

정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

Copyright © 2009- Charles Severance.

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth.
Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. This license is available at creativecommons.org/licenses/by-nc-sa/3.0/. You can see what the author considers commercial and non-commercial uses of this material as well as license exemptions in the Appendix titled Copyright Detail.

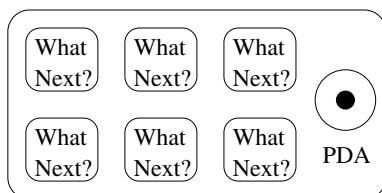
The L^AT_EX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from <http://www.thinkpython.com>.

Chapter 1

왜 프로그래밍을 배워야 하는가?

컴퓨터 프로그램을 만드는 행위(프로그래밍)는 매우 창의적이며 향후 뿐만 아니라 이상으로 얻을 것이 많다. 프로그램을 만드는 이유는 어려운 자료분석의 문제를 해결하려는 것에서부터 다른 사람의 문제를 해결해주는 재미를 느끼는 것까지 다양한 이유가 있다. 이 책에서 모든 사람이 어떻게 프로그램을 만드는지를 알고, 프로그램을 만드는지를 알게되면, 새로 습득한 프로그래밍 기술로 원하는 것을 해결할 수 있는 것을 배우게 된다.

우리의 일상은 노트북부터 핸드폰까지 다양한 컴퓨터에 둘러싸여 있다. 이러한 컴퓨터가 개인비서로 우리를 위해서 많은 일을 대신해 준다고 생각한다. 일상생활에서 접하는 컴퓨터 하드웨어는 우리에게 ”다음에 무엇을 하면 좋겠습니까?”라는 질문을 지속적으로 물어보게 만들어 졌다.



프로그래머는 운영체제와 하드웨어에 응용 프로그램을 만들었고, 결국 많은 것들을 도와주는 PDA(Personal Digital Assistant)로 진화했다. 컴퓨터는 빠르며, 큰 저장소를 가지고 있어 우리가 컴퓨터에게 ”다음꺼 실행해(do next)를 컴퓨터가 이해할 수 있는 말로 지시를 하게되면 우리에게 매우 유용할 수 있다.

예를 들어, 다음의 세 문단을 보고 가장 많이 나오는 문단의 단어를 찾아보고 얼마나 나오는지를 알려주세요라고 컴퓨터에게 시킬 수 있다. 사람이 몇초만에 단어를 읽고 이해할 수는 있지만, 그 단어가 몇번 나오는지를 세는 것은 매우 고생스러운 과정이다. 왜냐하면 사람은 지루하고 반복되는 일의 문제를 해결하는데 적합하지 않기 때문이다. 컴퓨터는 정반대이다. 논문이나 책에서 텍스트를 읽고 이해하는 것은 컴퓨터에게 어렵다. 하지만 단어를 세고 가장 많이 사용되는 단어를 말해주는 것은 컴퓨터에게는 무척이나 쉽다.

python words.py

```
Enter file:words.txt
to 16
```

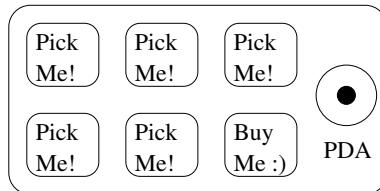
우리의 개인 정보분석 비서는 ”to”라는 단어가 가장 많이 사용되었고 16번 나왔다고 바로 답을 준다.

사람이 잘하지 못하는 점을 컴퓨터가 잘할 수 있다는 사실을 이해하면 왜 컴퓨터 언어로 컴퓨터와 대화해야 하는지를 알 수 있다. 컴퓨터와 대화할 수 있는 언어(Python)를 배우게되면 지루하고 반복되는 일을 컴퓨터가 처리하게 하면 더 많은 시간을 창의적이고, 직관적이며, 창조적인 시간을 컴퓨터와 함께 할 수 있다.

1.1 창의성과 동기

이책은 직업 프로그래머를 위해서 저작된 것은 아니지만, 직업적으로 프로그램을 만드는 작업은 개인적으로나 경제적인 면에서 꽤 매력적인 일이다. 특히, 유용하며, 심미적이고, 똑똑한 프로그램을 다른 사람이 사용할 수 있도록 만드는 것은 매우 창의적인 일이다. 컴퓨터는 다양한 그룹의 프로그래머들이 사용자의 관심과 시선을 빼았기 위해서 경쟁적으로 다양한 프로그램을 가지고 있다. 이렇게 개발된 프로그램은 사용자가 원하는 바를 충족시키고 훌륭한 사용자 경험을 주려고 노력한다. 이러한 상황에서 사용자가 소프트웨어를 고르게 될 때 고객의 선택에 대해서 프로그래머는 직접적으로 보상을 받게된다.

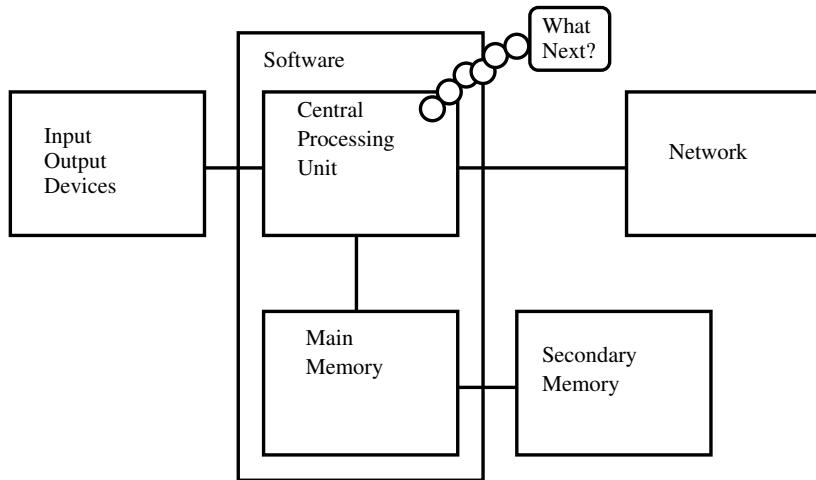
만약 프로그램을 프로그래머 집단의 창의적인 결과물로 바라본다면, 아마도 다음의 그림이 PDA 컴퓨터에 의미가 있을 듯 하다.



우선은 프로그램을 만드는 주된 동기가 사업을 위한다던가 사용자를 기쁘게 한다기보다는 일상생활에서 맞닥드리는 자료와 정보를 잘 다뤄 좀더 생산적으로 우리의 삶을 만드는데 초점을 잡아보자. 프로그램을 만들기 시작할 때 여러분 모두는 프로그래머이면서 동시에 자신이 만든 프로그램의 사용자가 된다. 프로그래머로서 기술을 습득하고 프로그래밍 자체로 창의적으로 느껴진다면, 여러분은 다른 사람을 위해 프로그램을 개발하게 준비가 된 것이다.

1.2 컴퓨터 하드웨어 아키텍처

소프트웨어 개발을 위해 컴퓨터에 지시 명령어를 전달하기 위한 컴퓨터 언어를 배우기 전에, 컴퓨터가 어떻게 구성되어 있는지 이해할 필요가 있다. 컴퓨터 혹은 핸드폰을 분해해서 안쪽을 살펴보면, 다음의 주요 부품을 발견할 수 있다.



주요 부품을 살펴보자.

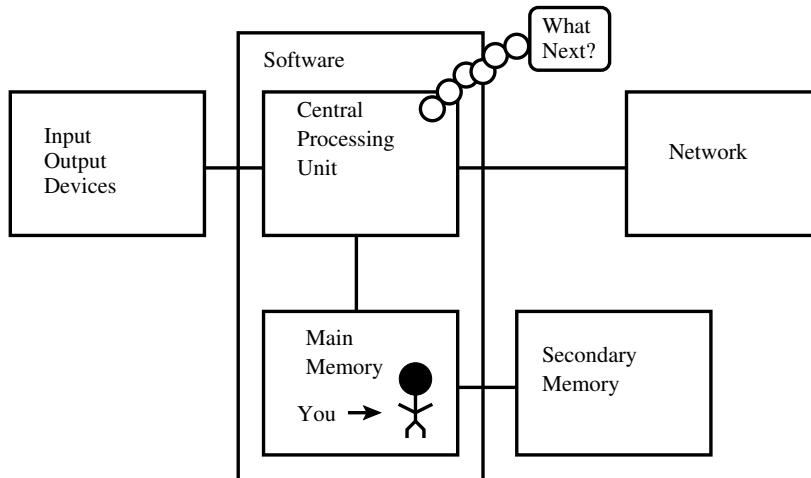
- **중앙처리장치(Central Processing Unit, CPU):** 다음은 무엇을 할까요? (“What is next?”) 명령어를 처리하는 주요 부분이다. 컴퓨터가 3.0 GHz라면 초당 명령어(다음은 무엇을 할까요? What is next?)를 삼백만번 처리할 수 있다고 계속 물을 수 있다. CPU의 처리속도를 따라서 컴퓨터와 빠르게 대화하는 것을 배울 것이다.
- **주 기억장치(Main Memory):** 주 기억장치는 중앙처리장치(CPU)가 급하게 명령어를 처리하기 위해 필요로 하는 정보를 저장하는 용도로 사용된다. 주 기억장치는 중앙처리장치만큼이나 빠르다. 그러나 주 기억장치에 저장된 정보는 컴퓨터가 꺼지면 자동으로 지워진다.

보조 기억장치 보조 기억장치는 정보를 저장하기 위해 사용되지만, 주 기억장치보다 속도가 느리다. 전기가 나갔을 때도 정보를 기억하는 것이 장점이다. 휴대용 USB 기억장치나 이동 MP3 플레이어에 사용되는 USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

- **입출력장치(Input Output Devices):** 간단하게 화면, 키보드, 마우스, 마이크, 스피커, 터치패드가 포함된다. 컴퓨터와 사람이 상호작용하는 방식이다.
- **네트워크(Network):** 요즘 거의 모든 컴퓨터는 네트워크로 정보를 주고 받는 네트워크 커넥션(Network Connection) 하드웨어를 가지고 있다. 네트워크는 정보를 저장하는 느린 저장소로 혹은 때때로 원하는 정보를 가져오지 못하는 것으로 보조 기억장치(Secondary memory)로 생각할 수 있다.

어떻게 이러한 주요 부품들이 작동하는지에 대한 자세한 사항은 컴퓨터를 만드는 사람에게 달려있지만, 프로그램을 만들 때 컴퓨터 주요부품에 대해서 언급되어 컴퓨터 전문용어를 습득하고 이해하는 것은 도움이 된다.

프로그래머로서 여러분들은 사용자가 원하는 자료를 분석하고 문제를 풀 수 있는 컴퓨터 자원들을 사용하고 오케스트레이션하는 것이다.



프로그래머로 중앙처리장치(CPU)와 대화하며 ”다음은 무엇을 수행하세요”라고 지시할 것이다. 때때로 중앙처리장치(CPU)에게 주 기억장치, 보조 기억장치, 네트워크, 입출력 장치를 사용하라고 지시할 것이다.

프로그래머는 컴퓨터의 ”다음은 무엇을 수행할까요”에 대한 답을 하는 사람이기도 하다. 하지만, 컴퓨터에 답하기 위해서 5mm 크기로 컴퓨터에 프로그래머를 집어넣고 초당 30억개의 명령어로 답을 하는 것은 매우 불편할 것이다. 그래서, 미리 컴퓨터에게 수행할 명령문을 써놔야한다. 이렇게 미리 작성된 명령문의 집합을 **프로그램(Program)**이라고 하며, 명령어 집합을 작성하고 명령어 집합이 올바르게 작성될 수 있도록 하는 행위를 **프로그래밍(Programming)**이라고 부른다.

1.3 프로그래밍 이해하기

책의 나머지 장을 통해서 책을 읽고 있는 당신을 프로그래밍의 장인으로 인도할 것입니다. 중국에는 책을 읽고 있는 여러분은 **프로그래머**가 될 것입니다. 아마도 전문적인 프로그래머는 아닐지라도 적어도 자료/정보 분석 문제를 보고 그 문제를 풀수 있는 기술을 가지게는 될 것입니다.

이런 점에서 프로그래머가 되려면 두가지 기술이 필요로 합니다.

- 첫째, 파이썬같은 프로그래밍 언어 - 어휘와 문법을 알 필요가 있습니다. 단어를 새로운 언어에 맞추어 쓸 수 있어야 하며 새로운 언어를 잘 표현된 문장으로 구성하는지를 알아야 합니다.
- 둘째, 스토리(Story)를 말 할 수 있어야 합니다. 스토리를 만들 때, 독자에게 우리의 아이디어(idea)를 전달하기 위해서 단어와 문장을 조합합니다. 스토리를 만들 때 기술과 예술적인 면이 있으며, 기술과 예술적인 면은 여러번 쓰기 연습을 통하고 피드백을 받으므로써 향상됩니다. 프로그래밍에서, 우리가 만든 프로그램은 스토리이고, 풀려고 하는 문제는 ”아이디어”에 해당합니다.

파이썬과 같은 프로그래밍 언어를 배우게 되면, 자바스크립트나 C++ 같은 두 번째 언어를 배우는 것은 무척이나 쉽습니다. 새로운 프로그래밍 언어는 매우 다른 어휘와 문법을 가지지만, 문제푸는 기술을 배우기만 하면, 모든 프로그래밍 언어에서도 동일하게 접근할 수 있습니다.

파이썬의 어휘와 문단은 금방 배웁니다. 새로운 종류의 문제를 풀기 위해 논리적인 프로그램을 짜는 것은 오래 걸립니다. 여러분은 작문을 배우듯이 프로그래밍을 배우게 될 것입니다. 프로그래밍을 읽고 설명하는 것으로 시작해서 간단한 프로그램을 작성하고, 점차적으로 복잡한 프로그램을 작성하게 될 것입니다. 어느 순간에 명상에 잠기게 되고, 문제해결과 프로그램의 패턴을 보게되고, 좀더 자연스럽게 어떻게 문제를 받아들여 그 문제를 해결할 수 있는 프로그램을 작성하게 될 것입니다. 그리고, 그 순간에 도착하게 되면, 프로그래밍은 매우 즐겁고 창의적인 과정이 될 것입니다.

파이썬 프로그램의 어휘와 구조로 시작을 합니다. 간단한 예제가 언제 처음으로 프로그램을 읽기 시작했는지를 일깨워 주니 인내심을 가지세요.

1.4 단어와 문장

사람의 언어와 달리, 파이썬의 어휘는 실질적으로 매우 적다. 어휘를 예약어 (reserved words)로 부른다. 이들 단어는 파이썬에 매우 특별한 의미를 가진다. 파이썬 프로그램에서 파이썬이 이들 단어를 보게되면, 이들 단어는 파이썬에게 단 하나의 유일한 의미를 지니게 된다. 후에 여러분들이 프로그램을 작성할 때 여러분들이 만든 자신만의 단어를 작성하게 되는데 이를 **변수(Variable)**라고 합니다. 여러분의 변수의 이름을 지을 때 폭넓은 자유를 가질 수 있지만, 변수의 이름으로 파이썬의 예약어를 사용할 수는 없습니다.

이런 점에서 강아지를 훈련시킬 때 ”앉아”, ”기달려”, 가져와 같은 특별한 어휘를 사용합니다. 강아지에게 이런 특별한 예약어를 사용하지 않을 때, 강아지는 주인이 특별한 어휘를 사용하지 않을 때 주인을 물끄러미 쳐다보기만 합니다. 예를 들어, ”여러분이 더 많은 사람들이 건강을 전반적으로 향상하는 방향으로 가자 원한다”고 말하면, 강아지가 듣는 것은 ”뭐라 뭐라 뭐라 **walk** 뭐라” 이렇게 들릴 것이다. 왜냐하면 ”가자”가 강아지의 언어에는 예약어¹이기 때문이다. 이러한 사실은 개와 사람사이에는 예약어가 없다는 것을 의미할지도 모른다.

사람이 파이썬에게 말을 하는 예약어는 다음과 같은 것이 있다.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

¹<http://xkcd.com/231/>

강아지의 사례와는 다르게 파이썬은 이미 완벽하게 훈련이 되어 있다. 여러분이 “try”라고 말하면, 파이썬은 여러분이 매번 “try”라고 말할 때마다 실패 없이 시도를 할 것이다.

이러한 예약어를 배울 것이고 어떻게 잘 사용되는지도 함께 배울것이지만, 지금은 파이썬에 말하는 것에 집중할 것이다. 파이썬에게 말하는 것 중 좋은 것은 다음과 같은 메세지를 던지는 것으로도 파이썬에 말을 할 수 있다는 것이다.

```
print 'Hello world!'
```

이 간단한 문장이 파이썬의 구문(Syntax)론적으로 완벽하다. 위 문장은 예약어 ‘print’로 시작해서 출력하고자 하는 문자열을 작은 따옴표로 감싸안아 올바르게 파이썬에게 전달했다.

1.5 파이썬과 대화하기

파이썬으로 우리가 알고 있는 단어를 가지고 간단한 문장을 만들었고, 새로운 언어 기술을 시험하기 위해서 파이썬과 대화를 어떻게 하는지 알 필요가 있다.

파이썬과 대화를 시작하기 전에, 파이썬 소프트웨어를 컴퓨터에 설치하고 컴퓨터에서 파이썬을 어떻게 실행하는지를 배워야 한다. 이번장에서 다루기에는 너무 구체적이고 자세한 사항이기 때문에 [www.pythonguides.com](http://www.pythonguides.com/installing-python-on-mac-os-x/)을 참조하는 것을 권고한다. 자세한 설치 방법과 화면을 캡쳐하여 윈도우와 맥킨토쉬 시스템 및 실행하는 방법을 설명하였다. 설치가 마무리되고 터미널이나 명령어 실행창에서 **python**을 치게되면 파이썬 인터프리터가 인터랙티브 모드로 실행을 시작하고 다음과 같은 것이 화면에 뿌려진다.

```
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

>>> 프롬프트는 파이썬 인터프리터가 여러분에게 요청하는 방식이다. ”다음에 파이썬이 무엇을 실행하기를 원합니까?” 파이썬은 여러분과 대화를 나눌 준비가 되었다. 이제 남은 것은 파이썬 언어로 어떻게 말하는 지를 여러분이 아는 것이고 여러분은 대화를 할 수 있다.

예를들어 여러분이 가장 간단한 파이썬 언어의 단어나 문장 조차도 알수가 없다고 가정해 봅시다. 우주 비행사가 저 멀리 떨어진 행성에 착륙할 때 사용하는 간단한 말을 사용하여 행성의 거주민에게 대화를 시도한다고 생각해 봅시다.

```
>>> I come in peace, please take me to your leader
File "<stdin>", line 1
    I come in peace, please take me to your leader
^
SyntaxError: invalid syntax
>>>
```

잘 되는것 같지 않습니다. 뭔가 빨리 다른 생각을 하지 않는다면, 행성의 거주민은 창으로 찔르고, 기름에 잘 발라 불위에서 바베큐를 만들어 저녁으로 먹을 듯합니다.

운 좋게도 기나긴 우주 여행중 이책의 복사본을 가지고 와서 다음과 같이 빠르게 친다고 생각해봅시다.

```
>>> print 'Hello world!'
Hello world!
```

훨씬 좋아보기고, 좀더 커뮤니케이션을 이어갈 수 있을 것으로 보입니다.

```
>>> print 'You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
>>> print 'We have been waiting for you for a long time'
We have been waiting for you for a long time
>>> print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
      File "<stdin>", line 1
          print 'We will have a feast tonight unless you say
                                         ^
SyntaxError: EOL while scanning string literal
>>>
```

이번 대화는 잠시동안 잘 진행되다가 여러분이 파이썬 언어로 말하다가 정말 사소한 실수를 저질러 파이썬이 오류를 뱉어낸다.

이번에 파이썬이 놀랍도록 복잡하고 강력하고 파이썬과 의사소통을 할때 사용하는 신택스(syntax)가 매우 까다롭다는 것은 알 수 있었다. 파이썬은 다른말로 똑똑(Intelligent)하지는 않다. 지금까지 여러분은 자신과 대화를 저절한 신택스(syntax)를 가지고 대화를 했습니다.

여러분이 다른사람이 작성한 프로그램을 사용한다는 것은 여러분과 파이썬을 사용하는 다른 프로그래머가 파이썬을 중간 매개체로 대화를 하는 것으로 볼 수 있습니다. 파이썬은 프로그램을 만든 저작자가 어떻게 대화가 진행되어져야 하는지를 표현하는 방식입니다. 다음 몇 장에 걸쳐서 여러분은 파이썬을 이용하여 여러분의 프로그램을 이용하는 다른 많은 프로그래머 중의 한명이 될 것입니다.

파이썬 인터프리터와 대화하는 첫장을 끝내기 전에, 파이썬 행성의 거주자에게 ”안녕히 계세요”를 말하는 적절한 방법을 알아야 한다.

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined

>>> if you don't mind, I need to leave
  File "<stdin>", line 1
      if you don't mind, I need to leave
                                         ^
SyntaxError: invalid syntax

>>> quit()
```

위 처음 두개의 시도는 다른 오류 메세지를 출력한다. 두번째 오류는 다른데 이유는 **if**가 예약어이기 때문에 파이썬은 이 예약어를 보고 뭔가 다른 것을 말한다고 생각하지만, 잠시 후 문장의 신택스가 잘못됐다고 판정하고 오류를 뱉어낸다.

파이썬에게 ”안녕히 계세요”를 말하는 올바른 방식은 인터렉티브 >>> 프롬프트에서 **quit()**를 입력하는 것이다.

1.6 전문용어: 인터프리터와 컴파일러

파이썬은 사람이 읽고 쓸수 있고 컴퓨터도 읽고 쓸 수 있도록 고안된 **하이레벨(High-level)** 언어이다. 다른 하이레벨 언어는 자바, C++, PHP, 루비, 베이직, 펄, 자바스크립트 등 다수가 있다. 중앙처리장치(CPU)내에서 실제 하드웨어 수준에서 이런 하이레벨 언어를 이해하지 못한다.

중앙처리장치는 우리가 **기계어(machine-language)**로 부르는 언어만 이해한다. 기계어는 매우 간단하고 솔직히 작성하기에는 매우 지루하다. 왜냐하면 모두 0과 1로만 표현되기 때문이다.

```
01010001110100100101010000001111  
11100110000011101010010101101101  
...
```

0과 1로만 되어 있기 때문에 기계어가 간단해 보이지만, 시택스는 복잡하고 파이썬보다 훨씬 어렵다. 그래서 매우 소수의 프로그래머만이 기계어를 쓸수 있다. 대신에 파이썬과 자바스크립트 같은 하이레벨 언어로 프로그래머가 작성할 수 있도록 다양한 번역기(translator)를 만들었다. 이들 번역기는 프로그램을 중앙처리장치에 의해서 실제 실행이 가능한 기계어로 변환하여 준다.

기계어는 컴퓨터하드웨어에 묶여있기 때문에 기계어는 다른 형식의 하드웨어에 **이식(portable)**이 되지 않는다. 하이레벨 언어로 작성된 프로그램은 새로운 하드웨어위에 다른 인터프리터를 이용하여 옮겨 실행이 가능하고 다른 하드웨어에 사용할 수 있도록 프로그램을 다시 컴파일하여 사용할 수 있다.

프로그래밍 언어의 번역기는 두가지 범주가 있다. (1) 인터프리터 (2) 컴파일러

인터프리터는 프로그래머에 의해서 쓰여진 소스코드를 읽고, 소스코드를 파싱하고, 즉석에서 명령어를 해석한다. 파이썬은 인터프리터다. 파이썬을 인터렉티브 모드로 실행할때, 파이썬 명령문을 쓰면, 파이썬이 즉성에서 처리하고, 다른 파이썬 명령어를 여러분으로부터 기다린다.

파이썬 명령어는 파이썬이 나중에 사용될 값을 기억하기를 바란다. 적당한 이름을 잡아서 그 값을 기억시키고, 나중에 그 이름을 호출하여 값을 사용할 수 있다. 이러한 목적으로 값을 저장하는 것을 **변수(variable)**라고 한다.

```
>>> x = 6  
>>> print x  
6  
>>> y = x * 7
```

```
>>> print y  
42  
>>>
```

이 예제에서 파이썬이 `x`라는 라벨을 사용하여 6이라는 값을 저장하기를 바라고 나중에 사용코자 한다. `print` 예약어를 사용하여 파이썬이 잘 기억하고 있는지를 검증한다. 그리고 `x`를 반환하여 7을 곱하고 새로운 변수 `y`에 값을 집어 넣는다. 그리고 `y`에 현재 무엇이 저장되어 있는지 출력하라고 파이썬에게 요청한다.

파이썬에 한줄 한줄 명령어를 쳐 넣고 있지만, 파이썬은 앞쪽에 명령문에서 생성된 자료가 나중의 실행 명령문에서 사용될 수 있도록 정렬된 명령문으로 처리한다. 방금전 논리적이고 의미있는 순서로 4줄의 명령문을 가진 한 단락을 작성한 것이다.

위에서 본것처럼 파일과 대화를 주고받을 수 있는 것이 인터프리터의 본질이다. 컴파일러는 완전한 프로그램을 하나의 파일에 담겨지고, 하이레벨 소스코드가 기계어로 번역되고, 컴파일러가 나중에 실행되도록 기계어를 파일에 담아놓는다.

윈도우를 사용한다면, 실행 가능한 기계어 프로그램의 확장자가 ”.exe”(executable), 혹은 ”.dll”(dynamically loadable library)임을 확인할 수 있다. 리눅스와 맥 키토쉬에서는 실행화일을 의미하는 확장자는 없다.

텍스트 편집기에서 실행파일을 열게되면 다음과 같은 읽을 수 없는 좀 괴상한 출력물을 화면상에서 확인할 수 있다.

It is not easy to read or write machine language so it is nice that we have **interpreters** and **compilers** that allow us to write in a high-level language like Python or C.

Now at this point in our discussion of compilers and interpreters, you should be wondering a bit about the Python interpreter itself. What language is it written in? Is it written in a compiled language? When we type “python”, what exactly is happening?

The Python interpreter is written in a high level language called “C”. You can look at the actual source code for the Python interpreter by going to www.python.org and working your way to their source code. So Python is a program itself and it is compiled into machine code and when you installed Python on your computer (or the vendor installed it), you copied a machine-code copy of the translated Python program onto your system. In Windows the executable machine code for Python itself is likely in a file with a name like:

That is more than you really need to know to be a Python programmer, but sometimes it pays to answer those little nagging questions right at the beginning.

1.7 프로그램 작성하기

파이썬 인터프리터에 명령어를 치는 것은 파이썬의 주요기능을 알아볼 수 있는 좋은 방법이지만 좀더 복잡한 문제를 해결하기 위해서는 권해드리지는 않습니다.

프로그램을 작성할 때 **스크립트(script)**로 불리는 파일에 명령어 집합을 작성하기 위해서 텍스트 편집기를 주로 사용합니다. 파이썬 스크립트는 .py라는 확장자로 가집니다.

스크립트를 실행하기 위해서 파이썬 인터프리터에게 파일의 이름을 말해줍니다. 유니스나 윈도우 명령창에서 python hello.py를 치게 되면 다음과 같은 결과를 얻게 됩니다.

```
csev$ cat hello.py
print 'Hello world!'
csev$ python hello.py
Hello world!
csev$
```

”csev\$”은 운영시스템의 명령어 프롬프트이고, ”cat hello.py”는 문자열을 출력하라는 한줄의 파이썬 프로그램을 표시하라는 명령어입니다.

인터랙트브 모드로 파이썬 코드를 보여달라는 방식 대신에 파이썬 인터프리터를 호출하고 ”hello.py” 파일에서 소스코드를 읽으라고 말하는 것입니다.

이 새로운 방식은 파이썬 프로그램을 끝내기 위해 **quit()**을 사용할 필요가 없는 것이 좋은 점입니다. 파이썬이 파일에서 소스코드를 읽을 때, 파일 끝까지 읽게되면 파이썬은 자동으로 끝낼 줄을 알고 있습니다.

1.8 프로그램이란 무엇인가?

프로그램(Program)의 가장 기본적인 정의는 어떠한 일을 할 수 있도록 조작된 일련의 파이썬 명령문의 집합이다. 가장 간단한 **hello.py** 스크립트도 프로그램이다. 한줄의 프로그램으로 특별히 유익하고 쓸모가 있는 것은 아니지만 엄격한 의미에서 파이썬 프로그램이 맞다.

프로그램을 이해하는 가장 쉬운 방법은 프로그램이 어떠한 문제를 풀려고 만들 어졌는지 문제를 먼저 생각하는 것이다. 그리고 그 문제를 풀려고 작성된 프로그램을 살펴보는 것이다.

예를 들어 여러분이 페이스북의 일련의 게시된 글에 가장 자주 사용된 단어에 관심을 가지고 소셜 컴퓨팅 연구를 한다고 생각해 봅시다. 페이스북에 게시된

글들을 출력하고 가장 흔한 단어를 찾으로 열심히 들여다 볼 것이지만 매우 오래 걸리고 실수하기 쉽습니다. 하지만 파이썬 프로그램을 이용하면, 빨리 정확하게 작업을 할 수 있는 파이썬 프로그램을 작성해서 주말에 뭔가 재미있는 일로 보낼 수 있습니다.

예를 들어 다음의 자동차(car)와 광대(clown)를 보고, 가장 많이 나오는 단어는 무엇이며 몇번 나왔는지 세어보세요.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

그리고, 몇백만줄의 텍스트를 보고서 동일한 일을 있다고 상상해 보세요. 솔직히 수작업으로 단어를 세는 것보다 파이썬을 배워 프로그램을 배우는 것이 훨씬 빠를 것입니다.

더 좋은 소식은 이미 텍스트 파일에서 가장 자주 나오는 단어를 찾아내는 간단한 프로그램을 제시했고, 작성했고, 시험까지 했다. 이걸 가지고 여러분들은 바로 사용을 할 수 있기 때문에 수고를 덜 수 있다.

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
    counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

이 프로그램을 사용하려고 파이썬을 알 필요도 없다. 10장에 걸쳐서 멋진 파이썬 프로그램을 만드는 방법을 배우게 될 것입니다. 지금 여러분은 사용자로 단순히 프로그램을 사용하지만 이 프로그램이 얼마나 많은 수작업을 줄일 수 있는지 보여주고 있다. 코드를 작성하고 **words.py**로 저장하여 실행을 하거나, <http://www.pythontutorial.net/code/>에서 소스코드를 다운로드 받아 실행하면 된다.

파이썬과 파이썬 언어가 중간의 중개자로서 여러분(사용자)과 저자(프로그래머)사이에서 중개자의 역할을 훌륭히 하고 있는 것을 보여주고 있습니다. 파이썬이 설치된 컴퓨터에서 누구나 사용할 수 있는 공통의 언어로 유용한 명령 순서(즉, 프로그램)를 주고받는지 보여준다. 그래서 파이썬과 직접 의사소통하지 않고 파이썬을 통해서 서로 의사소통할 수 있다.

1.9 프로그램 구성요소

다음의 몇장에 걸쳐서 파이썬 어휘, 문장구조, 문단구조에 대해서 배울 것이다. 파이썬의 강력한 점에 대해서 배울 것이고, 유용한 프로그램을 작성하기 위해서 파이썬의 역량을 조합하는 법을 배울 것이다.

프로그램을 작성하기 위해서 사용하는 로우레벨(low-level) 패턴이 몇가지 있다. 파이썬을 위해서 만들어졌다기 보다는 기계어부터 하이레벨 언어에 이르기까지 모든 언어에 공통된 사항이다.

입력: 컴퓨터 바깥세계에서 데이터를 가져온다. 파일로부터 데이터를 읽을 수도 있고, 마이크나 GPS 같은 센서에서 데이터를 입력받을 수도 있다. 위 프로그램에서 사용자의 키보드로 데이터를 입력받아 입력값으로 사용된 사례이다.

출력: 화면에 프로그램의 결과값을 보여주거나 파일에 저장한다. 혹은 음악을 연주하거나 텍스트를 읽도록 스피커 같은 장치에 데이터를 보낸다.

순차 실행: 스크립트에 작성된 순서에 맞추어 한줄 한줄 실행된다.

조건 실행: 조건을 확인하고 명령문을 실행하거나 건너뛴다.

반복 실행: 반복적으로 명령문을 실행한다. 대체로 반복 실행시 변화를 수반 한다.

재사용: 명령어를 한번 작성하고 이름을 주어 저장하고 프로그램의 필요에 따라 이름을 불러 몇차례 다시 사용한다.

너무나 간단하게 들리지만, 전혀 간단하지는 않다. 걸음거리를 간단히 ”한 다리를 다를 다리 앞에 놓으세요”라고 말하는 것 같다. 프로그램을 짜는 ”예술”은 이러한 기본 요소를 조합하고 엮어 사용자에게 유용한 무언가를 만드는 것이다.

단어를 세는 프로그램은 위 프로그램의 기본요소를 하나만 빼고 모두 사용하여 작성되었다.

1.10 프로그램이 잘못되면?

파이썬과 처음에 대화를 할때, 파이썬 코드를 명확하게 작성하여 의사소통을 해야한다. 작은 차이 혹은 실수는 파이썬이 여러분이 작성한 프로그램을 보다가 포기하게 만듭니다.

초보 파이썬 프로그래머는 파이썬이 오류에 대해서는 인정사정볼 것 없다고 생각합니다. 파이썬이 모든 사람을 좋아하는 것 같지만, 파이썬은 사적으로 사람들을 알고 있고, 뒤끝이 있습니다. 이러한 사실로 인해서 파이썬은 완벽하게 작성된 프로그램만을 가지게 되고 ”잘못 작성되었어요”라고 뱉어내고 여러분에게 고통을 줍니다.

```

>>> print 'Hello world!'
      File "<stdin>", line 1
          print 'Hello world!'
                  ^
SyntaxError: invalid syntax
>>> print 'Hello world'
      File "<stdin>", line 1
          print 'Hello world'
                  ^
SyntaxError: invalid syntax
>>> I hate you Python!
      File "<stdin>", line 1
          I hate you Python!
                  ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
      File "<stdin>", line 1
          if you come out of there, I would teach you a lesson
                  ^
SyntaxError: invalid syntax
>>>

```

파이썬과 다뤄봐야 얻을 것은 없어요. 파이썬은 도구고 감정이 없습니다. 여러분이 필요로 할 때마다 여러분에게 봉사하고 기쁨을 주기 위해서 존재할 뿐입니다. 오류 메세지가 심하게 들릴지는 모르지만 단지 파이썬이 도와달라고 하는 요청일 뿐입니다. 여러분이 입력한 것을 쭉 읽어보고 여러분이 입력한 것을 이해할 수 없다고만 말할 뿐입니다.

파이썬은 어떤면에서 강아지와 닮았습니다. 맹목적으로 여러분을 사랑하고, 강아지가 이해하는 몇몇 단어만 이해하며, 웃는 표정(>>> 명령 프롬프트)으로 여러분이 무언가를 말하기만을 기다립니다. 파이썬이 "SyntaxError: invalid syntax"을 뱉어낼 때, 꼬리를 흔들면서 "뭔가 말씀하시는 것 같은데요... 주인님 말씀을 이해하지 못하겠어요, 다시 말씀해 주세요 (>>>)" 말하는 것과 같다.

여러분이 작성하는 프로그램이 점점 유용해지고 복잡해짐에 따라 3가지 유형의 오류를 마주치게 된다.

구문 오류(Syntax Error): 첫번째 마주치는 오류로 고치기 가장 쉽습니다. 구문 오류는 파이썬의 문법에 맞지 않는다는 것을 의미합니다. 파이썬은 구문 오류가 발생한 줄을 찾아 정확한 위치를 알려줍니다. 하지만, 파이썬이 제시하는 오류가 그 이전 프로그램 부문에서 발생했을 수도 있기 때문에 파이썬이 알려주는 곳 뿐만 아니라 그 앞쪽도 살펴볼 필요가 있다. 따라서 파이썬이 제시하는 구문 오류의 경우 오류가 난 곳은 오류를 고치기 위한 시작점으로 의미가 있다.

논리 오류(Logic Error): 논리 오류는 프로그램의 구문은 완벽하지만 명령문의 실행에 혹은 다른 명령어부분과 관련된 부문에서 실수가 있는 것이다. 논리 오류의 예를 들어보자. "물병에서 한모금 마시고, 가방에 넣고, 도서관으로 걸어가서, 물병을 닫는다"

시맨틱 오류(Semantic Error): 시맨틱 오류는 구문론적으로 완벽하고 올바른 순서로 프로그램의 명령문이 작성되었지만 프로그램에 오류가 숨어있다.

프로그램은 완벽하게 작동하지만 여러분이 의도한 바를 수행하지는 못합니다. 간단한 예로 여러분이 식당으로 가는 방향을 알려주고 있습니다.” “… 주유소 사거리에 도착했을 때, 왼쪽으로 돌아 1.6km 쭉 가면 왼쪽편에 빨간색 빌딩에 식당이 있습니다.” 친구가 매우 늦어 전화로 지금 농장에 있고 혀간으로 걸어가고 있는데 식당을 발견할 수 없다고 전화를 합니다. 그러면 여러분은 ”주유소 왼쪽 혹은 오른쪽을 돋거야” 말하면, 그 친구는 ”말한대로 완벽하게 따라서 했고, 말한대로 적기까지 했는데, 왼쪽으로 돌아 1.6km에 주요소가 있다고 했어”, 그러면 여러분은 ”미안해, 내가 가지고 있는 건 구문론적으로는 완벽한데, 사소한 시맨틱 오류가 있네!”라고 말할 것이다.

위 세 종류의 오류에 대해서 파이썬은 여러분이 요청한 것을 충실히 수행하기 위해서 최선을 다합니다.

1.11 학습으로의 여정

여러분이 책을 읽으면서 개념들이 처음에 잘 와 닿지 않는다고 기죽을 필요는 없어요. 어렵누이 말하는 것을 배울 때, 처음 몇년동안 웅얼거리는 것은 문제가 아닙니다. 간단한 어휘에서 간단한 문장으로 옮겨가고, 문장에서 문단으로 옮겨가는데 6개월이 걸려도 괜찮습니다. 흥미로운 완벽한 짧은 스토리를 자신의 언어로 작성하는데 몇 년이 걸립니다.

여러분이 파이썬을 빨리 배울수 있도록 다음의 몇장에 걸쳐서 정보를 제공해 드릴 것입니다. 새로운 언어를 습득하는 것과 같아서 자연스럽게 느껴지기까지 흡수하고 이해하기까지 시간이 걸립니다. 큰 그림(Big Picture)을 이루는 작은 조각을 정의하면서 여러분을 큰 그림을 볼 수 있도록 여러 주제를 찾고, 또 찾으면서 혼란이 생길 수 있다. 이책이 순차 선형적으로 쓰여져서 본 과정을 선형적으로 배워갈 수도 있지만, 비선형적으로 본 교재를 활용하는 것도 괜찮다. 가볍게 앞쪽과 뒷쪽을 넘나들며 책을 읽을 수도 있다. 구체적이고 세세한 점을 완벽하게 이해하지 않고 고급 과정을 가볍게 읽으면서 프로그래밍의 ”왜(Why)”에 대해서 더 잘 이해할 수도 있다. 앞에서 배운것을 다시 리뷰하고 앞의 연습 문제를 다시 하면서 처음에 난공불락이라 여겼던 어려운 주제도 더 잘 배우고 이해할 수 있다는 것을 알게될 것이다.

대체적으로 처음 프로그래밍 언어를 배울 때 망치로 돌을 내리치고, 끌로 깎아내고 하면서 아름다운 조각품을 만들면서 겪게되는 몇 번의 ”유레카, 아 하” 순간이 있다.

만약 어떤 것이 특별히 힘들다면, 밤새도록 앉아서 지켜보고 노력하는 것은 별로 의미가 없다. 잠시 쉬고, 낮잠을 자고, 간식을 먹고 다른사람이나 강아지에게 문제를 설명하고 자문을 구한 후에 깨끗한 정신과 눈으로 돌아와서 다시 시도해 보라. 단연컨데 이책의 프로그래밍 개념을 배우자마자 정말 쉽고 멋지다는 것을 돌이켜 보면 알게될 것이다. 프로그래밍 언어는 정말 배울 가치가 있다.

1.12 용어사전

버그(bug): 프로그램 오류

중앙처리장치(central processing unit, CPU): 컴퓨터의 심장, 여러분이 작성한 프로그램을 실행하는 장치, ”CPU” 혹은 프로세서라고 불립니다.

컴파일러(compile): 하이레벨 언어로 작성된 프로그램을 로우레벨 언어로 즉시 혹은 나중에 사용하도록 번역하는 번역기

하이레벨 언어(high-level language): 사람이 읽고 쓰기 쉽게 설계된 파이썬과 같은 프로그래밍 언어

인터랙티브 모드(interactive mode): 프롬프트에서 명령어나 표현식을 입력함으로써 파이썬 인터프리터를 사용하는 방식

인터프리트(interpret): 하이레벨 언어의 프로그램을 한번에 한줄씩 번형해서 실행하는 것

로우레벨 언어(low-level language): 컴퓨터가 실행하기 쉽게 설계된 프로그래밍 언어, ”기계어 코드”, ”어셈블리 언어”로 불린다.

기계어 코드(machine code): 중앙처리장치에 의해서 바로 실행될수 있는 가장 낮은 수준의 언어로된 소프트웨어

주메모리(main memory): 프로그램과 데이터를 저장한다. 전기가 나가게 되면 주메모리에 저장된 정보는 사라진다.

파싱(parse): 프로그램을 검사하고 구문론적 구조를 분석하는 것

이식(portability): 하나 이상의 컴퓨터에서 실행될 수 있는 프로그램의 특성

출력문(print statement): 파이썬 인터프리터가 화면에 값을 출력할 수 있게 만드는 명령문

문제해결(problem solving): 문제를 만들고, 답을 찾고, 답을 표현하는 과정

프로그램(program:) 컴퓨터이션(Computation)을 명세하는 명령어의 집합

프롬프트(prompt): 프로그램이 메세지를 출력하고 사용자가 프로그램에 입력하도록 기다릴 때.

보조 기억장치 전기가 나갔을 때도 정보를 기억하고 프로그램을 저장하는 저장소. 일반적으로 주메모리보다 속도가 느린다. USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.

시맨틱(semantics): 프로그램의 의미

시맨틱 오류(semantic error): 프로그래머가 의도한 것과 다른 행동을 하는 프로그램의 오류

소스코드(source code): 하이레벨 언어로 기술된 프로그램

1.13 연습문

Exercise 1.1 컴퓨터 보조기억장치의 기능은 무엇입니까?

- a) 모든 연산과 프로그램의 로직을 수행한다.
- b) 인터넷의 웹페이지를 불러온다.
- c) 파워가 없을 때도 장시간 정보를 저장한다.
- d) 사용자로부터 입력정보를 받는다.

Exercise 1.2 프로그램은 무엇입니까?

Exercise 1.3 컴파일러와 인터프리터의 차이점을 설명하세요.

Exercise 1.4 기계어 코드는 다음 중 어느 것입니까?

- a) 파일 인터프리터
- b) 키보드
- c) 파일 소스코드 파일
- d) 워드 프로세싱 문서

Exercise 1.5 다음 코드에서 잘못된 점을 설명하세요.

```
>>> print 'Hello world!'
      File "<stdin>", line 1
          print 'Hello world!'
                  ^
SyntaxError: invalid syntax
>>>
```

Exercise 1.6 다음의 파이썬 프로그램이 실행된 후에 변수 "X"는 어디에 저장됩니까?

```
x = 123
```

- a) 중앙처리장치
- b) 주메모리
- c) 보조메모리
- d) 입력장치
- e) 출력장치

Exercise 1.7 다음 프로그램에서 출력되는 것은 무엇입니까?

```
x = 43
x = x + 1
print x
```

- a) 43
- b) 44
- c) $x + 1$
- d) 오류, 왜냐하면 $x = x + 1$ 은 수학적으로 불가능하다.

Exercise 1.8 사람의 어느 능력부위에 해당하는지 예로하여 다음을 설명하세요. (1) 중앙처리장치, (2) 주메모리, (3) 보조메모리, (4) 입력장치 (5) 출력장치 중앙처리장치에 상응하는 사람의 몸 부위는 어디입니까?

Exercise 1.9 구문오류("Syntax Error")는 어떻게 고칩니다?

Chapter 2

변수, 표현식, 스테이트먼트 (Statement)

2.1 값(Value)과 형식(Type)

값(Value)은 문자와 숫자처럼 프로그램이 다루는 가장 기본이 되는 단위이다. 우리가 지금까지 살펴본 값은 1, 2 그리고 'Hello, World!' ('안녕 세상!') 이다.

이들 값들은 다른 형식에 속해 있는데, 2는 정수, 'Hello, World!' ('안녕 세상!') 는 문자열(String)에 속해 있다. 문자(Letter)의 열에 있어서 문자열이라고 부른다. 여러분과 인터프리터는 문자열을 확인할 수 있는데 이유는 인용부호 따옴표에 쌓여 있기 때문이다.

`print` 문은 정수에도 사용할 수 있다. `python` 명령어를 실행하여 인터프리터를 구동시키자.

```
python
>>> print 4
4
```

값의 형식을 확신을 못한다면, 인터프리터가 알려준다.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

놀랍지도 않게, strings은 str 형식이고, 정수는 int 형식이다. 소수점을 가진 숫자는 float 형식이다. 왜냐하면 이들 숫자가 부동소수점 형식으로 표현되기 때문이다.

```
>>> type(3.2)
<type 'float'>
```

'17', '3.2' 같은 값은 어떨가? 문자처럼 보이지만 문자처럼 따옴표안에 쌓여 있다.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

'17', '3.2' 은 문자열이다.

아주 큰 정수를 입력할, 1,000,000 처럼 세자리 숫자마다 콤마(,)를 입력한다. 하지만, 파이썬에서 적법한 정수는 아니지만 적법하다.

```
>>> print 1,000,000
1 0 0
```

하지만, 파이썬이 뺄은 값은 우리가 기대했던 것이 아니다. 파이썬은 1,000,000 을 콤마로 구분된 정수로 인식한다. 따라서 사이 사이 공백을 넣어 출력했다.

이 사례가 여러분이 처음 경험하게 되는 시맨틱 오류이다. 코드가 예리 메세지 없이 실행이되지만, ”올바르게(right)” 작동을 하는 것은 아니다.

2.2 변수(Variable)

프로그래밍 언어의 가장 강력한 기능중의 하나는 변수를 다룰 수 있는 능력이다. **변수(Variable)**는 값을 참조할 수 있는 이름이다.

할당 스테이스먼트(Assignment statement)는 새로운 변수를 생성하고 값을 준다.

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

위의 예제는 세개의 할당을 보여준다. 첫번째 할당의 예제는 message 변수에 문자열을 할당한다. 두번째 예제는 변수 n에 정수 17을 할당한다. 세번째 예제는 pi 변수에 π 근사값을 할당하는 경우이다.

변수의 값을 출력하기 위해서 print 스테이트먼트를 사용한다.

```
>>> print n
17
>>> print pi
3.14159265359
```

변수의 형식은 변수가 참조하는 값의 형식이다.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

2.3 변수명(Variable name)과 예약어(keywords)

대체로 프로그래머는 의미 있는 변수명을 고른다. 프로그래머는 변수가 어디에 사용되는지를 문서화도 한다.

변수명은 임의로 길 수 있다. 변수명은 문자와 숫자를 포함할 수 있지만, 통상 문자로 시작한다. 대문자를 사용하는 것도 적합하지만 소문자로 시작하는 변수명으로 시작하는 것도 좋은 생각이다. (후에 왜 그런지 보게될 것이다.)

변수명에 밑줄(underscore character, _)이 들어갈 수 있다. 밑줄은 `my_name` 혹은 `airspeed_of_unladen_swallow`처럼 여러 단어와 함께 사용된다.

적합하지 못한 변수명을 사용하면, 구문 오류를 보게된다.

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

`76trombones` 변수명은 문자로 시작하지 않아서 적합하지 않다. `more@`는 특수 문자 (@)를 변수명에 포함해서 적합하지 않다. `class` 변수명은 뭐가 잘못된 것일까?

`class`는 파이썬의 예약어 중의 하나이다. 인터프리터는 예약어를 프로그램 구조를 파악하기 위해서 사용하고 변수명으로는 사용할 수 없다.

파이썬의 31개의 키워드¹를 예약어로 이미 가지고 있다.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

여러분은 예약어 목록을 잘 가지고 다니고 싶을 것입니다. 인터프리터가 변수명 중에서 불평하고 이유를 모를 경우 예약어 목록에 있는지 확인해 보세요.

2.4 스테이트먼트(Statement)

스테이트먼트(statement)는 파이썬 인터프리터가 실행하는 코드의 단위입니다. `print`, `assignment` 두 종류의 스테이트먼트를 봤습니다.

인터랙트브 모드에서 스테이트먼트를 입력할 때, 만약 한줄이면 인터프리터는 스테이트먼트를 실행하고 결과를 출력합니다.

¹In Python 3.0, `exec` is no longer a keyword, but `nonlocal` is.

스크립트는 보통 여러줄의 스테이트먼트로 구성됩니다. 하나 이상의 스테이트먼트가 있다면, 스크립트가 실행되며 결과가 한번에 나타납니다.

예를 들어, 다음의 스크립트를 생각해 봅시다.

```
print 1
x = 2
print x
```

위 스크립트는 다음의 결과를 출력합니다.

```
1
2
```

할당 스테이트먼트 ($x=2$)는 결과를 출력하지 않습니다.

2.5 연산자(Operator)와 피연산자(Operands)

연산자(Operators)는 덧셈과 곱셈 같은 연산(Computation)을 표현하는 특별한 기호입니다. 연산가자 적용되는 값을 **피연산자(operands)**라고 합니다.

다음의 예제에서 보듯이, $+$, $-$, $*$, $/$, $**$ 연산자는 덧셈, 뺄셈, 곱셈, 나눗셈, 누승을 수행합니다.

```
20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)
```

나눗셈 연산자는 여러분이 기대하는 행동을 하지 않을 수도 있습니다.

```
>>> minute = 59
>>> minute/60
0
```

minute 값은 59, 보통 59를 60으로 나누면 0 대신에 0.98333 입니다. 이런 차이가 발생하는 이유는 파이썬이 부동 소수점 나눗셈을 하기 때문입니다.

두 개의 피연산자가 정수이면, 결과도 정수입니다. **부동 소수점 나눗셈²** 은 소수점 이하를 절사해서 이 예제에서는 소수점이하 잘라버려 0 이 됩니다.

```
>>> minute/60.0
0.9833333333333328
```

2.6 (표현)식(Expression)

(표현)식 (expression)은 값, 변수, 연산자의 조합입니다. 값은 자체로 표현식이고, 변수도 동일합니다. 따라서 다음의 표현식은 모두 적합합니다. (변수 x 에 사전에 어떤 값이 할당되었다고 가정하고)

²파이썬 3.0에서 이 나눗셈의 값은 소수점입니다. 파이썬 3.0에서 새로운 연산자 //는 정수 나눗셈을 수행합니다.

```
17
x
x + 17
```

인터랙티브 모드에서 표현식을 입력하면, 인터프리터는 표현식을 평가(evaluate)하고 값을 표시합니다.

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

하지만, 스크립트에서는 표현식 자체로 어떠한 것도 수행하지는 않습니다. 초심자에게 혼란스러운 점입니다.

Exercise 2.1 파이썬 인터프리터에 다음의 스테이트먼트를 입력하고 결과를 보세요.

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

2.7 연산자 적용 우선순위 (Order of Operations)

1개 이상의 연산자가 표현식에 등장할 때 연산자 실행의 순서는 순위 규칙(rules of precedence)에 따른다. 수학 연산자에 대해서 파이썬은 수학의 관례를 동일하게 따른다. 영어 두문어 PEMDAS는 기억하기 좋은 방식이다.

- **괄호(Parentheses)**는 가장 높은 순위를 가지고 여러분이 원하는 순위에 맞춰 실행할 때 사용한다. 괄호안에 있는 식이 먼저 실행되기 때문에 $2 * (3-1)$ 은 4가 정답이고, $(1+1)**(5-2)$ 는 8이다. 괄호를 표현식을 읽기 쉽게 하려고 사용하기도 한다. $(\text{minute} * 100) / 60$ 는 실행순서가 결과값에 영향을 주지 않지만 가독성이 상대적으로 좋다.
- **멱승(Exponentiation)**이 다음으로 높은 우선순위를 가진다. 그래서 $2**1+1$ 은 4가 아니라 3이고, $3*1**3$ 은 27이 아니고 3이다.
- **곱셈(Multiplication)**과 **나눗셈(Division)**은 덧셈(Addition), 뺄셈(Subtraction)보다 높은 우선 순위를 가지고 같은 실행의 순위를 갖는다. $2*3-1$ 은 4가 아니고 5이고, $6+4/2$ 는 5가 아니라 8이다.
- 같은 실행의 순위를 갖는 연산자는 왼쪽에서 오른쪽으로 실행된다. $5-3-1$ 표현식은 3이 아니고 1이다. 왜냐하면 5-3이 먼저 실행되고 나서 2에서 1을 빼기 때문이다.

여러분이 의도한 순서대로 연산이 일어날 수 있도록 좀 의심스러운 경우는 괄호를 사용하세요.

2.8 나머지 연산자 (Modulus Operator)

나머지 연산자(**modulus operator**)는 정수에 사용하며, 첫번째 피연산자가 두 번째 피연산자로 나눌 때 나머지 값을 만들어 냅니다. 파이썬에서 나머지 연산자는 퍼센트 기호(%)입니다. 구문은 다른 연산자와 동일합니다.

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

7을 3으로 나누면 2가 되고 1을 나머지로 갖게됩니다.

나머지 연산자가 놀랍도록 유용합니다. 예를 들어 한 숫자가 다른 숫자로 나눌 수 있는지 없는지를 확인할 수도 있습니다. $x \% y$ 가 0이라면 x 는 y 로 나눌 수 있습니다.

또한, 숫자로부터 가장 오른쪽의 숫자를 분리하는데 사용할 수도 있습니다. 예를 들어 $x \% 10$ 은 x 의 10진수인 경우 가장 오른쪽 숫자를 뽑아낼 수 있고, 동일하게 $x \% 100$ 은 가장 오른쪽 2개 숫자를 뽑아낼 수도 있습니다.

2.9 문자열 연산자 (String Operator)

+ 연산자는 문자열에도 사용할 수 있지만, 수학에서의 덧셈의 의미는 아닙니다. 대신에 문자열의 끝과 끝을 연결하여 **접합(concatenation)**을 수행합니다. 예를 들어

```
>>> first = 10
>>> second = 15
>>> print first+second
25
>>> first = '100'
>>> second = '150'
>>> print first + second
100150
```

이 프로그램의 출력은 100150입니다.

2.10 사용자에게서 입력값 받기

때때로 키보드를 통해서 사용자에게서 변수에 대한 값을 받고 싶을 때가 있습니다. 키보드³로부터 입력값을 받을 수 있는 `raw_input`이라는 빌트인(built-in) 함수를 파이썬은 제공합니다. 이 함수가 실행될 때 파이썬은 실행을 멈추고 사용자로부터 입력 받기를 기다립니다. 사용자가 Return 혹은 엔터를 누르게되면

³파이썬 3.0에서 이 함수는 `input`으로 명명되었습니다.

프로그램은 다시 실행되고 raw_input은 사용자가 입력한 값을 문자열로 반환합니다.

```
>>> input = raw_input()
Some silly stuff
>>> print input
Some silly stuff
```

사용자로부터 입력을 받기 전에 프롬프트에서 사용자에게 어떤 값을 입력해야하는지 출력하는 것도 좋은 생각이다. 입력값을 위해 잠시 멈춰있기 전에 raw_input 함수에 출력될 문자열에 대한 정보를 사용자에게 전달하는 것이다.

```
>>> name = raw_input('What is your name?\n')
What is your name?
Chuck
>>> print name
Chuck
```

프롬프트의 끝에 \n 은 새줄(newline)을 의미합니다. 새줄은 줄을 바꾸게 하는 특수 문자입니다. 이런 이유 때문에 사용자의 입력이 프롬프트 밑에 출력이 됩니다.

만약 사용자가 정수를 입력하기를 바란다면, int() 함수를 사용하여 반환되는 값을 정수(int)로 형변환할 수 있습니다.

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

하지만, 사용자가 숫자 문자열이 아닌 다른 것을 입력하게 되면 오류가 발생합니다.

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

이런 종류의 오류를 나중에 더 만나게 될 것입니다.

2.11 주석(Comment)

프로그램이 커지고 복잡해짐에 따라 읽기는 점점 어려워집니다. 형식언어로 촘촘하고 코드의 일부분을 읽기 어렵고 무슨 역할을 왜 수행하는지 이해하기 어렵습니다.

이런 이유로 프로그램이 무엇을 하는지를 일반적인 언어로 프로그램에 노트를 달아놓는게 좋은 습관입니다. 이러한 노트를 주석(**Comments**)라고 하고 # 기호로 시작합니다.

```
# 경과한 시간의 퍼센트를 계산
percentage = (minute * 100) / 60
```

이 경우, 주석 자체가 한줄이다. 주석을 프로그램의 뒤에 놓을 수도 있다.

```
percentage = (minute * 100) / 60      # 경과한 시간의 퍼센트를 계산
```

뒤의 모든 것은 무시되기 때문에 프로그램에는 아무런 영향이 없습니다. 주석은 코드의 명확하지 않은 점을 문서화할 때 가장 유용합니다. 프로그램을 읽는 사람이 코드가 무엇을 하는지 이해할 수 있다고 가정하는 것은 그럴 듯합니다. 왜 그런지를 설명하는 것은 더더욱 유용합니다.

다음의 주석은 코드와 중복으로 쓸모가 없습니다.

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

다음의 주석은 코드에 없는 유용한 정보가 있습니다.

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

좋은 변수명은 주석을 할 필요를 없게 만들지만, 지나치게 긴 변수명은 읽기 어려운 복잡한 표현식이 될 수도 있기 때문에 상충관계(trade-off)가 존재합니다.

2.12 기억하기 쉬운 변수명 만들기

변수를 이름 짓는 간단한 규칙을 따르고, 예약어를 피하기만 하면 변수를 이름지 을 수 있는 무척이나 많은 경우의 수가 존재합니다. 처음에 이러한 선택의 폭이 프로그램을 읽는 사람이나 프로그램을 작성하는 사람 모두에게 혼란스러울 수 있습니다. 예를 들어, 다음의 3개 프로그램은 수행하는 것이 동일하다는 점에서 동일하지만 여러분이 읽고 이해하는데는 많은 차이점이 있습니다.

```
a = 35.0
b = 12.50
c = a * b
print c

hours = 35.0
rate = 12.50
pay = hours * rate
print pay

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print x1q3p9afd
```

파이썬 인터프리터는 이들 3개 프로그램을 정확하게 동일하게 바라보지만, 사람은 이들 프로그램을 매우 다르게 바라보고 이해하게 됩니다. 사람은 가장 빨리

두번째 프로그램의 의도를 알아차립니다. 왜냐하면 프로그래머가 변수에 저장되는 값에 관계없이 프로그래머의 의도를 나타내는 변수명을 사용했기 때문입니다.

현명하게 선택된 변수명을 기억하기 쉬운 변수명(*"mnemonic variable name"*)이라고 합니다. 기억하기 좋은 영어 단어 "mnemonic"⁴은 기억을 돋는다는 뜻입니다. 왜 변수를 생성했는지 기억하기 좋게 하기 위해서 기억하기 좋은 변수명을 선택합니다.

매우 훌륭하게 들리고, 기억하기 좋은 변수명을 만드는게 좋은 아이디어 같지만, 기억하기 좋은 변수명은 초보 프로그래머가 코드를 파싱하고 이해하는데 걸림돌이 되기도 한다. 왜냐하면 31개의 예약어도 기억하지 못하고 때때로 너무 서술적인 이름의 변수가 마치 우리가 일반적으로 사용하는 언어처럼 보여 잘 선택된 변수명처럼 보이지 않기 때문이다.

어떤 데이터를 반복하는 다음의 파이썬 코드를 살펴보자. 여러분은 곧 반복 루프를 보지만 이것이 무엇을 의미하는지 알기 위해서 퍼즐을 풀기 시작할 것이다.

```
for word in words:  
    print word
```

무엇이 일어나고 있는 것일까요? for, word, in 등등 어느 것이 예약어일까요? 변수명은 무엇일까요? 파이썬이 기본적으로 단어의 개념을 이해할까요? 초보 프로그래머는 어떤 부분의 코드가 이 예제와 동일해야 하는지와 단지 프로그래머의 선택에 의한 부분이 코드의 어느부분인지 분간하는데 애를 먹는다.

다음의 코드는 위의 코드와 동일하다.

```
for slice in pizza:  
    print slice
```

초보 프로그래머가 이 코드를 보고 어떤 부분이 파이썬의 예약어이고 어느 부분이 프로그래머가 선택한 변수명인지 알 수 있다. 파이썬이 피자와 피자조각에 대한 근본적인 이해가 없고 피자는 하나 혹은 여러 조각으로 구성된다는 근본적인 사실을 알지 못한다.

하지만, 여러분의 프로그램이 데이터를 읽고 데이터의 단어를 찾는다면 피자(pizza)와 피자조각(slice)은 기억하기 좋은 변수명이 아니다. 이를 변수명은 프로그램의 의미를 왜곡시킬 수 있다.

좀 시간을 보낸 후에 흔한 예약어에 대해서 알게될 것이고 이들 예약어가 여러분에게 눈에 띄게 될 것이다.

```
for word in words:  
    print word
```

for, in, print, :은 파이썬에서 정의된 예약어로 굵게 표시되어 있고, 프로그래머가 생성한 word, words는 굵게 표시되어 있지 않다. 많은 텍스트 에디터는 파이썬의 구문을 알고 있고 파이썬 예약어와 프로그래머의 변수를 구분하기 위해서 다른 색깔로 구분지어준다. 잠시 후에 여러분은 이제 파이썬을 읽고 변수와 예약어에 대해서 빨리 구분할 수 있을 것이다.

⁴<http://en.wikipedia.org/wiki/Mnemonic>.

2.13 디버깅(Debugging)

이 지점에서 여러분이 만들 것 같은 구문 오류는 `odd~ job`, `US$` 같은 특수문자 를 포함한 잘못된 변수명과 `class`, `yield` 같은 예약어를 변수명으로 사용하는 것이다.

변수명에 공백을 넣는다면, 파이썬은 연산자 없이 두 개의 피연산자로 생각합니다.

```
>>> bad name = 5
SyntaxError: invalid syntax
```

구문 오류에 대해서, 구문오류 메세지는 그다지 도움이 되지 못합니다. 가장 흔한 오류 메세지는 `SyntaxError: invalid syntax`, `SyntaxError: invalid token`인데 둘다 그다지 도움이 되지 못합니다.

여러분이 많이 만드는 실행 오류는 정의전에 사용("use before def")하는 것으로 변수에 값을 할당하기 전에 변수를 사용할 경우 발생합니다. 여러분이 변수명을 쓸 때도 발생할 수 있습니다.

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

변수명은 대소문자를 구분합니다. `LaTeX`는 `latex`와 같지 않습니다.

이 지점에서 여러분이 저지르기 쉬운 구문 오류는 연산자의 우선 순위일 것입니다. 예를 들어 $\frac{1}{2\pi}$ 를 계산하기 위해서 다음과 같이 프로그램을 작성하게 되면

...

```
>>> 1.0 / 2.0 * pi
```

나눗셈이 먼저 일어나서 $\pi/2$ 를 같은 것이 아닙니다. 파이썬이 여러분이 쓴 의도를 알게할 수는 없습니다. 그래서 이런 경우 오류 메세지를 얻지는 않지만 잘못된 답을 여러분은 얻게 될 것입니다.

2.14 용어 설명

할당(assignment): 변수에 값을 할당하는 스테이트먼트

결합(concatenate): 두 개의 피연산자 끝과 끝을 합치는 것

주석(comment): 다른 프로그래머나 소스코드를 읽는 다른 사람을 위한 프로그램의 정보로 프로그램의 실행에는 아무런 영향이 없다.

평가(evaluate): 하나의 값을 만들도록 연산을 실행함으로써 표현식을 간단히 하는 것

(표현)식(expression): 하나의 결과값을 만드는 변수, 연산자, 값의 조합

부동 소수점(floating-point): 분수를 가진 숫자를 표현하는 방식

플로어 나눗셈(floor division) 두 숫자를 나누어 소수점이하 부분을 절사하는 연산자

정수(integer): 완전수를 나타내는 형식

예약어(keyword): 프로그램을 파싱하는 컴파일러가 사용하는 이미 예약된 단어; if, def, while 같은 예약어를 변수명으로 사용할 수 없다.

니모닉(mnemonic): 기억 보조, 변수에 저장된 것을 기억하기 좋게 변수에 니모닉 이름을 부여한다.

나머지 연산자(modulus operator): 퍼센트 기호 (%) 로 표시되고 정수를 가지고 한 숫자를 다른 숫자로 나누었을 때 나머지

피연산자(operand): 연산자가 연산을 수행하는 값의 하나

연산자(operator): 덧셈, 곱셈, 문자열 결합 같은 간단한 연산을 나타내는 특별 기호

순위 규칙(rules of precedence): 여러 개의 연산자와 피연산자를 포함한 표현식이 평가되는 실행 순서를 정한 규칙 집합

스테이트먼트(statement): 명령이나 액션을 나타내는 코드 부문. 지금까지 assignment, print 스테이트먼트를 보았습니다.

문자열(string): 일련의 문자를 나타내는 형식

형(type): 값의 범주. 지금까지 여러분이 살펴본 형은 정수 (type int), 부동 소수점수 (type float), 문자열 (type str)입니다.

값(value): 프로그램이 다루는 숫자나 문자 같은 데이터의 기본 단위중 하나

변수(variable): 값을 참조하는 이름

2.15 연습문제

Exercise 2.2 `raw_input`을 사용하여 사용자의 이름을 입력받고 환영하는 프로그램을 작성하세요.

```
Enter your name: Chuck  
Hello Chuck
```

Exercise 2.3 급여를 지불하기 위해서 사용자로부터 근로시간과 시간당 임금을 계산하는 프로그램을 작성하세요.

```
Enter Hours: 35  
Enter Rate: 2.75  
Pay: 96.25
```

지금은 급여가 정확하게 소수점 두자리까지 표현되지 않아도 된다. 만약 원한다면, 파이썬의 빌트인 `round` 함수를 사용하여 소수점 아래 두자리까지 반올림하여 작성할 수 있다.

Exercise 2.4 다음 할당 스테이트먼트를 실행한다고 합시다.

```
width = 17  
height = 12.0
```

다음 표현식에 대해서 표현식의 값과 표현식의 값의 형을 작성하세요.

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

정답을 확인하기 위해서 파이썬 인터프리터를 사용하세요.

Exercise 2.5 사용자에게서 섭시 온도를 입력받아 화씨온도를 변환하고 변환된 온도를 출력하는 프로그램을 작성하세요.

Chapter 3

조건부 실행

3.1 불 연산식(Boolean expressions)

불 연산식(boolean expression)은 참(True) 혹은 거짓(False)를 가진 연산 표현식이다. 다음 예제는 == 연산자를 사용하는데 두개의 피연산자를 비교하여 값이 동일하면 참(True), 그렇지 않으면 거짓(False)을 출력한다.

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

참(True)과 거짓(False)은 불(bool) 형식에 속하는 특별한 값들이고, 문자열은 아니다.

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

==연산자는 비교(comparison operators) 연산자 중의 하나이고, 다른 연산자는 다음과 같다.

x != y	# x는 y와 값이 같지 않다.
x > y	# x는 y보다 크다.
x < y	# x는 y보다 작다.
x >= y	# x는 y보다 크거나 같다.
x <= y	# x는 y보다 작거나 같다.
x is y	# x는 y와 같다.
x is not y	# x는 y와 개체가 동일하지 않다.

여러분에게 이들 연산자가 친숙할지 모르지만, 파이썬 기호는 수학 기호와는 다르다. 일반적인 오류에는 비교의 같다의 의미로 == 연산자 대신에 =를 사용하는 것이다. = 연산자는 할당 연산자이고, ==연산자는 비교 연산자이다. =<, => 비교 연산자는 파이썬에는 없다.

3.2 논리 연산자

3개의 논리 연산자(logical operators): and, or, not가 있다. 논리 연산자 의미는 영어 의미와 유사하다. 예를 들어,

`x > 0 and x < 10`

`x > 0` 보다 크다. 그리고(and), 10 보다 작으면 참이다.

`n%2 == 0 or n%3 == 0`은 두 조건문 중의 하나만 참이되면, 즉, 숫자가 2 혹은(or) 3으로 나누어진다면 참이다.

마지막으로 not 연산자는 불 연산 표현을 부정한다. `x > y`이 거짓이면, `not (x > y)`은 참이다. 즉, `x`이 `y` 보다 작거나 같으면 참이다.

엄밀히 말해서, 논리 연산자의 두 피연산자는 모두 불 연산 표현이여야 하지만, 파이썬은 그렇게 엄격하지는 않는다. 어떤 0이 아닌 숫자 모두 ”true”로 해석된다.

```
>>> 17 and True
True
```

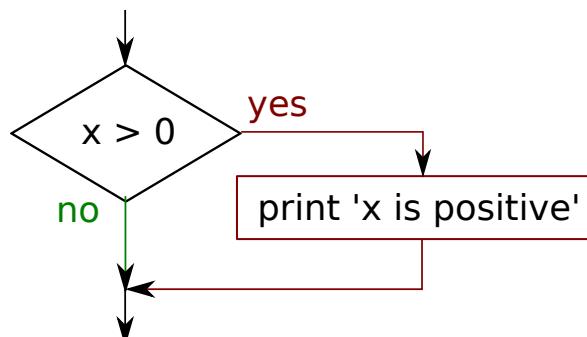
이러한 유연함이 유용할 수 있으나, 혼란을 줄 수도 있으니 유의해서 사용해야 됩니다. 무슨 일을 하고 있는지 정확하게 알지 못한다면 피하는게 좋습니다.

3.3 조건문 실행

유용한 프로그램을 작성하기 위해서는 조건을 확인하고 조건에 따라 프로그램의 실행을 바꿀 수 있어야 한다. 조건문(Conditional statements)은 그럴 수 있는 능력을 부여한다. 가장 간단한 형태는 if 문이다.

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

if문 뒤에 불 연산 표현문을 조건(condition)이라고 한다.



만약 조건문이 참이면, 들여쓰기 된 스테이트먼트가 실행된다. 만약 조건문이 거짓이면, 들여쓰기 된 스테이트먼트의 실행을 생략한다.

if문은 함수 정의나 for 반복문과 동일한 구조를 가진다. if문은 콜론(:)으로 끝나는 헤더 머리부문과 들여쓰기 블록으로 구성된다. if문과 같은 구문을 한 줄 이상에 걸쳐 작성되기 때문에 **복합문(compound statements)**이라고 한다.

if문 본문에 실행 명령문의 제한은 없으나 최소한 한 줄은 있어야 한다. 때때로, 본문에 하나의 실행명령문이 없는 경우가 있다. 아직 코드를 작성하지 않고 자리를 잡아 놓는 경우로, 아무것도 수행하지 않는 pass문을 사용할 수 있다.

```
if x < 0 :  
    pass          # 음수값을 처리 예정!
```

if문을 파이썬 인터프리터에서 타이핑하고 엔터를 치게 되면 명령 프롬프트가 갈매기 세마리에서 점 세개로 바뀌어 본문을 작성중에 있다는 것을 다음과 같이 보여줍니다.

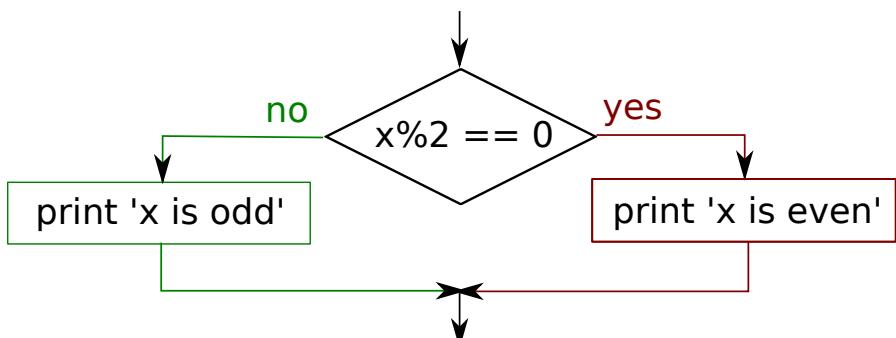
```
>>> x = 3  
>>> if x < 10:  
...     print 'Small'  
...  
Small  
>>>
```

3.4 대한 실행

if문의 두 번째 형태는 **대안 실행(alternative execution)**으로 두 가지 경우의 수가 존재하고, 조건이 어느 방향인지를 결정한다. 구문(Syntax)은 아래와 같다.

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

x가 2로 나누었을 때, 0이 되면, x는 짝수이고, 프로그램은 짝수 결과 메시지를 출력한다. 만약 조건이 거짓이라면, 두 번째 명령문 블록이 실행된다.



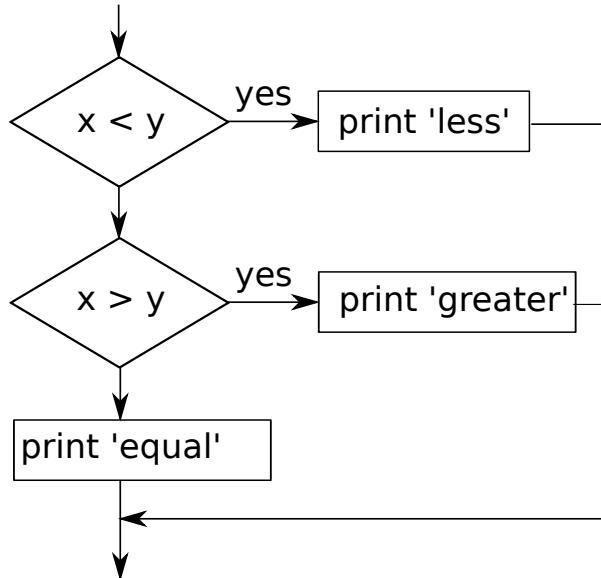
조건은 참 혹은 거짓이서, 대안 중 하나만 정확하게 실행될 것이다. 대안을 **분기(Branch)**라고도 하는데 이유는 실행 흐름의 분기가 되기 때문이다.

3.5 연쇄 조건문

때때로, 두 가지 이상의 경우의 수가 있으며, 두 가지 이상의 분기가 필요하다. 이 같은 연산을 표현하는 방법이 **연쇄 조건문(chained conditional)**이다.

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

`elif`는 ”`else if`”의 축약어이다. 이번에도 단 한번의 분기만 실행된다.



`elif`문의 갯수에 제한은 없다. `else`문이 있다면, 거기서 끝마쳐야 하지만, 연쇄 조건문에 필히 있어야 하는 것은 아니다.

```
if choice == 'a':
    print 'Bad guess'
elif choice == 'b':
    print 'Good guess'
elif choice == 'c':
    print 'Close, but not correct'
```

각 조건은 순서대로 점검한다. 만약 첫 번째가 거짓이면, 다음을 점검하고 계속 점검해 나간다. 순서대로 진행 주에 하나의 조건이 참이면, 해당 분기가 수행되고, `if`문 전체는 종료된다. 설사 하나 이상의 조건이 참이라고 하더라도, 첫 번째 참 분기만 수행된다.

3.6 중첩 조건문

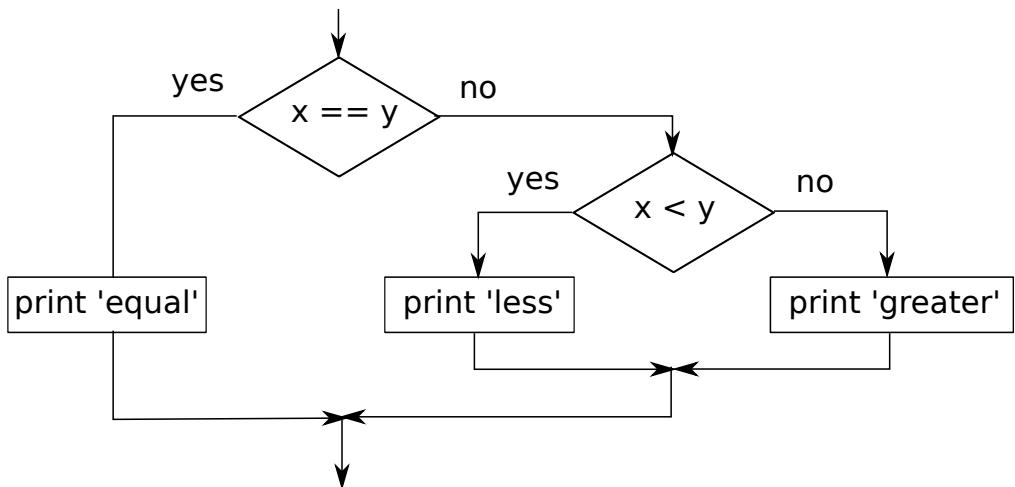
하나의 조건문이 조건문 내부에 중첩될 수도 있다. 다음처럼 삼분 예제를 작성할 수 있다.

```

if x == y:
    print 'x and y are equal'
else:
    if x < y:
        print 'x is less than y'
    else:
        print 'x is greater than y'

```

바깥 조건문에 두 개의 분기가 있다. 첫 분기는 간단한 명령 실행문을 담고 있다. 두 번째 분기는 자체가 두 개의 분기를 가지고 있는 또 다른 if문을 담고 있다. 자체로 둘다 조건문이지만, 두 분기 모두 간단한 실행 명령문이다.



Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```

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

```

The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```

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

```

3.7 Catching exceptions using try and except

Earlier we saw a code segment where we used the `raw_input` and `int` functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

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

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think “oops” and move on to our next statement.

However if this code is placed in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”. The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the try and except feature in Python as an “insurance policy” on a sequence of statements.

We can rewrite our temperature converter as follows:

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print 'Please enter a number'
```

Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short circuit evaluation of logical expressions

When Python is processing a logical expression such as `x >= 2 and (x/y) > 2`, it evaluates the expression from left-to-right. Because of the definition of `and`, if `x` is less than 2, the expression `x >= 2` is `False` and so the whole expression is `False` regardless of whether `(x/y) > 2` evaluates to `True` or `False`.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called **short-circuiting** the evaluation.

While this may seem like a fine point, the short circuit behavior leads to a clever technique called the **guardian pattern**. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

The third calculation failed because Python was evaluating (x/y) and y was zero which causes a runtime error. But the second example did *not* fail because the first part of the expression $x \geq 2$ evaluated to `False` so the (x/y) was not ever executed due to the **short circuit** rule and there was no error.

We can construct the logical expression to strategically place a **guard** evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

In the first logical expression, $x \geq 2$ is `False` so the evaluation stops at the `and`. In the second logical expression $x \geq 2$ is `True` but $y \neq 0$ is `False` so we never reach (x/y) .

In the third logical expression, the $y \neq 0$ is *after* the (x/y) calculation so the expression fails with an error.

In the second expression, we say that $y \neq 0$ acts as a **guard** to insure that we only execute (x/y) if y is non-zero.

3.9 Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>> y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

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

But when you run it, you get an error message¹:

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

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, because dividing two integers does floor division. The solution is to represent signal power and noise power with floating-point values.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

3.10 Glossary

body: The sequence of statements within a compound statement.

boolean expression: An expression whose value is either `True` or `False`.

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

comparison operator: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

¹In Python 3.0, you no longer get an error message; the division operator performs floating-point division even with integer operands.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch is executed.

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

guardian pattern: Where we construct a logical expression with additional comparisons to take advantage of the short circuit behavior.

logical operator: One of the operators that combines boolean expressions: and, or, and not.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

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

short circuit: When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

3.11 Exercises

Exercise 3.1 Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

```
Enter Hours: 45
```

```
Enter Rate: 10
```

```
Pay: 475.0
```

Exercise 3.2 Rewrite your pay program using try and except so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

```
Enter Hours: 20
```

```
Enter Rate: nine
```

```
Error, please enter numeric input
```

```
Enter Hours: forty
```

```
Error, please enter numeric input
```

Exercise 3.3 Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range print an error. If the score is between 0.0 and 1.0, print a grade using the following table:

Score	Grade
≥ 0.9	A
≥ 0.8	B
≥ 0.7	C
≥ 0.6	D
< 0.6	F

Enter score: 0.95

A

Enter score: perfect

Bad score

Enter score: 10.0

Bad score

Enter score: 0.75

C

Enter score: 0.5

F

Run the program repeatedly as shown above to test the various different values for input.

