้ว่าจะมีค่าสัมประสิทธิ์ต่างกันก็ตาม

ลำดับการเติบโต (order of growth) เป็นกลุ่มของฟังก์ชัน(หรืออัลกอริธึม)ที่พฤติกรรมการเติบโตในระดับเดียวกัน พฤติกรรมการเติบโต หมายถึง พฤติกรรมการเพิ่มเวลาทำงานเมื่อขนาดปัญหาเพิ่ม[2] ตัวอย่างเช่น อัลกอริธึมต่าง ๆ ที่มีเวลาทำงานในพจน์ของขนาดปัญหาเป็น $2n$, $100n$ และ $n + 1$ จะจัดอยู่ในลำดับการเติบโตเดียวกัน ซึ่งเขียน $O(n)$ ใน**สัญกรณ์บิ๊กโอ (Big-Oh notation)** และมักเรียกว่าเป็นการ

[2]นอกจากเวลาแล้ว แนวคิดเดียวกันนี้ยังใช้กับการวิเคราะห์ในแง่การใช้หน่วยความจำอีกด้วย

เติบโต**เชิงเส้น (linear)** เพราะว่าอัลกอริธึมต่าง ๆ มีเวลาทำงานที่เพิ่มขึ้นเป็นเชิงเส้นเมื่อเทียบกับขนาดปัญหา n

ฟังก์ชันทั้งหมดที่มีพจน์นำ n^2 เป็นกลุ่ม $O(n^2)$ เรียกว่า **กำลังสอง (quadratic)**

ตารางต่อไปนี้แสดงลำดับการเติบโตบางส่วนที่พบบ่อยที่สุดในการวิเคราะห์อัลกอริธึม โดยเรียงตามลำดับที่แย่ลง

ลำดับของการเติบโต	ชื่อ
$O(1)$	คงที่
$O(\log_b n)$	ลอการิทึม (สำหรับฐาน b ใด ๆ)
$O(n)$	เชิงเส้น
$O(n \log_b n)$	ลิเนียริธมิค (linearithmic)
$O(n^2)$	กำลังสอง
$O(n^3)$	กำลังสาม
$O(c^n)$	เลขชี้กำลัง (สำหรับฐาน c ใด ๆ)

สำหรับนิพจน์ลอการิทึม ฐานของลอการิทึมไม่สำคัญ การเปลี่ยนฐานนั้นเทียบเท่ากับการคูณด้วยค่าคงที่ซึ่งไม่เปลี่ยนลำดับการเติบโต ในทำนองเดียวกัน อัลกอริธึมทั้งหมดในกลุ่มเลขชี้กำลังอยู่ในลำดับการเติบโตเดียวกัน โดยไม่คำนึงถึงฐานของเลขชี้กำลัง อัลกอริธึมในกลุ่มเลขชี้กำลังเติบโตอย่างรวดเร็ว ดังนั้นอัลกอริธึมในกลุ่มเลขชี้กำลังจึงมีประโยชน์สำหรับปัญหาขนาดเล็กเท่านั้น

แบบฝึกหัด **B.1.** *อ่านหน้าวิกิพีเดียเกี่ยวกับสัญกรณ์บิ๊กโอที่* `http://en.wikipedia.org/wiki/Big_O_notation` *และตอบคำถามต่อไปนี้*

1. *ลำดับการเติบโตของ $n^3 + n^2$ คืออะไร? แล้ว $1000000n^3 + n^2$ ล่ะคืออะไร? แล้ว $n^3 + 1000000n^2$ ล่ะคืออะไร?*
2. *ลำดับการเติบโตของ $(n^2 + n) \cdot (n + 1)$ คืออะไร? ก่อนที่คุณจะเริ่มคูณ จำไว้ว่าคุณต้องการแค่พจน์นำเท่านั้น*
3. *ถ้า f อยู่ใน $O(g)$ สำหรับฟังก์ชันที่ไม่ระบุ g แล้วเราจะพูดอะไรเกี่ยวกับ $af + b$ ได้บ้าง?*
4. *ถ้า f_1 และ f_2 อยู่ใน $O(g)$ แล้วเราจะพูดอะไรเกี่ยวกับ $f_1 + f_2$ ได้บ้าง?*
5. *ถ้า f_1 อยู่ใน $O(g)$ และ f_2 อยู่ใน $O(h)$ แล้วเราจะพูดอะไรเกี่ยวกับ $f_1 + f_2$ ได้บ้าง?*

   ถ้า f_1 อยู่ใน $O(g)$ และ f_2 อยู่ใน $O(h)$ แล้วเราจะพูดอะไรเกี่ยวกับ $f_1 \cdot f_2$ ได้บ้าง?

โปรแกรมเมอร์ที่ใส่ใจในประสิทธิภาพบางครั้งอาจรู้สึกไม่ค่อยเชื่อการวิเคราะห์ประเภทนี้ ซึ่งก็เข้าใจได้ บางครั้งสัมประสิทธิ์และพจน์ที่ไม่ใช่พจน์นำอาจทำให้เห็นการทำงานที่ต่างกันมาก บางครั้งรายละเอียดของฮาร์ดแวร์ ภาษาโปรแกรม และลักษณะของอินพุตสร้างความแตกต่างอย่างมาก และสำหรับปัญหาขนาดเล็ก ๆ พฤติกรรมเติบโตอาจมองเห็นไม่ชัดเท่าไร

แต่ถ้าคุณคำนึงถึงคำเตือนเหล่านั้น การวิเคราะห์อัลกอริธึมเป็นเครื่องมือที่มีประโยชน์ อย่างน้อยสำหรับปัญหาขนาดใหญ่ อัลกอริธึมที่ “ดีกว่า” มักจะดีกว่า และบางครั้งก็ดีกว่า*มาก* ความแตกต่างระหว่างอัลกอริธึมสองอัลกอริธึมที่มีลำดับการเติบโตเท่ากันมักจะเป็นปัจจัยคงที่ แต่ความแตกต่างระหว่างอัลกอริธึมที่ดีและอัลกอริธึมที่ไม่ดีนั้นไม่มีขอบเขต!

B.2. การวิเคราะห์การทำงานพื้นฐานของไพธอน

การดำเนินการทางพีชคณิตของไพธอนส่วนใหญ่แล้วจะเป็นกลุ่มเวลาคงที่ การคูณมักใช้เวลานานกว่าการบวกและการลบ และการหารใช้เวลานานกว่านั้นอีก แต่เวลาดำเนินการเหล่านี้ไม่ได้ขึ้นอยู่กับขนาดของตัวถูกดำเนินการ (จำนวนหลักของตัวเลข) ยกเว้นจำนวนเต็มที่มีขนาดใหญ่มาก ๆ ในกรณีนั้นเวลาทำงานจะเพิ่มขึ้นตามจำนวนหลัก

คำสั่งเชิงดัชนีได้แก่การอ่านหรือการเขียนอิลิเมนต์ในข้อมูลแบบลำดับหรือดิกชันนารีก็เป็นกลุ่มเวลาคงที่เช่นกัน โดยเกี่ยวกับขนาดของโครงสร้างข้อมูล

ลูป `for` ที่ท่องสำรวจข้อมูลแบบลำดับหรือดิกชันนารีมักจะเป็นกลุ่มเชิงเส้น ตราบเท่าที่การดำเนินการทั้งหมดในตัวลูปเป็นเวลาคงที่ ตัวอย่างเช่น การเพิ่มอิลิเมนต์ของลิสต์เป็นกลุ่มเชิงเส้น

```
total = 0
for x in t:
    total += x
```

ฟังก์ชันสำเร็จรูป `sum` ยังเป็นกลุ่มเชิงเส้นด้วยเพราะมันทำแบบเดียวกัน แต่มีแนวโน้มที่จะเร็วกว่าเพราะเป็นการใช้งานที่มีประสิทธิภาพมากกว่า ซึ่งในภาษาของการวิเคราะห์อัลกอริธึม เรียกว่า สัมประสิทธิ์ของพจน์นำมีค่าน้อยกว่า

ตามหลักการทั่วไป ถ้าเนื้อหาของลูปอยู่ใน $O(n^a)$ แล้วลูปทั้งหมดจะอยู่ใน $O(n^{a+1})$ ยกเว้น ถ้าคุณสามารถแสดงให้เห็นได้ว่ามีการออกจากลูปภายหลังจำนวนวนซ้ำที่กำหนด นั่นคือ ถ้าลูปวนซ้ำ k รอบ ไม่ว่าขนาดของปัญหา n จะเป็นเท่าไร ลูปก็อยู่ใน $O(n^a)$ แม้ค่า k ใหญ่ ๆ ก็ตาม

การคูณด้วย k ไม่ได้เปลี่ยนลำดับการเติบโต แม้แต่การหารด้วย ดังนั้นถ้าเนื้อความของลูปอยู่ใน $O(n^a)$ และรัน n/k ครั้ง การวนซ้ำจะอยู่ใน $O(n^{a+1})$ แม้สำหรับ k ขนาดใหญ่

การดำเนินการของสายอักขระและทูเพิลส่วนใหญ่เป็นกลุ่มเชิงเส้น ยกเว้นการเข้าถึงตำแหน่งตามดัชนีและการหาความยาว `len` ซึ่งเป็นเวลาคงที่ ฟังก์ชันสำเร็จรูป `min` และ `max` เป็นกลุ่มเชิงเส้น เวลาในการทำงานของการดำเนินการตัดช่วงเป็นสัดส่วนกับความยาวของเอาต์พุต แต่ไม่ขึ้นกับขนาดของอินพุต

การต่อสายอักขระมีการเติบโตเป็นกลุ่มเชิงเส้น เวลาในการทำงานขึ้นอยู่กับผลรวมของความยาวของตัวถูกดำเนินการ

เมธอดของสายอักขระทั้งหมดเป็นกลุ่มเชิงเส้น แต่ถ้าความยาวของสายอักขระถูกจำกัดด้วยค่าคงที่ ตัวอย่างเช่น การดำเนินการกับอักขระตัวเดียว จะถือเป็นเวลาคงที่ เมธอดของสายอักขระ `join` เป็นแบบเชิงเส้น เวลาในการทำงานขึ้นอยู่กับความยาวทั้งหมดของสายอักขระ

เมธอดของลิสต์ส่วนใหญ่เป็นกลุ่มเชิงเส้น ยกเว้น

- การเพิ่มอิลิเมนต์ที่ส่วนท้ายของรายการ โดยเฉลี่ยแล้ว เป็นกลุ่มคงที่ เพราะว่าเมื่อพื้นที่หน่วยความจำที่ลิสต์อยู่มันหมดที่ว่าง บางครั้งลิสต์จะถูกคัดลอกไปยังตำแหน่งอื่นที่มีพื้นที่ใหญ่กว่า แต่เวลาทั้งหมดสำหรับการดำเนินการ n ครั้งคือ $O(n)$ ดังนั้นเวลาเฉลี่ยสำหรับการดำเนินการแต่ละครั้งคือ $O(1)$
- การลบอิลิเมนต์ออกจากตำแหน่งสุดท้ายของลิสต์เป็นเวลาคงที่
- การจัดเรียงเป็น $O(n \log n)$

การดำเนินการและเมธอดสำหรับดิกชันนารีส่วนใหญ่เป็นเวลาที่คงที่ ยกเว้น

- เวลาในการทำงานของ `update` เป็นสัดส่วนกับขนาดของดิกชันนารีที่ส่งผ่านเป็นพารามิเตอร์ ไม่ใช่ดิกชันนารีที่กำลังถูกปรับปรุงค่า
- `keys`, `values` และ `items` ทำงานด้วยเวลาคงที่ เพราะมันส่งคืนตัววนซ้ำ แต่ถ้าคุณวนซ้ำผ่านตัววนซ้ำ การวนซ้ำจะเป็นกลุ่มเชิงเส้น

ประสิทธิภาพการทำงานของดิกชันนารีเป็นหนึ่งในปาฏิหาริย์เล็ก ๆ ของวิทยาการคอมพิวเตอร์ เราจะดูว่ามันทำงานอย่างไรในหัวข้อ B.4

***แบบฝึกหัด* B.2.** *อ่านหน้าวิกิพีเดียเกี่ยวกับอัลกอริธึมการเรียงลำดับที่ `http://en.wikipedia.org/wiki/Sorting_algorithm` และตอบคำถามต่อไปนี้:*

1. *วิธีจัดลำดับเปรียบเทียบ (comparison sort) คืออะไร? มันมีลำดับการเติบโตที่ดีที่สุด ในกรณีที่แย่ที่สุด เป็นอะไร? อะไรคือลำดับการเติบโตที่ดีที่สุด ในกรณีที่แย่ที่สุดของทุก ๆ อัลกอริธึมการจัดเรียง?*

2. *อะไรคือลำดับการเติบโตของบับเบิลสอร์ท (bubble sort) และทำไมบารัค โอบามาถึงคิดว่า มันไม่น่าจะใช่?*

3. *ลำดับการเติบโตของเรดิซสอร์ท (radix sort) คืออะไร? เราจำเป็นต้องใช้เงื่อนไขก่อนอะไรบ้าง?*

4. *การจัดเรียงที่มั่นคง (stable sort) คืออะไร และทำไมมันอาจมีความสำคัญในทางปฏิบัติ*

5. *อัลกอริธึมการเรียงลำดับที่แย่ที่สุด (ที่มีชื่อ) คืออะไร?*

6. *ไลบรารีภาษาซี (C library) ใช้อัลกอริธึมการเรียงลำดับแบบใด? ไพธอนใช้อัลกอริธึมการเรียงลำดับแบบใด? อัลกอริธึมเหล่านี้เสถียรหรือไม่ คุณอาจต้องใช้กูเกิ้ลเพื่อค้นหาคำตอบเหล่านี้*

7. *การเรียงลำดับที่ไม่เปรียบเทียบจำนวนมากเป็นกลุ่มเชิงเส้น ทำไมไพธอนจึงใช้การเรียงลำดับที่ใช้การเปรียบเทียบที่อยู่ในกลุ่ม $O(n \log n)$?*

B.3. การวิเคราะห์อัลกอริธึมการค้นหา

การค้นหา (search) คืออัลกอริธึมที่รับกลุ่มหมู่และเป้าหมาย แล้วหาว่าเป้าหมายอยู่ในกลุ่มหมู่หรือไม่ ซึ่งมักจะส่งคืนดัชนีของเป้าหมายออกมา

อัลกอริธึมการค้นหาที่ง่ายที่สุดคือ “การค้นหาเชิงเส้น” (linear search) ซึ่งท่องสำรวจรายการต่าง ๆ ในกลุ่มหมู่ตามลำดับ และหยุดหากพบเป้าหมาย ในกรณีที่เลวร้ายที่สุด มันอาจต้องสำรวจกลุ่มหมู่ทั้งหมด ดังนั้นเวลาทำงานจึงเป็นเชิงเส้น

ตัวดำเนินการ `in` สำหรับข้อมูลแบบลำดับใช้การค้นหาเชิงเส้น เมธอดของสายอักขระก็มีเช่นกัน เช่น `find` และ `count`

หากอิลิเมนต์ของข้อมูลแบบลำดับเรียงลำดับอยู่ เราสามารถใช้ **การค้นหาแบบแบ่งสองส่วน (bisection search)** ซึ่งเป็น $O(\log n)$ การค้นหาแบบสองส่วนคล้ายกับวิธีที่เราอาจใช้ เพื่อค้นหาคำในพจนานุกรม (พจนานุกรมกระดาษ ไม่ใช่โครงสร้างข้อมูล ดิกชันนารี) แทนที่จะเริ่มต้นที่จุดเริ่มต้นและตรวจสอบทีละรายการตามลำดับ เราเริ่มต้นด้วยรายการที่อยู่ตรงกลางและตรวจดูว่าคำที่เรากำลังมองหามาก่อนหรือหลัง

ถ้ามันมาก่อน ให้ค้นหาครึ่งแรกของข้อมูลแบบลำดับ ถ้าไม่อย่างนั้นเราจะไปหาที่ครึ่งหลัง อย่างไรก็ตามเราก็จะลดจำนวนรายการที่เหลือลงครึ่งหนึ่งได้

ถ้าข้อมูลแบบลำดับมี 1,000,000 อิลิเมนต์ จะใช้เวลาประมาณ 20 ขั้นในการหาคำที่ต้องการหรือสรุปว่าไม่มี ซึ่งเร็วกว่าการค้นหาเชิงเส้นประมาณ 50,000 เท่า

การค้นหาแบบแบ่งสองส่วนอาจเร็วกว่าการค้นหาเชิงเส้นมาก แต่ต้องอาศัยว่าข้อมูลเรียงตามลำดับมาแล้ว ซึ่งอาจต้องมีการทำงานเพิ่มเติม (ถ้าข้อมูลไม่ได้เรียงมา)

มีโครงสร้างข้อมูลอื่นซึ่ง**สามารถทำแฮชได้ (hashable)** จะค้นหาได้เร็วยิ่งขึ้น สามารถค้นหาได้ในเวลาคงที่และไม่ต้องการข้อมูลที่เรียงลำดับไว้ก่อน ดิกชันนารีของไพธอนถูกพัฒนาโดยใช้ตารางแฮช ซึ่งเป็นสาเหตุที่การดำเนินการกับดิกชันนารีส่วนใหญ่ รวมถึงตัวดำเนินการ `in` เป็นเวลาคงที่

B.4. ตารางแฮช (Hashtables)

เพื่อเข้าใจว่า **ตารางแฮช (hashtable)** ทำงานอย่างไร ทำไมประสิทธิภาพของมันถึงดีมาก เราจะเริ่มต้นด้วยการใช้งานแผนที่อย่างง่าย และค่อย ๆ ปรับปรุงจนกระทั่งเป็นตารางแฮช

ผมใช้ไพธอนเพื่อสาธิตการใช้งานเหล่านี้ แต่ในชีวิตจริง เราจะไม่เขียนโค้ดแบบนี้ในไพธอน คุณแค่ใช้ดิกชันนารี! ดังนั้นสำหรับส่วนที่เหลือของบทนี้ เราต้องจินตนาการว่าเราไม่มีดิกชันนารีใช้ และเราต้องการใช้โครงสร้างข้อมูลที่จับคู่จากกุญแจไปหาค่าของมัน การดำเนินการที่เราต้องดำเนินการคือ

`add(k, v)`: เพิ่มรายการใหม่ที่แปลงจากกุญแจ `k` ไปค่า `v` ถ้าเป็นไพธอนดิกชันนารี `d` แล้ว การดำเนินการนี้จะถูกเขียน `d[k] = v`

`get(k)`: ค้นหาและส่งคืนค่าที่สอดคล้องกับกุญแจ `k` ถ้าเป็นไพธอนดิกชันนารี `d` แล้ว การดำเนินการนี้จะถูกเขียน `d[k]` หรือ `d.get(k)`

สำหรับตอนนี้ เราสมมติว่ากุญแจแต่ละดอกจะปรากฏเพียงครั้งเดียว อิมพลิเมนเตชั่นที่ง่ายที่สุดของส่วนต่อประสานนี้คือใช้ลิสต์ของทูเพิล โดยที่ทูเพิลแต่ละตัวเป็นคู่กุญแจกับค่า

```
class LinearMap:

    def __init__(self):
        self.items = []
```

```
    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

เมธอด add เพิ่มทูเพิลของคู่กุญแจกับค่าเข้าไปในลิสต์ ซึ่งการดำเนินการนี้ใช้เวลาคงที่

เมธอด get ใช้ลูป for เพื่อค้นหาในลิสต์ หากพบกุญแจเป้าหมาย จะส่งคืนค่าที่คู่กันออกมา ไม่อย่างนั้นมันจะแจ้ง KeyError ดังนั้น get จึงเป็นกลุ่มเชิงเส้น

อีกทางเลือกหนึ่งคือเรียงลิสต์ตามกุญแจ จากนั้น get สามารถใช้การค้นหาแบบแบ่งสองส่วน ซึ่งเป็นกลุ่ม $O(\log n)$ แต่การแทรกอิลิเมนต์ใหม่กลางลิสต์เป็นกลุ่มเชิงเส้น ดังนั้นนี่อาจไม่ใช่วิธีที่ดีที่สุด มีโครงสร้างข้อมูลอื่น ๆ ที่สามารถใช้ในการเขียนโค้ดของ add และ get เป็นเวลาลอการิทึม (log time) ได้ แต่อย่างไรก็ยังไม่ดีเท่ากับเวลาคงที่ ดังนั้นก็เอาแบบนี้ไปก่อน

วิธีหนึ่งในการปรับปรุง LinearMap คือการแบ่งลิสต์คู่กุญแจกับค่าออกเป็นลิสต์ที่เล็กลงหลาย ๆ อัน นี่คืออิมพลิเมนเตชั่นที่เรียกว่า BetterMap ซึ่งเป็นลิสต์ของ LinearMap หนึ่งร้อยอัน เดี๋ยวเราจะได้เห็นว่า ลำดับการเติบโตของ get จะยังคงเป็นเชิงเส้นอยู่ แต่ BetterMap เป็นการก้าวไปสู่เส้นทางของตารางแฮช

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]
```

```
    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

เมธอด `__init__` สร้างลิสต์ของ `LinearMap` ขึ้นมา `n` ชุด

เมธอด `find_map` ถูกเรียกใช้ใน `add` และ `get` เพื่อหาว่าแผนที่ไหนที่จะใส่รายการใหม่เข้าไป หรือแผนที่ไหนที่จะเข้าไปค้นหา

เมธอด `find_map` ใช้ฟังก์ชันสำเร็จรูป `hash` ซึ่งฟังก์ชัน `hash` รับไพธอนอ็อบเจกต์และจะส่งคืนเลขจำนวนเต็มออกมา (ค่าแฮชของอ็อบเจกต์) ข้อจำกัดของอิมพลิเมนเตชั่นนี้คือ มันใช้งานได้เฉพาะกับกุญแจที่แฮชได้เท่านั้น ชนิดข้อมูลประเภทที่เปลี่ยนแปลงได้ เช่น ลิสต์และดิกชันนารีจะแฮชไม่ได้

อ็อบเจกต์ที่แฮชได้ที่เทียบเท่ากัน จะได้ค่าแฮชเดียวกัน แต่ในทางกลับกันอาจไม่จริง คืออ็อบเจกต์สองรายการที่มีค่าต่างกันสามารถได้ค่าแฮชเดียวกันได้

เมธอด `find_map` ใช้โมดูลัส `%` เพื่อคุมค่าแฮชให้อยู่ในช่วงตั้งแต่ 0 ถึง `len(self.maps)` ดังนั้นผลลัพธ์จึงเป็นดัชนีที่ถูกต้องสำหรับลิสต์แน่นอน นี่หมายความว่าค่าแฮชหลาย ๆ ค่าจะกลายมาเป็นค่าดัชนีเดียวกัน แต่ถ้าฟังก์ชันแฮชกระจายค่าแฮชไปได้ค่อนข้างดี (ซึ่งเป็นสิ่งที่ฟังก์ชันแฮชถูกสร้างมา) เราก็คาดหวังได้ว่า จะการจัดเก็บจะประมาณ $n/100$ อิลิเมนต์ต่อ `LinearMap` เมื่อ n คือจำนวนอิลิเมนต์ทั้งหมดที่จัดเก็บเข้าไป

เนื่องจากเวลารันของ `LinearMap.get` เป็นสัดส่วนกับจำนวนอิลิเมนต์ เราก็คิดว่า `BetterMap` น่าจะเร็วกว่า `LinearMap` ประมาณ 100 เท่า และลำดับการเติบโตยังคงเป็นเชิงเส้น แต่ค่าสัมประสิทธิ์ของพจน์นำจะน้อยกว่า นั่นถือว่าดีแต่ก็ยังไม่ดีเท่าตารางแฮช

ทีนี้เป็นแนวคิดสำคัญที่ทำให้ตารางแฮชเร็วขึ้น ถ้าเราสามารถคุมความยาวสูงสุดของ `LinearMap` ต่าง ๆ ได้ เมธอด `LinearMap.get` จะเป็นการดำเนินการในกลุ่มเวลาคงที่ ดังนั้นสิ่งที่เราต้องทำก็คือนับจำนวนอิลิเมนต์ และถ้าจำนวนอิลิเมนต์ของ `LinearMap` เกินเกณฑ์ที่กำหนด ให้ปรับขนาดของตารางแฮชโดยการเพิ่มจำนวน `LinearMap` เข้าไป

นี่คืออิมพลิเมนเตชั่นของตารางแฮช:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

คลาส HashMap แต่ละอันมี BetterMap เมธอด __init__ เริ่มต้นด้วย BetterMap เพียง 2 อัน และกำหนดค่าเริ่มต้น num ซึ่งจะใช้นับจำนวนอิลิเมนต์

เมธอด get แค่ส่งกุญแจผ่านไปให้ BetterMap เมธอด add มีการทำงานจริง ๆ อยู่ ซึ่งคือ การตรวจสอบจำนวนอิลิเมนต์และขนาดของ BetterMap หากเท่ากัน แปลว่าจำนวนเฉลี่ยของอิลิเมนต์ต่อ LinearMap เป็น 1 แล้ว ดังนั้นจึงเรียกใช้ resize

เมธอด resize สร้าง BetterMap ขึ้นมาใหม่ โดยให้ใหญ่เป็นสองเท่าของแผนที่เดิม จากนั้นก็ทำแฮช

ใหม่ (rehash) กับอิลิเมนต์จากแผนที่เดิมไปแผนที่ใหม่

เราจำเป็นต้องทำการแฮชใหม่ เนื่องจากการเปลี่ยนจำนวน `LinearMap` จะเปลี่ยนตัวหารของโมดูลัสใน `find_map` นั่นหมายความว่าบางอ็อบเจกต์ที่เคยแฮชลงใน `LinearMap` อันเดียวกันจะถูกแยกออกออกไปคนละอัน

การแฮชใหม่เป็นเวลาเชิงเส้น ดังนั้น `resize` จึงเป็นเวลาเชิงเส้น ซึ่งอาจฟังดูแย่ เพราะผมเคยสัญญาว่า `add` จะเป็นเวลาคงที่ แต่จำไว้ว่าเราไม่จำเป็นต้องปรับขนาดทุกครั้ง ดังนั้น `add` ส่วนใหญ่แล้วจะเป็นเวลาคงที่และจะเป็นเวลาเชิงเส้นแค่บางครั้งเท่านั้น จำนวนครั้งที่จะรันเมธอด `add` ทั้งหมด n ครั้งเป็นสัดส่วนกับค่า n ดังนั้นเวลาเฉลี่ยของ `add` แต่ละครั้งคือเวลาคงที่!

หากต้องการดูวิธีการทำงาน ให้นึกถึงการเริ่มต้นด้วยตารางแฮชว่าง ๆ แล้วค่อย ๆ เพิ่มอิลิเมนต์เข้าไป เราเริ่มต้นด้วย `LinearMap` สองอัน ดังนั้นการเพิ่ม 2 รายการแรกนั้นรวดเร็ว (ไม่จำเป็นต้องปรับขนาด) สมมติว่าแต่ละครั้งทำงานหนึ่งหน่วย การเพิ่มครั้งต่อไปต้องมีการปรับขนาด ดังนั้นเราจึงต้องทำแฮชใหม่สองรายการแรก (ซึ่งเรียกเพิ่มอีก 2 หน่วยทำงาน) แล้วค่อยเพิ่มรายการที่สาม (อีกหนึ่งหน่วย) การเพิ่มรายการถัดไปนับเป็น 1 หน่วยดังนั้นยอดรวมคือ 6 หน่วยสำหรับ 4 รายการ

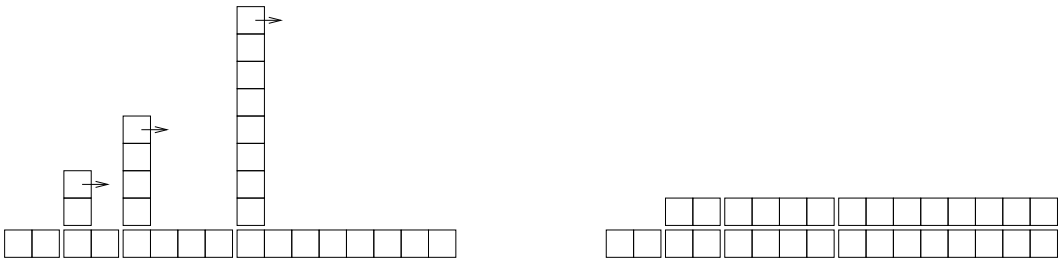
การ `add` ครั้งต่อไปนับเป็นงาน 5 หน่วย (ปรับขนาด ทำแฮชใหม่สี่รายการแรก และแล้วค่อยเพิ่มรายการที่ห้า) แต่สามรายการถัดไปมีเพียงครั้งละหนึ่งหน่วย ดังนั้นยอดรวมคือ 14 หน่วยสำหรับการเพิ่ม 8 ครั้งแรก

การ `add` ครั้งต่อไปนับเป็นงาน 9 หน่วย แต่จากนั้นเราสามารถเพิ่มอีก 7 ครั้งก่อนการปรับขนาดครั้งต่อไป ดังนั้นยอดรวมคือ 30 หน่วยสำหรับการเพิ่ม 16 รายการแรก

หลังจากเพิ่ม 32 รายการ นับงานรวมได้ทั้งหมดคือ 62 หน่วย และผมว่าคุณน่าจะเริ่มเห็นรูปแบบแล้ว (ผู้แปล: $n = 2$, $cost = 2$; $n = 4$, $cost = 6$; $n = 8$, $cost = 14$; $n = 16$, $cost = 30$; $n = 32$, $cost = 62$) หลังจากเพิ่ม n ครั้ง ณ ค่า n เป็นค่ายกกำลังของสอง (ผู้แปล: เช่น 2, 4, 8, 16, 32) ต้นทุนงานรวมคือ $2n - 2$ หน่วย ดังนั้นงานเฉลี่ยต่อการเพิ่มจะน้อยกว่า 2 เท่าเล็กน้อย นั่นคือ ที่ค่า n เป็นค่ายกกำลังของสองคือกรณีที่ดีที่สุด สำหรับค่าอื่น ๆ ของ n งานเฉลี่ยจะต่างไปเล็กน้อย แต่นั่นไม่สำคัญ ที่สำคัญคือ การ `add` เป็นกลุ่ม $O(1)$

รูปที่ B.1 แสดงวิธีการทำงานแบบกราฟิก แต่ละบล็อกแสดงถึงหน่วยของงาน คอลัมน์แสดงงานทั้งหมดสำหรับการเพิ่มแต่ละรายการโดยเรียงลำดับจากซ้ายไปขวา สองรายการแรก `add` ต้นทุน 1 หน่วย รายการที่สามต้นทุน 3 หน่วย ฯลฯ

งานที่เพิ่มขึ้นมาเพื่อทำการแฮชใหม่แสดงเป็นเหมือนหอคอยของกองบล็อกที่สูงขึ้น ๆ และระยะห่างระหว่างหอคอยที่เพิ่มขึ้น ๆ ตอนนี้ถ้ามองเหมือนว่า เราแตกกองบล็อกจากหอคอย แล้วกระจายค่าใช้จ่าย



รูปที่ B.1.: ค่าใช้จ่าย (หน่วยงานที่ต้องทำ) ของการเพิ่มแฮชเทเบิล

ในการปรับขนาดไปยังการดำเนินการ **add** ทั้งหมด เราจะเห็นภาพว่าค่าใช้จ่ายทั้งหมดหลังจากเราทำ **add** ไป n ครั้ง จะเป็นราว ๆ $2n - 2$

คุณลักษณะที่สำคัญของอัลกอริธึมนี้คือเมื่อเราปรับขนาดตารางแฮชแล้ว มันจะเติบโตเชิงเรขาคณิต นั่นคือเราคูณขนาดด้วยค่าคงที่ หากเราเพิ่มขนาดเชิงพีชคณิต (การบวกจำนวนคงที่ในแต่ละครั้ง) เวลาเฉลี่ยต่อการทำ **add** จึงเป็นกลุ่มเชิงเส้น

คุณสามารถดาวน์โหลด `HashMap` ได้จาก `http://thinkpython2.com/code/Map.py` แต่ระลึกไว้ว่าไม่มีเหตุผลที่จะต้องใช้มัน หากคุณต้องการแผนที่ก็แค่ใช้ดิกชันนารีของไพธอน

B.5. อภิธานศัพท์

การวิเคราะห์อัลกอริธึม (analysis of algorithms): วิธีเปรียบเทียบอัลกอริธึมในแง่ของเวลาใช้งานและหรือข้อกำหนดด้านหน่วยความจำ

โมเดลของเครื่อง (machine model): แบบจำลองคอมพิวเตอร์ หรือภาพที่มองคอมพิวเตอร์แบบง่าย ๆ เพื่อใช้อธิบายอัลกอริธึม

กรณีที่แย่ที่สุด (worst case): อินพุตที่ทำให้อัลกอริธึมที่สนใจทำงานช้าที่สุด (หรือต้องการพื้นที่หน่วยความจำมากที่สุด)

พจน์นำ (leading term): พจน์ที่มีเลขชี้กำลังสูงสุดในพหุนาม

จุดเปลี่ยน (crossover point): ขนาดปัญหาที่อัลกอริธึมสองตัวต้องใช้เวลาหรือพื้นที่ทำงานเท่ากัน

ลำดับการเติบโต (order of growth): ชุดของฟังก์ชันที่เติบโตในลักษณะที่เทียบเท่ากันตาม

วัตถุประสงค์ของการวิเคราะห์อัลกอริธึม ตัวอย่างเช่น ฟังก์ชันทั้งหมดที่ใช้เวลาทำงานแบบเชิงเส้นอยู่ในลำดับการเติบโตกลุ่มเดียวกัน

สัญกรณ์บิ๊กโอ (Big-Oh notation): สัญกรณ์สำหรับแสดงลำดับการเติบโต ตัวอย่างเช่น $O(n)$ แทนชุดของฟังก์ชันที่เติบโตในกลุ่มเชิงเส้น

เชิงเส้น (linear): อัลกอริธึมที่เวลาทำงานเป็นสัดส่วนกับขนาดของปัญหา อย่างน้อยก็สำหรับปัญหาขนาดใหญ่

กำลังสอง (quadratic): อัลกอริธึมที่มีเวลาทำงานเป็นสัดส่วนกับ n^2 โดยที่ n คือขนาดของปัญหา

การค้นหา (search): ปัญหาในการหาตำแหน่งอิลิเมนต์ของกลุ่มหมู่ (เช่น ลิสต์หรือดิกชันนารี) หรือการพบว่าไม่มีอิลิเมนต์นั้นอยู่

ตารางแฮช (hashtable): โครงสร้างข้อมูลที่แสดงถึงชุดของคู่กุญแจกับค่าและทำการค้นหาในเวลาคงที่

ดรรชนี