A Whirlwind Tour of Python (Thai Translation)

The translation is done by Sukanya Suranauwarat
under a non-commercial, non-exclusive license.

The original English version written by Jake VanderPlas can be found at:
https://learning.oreilly.com/library/view/a-whirlwind-tour/9781492037859/

สารบัญ

Introduction

ภาษา Python เป็นภาษาที่ถูกพัฒนาขึ้นมาในช่วงปลายทศวรรษ 1980 เพื่อใช้เป็นภาษาสำหรับการสอนและการเขียนสคริปต์ นับตั้งแต่นั้นเป็นต้นมา ภาษา Python ได้กลายเป็นเครื่องมือสำคัญของโปรแกรมเมอร์วิศวกร นักวิจัยและนักวิทยาศาสตร์ข้อมูลจำนวนมากทั้งจากภาคสถาบันการศึกษาและภาคอุตสาหกรรม ในฐานะนักดาราศาสตร์ที่มุ่งเน้นการพัฒนาและการส่งเสริมเครื่องมือแบบฟรีโอเพ่นซอร์สสำหรับงานวิทยาศาสตร์ที่มีการใช้ข้อมูลจำนวนมาก ผู้เขียนพบว่า Python เป็นภาษาที่เหมาะกับการแก้ปัญหาประเภทต่าง ๆ ที่ผู้เขียนพบเจอในแต่ละวัน ไม่ว่าจะเป็นการค้นหาความหมายจากชุดข้อมูลทางดาราศาสตร์ขนาดใหญ่ การดึงข้อมูลบนเว็บไซต์เพื่อนำไปใช้งาน การทำความสะอาดข้อมูลให้อยู่ในรูปที่สามารถนำเอาไปใช้งานได้ หรือการทำ automation กับงานที่ต้องใช้ในงานวิจัย

เสน่ห์ของภาษา Python อยู่ที่ความเรียบง่ายและความสวยงามของตัวภาษา รวมทั้งความพร้อมของเครื่องมือสำหรับใช้ในการแก้ปัญหาเฉพาะด้านที่มีอยู่จำนวนมากและสามารถนำเอาไปใช้ในโปแกรมได้เลย ยกตัวอย่างเช่น โค้ดภาษา Python ส่วนใหญ่ที่ใช้ในงานคำนวณทางวิทยาศาสตร์และงานวิทยาการข้อมูล จะถูกเขียนโดยอาศัยแพ็คเกจ (package) ที่ผ่านการพัฒนาแล้วมาเป็นอย่างดีและมีประโยชน์ อย่างเช่น

- NumPy มีความสามารถในการจัดการกับอาเรย์หลายมิติ
- SciPy ประกอบด้วยเครื่องมือทางคณิตศาสตร์จำนวนมาก อย่างเช่น numerical integration และ interpolation
- Pandas มีออบเจ็กต์ DataFrame พร้อมกับชุดเมทธอดที่มีประสิทธิภาพในการจัดการ การกรอง การจัดกลุ่ม และการแปลงข้อมูล
- Matplotlib มีอินเทอร์เฟซที่เป็นประโยชน์สำหรับการพล็อตกราฟ
- Scikit-Learn มีชุดเครื่องมือสำหรับประยุกต์ใช้อัลกอริทึมการเรียนรู้ของเครื่องจักร (machine learning algorithm) ที่ใช้กันอยู่ทั่วไปกับข้อมูล
- IPython/Jupyter เป็น terminal ที่มีขีดความสามารถสูงและ interactive notebook environment ที่สามารถนำมาใช้สร้างเอกสารที่ประกอบด้วยคำบรรยาย การวิเคราะข้อมูล และโค้ดที่สามารถเอ็กซิคิวท์ได้ ตัวอย่างเช่น หนังสือเล่มนี้เขียนโดยใช้ Jupyter notebook

นอกจากนี้ ยังมีเครื่องมือและแพ็คเกจอื่น ๆ อีกมากมายที่ไม่ได้มีความสำคัญน้อยไปกว่าแพ็คเกจเหล่านี้ ถ้าหากผู้อ่านมีงานที่เกี่ยวข้องกับงานทางด้านวิทยาศาสตร์หรือการวิเคราะห์ข้อมูลที่ต้องทำ จะมีโอกาสสูงมากที่มีคนเขียนแพ็คเกจเอาไว้แล้วและสามารถนำไปใช้กับงานของผู้อ่านได้เลย

อย่างไรก็ตาม ก่อนที่จะใช้แพ็คเกจทางด้านวิทยาการข้อมูลอันทรงพลังนี้ได้ ขั้นแรกเราต้องทำความคุ้นเคยกับตัวภาษา Python ก่อน ผู้เขียนมักพบนักเรียนและเพื่อนร่วมงานที่มีแบคกราวน์ (บางครั้งมากด้วย) ในบางภาษา อย่างเช่น MATLAB, IDL, R, Java, C++ และอื่น ๆ และกำลังมองหาหนังสือที่แนะนำภาษา Python ที่มีเนื้อหาครอบคลุมแต่สั้นกระชับและเหมาะกับระดับความรู้แบคกราวน์ มากกว่าหนังสือที่แนะนำภาษาที่เริ่มต้นจากศูนย์ หนังสือเล่มนี้พยายามเติมเต็มความต้องการดังกล่าวนี้

ด้วยเหตุนี้ หนังสือเล่มนี้จึงไม่ได้มีจุดมุ่งหมายเพื่อเป็นหนังสือสำหรับแนะนำการเขียนโปรแกรมเบื้องต้นที่มีเนื้อหาครอบคลุมทุกเรื่อง หรือหนังสือสำหรับแนะนำภาษา Python อย่างสมบูรณ์ ซึ่งหากนั่นคือสิ่งที่ผู้อ่านกำลังมองหา ผู้อ่านอาจจะลองดูหนึ่งในรายการหนังสือแนะนำที่อยู่ในบทเรื่อง Resources for Further Learning หนังสือเล่มนี้จะเป็นการแนะนำบางส่วนของไวยากรณ์ (syntax) และความหมาย (semantic) ที่สำคัญของภาษา Python, ประเภทข้อมูลในตัวภาษา Python (built-in data type) และโครงสร้าง, การนิยามฟังก์ชัน (function definition), คำสั่งควบคุม (control flow statement) และแง่มุมอื่น ๆ ของภาษา เป้าหมายของผู้เขียนคือเมื่อผู้อ่านอ่านหนังสือเล่มนี้จบแล้ว ผู้อ่านจะมีพื้นฐานภาษา Python ที่ดีที่สามารถจะนำไปใช้ต่อยอดกับงานทางด้านวิทยาการข้อมูลที่ได้กล่าวไปในข้างต้น

Using Code Examples

สื่อการเรียนเสริมของหนังสือเล่มนี้ (เช่น โค้ดตัวอย่างและ IPython notebook) สามารถดาวน์โหลดได้ที่ https://github.com/jakevdp/WhirlwindTourOfPython/

หนังสือเล่มนี้มีไว้เพื่อช่วยให้ผู้อ่านทำงานได้สำเร็จ ดังนั้นผู้อ่านสามารถนำโค้ดตัวอย่างที่มาพร้อมกับหนังสือเล่มนี้ไปใช้ในโปรแกรมและเอกสารของผู้อ่านได้ ผู้อ่านไม่จำเป็นต้องติดต่อสำนักพิมพ์เพื่อขออนุญาตใช้โค้ดตัวอย่าง ยกเว้นกรณีที่ผู้อ่านจะนำโค้ดส่วนใหญ่ไปแจกจ่าย ยกตัวอย่างเช่น การเขียนโปรแกรมที่ใช้โค้ดหลายส่วนจากหนังสือเล่มนี้ไม่จำเป็นต้องขออนุญาต การขายหรือแจกจ่าย CD-ROM ของโค้ดตัวอย่างจากหนังสือของสำนักพิมพ์ O'Reilly ต้องได้รับอนุญาตก่อน การตอบคำถามโดยอ้างอิงจากหนังสือเล่มนี้และโค้ดตัวอย่างไม่ต้องขออนุญาตการผนวกโค้ดตัวอย่างจากหนังสือเล่มนี้จำนวนมากลงในเอกสารประกอบผลิตภัณฑ์ของผู้อ่านต้องได้รับอนุญาตก่อน

ทางสำนักพิมพ์ขอขอบคุณถ้าผู้อ่านจะระบุแหล่งที่มาของหนังสือเล่มนี้เวลาอ้างอิง แต่ไม่จำเป็น โดยปกติแล้วการระบุแหล่งที่มาจะประกอบด้วย ชื่อหนังสือ ชื่อผู้แต่ง สำนักพิมพ์ และ ISBN เช่น A Whirlwind Tour of Python by Jake VanderPlas (O'Reilly). Copyright 2016 O'Reilly Media, Inc., 978-1-491-96465-1.

ถ้าหากผู้อ่านรู้สึกว่าการใช้โค้ดตัวอย่างของผู้อ่านอยู่นอกเหนือการใช้งานที่ได้รับอนุญาตที่ได้กล่าวไว้ในข้างต้น ผู้อ่านสามารถติดต่อทางสำนักพิมพ์ได้ที่ permissions@oreilly.com

Installation and Practical Considerations

การติดตั้ง Python และชุดไลบรารี (library) สำหรับงานคำนวณทางวิทยาศาสตร์นั้นมีขั้นตอนที่ตรงไปตรงมา ไม่ว่าผู้อ่านจะใช้ Windows, Linux หรือ Mac OS X ในหัวข้อนี้ ผู้เขียนจะสรุปข้อควรพิจารณาบางประการในการติดตั้ง Python

Python 2 versus Python 3

หนังสือเล่มนี้ใช้ไวยากรณ์ของภาษา Python เวอร์ชั่น 3 ซึ่งมีส่วนที่ปรับปรุงเพิ่มเติมที่ไม่สามารถใช้งานได้กับ Python เวอร์ชั่น 2.x ซีรีย์ ถึงแม้ว่า Python เวอร์ชั่น 3.0 จะเปิดตัวครั้งแรกในปี 2008 แต่การเปลี่ยนไปใช้เวอร์ชั่นนี้เป็นไปค่อนข้างช้าโดยเฉพาะอย่างยิ่งในกลุ่มผู้ที่ทำงานทางด้านวิทยาศาสตร์และกลุ่มผู้พัฒนาเว็บ ทั้งนี้เป็นเพราะว่าต้องใช้เวลาพอสมควรในการทำให้แพ็คเกจและชุดเครื่องมือที่จำเป็นนั้นสามารถทำงานกับภาษา Python เวอร์ชั่นใหม่ได้ อย่างไรก็ตาม ตั้งแต่ต้นปี 2014 เป็นต้นมา เครื่องมือที่สำคัญสำหรับงานวิทยาการข้อมูลส่วนใหญ่ถูกทำให้สามารถทำงานได้กับทั้ง Python เวอร์ชั่น 2 และเวอร์ชั่น 3 ดังนั้นหนังสือเล่มนี้จะใช้ไวยากรณ์

ของ Python เวอร์ชั่น 3 ซึ่งใหม่กว่า ถึงแม้ว่าจะเป็นเช่นนี้ โค้ดตัวอย่างส่วนใหญ่ในหนังสือเล่มนี้สามารถใช้งานได้โดยไม่ต้องมีการแก้ไขกับภาษา Python เวอร์ชั่น 2 ในกรณีที่โค้ดตัวอย่างมีการใช้ไวยากรณ์ที่ไม่สามารถทำงานกับ Python เวอร์ชั่น 2 ได้ ผู้เขียนจะพยายามอย่างเต็มที่ที่จะเขียนแจ้งให้เห็นอย่างชัดเจน

Installation with conda

ถึงแม้ว่าจะมีหลายวิธีในการติดตั้ง Python แต่ถ้าผู้อ่านต้องการใช้เครื่องมือสำหรับงานทางด้านวิทยาการข้อมูลที่ได้กล่าวไปก่อนหน้านี้ในภายหลัง วิธีการติดตั้งที่ผู้เขียนจะแนะนำ คือ วิธีการติดตั้งโดยใช้ Anaconda ดิสทริบิวชั่น ซึ่งมีอยู่ 2 แบบ ดังนี้

- Miniconda จะติดตั้งตัวแปลภาษาภาษา Python พร้อมกับเครื่องมือที่เป็นแบบคอมมานด์ไลน์ที่เรียกว่า conda ซึ่งเป็นผู้จัดการแพ็คเกจ (package manager) ที่มีหน้าที่ในการดาว์นโหลดและติดตั้งแพ็คเกจต่าง ๆ ของ Python ในทำนองเดียวกันกับ apt หรือ yum ที่ผู้ใช้ Linux อาจจะคุ้นเคย ผู้อ่านสามารถดาว์นโหลด Miniconda ได้ที่ https://docs.conda.io/en/latest/ miniconda.html
- Anaconda มีทั้งตัวแปลภาษา Python และ conda นอกจากนี้ยังมีชุดของแพ็คเกจสำหรับงานคำนวณทางวิทยาศาสตร์จำนวนมากรวมอยู่ด้วย ผู้อ่านสามารถดาว์นโหลด Anaconda ได้ที่ https://www.anaconda.com/products/individual

เนื่องจากผู้อ่านสามารถติดตั้งแพ็คเกจใด ๆ ที่มาพร้อมกับ Anaconda ในภายหลังได้ด้วยตนเอง ดังนั้นผู้เขียนจึงอยากแนะนำให้เริ่มต้นด้วย Miniconda

ในการเริ่มต้น ให้ดาวน์โหลดและติดตั้งแพ็คเกจ Miniconda และอย่าลืมที่จะเลือกเวอร์ชั่นของ Minicoda ที่มี Python เวอร์ชั่น 3 จากนั้นให้ติดตั้งแพ็คเกจ IPython notebook ดังนี้

```
$ conda install ipython-notebook
```

ผู้อ่านสามารถศึกษาข้อมูลเพิ่มเติมเกี่ยวกับ conda รวมทั้งข้อมูลเกี่ยวกับการสร้างและการใช้สภาพแวดล้อม conda ได้ที่เว็บไซต์ของ conda ที่ได้ระบุไปในข้างต้นแล้วในเรื่องแพ็คเกจ Miniconda

The Zen of Python

ผู้ที่ชื่นชอบภาษา Python มักจะชี้ให้เห็นอย่างรวดเร็วว่าภาษา Python เป็นภาษาที่ “ใช้งานง่าย” “สวยงาม” หรือ“ สนุก” ถึงแม้ว่าผู้อ่านจะค่อนข้างเห็นด้วยกับคำกล่าวเหล่านี้ก็ตาม แต่ผู้อ่านก็รับรู้เช่นกันว่าความง่าย ความสวยงาม และความสนุกของภาษา Python นั้นมักจะพร้อมกับความคุ้นเคยกับตัวภาษาเอง และสำหรับผู้ที่คุ้นเคยกับภาษาโปรแกรมมิ่งอื่น ๆ อาจจะคิดว่าคำกล่าวอ้างดังกล่าวนี้เกินความเป็นจริง อย่างไรก็ตาม

ผู้เขียนหวังว่าถ้าผู้อ่านจะลองเปิดใจให้โอกาสกับภาษา Python ผู้อ่านอาจจะเห็นว่าความประทับใจของผู้ที่ชื่นชอบภาษา Python นั้นมาจากไหน และถ้าผู้อ่านต้องการเจาะลึกปรัชญาการเขียนโปรแกรมที่ขับเคลื่อนการเขียนโค้ดของผู้เขียนโปรแกรมภาษา Python ผู้อ่านสามารถดูผ่านตัวแปลภาษา Python ได้ ด้วยการรัน `import this` ดังนี้

```
In [1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way
    to do it.
Although that way may not be obvious at first unless
    you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea--let's do more of those!
```

ถัดไปให้เรามาเริ่มดูภาษา Python กัน

How to Run Python Code

Python เป็นภาษาที่ยืดหยุ่นและมีหลายวิธีในการใช้งานซึ่งจะขึ้นอยู่กับงานของผู้ใช้งาน สิ่งหนึ่งที่ทำให้ภาษา Python แตกต่างจากภาษาโปรแกรมมิ่งอื่น ๆ อีกหลายภาษา คือ Python เป็นภาษาที่ใช้ตัวแปลภาษาแบบ interpreter ในขณะที่อีกหลาย ๆ ภาษาจะใช้ตัวแปลภาษาแบบ compiler หมายความว่า โปรแกรมภาษา Python จะถูกเอ็กซิคิวท์ทีละบรรทัด ทำให้การเขียนโปรแกรมเป็นแบบ interactive ในลักษณะที่ไม่สามารถเป็นไปได้โดยตรงกับภาษาที่ใช้ compiler อย่างเช่น Fortran, C หรือ Java ในบทนี้ผู้เขียนจะอธิบายสี่วิธีการหลักในการรันโค้ด Python ซึ่งได้แก่ การใช้ Python interpreter, การใช้ IPython interpreter, การใช้ self-contained สคริปต์ และการใช้ Jupyter notebook.

The Python interpreter

วิธีพื้นฐานที่สุดในการรันโค้ด Python คือ รันทีละบรรทัดภายใน Python interpreter ซึ่งก่อนอื่นผู้อ่านจะต้องติดตั้งภาษา Python ก่อน (ดูบทก่อนหน้า) จากนั้นผู้อ่านสามารถเริ่ม Python interpreter ด้วยการพิมพ์คำว่า `python` ที่พรอมต์คำสั่ง (มองหาโปรแกรม Terminal ใน Mac OS X และ Unix/Linux หรือโปรแกรม Command Prompt ใน Windows) ดังนี้

```
$ python
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7...
Type "help", "copyright", "credits" or "license" for more...
>>>
```

ในขณะที่ interpreter กำลังทำงานอยู่ ผู้อ่านสามารถพิมพ์และเอ็กซิคิวท์โค้ดได้ ดังแสดงในตัวอย่างต่อไปนี้ ซึ่งเป็นตัวอย่างที่ใช้ interpreter เสมือนกับเป็นเครื่องคิดเลขง่าย ๆ ที่ทำการคำนวณและกำหนดค่าให้กับตัวแปร

```
>>> 1 + 1
2
>>> x = 5
>>> x * 3
15
```

interpreter ช่วยเราเขียนและทดสอบโค้ดสั้น ๆ ได้ง่ายมาก

The IPython interpreter

ถ้าหากผู้อ่านใช้เวลามากพอสมควรกับ Python interpreter ผู้อ่านจะพบว่ามันขาดคุณสมบัติหลายประการของ interactive development environment แบบเต็มรูปแบบ มี interpreter ทางเลือกที่เรียกว่า IPython

(สำหรับ Interactive Python) ซึ่งมาพร้อมกับ Anaconda ดิสทริบิวชั่น interpreter นี้มีขีดความสามารถเพิ่มเติมจาก Python interpreter พื้นฐานเป็นอย่างมาก เราสามารถเริ่ม interpreter นี้ ได้โดยพิมพ์ `ipython` ที่พรอมต์คำสั่งดังนี้

```
$ ipython
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7...
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra...

In [1]:
```

ความแตกต่างที่สำคัญที่สามารถมองเห็นได้ทันทีระหว่าง Python interpreter และ IPython คือพรอมต์คำสั่ง ซึ่ง Python interpreter ใช้ >>> โดยดีฟอลต์ ในขณะที่ IPython ใช้คำสั่งที่มีหมายเลขลำดับระบุอยู่ (เช่น `In[1]:`) อย่างไรก็ตาม เรายังสามารถรันโค้ดทีละบรรทัดใน IPython ได้เช่นเดียวกันกับที่เราเคยทำมาก่อนหน้านี้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [1]: 1 + 1
Out[1]: 2

In [2]: x = 5

In [3]: x * 3
Out[3]: 15
```

ให้ผู้อ่านสังเกตด้วยว่า เอาต์พุตของแต่ละคำสั่งนั้นจะมีหมายเลขระบุลำดับของคำสั่งนั้นอยู่ในทำนองเดียวกับอินพุต IPython มีฟีเจอร์ต่าง ๆ ที่เป็นประโยชน์มากมาย ซึ่งผู้อ่านสามารถศึกษาเพิ่มเติมได้โดยดูจากคำแนะนำในบทเรื่อง Resources for Further Learning

Self-contained Python scripts

ถึงแม้ว่าการรันโค้ด Python ทีละบรรทัดจะมีประโยชน์ แต่สำหรับกรณีที่โปรแกรมมีความซับซ้อนมากขึ้น จะเป็นการสะดวกกว่า หากเราจะทำการบันทึกโปรแกรมลงในไฟล์และเอ็กซิคิวท์โค้ดทั้งหมดรวดเดียวจนจบ ตามธรรมเนียมปฏิบัติแล้ว เราจะบันทึก Python สคริปต์ปรแกรมลงในไฟล์ที่มีนามสกุล `.py`

ให้เรามาลองสร้างสคริปต์ที่เรียกว่า `test.py` ซึ่งประกอบด้วยโค้ดดังนี้

```
# file: test.py
print("Running test.py")
x = 5
print("Result is", 3 * x)
```

ในการเรียกรันไฟล์นี้ เราต้องตรวจสอบให้แน่ใจก่อนว่า ไฟล์นี้อยู่ในไดเร็กทอรีปัจจุบัน จากนั้นจึงพิมพ์ `python` *`filename`* ที่พรอมต์คำสั่ง ดังนี้

```
$ python test.py
Running test.py
Result is 15
```

สำหรับโปรแกรมที่ซับซ้อนมากขึ้น การสร้างสคริปต์ไฟล์อย่างในตัวอย่างนี้เป็นสิ่งที่จำเป็น

The Jupyter notebook

Jupyter notebook เป็นการผสมผสานกันระหว่าง interactive terminal และ self-contained สคริปต์ Jupyter notebook เป็นรูปแบบเอกสารที่อนุญาตให้เอ็กซิคิวท์โค้ดและใส่ข้อความที่มีการจัดฟอร์แมตและรูปภาพได้ นอกจากนี้ยังมี interactive ฟีเจอร์รวมอยู่ด้ว ถึงแม้ว่าแรกเริ่ม Jupyter notebook จะใช้งานได้กับภาษา Python เพียงภาษาเดียวเท่านั้น แต่ในปัจจุบันนี้ เราสามารถนำไปใช้กับภาษาโปรแกรมมิ่งอื่น ๆ จำนวนมากได้ และในตอนนี้ Jupyter notebook เป็นส่วนหลักส่วนหนึ่งของโครงการ Jupyter (Jupyter Project) Jupyter notebook มีประโยชน์ทั้งในด้านที่เป็นสภาพแวดล้อมสำหรับการพัฒนาโปรแกรมและในด้านที่เป็นวิธีการสำหรับแบ่งปันหรือแชร์งานที่สามารถประกอบด้วยโค้ด ข้อมูล ข้อความและรูปภาพได้

A Quick Tour of Python Language Syntax

แรกเริ่มภาษา Python ได้ถูกพัฒนาขึ้นมาเพื่อเป็นภาษาสำหรับใช้ในการสอน แต่ด้วยความที่ตัวภาษาเองนั้นใช้งานได้ง่ายและมีรูปแบบไวยากรณ์ที่เรียบง่าย ทำให้ภาษา Python เป็นที่ยอมรับและใช้งานกันทั้งในกลุ่มผู้เริ่มหัดเขียนโปรแกรมและกลุ่มผู้เชี่ยวชาญการเขียนโปรแกรม ด้วยความเรียบง่ายของไวยากรณ์ของภาษา Python ทำให้ผู้เขียนโปรแกรมบางคนกล่าวถึงภาษา Python ว่าเป็น executable pseudocode ซึ่งสอดคล้องกับประสบการณ์ของตัวผู้เขียนเอง ที่บ่อยครั้งพบว่า การอ่านและทำความเข้าใจโปรแกรมที่เขียนด้วยภาษา Python นั้น จะง่ายกว่าโปรแกรมที่ใช้แก้ปัญหาแบบเดียวกันแต่เขียนด้วยภาษาอื่น อย่างเช่น ภาษา C เป็นต้น ในบทนี้ผู้เขียนจะอธิบายฟีเจอร์ที่สำคัญของไวยากรณ์ของภาษา Python

ไวยากรณ์ หมายถึง โครงสร้างของภาษา (เช่น โปรแกรมที่เขียนอย่างถูกต้องจะต้องประกอบไปด้วยอะไรบ้าง) ในบทนี้ผู้เขียนจะยังไม่อธิบายเกี่ยวกับ sematic ซึ่งหมายถึง ความหมายของคำและสัญลักษณ์ที่ใช้ในไวยากรณ์

ให้เรามาพิจารณาโค้ดตัวอย่างข้างล่างนี้

```
In [1]: # set the midpoint
        midpoint = 5

        # make two empty lists
        lower = []; upper = []

        # split the numbers into lower and upper
        for i in range(10):
            if (i < midpoint):
                lower.append(i)
            else:
                upper.append(i)

        print("lower:", lower)
        print("upper:", upper)
lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

ถึงแม้ว่าโค้ดตัวอย่างนี้จะค่อนข้างง่าย แต่ก็แสดงให้เห็นถึงลักษณะสำคัญทางไวยากรณ์ของภาษา Python ซึ่งผู้เขียนจะใช้ประกอบการอธิบายต่อไป

Comments Are Marked by

โค้ดตัวอย่างนี้เริ่มด้วยคอมเมนต์ (comment):

```
# set the midpoint
```

เราเขียนคอมเมนต์โดยใช้เครื่องหมาย # และอะไรก็ตามที่อยู่ตามหลังเครื่องหมายนี้ interpreter จะมองข้ามไปไม่นำมาพิจารณา ดังนั้นเราสามารถเขียนคอมเมนต์แยกเดี่ยว ๆ อย่างโค้ดตัวอย่างข้างบน หรือเขียนข้างหลังคำสั่ง (statement) ในภาษา Python ก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
x += 2  # shorthand for x = x + 2
```

Python ไม่มีไวยากรณ์สำหรับเขียนคอมเมนต์ที่มีเนื้อหาครอบคลุมหลายบรรทัดอย่างเช่น /*…*/ ในภาษา C และ C++ แต่เราสามารถเอา multiline string มาประยุกต์ใช้แทนได้ (multiline string จะอยู่ในบทเรื่อง String Manipulation and Regular Expressions)

End-of-Line Terminates a Statement

บรรทัดถัดไปในโค้ดตัวอย่างเป็น

```
midpoint = 5
```

ซึ่งเป็นคำสั่งที่ทำการสร้างตัวแปรชื่อ midpoint และให้ค่า 5 แก่ตัวแปรนี้ ให้ผู้อ่านสังเกตด้วยว่า เราสามารถจบคำสั่งหนึ่งคำสั่งในภาษา Python ด้วยการขึ้นบรรทัดใหม่ ในขณะที่คำสั่งอย่างในภาษา C และ C++ จะต้องจบด้วยการเขียนเครื่องหมายอัฒภาค(;) ต่อท้ายคำสั่ง

ถ้าเราอยากเขียนคำสั่งหนึ่งคำสั่งแต่ใช้หลายบรรทัด ในกรณีนี้เราต้องใช้เครื่องหมาย \ เพื่อบอกให้รู้ว่าคำสั่งของเรายังไม่จบ เช่น

```
In [2]: x = 1 + 2 + 3 + 4 +\
        5 + 6 + 7 + 8
```

หรือเราอาจจะใช้วงเล็บ ()แทนเครื่องหมาย \ ก็ได้

```
In [3]: x = (1 + 2 + 3 + 4 +
        5 + 6 + 7 + 8)
```

คู่มือสไตล์การเขียน Python (Python style guide) ส่วนใหญ่จะแนะนำให้ใช้วงเล็บมากกว่าเครื่องหมาย \

Semicolon Can Optionally Terminate a Statement

บางครั้ง การเขียนคำสั่งหลาย ๆ คำสั่งภายในบรรทัดเดียวกันก็มีประโยชน์ ซึ่งจะเห็นได้จากโค้ดตัวอย่างในบรรทัดถัดไป ดังนี้

```
lower = []; upper = []
```

เราสามารถใช้เครื่องหมายอัฒภาค (;) ในการคั่นระหว่างคำสั่งที่อยู่ภายในบรรทัดเดียวกันได้ ซึ่งโค้ดตัวอย่างนี้ก็จะมีความหมายเหมือนกับการเขียนโค้ดแบบนี้

```
lower = []
upper = []
```

คู่มือสไตล์การเขียน Python ส่วนใหญ่จะไม่แนะให้เขียนหลาย ๆ คำสั่งในบรรทัดเดียวกัน ถึงแม้ว่าในบางครั้งการเขียนแบบนี้จะทำให้เขียนโค้ดได้กระชับกว่าก็ตาม

Indentation: Whitespace Matters!

ถัดไป เราจะมาดูเรื่องโค้ดบล็อก (code block):

```
for i in range(10):
    if i < midpoint:
        lower.append(i)
    else:
        upper.append(i)
```

โค้ดตัวอย่างนี้เป็นคำสั่งควบคุม (control flow statement) ประกอบด้วยคำสั่งลูป (loop statement) และคำสั่งเงื่อนไข (conditional statement) ซึ่งจะอธิบายในภายหลัง ณ ตอนนี้ ผู้เขียนอยากให้ผู้อ่านดูฟีเจอร์ที่เป็นที่ถกเถียงกันอย่างมากในภาษา Python ซึ่งก็คือ การใช้ช่องว่าง (whitespace) ในการย่อหน้า (indent)โค้ด

โค้ดบล็อก คือ คำสั่งหลาย ๆ คำสั่งที่ถูกจัดรวมเข้าด้วยกันเป็นกลุ่มเดียวกัน ยกตัวอย่างเช่น ในภาษา C เราจะใช้เครื่องหมายปีกกา ({}) ในการแสดงโค้ดบล็อก ดังนี้

```
// C code
for(int i=0; i<100; i++)
   {
      // curly braces indicate code block
      total += i;
   }
```

ในขณะที่ในภาษา Python เราจะใช้การย่อหน้าในการแสดงโค้ดบล็อก ดังนี้

```
for i in range(100):
    # indentation indicates code block
    total += i
```

โดยที่โค้ดบล็อกจะต้องถูกนำหน้าด้วยเครื่องหมายทวิภาค (:) ในบรรทัดก่อนหน้าเสมอ

การใช้การย่อหน้าช่วยบังคับให้โค้ดในภาษา Python อ่านง่ายและเป็นรูปแบบเดียวกัน แต่อาจจะสร้างความสับสนให้แก่ผู้ที่ไม่เคยใช้ภาษา Python มาก่อน ยกตัวอย่างเช่น โค้ดตัวอย่างทางด้านซ้ายและขวาดังต่อไปนี้จะให้ผลลัพธ์ที่แตกต่างกัน

```
>>> if x < 4:          >>> if x < 4:
...     y = x * 2      ...     y = x * 2
...     print(x)       ... print(x)
```

ในโค้ดตัวอย่างทางด้านซ้าย `print(x)` เป็นส่วนหนึ่งของโค้ดบล็อก และจะถูกเอ็กซิคิวท์ก็ต่อเมื่อ `x` มีค่าน้อยกว่า 4 เท่านั้น ในขณะที่ในโค้ดตัวอย่างทางด้านขวา `print(x)` อยู่ข้างนอกโค้ดบล็อก และจะถูกเอ็กซิคิวท์เสมอ ไม่ว่า `x` จะมีค่าเป็นอะไรก็ตาม

สุดท้ายนี้ ผู้อ่านควรรู้ว่า จำนวนช่องว่างที่ใช้ในการย่อหน้าโค้ดบล็อกนั้น จะมีจำนวนเท่าไหร่ก็ได้แล้วแต่ผู้เขียนโปรแกรม แต่จะต้องย่อหน้าด้วยจำนวนช่องว่างที่เท่ากันตลอดทั้งโปรแกรม อย่างไรก็ตาม คู่มือสไตล์การเขียน Python ส่วนใหญ่จะแนะนำให้ใช้ช่องว่างจำนวน 4 ช่อง ซึ่งหนังสือเล่มนี้ก็จะยึดถือตามนี้ นอกจากนี้ text editor ส่วนใหญ่ อย่างเช่น Emacs และ Vim จะมี Python mode ที่จะทำการย่อหน้าโค้ดให้ 4 ช่องโดยอัตโนมัติ

Whitespace *Within* Lines Does Not Matter

ถึงแม้ว่าช่องว่างที่ใช้ในการย่อหน้าโค้ดจะมีความหมายในภาษา Python แต่ช่องว่างที่อยู่ภายในโค้ด จะไม่มีความหมายใด ๆ เป็นพิเศษ ยกตัวอย่างเช่น โค้ดทั้งสามกรณีดังต่อไปนี้ จะให้ผลลัพธ์เหมือนกัน

```
In [4]: x=1+2
        x = 1 + 2
        x           =       1    +              2
```

การใช้ช่องว่างภายในโค้ดอย่างไม่เหมาะสม จะทำให้โค้ดของเราอ่านยาก ในขณะที่ถ้าเราใช้ช่องว่างอย่างมีประสิทธิภาพ จะทำให้โค้ดของเราอ่านเข้าใจได้ง่ายขึ้น โดยเฉพาะอย่างยิ่งในกรณีที่มีโอเปอเรเตอร์ (operator) อยู่ติดกัน ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นตัวอย่างการหาค่าของจำนวนที่มีเลขยกกำลังเป็นค่าลบ ซึ่งถ้าเราลองเปรียบเทียบโค้ดตัวอย่างแรก ซึ่งเป็นดังนี้

```
x=10**-2
```

กับโค้ดตัวอย่างที่สอง ซึ่งเป็นดังนี้

```
x = 10 ** -2
```

ผู้เขียนพบว่าโค้ดตัวอย่างที่สองสามารถอ่านได้เข้าใจง่ายกว่า ซึ่งคู่มือสไตล์การเขียน Python ส่วนใหญ่ก็แนะนำว่าควรมีช่องว่างหนึ่งช่องรอบ ๆ โอเปอเรเตอร์ไบนารี (binary operator) แต่ไม่ควรมีช่องว่างรอบ ๆ โอเปอเรเตอร์ยูนารี (unary operator) เราจะเรียนโอเปอเรเตอร์ของภาษา Python ในบทเรื่อง Basic Python Semantics: Operators

Parentheses Are for Grouping or Calling

ตัวอย่างต่อไปนี้แสดงให้เห็นการใช้วงเล็บใน 2 รูปแบบ โดยที่ตัวอย่างแรกแสดงการใช้วงเล็บในการจัดกลุ่มการดำเนินการทางคณิตศาสตร์ ดังนี้

```
In [5]: 2 * (3 + 4)
Out [5]: 14
```

ในขณะที่ตัวอย่างที่สองแสดงการใช้วงเล็บในการเรียกฟังก์ชั่น (function) ซึ่งในตัวอย่างนี้เราเรียกฟังก์ชั่น `print()` พร้อมทั้งส่งอากิวเมนต์ (argument) ไปภายในวงเล็บ ดังนี้

```
In [6]: print('first value:', 1)
first value: 1
In [7]: print('second value:', 2)
second value: 2
```

ฟังก์ชั่นบางตัวสามารถเรียกใช้ได้โดยที่ไม่ต้องมีอากิวเมนต์ ซึ่งในกรณีนี้เรายังคงต้องมีวงเล็บเปิดและปิดเพื่อระบุว่าเป็นการเรียกฟังก์ชั่น ตัวอย่างของฟังก์ชั่นแบบนี้ ได้แก่ เมทธอด (method) `sort()` ของลิสต์ (list) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [8]: L = [4,2,3,1]
        L.sort()
        print(L)

        [1, 2, 3, 4]
```

วงเล็บ `()` ที่ตามหลัง `sort` เป็นตัวบ่งชี้ว่าฟังก์ชั่นควรจะถูกเอ็กซิคิวท์ และจำเป็นต้องมีเสมอ ถึงแม้ว่าจะไม่มีอากิวเมนต์ก็ตาม

A Note on the print() Function

ฟังก์ชั่น `print()` เป็นหนึ่งในโครงสร้างที่มีการเปลี่ยนแปลงระหว่าง Python เวอร์ชั่น 2 และเวอร์ชั่น 3 ใน Python เวอร์ชั่น 2 `print` จะทำงานแบบคำสั่ง ซึ่งเราสามารถเขียนได้ดังนี้

```
# Python 2 only!
>> print "first value:", 1
first value: 1
```

ด้วยเหตุผลหลาย ๆ อย่าง ใน Python เวอร์ชั่น 3 `print` จะทำงานแบบฟังก์ชั่น ซึ่งเราสามารถเขียนได้ดังนี้

```
# Python 3 only!
>>> print("first value:", 1)
first value: 1
```

`print` เป็นหนึ่งในโครงสร้างหลาย ๆ ตัวที่ไม่ backward compatible ระหว่าง Python เวอร์ชั่น 2 กับเวอร์ชั่น 3

Finishing Up and Learning More

ในบทนี้ผู้เขียนได้ทำการสำรวจฟีเจอร์ที่สำคัญของไวยากรณ์ในภาษา Python อย่างสังเขป จุดประสงค์ก็เพื่อให้ผู้อ่านมีกรอบที่ดีสำหรับใช้ในการอ้างอิงเวลาอ่านโค้ดในบทถัด ๆ ไป นอกจากนี้ในบทนี้ยังได้พูดถึงคู่มือสไตล์การเขียน Python หลายครั้ง ซึ่งสามารถช่วยให้เราและเพื่อนร่วมทีมเขียนโค้ดในรูปแบบที่สอดคล้องกัน คู่มือสไตล์การเขียน Python ที่ใช้กันอย่างแพร่หลายนั้นเรียกว่า PEP8 และสามารถหาอ่านได้ที่ https://www.python.org/dev/peps/pep-0008/ คำแนะนำในคู่มือนี้มาจากภูมิปัญญาของผู้เชี่ยวชาญจำนวนมาก ส่วนใหญ่จะอิงมาจากประสบการณ์ของผู้เชี่ยวชาญ ซึ่งจะช่วยให้เราหลีกเลี่ยงการเขียนโค้ดที่มีบัก (bug) ได้

Basic Python Semantics: Variables and Objects

บทนี้จะเป็นบทแรกที่ผู้เขียนจะเริ่มอธิบายเกี่ยวกับความหมายของคำสั่งในภาษา Python โดยบทนี้จะเน้นเรื่องความหมายของตัวแปรและออบเจ็คต์ (object)

Python Variables Are Pointers

การให้ค่าแก่ตัวแปรในภาษา Python นั้น ทำได้ง่าย ๆ โดยการเขียนตัวแปรไว้ทางด้านซ้าย ตามด้วยเครื่องหมายเท่ากับ (=) จากนั้นตามด้วยข้อมูลหรือออบเจ็คต์ที่เราต้องการให้ตัวแปรนั้นอ้างอิงถึง ดังนี้

```
# assign 4 to the variable x
x = 4
```

อย่างไรก็ตาม ถ้าผู้อ่านไม่ได้มีความเข้าใจที่ถูกต้องเกี่ยวกับการให้ค่าแก่ตัวแปร สุดท้ายผู้อ่านจะสับสนเกี่ยวกับการทำงานของภาษา Python

ในภาษาโปรแกรมมิ่งหลาย ๆ ภาษา ตัวแปรจะเปรียบเสมือนภาชนะ (container) หรือถัง (bucket) ที่ใช้สำหรับเก็บข้อมูล ยกตัวอย่างเช่น ในภาษา C ถ้าเราเขียนโค้ดดังต่อไปนี้

```
// C code
int x = 4;
```

จะหมายความว่า เรานิยามภาชนะสำหรับเก็บข้อมูลในหน่วยความจำหลักที่มีชื่อว่า x และเราจะใส่ข้อมูลที่มีค่าเป็น 4 เข้าไปในภาชนะนี้ ซึ่งจะแตกต่างจากภาษา Python ในภาษา Python เราจะไม่ใช้ตัวแปรในการเก็บข้อมูลโดยตรง แต่จะเก็บที่อยู่ของข้อมูลแทน ดังนั้นตัวแปรในภาษา Python จึงเปรียบเสมือนพอยน์เตอร์ (pointer) ที่ชี้ไปที่ข้อมูล ทำให้เราสามารถหาข้อมูลนั้นและนำเอามาใช้งานได้ ดังนั้นในภาษา Python ถ้าเราเขียนโค้ดดังต่อไปนี้

```
x = 4
```

จะหมายความว่า เรานิยามพอยน์เตอร์ที่มีชื่อว่า x ซึ่งใช้ในการอ้างอิงข้อมูลที่มีค่าเป็น 4 ดังนั้นในภาษา Python เราจึงไม่จำเป็นต้องประกาศ (declare) ตัวแปร ก่อนที่จะใช้งานมัน เพราะว่าการประกาศตัวแปร คือ การบอกให้รู้ว่า เราจะใช้ตัวแปรนั้น ๆ เก็บข้อมูลชนิดใด ซึ่งตัวแปรในภาษา Python ไม่ได้เก็บข้อมูลโดยตรง เพียงแต่ชี้ไปที่ที่ข้อมูลนั้นอยู่ นอกจากนี้ ตัวแปรไม่จำเป็นต้องชี้ไปที่ข้อมูลชนิดเดียวกันเสมอ เราสามารถใช้ตัวแปรหนึ่ง ๆ ในการชี้ไปที่ข้อมูลที่ต่างชนิดกันก็ได้ ดังแสดงในตัวอย่างต่อไปนี้ ซึ่งเราจะเรียกคุณสมบัติดังกล่าวนี้ว่า dynamically typed

```
In [1]: x = 1          # x is an integer
        x = 'hello'    # now x is a string
        x = [1, 2, 3] # now x is a list
```

ในทางตรงกันข้าม ในภาษาที่เป็นแบบ statically typed เราจะต้องประกาศตัวแปรก่อนใช้งาน อย่างในตัวอย่างโค้ดภาษา C ข้างล่างนี้

```
int x = 4;
```

ผู้ที่เคยเขียนโปรแกรมโดยใช้ภาษาที่เป็นแบบ statically typed อย่างภาษา C อาจจะรู้สึกเสียดายคุณสมบัติ type-safety ที่มาพร้อมกับการประกาศตัวแปร แต่คุณสมบัติ dynamically typed ของภาษา Python นั้นเป็นส่วนหนึ่งที่ช่วยให้เราสามารถเขียนโปรแกรมได้เร็วและอ่านเข้าใจได้ง่าย

ข้อควรระวังเกี่ยวกับตัวแปรที่เป็นพอยน์เตอร์ คือ ถ้าเรามีตัวแปร 2 ตัว ชี้ไปที่ออบเจ็คต์ตัวเดียวกัน และออบเจ็คต์นั้นเป็นแบบ mutable (หมายความว่า ค่าของออบเจ็คต์นั้นเปลี่ยนแปลงได้) เวลาเราเปลี่ยนค่าของออบเจ็คต์โดยใช้ตัวแปรตัวใดตัวหนึ่ง มันจะส่งผลให้เราเห็นค่าที่เปลี่ยนนั้นด้วย เมื่อเราอ้างอิงถึงออบเจ็คต์ตัวนั้นผ่านตัวแปรอีกตัวหนึ่ง ดังแสดงในตัวอย่างข้างล่างไปนี้ ซึ่งเป็นตัวอย่างการสร้างและเปลี่ยนแปลงค่าของลิสต์

```
In [2]: x = [1, 2, 3]
        y = x
```

ในตัวอย่างนี้ เราสร้างตัวแปร x และ y ซึ่งชี้ไปที่ลิสต์เดียวกันทั้งคู่ จากนั้นเราเปลี่ยนค่าของลิสต์ผ่านตัวแปร x ซึ่งจะส่งผลให้เราเห็นค่าที่เปลี่ยนนั้นผ่านตัวแปร y ด้วย ดังนี้

```
In [3]: print(y)

[1, 2, 3]

In [4]: x.append(4) # append 4 to the list pointed to by x
        print(y) # y's list is modified as well!

[1, 2, 3, 4]
```

ดังนั้นถ้าเรามีความเข้าใจที่ไม่ถูกต้องเกี่ยวกับตัวแปรในภาษา Python คือ ถ้าเราคิดว่ามันเป็นเสมือนภาชนะสำหรับเก็บข้อมูล เราก็จะงงกับผลลัพธ์ของโค้ดตัวอย่างนี้ แต่ถ้าเราเข้าใจว่าตัวแปรในภาษา Python เป็นเสมือนพอยน์เตอร์ เราก็จะไม่ประหลาดใจกับผลลัพธ์ของโค้ดนี้เลย

ให้ผู้อ่านสังเกตด้วยว่า ถ้าเราใช้ = ในการให้ค่าอื่นแก่ตัวแปร x มันก็ไม่ได้มีผลอะไรกับตัวแปร y เพราะการให้ค่าแก่ตัวแปรเป็นเพียงการเปลี่ยนออบเจ็คต์ที่ตัวแปรนั้นชี้อยู่เท่านั้นเอง

```
In [5]: x = 'something else'
        print(y)  # y is unchanged

[1, 2, 3, 4]
```

การที่ตัวแปรเป็นพอยน์เตอร์นั้น ผู้อ่านอาจจะคิดว่ามันน่าจะทำให้การคำนวณทางคณิตศาสตร์ในภาษา Python ซับซ้อนขึ้น แต่ในความเป็นจริงแล้ว ภาษา Python ได้ถูกออกแบบมาในลักษณะที่แนวคิดพอยน์เตอร์นี้ไม่ได้สร้างปัญหาดังกล่าวเลย ดังแสดงในตัวอย่างข้างล่างนี้ ที่เป็นเช่นนี้เพราะว่า ตัวเลข สตริง (string) และข้อมูลพื้นฐานอื่น ๆ ในภาษา Python นั้น จะเป็น immutable หมายความว่า เราไม่สามารถเปลี่ยนค่าของมันได้ ดังนั้นเราจึงเปลี่ยนได้แค่ออบเจ็คต์ที่ตัวแปรชี้อยู่เท่านั้น

```
In [6]: x = 10
        y = x
        x += 5  # add 5 to x's value, and assign it to x
        print("x =", x)
        print("y =", y)

x = 15
y = 10
```

ในบรรทัดที่เราเขียน `x += 5` นั้น เราไม่ได้เปลี่ยนค่าของออบเจ็คต์ `10` ที่ตัวแปร `x` ชี้อยู่ให้กลายเป็น `15` แต่เราเปลี่ยนออบเจ็คต์ที่ตัวแปร `x` นี้ชี้อยู่ให้เป็นออบเจ็คต์ตัวที่มีค่าเป็น `15` แทน ดังนั้นค่าของตัวแปร `y` ก็ยังคงเป็นเหมือนเดิม คือ ยังชี้ไปที่ออบเจ็คต์ `10` เหมือนเดิม

Everything Is an Object

ภาษา Python เป็นภาษาเชิงวัตถุ (object-oriented programming language) และในภาษา Python ทุกอย่างจะเป็นวัตถุหรือออบเจ็คต์

ในหัวข้อก่อนหน้านี้ เราเห็นแล้วว่า ตัวแปรเป็นเพียงพอยน์เตอร์ และตัวแปรก็ไม่ได้มีชนิดของข้อมูล (type) มาเกี่ยวข้อง ทำให้บางคนคิดว่าภาษา Python เป็นภาษาแบบ type-free language ซึ่งไม่ถูกต้อง ดังจะเห็นได้จากตัวอย่างต่อไปนี้

```
In [7]:  x = 4
         type(x)

Out [7]: int

In [8]:  x = 'hello'
         type(x)

Out [8]: str

In [9]:  x = 3.14159
         type(x)

Out [9]: float
```

เราจะเห็นได้ว่าภาษา Python มีชนิดของข้อมูล แต่ชนิดของข้อมูลไม่ได้ผูกอยู่กับตัวแปร แต่ผูกอยู่กับออบเจ็คต์ที่ตัวแปรชี้อยู่

ออบเจ็คต์ในภาษา Python จะประกอบด้วยแอตทริบิวต์ (attribute) และเมทธอด (method) และเราสามารถ access แอตทริบิวต์และเมทธอดได้โดยใช้ dot syntax

ยกตัวอย่างเช่น ลิสต์มีใช้เมทธอด `append()` สำหรับใช้เพิ่มสมาชิกเข้าไปในลิสต์และสามารถเรียกใช้งานได้โดยใช้ dot syntax ดังนี้

```
In [10]: L = [1, 2, 3]
         L.append(100)
         print(L)

[1, 2, 3, 100]
```

ในภาษา Python ข้อมูลชนิดพื้นฐานก็เป็นออบเจ็คต์ ดังนั้นมันก็จะมีแอตทริบิวต์และเมทธอดด้วยเหมือนกัน ยกตัวอย่างเช่น ข้อมูลพื้นฐานชนิดตัวเลขจะมีแอตทริบิวต์ real และ imag ซึ่งจะคืนค่าส่วนจริง (real part) และส่วนจินตภาพ (imaginary part) ออกมาให้ ในกรณีที่เรามองตัวเลขนั้นเป็นเลขจำนวนเชิงซ้อน (complex number) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]: x = 4.5
         print(x.real, "+", x.imag, 'i')

4.5 + 0.0 i
```

เมทธอดก็เหมือนกับแอตทริบิวต์ แต่แตกต่างกันตรงที่ว่า เมทธอดเป็นฟังก์ชั่น ซึ่งเราสามารถเรียกได้โดยใช้วงเล็บ ยกตัวอย่างเช่น เลขทศนิยมจะมีเมทธอด `is_interger` ที่ใช้สำหรับตรวจสอบว่าเลขนั้นเป็นเลขจำนวนเต็มหรือไม่ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [12]:  x = 4.5
          x.is_integer()

Out [12]: False

In [13]:  x = 4.0
          x.is_integer()

Out [13]: True
```

ดังที่ได้กล่าวไปในข้างต้นแล้วว่า ทุกอย่างในภาษา Python เป็นออบเจ็คต์ ซึ่งก็หมายความอย่างนั้นจริง ๆ เพราะว่าแม้กระทั่งตัวแอตทริบิวต์หรือตัวเมทธอดเองต่างก็เป็นออบเจ็คต์ที่มีชนิดเป็นของมันเอง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [14]:  type(x.is_integer)

Out [14]: builtin_function_or_method
```

เราจะพบว่าการที่เลือกออกแบบให้ทุกอย่างในภาษา Python เป็นออบเจ็คต์นั้น ทำให้สามารถสร้างโครงสร้างทางภาษาที่เอื้ออำนวยต่อการเขียนโปรแกรมเป็นอย่างมาก

Basic Python Semantics: Operators

ในบทที่แล้วเราได้ดูความหมายของตัวแปรและออบเจ็คต์ในภาษา Python ไปแล้ว ในบทนี้เราจะมาดูความหมายของโอเปอเรเตอร์ประเภทต่างในภาษา Python

Arithmetic Operations

ในภาษา Python จะมีโอเปอเรเตอร์ไบนารีทางคณิตศาสตร์ทั้งหมด 7 ตัว ดังแสดงในตารางข้างล่างนี้ ซึ่งจะมีโอเปอรเตอร์สองตัวที่สามารถเป็นโอเปอเรเตอร์ยูนารีได้ด้วย

Operator	Name	Description
a + b	Addition	Sum of a and b
a - b	Subtraction	Difference of a and b
a * b	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
a // b	Floor division	Quotient of a and b, removing fractional parts
a % b	Modulus	Remainder after division of a by b
a ** b	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

โอเปอเรเตอร์ต่าง ๆ เหล่านี้สามารถนำมาใช้ร่วมกันได้โดยใช้วงเล็บในการจัดกลุ่ม ดังแสดงในตัวอย่างข้างล่างนี้

```
In [1]:  # addition, subtraction, multiplication
         (4 + 8) * (6.5 - 3)

Out [1]: 42.0
```

การทำ floor division จะเหมือนกันกับ true division ยกเว้นเศษทศนิยมจะถูกตัดทิ้งไป ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]: # True division
        print(11 / 2)

5.5

In [3]: # Floor division
        print(11 // 2)

5
```

โอเปอเรเตอร์ `//` สำหรับทำ floor division นั้น ถูกเพิ่มเข้ามาใน Python เวอร์ชั่น 3 ดังนั้นใน Python เวอร์ชั่น 2 จะมีเฉพาะโอเปอเรเตอร์ `/` สำหรับใช้หารตัวเลข ซึ่งข้อควรระวังก็คือ ถ้าใช้โอเปอเรเตอร์ `/` ในการหารเศษและส่วนที่เป็นเลขจำนวนเต็มทั้งคู่ ผลลัพธ์ที่ได้จะเหมือนกับการทำ floor division ใน Python เวอร์ชั่น 3 แต่ถ้าเศษหรือส่วนหรือทั้งเศษและส่วนเป็นเลขทศนิยม ผลลัพธ์ที่ได้จะเหมือนกับการทำ true division ใน Python เวอร์ชั่น 3

สุดท้ายนี้ ผู้เขียนจะกล่าวถึงโอเปอเรเตอร์ตัวที่แปด ซึ่งก็คือ `@` ถ้าเราเขียน `a @ b` ก็จะหมายความว่า ให้หา matrix product ระหว่าง `a` กับ `b` โอเปอเรเตอร์ `@` เป็นโอเปอเรเตอร์ที่เพิ่มเข้ามาใน Python เวอร์ชั่น 3.5 สำหรับใช้ในแพ็คเกจต่าง ๆ ที่เกี่ยวข้องกับพีชคณิตเชิงเส้น (linear algebra)

Bitwise Operations

นอกจากโอเปอเรเตอร์ทางคณิตศาสตร์ทั่ว ๆ ไปแล้ว ภาษา Python ยังมีโอเปอเรเตอร์ที่เรียกว่า bitwise ที่สามารถนำมาใช้กับตัวเลขได้ด้วย ซึ่งอาจจะไม่ได้ถูกนำมาใช้งานบ่อยเท่ากับ โอเปอเรเตอร์ทางคณิตศาสตร์ แต่อย่างน้อยเราก็ควรจะรู้ไว้ว่ามีโอเปอเรเตอร์แบบนี้ด้วย ซึ่งสรุปไว้ในตารางข้างล่างนี้

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both a and b
`a | b`	Bitwise OR	Bits defined in a or b or both
`a ^ b`	Bitwise XOR	Bits defined in a or b but not both
`a << b`	Bit shift left	Shift bits of a left by b units
`a >> b`	Bit shift right	Shift bits of a right by b units
`~a`	Bitwise NOT	Bitwise negation of a

เราจะเข้าใจการทำงานของโอเปอเรเตอร์ bitwise ถ้าตัวเลขที่นำมาใช้กับโอเปอเรเตอร์นี้อยู่ในรูปเลขฐานสอง เราสามารถทำให้ตัวเลขอยู่ในรูปเลขฐานสองได้โดยใช้ฟังก์ชั่น `bin` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]:  bin(10)
Out [4]: '0b1010'
```

ผลลัพธ์ที่ได้จะมี `0b` นำหน้า เพื่อบอกให้รู้ว่าเป็นเลขฐานสอง ส่วนตัวเลขที่เหลือ (`1010`) ใช้แสดงค่า `10` ซึ่งเราสามารถคำนวณหาได้ดังนี้

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

ในทำนองเดียวกัน เราสามารถแสดงเลขฐานสองของ `4` ได้ดังนี้

```
In [5]:  bin(4)
Out [5]: '0b100'
```

เมื่อเราใช้โอเปอเรเตอร์ bitwise OR เราสามารถหาตัวเลขที่เกิดจากการรวมบิตของ `4` และ `10` ได้ดังนี้

```
In [6]:  4 | 10
Out [6]: 14
In [7]:  bin(4 | 10)
Out [7]: '0b1110'
```

ถึงแม้โอเปอเรเตอร์ bitwise เหล่านี้ จะไม่ได้มีประโยชน์ในการใช้งานในทันทีทันใดเหมือนกับโอเปอเรเตอร์ทางคณิตศาสตร์ แต่อย่างน้อย เราก็ควรรู้จักโอเปอเรเตอร์เหล่านี้ไว้ โดยเฉพาะอย่างยิ่งผู้อ่านที่มีพื้นฐานการเขียนโปรแกรมที่มาจากภาษาอื่น ซึ่งในบางครั้งจะใช้โอเปอเรเตอร์ XOR (เช่น `a^b`) ในความหมายของเลขยกกำลัง (ซึ่งที่ถูกต้องจะต้องเขียนว่า `a ** b`)

Assignment Operations

ก่อนหน้านี้ เราได้เห็นมาแล้วว่า เราสามารถให้ค่าแก่ต้องแปรได้โดยใช้เครื่องหมาย `=` ซึ่งเรียกว่า assignment operator และเราให้ค่าแก่ตัวแปรก็เพื่อที่จะได้สามารถนำค่านั้นกลับมาใช้งานในภายหลังได้ เช่น

```
In [8]: a = 24
        print(a)
24
```

เราสามารถใช้ตัวแปรร่วมกับโอเปอเรเตอร์ต่าง ๆ ที่ได้กล่าวไปแล้วภายในนิพจน์ (expression) ได้ ยกตัวอย่างเช่น ถ้าเราต้องการบวก `2` เพิ่มเข้าไปที่ค่าที่ตัวแปร `a` ชี้อยู่ เราสามารถเขียนโค้ดได้ดังนี้

```
In [9]:  a + 2
Out [9]: 26
```

ในตัวอย่างนี้ เราอัพเดทค่าที่ตัวแปร `a` ชี้อยู่ ด้วยการใช้โอเปอเรเตอร์บวกและ assignment operator ซึ่งการอัพเดพค่าที่ตัวแปรชี้อยู่นี้ เป็นสิ่งที่กระทำกันบ่อยมากในโปรแกรม ดังนั้นในภาษา Python จึงมี augmented assignment operator ที่ช่วยในการอัพเดพค่าที่ตัวแปรชี้อยู่ ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งไม่ได้จำกัดอยู่แต่การอัพเดพที่ใช้โอเปอเรเตอร์บวกเท่านั้น แต่สำหรับทุกโอเปอเรเตอร์ทางคณิตศาสตร์เลย

```
In [10]: a += 2  # equivalent to a = a + 2
         print(a)
26
```

ตัวอย่างต่อไปนี้เป็นตัวอย่างของ augmented assignment operator ที่คู่กับโอเปอเรเตอร์ที่เราได้เรียนไปก่อนหน้านี้แล้ว

```
a += b   a -= b  a *= b   a /= b
a //= b  a %= b  a **= b  a &= b
a |= b   a ^= b  a <<= b  a >>= b
```

ให้เรามาดูความหมายของ augmented assignment operator เหล่านี้กัน สมมติให้ # แทนโอเปอเรเตอร์ที่อยู่ข้างหน้า = เมื่อเราเขียน a #= b จะมีความหมายเหมือนกับเราเขียน a = a # b ซึ่งจะให้ผลลัพธ์ที่แตกต่างกันระหว่างออบเจ็คต์ที่เป็นแบบ mutable กับ immutable ในกรณีที่ตัวแปร a ชี้ที่ออบเจ็คต์ที่เป็นแบบ mutable หลังจากใช้ augmented assignment operator แล้ว ตัวแปร a ก็ยังคงชี้ที่ออบเจ็คต์ตัวเดิมแต่ค่าของออบเจ็คต์จะเปลี่ยนไป ในขณะที่ถ้าตัวแปร a ชี้ที่ออบเจ็คต์ที่เป็นแบบ immutable หลังจากใช้ augmented assignment operator แล้ว ตัวแปร a จะชี้ที่ออบเจ็คต์ตัวใหม่ที่ถูกสร้างขึ้นมาเพื่อแสดงค่าใหม่

Comparison Operations

โอเปอเรเตอร์อีกประเภทหนึ่งที่มีประโยชน์มาก คือ comparison operator ที่ใช้ในการเปรียบเทียบค่า ดังแสดงในตารางข้างล่างนี้ โอเปอเรเตอร์เหล่านี้จะให้ค่าออกมาเป็น True หรือ False

Operation	Description
a == b	a equal to b
a != b	a not equal to b
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

เราสามารถใช้ comparison operator ร่วมกับโอเปอเรเตอร์ทางคณิตศาสตร์และโอเปอเรเตอร์ bitwise ได้ ยกตัวอย่างเช่น เราสามารถตรวจสอบว่าตัวเลขเป็นเลขคี่หรือไม่ ด้วยการดูว่าผลลัพธ์ที่ได้จากการใช้โอเปอเรเตอร์ % ระหว่างตัวเลขนั้น ๆ กับ 2 มีค่าเท่ากับ 1 หรือไม่ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]:  # 25 is odd
          25 % 2 == 1

Out [11]: True

In [12]:  # 66 is odd
          66 % 2 == 1

Out [12]: False
```

เราสามารถเอา comparison operator หลาย ๆ ตัวมาเชื่อมต่อกันเพื่อสร้างความสัมพันธ์ที่ซับซ้อนได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [13]:  # check if a is between 15 and 30
          a = 25
          15 < a < 30

Out [13]: True
```

ให้เรามาลองดูตัวอย่างที่ค่อนข้างเข้าใจยาก อย่างเช่น

```
In [14]:  -1 == ~0

Out [14]: True
```

ก่อนอื่นให้เรามาทบทวนกันก่อนว่า ~ เป็นโอเปอเรเตอร์ bitwise และถ้าผู้อ่านสงสัยว่าทำไมถึงได้ผลลัพธ์เช่นนี้ ให้ผู้อ่านลองศึกษาเรื่อง two's complement เพิ่มเติม

Boolean Operations

ภาษา Python มีโอเปอเรเตอร์สำหรับรวมค่าตรรกะเข้าด้วยกันโดยใช้หลักการการ "and" "or" และ "not" ซึ่งใช้สัญลักษณ์ and, or, และ not ในการแสดงโอเปอเรเตอร์เหล่านี้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [15]:  x = 4
          (x < 6) and (x > 2)

Out [15]: True

In [16]:  (x > 10) or (x % 2 == 0)

Out [16]: True

In [17]:  not (x < 6)

Out [17]: False
```

ผู้อ่านที่มีความรู้ทางด้านพีชคณิต Boolean อาจจะสังเกตเห็นว่าไม่มีโอเปอเรเตอร์ XOR แต่เราสามารถใช้โอเปอเรเตอร์ตัวอื่น ๆ มาช่วยในการเขียนเพื่อให้มีผลลัพธ์เหมือนโอเปอเรเตอร์ XOR ได้ในหลาย ๆ วิธี หรือเราอาจจะใช้เทคนิคดังต่อไปนี้ก็ได้

```
In [18]:  # (x > 1) xor (x < 10)
          (x > 1) != (x < 10)

Out [18]: False
```

ค่าตรรกะและโอเปอเรเตอร์ต่าง ๆ เหล่านี้จะถูกนำไปใช้ในการสร้างเงื่อนไขในบทเรื่อง Control Flow Statement

บางคนอาจจะสับสับว่าเมื่อไหร่ เราควรจะใช้โอเปอเรเตอร์ตัวไหน ระหว่างโอเปอเรเตอร์ Boolean (and, or และ not) กับโอเปอเรเตอร์ bitwise (&, | และ ~) ซึ่งคำตอบก็อยู่ที่ชื่อของโอเปอเรเตอร์นั่นเอง กล่าวคือ

เราควรใช้โอเปอเรเตอร์ Boolean เมื่อเราต้องการคำนวณหาค่าตรรกะของทั้งคำสั่ง และเราควรใช้โอเปอเรเตอร์ bitwise เมื่อเราต้องการดำเนินการระดับบิต

Identity and Membership Operators

ภาษา Python มีโอเปอเรเตอร์ที่ใช้สำหรับตรวจสอบ identity และ membership ดังแสดงในตางรางข้างล่างนี้ ซึ่งโอเปอเรเตอร์เหล่านี้ จะมีลักษณะเป็นเหมือนคำธรรมดาในภาษาอังกฤษในทำนองเดียวกันกับ and, or และ not

Operator	Description
a is b	True if a and b are identical objects
a is not b	True if a and b are not identical objects
a in b	True if a is a member of b
a not in b	True if a is not a member of b

Identity operators: is and is not

โอเปอเรเตอร์ที่ใช้สำหรับตรวจสอบ identity ได้แก่ is และ is not ซึ่ง identity ของออบเจ็คต์นั้นจะแตกต่าง equality ของออบเจ็คต์ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [19]:  a = [1, 2, 3]
          b = [1, 2, 3]

In [20]:  a == b

Out [20]: True

In [21]:  a is b

Out [21]: False

In [22]:  a is not b

Out [22]: True
```

ถัดไปเป็นตัวอย่างของออบเจ็คต์ที่มี identity เดียวกัน

```
In [23]:  a = [1, 2, 3]
          b = a
          a is b

Out [23]: True
```

ความแตกต่างของตัวอย่างในข้างต้นนั้นคือ ตัวแปร a และ b ในตัวอย่างแรก ชี้ไปที่ออบเจ็คต์คนละตัวกัน ในขณะที่ ตัวแปร a และ b ในตัวอย่างที่สอง ชี้ไปที่ออบเจ็คต์ตัวเดียวกัน โอเปอเรเตอร์ is ใช้ตรวจสอบว่าตัว

แปรสองตัว ชี้ไปที่ออบเจ็คต์ตัวเดียวกันหรือไม่ ไม่ได้ดูที่ค่าของออบเจ็คต์ ซึ่งบ่อยครั้ง ผู้เริ่มหัดเขียนโปรแกรมจะใช้โอเปอเรเตอร์ `is` ทั้งที่ในความเป็นจริงแล้วควรจะใช้โอเปอเรเตอร์ `==`

Membership operators

`in` เป็นโอเปอเรเตอร์ membership ใช้สำหรับการตรวจว่าออบเจ็คต์หนึ่ง ๆ เป็นสมาชิกของออบเจ็คต์อีกตัวหรือไม่ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [24]:  1 in [1, 2, 3]
Out [24]: True
In [25]:  2 not in [1, 2, 3]
Out [25]: False
```

การตรวจสอบเรื่อง membership ในภาษา Python เป็นตัวอย่างหนึ่งที่ทำให้เห็นว่า การเขียนโปรแกรมภาษา Python นั้นเขียนง่ายกว่าภาษาระดับต่ำอย่างเช่น ภาษา C ในภาษา C เราสามารถตรวจสอบเรื่อง membership ได้ด้วยการเขียนโค้ดวนลูปเอง โดยที่แต่ละรอบของการวนลูป เราจะตรวจสอบว่าออบเจ็คต์ที่เราต้องการรู้กับออบเจ็คต์แต่ละตัวในลิสต์มีค่าเท่ากันหรือไม่

Built-In Types: Simple Values

ผู้เขียนเคยกล่าวไว้ในเรื่องตัวแปรและออบเจ็คต์ว่า ออบเจ็คต์ทุกตัวในภาษา Python จะมีชนิดของข้อมูล ซึ่งในบทนี้จะพูดถึงชนิดของข้อมูลที่เป็นแบบ simple type ที่มากับตัวภาษา Python โดยตรง และได้สรุปในตารางข้างล่างนี้ ในที่นี้ผู้เขียนใช้คำว่า “simple type” เพื่อจะได้เปรียบเทียบกับ “compound type” ที่จะอธิบายในบทถัดไป

Type	Example	Description
int	x = 1	Integers (i.e., whole numbers)
float	x = 1.0	Floating-point numbers (i.e., real numbers)
complex	x = 1 + 2j	Complex numbers (i.e., numbers with a real and imaginary part)
bool	x = True	Boolean: True/False values
str	x = 'abc'	String: characters or text
NoneType	x = None	Special object indicating nulls

ผู้เขียนจะอธิบายชนิดของข้อมูลแต่ละตัวในตารางนี้โดยสังเขป

Integers

เลขจำนวนเต็ม (integer) หรือเลขที่ไม่มีจุดทศนิยม เป็นข้อมูลพื้นฐานที่สุดในบรรดาข้อมูลประเภทตัวเลข

```
In [1]:  x = 1
         type(x)
Out [1]: int
```

เลขจำนวนเต็มในภาษา Python มีความซับซ้อนมากกว่าเลขจำนวนเต็มในภาษาอื่น ๆ อย่างเช่น ภาษา C เลขจำนวนเต็มในภาษา C จะเป็นแบบ fixed-precision (จำนวนบิตที่ใช้แสดงตัวเลขจะมีจำนวนคงที่) และจะเกิดปัญหา overflow กับเลขบางจำนวน (โดยเฉพาะเลขที่มีค่าใกล้ 2^{31} และ 2^{63} ซึ่งก็จะขึ้นอยู่กับระบบที่เราใช้) ส่วนเลขจำนวนเต็มในภาษา Python จะเป็น variable-precision (จำนวนบิตที่ใช้แสดงตัวเลขจะมีจำนวนไม่คงที่) ทำให้เวลาคำนวณเลขจำนวนเต็มในภาษา Python จะไม่เกิด overflow ดังในภาษาอื่น ๆ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]:  2 ** 200
Out [2]:
1606938044258990275541962092341162602522202993782792835301376
```

ฟีเจอร์อีกอย่างหนึ่งของเลขจำนวนเต็มในภาษา Python ที่ช่วยอำนวยความสะดวกในการเขียนโปรแกรม คือ เมื่อเราเอาเลขจำนวนเต็มหารด้วยเลขจำนวนเต็ม เราจะได้ผลลัพธ์ที่เป็นเลขที่มีจุดทศนิยมโดยอัตโนมัติ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [3]:  5 / 2
Out [3]: 2.5
```

ข้อควรระวังก็คือ ฟีเจอร์นี้ใช้ได้กับเฉพาะ Python เวอร์ชั่น 3 เท่านั้น ในขณะที่ Python เวอร์ชั่น 2 จะตัดเลขทศนิยมที่ได้จากการหารเลขจำนวนเต็มทิ้งไป ซึ่งก็จะเหมือนกับภาษาที่เป็นแบบ statically typed หลาย ๆ ภาษา เช่น ภาษา C

```
# Python 2 behavior
>>> 5 / 2
2
```

ถ้าเราอยากได้ผลลัพธ์ของการหารเลขจำนวนเต็มเหมือนกันกับ Python เวอร์ชั่น 2 ใน Python เวอร์ชั่น 3 เราต้องใช้โอเปอเรเตอร์ `//` สำหรับทำ floor division แทน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]:  5 // 2
Out [4]: 2
```

สุดท้ายนี้ อยากจะให้ผู้อ่านสังเกตด้วยว่า ใน Python เวอร์ชั่น 2 จะมีเลขจำนวนเต็ม 2 ชนิด คือ `int` และ `long` ในขณะที่ Python เวอร์ชั่น 3 ได้รวมเลขจำนวนเต็มทั้ง 2 ชนิดให้เป็นชนิดเดียว คือ `int`

Floating-Point Numbers

เลขทศนิยม (floating-point number) ใช้แสดงข้อมูลที่เป็นเศษส่วน โดยสามารถเขียนได้ใน 2 รูปแบบ คือ รูปแบบจุดทศนิยมหรือรูปแบบเลขยกกำลัง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [5]: x = 0.000005
        y = 5e-6
        print(x == y)
True
In [6]: x = 1400000.00
        y = 1.4e6
        print(x == y)
True
```

ตัวอักษร `e` หรือ `E` ในรูปแบบเลขยกกำลัง หมายถึงสิบยกกำลัง ดังนั้น `1.4e6` จึงมีความหมายเหมือนกับ $1.4x10^6$

เราสามารถสร้างข้อมูลชนิด `float` จากเลขจำนวนเต็มได้โดยใช้ `float()` ดังแสดงในตัวอย่างต่อไปนี้

```
In [7]:  float(1)
Out [7]: 1.0
```

Floating-point precision

สิ่งหนึ่งที่เราต้องระวังเกี่ยวกับเลขทศนิยม ก็คือ เครื่องคอมพิวเตอร์สามารถเก็บเลขหลังจุดทศนิยมได้จำนวนจำกัด ส่งผลให้เมื่อเราตรวจสอบความเท่ากันของเลขทศนิยม เราอาจจะได้ผลลัพธ์ไม่ตรงกับทางคณิตศาสตร์ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [8]:  0.1 + 0.2 == 0.3
Out [8]: False
```

ปัญหานี้ไม่ได้เกิดขึ้นกับภาษา Python เพียงภาษาเดียวแต่เกิดขึ้นกับทุก ๆ ภาษา เพราะเครื่องคอมพิวเตอร์จะเก็บตัวเลขในรูปเลขฐานสอง และถ้าเลขฐานสองนั้นมีเลขทศนิยมไม่สิ้นสุด ซึ่งคอมพิวเตอร์ไม่สามารถเก็บได้ ก็จะเก็บเป็นเลขทศนิยมจำนวนจำกัดแทน ทำให้เลขทศนิยมบางตัวถูกเก็บเป็นค่าประมาณแทน ซึ่งเราจะเห็นได้จากตัวอย่างต่อไปนี้

```
In [9]: print("0.1 = {0:.17f}".format(0.1))
        print("0.2 = {0:.17f}".format(0.2))
        print("0.3 = {0:.17f}".format(0.3))

0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

โดยปกติ เราจะคุ้นเคยกับการคิดเลขโดยใช้เลขฐานสิบ ดังนั้นตัวเลขเศษส่วนจึงถูกแสดงให้อยู่ในรูปของผลบวกของตัวเลขคูณด้วยหลักที่เป็นเลขยกกำลังฐานสิบ ดังแสดงในตัวอย่างข้างล่างนี้

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

เราสามารถเขียนแสดงตัวเลขเศษส่วนนี้ในรูปเลขฐานสิบได้เป็น `0.125`

แต่เนื่องจากคอมพิวเตอร์จะเก็บตัวเลขในรูปเลขฐานสอง ดังนั้นตัวเลขแต่ละตัวจะถูกทำให้อยู่ในรูปของผลบวกของตัวเลขคูณด้วยหลักที่เป็นเลขยกกำลังฐานสอง ดังแสดงในตัวอย่างข้างล่างนี้

$$1/8 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

เราสามารถเขียนแสดงตัวเลขเศษส่วนนี้ในรูปเลขฐานสองได้เป็น 0.001_2 โดยที่ตัวเลข 2 ที่ห้อยอยู่นั้นเป็นตัวบ่งบอกให้รู้ว่าเป็นเลขฐานสอง ค่า `0.125` == 0.001_2 เป็นตัวเลขที่มีเลขหลังจุดทศนิยมที่มีจำนวนที่สิ้นสุดทั้งในรูปเลขฐานสิบและฐานสอง

ในระบบเลขฐานสิบ ผู้อ่านคงจะคุ้นเคยกับตัวเลขที่มีเลขหลังจุดทศนิยมจำนวนไม่สิ้นสุด อย่างเช่น ตัวเลขที่เกิดจากหาร 1 ด้วย 3 ดังนี้

$1/3 = 0.333333333...$

ในทำนองเดียวกัน ในระบบเลขฐานสอง ก็จะมีตัวเลขที่มีเลขหลังจุดทศนิยมจำนวนไม่สิ้นสุดเหมือนกัน ดังแสดงในตัวอย่างข้างล่างนี้

$1/10 = 0.00011001100110011..._2$

เนื่องจากเครื่องคอมพิวเตอร์ไม่สามารถเก็บเลขทศนิยมที่ไม่สิ้นสุดได้ ดังนั้นในภาษา Python จะปัดเศษทศนิยมทิ้งที่ตำแหน่งที่ 52

ดังนั้นเราควรจำไว้ว่า เลขทศนิยมในเครื่องคอมพิวเตอร์ของเราเป็นค่าประมาณ ดังนั้นเราไม่ควรนำเลขทศนิยมมาเปรียบเทียบเรื่องการเท่ากัน

Complex Numbers

เลขจำนวนเชิงซ้อน (complex number) คือ เลขที่ประกอบด้วยส่วนจริงและส่วนจินตภาพ เราสามารถใช้เลขจำนวนเต็มและเลขจำนวนจริงมาใช้ในการสร้างเลขจำนวนเชิงซ้อนได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [10]:  complex(1, 2)
Out [10]: (1+2j)
```

หรือเราอาจจะสร้างเลขจำนวนเชิงซ้อน ด้วยการใส่ j ต่อท้ายที่นิพจน์ใด ๆ เพื่อหมายถึงส่วนจินตภาพก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]:  1 + 2j
Out [11]: (1+2j)
```

เลขจำนวนเชิงซ้อนมีแอตทริบิวท์และเมทธอดที่น่าสนใจอยู่หลายตัว ดังแสดงในตัวอย่างข้างล่างนี้

```
In [12]:  c = 3 + 4j
In [13]:  c.real  # real part
Out [13]: 3.0
In [14]:  c.imag  # imaginary part
Out [14]: 4.0
```

```
In [15]:  c.conjugate()  # complex conjugate

Out [15]: (3-4j)

In [16]:
abs(c)  # magnitude--that is, sqrt(c.real ** 2 + c.imag ** 2)

Out [16]: 5.0
```

String Type

เราสามารถสร้างสตริงในภาษา Python ด้วยการใช้อัญประกาศเดี่ยว (single quote) หรืออัญประกาศคู่ (double quote) ก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [17]: message = "what do you like?"
         response = 'spam'
```

Python มีฟังก์ชั่นและเมทธอดของสตริงที่มีประโยชน์มากมายอยู่หลายตัว ผู้เขียนจะแนะนำให้รู้จักบางตัว ดังแสดงในตัวอย่างข้างล่างนี้

```
In [18]:  # length of string
          len(response)

Out [18]: 4

In [19]:  # Make uppercase. See also str.lower()
          response.upper()

Out [19]: 'SPAM'

In [20]:  # Capitalize. See also str.title()
          message.capitalize()

Out [20]: 'What do you like?'

In [21]:  # concatenation with +
          message + response

Out [21]: 'what do you like?spam'

In [22]:  # multiplication is multiple concatenation
          5 * response

Out [22]: 'spamspamspamspamspam'

In [23]:  # Access individual characters (zero-based indexing)
          message[0]

Out [23]: 'w'
```

ผู้เขียนจะมีอธิบายเพิ่มเติมเรื่อง indexing ในหัวข้อเรื่องลิสต์

None Type

Python มีชนิดของข้อมูลที่พิเศษที่เรียกว่า NoneType ซึ่งข้อมูลชนิดนี้มีอยู่เพียงตัวเดียวเท่านั้นคือ None ดังแสดงในตัวอย่างข้างล่างนี้

```
In [24]:  type(None)

Out [24]: NoneType
```

ผู้อ่านจะเห็น None ถูกใช้ในหลาย ๆ กรณี แต่กรณีที่จะพบบ่อยที่สุด คือ กรณีที่ถูกใช้เป็นค่าดีฟอลต์ในการคืนค่าของฟังก์ชั่นเมื่อผู้เขียนโปรแกรมไม่ได้เขียนให้ฟังก์ชั่นคืนค่ากลับมาให้ ยกตัวอย่างเช่น ฟังก์ชั่น print() ใน Python เวอร์ชั่น 3 ไม่ได้ถูกเขียนให้คืนค่าใด ๆ กลับมา ดังนั้นค่าที่ได้จากการเรียกฟังก์ชั่นนี้จึงเป็น None ดังแสดงในตัวอย่างข้างล่างนี้

```
In [25]: return_value = print('abc')

abc

In [26]: print(return_value)

None
```

ในทำนองเดียวกัน ฟังก์ชั่นอื่น ๆ ที่ไม่ได้ถูกเขียนให้มีการคืนค่ากลับมา Python จะคืนค่า None กลับมาให้

Boolean Type

ข้อมูลชนิด Boolean มีอยู่เพียงสองตัวเท่านั้น คือ True และ False และผลลัพธ์ที่ได้จาก comparison operator จะเป็นข้อมูลชนิด Boolean ดังแสดงในตัวอย่างข้างล่างนี้

```
In [27]:  result = (4 < 5)
          result

Out [27]: True

In [28]:  type(result)

Out [28]: bool
```

ผู้อ่านควรระวังที่จะเขียน True และ False ให้ถูกต้อง กล่าวคือ ตัวอักษรตัวแรกจะเป็นตัวใหญ่ ส่วนตัวอักษรที่เหลือจะเป็นตัวเล็ก ดังแสดงในตัวอย่างข้างล่างนี้

```
In [29]: print(True, False)

True False
```

เราสามารถสร้างข้อมูลชนิด Boolean จากข้อมูลชนิดอื่น ๆ ได้โดยใช้ bool() ซึ่งจะได้ค่าออกมาเป็น True หรือ False นั้นก็จะขึ้นอยู่กับกฎเกณฑ์ของภาษาซึ่งสามารถคาดเดาได้ไม่ยาก ยกตัวอย่างเช่น เมื่อใช้ bool()

กับข้อมูลที่เป็นตัวเลขใด ๆ ก็ตามที่ไม่ใช่ศูนย์ ก็จะให้ผลลัพธ์ออกมาเป็น `True` แต่ถ้าเป็นศูนย์ จะให้ผลลัพธ์ออกมาเป็น `False` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [30]:  bool(2014)
Out [30]: True
In [31]:  bool(0)
Out [31]: False
In [32]:  bool(3.1415)
Out [32]: True
```

เมื่อใช้ `bool()` กับข้อมูลชนิด `None` จะให้ผลลัพธ์ออกมาเป็น `False` เสมอ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [33]:  bool(None)
Out [33]: False
```

เมื่อใช้ `bool()` กับสตริงใด ๆ ก็ตามที่ไม่ใช่สตริงว่าง (empty string) ก็จะให้ผลลัพธ์ออกมาเป็น `True` แต่ถ้าไม่ใช่สตริงว่าง จะให้ผลลัพธ์ออกมาเป็น `False` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [34]:  bool("")
Out [34]: False
In [35]:  bool("abc")
Out [35]: True
```

ในกรณีของข้อมูลประเภท sequence ซึ่งจะเรียนในภายหลัง เมื่อเราใช้ `bool()` กับข้อมูลประเภทนี้ ในกรณีที่เป็น empty sequence เราก็จะได้ผลลัพธ์ออกมาเป็น `False` แต่ถ้าไม่ใช่ empty sequence เราก็จะได้ผลลัพธ์ออกมาเป็น `True` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [36]:  bool([1, 2, 3])
Out [36]: True
In [37]:  bool([])
Out [37]: False
```

Built-In Data Structures

เราได้เรียนข้อมูลประเภท simple type ในภาษา Python อย่างเช่น `int`, `float`, `complex`, `bool` และ `str` มาแล้ว ในบทนี้เราจะเรียนข้อมูลประเภท compound type ดังแสดงในตารางข้างล่างนี้ ซึ่งจะมีลักษณะเหมือนภาชนะสำหรับใส่ข้อมูลชนิดอื่น ๆ ลงไป

Type Name	Example	Description
list	[1, 2, 3]	Ordered collection
tuple	(1, 2, 3)	Immutable ordered collection
dict	{'a':1, 'b':2, 'c':3}	Unordered (key,value) mapping
set	{1, 2, 3}	Unordered collection of unique values

เราจะเห็นได้ว่า วงเล็บ (), ก้ามปู [], หรือปีกกา {} มีความหมายที่ใช้บ่งบอกชนิดของข้อมูลที่แตกต่างกัน ซึ่งผู้เขียนจะอธิบายชนิดของข้อมูลเหล่านี้โดยสังเขป

Lists

ลิสต์เป็นข้อมูลประเภท collection คือ มีสมาชิกได้หลายตัวภายในลิสต์ เราสร้างลิสต์โดยใช้เครื่องหมายก้ามปู [] และระบุสมาชิกแต่ละตัวภายในก้ามปู [] คั่นกันด้วยเครื่องหมายจุลภาค (,) สมาชิกของลิสต์สามารถเปลี่ยนแปลงได้ (mutable) และลำดับของสมาชิกที่อยู่ในลิสต์นั้นมีความสำคัญ (ordered) ตัวอย่างของลิสต์ที่มีสมาชิกเป็นเลขเฉพาะ 4 ตัวแรก เป็นดังนี้

```
In [1]: L = [2, 3, 5, 7]
```

ลิสต์มีแอตทริบิวท์และเมทธอดที่มีประโยชน์อยู่หลายตัว ในที่นี้ผู้เขียนจะแนะนำให้รู้จักบางตัวที่ใช้งานกันบ่อย ๆ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]:  # Length of a list
         len(L)

Out [2]: 4

In [3]:  # Append a value to the end
         L.append(11)
         L

Out [3]: [2, 3, 5, 7, 11]

In [4]:  # Addition concatenates lists
         L + [13, 17, 19]

Out [4]: [2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [5]:  # sort() method sorts in-place
         L = [2, 5, 1, 6, 3, 4]
         L.sort()
         L

Out [5]: [1, 2, 3, 4, 5, 6]
```

นอกจากนี้ ลิสต์ยังมีเมทธอดอีกมากมาย ซึ่งได้อธิบายอย่างละเอียดที่ https://docs.python.org/3/tutorial/datastructures.html

ในตัวอย่างข้างต้น เราจะเห็นว่าสมาชิกของลิสต์เป็นข้อมูลชนิดเดียวกัน ซึ่งหนึ่งในฟีเจอร์ที่โดดเด่นของภาษา Python ก็คือ สมาชิกแต่ละตัวของข้อมูลประเภท compound type ไม่จำเป็นต้องเป็นข้อมูลชนิดเดียวกัน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [6]: L = [1, 'two', 3.14, [0, 3, 5]]
```

ฟีเจอร์นี้เกิดมาจากการที่ภาษา Python เป็นภาษาแบบ dynamically typed การที่เราจะมีสมาชิกที่ต่างชนิดกันในภาษาที่เป็นแบบ statically typed อย่างเช่น ภาษา C จะเป็นเรื่องที่น่าปวดหัวมาก จากตัวอย่างนี้เราก็จะเห็นว่าสมาชิกของลิสต์สามารถเป็นลิสต์ก็ได้ การที่ภาษา Python มีความยืดหยุ่นในเรื่องชนิดของข้อมูลแบบนี้ ทำให้เราสามารถเขียนโปรแกรมได้เร็วและอ่านเข้าใจได้ง่าย

เราได้ดูการใช้งานลิสต์ในลักษณะที่เป็นกลุ่มเป็นก้อนแล้ว ต่อไปเราจะดูการใช้งานลิสต์ในลักษณะที่เราจะ access สมาชิกแต่ละตัวของลิสต์ ซึ่งสามารถทำได้โดยการใช้ indexing และ slicing ดังจะกล่าวในหัวข้อถัดไป

List indexing and slicing

เราสามารถใช้ indexing ในการ access สมาชิกแต่ละตัวของลิสต์และใช้ slicing ในการ access สมาชิกทีละหลาย ๆ ตัวได้ ทั้ง indexing และ slicing จะใช้เครื่องหมายก้ามปู [] ทั้งคู่ ดังแสดงในตัวอย่างข้างล่างนี้ โดยในตัวอย่างนี้เราจะเริ่มต้นด้วยลิสต์ที่มีสมาชิกเป็นเลขเฉพาะ 5 ตัวแรก

```
In [7]: L = [2, 3, 5, 7, 11]
```

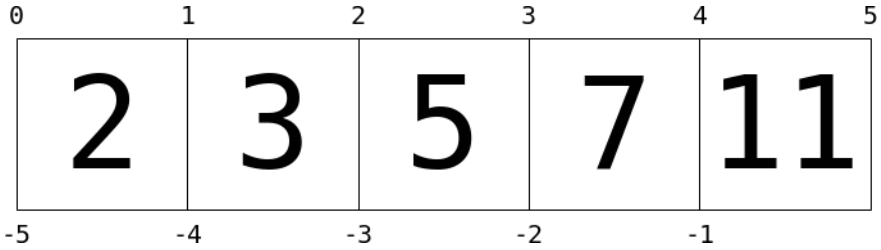
index ในภาษา Python จะเริ่มที่ศูนย์ ดังนั้นเราสามารถเข้า access ตัวที่หนึ่งและสองได้ดังนี้

```
In [8]:  L[0]
Out [8]: 2
In [9]:  L[1]
Out [9]: 3
```

เราสามารถ access สมาชิกของลิสต์จากด้านท้ายของลิสต์ก็ได้ โดยใช้ index ที่มีเป็นค่าลบ เริ่มต้นจาก -1 ดังแสดงในตัวอย่างต่อไปนี้

```
In [10]:  L[-1]
Out [10]: 11
In [12]:  L[-2]
Out [12]: 7
```

เราสามารถวาดภาพแสดงวิธีการใช้ index ได้ดังนี้



ในรูปนี้ ตัวเลขขนาดใหญ่ที่อยู่ภายในช่องแต่ละช่องแสดงถึงค่าของสมาชิกแต่ละตัวของลิสต์ ในขณะที่ตัวเลขขนาดเล็กที่อยู่ด้านบนและด้านล่างของช่องแสดงถึง index จากรูปนี้ L[2] จะคืนค่า 5 มาให้ เพราะว่ามันคือค่าที่อยู่ถัดจาก index หมายเลข 2

indexing คือ การ access สมาชิกแต่ละตัวของลิสต์ ในขณะที่ slicing คือ การสร้างลิสต์ตัวใหม่ที่มีสมาชิกบางส่วนเหมือนกับลิสต์ที่มีอยู่แล้ว โดยจะเหมือนตั้งแต่สมาชิกตัวที่อยู่ที่ index ที่ระบุไว้ข้างหน้าเครื่องหมายทวิภาค (:) จนไปถึงตัวที่อยู่ที่ index ที่ระบุข้างหลังเครื่องหมายทวิภาค (:) แต่ไม่รวมตัวนี้ ยกตัวอย่างเช่น ถ้าเราต้องการสร้างลิสต์ใหม่ที่มีสมาชิก 3 ตัวแรกเหมือนกับลิสต์ L เราสามารถเขียนโค้ดได้ดังนี้

```
In [12]:  L[0:3]
Out [12]: [2, 3, 5]
```

ให้เราสังเกตตำแหน่งของ index ที่ 0 และ 3 ในรูปภาพข้างบน และสังเกตด้วยว่า slicing จะสร้างลิสต์ตัวใหม่ที่มีเฉพาะค่าที่อยู่ระหว่าง index ทั้งสองตัวนี้ ถ้าเราไม่เขียน index ตัวที่อยู่ข้างหน้าเครื่องหมายทวิภาค (:) Python จะตีความว่าเป็น index ที่ศูนย์ ดังนั้นเราสามารถเขียนโค้ดดังตัวอย่างต่อไปนี้ก็ได้ ซึ่งจะให้ผลลัพธ์เหมือนกับโค้ดก่อนหน้านี้

```
In [13]:  L[:3]
Out [13]: [2, 3, 5]
```

ในทำนองเดียวกัน ถ้าเราไม่เขียน index ตัวที่อยู่ข้างหลังเครื่องหมายทวิภาค (:) Python จะตีความว่าเป็นความยาวของลิสต์ ดังนั้น ถ้าเราอยากสร้างลิสต์ใหม่ที่มีสมาชิก 3 ตัวหลังเหมือนกับลิสต์ L เราสามารถเขียนโค้ดได้ดังนี้

```
In [14]:  L[-3:]
Out [14]: [5, 7, 11]
```

สุดท้ายนี้ เราสามารถระบุตัวเลขตัวที่สามได้ ซึ่งจะหมายถึง step ยกตัวอย่างเช่น ถ้าเราต้องการสร้างลิสต์ตัวใหม่ที่มีสมาชิกเหมือนกับลิสต์ L เฉพาะสมาชิกทุกตัวที่สองของลิสต์ L เราสามารถเขียนโค้ดได้ดังนี้

```
In [15]:  L[::2]  # equivalent to L[0:len(L):2]
Out [15]: [2, 5, 11]
```

นอกจากนี้ เราสามารถระบุ step ที่เป็นค่าลบได้ ซึ่งจะสร้างลิสต์ใหม่ที่มีสมาชิกเรียงลำดับกลับด้านกับลิสต์ที่มีอยู่แล้ว ดังแสดงในตัวอย่างข้างล่างนี้

```
In [16]:  L[::-1]
Out [16]: [11, 7, 5, 3, 2]
```

นอกจากเราจะสามารถใช้ indexing และ slicing ในการเข้า access ลิสต์แล้ว เรายังสามารถใช้ในการกำหนดค่าได้ด้วย ดังแสดงในตัวอย่างข้างล่างนี้

```
In [17]: L[0] = 100
         print(L)

[100, 3, 5, 7, 11]

In [18]: L[1:3] = [55, 56]
         print(L)

[100, 55, 56, 7, 11]
```

แพ็คเกจสำหรับงานทางด้านวิทยาการข้อมูล อย่างเช่น NumPy และ Pandas (ที่ได้กล่าวถึงในบท Introduction) ก็มีการใช้รูปแบบไวยากรณ์ที่คล้ายคลึงกับ slicing ด้วย

ถัดไป เราจะไปดูชนิดของข้อมูลที่เป็นแบบ compound type ที่เหลืออีก 3 ตัว

Tuples

ทูเปิล (tuple) จะเหมือนกับลิสต์ในหลาย ๆ ด้าน แต่เวลาเราสร้างทูเปิล เราจะไม่ใช้ก้ามปู [] แต่เราจะใช้วงเล็บ () แทน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [19]: t = (1, 2, 3)
```

นอกจากนี้ เราสามารถสร้างทูเปิลได้ โดยที่ไม่ใช้วงเล็บเลยก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [20]: t = 1, 2, 3
         print(t)

(1, 2, 3)
```

ในทำนองเดียวกับลิสต์ที่ได้กล่าวไปก่อนหน้านี้แล้ว เราสามารถหาจำนวนสมาชิกของทูเปิลและ access สมาชิกแต่ละตัวของทูเปิลได้ด้วยการใช้ indexing ดังแสดงในตัวอย่างต่อไปนี้

```
In [21]:  len(t)

Out [21]: 3

In [22]:  t[0]

Out [22]: 1
```

ความแตกต่างหลัก ๆ ระหว่างลิสต์กับทูเปิล คือทูเปิลจะเป็น immutable หมายความว่า หลังจากที่เราสร้างทูเปิลแล้ว เราไม่สามารถเปลี่ยนค่าของสมาชิกหรือจำนวนของสมาชิกได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [23]: t[1] = 4

-----------------------------------------------------------

TypeError                Traceback (most recent call last)

<ipython-input-23-141c76cb54a2> in <module>()
----> 1 t[1] = 4


TypeError: 'tuple' object does not support item assignment

In [24]: t.append(4)

-----------------------------------------------------------

AttributeError                Traceback (most recent call last)

<ipython-input-24-e8bd1632f9dd> in <module>()
----> 1 t.append(4)
```

ทูเปิลถูกใช้งานบ่อยในโปรแกรม Python โดยเฉพาะอย่างยิ่งกรณีที่ฟังก์ชั่นมีค่าที่ต้องคืนหลายค่า ยกตัวอย่างเช่น เมทธอด `as_integer_ratio` ของออบเจ็คต์ประเภทเลขทศนิยม จะคืนค่าตัวเศษและตัวส่วนออกมาให้ โดยที่ค่าทั้งสองค่านี้จะถูกคืนออกมาอยู่ในรูปของทูเปิล ดังแสดงในตัวอย่างข้างล่างนี้

```
In [25]:  x = 0.125
          x.as_integer_ratio()

Out [25]: (1, 8)
```

ซึ่งเราสามารถทำ unpack เพื่อเอาค่าแต่ละค่าไปกำหนดให้แก่ตัวแปรได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [26]: numerator, denominator = x.as_integer_ratio()
         print(numerator / denominator)

0.125
```

เราสามารถใช้ indexing กับ slicing รวมทั้งเมทธอดหลาย ๆ ตัวที่ใช้กับลิสต์ไปใช้กับทูเปิลได้เหมือนกัน ซึ่งผู้อ่านสามารถศึกษาเพิ่มเติมได้ที่ https://docs.python.org/3/tutorial/datastructures.html

Dictionaries

เราสามารถสร้างแ (dictionary) ได้โดยใช้เครื่องหมายปีกกา {} โดยที่สมาชิกแต่ละตัวของ ดิกชันนารีจะประกอบด้วย key และ value ซึ่งจะคั่นกันด้วยเครื่องหมายทวิภาค (:) และถ้ามีสมาชิกหลายตัว แต่ละตัวจะคั่นกันด้วยเครื่องหมายจุลภาค (,) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [27]: numbers = {'one':1, 'two':2, 'three':3}
```

การเข้า access และกำหนดค่าให้แก่สมาชิกของดิกชันนารีก็จะใช้ index ได้เหมือนกับลิสต์และทูเปิล แต่แตกต่างกันตรงเพียงที่ว่าดิกชันนารีจะไม่ใช้ตัวเลข แต่จะใช้ key เป็น index แทน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [28]:  # Access a value via the key
          numbers['two']

Out [28]: 2
```

เราสามารถเพิ่มสมาชิกให้แก่ดิกชันนารีโดยการใช้ index ได้ด้วย ดังแสดงในตัวอย่างข้างล่างนี้

```
In [29]: # Set a new key/value pair
         numbers['ninety'] = 90
         print(numbers)

{'three': 3, 'ninety': 90, 'two': 2, 'one': 1}
```

สิ่งที่เราควรจำ ก็คือ ดิกชันนารีไม่สนใจลำดับของสมาชิกที่อยู่ในดิกชันนารี ส่วนสาเหตุที่ดิกชันนารีถูกออกแบบมาในลักษณะนี้ ก็เพื่อให้การ access สมาชิกเป็นไปได้อย่างรวดเร็วและมีประสิทธิภาพ โดยไม่ขึ้นกับจำนวนสมาชิกในดิกชันนารีว่ามีมากน้อยเพียงไร (ถ้าผู้อ่านอยากจะเข้าใจเกี่ยวกับเรื่องนี้มากยิ่งขึ้น ให้ผู้อ่านศึกษาเรื่อง hash table) ผู้อ่านสามารถศึกษาเมทธอดต่าง ๆ ของดิกชันนารีได้ที่ https://docs.python.org/3/library/stdtypes.html

Sets

เซต (set) เป็นข้อมูลที่สามารถมีสมาชิกจำนวนหลายตัวได้ แต่สมาชิกแต่ละตัวจะต้อง unique ไม่ซ้ำกัน และเซตจะไม่สนใจลำดับของสมาชิกที่อยู่ในเซต เราสามารถสร้างเซตได้ในทำนองเดียวกันกับลิสต์และทูเปิล แต่จะแตกต่างกันตรงที่ว่าเซตจะใช้หมายปีกกา {} แทน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [30]: primes = {2, 3, 5, 7}
         odds = {1, 3, 5, 7, 9}
```

ถ้าผู้อ่านคุ้นเคยกับเรื่องเซตทางคณิตศาสตร์ ผู้อ่านก็จะคุ้นเคยกับการทำงานของเซต อย่างเช่น union, intersection, difference และ symmetric difference เป็นต้น ซึ่งเซตในภาษา Python ก็จะมีเมทธอดและโอเปอเรเตอร์ที่ครอบคลุมการทำงานดังกล่าวของเซตทางคณิตศาสตร์ ตัวอย่างต่อไปนี้แสดงการใช้เมทธอดและโอเปอเรเตอร์ที่ทำงานเหมือนกันของเซต

```
In [31]:  # union: items appearing in either
          primes | odds      # with an operator
          primes.union(odds) # equivalently with a method

Out [31]: {1, 2, 3, 5, 7, 9}

In [32]:  # intersection: items appearing in both
          primes & odds             # with an operator
          primes.intersection(odds) # equivalently with a method

Out [32]: {3, 5, 7}

In [33]:  # difference: items in primes but not in odds
          primes - odds           # with an operator
          primes.difference(odds) # equivalently with a method

Out [33]: {2}

In [34]:
# symmetric difference: items appearing in only one set
primes ^ odds                     # with an operator
primes.symmetric_difference(odds) # equivalently with a method

Out [34]: {1, 2, 9}
```

ผู้อ่านสามารถศึกษาเมทธอดของเซตเพิ่มเติมได้ที่ https://docs.python.org/3/library/stdtypes.html

More Specialized Data Structures

Python ยังมีโครงสร้างข้อมูล (data structure) อื่น ๆ อีกหลายตัวที่อาจจะเป็นประโยชน์กับผู้อ่าน ซึ่งส่วนใหญ่จะอยู่ในโมดูล (module) ที่เรียกว่า `collection` ซึ่งเป็นโมดูลแบบบิวท์อิน (built-in) หมายความว่า เป็นโมดูลที่มากับตัวภาษา Python ไม่ต้องติดตั้งเพิ่มเติม ผู้อ่านสามารถศึกษารายละเอียดเกี่ยวกับโมดูลนี้เพิ่มเติมได้ที่ https://docs.python.org/3/library/collections.html

ในบางโอกาส ผู้เขียนพบว่าโครงสร้างข้อมูลดังต่อไปนี้มีประโยชน์มาก

`collections.namedtuple`

จะคล้าย ๆ กับทูเปิล แต่แตกต่างกันตรงที่ว่า เราสามารถ access แต่ละตัวได้โดยใช้ชื่อนอกเหนือจากการใช้ index

`collections.defaultdict`

จะคล้าย ๆ กับดิกชันนารี แต่แตกต่างกันตรงที่ว่า ถ้าเราใช้ key ที่ไม่มีอยู่ในดิกชันนารีในการหาค่า (value) เราจะได้ค่าดีฟอลต์ ที่ผู้เขียนโปรแกรมระบุไว้คืนกลับมา แทนที่จะเกิด `KeyError` เหมือนอย่างในกรณีของดิกชันนารี

`collections.OrderedDict`

จะคล้าย ๆกับดิกชันนารี แต่ลำดับของ key ที่ถูกใส่เข้าไปในดิกชันนารีจะถูกจดจำไว้

หลังจากที่ผู้อ่านได้เห็นการใช้งาน collection มาตรฐานของภาษา Python อย่างลิสต์ ทูเปิล ดิกชันนารี และเซตแล้ว การใช้งาน collection อื่น ๆ ที่เพิ่มเติมเข้ามาในโมดูลต่าง ๆ ก็จะคล้าย ๆ กัน ซึ่งผู้อ่านสามารถศึกษาเพิ่มเติมได้ที่ https://docs.python.org/3/ library/collections.html

Control Flow

control flow คือ ลำดับที่คำสั่งในโปรแกรมถูกเอ็กซิคิวท์ โดยปกติโปรแกรมของเราจะถูกเอ็กซ์ซีคิวท์ทีละบรรทัดเรียงลำดับกันไป แต่เราสามารถเปลี่ยนลำดับการเอ็กซิคิวท์นี้ได้โดยการใช้คำสั่งควบคุม ซึ่งแบ่งออกเป็น 2 ประเภท คือ คำสั่งเงื่อนไขและ คำสั่งลูป โดยที่คำสั่งเงื่อนไขจะช่วยให้เราเขียนโปรแกรมที่สามารถทำงานบางคำสั่งหรือโค้ดบล็อกตามเงื่อนไข ไม่จำเป็นต้องทำทุกบรรทัด ในขณะที่คำสั่งลูปจะช่วยให้เราเขียนโปรแกรมที่สามารถทำงานซ้ำหลาย ๆ ครั้งได้ ดังนั้นคำสั่งควบคุมจะเป็นเครื่องมือสำคัญที่ช่วยให้เราเขียนโปรแกรมที่ซับซ้อนได้

ในบทนี้ผู้เขียนจะอธิบายคำสั่งเงื่อนไขที่เขียนโดยใช้คีย์เวิร์ด `if`, `elif` และ `else` และจะอธิบายคำสั่งลูปที่เขียนโดยใช้คีย์เวิร์ด `for` และ `while` รวมทั้งคีย์เวิร์ดอื่น ๆ ที่เกี่ยวข้องอย่าง `break`, `continue` และ `pass`

Conditional Statements: if, elif, and else

คำสั่งเงื่อนไขหรือบ่อยครั้งที่เราเรียกว่าคำสั่ง if-then (if-then statement) จะควบคุมให้โปรแกรมทำงานบางคำสั่งตามเงื่อนไขที่เป็น Boolean ดังแสดงในตัวอย่างต่อไปนี้ ซึ่งเป็นโปรแกรมที่ใช้คำสั่งเงื่อนไขแบบง่าย ๆ

```
In [1]: x = -15

        if x == 0:
            print(x, "is zero")
        elif x > 0:
            print(x, "is positive")
        elif x < 0:
            print(x, "is negative")
        else:
            print(x, "is unlike anything I've ever seen...")

-15 is negative
```

ผู้อ่านควรระมัดระวังในการใช้เครื่องหมายทวิภาค (`:`) และช่องว่างที่ตามหลังให้ถูกต้อง ในการบ่งบอกถึงโค้ดบล็อก

ภาษา Python ใช้คีย์เวิร์ด `if` และ `else` เหมือนกันกับในภาษาโปรแกรมมิ่งอื่น ๆ อีกหลายภาษา ส่วนคีย์เวิร์ด `elif` เกิดจากการรวมคำว่า `else` และ `if` เข้าด้วยกันและทำให้สั้นลง ซึ่งจะไม่มีในภาษาอื่น นอกจากนี้เราจะมี `elif` กี่ตัวก็ได้ในโปรแกรมของเรา หรือจะไม่มีเลยก็ได้ ส่วนคีย์เวิร์ด `else` จะมีหรือไม่มีก็ได้

for loops

คำสั่งลูปเป็นคำสั่งที่เราใช้เพื่อให้โปรแกรมทำงานซ้ำหรือวนลูป ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นตัวอย่างการใช้คำสั่ง `for` ลูป (คำสั่งลูปที่ใช้คีย์เวิร์ด `for`) ในการพิมพ์สมาชิกแต่ละตัวของลิสต์ออกหน้าจอ

```
In [2]: for N in [2, 3, 5, 7]:
            print(N, end=' ') # print all on same line

2 3 5 7
```

ผู้เขียนอยากให้ผู้อ่านสังเกตถึงความง่ายในการเขียนคำสั่ง `for` ลูป ในภาษา Python เราแค่ระบุตัวแปรที่ต้องการใช้ จากนั้นระบุออบเจ็คต์ที่เป็นแบบ sequence ที่เราต้องการวนลูป และใช้โอเปอเรเตอร์ `in` ในการเชื่อมความสัมพันธ์ระหว่างสองสิ่งนี้เข้าด้วยกัน อันที่จริงแล้วออบเจ็คต์ที่อยู่ตามหลัง `in` นั้นจะเป็นออบเจ็คต์ iterable ซึ่งจะสร้างหรือคืนออบเจ็คต์ iterator ออกมาให้โปรแกรมใช้ในการวนลูป เราสามารถคิดว่า iterator เป็น generalized sequence ก็ได้ ผู้เขียนจะอธิบายเกี่ยวกับ iterator ในบทเรื่อง Iterator

ยกตัวอย่างเช่น ออบเจ็คต์ `range` ซึ่งเป็นหนึ่งในบรรดาออบเจ็คต์ iterable ที่เราใช้งานกันมากที่สุดในภาษา Python ออบเจ็คต์ `range` จะสร้าง iterator ที่ใช้สำหรับสร้างลำดับของเลขจำนวนเต็ม (sequence of numbers) เพื่อให้นำมาใช้ในการวนลูป ดังแสดงในตัวอย่างข้างล่างนี้

```
In [3]: for i in range(10):
            print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

ผู้อ่านควรทราบด้วยว่า โดยปกติตัวเลขตัวแรกของลำดับที่สร้างผ่าน iterator ของออบเจ็คต์ `range` จะเป็นศูนย์และตัวเลขตัวสุดท้ายจะไม่รวมตัวเลขที่เราระบุในวงเล็บของ `range()` นอกจากนี้เรายังสามารถใช้ `range()` ในลักษณะที่ซับซ้อนขึ้นในการสร้างลำดับของตัวเลขได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]:  # range from 5 to 10
         list(range(5, 10))

Out [4]: [5, 6, 7, 8, 9]

In [5]:  # range from 0 to 10 by 2
         list(range(0, 10, 2))

Out [5]: [0, 2, 4, 6, 8]
```

ผู้อ่านอาจจะสังเกตเห็นว่า ตัวเลขที่เราระบุภายในวงเล็บของ `range()` จะมีความหมายในทำนองเดียวกันกับตัวเลขที่เราระบุในการทำ slicing ของลิสต์

สิ่งที่ผู้อ่านควรระวัง ก็คือ พฤติกรรมของ range() ใน Python เวอร์ชั่น 2 จะแตกต่างจากเวอร์ชั่น 3 กล่าวคือ range() ใน Python เวอร์ชั่น 2 จะสร้างลิสต์ของลำดับของตัวเลข ในขณะที่เวอร์ชั่น 3 จะสร้างออบเจ็คต์ iterator

while loops

คำสั่งลูปอีกชนิดหนึ่งคือ คำสั่งลูป while (คำสั่งลูปที่ใช้คีย์เวิร์ด while) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [6]: i = 0
        while i < 10:
            print(i, end=' ')
            i += 1

0 1 2 3 4 5 6 7 8 9
```

คำสั่งลูป while จะวนลูปหรือทำซ้ำตามเงื่อนไขที่เขียนอยู่ข้างหลังคีย์เวิร์ด while ซึ่งจะถูกตีความให้เป็นค่า Boolean และจะวนลูปจนกว่าเงื่อนไขนี้จะเป็น False

break and continue: Fine-Tuning Your Loops

เราสามารถใช้คำสั่งดังต่อไปนี้ภายในลูป เพื่อปรับแต่งทำงานของลูป

- คำสั่ง break (break statement) เป็นคำสั่งที่ใช้คีย์เวิร์ด break ในการสั่งให้ออกจากลูปที่คำสั่งนี้อยู่
- คำสั่ง continue (continue statement) เป็นคำสั่งที่ใช้คีย์เวิร์ด continue ในการสั่งให้ยุติการทำงานที่เหลือในลูปรอบปัจจุบันแล้วข้ามไปทำลูปรอบถัดไปแทน

เราสามารถนำทั้งสองคำสั่งนี้ไปใช้กับคำสั่งลูป for หรือคำสั่งลูป while ก็ได้

ตัวอย่างต่อไปนี้เป็นตัวอย่างการใช้คำสั่ง continue ในโปรแกรมที่ทำการพิมพ์เฉพาะเลขคู่ออกหน้าจอ ในกรณีตัวอย่างนี้ เราอาจจะใช้เพียงแค่คำสั่ง if-else ก็ได้ ซึ่งก็จะให้ผลลัพธ์ออกมาเหมือนกัน แต่การที่เราใช้คำสั่ง continue ทำให้เราเขียนโค้ดที่สื่อความคิดของเราได้ง่ายกว่า

```
In [7]: for n in range(20):
            # check if n is even
            if n % 2 == 0:
                continue
            print(n, end=' ')

1 3 5 7 9 11 13 15 17 19
```

ตัวอย่างถัดไปเป็นตัวอย่างการใช้คำสั่ง break ในโปรแกรมที่ทำการสร้างลิสต์ที่มีสมาชิกเป็นตัวเลข Fibonacci ที่มีค่าไม่เกิน 100

```
In [8]: a, b = 0, 1
        amax = 100
        L = []

        while True:
            (a, b) = (b, a + b)
            if a > amax:
                break
            L.append(a)

        print(L)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

ให้ผู้อ่านสังเกตด้วยว่าคำสั่งลูป `while` นี้จะวนลูปไปเรื่อย ๆ ไม่มีสิ้นสุด ถ้าเราไม่มีมีคำสั่ง `break`

Loops with an else Block

ในภาษา Python มีรูปแบบที่เรียกว่าลูป `else` ซึ่งเป็นรูปแบบที่เราเอาคำสั่ง `else` ไปต่อท้ายคำสั่งลูป `for` หรือคำสั่งลูป `while` แต่เราจะไม่ค่อยเห็นรูปแบบนี้กันบ่อยนัก เราได้เรียนเกี่ยวกับ else มาแล้วในก่อนหน้านี้ ซึ่งโค้ดในส่วนของ else จะถูกเอ็กซิคิวท์ก็ต่อเมื่อเงื่อนไขที่อยู่ในคำสั่ง `if` และ `elif` ทั้งหมดให้ค่าออกมาเป็น False เท่านั้น ลูป `else` เป็นหนึ่งในคำสั่งในภาษา Python ที่ชื่อของมันทำให้ผู้เขียนโค้ดสับสน โดยส่วนตัวแล้ว ผู้เขียนอยากจะเรียกคำสั่งนี้ว่าคำสั่ง `nobreak` กล่าวคือ โค้ดในส่วนของ else จะถูกเอ็กซีคิวท์ก็ต่อเมื่อลูปนั้นทำงานสิ้นสุดตามปกติ ไม่ได้ออกจากลูปเนื่องมาจากการใช้คำสั่ง `break`

ตัวอย่างต่อไปนี้เป็นตัวอย่างที่แสดงประโยชน์ของการใช้ลูป `else` โดยตัวอย่างนี้เป็นการอิมพลีเมนต์อัลกอริทึม Sieve of Eratosthenes ซึ่งเป็นอัลกอริทึมที่ใช้หาเลขเฉพาะ

```
In [9]: L = []
        nmax = 30

        for n in range(2, nmax):
            for factor in L:
                if n % factor == 0:
                    break
            else: # no break
                L.append(n)
        print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

เราสามารถลูป `else` ในทำเดียวกันกับตัวอย่างข้างต้นกับคำสั่งลูป `while`

Defining and Using Functions

โค้ดในตัวอย่างที่ผ่าน ๆ มาจะเป็นลักษณะง่าย ๆ คือ โค้ดทั้งหมดเขียนรวมอยู่ด้วยกันและใช้แค่ครั้งเดียวในโปรแกรม ซึ่งจะไม่เหมาะกับการเขียนโปรแกรมที่มีขนาดใหญ่ ฟังก์ชั่นเป็นวิธีการหนึ่งที่ทำให้เราสามารถจัดระเบียบของโค้ด ทำให้อ่านโค้ดได้เข้าใจได้ง่ายขึ้น และสามารถนำกลับมาใช้หลาย ๆ ครั้ง (reuse) ในโปรแกรมได้ ในบทนี้เราจะเรียนการสร้างฟังก์ชั่น 2 วิธี คือ การสร้างฟังก์ชั่นโดยใช้คีย์เวิร์ด `def` ซึ่งสามารถใช้สร้างฟังก์ชั่นในทุกรูปแบบได้ และการสร้างฟังก์ชั่นโดยใช้คำสั่ง `lambda` (`lambda` statement) ซึ่งจะเหมาะกับการสร้างฟังก์ชั่นสั้น ๆ ที่ไม่มีชื่อฟังก์ชั่น

Using Functions

ฟังก์ชั่น คือ กลุ่มของโค้ดที่มีชื่อ และถูกเรียกโดยใช้วงเล็บ เราเคยเห็นฟังก์ชั่นมาแล้ว ยกตัวอย่างเช่น `print` ใน Python เวอร์ชั่น 3 เป็นฟังก์ชั่น

```
In [1]: print('abc')
abc
```

ในตัวอย่างนี้ `print` เป็นชื่อของฟังก์ชั่น ส่วน `abc` เป็นอากิวเมนต์ที่ส่งให้ฟังก์ชั่น

นอกจากอากิวเมนต์ที่แสดงในตัวอย่างข้างต้นแล้ว เรายังมีคีย์เวิร์ดอากิวเมนต์ (keyword argument) ซึ่งเราจะใช้ชื่อในการระบุ ในกรณีของฟังก์ชั่น `print` จะมีคีย์เวิร์ดอากิวเมนต์ อย่างเช่น `sep` สำหรับใช้ระบุอักขระที่ใช้ในการแยกหรือคั่นระหว่างข้อมูลแต่ละตัวเวลาแสดงผลลัพธ์ออกหน้าจอ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]: print(1, 2, 3)
1 2 3
In [3]: print(1, 2, 3, sep='--')
1--2--3
```

เวลาเราใช้อากิวเมนต์ธรรมดาที่ไม่ใช่คีย์เวิร์ดอากิวเมนต์ร่วมกับคีย์เวิร์ดอากิวเมนต์ เราต้องเขียนคีย์เวิร์ดอากิวเมนต์หลังจากอากิวเมนต์ธรรมดาเสมอ

Defining Functions

ฟังก์ชั่นจะมีประโยชน์มากขึ้น เมื่อเราสามารถนิยามฟังก์ชั่นของเราเองได้ และนำไปใช้ในส่วนต่าง ๆ ของโปรแกรม เราสามารถนิยามฟังก์ชั่นได้โดยใช้คีย์เวิร์ด `def` ดังแสดงในตัวอย่างต่อไปนี้ ซึ่งเป็นการเขียนฟังก์ชั่นโดยใช้โค้ดในตัวอย่างก่อนหน้านี้ คือ โค้ดที่ทำการสร้างลิสต์ที่มีสมาชิกเป็นตัวเลข Fibonacci

```
In [4]: def fibonacci(N):
            L = []
            a, b = 0, 1
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

ณ ตอนนี้ เรามีฟังก์ชั่นที่ชื่อ `fibonacci` ซึ่งมีพารามิเตอร์ `N` สำหรับรับค่าอากิวเมนต์ที่ส่งมาให้กับฟังก์ชั่นเพื่อใช้ในการประมวลผลภายในฟังก์ชั่น ฟังก์ชั่นนี้จะคืนค่ากลับมาให้เป็นลิสต์ที่มีสมาชิกประกอบด้วย `N` ตัวแรกของเลข Fibonacci ดังแสดงในตัวอย่างข้างล่างนี้

```
In [5]:  fibonacci(10)

Out [5]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

ถ้าผู้อ่านคุ้นเคยกับภาษาที่เป็นแบบ strongly typed อย่างเช่น ภาษา C ผู้อ่านจะสังเกตเห็นว่าทั้งพารามิเตอร์ที่ใช้ในการรับค่าและค่าที่ฟังก์ชั่นคืนกลับไปให้ ต่างก็ไม่มีชนิดของข้อมูลผูกอยู่ การที่ฟังก์ชั่นในภาษา Python สามารถคืนค่าออกมาเป็นออบเจ็คต์ชนิดใดก็ได้ ทำให้โค้ดที่เขียนแล้วค่อนข้างซับซ้อนในภาษาอื่นสามารถนำมาเขียนในภาษา Python ได้อย่างตรงไปตรงมาไม่ซับซ้อน

ยกตัวอย่างเช่น ถ้าเราเขียนฟังก์ชั่นให้คืนค่าเป็นออบเจ็คต์ชนิดทูเปิล ก็จะทำให้เราคืนค่าหลาย ๆ ค่าจากฟังก์ชั่นได้ โดยค่าเหล่านี้จะเป็นสมาชิกของทูเปิล ซึ่งคั่นกันด้วยเครื่องหมายจุลภาค (`,`) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [6]: def real_imag_conj(val):
            return val.real, val.imag, val.conjugate()

        r, i, c = real_imag_conj(3 + 4j)
        print(r, i, c)

3.0 4.0 (3-4j)
```

Default Argument Values

บ่อยครั้งเมื่อเรานิยามฟังก์ชั่น เราอาจจะมีค่าบางค่าที่ต้องการให้ฟังก์ชั่นนั้นใช้เกือบตลอดเวลา แต่ก็ยังอยากให้ฟังก์ชั่นมีความยืดหยุ่นที่จะสามารถรับค่าจากผู้ที่เรียกใช้งานฟังก์ชั่นได้ ในกรณีเช่นนี้ เราสามารถใช้ค่าดีฟอลต์กับพารามิเตอร์ของฟังก์ชั่นได้ ให้เราลองกลับมาพิจารณาฟังก์ชั่น `fibonacci` ในตัวอย่างก่อนหน้านี้ เราสามารถเขียนโค้ดที่กำหนดค่าดีฟอลต์ให้แก่ค่าเริ่มต้นที่จะนำมาใช้ในการหาตัวเลข Fibonacci ได้ กล่าวคือ เราสามารถกำหนดค่าดีฟอลต์ให้ `a` เป็นอะไรก็ได้ อย่างเช่นเป็น `0` และให้ `b` เป็น `1` แต่ก็ยังอนุญาตให้ผู้เรียกใช้ฟังก์ชั่นนี้สามารถระบุค่าเริ่มต้นที่เป็นค่าอื่นให้แก่ `a` และ `b` ได้ ถ้าต้องการ เราสามารถปรับโค้ดได้ดังนี้

```
In [7]: def fibonacci(N, a=0, b=1):
            L = []
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

ในกรณีที่เราส่งอากิวเมนต์ตัวเดียวให้กับฟังก์ชั่นนี้ เราก็จะได้ผลลัพธ์กลับมาเหมือนเดิม ก่อนเราปรับฟังก์ชั่น fibonacci

```
In [8]:  fibonacci(10)

Out [8]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

แต่เราสามารถระบุค่าเริ่มต้นในการหาตัวเลข Fibonacci ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [9]:  fibonacci(10, 0, 2)

Out [9]: [2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

เราสามารถระบุค่าเริ่มต้นโดยการใช้ชื่อพารามิเตอร์ก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งในกรณีนี้ลำดับจะไม่มีความสำคัญอีกต่อไป กล่าวคือ เราจะระบุค่าของ b ก่อน a ก็ได้

```
In [10]:  fibonacci(10, b=3, a=1)

Out [10]: [3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

*args and **kwargs: Flexible Arguments

ในบางครั้ง เราอาจจะอยากเขียนฟังก์ชั่นที่สามารถรับอากิวเมนต์ที่เราไม่รู้ว่า ผู้เรียกใช้ฟังก์ชั่นจะส่งมาให้เป็นจำนวนเท่าไร ในกรณีนี้ เราสามารถใช้พารามิเตอร์ที่อยู่ในรูป *args และ **kwargs ซึ่งจะสามารถรับอากิวเมนต์ที่ถูกส่งมาให้ได้ทั้งหมด ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]: def catch_all(*args, **kwargs):
             print("args =", args)
             print("kwargs = ", kwargs)

In [12]: catch_all(1, 2, 3, a=4, b=5)

         args = (1, 2, 3)
         kwargs =  {'a': 4, 'b': 5}

In [13]: catch_all('a', keyword=2)

         args = ('a',)
         kwargs =  {'keyword': 2}
```

อันที่จริงแล้วชื่อของพารามิเตอร์ไม่จำเป็นต้องเป็น args หรือ kwargs จะเป็นชื่ออะไรก็ได้ แต่ในทางปฏิบัติเรานิยมใช้คำว่า args ซึ่งย่อมาจาก argument และคำว่า kwargs ซึ่งย่อมาจาก keyword argument สิ่งที่สำคัญในตัวอย่างนี้อยู่ที่อักขระ * ที่นำหน้าพารามิเตอร์ทั้งสองตัวนี้ ซึ่งอักขระ * หนึ่งตัวหมายถึง ให้รวมอากิว

เมนต์ที่ผู้ใช้ส่งมาให้ ให้อยู่ในรูปของทูเปิล ในขณะที่อักขระ ** จำนวนสองตัวหมายถึง ให้รวมคีย์เวิร์ดอากิวเมนต์ที่ผู้ใช้ส่งมาให้ ให้อยู่ในรูปของดิกชันนารี

นอกจากจะใช้อักขระ * ในตอนที่เรานิยามฟังก์ชั่นแล้ว เรายังสามารถใช้ * ในตอนที่เราเรียกฟังก์ชั่นได้ด้วย ซึ่งในกรณีนี้ Python จะไม่ส่งทูเปิลหรือดิกชันนารีไปให้กับฟังก์ชั่น แต่จะทำการส่งสมาชิกแต่ละตัวไปให้ฟังก์ชั่นแทน ดังนั้นหลังจากที่ฟังก์ชั่น `catch_all` ได้รับสมาชิกแต่ละตัวของทูเปิล `input` และดิกชันนารี `keywords` เป็นอากิวเมนต์แล้ว ฟังก์ชั่น `catch_all` ก็จะทำงานตามที่ได้อธิบายไปในข้างต้น กล่าวคือ * และ ** ก็จะรวมสมาชิกแต่ละตัวที่ส่งมาให้ ให้อยู่ในรูปทูเปิลกับดิกชันนารี ดังแสดงในตัวอย่างข้างล่างนี้

```
In [14]: inputs = (1, 2, 3)
         keywords = {'pi': 3.14}

         catch_all(*inputs, **keywords)

         args = (1, 2, 3)
         kwargs =  {'pi': 3.14}
```

Anonymous (lambda) Functions

ก่อนหน้านี้ ผู้เขียนได้อธิบายวิธีการนิยามฟังก์ชั่นโดยการใช้คีย์เวิร์ด `def` โดยสังเขปไปแล้ว วิธีการนิยามฟังก์ชั่นแบบนี้จะเป็นวิธีการที่ใช้กันทั่ว ๆ ไป อย่างไรก็ตาม ยังมีวิธีการนิยามฟังก์ชั่นอีกวิธีที่ผู้อ่านน่าจะมีโอกาสพบเห็น คือ การนิยามโดยใช้คำสั่ง `lambda` ซึ่งจะเหมาะกับการนิยามฟังก์ชั่นสั้น ๆ ที่ใช้เพียงครั้งเดียว ตัวอย่างต่อไปนี้เป็นตัวอย่างการสร้างฟังก์ชั่นโดยใช้คำสั่ง `lambda` และตั้งชื่อว่า `add`

```
In [15]:  add = lambda x, y: x + y
          add(1, 2)

Out [15]: 3
```

ซึ่งจะคล้าย ๆ กับการนิยามฟังก์ชั่นโดยใช้คีย์เวิร์ด `def` ดังนี้

```
In [16]: def add(x, y):
             return x + y
```

ผู้อ่านอาจจะสงสัยว่า ทำไมเราถึงอยากมี `lambda` ฟังก์ชั่น คำตอบก็คือ เราสามารถส่ง `lambda` ฟังชั่นนี้ไปเป็นอากิวเมนต์ของฟังก์ชั่นอื่นได้ อย่าลืมว่าทุกอย่างในภาษา Python เป็นออบเจ็คต์ ดังนั้นฟังก์ชั่นก็เป็นออบเจ็คต์ด้วยเหมือนกัน เราจึงสามารถใช้งานฟังก์ชั่นได้เหมือนอากิวเมนต์ทั่ว ๆ ไป

ให้เรามาดูตัวอย่างการใช้ `lambda` ฟังก์ชั่นกัน สมมติว่าเรามีข้อมูลอยู่ในรูปลิสต์ของดิกชันนารี ดังนี้

```
In [17]:
data = [{'first':'Guido', 'last':'Van Rossum', 'YOB':1956},
        {'first':'Grace', 'last':'Hopper',     'YOB':1906},
        {'first':'Alan',  'last':'Turing',     'YOB':1912}]
```

สมมติว่าเราต้องการ sort ข้อมูลนี้ ซึ่งในภาษา Python เรามีฟังก์ชั่น `sorted` ที่ใช้ในการ sort ข้อมูล ดังแสดงในตัวอย่างต่อไปนี้

```
In [18]:  sorted([2,4,3,5,1,6])

Out [18]: [1, 2, 3, 4, 5, 6]
```

แต่เนื่องจากข้อมูลของเราเป็นดิกชันนารี ซึ่งโดยปกติไม่สามารถเรียงลำดับของข้อมูลได้ ดังนั้นเราต้องมีวิธีการบอกให้ฟังก์ชั่น `sorted` รู้ว่า มันควรจะเรียงลำดับข้อมูลที่เป็นดิกชันนารีนี้อย่างไร ฟังก์ชั่น `sorted` จะมีอากิวเมนต์ `key` ที่อนุญาตให้เราระบุฟังก์ชั่นที่กำหนดวิธีการเรียงลำดับของข้อมูลได้ ยกตัวอย่างเช่น ถ้าเราต้องการให้เรียงลำดับของข้อมูลโดยใช้ชื่อคน เราก็จะเขียน `lambda` ฟังก์ชั่น ดังแสดงในตัวอย่างข้างล่างนี้ เมื่อฟังก์ชั่น `sorted` ทำงาน มันจะเรียก `lambda` ฟังก์ชั่นของเรานี้โดยใช้ชื่อว่า `key` และจะส่งดิกชันนารีแต่ละตัวที่เป็นข้อมูลของเราไปให้ `lambda` ฟังก์ชั่นนี้ `lambda` ฟังก์ชั่นก็จะอ้างอิงดิกชันนารีแต่ละตัวโดยใช้ชื่อว่า `item` จากนั้นก็จะคืนค่า `item['first']` ซึ่งเป็นชื่อของแต่ละคนที่เป็นข้อมูลอยู่ในดิกชันนารีออกมาให้ จากนั้นฟังก์ชั่น `sorted` ก็จะเอาค่านี้มาใช้ในการเรียงลำดับดิกชันนารีข้อมูลของเรา ซึ่งก็จะได้ผลลัพธ์ดังนี้

```
In [19]:  # sort alphabetically by first name
          sorted(data, key=lambda item: item['first'])

Out [19]:
[{'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
 {'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]

In [20]:  # sort by year of birth
          sorted(data, key=lambda item: item['YOB'])

Out [20]:
[{'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
 {'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]
```

ถึงแม้ว่าในตัวอย่างนี้ เราจะสามารถใช้ฟังก์ชั่นที่สร้างโดยใช้คีย์เวิร์ด `def` แทน `lambda` ฟังก์ชั่นได้ แต่การใช้ `lambda` ฟังก์ชั่นสำหรับฟังก์ชั่นสั้น ๆ และใช้เพียงครั้งเดียว ก็จะสะดวกกว่ามาก

Errors and Exceptions

ไม่ว่าเราจะมีทักษะในการเขียนโปรแกรมขนาดไหน สุดท้ายก็หนีไม่พ้นที่จะเขียนโปรแกรมที่มี error อยู่ดี error แบ่งออกเป็น 3 ประเภทดังนี้

syntax error เป็นข้อผิดพลาดที่เกิดเนื่องมาจากเราเขียนโค้ดไม่ถูกต้องตามไวยากรณ์ของภาษา Python (โดยปกติ สามารถแก้ไขให้ถูกต้องได้ง่าย)

runtime error เป็นข้อผิดพลาดที่เกิดขึ้นขณะที่คอมพิวเตอร์กำลังเอ็กซิคิวท์โค้ดของเรา ถึงแม้ว่าจะโค้ดของเราจะเขียนถูกต้องตามไวยากรณ์ก็ตาม บางทีอาจมีสาเหตุมาจากการที่ผู้ใช้งานกรอกมาค่าไม่ถูกต้องมาให้ทำให้ประมวลผลไม่ได้ (บางครั้ง สามารถแก้ไขให้ถูกต้องได้ง่าย)

semantic error เป็นข้อผิดพลาดทางตรรกะหรือวิธีคิดของเราที่ใช้ในการเขียนโค้ด ในกรณีนี้โค้ดของเราจะถูกเอ็กซิคิวท์ได้จนจบโปรแกรม แต่ผลลัพธ์ที่ได้จะไม่ตรงกับที่เราคาดหวังไว้ (บ่อยครั้ง จะหาที่ผิดและแก้ไขให้ถูกต้องได้ยาก)

ในบทนี้เราจะให้ความสนใจเฉพาะ runtime error และวิธีจัดการกับ error ประเภทนี้ผ่านกลไกที่เรียกว่า exception handling ในภาษา Python

Runtime Errors

ถ้าผู้อ่านได้ผ่านการเขียนโค้ดภาษา Python มาสักระยะหนึ่งแล้ว ก็คิดว่าผู้อ่านน่าจะเคยเจอ runtime error มาบ้างแล้ว error ประเภทนี้สามารถเกิดขึ้นได้จากหลากหลายสถานการณ์

ยกตัวอย่างเช่น ถ้าเราพยายามอ้างอิงถึงตัวแปรที่ไม่มีอยู่จริง ก็จะเกิด runtime error ดังแสดงในตัวอย่างข้างล่างนี้

```
In [1]: print(Q)
---------------------------------------------------------

NameError                 Traceback (most recent call last)

<ipython-input-3-e796bdcf24ff> in <module>()
----> 1 print(Q)


NameError: name 'Q' is not defined
```

หรือถ้าเราใช้โอเปอเรเตอร์อย่างไม่ถูกต้อง ไม่ได้ใช้ตามรูปแบบที่นิยามไว้ ก็จะเกิด runtime error ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]: 1 + 'abc'

---------------------------------------------------------

TypeError              Traceback (most recent call last)

<ipython-input-4-aab9e8ede4f7> in <module>()
----> 1 1 + 'abc'


TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

หรือถ้าเราเขียนโค้ดสั่งให้เครื่องคอมพิวเตอร์คำนวณสิ่งที่เป็นไปไม่ได้ทางคณิตศาสตร์ ก็จะเกิด runtime error ดังแสดงในตัวอย่างข้างล่างนี้

```
In [3]: 2 / 0

---------------------------------------------------------

ZeroDivisionError              Traceback (most recent call last)

<ipython-input-5-ae0c5d243292> in <module>()
----> 1 2 / 0


ZeroDivisionError: division by zero
```

หรือถ้าเราพยายาม access สมาชิกที่ไม่มีอยู่จริงของลิสต์ ก็จะเกิด runtime error ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]: L = [1, 2, 3]
        L[1000]

---------------------------------------------------------

IndexError              Traceback (most recent call last)

<ipython-input-6-06b6eb1b8957> in <module>()
      1 L = [1, 2, 3]
----> 2 L[1000]


IndexError: list index out of range
```

เราจะสังเกตเห็นว่า ในแต่ละสถานการณ์ที่เกิด runtime error ขึ้นมา Python ไม่เพียงแค่แจ้งว่ามี error เกิดขึ้น แต่ยังบอกรายละเอียดและบรรทัดที่ก่อให้เกิด error นั้น ๆ ซึ่งเป็นประโยชน์ในการช่วยให้เราแก้ไขข้อ error ที่เกิดขึ้น

Catching Exceptions: try and except

เครื่องมือที่ Python เตรียมไว้ให้เราใช้ ในการจัดการกับ runtime error คือ `try...except` ซึ่งมีโครงสร้างดังนี้

```
In [5]:
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")

this gets executed first
```

ผู้อ่านจะสังเกตเห็นว่า โค้ดในส่วน `except` ไม่ได้ถูกเอ็กซิคิวท์ ทั้งนี้เป็นเพราะว่าโค้ดในส่วน `try` นั้นไม่ได้เกิด error ให้เรามาลองสร้าง error ให้เกิดขึ้นในส่วน `try` ดังแสดงในตัวอย่างข้างล่างนี้ และมาลองสังเกตดูว่าเกิดอะไรขึ้น

```
In [6]: try:
            print("let's try something:")
            x = 1 / 0 # ZeroDivisionError
        except:
            print("something bad happened!")

        let's try something:
        something bad happened!
```

ในตัวอย่างนี้ จะเกิด error ประเภท `ZeroDivisionError` ขึ้นในส่วน `try` และ error ดังกล่าวนี้จะถูกโยนออกมาออกมากับให้โค้ดในส่วน `except` ซึ่งในกรณีตัวอย่างนี้ ข้างหลังคีย์เวิร์ด `except` ไม่ได้ระบุประเภทของ error ที่ต้องการดักจับ ดังนั้นก็จะจับ error ทุกประเภทที่ถูกโยนออกมาจากส่วน `try` ทำให้โค้ดในส่วน `except` ถูกเอ็กซิคิวท์

โครงสร้าง `try...except` จะถูกใช้บ่อย ๆ ในกรณีที่เราต้องการตรวจสอบ user input ยกตัวอย่างเช่น เราอาจจะอยากมีฟังก์ชั่นที่คอยตรวจสอบว่า ค่าที่นำมาใช้เป็นตัวหารเป็นศูนย์หรือไม่ ถ้าใช่ ก็ให้คืนเป็นค่าอื่นที่สมเหตุสมผลแทน อย่างเช่น คืนเป็นค่าที่มีขนาดใหญ่อย่าง 10^{100} แทน ดังนี้

```
In [7]:  def safe_divide(a, b):
             try:
                 return a / b
             except:
                 return 1E100

In [8]:  safe_divide(1, 2)

Out [8]: 0.5

In [9]:  safe_divide(2, 0)

Out [9]: 1e+100
```

อย่างไรก็ตาม โค้ดนี้ก็อาจจะไม่ได้ทำงานตามที่เราคาดหวังไว้ทุกประการ โดยเฉพาะอย่างยิ่งกรณีที่เกิด error ประเภทอื่นขึ้นมา ดังแสดงในตัวอย่างข้างล่างนี้

```
In [10]:  safe_divide (1, '2')

Out [10]: 1e+100
```

การหารเลขจำนวนเต็มด้วยสตริงจะก่อให้เกิด error ประเภท `TypeError` ซึ่งถึงแม้ว่าไม่ใช่ `ZeroDivisionError` แต่ก็จะถูกจับและถูกจัดการโดยโค้ดในส่วน `except` ของเรา ดังนั้นในการเขียนโค้ดในส่วน `except` เราควรระบุให้ชัดเจนว่า เราต้องการจัดการกับ error ประเภทไหน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]:  def safe_divide(a, b):
              try:
                  return a / b
              except ZeroDivisionError:
                  return 1E100

In [12]:  safe_divide(1, 0)

Out [12]: 1e+100

In [13]:  safe_divide(1, '2')

---------------------------------------------------------

TypeError              Traceback (most recent call last)

<ipython-input-15-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')

<ipython-input-13-10b5f0163af8> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100


TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

ณ ตอนนี้ โค้ดเขาเราจะจัดการเฉพาะ error ประเภท `ZeroDivisionError` ส่วน error ประเภทอื่น ๆ เราก็ปล่อยให้มันเกิดไป โดยไม่ทำอะไร

Raising Exceptions: raise

ภาษา Python ได้นิยาม error ประเภทต่าง ๆ และสถานการณ์ที่ทำให้เกิด error ประเภทนั้น ๆ รวมทั้งได้กำหนดนิยามของข้อมูลหรือรายละเอียดเกี่ยวกับ error นั้น ๆ และส่งออกมาให้เราตอนเกิด error และเราก็ได้เห็นมาแล้วว่า รายละเอียดเกี่ยวกับ error ที่ Python แจ้งให้เราทราบนั้น มีประโยชน์เป็นอย่างมาก

เราสามารถนิยามสถานการณ์ที่ทำให้เกิด error ของเราเองได้ ซึ่งในกรณีนี้ก็เช่นเดียวกัน เราควรจะเขียนโค้ดให้แจ้งรายละเอียดเกี่ยวกับ error เมื่อเกิด error ขึ้นมา เพื่อทั้งตัวเราเองและผู้ที่ใช้งานโปรแกรมของเราจะได้เข้าใจถึงสาเหตุที่เกิด error นั้น

ตัวอย่างต่อไปนี้แสดงวิธีการสร้าง error ภายในโปรแกรมของเราเอง ซึ่งตัวอย่างนี้ได้สร้าง error ประเภท `RuntimeError` และแจ้งรายละเอียดเกี่ยวกับ error นี้แบบง่าย ๆ ว่า `my error message`

```
In [14]: raise RuntimeError("my error message")

---------------------------------------------------------

RuntimeError                 Traceback (most recent call last)

<ipython-input-16-c6a4c1ed2f34> in <module>()
----> 1 raise RuntimeError("my error message")


RuntimeError: my error message
```

ถัดไป เราจะมาดูตัวอย่างการนำเอาคำสั่ง `raise` มาใช้ในการสร้าง error ตามสถานการณ์ที่เรากำหนดขึ้นเอง แต่ก่อนอื่นให้เรากลับมาพิจารณาฟังก์ชั่น `fibonacci` ที่เราได้นิยามไว้ก่อนหน้านี้ ซึ่งมีรายละเอียดดังนี้

```
In [15]: def fibonacci(N):
             L = []
             a, b = 0, 1
             while len(L) < N:
                 a, b = b, a + b
                 L.append(a)
             return L
```

ปัญหาอย่างหนึ่งที่อาจจะเกิดขึ้นกับฟังก์ชั่นนี้ คือ การผู้ที่เรียกใช้ฟังก์ชั่นส่งค่าอากิวเมนต์ที่เป็นค่าลบมาให้ ถึงแม้ว่าจะไม่ได้ทำให้เกิด error ตามที่กำหนดไว้ในภาษา Python แต่จากมุมมองของการใช้งานโปรแกรมนี้ เราสามารถกำหนดให้สถานการณ์เช่นนี้เป็นสถานการณ์ที่เกิด error ซึ่งตามธรรมเนียมปฏิบัติแล้ว error อันเนื่องมาจากค่าพารามิเตอร์ที่ไม่ถูกต้องในภาษา Python จะเป็น error ประเภท `ValueError` ดังนั้นเราจึงใช้คำสั่ง `raise` ในการสร้าง error ชนิดนี้ พร้อมทั้งให้ข้อมูลเกี่ยวกับ error นี้ว่า พารามิเตอร์ `N` ของฟังก์ชั่นไม่ได้รองรับค่าที่เป็นค่าลบ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [16]:  def fibonacci(N):
              if N < 0:
                  raise ValueError("N must be non-negative")
              L = []
              a, b = 0, 1
              while len(L) < N:
                  a, b = b, a + b
                  L.append(a)
              return L
```

```
In [17]:  fibonacci(10)

Out [17]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

In [18]:  fibonacci(-10)

---------------------------------------------------------

RuntimeError              Traceback (most recent call last)

<ipython-input-20-3d291499cfa7> in <module>()
----> 1 fibonacci(-10)


<ipython-input-18-01d0cf168d63> in fibonacci(N)
      1 def fibonacci(N):
      2     if N < 0:
----> 3         raise ValueError("N must be non-negative")
      4     L = []
      5     a, b = 0, 1


ValueError: N must be non-negative
```

เมื่อผู้ใช้งานฟังก์ชั่นเข้าใจถึงสาเหตุที่ทำให้เกิด error แล้ว ผู้ใช้งานฟังก์ชั่นอาจจะเลือกที่จะจัดการกับ error นี้โดยการใช้โครงสร้าง `try...except` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [19]: N = -10
         try:
             print("trying this...")
             print(fibonacci(N))
         except ValueError:
             print("Bad value: need to do something else")

         trying this...
         Bad value: need to do something else
```

Diving Deeper into Exceptions

ผู้เขียนจะกล่าวถึงบางเรื่องที่ผู้อ่านอาจจะได้พบเห็นในอนาคต ผู้เขียนจะไม่ลงในรายละเอียด แต่จะเพียงแค่แสดงรูปแบบไวยากรณ์ที่ใช้ในการเขียน เพื่อให้ผู้อ่านสามารถไปค้นคว้าเพิ่มเติมเองได้ในภายหลัง

Accessing the error message

อันที่จริงแล้ว error ก็คือ ออบเจ็คต์ที่สร้างมาจากคลาสที่ได้นิยามไว้แล้วในภาษา Python อย่างเช่น คลาส `ZeroDivisionError` ดังนั้นถ้าเราต้องการใช้งานออบเจ็คต์เหล่านี้โดยตรง อย่างเช่น ถ้าเราต้องการ access ข้อมูลที่เป็นรายละเอียดของ error โดยตรง เราต้องตั้งชื่อให้แก่ออบเจ็คต์ error เหล่านี้ก่อน โดยใช้คีย์เวิร์ด `as`

แล้วตามด้วยชื่อที่เราต้องการตั้ง จากนั้นเราก็สามารถใช้งานออบเจ็คต์เหล่านี้ได้โดยใช้ชื่อที่เราตั้ง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [20]: try:
             x = 1 / 0
         except ZeroDivisionError as err:
             print("Error class is:  ", type(err))
             print("Error message is:", err)

         Error class is:   <class 'ZeroDivisionError'>
         Error message is: division by zero
```

ด้วยรูปแบบการใช้คีย์เวิร์ด `as` นี้ ผู้อ่านสามารถปรับแต่งการจัดการกับ error ให้เหมาะสมกับการใช้งานได้มากยิ่งขึ้น

Defining custom exceptions

นอกเหนือจาก error ที่สร้างมาจากคลาสที่ได้นิยามไว้ในตัวภาษา Python แล้ว เราสามารถนิยามคลาสที่ใช้ในการสร้าง error ของเราเองได้ โดยผ่านกลไกที่เรียกว่า class inheritance ยกตัวอย่างเช่น เราสามารถสร้าง error ประเภท `ValueError` ของเราเอง ที่มีคุณสมบัติเพิ่มเติมไปจาก `ValueError` ดั้งเดิมได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [21]: class MySpecialError(ValueError):
             pass

         raise MySpecialError("here's the message")
---------------------------------------------------------

MySpecialError              Traceback (most recent call last)

<ipython-input-23-92c36e04a9d0> in <module>()
      2     pass
      3
----> 4 raise MySpecialError("here's the message")


MySpecialError: here's the message
```

การทำแบบนี้ ทำให้เราสามารถเขียนโค้ด `try...except` ในการดักจับเฉพาะ error ประเภทที่เรานิยามขึ้นมาเองเท่านั้นได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [22]:
try:
    print("do something")
    raise MySpecialError("[informative error message here]")
except MySpecialError:
    print("do something else")

do something
do something else
```

ผู้อ่านอาจพบว่าการทำแบบนี้มีประโยชน์ เมื่อผู้อ่านได้พัฒนาโค้ดให้เหมาะสมกับงานมากยิ่งขึ้น

try...except...else...finally

นอกเหนือจาก `try` และ `except` แล้ว ในภาษา Python ยังมีคีย์เวิร์ด `else` กับ `finally` ที่สามารถนำมาใช้ในการปรับแต่งการจัดการกับ error เพิ่มเติมได้ โดยมีโครงสร้างดังนี้

```
In [23]: try:
             print("try something here")
         except:
             print("this happens only if it fails")
         else:
             print("this happens only if it succeeds")
         finally:
             print("this happens no matter what")

try something here
this happens only if it succeeds
this happens no matter what
```

ความหมายของการใช้ `else` ในกรณีนี้ น่าจะชัดเจนอยู่แล้ว แต่ในส่วน `finally` อาจจะยังไม่ชัดเจนว่ามีไว้เพื่ออะไร โค้ดในส่วน `finally` นั้น จะถูกเอ็กซิคิวท์เสมอไม่ว่าจะเกิด error หรือไม่ก็ตาม เท่าที่ผู้เขียนเห็นโค้ดส่วน `finally` จะเป็นโค้ดที่ทำการ cleanup ทรัพยากรต่าง ๆ ที่เราใช้ในโปรแกรม อย่างเช่น ถ้ามีไฟล์เปิดค้างไว้อยู่ ก็จะทำการปิดไฟล์ให้ เป็นต้น

Iterators

บ่อยครั้ง งานสำคัญที่เป็นส่วนหนึ่งของการวิเคราะห์ข้อมูลคือ การทำการคำนวณในลักษณะเดิม ๆ ซ้ำแล้วซ้ำอีกในรูปแบบอัตโนมัติ ยกตัวอย่างเช่น ผู้อ่านอาจจะมีตารางรายชื่อที่ผู้อ่านต้องการแยกเป็นส่วนชื่อและส่วนนามสกุล หรืออาจเป็นวันที่ที่ผู้อ่านต้องการจัดฟอร์แมตให้อยู่ในรูปแบบมาตรฐาน เราสามารถเอา Python เข้ามาช่วยในสถานการณ์แบบนี้ได้ ซึ่งสามารถทำได้หลายวิธี แต่ในบทนี้จะแนะนำวิธีที่ใช้ iterator เราได้เห็น iterator มาแล้วตอนที่เราใช้ออบเจ็คต์ `range` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [1]: for i in range(10):
            print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

ใน Python เวอร์ชั่น 3 range ไม่ใช่ลิสต์แต่เป็นออบเจ็คต์ iterable ซึ่งสามารถสร้างสิ่งที่เรียกว่า iterator ได้ ความเข้าใจในเรื่อง iterator เป็นกุญแจสำคัญในการทำความเข้าใจฟังก์ชั่นการทำงาน (functionality) ต่าง ๆ ที่มีประโยชน์ของภาษา Python

Iterating over lists

เราจะเข้าใจ iterator ได้ง่ายที่สุด ถ้าเรามาดูว่าเกิดอะไรขึ้นตอนเราวนลูปลิสต์ อย่างเช่น

```
In [2]: for value in [2, 4, 6, 8, 10]:
            # do some operation
            print(value + 1, end=' ')

3 5 7 9 11
```

เบื้องหลังของคำสั่ง `for val in L` เป็นดังนี้ คือ Python interpreter จะเรียกบิวท์อินฟังก์ชั่น `iter()` (ซึ่งจะไปเรียกเมทธอดพิเศษ `__iter__()` ของออบเจ็คต์ iterable ที่อยู่ข้างหลังคีย์เวิร์ด `in` ซึ่งในกรณีนี้คือ `L` อีกทีหนึ่ง) เพื่อเอาออบเจ็คต์ iterator มาใช้ในการวนลูป ซึ่งเราสามารถทำแบบเดียวกันนี้ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [3]:  iter([2, 4, 6, 8, 10])

Out [3]: <list_iterator at 0x104722400>
```

จากนั้นก็จะเรียกเมทธอดพิเศษ `__next__()` ของออบเจ็คต์ iterator ซึ่งในการเรียกเมทธอดนี้แต่ละครั้งก็จะได้สมาชิกของลิสต์ (ออบเจ็คต์ iterable) ออกมาทีละตัว ซึ่งเราสามารถทำแบบเดียวกันนี้ได้โดยเรียกเมทธอดพิเศษนี้โดยตรงหรือใช้บิวท์อินฟังก์ชั่น `next()` ดังนี้

```
In [4]: I = iter([2, 4, 6, 8, 10])
In [5]: print(next(I))
2
In [6]: print(next(I))
4
In [7]: print(next(I))
6
```

ผู้อ่านอาจจะสงสัยว่า จุดประสงค์ของการ access อ้อม ๆ แบบนี้คืออะไร ทำไมถึงไม่ access สมาชิกของลิสต์ผ่านลิสต์โดยตรง แต่ไป access ผ่าน iterator การที่ทำแบบนี้มีประโยชน์เป็นอย่างมาก เพราะจะช่วยให้ Python สามารถใช้ลูปกับออบเจ็คต์ต่าง ๆ ได้เสมือนกับว่าเป็นลิสต์ทั้ง ๆ ที่ในความเป็นจริงแล้วไม่ใช่

range(): A List Is Not Always a List

บางทีตัวอย่างที่พบบ่อยที่สุดของการ access แบบอ้อม ๆ นี้ ก็คือ ฟังก์ชั่น `range()` ใน Python เวอร์ชั่น 3 (ซึ่งมีชื่อว่า `xrange()` ใน Python เวอร์ชั่น 2) เมื่อเราเรียนฟังก์ชั่น `range()` นี้ มันจะคืนค่าเป็นออบเจ็คต์ `range` กลับมาให้ ซึ่งไม่ใช่ลิสต์ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [8]:  range(10)
Out [8]: range(0, 10)
```

`range` ก็มีออบเจ็คต์ iterator เหมือนกับลิสต์ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [9]:  iter(range(10))
Out [9]: <range_iterator at 0x1045a1810>
```

ดังนั้น เราจึงสามารถใช้งาน `range` ได้เสมือนกับว่าเป็นลิสต์ ดังนี้

```
In [10]: for i in range(10):
             print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
```

ข้อดีของการใช้ iterator คือ จะไม่มีการสร้างลิสต์ ดังนั้นก็จะไม่มีปัญหาเรื่องหน่วยความจำหลักไม่พอเหมือนอย่างในเวอร์ชั่น 2 ที่จะสร้างลิสต์ทั้งตัวขึ้นมาก่อน ซึ่งถ้าเรารันโค้ดข้างล่างนี้โดยใช้ Python เวอร์ชั่น 2 ก็อาจจะเกิดปัญหาดังกล่าวได้

```
In [11]: N = 10 ** 12
         for i in range(N):
             if i >= 10: break
             print(i, end=', ')
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

จากตัวอย่างนี้ ถ้า range สร้างลิสต์ทั้งตัวขึ้นมา ก็จะใช้เนื้อที่ในหน่วยความจำหลักปริมาณมาก ซึ่งเป็นการสิ้นเปลืองเป็นอย่างมาก เพราะว่าค่าที่เราเอาจากลิสต์นี้มาใช้จริง ๆ ในโค้ดนี้ มีเพียงแค่ 10 ค่าแรกเท่านั้น

อันที่จริงแล้วไม่มีเหตุผลใด ๆ เลย ที่ iterator จะต้องมีค่าที่สิ้นสุด ในไลบรารี `itertools` ของ Python มีฟังก์ชั่น `count` ที่ทำหน้าที่เป็น infinite range ดังแสดงในตัวอย่างข้างล่างนี้

```
In [12]: from itertools import count

         for i in count():
             if i >= 10:
                 break
             print(i, end=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

ถ้าเราไม่ใส่คำสั่ง `break` ในตัวอย่างนี้ โค้ดนี้ก็จะวนลูปไปเรื่อย ๆ จนกว่าเราจะขัดจังหวะหรือฆ่าโพรเซสนี้ (อย่างเช่น ด้วยการกดปุ่ม ctrl-C)

Useful Iterators

ในหัวข้อนี้ เราเรียน iterator ที่มีประโยชน์ที่บิวท์อินมากับตัวภาษา Python

enumerate

บ่อยครั้ง เวลาเราวนลูปลิสต์ เราไม่เพียงแต่ต้องการ access สมาชิกของลิสต์เท่านั้น แต่ต้องการใช้ index ด้วย ในกรณีนี้เราอาจจะเขียนโค้ดดังนี้

```
In [13]: L = [2, 4, 6, 8, 10]
         for i in range(len(L)):
             print(i, L[i])

0 2
1 4
2 6
3 8
4 10
```

ถึงแม้ว่าโค้ดนี้จะทำงานได้ แต่เราควรใช้ฟังก์ชั่น `enumerate()` แทน ซึ่งฟังก์ชั่นนี้จะสร้าง `enumerate` iterator ที่จะช่วยให้เราเขียนโค้ดในกรณีนี้ได้สะอาดตายิ่งขึ้น ดังแสดงในตัวอย่างต่อไปนี้

```
In [17]: # find values up to 10 for which x % 2 is zero
         is_even = lambda x: x % 2 == 0
         for val in filter(is_even, range(10)):
             print(val, end=' ')

0 2 4 6 8
```

```
In [14]: for i, val in enumerate(L):
             print(i, val)
```

```
0 2
1 4
2 6
3 8
4 10
```

ซึ่งวิธีการเขียนแบบนี้เป็นวิธีการที่เป็น Pythonic มากกว่า

zip

บางครั้ง เราอาจจะต้องการวงลูปหลาย ๆ ลิสต์พร้อม ๆ กัน ซึ่งเราจะสามารถทำได้โดยใช้ index แต่เราได้เห็นไปในตัวอย่างก่อนหน้านี้แล้วว่า การใช้ index ในการวนลูป ไม่ใช่วิถีของการเขียนโปรแกรมด้วยภาษา Python ดังนั้นในกรณีเช่นนี้เราควรจะใช้ฟังก์ชั่น `zip()` แทน ซึ่งฟังก์ชั่นนี้จะสร้าง `zip` iterator ที่ทำการซิปออบเจ็คต์ iterable (เช่น ลิสต์) เข้าด้วยกัน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [15]: L = [2, 4, 6, 8, 10]
         R = [3, 6, 9, 12, 15]
         for lval, rval in zip(L, R):
             print(lval, rval)
```

```
2 3
4 6
6 9
8 12
10 15
```

เราสามารถซิปออบเจ็คต์ iterable กี่ตัวเข้าด้วยกันก็ได้ แต่ถ้าความยาวของออบเจ็คต์เหล่านี้ไม่เท่ากัน ความยาวของออบเจ็คต์ตัวที่สั้นที่สุดจะเป็นตัวกำหนดความยาวของผลลัพธ์ของการทำ `zip`

map and filter

ฟังก์ชั่น `map()` จะสร้าง `map` iterator ที่เอาฟังก์ชั่นไปทำกับสมาชิกแต่ละตัวของออบเจ็คต์ iterable อย่างเช่น `range` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [16]: # find the first 10 square numbers
         square = lambda x: x ** 2
         for val in map(square, range(10)):
             print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81
```

ฟังก์ชั่น `filter()` จะสร้าง `filter` iterator ที่ทำงานคล้าย ๆ กับ `map` iterator แต่จะเอาเฉพาะสมาชิกที่ทำให้ผลลัพธ์ของฟังก์ชั่นเป็น `True` เท่านั้น ดังแสดงในตัวอย่างต่อไปนี้

ฟังก์ชั่น `map()` และ `filter()` รวมทั้ง `reduce()` (ซึ่งอยู่ในโมดูล `functools` ของ Python) เป็นองค์ประกอบพื้นฐานของรูปแบบการเขียนโปรแกรมเชิงฟังก์ชัน (functional programming)

Iterators as function arguments

ในหัวข้อเรื่อง *args and **kwargs: Flexible Arguments เราได้เรียนการส่งอากิวเมนต์ที่ใช้ * และ ** ให้กับฟังก์ชั่นมาแล้ว ซึ่งในกรณีของ * เราได้เห็นว่าเราสามารถใช้ * กับลิสต์และทูเปิลได้ นอกจากนี้เรายังสามารถใช้กับ * กับ iterator ใด ๆ ก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [18]: print(*range(10))

0 1 2 3 4 5 6 7 8 9
```

เราสามารถเอา * ไปประยุกต์ใช้กับตัวอย่าง `map` ก่อนหน้านี้ ได้ดังนี้

```
In [19]: print(*map(lambda x: x ** 2, range(10)))

0 1 4 9 16 25 36 49 64 81
```

นานมาแล้วได้เคยมีผู้ถามคำถามในฟอรัมของผู้เรียนภาษา Python ว่า ทำไมถึงไม่มีฟังก์ชั่น `unzip()` ที่ทำงานตรงกันข้ามกับฟังก์ชั่น `zip()` ซึ่งถ้าผู้อ่านหยุดคิดสักนิด ผู้อ่านก็จะเข้าใจคำตอบ กุญแจสำคัญของคำตอบนี้อยู่ที่ข้อเท็จจริงที่ว่า ฟังก์ชั่น `zip()` จะจะสร้าง `zip` iterator ที่ทำการซิปออบเจ็คต์ iterable เข้าด้วยกัน ให้ผู้อ่านลองสังเกตโค้ดดังต่อไปนี้

```
In [20]: L1 = (1, 2, 3, 4)
         L2 = ('a', 'b', 'c', 'd')

In [21]: z = zip(L1, L2)
         print(*z)

(1, 'a') (2, 'b') (3, 'c') (4, 'd')

In [22]: z = zip(L1, L2)
         new_L1, new_L2 = zip(*z)
         print(new_L1, new_L2)

(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

ให้ผู้อ่านลองไตร่ตรองโค้ดนี้ดูสักพัก หากผู้อ่านเข้าใจว่าเหตุใดจึงได้ผลลัพธ์อย่างนี้ แสดงว่าผู้อ่านเข้าใจเรื่อง iterator

List Comprehensions

หากผู้อ่านได้มีโอกาสอ่านโค้ด Python มาพอสมควร ผู้อ่านคงจะเคยเจอ list comprehension มาบ้างแล้ว list comprehension เป็นฟีเจอร์หนึ่งในภาษา Python ที่ทำให้เขียนโค้ดได้สั้นและมีประสิทธิภาพ ดังแสดงในตัวอย่างข้างล่างนี้ และผู้เขียนคิดว่าผู้อ่านจะรู้สึกหลงรักฟีเจอร์นี้ ถ้าผู้อ่านยังไม่เคยใช้มันมาก่อน

```
In [1]:  [i for i in range(20) if i % 3 > 0]

Out [1]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

ผลลัพธ์ที่ได้จาก list comprehension นี้ จะเป็นลิสต์ที่ประกอบด้วยตัวเลขที่มีค่าตั้งแต่ 1 ถึง 20 (แต่ไม่รวม 20) และต้องไม่ใช่จำนวนเท่าของเลข 3 ถึงแม้ว่าตัวอย่างนี้อาจจะดูเข้าใจยากในตอนแรก แต่หลังจากที่เราคุ้นเคยกับภาษา Python แล้ว การอ่านหรือเขียน list comprehension จะกลายเป็นธรรมชาติของเราไปเลย

Basic List Comprehensions

list comprehension เป็นวิธีการที่ทำให้เราเขียนคำสั่งลูป `for` ในการสร้างลิสต์ ให้เหลือเพียงบรรทัดเดียว ให้เรามาลองดูตัวอย่างการใช้คำสั่งลูป `for` ในการสร้างลิสต์ที่ประกอบเลขยกกำลังสองของเลขจำนวนเต็ม 12 ตัวแรก ดังนี้

```
In [2]:  L = []
         for n in range(12):
             L.append(n ** 2)
         L

Out [2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

ซึ่งเราสามารถเขียนแทนโค้ดนี้ โดยใช้ list comprehension ได้ดังนี้

```
In [3]:  [n ** 2 for n in range(12)]

Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

เช่นเดียวกับคำสั่งอื่น ๆ ในภาษา Python อีกจำนวนมาก ผู้อ่านสามารถอ่านความหมายของ list comprehension ออกมาเป็นข้อความภาษาอังกฤษธรรมดา ๆ ที่เข้าใจได้ง่ายได้ ซึ่งในกรณีตัวอย่างนี้ เราสามารถอ่านเป็นข้อความภาษาอังกฤษได้ดังนี้ “Construct a list consisting of the square of n for each n up to 12”

รูปแบบการเขียน list comprehension เป็นดังนี้ คือ `[`*`expr`* `for` *`var`* `in` *`iterable`*`]` โดยที่ *`expr`* คือนิพจน์ใด ๆ ก็ได้ ที่เขียนถูกต้องตามไวยากรณ์ของภาษา Python ส่วน *`var`* เป็นชื่อตัวแปร และ *`iterable`* เป็นออบเจ็คต์ใด ๆ ที่เป็นชนิด iterable

Multiple Iteration

บางครั้ง เราต้องการสร้างลิสต์ที่ไม่ได้มาจากค่าเพียงค่าเดียว แต่เป็นสองค่า เราก็สามารถทำได้ง่าย ๆ ด้วยการเพิ่มอีก `for` expression อีกหนึ่งตัวเข้าไปใน list comprehension ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]:  [(i, j) for i in range(2) for j in range(3)]
Out [4]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

ให้ผู้อ่านสังเกตด้วยว่า `for` expression อันที่สองที่ใส่ต่อท้ายเพิ่มเข้าไป จะทำหน้าที่เหมือนเป็นลูปที่ซ้อนอยู่ข้างใน เราสามารถเพิ่มจำนวน `for` expression ที่ต่อท้ายเข้าไปใน list comprehension ได้ตามต้องการ แต่เมื่อเพิ่มเข้าไปจนถึงจุดหนึ่ง มันจะทำให้โค้ดของเราอ่านทำความเข้าใจได้ยาก

Conditionals on the Iterator

เราสามารถใส่เงื่อนไขในการวนลูปแต่ละรอบได้ โดยใส่เพิ่มเข้าไปที่ตอนท้าย ซึ่งเราได้เห็นมาแล้วในตัวอย่างแรกของเรา ให้เรามาพิจารณาตัวอย่างนี้กันอีกครั้งหนึ่ง ซึ่งเป็นตัวอย่างในการสร้างลิสต์ที่ประกอบด้วยตัวเลขที่มีค่าตั้งแต่ 1 ถึง 20 (แต่ไม่รวม 20) และต้องไม่ใช่จำนวนเท่าของเลข 3

```
In [5]:  [val for val in range(20) if val % 3 > 0]
Out [5]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

`val % 3 > 0` เป็น expression ที่จะให้ค่าออกเป็น `True` ยกเว้นกรณีที่ค่า `val` หารด้วย 3 ลงตัว เช่นเดียวกับตัวอย่างก่อนหน้านี้ เราสามารถอ่านความหมายของตัวอย่างนี้ ออกมาเป็นข้อความภาษาอังกฤษธรรมดาได้ดังนี้ “Construct a list of values for each value up to 20, but only if the value is not divisible by 3” หลังจากที่ผู้อ่านรู้สึกคุ้นเคยกับ list comprehension แล้ว ผู้อ่านก็จะสามารถเขียนและทำความเข้าใจได้รวดเร็วกว่า โค้ดที่ให้ผลลัพธ์ออกมาเหมือนกันแต่เขียนโดยใช้คำสั่งลูป `for` แทน ดังแสดงในตัวอย่างข้างล่างนี้

```
In [6]:  L = []
         for val in range(20):
             if val % 3:
                 L.append(val)
         L
Out [6]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Conditionals on the Value

ถ้าผู้อ่านเคยเขียนโปรแกรมภาษา C มาก่อน ผู้อ่านอาจจะคุ้ยเคยกับการเขียนงื่อนไขบรรทัดเดียวโดยใช้โอเปอเรเตอร์ `?` ดังนี้

```
int absval = (val < 0) ? -val : val
```

ภาษา Python ก็มีโครงสร้างที่คล้าย ๆ กับแบบนี้ ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งส่วนใหญ่จะถูกนำไปใช้ภายใน list comprehension, `lambda` ฟังก์ชั่น, หรือภายในโปรแกรมตรงส่วนที่ต้องการมีเงื่อนไขแบบง่าย ๆ

```
In [7]:  val = -10
         val if val >= 0 else -val

Out [7]: 10
```

เราจะเห็นได้ว่าโค้ดนี้มีการทำงานที่เหมือนกับฟังก์ชั่น `abs()` แต่รูปแบบการเขียนโค้ดลักษณะนี้ จะช่วยให้เราทำสิ่งที่น่าสนใจหลาย ๆ อย่างภายใน list comprehension ได้ ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นตัวอย่างที่ค่อนข้างดูซับซ้อนขึ้นกว่าเดิม

```
In [8]:  [val if val % 2 else -val
          for val in range(20) if val % 3]

Out [8]: [1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

ให้ผู้อ่านสังเกตด้วยว่า เราได้มีการขึ้นบรรทัดใหม่ตรง `for` expression ภายใน list comprehension ซึ่งเป็นการเขียนที่ถูกต้องตามภาษา Python และเป็นวิธีการที่เราใช้ในการแบ่ง list comprehension ที่ยาว ๆ เพื่อให้อ่านเข้าใจได้ง่ายขึ้น ให้เรากลับมาดูความหมายของตัวอย่างนี้กัน ตัวอย่างนี้ทำการสร้างลิสต์ โดยตัดตัวเลขที่เป็นจำนวนเท่าของ 3 ทิ้ง และทำให้เลขคู่เป็นค่าลบ

หลังจากที่ผู้อ่านได้เข้าใจเกี่ยวกับ list comprehension แล้ว ผู้เขียนก็จะแนะนำให้ผู้อ่านรู้จัก comprehension ชนิดอื่น ๆ ซึ่งมีรูปแบบการเขียนในลักษณะเดียวกัน แต่แต่ต่างเพียงอย่างเดียว คือ ไม่ใช้เครื่องหมายก้ามปู `[]` แต่จะใช้เป็นเครื่องหมายอื่นแทน

ยกตัวอย่างเช่น ถ้าเราใช้เครื่องหมายปีกกา `{}` แทน ก็จะเป็นการสร้าง set comprehension ดังแสดงในตัวอย่างข้างล่างนี้

```
In [9]:  {n**2 for n in range(12)}

Out [9]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

อย่าลืมว่าเซตจะประกอบด้วยสมาชิกที่ไม่ซ้ำกันเท่านั้น ซึ่งการใช้ set comprehension ก็ไม่ได้ทำให้คุณสมบัตินี้ของเซตที่สร้างขึ้นเปลี่ยนแปลงไป ดังแสดงในตัวอย่างข้างล่างนี้

```
In [10]:  {a % 3 for a in range(1000)}

Out [10]: {0, 1, 2}
```

ถ้าเราปรับโค้ดของ set comprehension เล็กน้อย และเพิ่มเครื่องหมายเครื่องหมายทวิภาค (:) เข้าไป เราก็จะได้ dictionary comprehension ดังแสดงในตัวอย่างข้างล่างนี้

```
In [11]:  {n:n**2 for n in range(6)}
Out [11]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

สุดท้ายนี้ ถ้าเราใช้วงเล็บแทนเครื่องหมายก้ามปู [] เราก็จะได้สิ่งที่เรียกว่า generator expression ดังแสดงในตัวอย่างข้างล่างนี้

```
In [12]:  (n**2 for n in range(12))
Out [12]: <generator object <genexpr> at 0x1027a5a50>
```

generator expression ก็เหมือนกับ list comprehension แต่เพียงแตกต่างกันตรงที่ว่า generator expression จะสร้างสมาชิกออกมาให้ทีละตัว เมื่อมีความต้องการ ในการใช้งานสมาชิกแต่ละตัวนั้น จะไม่ได้สร้างสมาชิกออกมาทีเดียวหมดเหมือน list comprehension เราจะศึกษาเกี่ยวกับเรื่องนี้ในบทถัดไป

Generators

ในบทนี้เราจะเรียนเกี่ยวกับ generator ในภาษา Python ซึ่งจะครอบคลุมเรื่อง generator expression และ generator function

Generator Expressions

บางครั้งเราก็จะสับสนเรื่องความแตกต่างระหว่าง list comprehension และ generator expression ในหัวข้อนี้เราจะมาพิจารณาดูว่าโครงสร้างทั้งสองประเภทนี้แตกต่างกันอย่างไร

List comprehensions use square brackets, while generator expressions use parentheses

list comprehension จะใช้เครื่องหมายก้ามปู `[]` ในขณะที่ generator expression ใช้วงเล็บ `()` ซึ่งตัวอย่างต่อไปนี้เป็นตัวอย่างของ list comprehension

```
In [1]:  [n ** 2 for n in range(12)]
Out [1]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

ในขณะที่ตัวอย่างถัดไปเป็นตัวอย่างของ generator expression

```
In [2]:  (n ** 2 for n in range(12))
Out [2]: <generator object <genexpr> at 0x104a60518>
```

ให้ผู้อ่านสังเกตด้วยว่า เราจะไม่เห็นสมาชิกของ generator expression วิธีการหนึ่งที่เราสามารถใช้ในการดูสมาชิกของ generator expression คือ การส่ง generator expression ไปเป็นอากิวเมนต์ของ `list()` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [3]:  G = (n ** 2 for n in range(12))
         list(G)
Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

A list is a collection of values, while a generator is a recipe for producing values

เมื่อเราสร้างลิสต์โดยใช้ list comprehension สมาชิกของลิสต์จะถูกสร้างออกมาทีเดียวทั้งหมดในหน่วยความจำหลัก แต่ถ้าเราใช้ generator expression เราจะได้ออบเจ็คต์ชนิด generator ที่เปรียบเสมือนกับสูตรที่เอาไปใช้ในการสร้างสมาชิกแต่ละตัว ทั้ง list comprehension และ generator expression ต่างก็เป็น iterable จึงนำเอาไปใช้กับคำสั่งลูป `for` ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121

In [5]: G = (n ** 2 for n in range(12))
        for val in G:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
```

ความแตกต่าง ก็คือ generator expression จะไม่สร้างสมาชิกออกมาจนกว่าจะมีความต้องการใช้งานสมาชิกนั้น ซึ่งไม่เพียงแต่จะนำไปสู่การใช้หน่วยความจำอย่างมีประสิทธิภาพ แต่ยังรวมถึงประสิทธิภาพในการคำนวณด้วย นอกจากนี้ยังหมายความว่า ขนาดหรือจำนวนสมาชิกที่สร้างโดย generator expression จะไม่ถูกจำกัดด้วยขนาดของหน่วยความจำหลักที่มีอยู่เหมือนกับกรณีของลิสต์

เราสามารถใช้ `count` iterator ที่นิยามในโมดูล `itertools` ในการสร้าง generator expression ที่สร้างลำดับตัวเลขที่ไม่สิ้นสุด (infinite sequence) ได้ แต่ก่อนอื่นให้เรามาดูการทำงานของ `count` iterator ในตัวอย่างข้างล่างนี้กันก่อน

```
In [6]:  from itertools import count
         count()

Out [6]: count(0)

In [7]:  for i in count():
             print(i, end=' ')
             if i >= 10: break

0 1 2 3 4 5 6 7 8 9 10
```

เราจะเห็นว่า `count` iterator จะนับไปเรื่อย ๆ จนกว่าเราจะบอกมันจะหยุด ดังนั้นเราสามารถ `count` iterator นี้มาช่วยในการสร้างออบเจ็คต์ generator ที่สร้างลำดับตัวเลขที่ไม่สิ้นสุดได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [8]:
factors = [2, 3, 5, 7]
G = (i for i in count() if all(i % n > 0 for n in factors))
for val in G:
    print(val, end=' ')
    if val > 40: break

1 11 13 17 19 23 29 31 37 41
```

หลังจากที่เห็นผลลัพธ์นี้แล้ว ถ้าเราทำการขยายขนาดของลิสต์นี้อย่างเหมาะสม สิ่งที่เราจะได้ก็คือจุดเริ่มต้นของ prime number generator ที่ใช้อัลกอริทึม Sieve of Eratosthenes เราจะเรียนเพิ่มเติมเกี่ยวกับเรื่องนี้ในอีกสักครู่

A list can be iterated multiple times; a generator expression is single use

เราสามารถใช้งานสมาชิกของลิสต์ตัวเดียวกันนั้นกี่ครั้งก็ได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [9]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')
        print()

        for val in L:
            print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

ในทางตรงกันข้าม เราสามารถใช้งานสมาชิกที่สร้างโดย generator expression ตัวเดียวกันนั้น ได้เพียงแค่ครั้งเดียวเท่านั้น ดังแสดงในตัวอย่างข้างล่างนี้

```
In [10]:  G = (n ** 2 for n in range(12))
          list(G)

Out [10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

In [11]:  list(G)

Out [11]: []
```

คุณสมบัตินี้ของ generator expression มีประโยชน์ตรงที่ว่า เราสามารถหยุดการใช้งานที่สมาชิกตัวปัจจุบันที่สร้างโดย generator expression และกลับมาเริ่มใช้งานต่อที่สมาชิกตัวถัดไปได้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [12]: G = (n**2 for n in range(12))
         for n in G:
             print(n, end=' ')
             if n > 30: break

         print("\ndoing something in between")

         for n in G:
             print(n, end=' ')
```

```
0 1 4 9 16 25 36
doing something in between
49 64 81 100 121
```

ผู้เขียนพบว่า คุณสมบัติดังกล่าวนี้ มีประโยชน์ในกรณีที่เราต้องใช้งานไฟล์ข้อมูลจำนวนหลาย ๆ ไฟล์บนดิสก์ กล่าวคือ เราสามารถวิเคราะห์ข้อมูลในไฟล์หลาย ๆ ไฟล์นี้ในคราวเดียวกันได้ โดยใช้ generator ในการติดตามดูว่าไฟล์ไหนเรายังไม่ได้อ่านข้อมูลเข้ามา

Generator Functions: Using yield

เราได้เห็นในบทที่ผ่านมาแล้วว่า list comprehension จะเหมาะสำหรับใช้ในการสร้างลิสต์ที่ค่อนข้างง่าย ในขณะที่การสร้างลิสต์โดยใช้คำสั่งลูป `for` ตามปกติ จะเหมาะกับการสร้างลิสต์ที่ซับซ้อนมากกว่า ในทำนองเดียวกัน generator expression จะเหมาะสำหรับใช้ในการสร้าง generator ที่ค่อนข้างง่าย ในขณะที่การสร้าง generator โดยใช้ generator function จะเหมาะกับการสร้าง generator ที่ซับซ้อนมากกว่า generator function จะใช้ประโยชน์จากคำสั่ง `yield` (`yield` statement) ให้เรามาลองเปรียบเทียบการสร้างลิสต์และ generator โดยใช้ทั้งสองวิธีการดังกล่างนี้กัน

ตัวอย่างนี้แสดงการสร้างลิสต์โดยใช้ list comprehension และคำสั่งลูป `for`

```
In [13]: L1 = [n ** 2 for n in range(12)]

         L2 = []
         for n in range(12):
             L2.append(n ** 2)

         print(L1)
         print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

ตัวอย่างถัดไปแสดงการสร้าง generator โดยใช้ generator expression และ generator function

```
In [14]: G1 = (n ** 2 for n in range(12))

         def gen():
             for n in range(12):
                 yield n ** 2

         G2 = gen()
         print(*G1)
         print(*G2)
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

generator function เป็นฟังก์ชั่นที่ไม่ได้ใช้ `return` แต่ใช้ `yield` ในการคืนค่าแทน เมื่อเราเรียก generator function เราจะได้ออบเจ็คต์ generator ซึ่งจะสร้างและส่งคืนค่ามาให้เราที่ละค่า ซึ่งในตัวอย่างนี้ จะคืนค่าที่สร้างขึ้นในแต่ละรอบของการวนลูปมาให้ และในทำนองเดียวกันกับ generator expression สถานะของออบเจ็คต์ generator ที่สร้างโดย generator function จะถูกจดจำไว้ ทำให้สามารถคืนค่าที่อยู่ในลำดับถัด ๆ ไปได้ ซึ่งในตัวอย่างนี้ ก็คือ การคืนค่าที่สร้างขึ้นในลูปรอบถัด ๆ แต่ถ้าเราอยากได้ค่าที่เริ่มใหม่ตั้งแต่ต้น เราต้องเรียก generator function อีกครั้ง เพื่อให้สร้างออบเจ็คต์ generator ตัวใหม่ขึ้นมา

Example: Prime Number Generator

ในหัวข้อนี้ ผู้เขียนจะยกตัวอย่าง generator function ที่ผู้เขียนชื่นชอบ ซึ่งเป็นฟังก์ชันสำหรับสร้างชุดของเลขจำนวนเฉพาะที่ไม่จำกับขอบเขต และอัลกอริทึมที่จะใช้ในการสร้าง คือ Sieve of Eratosthenes ซึ่งมีการทำงานดังนี้

```
In [15]: # Generate a list of candidates
         L = [n for n in range(2, 40)]
         print(L)

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, \
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,  \
30, 31, 32, 33, 34, 35, 36, 37, 38, 39]

In [16]: # Remove all multiples of the first value
         L = [n for n in L if n == L[0] or n % L[0] > 0]
         print(L)

[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, \
29, 31, 33, 35, 37, 39]

In [17]: # Remove all multiples of the second value
         L = [n for n in L if n == L[1] or n % L[1] > 0]
         print(L)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37]

In [18]: # Remove all multiples of the third value
         L = [n for n in L if n == L[2] or n % L[2] > 0]
         print(L)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

หากเราทำซ้ำขั้นตอนนี้หลายครั้งอย่างเพียงพอกับลิสต์ที่มีขนาดใหญ่พอ เราก็จะสามารถสร้างเลขจำนวนเฉพาะซึ่งมีจำนวนมากตามที่เราต้องการได้

เมื่อเรานำตรรกะการคิดนี้ไปใช้ใน generator function เราสามารถเขียนโค้ดได้ดังนี้

```
In [19]: def gen_primes(N):
             """Generate primes up to N"""
             primes = set()
             for n in range(2, N):
                 if all(n % p > 0 for p in primes):
                     primes.add(n)
                     yield n

         print(*gen_primes(70))

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
```

ถึงแม้ว่าโค้ดตัวอย่างนี้จะไม่ใช่การอิมพลีเมนต์อัลกอริทึม Sieve of Eratosthenes ที่มีประสิทธิภาพมากที่สุด แต่อย่างน้อยก็แสดงให้เห็นว่า เราสามารถนำเอา generator function ไปใช้ในการสร้างลำดับที่ซับซ้อนมากขึ้นได้ง่ายเพียงใด

Modules and Packages

ฟีเจอร์หนึ่งที่ทำให้ภาษา Python เป็นภาษาที่มีประโยชน์กับงานหลากหลายประเภท คือ การที่ภาษา Python มาพร้อมกับไลบรารีมาตรฐาน (standard library) ที่ประกอบด้วยเครื่องมือที่มีประโยชน์สำหรับงานหลากหลายประเภท นอกจากนี้ยังมีกลุ่มนักพัฒนาที่เป็น third party หลากหลายกลุ่มที่ได้พัฒนาเครื่องมือและแพ็คเกจจำนวนมากที่มีฟังก์ชั่นการทำงานที่พิเศษหรือเพิ่มเติมจากไลบรารีมาตรฐาน ในบทนี้เราจะเรียนการโหลดโมดูลไลบรารีมาตรฐาน (standard library module), เครื่องมือที่ใช้ในการติดตั้งโมดูลของ third party และวิธีการสร้างโมดูลของเราเอง

Loading Modules: the import Statement

เราสามารถโหลดโมดูลไลบรารีมาตรฐานและโมดูลของ third party ได้ โดยใช้คำสั่ง `import` (`import` statement) ซึ่งมีอยู่สองสามวิธี ผู้เขียนจะแนะนำแต่ละวิธีโดยสังเขป โดยจะเริ่มต้นจากวิธีที่อยากจะให้ใช้มากที่สุดไปจนถึงวิธีที่อยากจะให้ใช้น้อยที่สุด

Explicit module import

การโหลดโมดูลด้วยวิธีนี้ จะรักษาสถานภาพของชื่อต่าง ๆ ที่เขียนอยู่ในโมดูล (เช่น ชื่อของตัวแปรและฟังก์ชั่น) ให้ยังคงอยู่ภายใน namespace ของโมดูลนั้น จากนั้นเราสามารถใช้งานโค้ดภายโมดูลได้ โดยการใช้ชื่อโมดูล คั่นด้วยจุด (`.`) แล้วตามด้วยชื่อ (เช่น ชื่อของตัวแปรและฟังก์ชั่น) ภายใน namespace ของโมดูลนั้น ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นตัวอย่างการโหลดโมดูล `math` จากไลบรารีมาตรฐาน และเรียกใช้ฟังก์ชั่น `cos` ของโมดูลนี้ เพื่อหาค่า cosine ของ pi

```
In [1]:  import math
         math.cos(math.pi)

Out [1]: -1.0
```

Explicit module import by alias

ในกรณีที่ชื่อโมดูลยาว ๆ มันจะเป็นการไม่สะดวกที่เราจะต้องใช้ชื่อโมดูลที่ยาวนั้นทุกครั้ง ในการเรียกใช้โค้ดของโมดูล ด้วยสาเหตุนี้ เราจึงนิยมใช้คำสั่ง `import` ในรูปแบบ `import ... as ...` เพื่อสร้างชื่อที่สั้นลงเพื่อใช้แทนชื่อจริงของโมดูล ยกตัวอย่างเช่น NumPy (Numerical Python) ซึ่งเป็นแพ็คเกจของ third party ที่เป็นที่นิยมใช้กันอย่างแพร่หลายในงานทางด้านวิทยาการข้อมูล ก็มักถูกใช้ด้วยชื่อว่า `np` แทนชื่อ `numpy` ดังแสดงในตัวอย่างต่อไปนี้

```
In [2]:  import numpy as np
         np.cos(np.pi)

Out [2]: -1.0
```

Explicit import of module contents

ในบางครั้ง เราอาจจะไม่อยากโหลดโมดูลเข้ามาทั้ง namespace แต่อยากจะโหลดเพียงบางส่วนเท่านั้น เราสามารถทำได้โดยใช้รูปแบบ `from … import …` ยกตัวอย่างเช่น เราสามารถโหลด เฉพาะฟังก์ชั่น `cos` และค่าคงที่ `pi` จากโมดูล `math` ได้ดังนี้

```
In [3]:  from math import cos, pi
         cos(pi)

Out [3]: -1.0
```

Importing from Python's Standard Library

บางครั้งการโหลดทั้งโมดูลให้เข้ามาอยู่ภายใน namespace ของโปรแกรมของเราก็มีประโยชน์ ซึ่งสามารถทำได้โดยการใช้รูปแบบ `from … import *` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [4]:  from math import *
         sin(pi) ** 2 + cos(pi) ** 2

Out [4]: 1.0
```

แต่อย่างไรก็ตาม เราควรใช้รูปแบบนี้เท่าที่จำเป็นเท่านั้น เพราะว่าการโหลดแบบนี้ อาจจะทำให้เกิดการ overwrite ชื่อ (เช่น ชื่อของฟังก์ชั่นและตัวแปร) โดยไม่ได้ตั้งใจและยากที่จะระบุว่ามีอะไรเปลี่ยนแปลงไปบ้าง

ยกตัวอย่างเช่น Python มีฟังก์ชั่น `sum` ที่บิวท์อินมากับตัวภาษา เราสามารถใช้ฟังก์ชั่น `help` ซึ่งเป็นบิวท์อินฟังก์ชั่นเหมือนกันในการดูคำอธิบายเกี่ยวกับฟังก์ชั่น `sum` ได้ ดังนี้

```
In [5]: help(sum)

        Help on built-in function sum in module builtins:

        sum(iterable[, start]) -> value

        Return the sum of an iterable of numbers
        (NOT strings) plus the value of parameter
        'start' (which defaults to 0).
        When the iterable is empty, return start.
```

เราจะเห็นว่า เราสามารถใช้ฟังก์ชั่นนี้ในการหาผลบวกของลำดับ โดยที่เราสามารถกำหนดค่าเริ่มต้น `start` ได้ ซึ่งในกรณีตัวอย่างของเรา จะให้ค่าเป็น `-1` ดังนี้

```
In [6]:  sum(range(5), -1)

Out [6]: 9
```

มาถึงตรงนี้ ให้เรามาลองสังเกตดูว่าจะเกิดอะไรขึ้น ถ้าเราเรียกฟังก์ชั่นนี้อีกครั้งหลังจากที่เราทำ `import * จากโมดูล numpy`

```
In [7]:  from numpy import *
In [8]:  sum(range(5), -1)
Out [8]: 10
```

เราจะเห็นว่า ผลลัพธ์ต่างจากเดิมไปหนึ่ง สาเหตุที่เป็นเช่นนี้เพราะว่า การทำ `import *` ทำให้ชื่อของฟังก์ชั่น `sum` ที่มากับตัวภาษา Python ถูก overwrite ด้วยชื่อของฟังก์ชั่น `sum` ของ `numpy` ดังนั้นเมื่อเราเรียกฟังก์ชั่น sum ในตัวอย่างนี้ ก็จะกลายเป็นฟังก์ชั่น `sum` ของ `numpy` ซึ่งมี signature ของฟังก์ชั่นที่แตกต่างออกไป กล่าวคือ ฟังก์ชั่น sum ที่มากับตัวภาษา Python ในตัวอย่างก่อนหน้านี้ จะทำการหาผลบวกของลำดับที่เริ่มต้นด้วย `-1` ในขณะที่ฟังก์ชั่น `sum` ของ `numpy` ในตัวอย่างนี้ จะหาผลบวกตาม axis สุดท้าย (ซึ่งระบุโดย `-1`) สถานการณ์แบบนี้อาจเกิดขึ้นได้ หากเราไม่ระมัดระวังเวลาใช้ `import *` ดังนั้นจึงเป็นการดีที่สุดที่จะหลีกเลี่ยงการ import วิธีนี้ เว้นเสียแต่ผู้อ่านเข้าใจผลลัพธ์ที่จะเกิดขึ้นอย่างชัดเจน

Importing from Python's Standard Library

ไลบรารีมาตรฐานของภาษา Python ประกอบด้วยโมดูลที่มีประโยชน์มากมาย ซึ่งผู้อ่านสามารถอ่านเพิ่มเติมได้ที่ https://docs.python.org/3/library/ นอกจากนี้ เราสามารถโหลดโมดูลเหล่านี้โดยใช้คำสั่ง `import` จากนั้นใช้ฟังก์ชั่น `help` ในการศึกษาข้อมูลเพิ่มเติมได้ ผู้เขียนจะแนะนำโมดูลส่วนหนึ่งที่ผู้อ่านอาจจะสนใจเรียนรู้และศึกษาเพิ่มเติม ดังนี้

- `os` และ `sys` เครื่องมือสำหรับเชื่อมต่อกับระบบปฏิบัติการเพื่อใช้งานไฟล์และไดเรกทอรีและเอ็กซิคิวท์คำสั่งของ shell
- `math` และ `cmath` ฟังก์ชันทางคณิตศาสตร์และการดำเนินการกับจำนวนจริงและจำนวนเชิงซ้อน
- `itertools` เครื่องมือสำหรับสร้างและใช้งาน iterator และ generator
- `functools` เครื่องมือที่ช่วยในการเขียนโปรแกรมเชิงฟังก์ชัน
- `random` เครื่องมือสำหรับสร้างเลขสุ่มเทียม (pseudorandom number)
- `pickle` เครื่องมือสำหรับ object persistence ซึ่งได้แก่ การจัดเก็บออบเจ็คต์และโหลดออบเจ็คต์จากดิสก์
- `json` และ `csv` เครื่องมือสำหรับอ่านไฟล์ในรูป JSON และ CSV
- `urllib` เครื่องมือสำหรับทำ HTTP และงานอื่น ๆ ที่เกี่ยวกับ web

ผู้อ่านสามารถศึกษาข้อมูลเกี่ยวกับโมดูลเหล่านี้หรือโมดูลอื่น ๆ อีกมากมายได้ที่ https://docs.python.org/3/library/

Importing from Third-Party Modules

สิ่งหนึ่งที่ทำให้ภาษา Python เป็นภาษาที่มีประโยชน์อย่างมากโดยเฉพาะอย่างยิ่งในวงการวิทยาการข้อมูล คือ โมดูลของ third party เราสามารถโหลดโมดูลเหล่านี้ได้ ในลักษณะเดียวกันกับโมดูลไลบรารีมาตรฐาน แต่เราต้องติดตั้งโมดูลเหล่านี้ก่อนทำการโหลด โดยปกติโมดูลของ third party จะถูกขึ้นทะเบียนเพื่อให้ดาวน์โหลดเอาไปใช้งานได้จาก Python Package Index (เรียกสั้น ๆ ว่า PyPI) ซึ่งอยู่ที่ http://pypi.python.org/ และเพื่อเป็นการอำนวยความสะดวกให้แก่ผู้เขียนโปรแกรม Python เวอร์ชั่น 3 มาพร้อมกับโปรแกรมที่เรียกว่า `pip` (คำย่อแบบ recursive หมายถึง “pip installs packages”) ซึ่งจะทำการดาวน์โหลดแพ็คเกจที่อยู่ที่ PyPI พร้อมทั้งติดตั้งให้อัตโนมัติ (ถ้าผู้อ่านใช้ Python เวอร์ชั่น 2 ผู้อ่านต้องทำการติดตั้ง `pip` เอง)

ยกตัวอย่างเช่น ถ้าผู้อ่านต้องการติดตั้งแพ็คเกจ `supersmoother` ที่ผู้เขียนได้เขียนขึ้นมา สิ่งที่ผู้อ่านต้องทำมีเพียงแค่พิมพ์คำสั่งที่คอมมานด์ไลน์ ดังนี้

```
$ pip install supersmoother
```

ซอร์สโค้ด (source code) ของแพ็คเกจนี้จะถูกดาวน์โหลดจาก PyPI และแพ็คเกจนี้จะถูกติดตั้งให้โดยอัตโนมัติ (ภายใต้สมมติฐานว่าผู้อ่านมีสิทธิในการกระทำเช่นนี้บนเครื่องคอมพิวเตอร์ที่ผู้อ่านกำลังใช้งานอยู่)

ผู้อ่านสามารถศึกษาข้อมูลเกี่ยวกับ PyPI และการ install โดยใช้ `pip` เพิ่มเติมได้ที่ http://pypi.python.org/

String Manipulation and Regular Expressions

จุดหนึ่งที่ภาษา Python โดดเด่นเป็นอย่างมาก คือ เรื่องการจัดการสตริง ในบทนี้เราจะเรียนเกี่ยวกับสตริงเมทธอดและการจัด format และเรื่อง regular expression ซึ่งมีประโยชน์เป็นอย่างมาก การจัดการสตริงดังกล่าวมักเกิดขึ้นในบริบทของงานด้านวิทยาการข้อมูลและเป็นข้อดีอย่างหนึ่งของภาษา Python ในบริบทนี้

เราสามารถนิยามหรือสร้างสตริงโดยการใช้อัญประกาศเดี่ยว (single quotes) หรืออัญประกาศคู่ (double quotes) ก็ได้ (ซึ่งจะได้ผลลัพธ์ออกมาเหมือนกัน) ดังแสดงในตัวอย่างข้างล่างนี้

```
In [1]:  x = 'a string'
         y = "a string"
         x == y

Out [1]: True
```

นอกจากนี้ เรายังสามารถสร้างสตริงที่ประกอบด้วยหลายบรรทัดได้โดยใช้ triple quote ดังแสดงในตัวอย่างข้างล่างนี้

```
In [2]: multiline = """
        one
        two
        three
        """
```

ถัดไป ผู้เขียนจะแนะเกี่ยวกับเครื่องมือที่ใช้ในการจัดการสตริงของภาษา Python โดยสังเขป

Simple String Manipulation in Python

เราสามารถใช้สตริงเมทธอดที่มากับตัวภาษา Python ในการจัดการสตริงระดับพื้นฐานได้ง่ายมาก หากผู้อ่านเคยใช้งานภาษา C หรือภาษาระดับต่ำอื่น ๆ มาก่อน ผู้อ่านอาจจะประหลาดใจมากกับความเรียบง่ายสตริงเมทธอดในภาษา Python

Formatting strings: Adjusting case

Python มีเมทธอดที่ช่วยให้เราปรับตัวอักษรเล็กใหญ่ของสตริงได้ง่าย ๆ หลายเมทธอดดังต่อไปนี้ ได้แก่ `upper()`, `lower()`, `capitalize()`, `title()` และ `swapcase()` และเราจะมาดูตัวอย่างการใช้เมทธอดเหล่านี้กับสตริงที่ดูยุ่ง ๆ ดังนี้

```
In [3]: fox = "tHe qUICk bROWn fOx."
```

เราสามารถใช้เมทธอด `upper()` หรือ `lower()` ในการทำให้สตริงเป็นตัวอักษรตัวใหญ่หรือตัวเล็กทั้งหมดได้ ดังนี้

```
In [4]:  fox.upper()
Out [4]: 'THE QUICK BROWN FOX.'
In [5]:  fox.lower()
Out [5]: 'the quick brown fox.'
```

รูปแบบการ format สตริงที่มักจะจำเป็นอยู่เสมอ คือ การทำให้ตัวอักษรตัวแรกของแต่ละคำหรือตัวอักษรตัวแรกของแต่ละประโยคเป็นตัวอักษรใหญ่ ซึ่งเราสามารถทำได้โดยการใช้เมทธอด `title()` หรือ `capitalize()` ดังนี้

```
In [6]:  fox.title()
Out [6]: 'The Quick Brown Fox.'
In [7]:  fox.capitalize()
Out [7]: 'The quick brown fox.'
```

เราสามารถทำให้ตัวอักษรในสตริงสลับกันระหว่างตัวอักษรตัวเล็กและตัวใหญ่ได้โดยการใช้เมทธอด `swapcase()` ดังนี้

```
In [8]:  fox.swapcase()
Out [8]: 'ThE QuicK BrowN FoX.'
```

Formatting strings: Adding and removing spaces

รูปแบบการ format สตริงอีกรูปแบบหนึ่งที่มักจะจำเป็นอยู่เสมอ คือ การลบช่องว่าง (หรืออักขระตัวอื่น ๆ) จากต้นหรือจากท้ายของสตริง เมทธอดพื้นฐานที่ใช้ในการลบอักขระ คือ `strip()` ซึ่งจะใช้ในการลบช่องว่างจากต้นหรือจากท้ายของสตริงออก ดังนี้

```
In [9]:  line = '        this is the content        '
         line.strip()
Out [9]: 'this is the content'
```

เราสามารถเอาเฉพาะช่องว่างที่อยู่ทางด้านขวาสุดหรือทางด้านซ้ายสุดออกก็ได้ โดยการใช้ `rstrip()` หรือ `lstrip()` ดังนี้

```
In [10]:  line.rstrip()
Out [10]: '        this is the content'
In [11]:  line.lstrip()
Out [11]: 'this is the content        '
```

นอกเหนือจากช่องว่างแล้ว เราสามารถเอาอักขระอื่นออกก็ได้ โดยการระบุอักขระนั้นที่เมทธอด `strip()` ดังนี้

```
In [12]:  num = "000000000000435"
          num.strip('0')

Out [12]: '435'
```

ในทางตรงกันข้าม เราสามารถใส่ช่องว่างหรืออักขระอื่น ๆ เข้าไปเป็นส่วนหนึ่งของสตริงได้ โดยใช้เมทธอด `center()`, `ljust()` หรือ `rjust()`

ยกตัวอย่างเช่น เราสามารถใช้เมทธอด `center()` ในการจัดกึ่งกลางสตริงภายในจำนวนช่องว่างที่กำหนด ดังนี้

```
In [13]:  line = "this is the content"
          line.center(30)

Out [13]: '     this is the content      '
```

ในทำนองเดียวกัน `ljust()` และ `rjust()` จะจัดสตริงให้ชิดซ้ายหรือชิดขวาภายในจำนวนช่องว่างที่กำหนด ดังนี้

```
In [14]:  line.ljust(30)

Out [14]: 'this is the content           '

In [15]:  line.rjust(30)

Out [15]: '           this is the content'
```

เมทธอดเหล่านี้ ยังสามารถรับตัวอักษรใด ๆ ก็ได้ เพื่อเติมเต็มช่องว่าง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [16]:  '435'.rjust(10, '0')

Out [16]: '0000000435'
```

เนื่องจากการเติมศูนย์เป็นความต้องการปกติทั่วไป Python จึงได้จัดเตรียมเมทธอด `zfill()` ไว้ให้ใช้สำหรับกรณีดังกล่าว ดังแสดงในตัวอย่างข้างล่างนี้

```
In [17]:  '435'.zfill(10)

Out [17]: '0000000435'
```

Finding and replacing substrings

ถ้าหากผู้อ่านต้องการหาว่ามีอักขระบางตัวในสตริงไหม ผู้อ่านสามารถทำได้โดยใช้เมทธอดดังต่อไปนี้ ได้แก่ `find()/rfind()`, `index()/rindex()` และ `replace()`

`find()` และ `index()` มีความคล้ายคลึงกันมาก โดยที่เมทธอดทั้งสองนี้จะค้นหาการเกิดขึ้นครั้งแรก (first occurrence) ของอักขระหรือสตริงย่อยภายในสตริงและจะคืนค่า index ของสตริงย่อยกลับมาให้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [18]:  line = 'the quick brown fox jumped over a lazy dog'
          line.find('fox')

Out [18]: 16

In [19]:  line.index('fox')

Out [19]: 16
```

ความแตกต่างเพียงอย่างเดียวระหว่าง `find()` และ `index()` คือ พฤติกรรมของเมทธอดทั้งสองนี้เมื่อหาอักขระหรือสตริงย่อยที่ต้องการไม่เจอ ซึ่งในกรณีนี้ `find()` จะคืนค่ากลับมาให้เป็น -1 ในขณะที่ `index()` จะเกิด error ประเภท `ValueError` ดังแสดงในตัวอย่างข้างล่างนี้

```
In [20]:  line.find('bear')

Out [20]: -1

In [21]: line.index('bear')

---------------------------------------------------------

ValueError              Traceback (most recent call last)

<ipython-input-21-4cbe6ee9b0eb> in <module>()
----> 1 line.index('bear')


ValueError: substring not found
```

ส่วน `rfind()` และ `rindex()` ก็จะทำงานในลักษณะเดียวกันกับ `find()` และ `index()` แต่แตกต่างกันตรงเพียงที่ว่าจะเริ่มหาจากอักขระหรือสตริงย่อยที่ต้องการจากทางขวาของสตริง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [22]:  line.rfind('a')

Out [22]: 35
```

สำหรับกรณีพิเศษอย่างกรณีที่เราต้องการตรวจสอบว่า สตริงย่อยที่เราต้องการหานั้นอยู่ที่ต้นหรือท้ายของสตริง Python ได้เตรียมเมทธอด `startswith()` และ `endswith()` ไว้ให้ใช้ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [23]:  line.endswith('dog')

Out [23]: True

In [24]:  line.startswith('fox')

Out [24]: False
```

นอกจากนี้ เราสามารถใช้ `replace()` ในการแทนที่สตริงย่อยด้วยสตริงชุดใหม่ได้ ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นการแทนที่ `'brown'` ด้วย `'red'`

```
In [25]:  line.replace('brown', 'red')
Out [25]: 'the quick red fox jumped over a lazy dog'
```

`replace()` จะแทนที่สตริงย่อยทั้งหมดที่มีอยู่ด้วยสตริงย่อยชุดใหม่ และคืนค่ากลับมาเป็นสตริงตัวใหม่ ดังแสดงในตัวอย่างข้างล่างนี้

```
In [26]:  line.replace('o', '--')
Out [26]: 'the quick br--wn f--x jumped --ver a lazy d--g'
```

Splitting and partitioning strings

ถ้าหากว่า ผู้อ่านต้องการค้นหาสตริงย่อยแล้วแบ่งสตริงตรงตำแหน่งที่เจอสตริงย่อยนั้น เมทธอด `partition()` และ `split()` คือสิ่งที่ผู้อ่านกำลังมองห เราจะเรียกสตริงที่ต้องการค้นหาและใช้เป็นตัวแบ่งว่า จุดแบ่ง (split-point)

เมทธอด `partition()` จะคืนทูเปิลที่ประกอบด้วยสมาชิก 3 ตัว คือ สตริงย่อยก่อนจุดแบ่ง, ตัวจุดแบ่ง (ซึ่งก็คือสตริงย่อยที่เราใช้ค้นหา) และสตริงย่อยหลังจุดแบ่ง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [27]:  line.partition('fox')
Out [27]: ('the quick brown ', 'fox', ' jumped over a lazy dog')
```

เมทธอด `rpartition()` ก็จะทำงานในลักษณะคล้ายกัน แต่จะค้นหาจุดแบ่งจากด้านขวา

เมทธอด `split()` อาจจะมีประโยชน์มากกว่า กล่าวคือ เมทธอดนี้จะค้นหาจุดแบ่งทุกตัวในสตริง จากนั้นจะส่งคืนสตริงย่อยที่อยู่ระหว่างจุดแบ่งทุกตัวนั้นกลับมาให้ ค่าดีฟอลต์ของจุดแบ่งคือ ช่องว่าง (white space) ดังนั้น `split()` จะคืนลิสต์ที่ประกอบไปด้วยคำที่มีอยู่ในสตริง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [28]:  line.split()
Out [28]: ['the', 'quick', 'brown', 'fox', 'jumped',\
           'over', 'a', 'lazy', 'dog']
```

เมทธอดที่เกี่ยวข้องกับเมทธอดนี้ คือ `splitlines()` ซึ่งมีตัวแบ่งเป็นอักขระขึ้นบรรทัดใหม่ (newline character) เราจะลองใช้เมทธอดนี้กับไฮกุ (haiku) ที่เป็นที่นิยมในศตวรรษที่ 17 ซึ่งประพันธ์โดย Matsuo Bashō ดังแสดงในตัวอย่างต่อไปนี้

```
In [29]: haiku = """matsushima-ya
         aah matsushima-ya
         matsushima-ya"""

         haiku.splitlines()

['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

ถ้าหากผู้อ่านต้องการผลลัพธ์ที่ตรงกันข้ามกับ `split()` ผู้อ่านสามารถใช้เมทธอด `join()` ได้ ซึ่งจะคืนค่าเป็นสตริงที่สร้างจากจุดแบ่งและ iterable ดังแสดงในตัวอย่างข้างล่างนี้

```
In [30]:  '--'.join(['1', '2', '3'])

Out [30]: '1--2--3'
```

รูปแบบที่ใช้งานกันบ่อยกับเมทธอด `join()` คือ การใช้อักขระพิเศษ `\n` (อักขระขึ้นบรรทัดใหม่) เพื่อรวมบรรทัดที่แยกไว้ก่อนหน้านี้ให้กลับมาเป็นสตริงอีกครั้ง ดังแสดงในตัวอย่างข้างล่างนี้

```
In [31]: print("\n".join(['matsushima-ya', 'aah matsushima-ya',
                          'matsushima-ya']))

matsushima-ya
aah matsushima-ya
matsushima-ya
```

Format Strings

เราได้เรียนรู้การใช้สตริงเมทธอดหลาย ๆ ตัวในหัวข้อก่อนหน้านี้ไปแล้ว ในการจัดการสตริงให้เป็นรูปแบบที่ต้องการ ในหัวข้อนี้เราจะเรียนการจัดการ string representation ของข้อมูลประเภทอื่น ๆ

เราสามารถสร้าง string representation ของข้อมูลประเภทอื่น ๆ ได้ โดยการใช้ `str()` ดังแสงในตัวอย่างข้างล่างนี้

```
In [32]:  pi = 3.14159
          str(pi)

Out [32]: '3.14159'
```

สำหรับรูปแบบที่ซับซ้อนมากขึ้น ผู้อ่านอาจจะใช้การบวกกันของสตริง ที่ได้เคยเรียนมาแล้วในบทเรื่อง Basic Python Semantics: Operators ดังแสดงในตัวอย่างข้างล่างนี้

```
In [33]:  "The value of pi is " + str(pi)

Out [33]: 'The value of pi is 3.14159'
```

วิธีการที่ยืดหยุ่นกว่านี้ คือ คือการใช้ format string ซึ่งเป็นสตริงที่มีเครื่องหมายพิเศษ (ระบุโดยใช้เครื่องหมายปีกกา {}) เพื่อใช้ในการจัดรูปแบบของค่าที่ใส่เข้าไปในปีกกา {} ให้อยู่ในรูปแบบของสตริงที่ต้องการ ดังนี้

```
In [34]:  "The value of pi is {}".format(pi)

Out [34]: 'The value of pi is 3.14159'
```

ภายในเครื่องหมายปีก {} เราสามารถใส่ข้อมูลเกี่ยวกับสิ่งที่เราต้องการให้ปรากฏที่ตรงนั้น ยกตัวอย่างเช่น ถ้าเราใส่ตัวเลข ก็จะหมายถึง index ของอากิวเมนต์ของเมทธอด `format()`

```
In [35]:
"""First letter: {0}. Last letter: {1}.""".format('A', 'Z')

Out [35]: 'First letter: A. Last letter: Z.'
```

ถ้าเราใส่สตริง ก็จะหมายถึงคีย์เวิร์ดอากิวเมนต์ของเมทธอด `format()`

```
In [36]:
"""First: {first}. Last: {last}.""".format(last='Z', first='A')

Out [36]: 'First: A. Last: Z.'
```

สุดท้ายนี้ สำหรับอินพุตที่เป็นตัวเลข ผู้อ่านสามารถใส่ format code ภายในเครื่องหมายปีกกา {} เพื่อระบุวิธีการแปลงค่าให้เป็นสตริงได้ ยกตัวอย่างเช่น ถ้าเราต้องการพิมพ์ตัวเลขให้อยู่ในรูปเลขทศนิยมที่มีตัวเลขหลังจุดทศนิยมจำนวน 3 ตัว เราสามารถทำได้ดังนี้

```
In [37]:  "pi = {0:.3f}".format(pi)

Out [37]: 'pi = 3.142'
```

`0` ในที่นี้หมายถึง index ของอากิวเมนต์ ดังที่กล่าวไปแล้วในข้างต้น ส่วนเครื่องหมายทวิภาค (`:`) ใช้ระบุว่ามี format code ตามหลังมา ส่วน `.3f` ใช้ระบุตัวเลขหลังจุดทศนิยมจำนวน 3 ตัว

การจัดรูปแบบสตริงวิธีนี้จะมีความยืดหยุ่นมากและตัวอย่างที่ยกมาทั้งหมดนี้ก็เป็นเพียงแค่เปลือกนอกเท่านั้น ยังมีรายละเอียดอีกมากซึ่งสามารถศึกษาเพิ่มเติมได้ในหัวข้อ Format Specification ที่ https://docs.python.org/3/library/string.html#formatspec

A Preview of Data Science Tools

ถ้าหากผู้อ่านต้องการต่อยอดการใช้งาน Python สำหรับงานทางด้านการคำนวณทางวิทยาศาสตร์หรืองานทางด้านวิทยาการข้อมูล จะมีแพ็คเกจบางตัวที่จะทำให้ชีวิตของผู้อ่านง่ายขึ้นมาก บทนี้จะแนะนำและแสดงตัวอย่างแพ็คเกจที่สำคัญ ๆ เพื่อให้ผู้อ่านพอมองเห็นภาพว่าแพ็คเกจเหล่านี้เหมาะสำหรับนำไปใช้กับงานประเภทใด ถ้าสภาพแวดล้อม Python ของผู้อ่านเป็น Anaconda หรือ Miniconda ตามที่ได้แนะนำในตอนต้นของหนังสือเล่มนี้ ผู้อ่านสามารถติดตั้งแพ็คเกจที่จะใช้งานในบทนี้ได้โดยใช้คำสั่งดังต่อไปนี้

```
$ conda install numpy scipy pandas matplotlib scikit-learn
```

ให้เรามาดูแพ็คเกจต่าง ๆ เหล่านี้ทีละตัวโดยสังเขป

NumPy: Numerical Python

NumPy เป็นแพ็คเกจที่มีประสิทธิภาพในการจัดเก็บและจัดการกับอาร์เรย์หลายมิติใน Python คุณสมบัติที่สำคัญของ NumPy มีดังนี้

- มีโครงสร้าง `ndarray` ซึ่งช่วยในการจัดเก็บข้อมูลและการจัดการเวกเตอร์ เมทริกซ์ และ higher-dimensional dataset ได้อย่างมีประสิทธิภาพ
- มีไวยากรณ์ที่อ่านง่ายและมีประสิทธิภาพสำหรับการดำเนินการกับข้อมูล ตั้งแต่การดำเนินการทางคณิตศาสตร์พื้นฐานกับข้อมูลแต่ละตัวในอาร์เรย์ไปจนถึงการดำเนินการทางพีชคณิตเชิงเส้นที่ซับซ้อน

ในกรณีที่ง่ายที่สุด อาร์เรย์ NumPy จะมีหน้าตาเหมือน Python ลิสต์เป็นอย่างมาก ดังแสดงในตัวอย่างข้างล่างนี้ ซึ่งเป็นอาร์เรย์ NumPy ที่มีช่วงของตัวเลขตั้งแต่ 1 ถึง 9 (เปรียบเทียบตัวอย่างนี้กับ `range()` ของ Python)

```
In [1]:  import numpy as np
         x = np.arange(1, 10)
         x

Out [1]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

อาร์เรย์ NumPy มีโครงสร้างที่เอื้ออำนวยต่อการจัดเก็บข้อมูลและการดำเนินการทางคณิตศาสตร์พื้นฐานกับสมาชิกแต่ละตัวในอาร์เรย์ได้อย่างมีประสิทธิภาพ ยกตัวอย่างเช่น เราสามารถคำนวณกำลังสองกับสมาชิกแต่ละตัวในอาร์เรย์ได้โดยการใช้โอเปอเรเตอร์ ** กับอาร์เรย์โดยตรง ดังนี้

```
In [2]:  x ** 2

Out [2]: array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

ให้เรามาลองเปรียบเทียบโค้ดที่ใช้อาร์เรย์ NumPy นี้กับโค้ดที่ใช้ list comprehension ดังแสดงในตัวอย่างข้างล่างนี้ ถึงแม้ว่าทั้งสองวิธีจะให้ผลลัพธ์ออกมาเหมือนกัน แต่โค้ดที่ใช้ list comprehension จะค่อนข้างเยิ่นเย้อกว่า

```
In [3]:  [val ** 2 for val in range(1, 10)]

Out [3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

อาร์เรย์ NumPy สามารถมีได้หลายมิติ อย่างเช่น อาร์เรย์ x ในตัวอย่างข้างล่างนี้ ถูก reshape ให้เป็นอาร์เรย์ขนาด 3x3 ซึ่งจะแตกต่างจาก Python ลิสต์ที่จำกัดไว้ที่มิติเดียว

```
In [4]:  M = x.reshape((3, 3))
         M

Out [4]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

อาร์เรย์สองมิติคือ รูปแบบหนึ่งในการแสดงเมทริกซ์และ NumPy สามารถดำเนินการกับเมทริกซ์ได้อย่างมีประสิทธิภาพ ยกตัวอย่างเช่น เราสามารถคำนวณทรานสโพส (transpose) โดยใช้ `.T` ได้ ดังนี้

```
In [5]:  M.T

Out [5]: array([[1, 4, 7],
                [2, 5, 8],
                [3, 6, 9]])
```

หรือทำการคูณเมทริกซ์กับเวกเตอร์ (matrix-vector product) โดยใช้ `np.dot` ได้ ดังนี้

```
In [6]:  np.dot(M, [5, 6, 7])

Out [6]: array([ 38,  92, 146])
```

หรือแม้กระทั่งทำการคำนวณที่ซับซ้อนอย่าง eigenvalue decomposition ได้ ดังนี้

```
In [7]: np.linalg.eigvals(M)

Out [7]:
array([  1.61168440e+01,  -1.11684397e+00,  -1.30367773e-15])
```

การจัดการพีชคณิตเชิงเส้นดังกล่าวเป็นพื้นฐานในการวิเคราะห์ข้อมูลสมัยใหม่ โดยเฉพาะอย่างยิ่งในด้านการเรียนรู้ของเครื่องจักรและการทำเหมืองข้อมูล

ผู้อ่านสามารถศึกษาเพิ่มเติมเกี่ยวกับ NumPy ได้ โดยดูจากคำแนะนำในบทเรื่อง Resources for Further Learning

Pandas: Labeled Column-Oriented Data

Pandas เป็นแพ็คเกจที่ใหม่กว่า NumPy มาก และอันที่จริงแล้ว แพ็คเกจนี้สร้างต่อยอดมาจาก NumPy แพ็คเกจ Pandas ทำให้เราใช้งานอาร์เรย์หลายมิติได้ในรูปของออบเจ็คต์ DataFrame ที่ผู้ใช้ภาษา R และภาษาที่เกี่ยวข้องคุ้นเคยเป็นอย่างดี ออบเจ็คต์ DataFrame มีหน้าตาดังนี้

```
In [8]:
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
```

```
Out [8]:    label value
         0  A     1
         1  B     2
         2  C     3
         3  A     4
         4  B     5
         5  C     6
```

ออบเจ็คต์ DataFrame ช่วยให้เราดึงข้อมูลเฉพาะคอลัมน์ที่ต้องการโดยใช้ชื่อได้ ดังนี้

```
In [9]:  df['label']
```

```
Out [9]: 0    A
         1    B
         2    C
         3    A
         4    B
         5    C
         Name: label, dtype: object
```

ออบเจ็คต์ DataFrame ยังช่วยให้เราจัดการข้อมูลที่เป็นสตริงในทุกแถวได้ ดังแสดงในตัวข้างล่างนี้

```
In [10]:  df['label'].str.lower()
```

```
Out [10]: 0    a
          1    b
          2    c
          3    a
          4    b
          5    c
          Name: label, dtype: object
```

นอกจากนี้ ออบเจ็คต์ DataFrame ยังช่วยให้เราหาผลบวกของข้อมูลที่เป็นตัวเลขในทุกแถวได้ ดังนี้

```
In [11]:  df['value'].sum()
```

```
Out [11]: 21
```

และที่สำคัญที่สุด ออบเจ็คต์ DataFrame ช่วยให้เรา join และ group ข้อมูลในลักษณะที่คล้าย ๆ กับที่ทำในฐานข้อมูลได้ ดังแสดงในตัวอย่างต่อไปนี้

```
In [12]:  df.groupby('label').sum()

Out [12]:       value
          label
          A     5
          B     7
          C     9
```

จากตัวอย่างนี้ เราจะเห็นว่าเราสามารถคำนวณผลรวมของทุกตัวเลขที่มี label เดียวกันได้ โดยการเขียนโค้ดเพียงบรรทัดเดียว ซึ่งเราจะไม่สามารถเขียนให้โค้ดสั้นกระชับและมีประสิทธิภาพอย่างนี้ ถ้าเราเขียนโค้ดโดยใช้เครื่องมือที่มีมาให้ใน NumPy และในตัวภาษา Python

ผู้อ่านสามารถศึกษาเพิ่มเติมเกี่ยวกับ Pandas ได้ โดยดูจากคำแนะนำในบทเรื่อง Resources for Further Learning

Matplotlib: MATLAB-style scientific visualization

ปัจจุบัน Matplotlib เป็นแพ็คเกจสำหรับทำ visualization ทางวิทยาศาสตร์ที่ได้รับความนิยมมากที่สุด ถึงแม้ว่าจะมีรูปแบบการเขียนที่บางครั้งก็ดูเยิ่นเย้อจนเกินไป แต่ก็เป็นไลบรารีที่มีประสิทธิภาพสำหรับการวาดกราฟประเภทต่าง ๆ

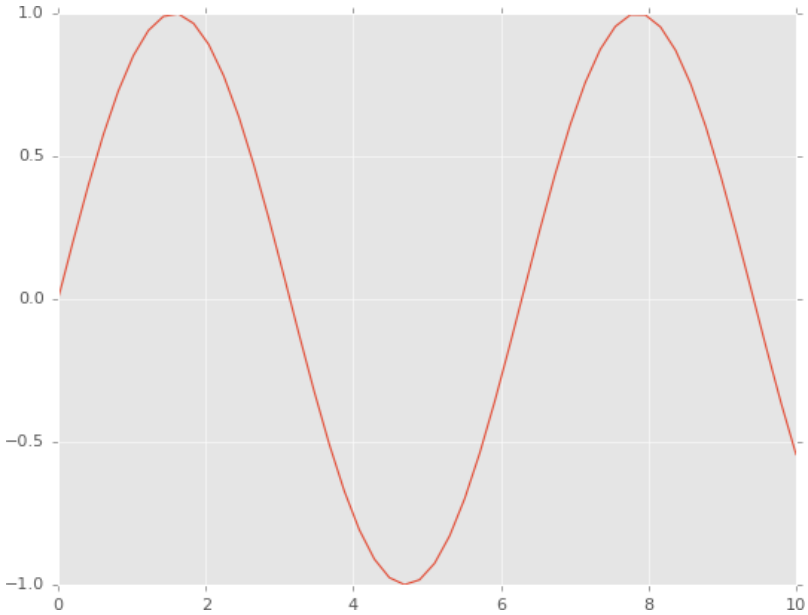
ในการใช้ Matplotlib เราสามารถเริ่มต้นด้วยการเปิดใช้งานโหมด notebook (สำหรับการใช้งานใน Jupyter notebook) จากนั้นทำการโหลดแพ็คเกจโดยใช้ชื่อ `plt` ดังนี้

```
In [13]: # run this if using Jupyter notebook
         %matplotlib notebook

In [14]:
import matplotlib.pyplot as plt
plt.style.use('ggplot')  # make graphs in the style of R's ggplot
```

จากนั้น ให้เราสร้างข้อมูล (แน่นอนว่าเป็นอาร์เรย์ NumPy) และวาดกราฟ ดังนี้

```
In [15]: x = np.linspace(0, 10)  # range of values from 0 to 10
         y = np.sin(x)           # sine of these values
         plt.plot(x, y);         # plot as a line
```



ถ้าผู้อ่านได้ลองรันโค้ดนี้ ผู้อ่านจะเห็นว่าผู้อ่านสามารถมีปฏิสัมพันธ์กับกราฟนี้ได้ ไม่ว่าจะเป็นการแพน (pan) การซูม (zoom) และการเลื่อน (scroll) เพื่อสำรวจข้อมูล

ตัวอย่างนี้เป็นตัวอย่างที่ง่ายที่สุดที่แสดงการใช้ Matplotlib ในการวาดกราฟ ผู้อ่านสามารถศึกษาการวาดกราฟประเภทอื่น ๆ ได้จากแกลเลอรีออนไลน์ของ Matplotlib ที่ https://matplotlib.org/gallery.html และแหล่งข้อมูลอ้างอิงอื่น ๆ ที่ระบุไว้ในบทเรื่อง Resources for Further Learning

SciPy: Scientific Python

SciPy คือชุดฟังก์ชั่นทางวิทยาศาสตร์ที่สร้างต่อยอดมาจาก NumPy แพ็คเกจนี้เริ่มต้นจากชุดของ Python wrapper ที่ห่อหุ้มไลบรารีสำหรับการคำนวณเชิงตัวเลขของภาษา Fortran และถูกพัฒนาต่อเนื่องมา แพ็คเกจนี้ถูกจัดกลุ่มเป็นชุดของโมดูลย่อย ๆ โดยที่แต่ละโมดูลย่อยจะอิมพลีเมนต์อัลกอริทึมทางคณิตศาสตร์บางประเภท รายการต่อไปนี้เป็นตัวอย่างของโมดูลย่อยที่สำคัญสำหรับงานทางด้านวิทยาศาสตร์

- `scipy.fftpack` Fast Fourier transforms
- `scipy.integrate` Numerical integration
- `scipy.interpolate` Numerical interpolation
- `scipy.linalg` Linear algebra routines
- `scipy.optimize` Numerical optimization of functions
- `scipy.sparse` Sparse matrix storage and linear algebra
- `scipy.stats` Statistical analysis routines

ให้เรามาดูตัวอย่างการทำ interpolate โดยใช้แพ็คเกจ scipy ดังนี้

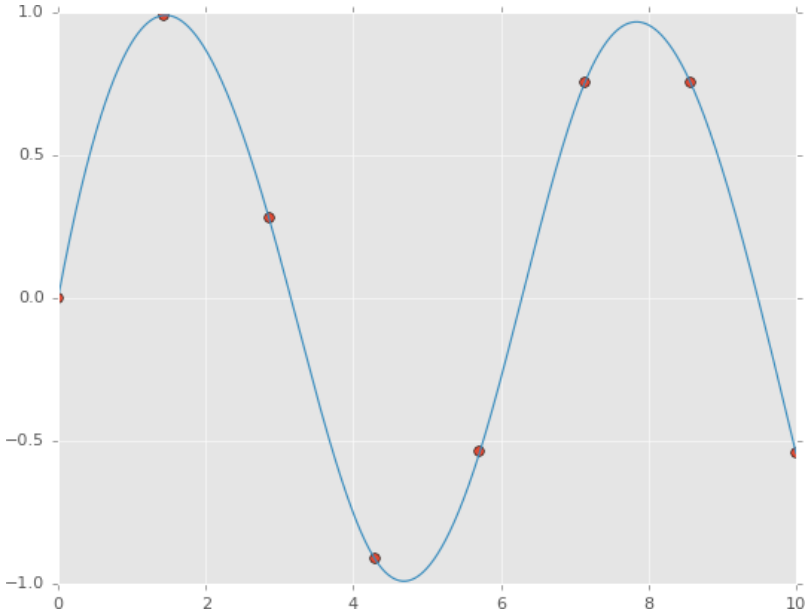
```
In [16]: from scipy import interpolate

         # choose eight points between 0 and 10
         x = np.linspace(0, 10, 8)
         y = np.sin(x)

         # create a cubic interpolation function
         func = interpolate.interp1d(x, y, kind='cubic')

         # interpolate on a grid of 1,000 points
         x_interp = np.linspace(0, 10, 1000)
         y_interp = func(x_interp)

         # plot the results
         plt.figure()  # new figure
         plt.plot(x, y, 'o')
         plt.plot(x_interp, y_interp);
```



ผลลัพธ์ที่เราเห็น คือ การทำ smooth interpolation ระหว่างจุด

Other Data Science Packages

ยังมีแพ็คเกจสำหรับงานทางด้านวิทยาการข้อมูลอีกมากมายที่สร้างต่อยอดมาจากแพ็คเกจที่ได้แนะนำไปในข้างต้น อย่างเช่น แพ็คเกจ Scikit-Learn สำหรับงานทางด้านการเรียนรู้ของเครื่องจักร แพ็คเกจ Scikit-Image สำหรับงานทางด้านการวิเคราะห์ด้วยรูปภาพ (image analysis) แพ็คเกจ StatsModels สำหรับงานทางด้านการทำโมเดลทางสถิติ (statistical modeling) รวมทั้งแพ็คเกจสำหรับงานเฉพาะด้านมาก ๆ อย่างเช่น แพ็คเกจ

AstroPy สำหรับงานทางด้านดาราศาสตร์ (astronomy) และฟิสิกส์ดาราศาสตร์ (astrophysics) แพ็คเกจ NiPy สำหรับงานทางด้านการสร้างภาพระบบประสาท (neuro-imaging) และแพ็คเกจอื่น ๆ อีกมากมาย

ไม่ว่าผู้อ่านกำลังแก้โจทย์ทางด้านวิทยาศาสตร์หรือสถิติประเภทใด ๆ ก็ตาม จะมีโอกาสสูงมากที่มีคนเขียนแพ็คเกจเอาไว้แล้วและสามารถนำไปใช้กับงานของผู้อ่านได้เลย

Resources for Further Learning

ถ้าผู้อ่านได้อ่านจนมาถึงหน้านี้แล้ว ผู้เขียนก็คิดว่าผู้อ่านน่าจะเข้าใจเนื้อหาสำคัญ ๆ ของภาษา Python อย่าง ไวยากรณ์ ความหมายของคำและสัญลักษณ์ที่ใช้ในไวยากรณ์ โอเปอเรเตอร์ และฟังก์ชั่นการทำงาน (functionality) ต่าง ๆ ของภาษา Python รวมทั้งเครื่องมือหรือโค้ดที่เราสามารถไปศึกษาค้นคว้าเพิ่มเติมได้

ผู้เขียนได้อธิบายภาษา Python ในส่วนที่คิดว่ามีประโยชน์มากที่สุดแก่นักวิทยาศาสตร์ข้อมูลที่ต้องใช้งานภาษา Python ไม่ได้เป็นการแนะนำภาษา Python อย่างสมบูรณ์ ดังนั้นผู้เขียนอยากจะแนะนำแหล่งข้อมูลอื่น ๆ ดังข้างล่างนี้ เพื่อที่ผู้อ่านจะได้นำไปใช้เป็นแหล่งอ้างอิงในการศึกษาค้นคว้าเพิ่มเติม เพื่อให้เข้าใจภาษา Python มากยิ่งขึ้น และสามารถใช้งานภาษา Python ได้อย่างมีประสิทธิภาพ

Fluent Python โดย Luciano Ramalho เป็นหนังสือที่ดีมาก ๆ เล่มหนึ่งจากสำนักพิมพ์ O'Reilly book หนังสือเล่มนี้จะอธิบายเกี่ยวกับ best practices และ idiom ของภาษา Python ซึ่งครอบคลุมถึงการเขียนโค้ดโดยใช้ไลบรารีมาตรฐานให้เกิดประโยชน์สูงสุด

Dive Into Python โดย Mark Pilgrim เป็นหนังสือฟรี ที่เผยแพร่ออนไลน์ หนังสือนี้จะแนะนำภาษา Python โดยเริ่มต้นตั้งแต่พื้นฐานจริง ๆ

Learn Python the Hard Way โดย Zed Shaw เป็นหนังสือที่เขียนโดยใช้วิธีการ "เรียนโดยการลองทำ" (learn by trying) และเน้นการสร้างทักษะในค้นคว้าหาคำตอบในสิ่งที่ผู้อ่านไม่เข้าใจ ซึ่งน่าจะเป็นทักษะที่มีประโยชน์กับโปรแกรมเมอร์มากที่สุด

Python Essential Reference โดย David Beazley เป็นหนังสือขนาดใหญ่ มีเนื้อหาประมาณ 700 หน้า เป็นหนังสือที่เขียนได้ดีและครอบคลุมเกือบทุกเรื่องที่เราต้องการรู้เกี่ยวกับภาษา Python และไลบรารีมาตรฐาน นอกจากนี้ David Beazley ยังได้แต่หนังสืออีกเล่มที่ชื่อว่า Python Cookbook ซึ่งเป็นหนังสือที่เน้นการประยุกต์ใช้ภาษา Python

ถ้าต้องการศึกษาเพิ่มเติมเกี่ยวกับการใช้ Python กับงานทางด้านวิทยาการข้อมูลและงานทางด้านการคำนวณทางวิทยาศาสตร์ ผู้เขียนอยากจะแนะนำหนังสือดังต่อไปนี้

The Python Data Science Handbook โดย Jake VanderPlas (ผู้แต่งหนังสือเล่มนี้) เป็นหนังสือที่ให้คำแนะนำที่ครอบคลุมเกี่ยวกับเครื่องมือสำคัญในภาษา Python สำหรับงานทางด้านวิทยาการข้อมูล

Effective Computation in Physics โดย Katie Huff และ Anthony Scopatz เป็นหนังสือที่แนะนำการคำนวณทางวิทยาศาสตร์แบบเป็นขั้นเป็นตอน โดยเริ่มตั้งแต่ระดับพื้นฐาน

Python for Data Analysis โดย Wes McKinney เป็นหนังสือที่แต่งโดยผู้ที่สร้างแพ็คเกจ Pandas หนังสือเล่มนี้จะอธิบายแพ็คเกจ Pandas อย่างละเอียด พร้อมทั้งให้ข้อมูลที่เป็นประโยชน์เกี่ยวกับเครื่องมือที่นำมาใช้ในการสร้างแพ็คเกจนี้ด้วย

สุดท้ายนี้ เพื่อให้ได้ภาพที่กว้างขึ้นว่ามีอะไรอีกบ้าง ผู้เขียนอยากจะแนะนำแหล่งข้อมูลดังต่อไปนี้

O'Reilly Python Resources ซึ่งจะมีหนังสือดี ๆ จำนวนมากที่เกี่ยวกับตัวภาษา Python โดยตรง หรือเกี่ยวกับหัวข้อเฉพาะด้านที่มีการใช้ภาษา Python

PyCon, SciPy, and PyData เป็นคอนเฟอเรนซ์ที่ดึงดูดคนให้มาร่วมงานเป็นจำนวนมากทุกปี และมีการจัดเก็บการบรรยายของวิทยากรในงานในรูปแบบของฟรีวิดีโอออนไลน์ ทำให้เป็นแหล่งข้อมูลขนาดใหญ่ที่เกี่ยวกับภาษา Python