# 데이터 과학 프로그래밍

한국 알(R) 사용자회

2022-10-17

# Contents

# 기계와의 경쟁 [1]

기계와의 경쟁에서 인간이 승리할 수 있는 초석이 되고 있는 소프트웨어 카펜트리(Software Carpentry) 와 데이터 카펜트리(Data Carpentry) 비영리 공공 프로젝트가 많은 영감을 주었고, 특히 2015년 Python for Informatics: Exploring Information (Korean Edition) 한국어 오픈 프로젝트는 새로운 가능성을 보여줬습니다. 2018년부터 **소프트웨어 교육** 이 초중등 교육과정에 의무화되었고, 알파고가 이세돌을 이긴 **알파고 쇼크** 가 한국사회에 엄청난 파장을 일으켰다. 그와 더불어 청년, 중장년, 노년 할 것 없이 실업률이 사회적 문제로 대두되면서 기계가 인간의 직업을 빼앗아가는 주범으로 주목되고 있는 한편, **인공지능** 기술을 밑에 깔고 있는 다양한 제품과 서비스가 쏟아지면서 우리의 삶을 그 어느 때보다 풍요롭게 만들고 있다.

컴퓨팅 사고력, 데이터 과학, 인공지능, 로봇/기계를 이해하는 사람과 그렇지 못한 사람간에 삶의 질 차이는 점점더 현격히 벌어질 것이다. 지금이라도 늦지 않았다. 늦었다는 것을 알아차렸을 때가 가장 빠른 시점이다.

유발 하라리 교수가 지적했듯이 데이터가 권력과 부의 원천이 되는 세상으로 접어들었는데 이에 대해서 컴퓨터와 적절하게 의사소통할 수 있는 언어가 필요하다. **R**는 일반인에게 많이 알려져 있지 않지만, 파이썬과 더불어 데이터 프로그래밍에 있어 유구한 역사와 탄탄한 사용자 기반을 가지고 최근들어 혁신적인 변화를 이끌고 있는 언어 중의 하나다.

> "The future is here, it's just not evenly distributed yet."
>
> - William Gibson

> "고대에는 '땅'이 가장 중요했고 땅이 소수에게 집중되자 인간은 귀족과 평민으로 구분됐으며, 근대에는 '기계'가 중요해지면서 기계가 소수에게 집중되자 인간은 자본가와 노동자 계급으로 구분됐다". 이제는 **데이터**가 또 한번 인류를 구분하는 기준이 될 것이다. 향후 데이터가 소수에게 집중되면 단순 계급에 그치는 게 아니라 데이터를 가진 종과 그렇지 못한 종으로 분류될 것이이다. [2]
>
> - 유발 하라리(Yuval Noah Harari)

[1] '사피엔스' 저자 유발 하라리 "인간을 해킹하는 시대가 온다", "머신러닝·AI·생물학 발전…뇌과학 이해도 한층 높여"

[2] '사피엔스' 저자 유발 하라리 "인간을 해킹하는 시대가 온다", "머신러닝·AI·생물학 발전…뇌과학 이해도 한층 높여"

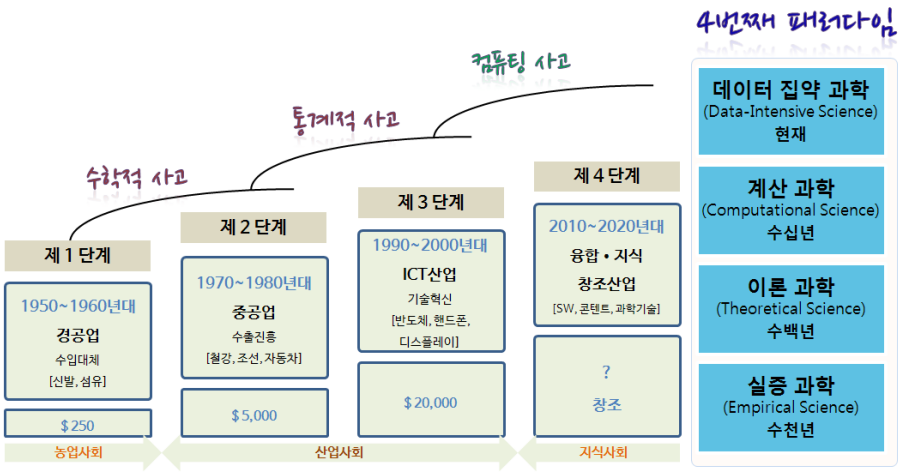



Figure 1: 패러다임 전환

---

사단법인 한국 알(R) 사용자회는 디지털 불평등 해소와 통계 대중화를 위해 2022년 설립되었습니다. 오픈 통계 패키지 개발을 비롯하여 최근에 데이터 사이언스 관련 교재도 함께 제작하여 발간하는 작업을 수행하고 있습니다. 그 첫번째 결과물로 John Fox 교수님이 개발한 설치형 오픈 통계 패키지 `Rcmdr`(Fox 2016) (Fox and Bouchet-Valat 2021) (Fox 2005) 를 신종화 님께서 한글화 및 문서화에 10년 넘게 기여해주신 한국알사용자회 저작권을 흔쾌히 허락해 주셔서 설치형 오픈 통계 패키지 - `Rcmdr`로 세상에 나왔습니다.

두번째 활동을 여기저기 산재되어 있던 시각화 관련 자료를 묶어 **데이터 시각화(Data Visualization)**를 전자책 형태로 공개하였고, 데이터 분석 관련 저술을 이어 진행하게 되었습니다.

세번째 활동으로 데이터 사이언스가 하나로 구성되지 않은 것을 간파하고 데이터 사이언스를 지탱하는 기본기술을 5가지로 정리한 **데이터 과학을 지탱하는 기본기** 전자책을 공개했습니다.

데이터 분석 언어 R에 관한 지식을 신속히 습득하여 독자들이 갖고 있는 문제에 접목시키고자 하시는 분은 한국 알(R) 사용자회에서 번역하여 공개한 R 신병훈련소(Bootcamp) 과정을 추천드립니다.

---

관련 문의와 연락이 필요한 경우 한국 알(R) 사용자회 admin@r2bit.com 대표전자우편으로 연락주세요.

---

**후원계좌**

디지털 불평등 해소를 위해 제작중인 오픈 통계패키지 개발과 고품질 콘텐츠 제작에 큰 힘이 됩니다.

- 하나은행 448-910057-06204
- 사단법인 한국알사용자회

# 들어가며

# Chapter 1

# R vs 파이썬

## 1.1 R 역사

Revolutions[1]에서 정리한 최근 R 역사는 1992년 처음 뉴질랜드 오클랜드에서 Robert Gentleman, Ross Ihaka 교수가 개발을 시작한 후에 GPL 라이선스를 장착하여 소스코드를 공개한 후에 R 코어 그룹이 만들어지고, 팩키지 배포 CRAN이 순차적으로 공개되고 나서, R 웹사이트가 만들어지고, 처음으로 2000년에 R 1.0.0으로 배포되고 R 저널, UseR! 컨퍼런스, R 재단, R 컨소시엄이 전세계 수많은 재능있고 열정있는 수많은 사람에 의해서 만들어졌습니다.

## 1.2 R 우수성

한국 R 사용자회 일원으로 코딩 바람이 시작하던 2015년 세브란스 교수의 "정보교육을 위한 파이썬(Python for informatics)"를 번역하여 오픈 전자책으로 GitHub과 웹사이트 공개하면서 많은 경험을 하게 되었다. 파이썬이 좋은 프로그래밍 언어이기는 하지만 곧이어 접한 tidyverse 체계를 따르는 R 언어와는 비교가 불가능한 것 같다. 예를 들어 판다스 데이터프레임 구문은 과거 Base R 구문을 그대로 따르고 있다. 하지만 R은 과거 Base R의 경험을 바탕으로 한단계 업그레이된 `tidyverse` 체계로 진화했기 때문이다. 관련 자세한 사항은 데이터 사이언스 운영체제 - `tidyverse`, 한국통계학회 소식지 2019년 10월호에 정리되어 있다.

`tidyverse` R을 접하면서 저자는 파이썬을 손을 놓고 딥러닝 관련 새로운 기술이 나올 때 종종 필요에 따라 사용하고 있으며 대부분의 데이터 작업은 R을 사용하고 있다. 이런 경험은 저자에게만 국한된 것이 아니고 Alfonso R. Reyes의 경험과도 일치하고 관련 내용을 촘촘하게 정리하였기에 출처를 밝히고 번역하여 함께 합니다.[2]

[1] Revolutions (2017), An Updated History of R

[2] Alfonso R. Reyes, Chief Data Scientist at AEM Enersol (September 11, 2018), "For what things R programming language is better than Python?", LinkedIn

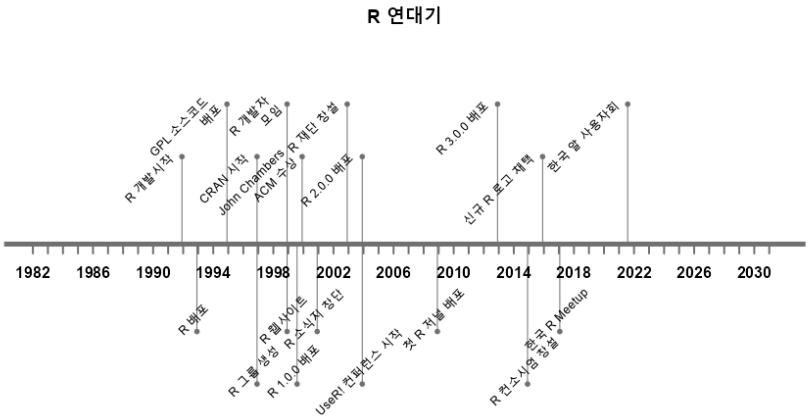



Figure 1.1: R 연대기 및 한국 R 사용자회

저자는 10년이상 파이썬 응용프로그램을 작성했지만, 지금은 R을 사용하고 있다. R을 사용하진 2년이 지난 시점에서 파이썬 보다 R의 우수성에 대해서 10가지로 요약하고 있다.

간략히 요약하면, R은 데이터 과학과 기계학습 프로젝트에 이례적인 도구라고 볼 수 있다. R로 개발할 때 생산성은 훨씬 더 크게 느껴진다. 하지만, R을 익숙하게 다루는데는 시간이 다소 소요되기 때문에 시간에 대한 투자도 당근 고려되어야 한다. 특히, 파이썬이 근접할 수 없는 분야는 프로토타입을 개발할 때 재현성을 비롯한 상당한 매력과 장점이 있다. 파이썬은 다용도 언어로 데이터 과학 선택지로서 입지를 다져가고 있다.

1. R은 과학, 통계학, 수학, 공학에 집중한다. 따라서 과학과 공학에 R로 접근하는 것이 엄청 필요하다.
2. 지원에 대해서는 세계적이다. 어디를 가나 StackOverflow, 포럼, 트위터, 링크트인, 팟캐스트 등을 쉽게 접할 수 있고, R 커뮤니티 자체가 매우 이타적이라 모두 기꺼이 도움을 주고 받고 해서 R 실력이 더 향상되도록 한다.
3. 재현성(reproducibility)이 황우석 사태 이후 큰 주목을 받고 있다. 재현성을 구현하는데 팩키지가 이미 다수 개발되어 있고 진화를 거듭하고 있다. 작업한 결과물은 팀뿐만 아니라, 외부에서도 재현되어야 하는데, 데이터 과학의 궁극의 목적으로 재현성을 최극단까지 도달할 수 있도록 R 커뮤니티가 노력을 경주하고 있다.
4. 로마시절에 라틴어가 만인의 언어이고, 현재 영어가 전세계 공용어이듯이, R마크다운은 R에서 공용 의사소통언어로 자리 잡았다. R마크다운으로 보고서, 슬라이드, 학위논문, 논문, 책을 집필할 수 있다. R마크다운을 쥬피터 노트북과 비교하면 생산성이 1000:1 정도 될 것이다. 하지만, 반대로 R을 학습하는데 상당한 시간을 투여해야 되는 것도 사실이다.

5. 정말 빠른 배포. RStudio 회사 직원이 어떻게 하는지 모르겠지만, 어쨌든 제품을 단 1,2년만에 뚝딱 만들어 냈다. Shiny를 말하고 싶은데, 파이썬으로 몇주 몇달 걸리는 것과 비교하여 단지, 몇분, 몇시간이면 뚝딱 웹앱을 배포할 수 있다. 다른 말로, 파이썬으로 개발한 것은 실무적으로 공유하기 힘들다(non-shareable).
6. 가장 단순하면서도 최상의 통합개발환경(IDE)가 발명되었다. RStudio는 매우 단순하지만, 동시에 개발에 확장이 가능한 개발환경을 제공하고 있다. 처음에 믿을 수가 없겠지만, 작은 창 4개를 가지고 어마어마한 작업을 수행할 수 있다. 단순하지만 더 단순할 수 없도록 만들었다.
7. 팩키지 품질관리. CRAN(the Comprehensive R Archive Network)에 팩키지를 올리려면 상당한 품질 기준을 만족해야 된다. 문서가 없는 팩키지는 승인되지 않는데, 이런 점이 현재 R을 만들었다. 현시점 기준 13,000개가 넘는 팩키지가 있는데 갯수가 중요한 것이 아니라, R을 강하고 확장가능하게 만든 것은 문서의 품질이라고 본다.
8. 확장성(extendable). 팩키지 품질관리로 통해서 품질 좋은 팩키지가 만들어져서 이후 만들어지는 팩키지는 이런 토대위에 제작되어 더 좋은 팩키지로 거듭남. 누가 가장 혜택을 볼 수 있을까? 바로 사용자.
9. 고품질 그래프. `matplotlib`에 관해 무엇을 언급하든, Base 그래프, 그래프 문법 ggplot2 그래프에는 근접하지 못한다. Alfonso R. Reyes는 matplotlib을 오랜동안 사용했지만, 비표준 그래프는 뭐든지 맨땅에서부터 작성해야 했다. 반대로 R은 ggplot2를 비롯한 방대한 기능을 아직도 모두 소진해본적은 없는듯 싶다. 물론, 그래프에 대해 지켜야할 몇가지 기본 규칙이 부수적으로 따라온다: 다중 y축을 사용하지 말고, 원그래프는 피하고, 3D 그래프를 과사용하지 말고, 특별한 이유가 없다면 맨 처음부터 그래프 작성을 시작한다.
10. 함수(function) 중심. 파이썬과 R을 기본적으로 다르게 만드는 것 중 하나다. 클래스는 파아썬에서 남용에 가깝게 사용되곤 한다. 때로는 특별한 이유도 없다. 클래스와 객체형 프로그래밍을 배우려고 한다면, 파이썬을 추천한다. 세상에서 가장 배우기 쉬운 것 중 하나다. 이런 관점에 반대하여 R 세상은 다르다; 함수가 R세상에서는 첫째 시민(first class citizen)이다. 다른 언어와 달리 R에서 클래스는 함수 아래서 동작한다. S3와 S4는 자바와 파이썬 클래스와는 근본적으로 다르게 동작한다; 가장 가까운 친적이 R6가 될 듯 싶다.

## 1.3 자료구조

우리나라에서 대부분 프로그래밍은 컴퓨터 과학 언플러그드, 리보그, 파이썬을 통해 학습을 하기 때문에 파이썬이 친숙할 것이다. 변수 선언, 제어, 함수, 반복 등은 동일한 개념을 표기법을 달리하기 때문에 R을 처음 접할 때 기본 프로그래밍 개념만 있다면 큰 어려움이 없지만 아마도 자료구조에 대해서 대응관계를 파악하게 되면 이후 학습 과정은 수월할 듯 싶다.

다행히 2018년부터 `reticulate` 패키지가 개발되면서 사실 데이터 문제를 R, 파이썬 혹은 "R과 파이썬"으로 각자 장점을 살펴 데이터 괴물을 처치하는 것이 가능해졌다. R Interface to Python에 R과 파이썬 자료구조가 잘 정리되어 있다.

| R | 파이썬 | 사례 |
|---|---|---|
| 단일 원소 벡터 | 스칼라 | 1, 1L, TRUE, “문자열” |
| 다중 원소 벡터 | 리스트 | c(1.0, 2.0, 30.), c(1L, 2L, 3L) |
| 다양한 자료형을 갖는 리스트 | 튜플 | list(1L, TRUE, “foo”) |
| 명칭갖는 리스트 | 딕셔너리 | `list(a = 1L, b = 2.0)`, `dict(x = x_data)` |
| 행열/배열 | 넘파이 ndarrary | matrix(c(1,2,3,4), nrow = 2, ncol = 2) |
| 데이터프레임 | 판다스 데이터프레임 | data.frame(x = c(1,2,3), y = c(“a”, “b”, “c”)) |
| 함수 | 파이썬 함수 | function(x) x + 1 |
| NULL, TRUE, FALSE | None, True, False | `NULL`, `TRUE`, `FALSE` |

## 1.4 통계패키지 구성요소 [3]

R은 현존하는 가장 강력한 통계 컴퓨팅 언어로, 그래픽과 자료분석을 위해 언어 + 팩키지 + 환경이 하나로 묶여있다. 특히, 컴퓨터 주기억장치 한계가 존재하지만, 오픈 소스로 모든 코드가 공개되어 있어 자유로이 이용이 가능하다. R은 John Chambers가 주축이 되어 벨연구소에서 개발된 유닉스와 역사를 함께하는 S을 Ross Ihaka 와 Robert Gentleman이 1996년 구현하여 대중에게 공개하였다.

자료분석을 위해 대중에게 널리 알려진 통계팩키지에는 SAS, SPSS, Stata, Minitab 등 상업용으로 많이 판매되고 있다. 어떤 통계팩키지도 다음과 같은 공통된 5가지 구성요소를 포함하고 있다.

- 데이터 입력과 조작 언어
- 통계와 그래픽 명령어
- 출력물 관리 시스템
- 매크로 언어
- 행렬 언어(SAS IML/SPSS Matrix/Stata Mata)

이와 비교하여 R은 5가지 구성요소가 **언어 + 팩키지 + 환경** 으로 구성된다는 점에서 차이가 크다.

## 1.5 두 언어 문제 [4]

두 언어 문제로 인해 편리함을 위해 R, 파이썬, Matlab을 사용하고 C/C++, 포트란 시스템 언어로 속도가 중요한 작업을 수행했다. 근본적으로 중국집에서 짜장이냐 짬뽕을 선택하느냐에 따라 두 언어가 갖는 장점이 너무나도 명확하다. 하지만, 시스템 언어와 스크립트 언어의 두가지 문제점을 해결하기 위해서 두가지 다른 언어의 장점을 취하고 이를 보완하려는 노력이 지속적으로 경주되고 있다.

[3] Muenchen, Robert A. R for SAS and SPSS users. Springer Science & Business Media, 2011.
[4] Ousterhout dichotomy

ODSC East 2016 - Stefan Karpinski - “Solving the Two Language Problem”

| 시스템 언어 | Ousterhout 이분법 | 스크립트 언어 |
|---|---|---|
| 정적 | - | 동적 |
| 컴파일 | - | 인터프리터 |
| 사용자정의 자료형 | - | 표준 자료형 |
| 빠른 속도 | - | 늦은 속도 |
| 어려움 | - | 쉬움 |

R Tidyverse 생태계에서 바라보는 또다른 관점은 기계처리시간(Machine time)과 사람생각시간(Human time)으로 나눠 바라보는 것이다. 기계관점을 바라보는 것이 파이썬이라면 사람 관점으로 바라보는 것이 R이라는 점에서 크게 대비된다.

## 1.6 데이터 과학 도구함

소프트웨어 개발과 통계 데이터 분석이 구분되어 독자적으로 발전되어 오다가 2000년대부터 데이터 사이언스의 원형이 되는 다양한 제품과 서비스들이 개발되면서 각 영역마다 특화된 도구들이 발전해나갔다. 선형대수는 매트랩, 통계 및 시각화는 R, 속도가 필요한 경우 C/C++, 통합작업을 위해 루비를 사용했다. 하지만, 2010년대에는 R과 파이썬에서 데이터를 기반으로 가치를 창출하는 소프트웨어 개발에 필요한 다양한 라이브러리와 패키지가 개발되면서 R 혹은 파이썬으로 전체 개발과정을 전담하고자 하는 노력이 생겨나면서 커다란 경쟁구도가 생겨났다. 하지만 2010년대 말부터 데이터 사이언스 문제를 가장 효율적인 도구를 통해 풀고자하면서 서로 다른 스킬셋을 갖춘 개발자들이 함께 개발하고 협업할 수 있는 환경과 경험이 축적되면서 이제는 R 과 파이썬을 모두 사용하는 방향으로 전환되었다.


Figure 1.2: 데이터 과학 툴체인

# Chapter 2

# 프로그래밍 학습 이유

컴퓨터 프로그램을 만드는 행위(프로그래밍)는 매우 창의적이며 향후 뿌린 것 이상으로 얻을 것이 많다. 프로그램을 만드는 이유는 어려운 자료분석 문제를 해결하려는 것에서부터 다른사람의 문제를 해결해주는데 재미를 느끼는 것까지 다양한 이유가 있다. 이 책을 통해서 모든 사람이 어떻게 프로그램을 만드는지를 알고, 프로그램이 어떻게 만드는지를 알게 되면, 새로 습득한 프로그래밍 기술로 하고자 하는 것을 해결할 수 있게 된다.

우리의 일상은 노트북에서부터 스마트폰까지 다양한 종류의 컴퓨터에 둘러싸여 있다. 이러한 컴퓨터를 우리를 위해서 많은 일을 대신해 주는 "개인비서"로 생각한다. 일상생활에서 접하는 컴퓨터 하드웨어는 우리에게 "다음에 무엇을 하면 좋겠습니까?" 라는 질문을 지속적으로 던지게 만들어 졌다.


Figure 2.1: 컴퓨터 소개

프로그래머는 운영체제와 하드웨어에 응용 프로그램을 추가했고, 결국 많은 것들을 도와주는 개인 휴대 정보 단말(Persoanl Digital Assistant, PDA)로 진화했다.

최근에는 인공지능 기능이 탑재된 스마트폰(Smart Phone)이 그 역할을 대신하고 있다.

사용자 여러분이 컴퓨터에게 "다음 실행해 (do next)"를 컴퓨터가 이해할 수 있는 언어로 지시를 할 수만 있다면, 컴퓨터는 빠르고, 저장소가 커서, 매우 유용하게 사용될 수 있다. 만약 컴퓨터 언어를 알고 있다면, 반복적인 작업을 사람을 대신해서 컴퓨터에 지시할 수 있다. 흥미롭게도, 컴퓨터가 가장 잘 할 수 있는 종류의 작업들은 종종 사람들이 재미없고, 너무나 지루하다고 생각하는 것이다.

예를 들어, 이번 장의 첫 세 문단을 보고, 가장 많이 나오는 단어를 찾아보고 얼마나 자주 나오는지를 알려주세요. 사람이 몇초내에 단어를 읽고 이해할 수는 있지만, 그 단어가 몇번 나오는지 세는 것은 매우 고생스러운 작업이다. 왜냐하면 사람이 지루하고 반복되는 문제를 해결하는데 적합하지 않기 때문이다. 컴퓨터는 정반대이다. 논문이나 책에서 텍스트를 읽고 이해하는 것은 컴퓨터에게 어렵다. 하지만, 단어를 세고 가장 많이 사용되는 단어를 찾는 것은 컴퓨터에게는 무척이나 쉬운 작업이다.

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

우리의 개인 정보분석 도우미는 이번장의 첫 세 문단에서 단어 "to" 가 가장 많이 사용되었고, 16번 나왔다고 바로 답을 준다.

사람이 잘하지 못하는 점을 컴퓨터가 잘할 수 있다는 사실을 이해하면 왜 "컴퓨터 언어"로 컴퓨터와 대화해야 하는데 능숙해야하는지 알 수 있다. 컴퓨터와 대화할 수 있는 새로운 언어(R 혹은 파이썬)를 배우게 되면, 지루하고 반복되는 일을 컴퓨터가 처리하고, 사람에게 적합한 일을 하는데 더 많은 시간을 할애할 수 있다. 그래서, 여러분은 직관, 창의성, 창의력을 컴퓨터 파트너와 함께 추진할 수 있다.

## 2.1 창의성과 동기

이책은 직업으로 프로그래밍을 하는 사람을 위해서 저작된 것은 아니지만, 직업적으로 프로그램을 만드는 작업은 개인적으로나 경제적인면에서 꽤 매력적인 일이다. 특히, 유용하며, 심미적이고, 똑똑한 프로그램을 다른 사람이 사용할 수 있도록 만드는 것은 매우 창의적인 활동이다. 다양한 그룹의 프로그래머들이 사용자의 관심과 시선을 차지하기 위해서 경쟁적으로 작성한 다양한 종류의 프로그램이 여러분의 컴퓨터와 개인 휴대 정보 단말기(Personal Digital Assistant, PDA)에 담겨있다. 이렇게 개발된 프로그램은 사용자가 원하는 바를 충족시키고 훌륭한 사용자 경험을 제공하려고 노력한다. 몇몇 상황에서 사용자가 소프트웨어를 골라 구매하게 될 때, 고객의 선택에 대해 프로그래머는 바로 경제적 보상을 받게 된다.

만약 프로그램을 프로그래머 집단의 창의적인 결과물로 생각해본다면, 아마도 다음 그림이 좀더 의미 있는 PDA 컴퓨터 혹은 스마트폰으로 보일 것이다.

우선은 프로그래을 만드는 주된 동기가 사업을 한다던가 사용자를 기쁘게 한다기보다, 일상생활에서 맞닥뜨리는 자료와 정보를 잘 다뤄 좀더 생산적으로


Figure 2.2: 컴퓨터 소개

우리의 삶을 만드는데 초점을 잡아본다. 프로그램을 만들기 시작할 때, 여러분 모두는 프로그래머이면서 동시에 자신이 만든 프로그램의 사용자가 된다. 프로그래머로서 기술을 습득하고 프로그래밍 자체가 좀더 창의적으로 느껴진다면, 여러분은 다른 사람을 위해 프로그램을 개발하게 준비가 된 것이다.

## 2.2 컴퓨터 하드웨어 아키텍처

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

주요 부품의 상위 수준 정의는 다음과 같다.

- **중앙처리장치(Central Processing Unit, CPU)**: 다음 무엇을 할까요? ("What is next?") 명령어를 처리하는 컴퓨터의 주요 부분이다. 만약 컴퓨터 중앙처리장치가 3.0 GHz 라면, 초당 명령어 (다음 무엇을 할까요? What is next?)를 삼백만번 처리할 수 있다는 것이다. CPU 처리속도를 따라서 빠르게 컴퓨터와 어떻게 대화하는지 학습할 것이다.
- **주기억장치(Main Memory)**: 주기억장치는 중앙처리장치(CPU)가 급하게 명령어를 처리하기 하는데 필요한 정보를 저장하는 용도로 사용된다. 주기억장치는 중앙처리장치만큼이나 빠르다. 그러나 주기억장치에 저장된 정보는 컴퓨터가 꺼지면 자동으로 지워진다.
- **보조 기억장치(Secondary Memory)**: 정보를 저장하기 위해 사용되지만, 주기억장치보다 속도는 느리다. 전기가 나갔을 때도 정보를 기억하는 것은 장점이다. 휴대용 USB나 휴대용 MP3 플레이어에 사용되는 USB 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.
- **입출력장치(Input Output Devices)**: 간단하게 화면, 키보드, 마우스, 마이크, 스피커, 터치패드가 포함된다. 컴퓨터와 사람이 상호작용하는 모든 방식이

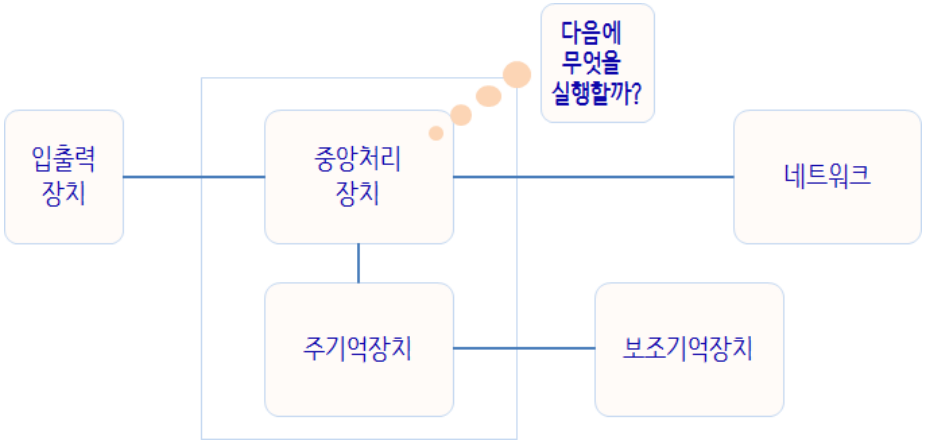



Figure 2.3: 컴퓨터 아키텍처

포함된다.

- **네트워크(Network)**: 요즘 거의 모든 컴퓨터는 네트워크로 정보를 주고 받는 네트워크 연결(Network Connection) 하드웨어가 있다. 네트워크는 항상 "이용가능" 하지 않을지도 모르는 데이터를 저장하고 가져오는 매우 느린 저장소로 볼 수 있다. 그러한 점에서 네트워크는 좀더 느리고, 때때로 신뢰성이 떨어지는 보조 기억장치(Secondary Memory)의 한 형태로 볼 수 있다.

주요 부품들이 어떻게 작동하는지에 대한 세세한 사항은 컴퓨터 제조자에게 맡겨져 있지만, 프로그램을 작성할 때 컴퓨터 주요 부품에 대해서 언급되어서, 컴퓨터 전문용어를 습득하고 이해하는 것은 도움이 된다.

프로그래머로서 임무는 자료를 분석하고 문제를 해결하도록, 컴퓨터 자원 각각을 사용하고 조율하는 것이다. 프로그래머로서 대체로 CPU와 "대화"해서 다음 무엇을 실행하라고 지시한다. 때때로 CPU에 주기억장치, 보조기억장치, 네트워크, 혹은 입출력장치도 사용하라고 지시한다.

프로그래머는 컴퓨터의 "다음 무엇을 수행할까요?"에 대한 답을 하는 사람이기도 하다. 하지만, 컴퓨터에 답하기 위해서 5mm 크기로 프로그래머를 컴퓨터에 집어넣고 초당 30억개 명령어로 답을 하게 만드는 것은 매우 불편하다. 그래서, 대신에 미리 컴퓨터에게 수행할 명령문을 작성해야 한다. 이렇게 미리 작성된 명령문 집합을 프로그램(Program)이라고 하며, 명령어 집합을 작성하고 명령어 집합이 올바르게 작성될 수 있도록 하는 행위를 프로그래밍(Programming)이라고 부른다.

## 2.3 프로그래밍 이해하기

책의 나머지 장을 통해서 책을 읽고 있는 여러분을 프로그래밍 장인으로 인도할 것이다. 종국에는 책을 읽고 있는 여러분 모두 **프로그래머**가 될 것이다. 아마도

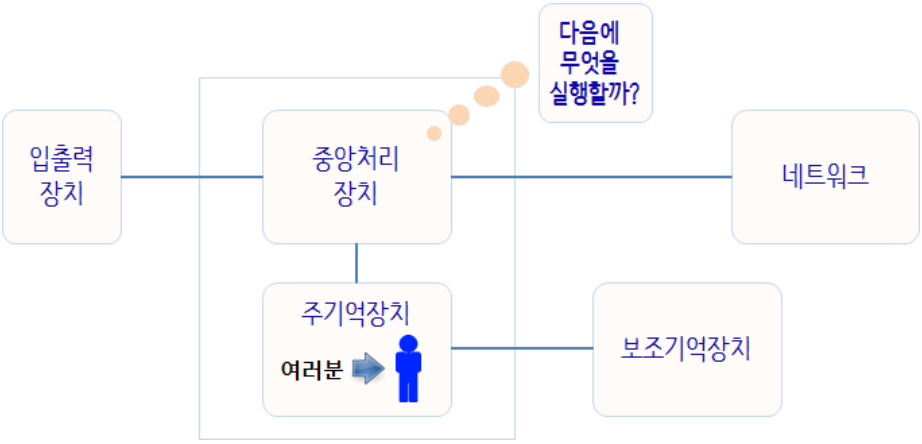



Figure 2.4: 컴퓨터 아키텍처 여러분

전문적인 프로그래머는 아닐지라도 적어도 자료/정보 분석 문제를 보고 그 문제를 해결할 수 있는 기술을 가지게는 될 것이다.

이런 점에서 프로그래머가 되기 위해서 두 가지 기술이 필요하다.

- 첫째, R 혹은 파이썬(Pythong)같은 프로그래밍 언어 - 어휘와 문법을 알 필요가 있다. 단어를 새로운 언어에 맞추어 작성할 수 있어야 하며 새로운 언어로 잘 표현된 "문장"으로 어떻게 작성하는지도 알아야 한다.
- 둘째, 스토리(Story)를 말 할 수 있어야 한다. 스토리를 작성할 때, 독자에게 아이디어(idea)를 전달하기 위해서 단어와 문장을 조합합니다. 스토리를 구성할 때 기술적인 면과 예술적인 면이 있는데, 기술적인 면은 쓰기 연습을 반복하고, 피트백을 받아 향상된다. 프로그래밍에서, 우리가 작성하는 프로그램은 "스토리"가 되고, 해결하려고 하는 문제는 "아이디어"에 해당된다.

R 혹은 파이썬과 같은 프로그래밍 언어를 배우게 되면, 자바스크립트나 C++, 고(Go) 같은 두번째 언어를 배우는 것은 무척이나 쉽다. 새로운 프로그래밍 언어는 매우 다른 어휘와 문법을 갖지만, 문제를 해결하는 기술을 배우면, 다른 모든 프로그래밍 언어를 통해서 동일하게 접근할 수 있습니다.

파이썬 어휘와 문장은 매우 빠르게 학습할 수 있다. 새로운 종류의 문제를 풀기 위해 논리적인 프로그램을 작성하는 것은 더 오래 걸린다. 여러분은 작문을 배우듯이 프로그래밍을 배우게 된다. 프로그래밍을 읽고 설명하는 것으로 시작해서, 간단한 프로그램을 작성하고, 점차적으로 복잡한 프로그램을 작성할 것이다. 어느 순간에 명상에 잠기게 되고, 스스로 패턴이 눈에 들어오게 된다. 그러면, 좀더 자연스럽게 문제를 어떻게 받아들이고, 그 문제를 해결할 수 있는 프로그램을 작성하게 된다. 마지막으로, 그 순간에 도착하게 되면, 프로그래밍은 매우 즐겁고 창의적인 과정이 된다.

파이썬 프로그램의 어휘와 구조로 시작한다. 간단한 예제가 처음으로 언제 프로그램을 읽기 시작했는지를 상기시켜주니 인내심을 가지세요.

## 2.4 단어와 문장

사람 언어와 달리, R 어휘는 사실 매우 적다. R 어휘를 **예약어(researved words)**로 부른다. 이들 단어는 R에 매우 특별한 의미를 부여한다. R 프로그램 관점에서 R이 이들 단어를 보게 되면, R에게는 단 하나의 유일한 의미를 갖는다. 나중에 여러분들이 프로그램을 작성할 때, 자신만의 단어를 작성하는데 이를 **변수(Variable)**라고 한다. 변수 이름을 지을 때 폭넓은 자유를 갖지만, 변수 이름으로 파이썬 예약어를 사용할 수는 없다.

이런 점에서 강아지를 훈련시킬 때 "걸어(walk)", "앉아", "기달려", "가져와" 같은 특별한 어휘를 사용한다. 강아지에게 이와 같은 특별한 예약어를 사용하지 않을 때는, 주인이 특별한 어휘를 사용할 때까지 강아지는 주인을 물끄러미 쳐다보기만 한다. 예를 들어, "더 많은 사람들의 건강을 전반적으로 향상하는 방향으로 동참하여"걷기(walk)"를 원한다"고 말하면, 강아지가 듣는 것은 "뭐라 뭐라 뭐라 **걷기(walk)** 뭐라"와 같이 들릴 것이다. 왜냐하면 "걸어(walk)"가 강아지 언어에는 예약어[1]이기 때문이다. 이러한 사실이 아마도 개와 고양이사이에는 어떠한 예약어도 존재하지 않는다는 것을 의미할지 모른다.

사람이 R과 대화하는 언어 예약어는 다음과 같다. R 콘솔에서 `? researved` 명령어를 입력하면 자세한 내용을 파악할 수 있다. 파이썬과 비교해도 상대적으로 데이터 분석에 집중된 것을 예약어만을 통해서도 쉽게 파악된다.

| R 예약어 | 설명 |
|---|---|
| If, else, repeat, while, function, for, in, next, break | 조건, 함수, 반복문에 사용 |
| TRUE, FALSE | 논리 상수(Logical constants) |
| NULL | 정의되지 않는 값 혹은 값이 없음을 표현 |
| Inf | 무한(Infinity) |
| NaN | 숫자가 아님(Not a Number) |
| NA | 결측값, 값이 없음 (Not Available) |
| NA_integer_, NA_real_, NA_complex_, NA_character_ | 결측값 처리하는 상수 |
| … | 함수가 다른 함수에 인자를 전달하도록 지원 |

강아지 사례와 사뭇 다르게 R은 이미 완벽하게 훈련이 되어 있다. 여러분이 "try" 라고 말하면, 매번 "try" 라고 말할 때마다 실패 없이 R은 항시 정확히 시도한다.

상기 예약어를 학습하고, 어떻게 잘 사용되는지도 함께 학습할 것이지만, 지금은 파이썬에 말하는 것에 집중할 것이다. R과 대화하는 것 중 좋은 점은 다음과 같이 괄호내부에 인용부호로 감싸 메세지를 던지는 것만으로도 R에 말을 할 수 있다는 것이다.

[1] Cat Proximity

```
print("   !")
```

```
## [1] "   !"
```

상기 간단한 문장은 R 구문(Syntax)론적으로도 완벽하다. 상기 문장은 예약어 'print'로 시작해서 출력하고자 하는 문자열을 괄호내부에 작은 따옴표로 감싸서 올바르게 R에게 전달했다.

## 2.5 R과 대화하기

R로 우리가 알고 있는 단어를 가지고 간단한 문장을 만들었으니 이제부터는 새로운 언어 기술을 시험하기 위해서 파이썬과 대화를 어떻게 시작하는지 알 필요가 있다.

R과 대화를 시작하기 전에, R 소프트웨어를 컴퓨터에 설치하고 R을 컴퓨터에서 어떻게 실행하는지를 학습해야 한다. 이번 장에서 다루기에는 너무 구체적이고 자세한 사항이기 때문에 http://statkclee.github.io/data-science/을 참조하는 것을 권한다. 윈도우와 리눅스, 매킨토쉬 시스템 상에서 설치하고 실행하는 방법을 자세한 설치절차와 함께 화면을 캡쳐하여 다양한 환경에서 설명해 놨다. 설치가 마무리되고 터미널이나 윈도우 명령어 실행창에서 R을 타이핑 하게 되면, R 인터프리터가 인터랙티브 모드로 실행을 시작하고 다음과 같은 것이 화면에 뿌려진다.

```
R version 4.2.0 (2022-04-22 ucrt) -- "Vigorous Calisthenics"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

R 인터프리터는 > 프롬프트를 통해서 여러분에게 요청사항("다음에 R이 무엇을 실행하기를 원합니까?")을 접수받는 방식을 취한다. 이제 R은 여러분과 대화를 나눌 준비가 되었다. 이제 남은 것은 R 언어로 어떻게 말하고 어떻게 R과 대화하는지 아는 것이다.

예를 들어, 여러분이 가장 간단한 R 언어 단어나 문장 조차도 알 수가 없다고 가정하자. 우주 비행사가 저 멀리 떨어진 행성에 착륙해서 행성의 거주민과 대화를 시도할 때

사용하는 간단한 말을 사용해 보자.

```
> I come in peace, please take me to your leader
Error: unexpected symbol in "I come"
>
```

잘 되는 것 같지 않다. 뭔가 빨리 다른 생각을 내지 않는다면, 행성 거주민은 여러분을 창으로 찌르고, 침으로 바르고, 불위 잘 구워 바베큐로 만들어 저녁으로 먹을 듯 하다.

운 좋게도 기나긴 우주 여행 중 이 책의 복사본을 가지고 와서 다음과 같이 빠르게 타이핑한다고 생각하자.

```
print("   !")
```

```
## [1] "   !"
```

훨씬 좋아보인다. 이제 좀더 커뮤니케이션을 이어갈 수 있을 것으로 보인다.

```
print('You must be the legendary god that comes from the sky')
print('We have been waiting for you for a long time')
print('Our legend says you will be very tasty with mustard')
print('We will have a feast tonight unless you say) #'
```

이번 대화는 잠시 동안 잘 진행되다가 여러분이 R 언어로 말하는데 정말 사소한 실수를 저질러 R이 다시 창을 여러분에게 겨눈다.

이 시점에 R은 놀랍도록 복잡하고 강력하며 R과 의사소통을 할 때 사용하는 구문(syntax)은 매우 까다롭다는 것은 알 수 있다. R은 다른 말로 안 똑똑(Intelligent)하다. 지금까지 여러분은 자신과 대화를 적절한 구문(syntax)을 사용해서 대화했다.

여러분이 다른 사람이 작성한 프로그램을 사용한다는 것은 R을 사용하는 다른 프로그래머가 R을 중간 매개체로 사용하여 대화한 것으로 볼 수 있다. 프로그램을 만든 저작자가 대화를 어떻게 진행되어져야 하는지를 표현하는 방식이다. 다음 몇 장에 걸쳐서 다른 많은 프로그래머 중의 한명처럼, R로 여러분이 작성한 프로그램을 이용하는 사용자와 대화하게 된다.

R 인터프리터와 첫번째 대화를 끝내기 전에, R 행성의 거주자에게 "안녕히 계세요"를 말하는 적절한 방법도 알아야 한다.

```
good-bye
if you don not mind, I need to leave
quit("yes")
```

상기 처음 두개 시도는 다른 오류 메세지를 출력한다. 두번째 오류는 다른데 이유는 **if**가 예약어이기 때문에 R은 이 예약어를 보고 뭔가 다른 것을 말한다고 생각하지만, 잠시 후 구문이 잘못됐다고 판정하고 오류를 뱉어낸다.

R에 "안녕히 계세요"를 말하는 올바른 방식은 인터렉티브 > 프롬프트에서 `quit()` 혹은 `q()`를 입력하는 것이다.

## 2.6 전문용어: 인터프리터와 컴파일러

R은 상대적으로 직접 사람이 읽고 쓸 수도 있고, 컴퓨터도 읽고 처리할 수 있도록 고안된 **하이 레벨(High-level), 고수준** 언어이다. 다른 하이 레벨 언어에는 자바, C++, PHP, 루비, 베이직, 펄, 자바스크립트 등 다수가 포함되어 있다. 실제 하드웨어 중앙처리장치(CPU)내에서는 하이레벨 언어를 조금도 이해하지 못한다.

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

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

표면적으로 0과 1로만 되어 있기 때문에 기계어가 간단해 보이지만, 구문은 매우 복잡하고 R보다 훨씬 어렵다. 그래서 매우 소수의 프로그래머만이 기계어로 작성할 수 있다. 대신에, 프로그래머가 파이썬과 자바스크립트 같은 하이 레벨 언어로 작성할 수 있게 다양한 번역기(translator)를 만들었다.
이러한 번역기는 프로그램을 중앙처리장치에 의해서 실제 실행이 가능한 기계어로 변환한다.

기계어는 특정 컴퓨터 하드웨어에 묶여있기 때문에 기계어는 다른 형식의 하드웨어에는 **이식(portable)**되지 않는다. 하이 레벨 언어로 작성된 프로그램은 두 가지 방식으로 이기종의 컴퓨터로 이식이 가능하다. 한 방법은 새로운 하드웨어에 맞게 기계어를 재컴파일(recompile)하는 것이고, 다른 방법은 새로운 하드웨어에 맞는 다른 인터프리터를 이용하는 것이다.

프로그래밍 언어 번역기는 일반적으로 두가지 범주가 있다.

- 인터프리터(Interpreter)
- 컴파일러(Compiler)

**인터프리터**는 프로그래머가 코드를 작성할 때 소스 코드를 읽고, 소스코드를 파싱하고, 즉석에서 명령을 해석한다. R은 인터프리터다. 따라서, R을 인터렉트브 모드로 실행할 때, R 명령문(한 문장)을 작성하면, R이 즉석에서 처리하고, 사용자가 다른 R 명령어를 입력하도록 준비를 한다.

R 코드의 일부는 나중에 사용될 것이니 R에게 기억하도록 명령한다. 적당한 이름을 골라서 값을 기억시키고, 나중에 그 이름을 호출하여 값을 사용한다. 이러한 목적으로 저장된 값을 참조하는 목적으로 사용되는 표식(label)을 **변수(variable)**라고 한다.

```
x <- 6
x
```

```
## [1] 6
```

```
y <-  x * 7
y
```

```
## [1] 42
```

상기 예제에서 R이 값 6 을 기억하고 있다가, 라벨 `x`를 사용하여 나중에 값을 가져오게 만들었다. 파이썬의 경우 `print` 예약어를 사용하여 파이썬이 잘 기억하고 있는지를 검증한다. 하지만, R이 파이썬과 비교하여 차이가 나는 점은 인터랙티브 방식으로 데이터 분석에 장점을 갖고 있어 바로 변수명 혹은 다른 자료구조명(요인, 데이터프레임 등)을 입력하면 변수명에 들어 있는 내용을 즉각 파악가능하다.

`x`를 가져와서 7을 곱하고 새로운 변수 `y`에 값을 집어 넣는다. 그리고 `y`에 현재 무슨 값이 저장되었는지 출력하라고 R에게 지시한다.

한줄 한줄 R에 명령어를 입력하고 있지만, 앞쪽 명령문에서 생성된 자료가 뒤쪽 실행 명령문에서 사용될 수 있도록 R은 순차적으로 정렬된 문장으로 처리한다. 방금전 논리적이고 의미있는 순서로 4줄 명령문을 간단하게 한 단락으로 작성했다.

위에서 본 것처럼 R과 인터랙티브하게 대화를 주고받는 것이 인터프리터의 본질이다. 컴파일러가 동작하기 위해서는 먼저 완전한 프로그램을 파일 하나에 담고, 하이 레벨 소스코드를 기계어로 번역하는 과정을 거치고, 마지막으로 나중에 실행되도록 변환된 기계어를 파일에 담는다.

윈도우를 사용한다면, 실행가능한 기계어 프로그램 확장자가 ".exe"(executable), 혹은 ".dll"(dynamically loadable library)임을 확인할 수 있다. 리눅스와 매킨토쉬에는 실행파일을 의미하는 특정 확장자는 없다.

텍스트 편집기에서 실행파일을 열게 되면, 다음과 같이 읽을 수 없는 좀 괴상한 출력결과를 화면상에서 확인한다.

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

기계어를 읽고 쓰는 것은 쉽지 않다. 그래서 C 나 R, 파이썬 같은 하이 레벨 언어로 작성된 프로그램을 기계어로 자동 번역해주는 **인터프리터**와 **컴파일러**가 있다는 것은 멋진 일이다.

컴파일러와, 인터프리터를 논의하는 이 시점에, R 인터프리터 자체에 대해서 약간 궁금해야 한다. 무슨 언어로 작성되었을까? 컴파일된 언어로 작성되었을까? "R"을 타이핑하게 될 때, 정확하게 무슨 일이 일어나는걸까?

R 인터프리터는 하이 레벨 언어 "C"로 작성되었다. R 인터프리터 실제 소스 코드를 보려면, https://www.r-project.org/ 웹사이트에 가서 여러분의 방식으로 개발할 수 있는 소스코드를 확인할 수 있다. 그래서, R 그 자체도 프로그램이다. 기계어로 컴파일되어 있어서 R을 여러분의 컴퓨터에 설치(혹은 컴퓨터 제조자가 설치를 대신 해주기도 함)한다는 것은 번역된 R 프로그램 기계어 코드 사본을 여러분 컴퓨터에 복사하는 것에 불과하다. 윈도우 시스템에서 R 실행가능한 기계어 코드는 파일에 다음과 같은 이름을 갖는다.

- `C:\Program Files\R\R-4.2.0\bin\R.exe`

지금까지 살펴본 것은 R 프로그래머가 되기 위해서 정말 알 필요가 있는 것 이상이다. 하지만, 때때로 처음에 이런 귀찮은 질문에 바로 답하는 것이 나중에 보상을 충분히 하고도 남는다.

## 2.7 프로그램 작성하기

R 인터프리터에 명령어를 타이핑 하는 것은 R 주요 기능을 알아보는 좋은 방법이지만, 좀더 복잡한 문제를 해결하는데 권하지는 않는다.

프로그램을 작성할 때, 텍스트 편집기를 사용해서 **스크립트(script)**로 불리는 파일에 명령어 집합을 작성한다. 관례로, R 스크립트 확장자는 `.R`가 된다.

스크립트를 실행하기 위해서, R 인터프리터에 파일 이름을 넘겨준다. 유니스나 윈도우 명령창에서 `R hello.R` 를 입력하게 되면 다음과 같은 결과를 얻는다.

```
$ cat hello.R
print("   !")
$ R hello.R
"   !"
$
```

"$"은 운영시스템 명령어 프롬프트이고, "R hello.R"는 문자열을 출력하는 한줄 R 프로그램을 담고 있는 "hello.R" 파일을 화면에 출력하라는 명령어입니다.

인터랙트브 모드에서 R 코드 입력하는 방식 대신에 R 인터프리터를 호출해서 "hello.R" 파일로부터 소스코드를 읽도록 지시합니다.

이 새로운 방식은 R 프로그램을 끝마치기 위해 `quit()`를 사용할 필요가 없다는 점에서 편리합니다. 파일에서 소스코드를 읽을 때, 파일 끝까지 읽게 되면 자동으로 R이 종료됩니다.

## 2.8 프로그램이란 무엇인가?

**프로그램(Program)**의 가장 본질적인 정의는 특정 작업을 수행할 수 있도록 조작된 일련의 R 문장의 집합이다. 가장 간단한 `hello.R` 스크립트도 프로그램이다. 한줄의 프로그램이 특별히 유익하고 쓸모가 있는 것은 아니지만 엄격한 의미에서 R 프로그램이 맞다.

프로그램을 이해하는 가장 쉬운 방법은 프로그램이 해결하려고 만들어진 문제를 먼저 생각해보고 나서, 그 문제를 풀어가는 프로그램을 살펴보는 것이다.

예를 들어, 페이스북에 게시된 일련의 글에서 가장 자주 사용된 단어에 관심을 가지고 소셜 컴퓨팅 연구를 한다고 생각해 봅시다. 페이스북에 게시된 글들을 쭉 출력해서 가장 흔한 단어를 찾으려고 열심히 들여다 볼 것이지만, 매우 오래 걸리고 실수하기도 쉽다. 하지만 R 프로그램을 작성해서 빨리 정확하게 작업을 마무리한다면 똑똑하게 주말을 재미나게 보낼 수 있다.

예를 들어 자동차(car)와 광대(clown)에 관한 다음 텍스트에서, 가장 많이 나오는 단어가 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 나서, 몇 백만줄의 텍스트를 보고서 동일한 일을 한다고 상상해 보자. 솔직히 수작업으로 단어를 세는 것보다 R을 배워 프로그램을 작성하는 것이 훨씬 빠를 것이다.

더 좋은 소식은 이미 텍스트 파일에서 가장 자주 나오는 단어를 찾아내는 간단한 프로그램을 개발했다. 저자가 직접 작성했고, 시험까지 했다. 바로 사용을 할 수 있도록 준비했기 때문에 여러분의 수고도 덜 수 있다.

```
# 0.    ------
library(tidyverse)
library(stringr)

text_dat <- readLines("data/ch01-text.txt")
split_word <- str_split(text_dat, " ") %>% .[[1]]

uniq_word <- unique(split_word)

res_v <- vector("integer", length(uniq_word))

for(i in seq_along(uniq_word)) {
  for(j in seq_along(split_word)) {
    if(uniq_word[i] == split_word[j]) {
      res_v[i] <- res_v[i] + 1
    }
  }
}

bind_cols("word" = uniq_word, "freq"=res_v) %>% arrange(desc(freq))
```

```
## # A tibble: 11 x 2
##   word   freq
##   <chr> <dbl>
## 1 the       7
## 2 car       3
## 3 and       3
## 4 clown     2
## 5 ran       2
## 6 tent      2
## # ... with 5 more rows
```

```
#   -----
```

```
text_v <- "the clown ran after the car and the car ran into the tent and the tent fell down on th
split_word <- str_split(text_v, " ") %>% unlist
table(split_word) %>% sort(decreasing=TRUE)
```

```
## split_word
##   the   and   car clown   ran  tent after  down  fell  into    on
##     7     3     3     2     2     2     1     1     1     1     1
```

상기 프로그램을 사용하려고 R을 공부할 필요도 없다. 10장에 걸쳐서 멋진 R 프로그램을 만드는 방법을 배우게 될 것이다. 지금 여러분은 단순 사용자로서 단순히 상기 프로그램을 사용하게 되면, 프로그램의 영리함과 동시에 얼마나 많은 수작업 노력을 줄일 수 있는지 감탄할 것이다. 단순하게 코드를 타이핑해서 `words.R` 파일로 저장하고 실행을 하거나, https://github.com/statkclee/r4inf/에서 소스코드를 다운받아 실행하면 된다.

R 언어가 어떻게 여러분(사용자)과 저자(프로그래머)사이에서 중개자 역할을 훌륭히 수행하고 있는지를 보여주는 좋은 사례다. 컴퓨터에 R을 설치한 누구나 사용할 수 있는 공통의 언어로 유용한 명령 순서(즉, 프로그램)를 우리가 주고받을 수 있는 방식이 R 방식이다. 그래서 누구도 R과 직접 의사소통하지 않고 R을 통해서 서로 의사소통한다.

## 2.9 프로그램 구성요소

다음 몇장에 걸쳐서 R 어휘, 문장구조, 문단구조, 스토리 구조에 대해서 학습할 것이다. R의 강력한 역량에 대해서 배울 것이고, 유용한 프로그램을 작성하기 위해서 R의 역량을 어떻게 조합할지도 학습할 것이다.

프로그램을 작성하기 위해서 사용하는 개념적인 하위 레벨(low-level) 패턴이 몇가지 있다. R 프로그램을 위해서 만들어졌다기 보다는 기계어부터 하이 레벨(high-level) 언어에 이르기까지 모든 언어에도 공통된 사항이기도 하다.

- **입력**: 컴퓨터 바깥 세계에서 데이터를 가져온다. 파일로부터 데이터를 읽을 수도 있고, 마이크나 GPS 같은 센서에서 데이터를 입력받을 수도 있다. 상기 초기 프로그램에서 입력값은 키보드를 사용하여 사용자가 데이터를 입력한 것이다.
- **출력**: 화면에 프로그램 결과값을 출력주거나 파일에 저장한다. 혹은 음악을 연주하거나 텍스트를 읽어 스피커 같은 장치에 데이터를 내보낸다.
- **순차 실행**: 스크립트에 작성된 순서에 맞춰 한줄 한줄 실행된다.
- **조건 실행**: 조건을 확인하고 명령문을 실행하거나 건너뛴다.
- **반복 실행**: 반복적으로 명령문을 실행한다. 대체로 반복 실행시 변화를 수반한다.
- **재사용**: 한벌의 명령문을 작성하여 이름을 부여하고 저장한다. 필요에 따라 프로그램 이름을 불러 몇번이고 재사용한다.

너무나 간단하게 들리지만, 전혀 간단하지는 않다. 단순히 걸음을 "한 다리를 다른 다리 앞에 놓으세요" 라고 말하는 것 같다. 프로그램을 작성하는 "예술"은 기본 요소를 조합하고 엮어 사용자에게 유용한 무언가를 만드는 것이다.

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

## 2.10 프로그램이 잘못되면?

처음 R과 대화에서 살펴봤듯이, R 코드를 명확하게 작성해서 의사소통 해야 한다. 작은 차이 혹은 실수는 여러분이 작성한 프로그램을 R이 들여다보다 조기에 포기하게 만든다.

초보 R 프로그래머는 R이 오류에 대해서는 인정사정 보지 않는다고 생각한다. R이 모든 사람을 좋아하는 것 같지만, R은 개인적으로만 사람들을 알고, 분노를 간직하고 있다. 이러한 사실로 인해서 R은 여러분이 완벽하게 작성된 프로그램을 받아서 "잘 맞지 않는군요"라고 거절하여 고통을 준다.

```
primt("   !")
primt("   !")
  R   !
    ,     !!!
```

R과 다퉈봐야 얻을 것은 없어요. R은 도구고 감정이 없다. 여러분이 필요로 할 때마다 여러분에게 봉사하고 기쁨을 주기 위해서 존재할 뿐이다. 오류 메세지가 심하게 들릴지는 모르지만 단지 R이 도와달라는 요청일 뿐이다. 입력한 것을 쭉 읽어 보고 여러분이 입력한 것을 이해할 수 없다고만 말할 뿐이다.

R은 어떤 면에서 강아지와 닮았다. 맹목적으로 여러분을 사랑하고, 강아지와 마찬가지로 몇몇 단어만 이해하며, 웃는 표정(> 명령 프롬프트)으로 여러분이 R이 이해하는 무언가를 말하기만을 기다린다. R이 "Error: object '나는' not found"을 뱉어낼 때는, 마치 강아지가 꼬리를 흔들면서 "뭔가 말씀하시는 것 같은데요… 주인님 말씀을 이해하지 못하겠어요, 다시 말씀해 주세요 (>)" 말하는 것과 같다.

여러분이 작성한 프로그램이 점점 유용해지고 복잡해짐에 따라 3가지 유형의 오류와 마주친다.

- **구문 오류(Syntax Error)**: 첫번째 마주치는 오류로 고치기 가장 쉽습니다. 구문 오류는 R 문법에 맞지 않는다는 것을 의미한다. R은 구문오류가 발생한 줄을 찾아 정확한 위치를 알려준다. 하지만, R이 제시하는 오류가 그 이전 프로그램 부문에서 발생했을 수도 있기 때문에 R이 제시하는 곳 뿐만 아니라 그 앞쪽도 살펴볼 필요가 있다. 따라서 구문 오류로 R이 지칭하는 행과 문자는 오류를 고치기 위한 시작점으로 의미가 있다.
- **논리 오류(Logic Error)**: 논리 오류의 경우 프로그램 구문은 완벽하지만 명령어 실행 순서에 실수가 있거나 혹은 문장이 서로 연관되는 방식에 오류가 있는 것이다. 논리 오류의 예를 들어보자. "물병에서 한모금 마시고, 가방에 넣고, 도서관으로 걸어가서, 물병을 닫는다"
- **의미론적 오류(Semantic Error)**: 의미론적 오류는 구문론적으로 완벽하고 올바른 순서로 프로그램의 명령문이 작성되었지만 단순하게 프로그램에 오류가 있다. 프로그램은 완벽하게 작동하지만 여러분이 의도한 바를 수행하지는 못한다. 간단한 예로 여러분이 식당으로 가는 방향을 알려주고

> 있다. ” … 주유소 사거리에 도착했을 때, 왼쪽으로 돌아 1.6km 쭉 가면 왼쪽편에 빨간색 빌딩에 식당이 있습니다.” 친구가 매우 늦어 전화로 지금 농장에 있고 헛간으로 걸어가고 있는데 식당을 발견할 수 없다고 전화를 합니다. 그러면 여러분은 “주유소에서 왼쪽으로 혹은 오른쪽으로 돈거야?” 말하면, 그 친구는 “말한대로 완벽하게 따라서 갔고, 말한대로 필기까지 했는데, 왼쪽으로 돌아 1.6km 지점에 주요소가 있다고 했어”, 그러면 여러분은 “미안해, 내가 가지고 있는 건 구문론적으로는 완벽한데, 슬프게도 사소하지만 탐지되지 않은 의미론적 오류가 있네!” 라고 말할 것이다.

다시 한번 위 세 종류의 오류에 대해서, R은 단지 여러분이 요청한 것을 충실히 수행하기 위해서 최선을 다합니다.

## 2.11 학습으로의 여정

책을 읽어 가면서 처음에 개념들이 잘 와 닿지 않는다고 기죽을 필요는 없다. 말하는 것을 배울 때, 처음 몇년 동안 웅얼거리는 것은 문제도 아니다. 간단한 어휘에서 간단한 문장으로 옮겨가데 6개월이 걸리고, 문장에서 문단으로 옮겨가는데 5-6년 이상 걸려도 괜찮다. 흥미로운 완전한 짧은 스토리를 자신의 언어로 작성하는데 몇 년이 걸린다.

R을 빨리 배울 수 있도록 다음 몇장에 걸쳐서 모든 정보를 제공한다. 하지만 새로운 언어를 습득하는 것과 마찬가지로 자연스럽게 느껴지기까지 R을 흡수하고 이해하기까지 시간이 걸린다. 큰 그림(Big Picture)을 이루는 작은 조각들을 정의하는 동안에, 큰 그림을 볼 수 있도록 여러 주제를 방문하고, 또 다시 재방문하면서 혼란이 생길 수도 있다. 이 책은 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 배운 것을 다시 리뷰하고 연습문제를 다시 풀면서 지금 난공불락이라 여겼던 어려운 주제를 통해서 사실 더 많은 것을 학습했다는 것을 깨달을 것이다.

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

만약 어떤 것이 특별히 힘들다면, 밤새도록 앉아서 노력하는 것은 별로 의미가 없다. 잠시 쉬고, 낮잠을 자고, 간식을 먹고 다른 사람이나 강아지에게 문제를 설명하고 자문을 구한 후에 깨끗한 정신과 눈으로 돌아와서 다시 시도해보라. 단언컨데 이 책에 있는 프로그래밍 개념을 깨우치게 되면, 돌이켜 생각해보면 프로그래밍은 정말 쉽고 멋지다는 것을 알게 될 것이다. 그래서 단순하게 프로그래밍 언어는 정말 시간을 들여서 배울 가치가 있다.

## 2.12 용어사전

- **버그(bug)**: 프로그램 오류

- **중앙처리장치(central processing unit, CPU)**: 컴퓨터의 심장, 작성한 프로그램을 실행하는 장치, “CPU” 혹은 프로세서라고 부른다.
- **컴파일(compile)**: 나중에 실행을 위해서 하이레벨 언어로 작성된 프로그램을 로우레벨 언어로 번역한다.
- **하이레벨 언어(high-level language)**: 사람이 읽고 쓰기 쉽게 설계된 파이썬과 같은 프로그래밍 언어
- **인터랙티브 모드(interactive mode)**: 프롬프트에서 명령어나 표현식을 타이핑함으로써 파이썬 인터프리터를 사용하는 방식
- **해석한다(interpret)**: 하이레벨 언어로 작성된 프로그램을 한번에 한줄씩 번역해서 실행한다.
- **로우레벨 언어(low-level language)**: 컴퓨터가 실행하기 좋게 설계된 프로그래밍 언어, “기계어 코드”, “어셈블리 언어”로 불린다.
- **기계어 코드(machine code)**: 중앙처리장치에 의해서 바로 실행될 수 있는 가장 낮은 수준의 언어로 된 소프트웨어
- **주기억장치(main memory)**: 프로그램과 데이터를 저장한다. 전기가 나가게 되면 주기억장치에 저장된 정보는 사라진다.
- **파싱(parsing)**: 프로그램을 검사하고 구문론적 구조를 분석한다.
- **이식성(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)**: 컴퓨테이션(Computation)을 명세하는 명령어 집합
- **프롬프트(prompt)**: 프로그램이 메세지를 출력하고 사용자가 프로그램에 입력하도록 잠시 멈춘 때.
- **보조 기억장치(secondary memory)**: 전기가 나갔을 때도 정보를 기억하고 프로그램을 저장하는 저장소. 일반적으로 주기억장치보다 속도가 느리다. USB의 플래쉬 메모리나 디스크 드라이브가 여기에 속한다.
- **의미론(semantics)**: 프로그램의 의미
- **의미론적 오류(semantic error)**: 프로그래머가 의도한 것과 다른 행동을 하는 프로그램 오류
- **소스 코드(source code)**: 하이레벨 언어로 기술된 프로그램

## 2.13 연습문제

1. 컴퓨터 보조기억장치 기능은 무엇입니까?
    1. 프로그램의 모든 연산과 로직을 실행한다.
    2. 인터넷을 통해 웹페이지를 불러온다.
    3. 파워가 없을 때도 정보를 장시간 저장한다.
    4. 사용자로부터 입력정보를 받는다.
2. 프로그램은 무엇입니까?
3. 컴파일러와 인터프리터의 차이점을 설명하세요.
4. 기계어 코드는 다음중 어는 것입니까?
    1. 파이썬 인터프리터
    2. 키보드
    3. 파이썬 소스코드 파일

    4. 워드 프로세싱 문서
5. 다음 R 프로그램이 실행된 후에, 변수 “X”는 어디에 저장됩니까?
    1. 중앙처리장치
    2. 주메모리
    3. 보조메모리
    4. 입력장치
    5. 출력장치
6. 사람의 어느 능력부위를 예제로 사용하여 다음 각각을 설명하세요. (1) 중앙처리장치, (2) 주메모리, (3) 보조메모리, (4) 입력장치, (5) 출력장치, 예를 들어 중앙처리장치에 상응하는 사람의 몸 부위는 어디입니까?
7. 구문오류(“Syntax Error”)는 어떻게 고칩니까?
8. 다음 코드에서 잘못된 점을 설명하세요.

```
primt("   !")
Error in primt("   !") : could not find function "primt"
```

9. 다음 프로그램에서 출력되는 것은 무엇입니까?
    1. 43
    2. 44
    3. x + 1
    4. 오류, 왜냐하면 x <- x + 1 은 수학적으로 불가능하다.

```
x <- 43
x <- x + 1
x
```

# 프로그래밍

# Chapter 3

# 변수, 표현식, 문장

## 3.1 값과 자료형

**값(Value)**은 문자와 숫자처럼 프로그램이 다루는 가장 기본이 되는 단위이다. 지금까지 살펴본 값은 1, 2 그리고 '헬로 월드!'다.

상기 값은 다른 **자료형(Type)**에 속하는데, 2는 **정수(integer)**, '헬로 월드!'는 **문자열(String)**에 속하는데, 문자(character)를 일련의 열(sequence)의 형태로 되어 있어서 문자열이라고 부른다. 인용부호에 감싸여 있어서, 여러분과 인터프리터는 문자열을 식별할 수 있다.

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

```
print(7)
```

```
## [1] 7
```

값이 어떤 형인지 확신을 못한다면, 인터프리터가 알려준다.

```
typeof("    !")
```

```
## [1] "character"
```

```
typeof(17)
```

```
## [1] "double"
```

다소 놀랄수도 있겠지만, 17은 **부동소수점** 숫자 형식 더블(double)이고 `!`는 문자(character)가 된다.

```
typeof(7L)
```

```
## [1] "integer"
```

정수형을 필히 표현하려면 7L와 같이 정수 뒤에 L을 붙이면 된다. '17', '3.2' 같은 값은 어떨가? 숫자처럼 보이지만 문자열처럼 인용부호에 감싸여 있다.

```
typeof('17L')
```

```
## [1] "character"
```

```
typeof('3.2')
```

```
## [1] "character"
```

'17', '3.2' 은 문자열이다.

1,000,000 처럼 아주 큰 정수를 입력할 때, 사람이 인식하기 편한 형태로 세자리 숫자마다 콤마(,)를 사용하고 싶을 것이다. 하지만, R에서는 오류가 난다.

```
> 1,000,000
Error: unexpected ',' in "1,"
```

파이썬의 경우 파이썬에서는 정상적으로 실행되나, 실행 결과는 우리가 기대했던 것이 아니다. 파이썬에서는 1,000,000 을 콤마(',')로 구분된 정수로 인식한다. 따라서 사이 사이 공백을 넣어 출력했다. 이 사례가 여러분이 처음 경험하게 되는 **의미론적 오류(semantic error)**다. 코드가 에러 메세지 없이 실행이되지만, "올바른(right)" 작동을 하는 것은 아니다.

## 3.2 변수

프로그래밍 언어의 가장 강력한 기능 중의 하나는 변수를 다룰 수 있는 능력이다. **변수(Variable)**는 값을 참조하는 이름이다. **대입문(Assignment statement)**는 새로운 변수를 생성하고 값을 변수에 대입한다.

```
message <- "        R Meetup     ."
n <- 17L
pi <- 3.1415926535897931
```

상기 예제는 세가지 대입 사례를 보여준다. 첫 번째 대입 예제는 message 변수에 문자열을 대입한다. 두 번째 예제는 변수 n에 정수 17을 대입한다. 세 번째 예제는 pi변수에 근사값을 대입한다.

변수 값을 출력하기 위해서 print문을 사용하도 되지만, 일반적으로 변수명을 콘솔에서 타이핑하면 된다.

```
print(message)
```

```
## [1] "        R Meetup     ."
```

```
n
```

```
## [1] 17
```

```
pi
```

```
## [1] 3.14
```

변수 자료형(type)은 변수가 참조하는 값의 자료형이다.

```
typeof(message)
```

```
## [1] "character"
```

```
typeof(n)
```

```
## [1] "integer"
```

```
typeof(pi)
```

```
## [1] "double"
```

## 3.3 변수명과 예약어

대체로 프로그래머는 의미있는 **변수명(variable name)**을 고른다. 프로그래머는 변수가 사용되는 것에 대해 문서화도 한다.

변수명은 임의로 길 수 있다. 변수명은 문자와 숫자를 포함할 수 있지만, 문자로 변수명을 시작해야 한다. 첫 변수명을 대문자로 사용해도 되지만 소문자로 변수명을 시작하는 것도 좋은 생각이다. (후에 왜 그런지 보게 될 것이다.)

변수명에 밑줄(underscore character, _)이 들어갈 수 있다. 종종 `my_name` 혹은 `airspeed_of_unladen_swallow` 처럼 밑줄은 여러 단어와 함께 사용된다. 변수명을 밑줄로 시작해서 작성할 수 있지만, 다른 사용자가 사용할 라이브러리를 작성하는 경우가 아니라면, 일반적으로 밑줄로 시작하는 변수명은 피한다. 한글을 변수명으로 사용하는 것도 가능하지만, 인코딩 등 여타 예기치 못한 문제가 생길 수도 있다는 점을 유념하고 사용한다. R이 다른 언어와 다른 점은 <-을 변수명에 값을 대입하는데 사용하는 점이다. 이유는 R이 한창 개발될 당시 가장 최신 이론에 바탕을 두고 있기 때문이다. 수학적으로 `variable_name = 123L`와 같은 문장이 맞는지 곰곰히 생각해 보면 그 당시 <- 기호를 사용한 이유를 유추할 수 있다.

변수명을 적합하게 작성하지 못하다면, 구문 오류가 발생한다.

```
76trombones <- 'big parade'
more@ <- 1000000
repeat <- 'Advanced Theoretical Zymurgy'
```

`76trombones` 변수명은 문자로 시작하지 않아서 적합하지 않다. `more@`은 특수 문자(@)를 변수명에 포함해서 적합하지 않다. 하지만, `repeat` 변수명은 뭐가 잘못된 것일까?

구문 오류 이유는 `repeat`이 R의 예약어 중의 하나라고 밝혀졌다. 인터프리터가 예약어를 사용하여 프로그램 구조를 파악하기 위해서 사용하지만, 변수명으로는 사용할 수 없다.

| R 예약어 | 설명 |
|---|---|
| If, else, repeat, while, function, for, in, next, break | 조건, 함수, 반복문에 사용 |
| TRUE, FALSE | 논리 상수(Logical constants) |
| NULL | 정의되지 않는 값 혹은 값이 없음을 표현 |
| Inf | 무한(Infinity) |
| NaN | 숫자가 아님(Not a Number) |
| NA | 결측값, 값이 없음 (Not Available) |
| NA_integer_, NA_real_, NA_complex_, NA_character_ | 결측값 처리하는 상수 |
| … | 함수가 다른 함수에 인자를 전달하도록 지원 |

상기 예약어 목록을 주머니에 넣고 잘 가지고 다니고 싶을 것이다. 만약 인터프리터가 변수명 중 하나에 대해 불평을 하지만 이유를 모르는 경우, 예약어 목록에 변수명이 있는지 확인해 보세요.

## 3.4 문장

**문장(statement)**은 R 인터프리터가 실행하는 코드 단위다. 지금까지 `print`, (assignment, <-) 두 종류의 문장을 살펴봤습니다.

인터랙트브 모드에서 문장을 입력하면, 인터프리터는 문장을 실행하고, 만약 출력할 것이 있다면 결과를 화면에 출력합니다. 스크립트는 보통 여러줄의 문장으로 구성됩니다. 하나 이상의 문장이 있다면, 문장이 순차적으로 실행되며서 결과가 한번에 하나씩 나타납니다.

예를 들어, 다음의 스크립트를 생각해 봅시다.

```
1
x <- 2
x
```

상기 스크립트는 다음 결과를 출력합니다.

```
## [1] 1

## [1] 2
```

대입 문장(`x <- 2`)은 결과를 출력하지 않습니다.

## 3.5 연산자와 피연산자

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

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

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

## 3.6 표현식

**표현식 (expression)**은 값, 변수, 연산자 조합이다. 값은 자체로 표현식이고, 변수도 동일하다. 따라서 다음 표현식은 모두 적합하다. (변수 x는 사전에 어떤 값이 대입되었다고 가정한다.)

```
17
x
x + 17
```

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

```
1 + 1
```

```
## [1] 2
```

하지만, 스크립트에서는 표현식 자체로 어떠한 것도 수행하지는 않는다. 초심자에게 혼란스러운 점이다.

**연습문제.** R 인터프리터에 다음 문장을 입력하고 결과를 보세요.

```
5
x <- 5
x + 1
```

## 3.7 연산자 적용 우선순위

1개 이상의 연산자가 표현식에 등장할 때, 연산자 평가 순서는 **우선순위 규칙(rules of precedence)**에 따른다. 수학 연산자에 대해서 파이썬은 수학적 관례를 동일하게 따른다. 영어 두문어 **PEMDAS**는 기억하기 좋은 방식이다.

- **괄호(Parentheses)**는 가장 높은 순위를 가지고 여러분이 원하는 순위에 맞춰 실행할 때 사용한다. 괄호내의 식이 먼저 실행되기 때문에 `2 * (3-1)` 은 4가 정답이고, `(1+1)**(5-2)`는 8이다. 괄호를 사용하여 표현식을 좀더 읽기 쉽게 하려고 사용하기도 한다. `(minute * 100) / 60` 는 실행순서가 결과값에 영향을 주지 않지만 가독성이 상대적으로 더 좋다.
- **지수승(Exponentiation)**이 다음으로 높은 우선순위를 가진다. 그래서 `2**1+1`는 4가 아니라 3이고, `3*1**3`는 27이 아니고 3이다.
- **곱셈(Multiplication)과 나눗셈(Division)**은 동일한 우선순위를 가지지만, 덧셈(Addition), 뺄셈(Substraction)보다 높은 우선 순위를 가진다. 덧셈과 뺄샘은 같은 실행 우선순위를 갖는다. `2*3-1`는 4가 아니고 5이고, `6+4/2`는 5가 아니라 8이다.

- 같은 실행 순위를 갖는 연산자는 왼쪽에서부터 오른쪽으로 실행된다. `5-3-1` 표현식은 3이 아니고 1이다. 왜냐하면 `5-3`이 먼저 실행되고 나서 2에서 1을 빼기 때문이다.

여러분이 의도한 순서대로 연산이 수행될 수 있도록, 좀 의심스러운 경우는 항상 괄호를 사용한다.

## 3.8 나머지 연산자

**나머지 연산자(modulus operator)**는 정수에 사용하며, 첫번째 피연산자를 두번째 피연산자가 나눌 때 나머지 값이 생성된다. 파이썬에서 나머지 연산자는 퍼센트 기호(%)다. 구문은 다른 연산자와 동일하다.

7을 3으로 나누면 몫이 2가 되고 나머지가 1이 된다.

나머지 연산자가 놀랍도록 유용다다. 예를 들어 한 숫자를 다른 숫자로 나눌 수 있는지 없는지를 확인할 수도 있다. `x %% y` 값이 0 이라면, x를 y로 나눌 수 있다.

또한, 숫자에서 가장 오른쪽 숫자를 분리하는데도 사용된다. 예를 들어 `x %% 10`은 x가 10진수인 경우 가장 오른쪽 숫자를 뽑아낼 수 있고, 동일한 방식으로 `x %% 100`은 가장 오른쪽 2개 숫자를 뽑아낼 수도 있다.

나눗셈 연산자의 경우 `minute` 값은 59, 보통 59를 60으로 나누면 0 대신에 0.98333 입니다.

```
minute <- 59
minute / 60
```

```
## [1] 0.983
```

하지만, 몫을 `minute %/% 60`와 같이 계산하여 0 얻고, 나머지를 `minute %% 60`와 같이 계산하여 59을 얻게 된다.

## 3.9 문자열 연산자

`+` 연산자는 문자열는 동작하지 않는다. 대신에 문자열 끝과 끝을 붙이는 **연결(concatenation)** 작업을 수행할 때 `paste()` 함수를 사용한다. 예를 들어,

```
first <- 10
second <- 15
first + second
```

```
## [1] 25
```

```
first <- '100'
second <- '150'
paste(first, second, sep="")
```

```
## [1] "100150"
```

상기 프로그램 출력은 100150 이다.

## 3.10 입력값 받기

때때로 키보드를 통해서 사용자로부터 변수에 대한 값을 입력받고 싶을 때가 있다. 키보드로부터 입력값을 받는 `readline()` 이라는 내장(built-in) 함수를 R에서 제공한다. 입력 함수가 호출되면, R은 실행을 멈추고 사용자가 무언가 입력하기를 기다린다. 사용자가 Return (리턴) 혹은 Enter (엔터) 키를 누르게 되면 프로그램이 다시 실행되고, `readline()`은 사용자가 입력한 것을 문자열로 반환한다.

```
input <- readline()

R    .
input
[1] "R    ."
```

사용자로부터 입력 받기 전에 프롬프트에서 사용자가 어떤 값을 입력해야 하는지 정보를 제공하는 것도 좋은 생각이다. 입력을 받기 위해 잠시 멈춰있을 때, 사용자에게 표시되도록 readline() 함수에 문자열을 전달할 수 있다.

```
input <- readline(prompt="      : ")

      : R    .
input
[1] "R    ."
```

경우에 따라서 프롬프트의 끝에 `\n` 을 넣는 경우도 있는데 **개행(newline)**을 의미한다. 개행은 줄을 바꾸게 하는 특수 문자다. 사용자 입력이 프롬프트 밑에 출력되도록 줄바꿈이 필요한 경우 사용한다.

만약 사용자가 정수를 입력하기를 바란다면, int()함수를 사용하여 반환되는 값을 정수(int)로 자료형을 변환한다.

```
prompts <- '       ? '
speed <- readline(prompt=prompts)
       ? 20
as.integer(speed) + 5
[1] 25
```

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

```
speed <- readline(prompt=prompts)
       ?      !!!
as.integer(speed) + 5
[1] NA
```

나중에 이런 종류의 오류를 어떻게 다루는지 배울 것이다.

## 3.11 주석

프로그램이 커지고 복잡해짐에 따라 가독성은 떨어진다. 형식 언어(formal language)는 촘촘하고 코드 일부분도 읽기 어렵고 무슨 역할을 왜 수행하는지 이해하기 어렵다.

이런 이유로 프로그램이 무엇을 하는지를 자연어로 프로그램에 노트를 달아두는 것은 좋은 생각이다. 이런 노트를 **주석(Comments)**이라고 하고 # 기호로 시작한다.

```
#
percentage <- (minute * 100) / 60
```

상기 사례의 경우, 주석 자체가 한줄이다. 주석을 프로그램의 맨 뒤에 놓을 수도 있다.

```
percentage <- (minute * 100) / 60 #
```

뒤의 모든 것은 무시되기 때문에 프로그램에는 아무런 영향이 없다.

명확하지 않은 코드의 기능을 문서화할 때 주석은 가장 유용하게 된다. 프로그램을 읽는 사람이 코드가 무엇을 하는지 이해한다고 가정하는 것은 일리가 있다. 왜 그런지를 이유를 설명하는 것은 더욱 유용하다.

다음의 주석은 코드와 중복되어 쓸모가 없다.

```
v <- 5 # 5 v
```

다음의 주석은 코드에 없는 유용한 정보가 있다.

```
v <- 5 #  /
```

좋은 변수명은 주석을 할 필요를 없게 만들지만, 지나치게 긴 변수명은 읽기 어려운 복잡한 표현식이 될 수 있다. 그래서 상충관계(trade-off)가 존재한다.

## 3.12 연상되는 변수명

변수를 이름 짓는데 단순한 규칙을 지키고 예약어를 피하기만 하다면, 변수이름을 작명할 수 있는 무척이나 많은 경우의 수가 존재한다. 처음에 이렇게 넓은 선택폭이 오히려 프로그램을 읽는 사람이나 프로그램을 작성하는 사람 모두에게 혼란을 줄 수 있다. 예를 들어, 다음의 3개 프로그램은 각 프로그램이 달성하려하는 관점에서 동일하지만, 여러분이 읽고 이해하는데는 많은 차이점이 있다.

```
a <- 35.0
b <- 12.50
c <- a * b
print(c)
```

```
## [1] 438
```

```
hours <- 35.0
rate <- 12.50
```

```
pay <- hours * rate
print(pay)
```

```
## [1] 438
```

```
x1q3z9ahd <- 35.0
x1q3z9afd <- 12.50
x1q3p9afd <- x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

```
## [1] 438
```

R 인터프리터는 상기 3개 프로그램을 정확하게 동일하게 바라보지만, 사람은 이들 프로그램을 매우 다르게 보고 이해한다. 사람은 가장 빨리 두 번째 프로그램의 **의도**를 알아차린다. 왜냐하면 각 변수에 무슨 데이터가 저장될지에 관해서, 프로그래머의 **의도**를 반영하는 변수명을 사용했기 때문이다.

현명하게 선택된 변수명을 연상기호 변수명("mnemonic variable name")이라고 한다. 연상되기 좋은 영어 단어 ("mnemonic")은 기억을 돕는다는 뜻이다. 왜 변수를 생성했는지 기억하기 좋게 하기 위해서 연상하기 좋은 변수명을 선택한다.

매우 훌륭하게 들리고, 연상하기 좋은 변수명을 만드는게 좋은 아이디어 같지만, 기억하기 좋은 변수명은 초보 프로그래머가 코드를 파싱(parsing)하고 이해하는데 걸림돌이 되기도 한다. 왜냐하면 얼마되지 않는 예약어도 기억하지 못하고, 변수명이 때때로 너무 서술적이라 마치 일반적으로 사용하는 언어처럼 보이고 잘 선택된 변수명처럼 보이지 않기 때문이다.

어떤 데이터를 반복하는 다음 파이썬 코드를 살펴보자. 곧 반복 루프를 살펴보겠지만, 다음 코드가 무엇을 의미하는지 알기 위해서 퍼즐을 풀어보자.

```
words <- c(" ", " ", " ", " ")

for(word in seq_along(words)) {
  print(words[word])
}
```

```
## [1] " "
## [1] " "
## [1] " "
## [1] " "
```

무엇이 일어나고 있는 것일까? for, word, in 등등 어느 토큰이 예약어일까? 변수명은 무엇일까? 파이썬은 기본적으로 단어의 개념을 이해할까? 초보 프로그래머는 어느 부분 코드가 이 예제와 동일해야만 하는지 그리고, 어느 부분 코드가 프로그래머 선택에 의한 것인지 분간하는데 고생을 한다.

다음의 코드는 위의 코드와 동일하다.

```
for(slice in seq_along(pizza)) {
  print(pizza[slice])
```

```
}
```

초보 프로그래머가 이 코드를 보고 어떤 부분이 R 예약어이고 어느 부분이 프로그래머가 선택한 변수명인지 알기 쉽다. R이 피자와 피자조각에 대한 근본적인 이해가 없고, 피자는 하나 혹은 여러 조각으로 구성된다는 근본적인 사실을 알지 못한다는 것은 자명하다.

하지만, 작성한 프로그램이 데이터를 읽고 데이터에 있는 단어를 찾는다면 피자(pizza)와 피자조각(slice)은 연상하기 좋은 변수명이 아니다. 이것을 변수명으로 선행하게 되면 프로그램의 의미를 왜곡시킬 수 있다.

좀 시간을 보낸 후에 가장 흔한 예약어에 대해서 알게 될 것이고, 이들 예약어가 어느 순간 여러분에게 눈에 띄게 될 것이다.

```
for(word in seq_along(words)) {
  print(words[word])
}
```

R에서 정의된 코드 일부분(**for**, **in**, **print**)은 예약어로 굵게 표시되어 있고, 프로그래머가 생성한 변수명(word, words)는 굵게 표시되어 있지 않다. 대다수 텍스트 편집기는 R 구문을 인지하고 있어서, R 예약어와 프로그래머가 작성한 변수를 구분하기 위해서 색깔을 다르게 표시한다. 잠시 후에 여러분은 R을 읽고 변수와 예약어를 빠르게 구분할 수 있을 것이다.

## 3.13 디버깅(Debugging)

이 지점에서 여러분이 저지르기 쉬운 구문 오류는 `odd~job`, `US$` 같은 특수문자를 포함해서 잘못된 변수명을 생성하는 것과 `repeat`, `while`같은 예약어를 변수명으로 사용하는 것이다.

변수명에 공백을 넣는다면, 파이썬은 연산자 없는 두 개의 피연산자로 생각한다.

```
bad name <-  5
Error: unexpected symbol in "bad name"
```

구문 오류에 대해서, 오류 메세지는 그다지 도움이 되지 못한다. 가장 흔한 오류 메세지는 `Error: unexpected symbol in "bad name"`인데 둘다 그다지 오류에 대한 많은 정보를 주지는 못한다.

여러분이 많이 범하는 실행 오류는 정의 전에 사용("use before def")하는 것으로 변수에 값을 대입하기 전에 변수를 사용할 경우 발생한다. 여러분이 변수명을 잘못 쓸 때도 발생할 수 있다.

```
principal <- 327.68
interest <- principle * rate
Error: object 'principle' not found
```

변수명은 대소문자를 구분한다. 그래서, `LaTeX` $LaTeX$, `latex`와 같지 않다.

이 지점에서 여러분이 범하기 쉬운 의미론적 오류는 연산자 우선 순위일 것이다. 예를 들어 $frac12\pi$를 계산하기 위해서 다음과 같이 프로그램을 작성하게 되면 …

```
1.0 / 2.0 * pi
```

나눗셈이 먼저 일어나서 이 되는데 의도한 것과 같지 않다. R으로 하여금 여러분이 작성한 의도를 알게할 수는 없다. 그래서 이런 경우 오류 메세지는 없지만, 여러분은 잘못된 답을 얻게 된다.

## 3.14 용어 설명

- **대입(assignment)**: 변수에 값을 대입하는 문장
- **연결(concatenate)**: 두 개의 피연산자 끝과 끝을 합치는 것
- **주석(comment)**: 다른 프로그래머나 소스코드를 읽는 다른 사람을 위한 프로그램 정보로 프로그램의 실행에는 아무런 영향이 없다.
- **평가(evaluate)**: 하나의 값을 만들도록 연산을 실행함으로써 표현식을 간단히 하는 것
- **표현식(expression)**: 하나의 결과값을 만드는 변수, 연산자, 값의 조합
- **부동 소수점(floating-point)**: 소수점을 가진 숫자를 표현하는 자료형
- **버림 나눗셈(floor division)] 두 숫자를 나누어 소수점이하 부분을 절사하는 연산자
- **정수(integer)**: 완전수를 나타내는 자료형
- **예약어(keyword)**: 컴파일러가 프로그램을 파싱하는데 사용하기 위해서 이미 예약된 단어; if, def, while 같은 예약어를 변수명으로 사용할 수 없다.
- **연상기호(mnemonic)**: 기억 보조. 변수에 저장된 것을 기억하기데 도움이 되도록 변수에 연상되는 이름을 부여한다.
- **나머지 연산자(modulus operator)**: 퍼센트 기호 ()로 표시되고 정수를 가지고 한 숫자를 다른 숫자로 나누었을 때 나머지를 생성하는 연산자
- **피연산자(operand)**: 연산자가 연산을 수행하는 값중의 하나
- **연산자(operator)**: 덧셈, 곱셈, 문자열 결합 같은 간단한 연산을 표현하는 특별 기호
- **우선순위 규칙(rules of precedence)**: 다수의 연산자와 피연산자를 포함한 표현식이 평가되는 실행 순서를 규정한 규칙 집합
- **문장(statement)**: 명령이나 액션을 나타내는 코드 부문. 지금까지 assignment, print 문을 보았다.
- **문자열(string)**: 일련의 문자를 나타내는 형식
- **자료형(type)**: 값의 범주. 지금까지 여러분이 살펴본 자료형은 정수 (int), 부동 소수점수 (float), 문자열 (str) 이다.
- **값(value)**: 숫자나 문자 같은 프로그램이 다루는 데이터의 기본 단위중 하나
- **변수(variable)**: 값을 참조하는 이름

## 3.15 연습문제

1. `readline()`을 사용하여 사용자의 이름을 입력받고 환영하는 프로그램을 작성하세요.

```
      :
```

2. 급여를 지불하기 위해서 사용자로부터 근로시간과 시간당 임금을 계산하는 프로그램을 작성하세요.

```
      : 35.51
      : 7530
  : 263550
```

지금은 급여가 정확하게 소수점 두자리까지 표현되지 않아도 된다. 만약 원하다면, R 내장 `round()` 함수를 사용하여 소수점 아래 반올림하여 정수로 작성할 수도 있다.

3. 다음 대입 문장을 실행한다고 가정합시다.

```
width <- 17
height <- 12.0
```

다음 표현식 각각에 대해서, 표현식의 값(value)과 (표현식 값의) 자료형(type)을 작성하세요.

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

정답을 확인하기 위해서 R 인터프리터를 사용하세요.

4. 사용자로부터 섭씨 온도를 입력받아 화씨온도로 변환하고, 변환된 온도를 출력하는 프로그램을 작성하세요.

# Chapter 4

# 조건부 실행

## 4.1 부울 표현식

**부울 표현식(boolean expression)**은 참(TRUE) 혹은 거짓(FALSE)를 지닌 표현식이다. 다음 예제는 == 연산자를 사용하여 두 개 피연산자를 비교하여 값이 동일하면 참(TRUE), 그렇지 않으면 거짓(FALSE)을 산출한다.

```
5 == 5
```

```
## [1] TRUE
```

```
5 == 6
```

```
## [1] FALSE
```

참(TRUE)과 거짓(FALSE)은 논리형(logical) 자료형(type)에 속하는 특별한 값으로 문자열은 아니다.

```
typeof(TRUE)
```

```
## [1] "logical"
```

```
typeof(FALSE)
```

```
## [1] "logical"
```

== 연산자는 **비교 연산자(comparison operators)** 중 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 == y # x는 y와 같다.

- x != y # x는 y와 개체가 동일하지 않다.

상기 연산자가 친숙할지 모르지만, R 기호는 수학 기호와 다르다. 일반적인 오류로 비교를 해서 동일하다는 의미로 == 연산자 대신에 = 를 사용하는 것이다. R에서 대입연산자로 <-을 사용하지만, = 으로 사용해도 프로그램은 돌아간다. = 연산자는 대입 연산자이고, == 연산자는 비교 연산자다. =<, => 같은 비교 연산자는 R에는 없다.

## 4.2 논리 연산자

세개 **논리 연산자(logical operators)**: `&`, `||`, `!` 이 있다. 논리 연산자 의미는 수식기호 의미와 유사하다. 영어로 표현하면 `&`은 `and`, `||`은 `or`, `!`은 `not`이 된다. 예를 들어,

- `x > 0 & x < 10`

x 가 0 보다 크다. 그리고(and), 10 보다 작으면 참이다.

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

마지막으로 `!` 연산자는 부울 연산 표현식을 부정한다. `x > y` 가 거짓이면, `!(x > y)`은 참이다. 즉, x이 y 보다 작거나 같으면 참이다.

엄밀히 말해서, 논리 연산자의 두 피연산자는 모두 부울 표현식이지만, R에서 그다지 엄격하지는 않다. 0 이 아닌 임의의 숫자 모두 "참(TRUE)"으로 해석된다. 일반적으로 참(TRUE)이면 `1`, 그렇지 않는 경우 `0`으로 표현해서 사용한다.

```
17 & TRUE
```

```
## [1] TRUE
```

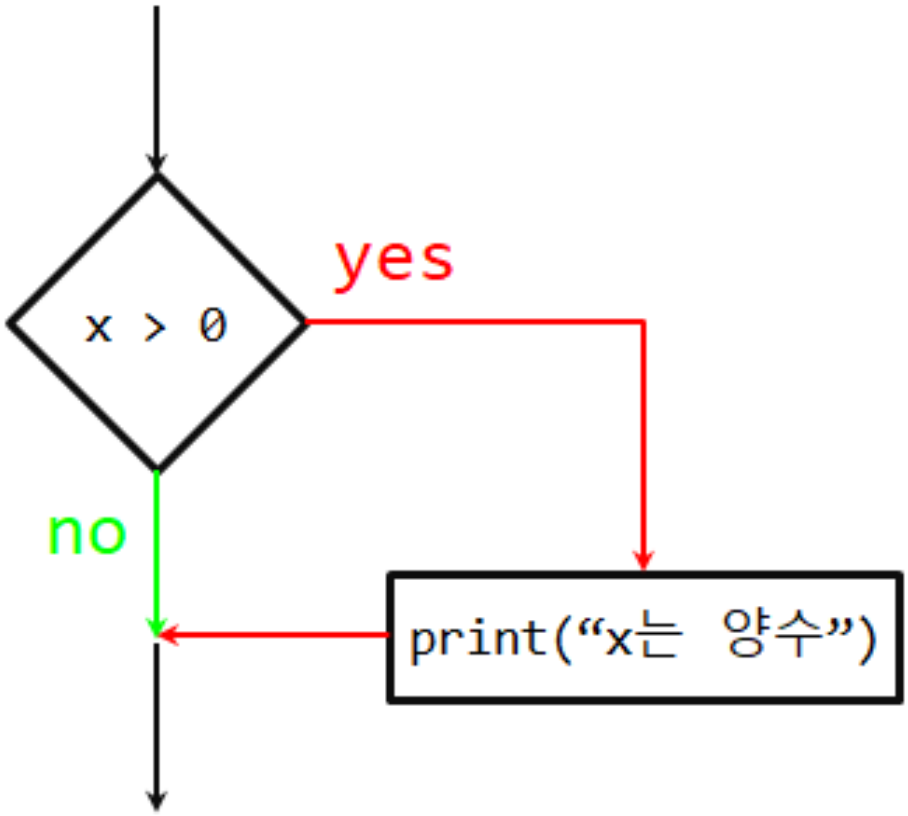
이러한 유연함이 유용할 수 있으나, 혼란을 줄 수도 있으니 유의해서 사용해야 한다. 무슨 일을 하고 있는지 정확하게 알지 못한다면 피하는게 상책이다.

## 4.3 조건문 실행

유용한 프로그램을 작성하기 위해서 거의 항상 조건을 확인하고 조건에 따라 프로그램 실행을 바꿀 수 있어야 한다. **조건문(Conditional statements)**은 그러한 능력을 부여한다. 가장 간단한 형태는 `if` 문이다.

```
if (x > 0) {
  print('x    ')
}
```

`if`문 뒤에 불 표현식(boolean expression)을 조건(condition)이라고 한다.




만약 조건문이 참이면, 첫번째 괄호로 둘러싼 문장이 실행된다. 만약 조건문이 거짓이면, 첫번째 괄호로 둘러싼 문장의 실행을 건너뛴다.

`if`문은 함수 정의, `for` 반복문과 동일한 구조를 가진다. `if`문은 `(`으로 시작되고, `)`으로 끝나는 헤더 머리부문과 괄호({, })로 둘러싼 몸통 블록(block)으로 구성된다. `if`문처럼 문장이 한 줄 이상에 걸쳐 작성되기 때문에 **복합 문장(compound statements)**이라고 한다.

`if`문 몸통 부문에 작성되는 실행 문장 숫자에 제한은 없으나 최소한 한 줄은 있어야 한다. 때때로, 몸통 부문에 어떤 문장도 없는 경우가 있다. 아직 코드를 작성하지 않아서 자리만 잡아 놓는 경우로, 그냥 놔두면 된다. 즉, 아무것도 작성하지 않고 괄호 내부를 텅비워든다. 파이썬의 경우 아무것도 수행하지 않는 `pass`문을 넣어야 되는 것과 대비된다.

```
if (x > 0) {
  #               .          .
}
```

`if`문을 R 인터프리터에서 타이핑하고 엔터를 치게 되면, 명령 프롬프트가 `+`로 바뀐다. 따라서 다음과 같이 `if`문 몸통 부분을 작성중에 있다는 것을 나타낸다.

```
> x <- 3
>
> if (x < 10) {
+ print(' ')
+ }
[1] " "
```

## 4.4 대안 실행

if문의 두 번째 형태는 **대안 실행(alternative execution)**이다. 대안 실행의 경우 두 가지 경우의 수가 존재하고, 조건이 어느 방향으로 실행할 것인지 결정한다. 구문(Syntax)은 아래와 같다.

```
if (x %% 2 == 0){
  print("x   ")
} else {
  print("x   ")
}
```

x를 2로 나누었을 때, 0 이되면, x는 짝수이고, 프로그램은 짝수("x는 짝수")라는 결과 메시지를 출력한다. 만약 조건이 거짓이라면, 두 번째 몸통 부문 문장이 실행된다.

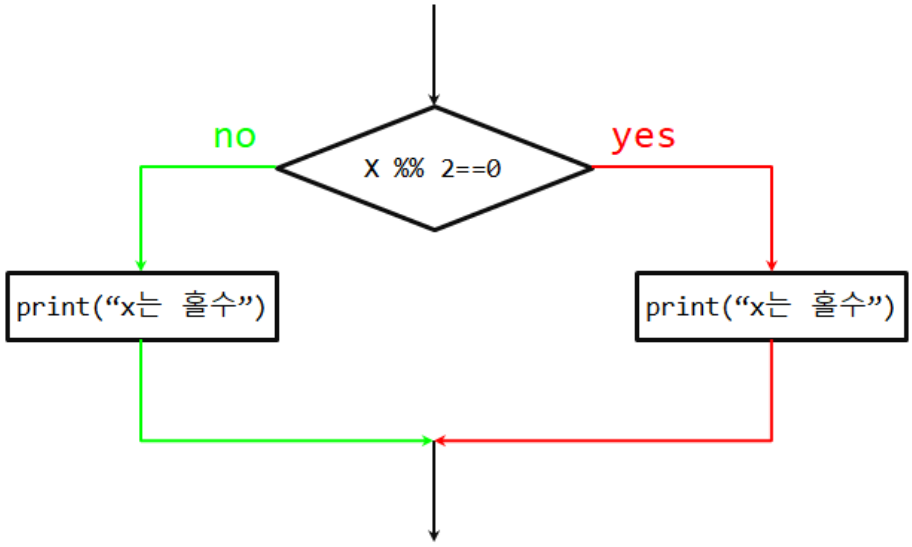



Figure 4.1: if-else문

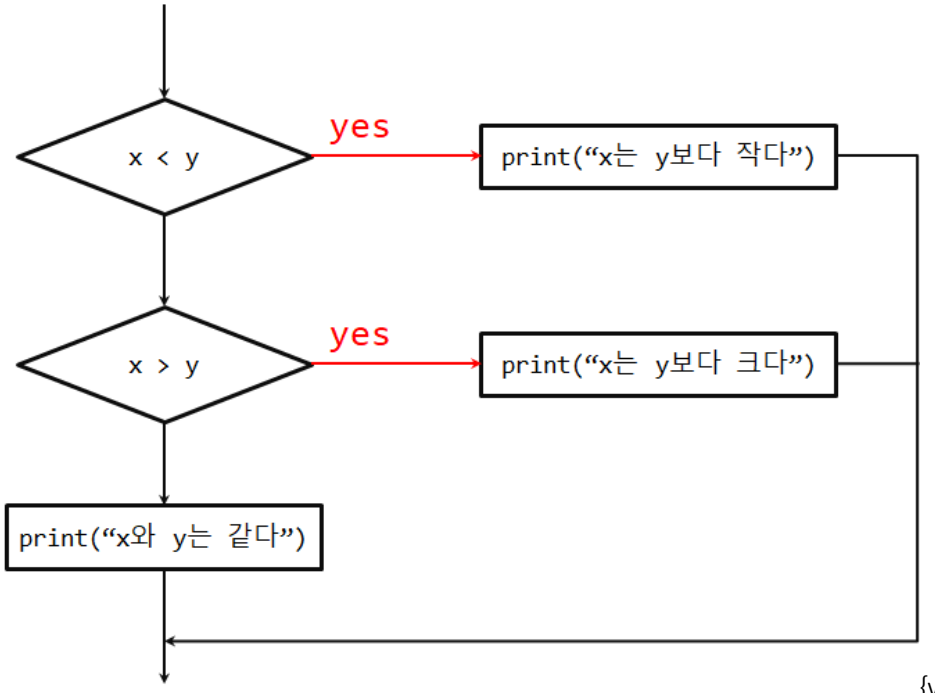
조건은 참 혹은 거짓이어서, 대안 중 하나만 정확하게 실행된다. 대안을 **분기(Branch)**라고도 하는데 이유는 실행 흐름이 분기되기 때문이다.

## 4.5 연쇄 조건문

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

```
if (x < y){
  print("x  y   ")
} else if (x > y) {
  print("x  y   ")
} else {
  print("x  y   ")
}
```

"else if"로 연쇄 조건문을 표현하는데 주목한다. 이번에도 단 한번의 분기만 실행된다.




{width = 100%}

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

```
if (choice == 'a') {
  print("Bad guess")
} else if(choice == 'b') {
  print("Good guess")
} else if (choice == 'c') {
```

```
  print('Close, but not correct')
}
```

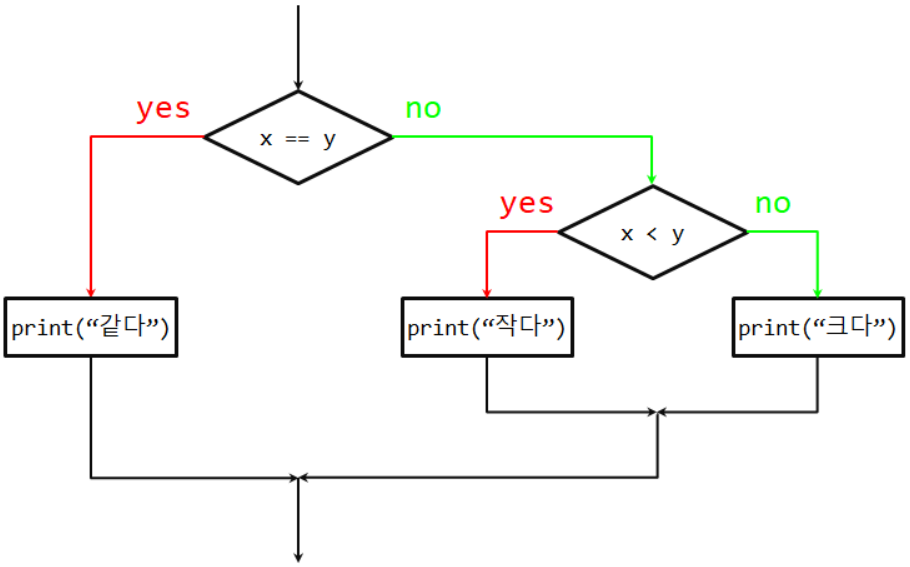
각 조건은 순서대로 점검한다. 만약 첫 번째가 거짓이면, 다음을 점검하고 계속 점검해 나간다. 순서대로 진행 중에 하나의 조건이 참이면, 해당 분기가 수행되고, if문 전체는 종료된다. 설사 하나 이상의 조건이 참이라고 하더라도, 첫 번째 참 분기만 수행된다.

## 4.6 중첩 조건문

하나의 조건문이 조건문 내부에 중첩될 수 있다. 다음과 같이 삼분 예제를 작성할 수 있다.

```
if (x < y){
  print("x  y   ")
} else {
  if (x > y) {
    print("x  y   ")
  } else {
    print("x  y   ")
  }
}
```

바깥 조건문에는 두 개의 분기가 있다. 첫 분기는 간단한 문장을 담고 있다. 두 번째 분기는 자체가 두 개의 분기를 가지고 있는 또 다른 if문을 담고 있다. 자체로 둘다 조건문이지만, 두 분기 모두 간단한 문장이다.




{width = 100%}

괄호를 사용하는 것이 구조를 명확히 하지만, **중첩 조건문의 경우 가독성이 급격히 저하된다. 일반적으로, 가능하면 중첩 조건문을 피하는 것을 권장한다.**

논리 연산자를 사용하여 중첩 조건문을 간략히 할 수 있다. 예를 들어, 단일 조건문으로 가지고 앞의 코드를 다음과 같이 재작성할 수 있다.

```
if (0 < x) {
  if (x < 10) {
    print('x      .')
  }
}
```

print문은 두 개 조건문을 통과될 때만 실행돼서, & 연산자와 동일한 효과를 거둘 수 있다.

```
if (0 < x & x < 10) {
  print("x      .")
}
```

## 4.7 try와 catch를 활용한 예외 처리

함수 readline()와 as.integer()을 사용하여 앞에서 사용자가 타이핑한 숫자를 읽어 정수로 파싱하는 프로그램 코드를 살펴보았다. 또한 이렇게 코딩하는 것이 얼마나 위험한 것인지도 살펴보았다.

```
speed <- readline(prompt=prompts)
       ?      !!!
as.integer(speed) + 5
[1] NA
```

R 인터프리터에서 상기 문장을 실행하면, 인터프리터에서 새로운 프롬프트로 되고, "이런(oops)" 잠시 후에, 다음 문장 실행으로 넘어간다.

하지만, 만약 코드가 R 스크립트로 실행이 되어 오류가 발생하면, 역추적해서 그 지점에서 즉시 멈추게 된다. 다음에 오는 문장은 실행하지 않는다.

화씨 온도를 섭씨 온도로 변환하는 간단한 프로그램이 있다. 다소 길이가 긴데, R 콘솔에서 실행하는 코드와 쉘에서 실행할 때 사용자 입력을 받는 것을 달리 처리하기 위함이다. 또한, 한글을 넣게 되면 오류가 발생하니, 프롬프트 메시지는 영어로 처리한다.

```
# fahrenheit.R
if ( interactive() ){
  inp <- readLines(prompt = "Enter Fahrenheit Temperature: ")
  fahr <- as.numeric(inp)
  cel <- (fahr - 32.0) * 5.0 / 9.0
  print(cel)
} else {
  cat("Enter Fahrenheit Temperature: ")
```

```
  inp <- readLines("stdin", n=1)
  fahr <- as.numeric(inp)
  cel <- (fahr - 32.0) * 5.0 / 9.0
  print(cel)
}
```

이 코드를 실행해서 적절하지 않은 입력값을 넣게 되면, 다소 불친절한 오류 메시지와 함께 간단히 작동을 멈춘다.

```
D:\docs\r4inf\code> Rscript fahrenheit.R
Enter Fahrenheit Temperature: 72
[1] 22.22222

D:\docs\r4inf\code> Rscript fahrenheit.R
Enter Fahrenheit Temperature: fred
[1] NA
():
   NA
```

이런 종류의 예측하거나, 예측하지 못한 오류를 다루기 위해서 R에는 "try / except"로 불리는 조건 실행 구조가 내장되어 있다. `try`와 `except`의 기본적인 생각은 일부 명령문에 문제가 있다는 것을 사전에 알고 있고, 만약 그 때문에 오류가 발생하게 된다면 대신 프로그램에 추가해서 명령문을 실행한다는 것이다. `except` 블록의 문장은 오류가 없다면 실행되지 않는다.

문장 실행에 대해서 R `try`, `except` 기능을 보험으로 생각할 수도 있다.

온도 변환기 프로그램을 다음과 같이 재작성한다.

```
if ( interactive() ){
  inp <- readLines(prompt = "Enter Fahrenheit Temperature: ")
  fahr <- as.numeric(inp)
  cel <- (fahr - 32.0) * 5.0 / 9.0
  print(cel)
} else {
  cat("Enter Fahrenheit Temperature: ")
  inp <- readLines("stdin", n=1)

  tryCatch({
    fahr <- as.numeric(inp)
    cel <- (fahr - 32.0) * 5.0 / 9.0
    print(cel)
  },
  error = function(err) print(paste("ERROR: ", err))
  )
}
```

R은 tryCatch 블록 문장을 우선 실행한다. 만약 모든 것이 순조롭다면, `error` 블록을

건너뛰고, 다음 코드를 실행한다. 만약 `tryCatch` 블록에서 `error`가 발생하면, R은 `tryCatch` 블록에서 빠져 나와 `error` 문장을 수행한다.

`tryCatch`문으로 예외사항을 다루는 것을 예외 처리한다(catching an exception)고 부른다. 예제에서 `error` 절에서는 단순히 오류 메시지를 출력만 한다. 대체로, 예외 처리를 통해서 오류를 고치거나, 재시작하거나, 최소한 프로그램이 정상적으로 종료될 수 있게 한다.

## 4.8 논리 연산식의 단락(Short circuit) 평가

`x >= 2 & (x/y) > 2` 와 같은 논리 표현식을 R에서 처리할 때, 왼쪽에서부터 오른쪽으로 표현식을 평가한다. `&` 정의 때문에 `x` 가 2보다 작다면, `x >= 2`은 거짓(FALSE)으로, 전체적으로 `(x/y) > 2` 이 참(TRUE) 혹은 거짓(FALSE) 이냐에 관계없이 거짓(FALSE)이 된다.

나머지 논리 표현식을 평가해도 나아지는 것이 없다고 R이 자동으로 탐지할 때, 평가를 멈추고 나머지 논리 표현식에 대한 연산도 중지한다. 최종값이 이미 결정되었기 때문에 더 이상의 논리 표현식의 평가가 멈출 때, 이를 **단락(Short-circuiting) 평가**라고 한다.

좋은 점처럼 보일 수 있지만, 단락 행동은 가디언 패턴(guardian pattern)으로 불리는 좀 더 똑똑한 기술로 연계된다. R 인터프리터의 다음 코드를 살펴보자.

```
x <- 6
y <- 2
x >= 2 & (x/y) > 2
```

```
## [1] TRUE
```

```
x <- 1
y <- 0
x >= 2 & (x/y) > 2
```

```
## [1] FALSE
```

```
x <- 6
y <- 0
x >= 2 & (x/y) > 2
```

```
## [1] TRUE
```

세번째 연산은 일반적으로 실패하는데 이유는 (x/y) 연산을 평가할 때 y 가 0 이어서 실행오류가 발생된다. 하지만, R에서는 0으로 나누게 되면 `Inf`가 되어 계산결과는 참이 되어 전체적으로 참이 된다. 하지만, 두 번째 예제의 경우 거짓(FALSE)하지 않는데 이유는 x >= 2 이 거짓(FALSE) 으로, 전체가 거짓(FALSE)이 되어 단락(Short-circuiting) 평가 규칙에 의해 (x/y) 평가는 실행되지 않게 된다.

평가 오류를 발생하기 전에 가디언(gardian) 평가식을 전략적으로 배치해서 논리 표현식을 다음과 같이 구성한다.

```
x <- 1
y <- 0
x >= 2 & y != 0 & (x/y) > 2
```

```
## [1] FALSE
```

```
x <- 6
y <- 0
x >= 2 & y != 0 & (x/y) > 2
```

```
## [1] FALSE
```

```
x >= 2 & (x/y) > 2 & y != 0
```

```
## [1] FALSE
```

첫 번째 논리 표현식은 `x >= 2` 이 거짓(FALSE) 이라 `&`에서 멈춘다. 두 번째 논리 표현식은 `x >= 2` 이 참(TRUE), `y != 0` 은 거짓(FALSE) 이라 `(x/y)`까지 갈 필요가 없다. 세 번째 논리 표현식은 `(x/y)` 연산이 끝난 후에 `y != 0` 이 수행되어서 오류가 발생한다.

두 번째 표현식에서 `y` 가 `0` 이 아닐 때만, (x/y)을 실행하도록 `y != 0` 이 가디언(gardian) 역할을 수행한다고 말할 수 있다.

## 4.9 디버깅

오류가 발생했을 때, 파이썬 화면에 출력되는 역추적(traceback)에는 상당한 정보가 담겨있다. 하지만,특히 스택에 많은 프레임이 있는 경우 엄청나게 보여 엄두가 나지 않을 수도 있다. 대체로 가장 유용한 정보는 다음과 같은 것이 있다.

- 어떤 종류의 오류인가.
- 어디서 발생했는가.

구문 오류는 대체로 발견하기 쉽지만, 몇 가지는 애매하다. 파이썬의 경우 공백(space)과 탭(tab)의 차이가 눈에 보이지 않아 통상 무시하고 넘어가기 쉽기 때문에 공백 오류를 잡아내기가 까다롭다. R로 텍스트 데이터를 분석할 경우 눈에는 보이지 않지만 공백문자(White space) 문자가 여러가지 문제를 일으키는 경우가 많다. 특히 한글 인코딩과 결합될 경우 더욱 그렇다.

대체로 오류 메시지는 문제가 어디에서 발견되었는지를 지칭하지만, 실제 오류는 코드 앞에 종종 선행하는 줄에 있을 수 있다.

동일한 문제가 실행 오류에도 있다. 데시벨(decibels)로 신호 대비 잡음비를 계산한다고 가정하자. 공식은 $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$ 이다. R에서 아래와 같이 작성할 수 있다.

```
signal_power <- 9
noise_power <- 10
ratio <- signal_power / noise_power
```

```
decibels <- 10 * log10(ratio)
print(decibels)
```

signal_power 와 noise_power 를 부동 소수점값으로 표현되어 R코드에는 문제가 없지만, 파이썬 2로 실행하게 되면 다음과 같은 오류가 나온다.

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

오류 메지지가 5번째 줄에 있다고 지칭하지만, 잘못된 것은 없다. 실제 오류를 발견하기위해서,출력값이0인ratio값을print문을사용해서출력하는것이 도움이 된다. 문제는 4번째 줄에 있는데, 왜냐하면 두 정수를 나눌 때 내림 나눗 셈을 했기 때문입니다.

대체로, 오류 메시지는 문제가 어디에서 발견되었는지를 알려주지만, 종종 문제의 원인이 어디에서 발생했는지는 알려주지 않는다.

## 4.10 용어 정의

- 몸통 부문(body)**: 복합 문장 내부에 일련의 문장문
- 부울 표현식(boolean expression)**: 참(TRUE) 혹은 거짓(FALSE)의 값을 가지는 표현식
- 분기(branch)**: 조건문에서 대안 문장의 한 흐름
- 연쇄 조건문(chained conditional)**: 일련의 대안 분기가 있는 조건문
- 비교 연산자(comparison operator)**: 피연산자를 ==, !=, >, <, >=, <=로 비교하는 연산자
- 조건문(conditional statement)**: 조건에 따라 명령의 흐름을 제어하는 명령문
- 조건(condition)**: 조건문에서 어느 분기를 실행할지 결정하는 불 표현식
- 복합문(compound statement)**: 머리부문(head)과 몸통부문(body)으로 구성된 문장. 머리부문은 콜론(:)으로 끝나며, 몸통부문은 머리부문을 기준으로 들여쓰기로 구별된다.
- 가디언 패턴(guardian pattern)**: 단락(short circuit) 행동을 잘 이용하도록 논리 표현식을 구성하는 것
- 논리 연산자(logical operator)**: 불 표현식을 결합하는 연산자 중의 하나 (and, or, not)
- 중첩 조건문(nested conditional)**: 하나의 조건문이 다른 조건문 분기에 나타나는 조건문.
- 역추적(traceback)**: 예외 사항이 발생했을 때 실행되고, 출력되는 함수 리스트
- 단락(short circuit)**: 나머지 표현식 평가를 할 필요없이 최종 결과를 알기 때문에, 파이썬이 논리 표현식 평가를 진행하는 중간에 평가를 멈출 때.

## 4.11 연습문제

1. 40시간 이상 일할 경우 시급을 1.5배 더 종업원에게 지급하는 봉급계산 프로그램을 다시 작성하세요.

```
     : 35.51
     : 7530
 : 263550
```

2. tryCatch를 사용하여 봉급계산 프로그램을 다시 작성하세요. 숫자가 아닌 입력값을 잘 처리해서 숫자 아닌 입력값이 들어왔을 때 메시지를 출력하고 정상적으로 프로그램을 종료하도록 합니다. 다음이 프로그램 출력 결과를 보여줍니다.

```
     : 35.51
     :
,

     : 7
,
```

3. 0.0과 1.0 사이의 점수를 출력하는 프로그램을 작성하세요. 만약 점수가 범위 밖이면 오류를 출력합니다. 만약 점수가 0.0과 1.0 사이라면, 다음의 테이블에 따라 등급을 출력합니다.

```
>= 0.9   A
>= 0.8   B
>= 0.7   C
>= 0.6   D
< 0.6    F

     : 0.95
A

     :
       .

     : 10.0
       .

     : 0.75
C

     : 0.5
F
```

4. 상기 보이는 것처럼 반복적으로 프로그램을 실행해서 다양한 다른 입력값을 테스트해 보세요.

# Chapter 5

# 함수

## 5.1 함수 호출

프로그래밍 문맥에서, **함수(function)**는 연산을 수행하는 명명된 일련의 문장이다. 함수를 정의할 때, 이름과 일련의 문장을 명기한다. 나중에, 함수를 이름으로 "호출(call)"한다. 이미 **함수 호출(function call)**의 예제를 살펴봤다.

```
typeof(32)
```

```
## [1] "double"
```

함수명은 `typeof()`이다. 괄호안의 표현식을 **함수의 인자(argument)**라고 한다. 인자는 함수 입력으로 함수 내부로 전달되는 값이나 변수이다. 앞선 `typeof()` 함수에 대한 결과는 인자의 자료형(type)이다.

통상 함수가 인자를 "받아" 결과를 "반환"한다. 결과를 **결과값(return value)**이라 부른다.

## 5.2 내장(Built-in) 함수

함수를 정의할 필요없이 사용할 수 있는 내장함수가 R에는 많다. 공통의 문제를 해결할 수 있는 함수를 R을 창시자(Ross Ihaka, Robert Gentleman)가 작성해서 누구나 사용할 수 있도록 R에 포함했다.

`max`와 `min` 함수는 벡터 최소값과 최대값을 각각 계산해서 출력한다.

```
max(c(1,2,3,4,5))
```

```
## [1] 5
```

```
min(c(1,2,3,4,5))
```

```
## [1] 1
```

max 함수는 벡터의 “가장 큰 값”, 상기 예제에서 “5”, min 함수는 “가장 작은 값”를, 상기 예제에서는 “1”을 출력한다.

매우 자주 사용되는 또 다른 내장 함수는 얼마나 많은 항목이 있는지 출력하는 length() 함수가 있다. 만약 length() 함수의 인수가 벡터이면 벡터에 있는 원소 갯수를 반환한다.

```
length(c(1,2,3,4,5))
```

```
## [1] 5
```

이들 함수는 벡터에만 국한된 것이 아니라, 뒷장에서 보듯이 다양한 자료형에 사용된다.

내장함수 이름은 사전에 점유된 예약어로 취급해야 한다. 예를 들어 “max”를 변수명으로 사용하지 말아야 한다.

## 5.3 자료형 변환 함수

이런 자료형(type)에서 저런 자료형(type)으로 값을 변환하는 내장 함수가 R에는 있다. as.integer()함수는 임의의 값을 입력받아 변환이 가능하면 정수형으로 변환하고, 그렇지 않으면 오류가 발생한다.

```
as.integer("32")
[1] 32
as.integer("Hello")
[1] NA
Warning message:
NAs introduced by coercion
```

as.integer()는 부동 소수점 값을 정수로 변환할 수 있지만 소수점 이하를 절사한다.

```
as.integer("3.99999")
```

```
## [1] 3
```

```
as.integer("-2.3")
```

```
## [1] -2
```

as.numeric()는 정수와 문자열을 부동 소수점으로 변환한다.

```
as.numeric(32)
```

```
## [1] 32
```

```
as.numeric('3.14159')
```

```
## [1] 3.14
```

as.characer()은 인자를 문자열로 변환한다.

```
as.character(32)
```

```
## [1] "32"
```

```
as.character(3.14159)
```

```
## [1] "3.14159"
```

파이썬을 비롯한 다른 언어에서 다뤄지지 않는 자료형이 **요인(factor)** 이다. 범주형 자료구조를 표현하는 일반적인 자료형으로 데이터 분석 및 모형 개발에 빈번하게 사용된다.

`as.factor()`은 인자를 요인형으로 변환한다.

```
as.factor(c(0, 1))
```

```
## [1] 0 1
## Levels: 0 1
```

```
as.factor(c("male", "female"))
```

```
## [1] male   female
## Levels: female male
```

## 5.4 난수

동일한 입력을 받을 때, 대부분의 컴퓨터는 매번 동일한 출력값을 생성하기 때문에 **결정적(deterministic)**이라고 한다. 결정론이 대체로 좋은 것이다. 왜냐하면, 동일한 결과를 얻는데 동일한 계산을 기대하기 때문입니다. 하지만, 어떤 응용프로그램에 대해서 컴퓨터가 예측불가능하길 바란다. 게임이 좋은 예가 되고, 더 많은 예는 얼마든지 많다.

진실되게 프로그램을 비결정론적으로 만드는 것이 쉽지 않은 것으로 밝혀졌지만, 적어도 비결정론적인 것처럼 보이게 하는 방법은 있다. **의사 난수(pseudorandom numbers)**를 생성하는 알고리즘을 사용하는 것이 방법 중의 하나다. 의사 난수는 이미 결정된 연산에 의해서 생성된다는 점에서 진정한 의미의 난수는 아니지만, 이렇게 생성된 숫자만 봐서는 진정한 난수와 구별하는 것은 불가능에 가깝다.

R은 데이터분석을 위해 태어난 언어라고 할 만큼 기본 내장함수로 다양한 난수 생성기를 갖추고 있다. 물론 난수 생성기로 생성되는 숫자는 의사난수다. 이하 의사 난수 대신 “랜덤(random)”으로 간략히 부르기로 한다.

`runif()` 함수는 0.0 과 1.0 사이 부동 소수점 난수를 반환한다. `runif()` 함수 내부에 `min`, `max` 인자를 지정하여 난수 최소, 최대값을 범위를 설정할 수 있다. 매번 `runif()` 함수를 호출할 때 마다, 이미 생성된 아주 긴 난수열에서 하나씩 하나씩 뽑아 쓰다. 사례로 다음 반복문을 실행하자.

```
for(i in 1:10) {
  cat(runif(1),"\n")
}
```

```
## 0.688
## 0.633
## 0.341
## 0.411
## 0.176
## 0.829
## 0.235
## 0.142
## 0.766
## 0.908
```

상기 프로그램은 0.0 에서 1.0 구간에서 10개 난수 리스트를 생성한다.

**연습문제** 여러분의 컴퓨터에 프로그램을 실행해서, 어떤 난수가 생성되는지 살펴보세요. 한번 이상 프로그램을 실행하여 보고, 어떤 난수가 생성되는지 다시 살펴보세요.

`runif()` 함수는 난수를 다루는 많은 함수 중의 하나다. `sample()` 함수는 정수 난수 범위와 난수 갯수를 매개 변수를 입력받아 최저값(low)과 최고값(high) 사이(최저값과, 최저값 모두 포함) 정수를 반환한다.

```
sample(1:10, 5)
```

```
## [1] 10  4  5  8  3
```

`sample(1:10, 5)` 실행문은 1 ~ 10 사이 정수 10개 중에서 난수로 5개를 추출한다는 뜻이다.

무작위로 특정 열에서 하나의 요소를 뽑아내기 위해, `sample()`를 동일하게 사용한다.

```
t <- c(1,2,3,4,5)
sample(t, 1)
```

```
## [1] 5
```

```
sample(t, 1)
```

```
## [1] 5
```

또한 runif() 모듈을 활용하여 정규분포, 지수분포, 감마분포 및 기타 연속형 분포에서 난수를 생성하는 함수도 제공된다.

## 5.5 수학 함수

R은 가장 친숙한 수학 함수를 제공하는 수학 모듈이 있다. 기본 수학 모듈이 내장함수로 기본 설치되어 있어서 별도 호출작업은 필요없다.

**점 표기법(dot notation)**이라고 불리는 표기법을 사용해서, 함수 중에서 한 함수에 접근하기 위해서 모듈/객체 이름과 함수 이름을 명시해서 파이썬에서 활용하기도

하지만, R에서는 필요없다.

```
signal_power <- 9
noise_power <- 10
ratio <- signal_power / noise_power
decibels <- 10 * log10(ratio)

radians <- 0.7
height <- sin(radians)
```

첫 예제는 신호-대-소음비의 로그 밑이 10(`log10()`) 을 계산한다.

두 번째 예제는 라디안의 사인값을 찾는 것이다. 변수의 이름이 힌트를 주는데, `sin()`과 다른 삼각함수(`cos()`, `tan()` 등)는 라디안을 인자로 받는다. 도(degree)에서 라디안(radian)으로 변환하기 위해서 360으로 나누고 $2\pi$를 곱한다.

```
degrees <- 45
radians <- degrees / 360.0 * 2 * pi
sin(radians)
```

```
## [1] 0.707
```

`pi` 표현식은 수학 모듈에서 `pi` 변수를 얻는데, $\pi$ 값과 비교하여 15 자리수까지 정확하고 근사적으로 수렴한다.

삼각함수를 배웠다면, 앞선 연산 결과를 2에 루트를 씌우고 2로 나누어 비교 검증한다.

```
sqrt(2) / 2
```

```
## [1] 0.707
```

## 5.6 신규 함수 추가

지금까지 R 설치 시 함께 설치되는 함수만 사용했지만 새로운 함수를 추가하는 것도 가능하다. **함수 정의(function definition)**는 신규 함수명과 함수가 호출될 때 실행될 일련의 문장을 명세한다. 신규로 함수를 정의하면, 프로그램 실행 중에 반복해서 함수를 재사용할 수 있다.

다음에 예제가 있다.

```
print_lyrics <- function() {
  print("I'm a lumberjack, and I'm okay.")
  print("I sleep all night and I work all day.")
}
```

`function`는 "이것이 함수 정의다"를 표시하는 예약어다. 함수 이름은 `print_lyrics()`이다. 함수 이름을 명명 규칙은 변수명과 동일하다. 문자, 숫자, 그리고 몇몇 문장 부호는

사용할 수 있지만, 첫 문자가 숫자는 될 수 없다. 함수 이름으로 예약어를 사용할 수 없고, 함수 이름과 동일한 변수명은 피해야 한다.

함수명 뒤 빈 괄호는 이 함수가 어떠한 인자도 갖지 않는다는 것을 나타낸다. 나중에, 입력값으로 인자를 가지는 함수를 작성해 볼 것이다.

함수 정의 첫번째 줄을 머리 부문(헤더, header), 나머지 부문을 몸통 부문(바디, body)라고 부른다. 머리 부문은 `()`으로 끝나고, 몸통 부문은 괄호로 감싸야 한다. 몸통 부문에는 제약 없이 문장을 작성할 수 있다.

`print()`문의 문자열은 이중 인용부호로 감싼다. 단일 인용부호나, 이중 인용부호나 차이는 없다. 대부분의 경우 이중 인용부호를 사용하고, 이중 인용부호가 문자열에 나타나는 경우, 단일 인용부호를 사용하여 이중 인용부호가 출력되게 감싼다.

만약 함수 정의를 인터렉티브 모드에서 타이핑을 하면, 함수 정의가 끝나지 않았다는 것을 의미로 더하기 부호(+)가 출력된다.

함수 정의를 끝내기 위해서 빈 줄을 입력한다. (스크립트에서는 반듯이 필요한 것은 아니다.) 함수를 정의하게 되면 동일한 이름의 변수도 생성된다.

```
print_lyrics()
```

```
## [1] "I'm a lumberjack, and I'm okay."
## [1] "I sleep all night and I work all day."
```

```
class(print_lyrics)
```

```
## [1] "function"
```

`print_lyrics()` 값은 'function' 형을 가지는 **함수 객체(function object)**다.

신규 함수를 호출하는 구문은 내장 함수의 경우와 동일하다.

```
print_lyrics()
```

```
## [1] "I'm a lumberjack, and I'm okay."
## [1] "I sleep all night and I work all day."
```

함수를 정의하면, 또 다른 함수 내부에서 사용이 가능하다. 예를 들어, 이전 후렴구를 반복하기 위해 `repeat_lyrics()` 함수를 작성할 수 있다.

```
repeat_lyrics <- function() {
  print_lyrics()
  print_lyrics()
}
```

그리고 나서, `repeat_lyrics()` 함수를 호출한다.

```
repeat_lyrics()
```

```
## [1] "I'm a lumberjack, and I'm okay."
## [1] "I sleep all night and I work all day."
## [1] "I'm a lumberjack, and I'm okay."
```

```
## [1] "I sleep all night and I work all day."
```

하지만, 이것이 실제 노래가 불려지는 것은 아니다.

## 5.7 함수 정의와 사용법

앞 절의 코드 조각을 모아서 작성한 전체 프로그램은 다음과 같다.

```
print_lyrics <- function() {
  print("I'm a lumberjack, and I'm okay.")
  print("I sleep all night and I work all day.")
}

repeat_lyrics <- function() {
  print_lyrics()
  print_lyrics()
}

repeat_lyrics()
```

상기 프로그램에는 두개의 함수(`print_lyrics()`, `repeat_lyrics()`)가 있다. 함수 정의는 다른 문장처럼 수행되지만, 함수 객체를 생성한다는 점에서 차이가 있다. 함수 내부 문장은 함수가 호출되기 전까지 수행되지 않고, 함수 정의는 출력값도 생성하지 않는다.

예상하듯이, 함수를 실행하기 전에 함수를 생성해야 한다. 다시 말해서, 처음으로 호출되기 전에 함수 정의가 실행되어야 한다.

**연습문제** 상기 프로그램의 마지막 줄을 최상단으로 옮겨서 함수 정의 전에 호출되도록 프로그램을 고쳐보세요. 프로그램을 실행서 오류 메시지를 확인하세요.

**연습문제** 함수 호출을 맨 마지막으로 옮기고, repeat_lyrics 함수 정의 뒤에 print_lyrics 함수를 옮기세요. 프로그램을 실행하게 되면 무슨 일이 발생하나요?

## 5.8 실행 흐름

처음으로 함수가 사용되기 전에 정의되었는지를 확인하기 위해서, 명령문 실행 순서를 파악해야 하는데 이를 **실행 흐름(flow of execution)**이라고 한다.

프로그램 실행은 항상 프로그램 첫 문장부터 시작한다. 명령문은 한번에 하나씩 위에서 아래로 실행된다.

함수 정의(definitions)가 프로그램 실행 순서를 바꾸지는 않는다. 하지만, 함수 내부의 문장은 함수가 호출될 때까지 실행이 되지 않는다는 것을 기억하자.

함수 호출은 프로그램 실행 흐름을 우회하는 것과 같다. 다음 문장으로 가기 전에, 실행 흐름은 함수 몸통 부문을 실행하고는 건너 뛰기를 시작한 지점으로 다시 돌아온다.

함수가 또 다른 함수를 호출한다는 것을 기억할 때까지는 매우 간단하게 들린다. 함수 중간에서 프로그램이 또 다른 함수의 문장을 수행할지도 모른다. 하지만, 새로운 함수를 실행하는 중간에 프로그램이 또 다른 함수를 실행할지도 모른다!

다행스럽게도, 파이썬은 프로그램 실행 위치를 정확히 추적한다. 그래서, 함수가 실행을 완료할 때마다, 프로그램을 함수를 호출해서 떠난 지점으로 정확히 되돌려 놓는다. 프로그램이 마지막에 도달했을 때, 프로그램은 종료한다.

이렇게 복잡한 이야기의 교훈은 무엇일까요? 프로그램을 읽을 때, 위에서부터 아래로 읽을 필요는 없다. 때때로, 실행 흐름을 따르는 것이 좀더 이치에 맞는다.

## 5.9 매개 변수와 인수

지금까지 살펴본 몇몇 내장 함수는 **인자(argument)**를 요구한다. 예를 들어, `sin()` 함수를 호출할 때, 숫자를 인자로 넘겨야 한다. 어떤 함수는 2개 이상의 인수를 받는다. `log()` 는 숫자와 밑 2개의 인자가 필요하다.

인자는 함수 내부에서 **매개 변수(parameters)**로 불리는 변수로 대입된다. 하나의 인자를 받는 **사용자 정의 함수(user-defined function)**가 예제로 있다.

```
print_twice <- function(bruce){
  cat(bruce, "\n")
  cat(bruce, "\n")
}
```

사용자 정의 함수는 인자를 받아 매개변수 bruce에 대입한다. 함수가 호출될 때, 매개변수의 값(무엇이든 관계 없이)을 두번 출력합니다.

사용자 정의 함수는 출력 가능한 임의의 값에 작동한다.

```
print_twice('Spam')
```

```
## Spam
## Spam
```

```
print_twice(17)
```

```
## 17
## 17
```

```
print_twice(pi)
```

```
## 3.14
## 3.14
```

내장함수에 적용되는 동일한 구성 규칙이 사용자 정의 함수에도 적용되어서, `print_twice()` 함수 인자로 표현식 어떤 종류도 가능하다.

```
print_twice(rep("spam",2))
```

```
## spam spam
```

```
## spam spam
```

```
print_twice(cos(pi))
```

```
## -1
## -1
```

함수가 호출되기 전에 인자에 대한 평가는 완료되어, 예제에서 `rep("spam",2)`과 `cos(pi)`은 단지 1회만 평가된다.

변수도 인자로 사용이 가능하다.

```
michael <- "Eric, the half a bee."
print_twice(michael)
```

```
## Eric, the half a bee.
## Eric, the half a bee.
```

인수자 넘기는 변수명(michael)은 매개 변수명(bruce)과 아무런 연관이 없다. 무슨 값이 호출된든지 호출하는 측과 상관이 없다. 여기 `print_twice()` 함수에서는 누구나 bruce라고 부르면 된다.

## 5.10 결과있는 함수와 빈 함수

수학 함수와 같은 몇몇 함수는 결과를 만들어 낸다. 좀더 좋은 이름이 없어서, 결과를 만들어 내는 함수를 **결과있는 함수(fruitful functions)**라고 명명한다. `print_twice()`와 같이 액션을 수행하지만, 결과를 만들어 내지 않는 함수를 **빈 함수(void functions)**라고 부른다.

결과있는 함수를 호출할 때는 결과값을 가지고 뭔가를 하려고 한다. 예를 들어, 결과값을 변수에 대입하거나, 표현식의 일부로 사용할 수 있다.

```
x <- cos(radians)
golden <- (sqrt(5) + 1) / 2
```

인터랙티브 모드에서 함수를 호출할 때, R은 결과를 화면에 출력한다.

```
sqrt(5)
```

```
## [1] 2.24
```

하지만, 스크립트에서 결과있는 함수를 호출하고 변수에 결과값을 저장하지 않으면 반환되는 결과값은 안개속에 사라져간다!

이 스크립트는 5의 제곱근을 계산하지만, 변수에 결과값을 저장하거나, 화면에 출력하지 않아서 그다지 유용하지는 않다.

빈 함수(Void functions)는 화면에 출력하거나 무엇인가 다른 효과를 가지지만, 반환값이 없다. 빈 함수를 사용하여 결과에 변수를 대입하면, `NULL`로 불리는 특별한 값을 얻게 된다.

```
result <- print_twice('Bing')
Bing
Bing

print(result)
NULL
```

NULL 값은 자신만의 특별한 값을 가지며, 문자열 'NULL' 과는 같지 않다.

함수에서 결과를 반환하기 위해서, 함수 내부에 return()문을 사용한다. 예를 들어, 두 숫자를 더해서 결과를 반환하는 addtwo()라는 간단한 함수를 작성할 수 있다.

```
addtwo <- function(a, b){
  added <- a + b
  return(added)
}

x <- addtwo(3, 5)
cat(x)
```

```
## 8
```

상기 스크립트가 실행될 때 cat() 출력문은 "8"을 출력한다. 왜냐하면, 3과 5를 인수로 받는 addtwo() 함수가 호출되기 때문이다. 함수 내부에 매개 변수 a, b는 각각 3, 5이다. addtwo() 함수는 두 숫자 덧셈을 수행하고 added라는 로컬 변수에 저장하고, return()문을 사용해서 덧셈 결과를 반환하고,x 라는 변수에 대입해서 출력한다.

R에서 명시적으로 return을 통해 밝히지 않더라도 함수에서 최종적으로 담고 있는 객체가 자동 반환되지만, return을 통해 명시적으로 하는 것이 추후 디버깅 등의 목적으로 더 유용하다.

## 5.11 함수 사용 이유?

프로그램을 함수로 나누는 고생을 할 가치가 있는지 명확하지 않을 수 있다. 다음에 여기 몇 가지 이유가 있다.

- 문장을 그룹으로 만들어 새로운 함수로 명명하는 것이 프로그램을 읽고, 이해하고, 디버그하기 좋게한다.
- 함수는 반복 코드를 제거해서 프로그램을 작고 콤팩트하게 만든다. 나중에 프로그램에 수정사항이 생기면, 단지 한 곳에서만 수정을 하면 된다.
- 긴 프로그램을 함수로 나누어 작성하는 것은 작은 부분에서 버그를 수정할 수 있게 하고, 이를 조합해서 전체적으로 동작하는 프로그램을 만들 수 있다.
- 잘 설계된 함수는 종종 많은 프로그램에서 유용하게 사용된다. 잘 설계된 프로그램을 작성하고 디버그를 해서 오류가 없이 만들게 되면, 나중에 재사용도 용이하다.

책의 나머지 부분에서 이 개념을 설명하는 함수 정의를 종종 사용한다. “리스트에서 가장 작은 값을 찾아내는 것”과 같이 아이디어를 적절하게 추상화하여 함수를 작성하는 것이 함수를 만들고 사용하는 기술의 일부가 된다. 나중에, 리스트에서 가장 작은 값을 찾아내는 코드를 보여 줄 것입니다. 리스트를 인수로 받아 가장 작은 값을 반환하는 `min()` 함수를 작성해서 여러분에게 보여드릴 것이다.

## 5.12 디버깅

텍스트 편집기로 스크립트를 작성한다면 공백과 탭으로 몇번씩 문제에 봉착했을 것입니다. 이런 문제를 피하는 가장 최선의 방식은 절대 탭을 사용하지 말고 공백(스페이스)를 사용하는 것이다. R을 인식하는 대부분의 텍스트 편집기는 디폴트로 이런 기능을 지원하지만, 몇몇 텍스트 편집기는 이런 기능을 지원하지 않아 탭과 공백 문제를 야기한다.

탭과 공백은 통상 눈에 보이지 않기 때문에 디버그를 어렵게 한다. 자동으로 들여쓰기를 해주는 편집기를 프로그램 작성 시 사용한다.

프로그램을 실행하기 전에 저장하는 것을 잊지 마세요. 몇몇 개발 환경은 자동저장 기능을 지원하지만 그렇지 않는 것도 있다. 이런 이유 때문에 텍스트 편집기에서 작성한 개발 프로그램과 실행운영하고 있는 프로그램이 같지 않을 수도 있다.

동일하고 잘못된 프로그램을 반복적으로 실행한다면, 디버깅은 오래 걸릴 수 있다.

작성하고 있는 코드와 실행하는 코드가 일치하는지 필히 확인하자. 확신을 하지 못한다면, 프로그램의 첫줄에 `print('hello')` 을 넣어서 실행해 보자. hello를 보지 못한다면, 작성하고 있는 프로그램과 실행하고 있는 프로그램은 다른 것이다.

## 5.13 용어정의 {r-func-terminology}

- **알고리즘(algorithm)**: 특정 범주의 문제를 해결하는 일반적인 프로세스
- **인자(argument)**: 함수가 호출될 때 함수에 제공되는 값. 이 값은 함수 내부에 상응하는 매개 변수에 대입된다.
- **몸통 부문(body)**: 함수 정의 내부에 일련의 문장
- **구성(composition)**: 좀더 큰 표현식의 일부분으로 표현식을 사용하거나, 좀더 큰 문장의 일부로서의 문장
- **결정론적(deterministic)**: 동일한 입력값이 주어지고 실행될 때마다 동일한 행동을 하는 프로그램에 관련된 것.
- **점 표기법(dot notation)**: 점과 함수명으로 모듈명을 명세함으로써 다른 모듈의 함수를 호출하는 구문.
- **실행 흐름(flow of execution)**: 프로그램 실행 동안 명령문이 실행되는 순서.
- **결과있는 함수(fruitful function)**: 반환값을 가지는 함수.
- **함수(function)**: 유용한 연산을 수행하는 이름을 가진 일련의 명령문. 함수는 인수를 가질 수도 갖지 않을 수도 있고, 결과값을 생성할 수도 생성하지 않을 수도 있다.
- **함수 호출(function call)**: 함수를 실행하는 명령문. 함수 이름과 인자 리스트로 구성된다.

- **함수 정의(function definition)**: 신규 함수를 정의하는 명령문으로 이름, 매개변수, 실행 명령문을 명세한다.
- **함수 객체(function object)**: 함수 정의로 생성되는 값. 함수명은 함수 객체를 참조하는 변수다.
- **머리 부문(header)**: 함수 정의의 첫번째 줄
- **가져오기 문(import statement)**: 모듈 파일을 읽어 모듈 개체를 생성하는 명령문
- **모듈 개체(module object)**: import문에 의해서 생성된 모듈에 정의된 코드와 데이터에 접근할 수 있는 값
- **매개 변수(parameter)**: 인자로 전달된 값을 참조하기 위해 함수 내부에 사용되는 이름
- **의사 난수(pseudorandom)**: 난수처럼 보이는 일련의 숫자와 관련되어 있지만, 결정론적 프로그램에 의해 생성된다.
- **반환 값(return value)**: 함수의 결과. 함수 호출이 표현식으로 사용된다면, 반환값은 표현식의 값이 된다.
- **빈 함수(void function)**: 반환값을 갖지 않는 함수

## 5.14 연습문제 {r-func-ex}

1. R “function” 키워드의 목적은 무엇입니까?
    1. “다음의 코드는 정말 좋다”라는 의미를 가진 속어
    2. 함수 정의를 표현한다.
    3. 다음의 들여쓰기 코드 부문은 나중을 위해 저장되야 된다는 것을 표시한다.
    4. 2와 3 모두 사실
    5. 위 모두 거짓
2. 다음 R 프로그램은 무엇을 출력할까요?

```
fred <- function(){
  print("Zap")
}

jane <- function(){
  print("ABC")
}

jane()
fred()
jane()
```

a) Zap ABC jane fred jane

b) Zap ABC Zap

c) ABC Zap jane

d) ABC Zap ABC

e) Zap Zap Zap

3. 프로그램 작성 시 (hours와 rate)을 매개 변수로 갖는 함수 computepay()을 생성하여, 초과근무에 대해서는 50% 초과 근무수당을 지급하는 봉급 계산 프로그램을 다시 작성하세요.

```
     : 45
     : 10
 : 475
```

4. 매개 변수로 점수를 받아 문자열로 등급을 반환하는 computegrade() 함수를 사용하여 앞장의 등급 프로그램을 다시 작성하세요.

```
>= 0.9   A
>= 0.8   B
>= 0.7   C
>= 0.6   D
< 0.6    F

     : 0.95
A

     :
       .

     : 10.0
       .

     : 0.75
C

     : 0.5
F
```

반복적으로 프로그램을 실행해서 다양한 다른 입력값을 테스트해 보세요.

# Chapter 6

# 반복(Iteration)

## 6.1 변수 갱신

대입문의 흔한 패턴은 변수를 갱신하는 대입문이다. 변수의 새로운 값은 이전 값에 의존한다.

```
x <- x + 1
```

상기 예제는 "현재 값 x에 1을 더해서 x를 새로운 값으로 갱신한다."

만약 존재하지 않는 변수를 갱신하면, 오류가 발생한다. 왜냐하면 x에 값을 대입하기 전에 R이 오른쪽을 먼저 평가하기 때문이다.

```
x <- x + 1
Error: object 'x' not found
```

변수를 갱신하기 전에 간단한 변수 대입으로 통상 먼저 초기화(initialize)한다.

```
x <- 0
x <- x + 1
```

1을 더해서 변수를 갱신하는 것을 증가(increment)라고 하고, 1을 빼서 변수를 갱신하는 것을 감소(decrement)라고 한다.

## 6.2 while문

종종 반복적인 작업을 자동화하기 위해서 컴퓨터를 사용한다. 오류 없이 동일하거나 비슷한 작업을 반복하는 일은 컴퓨터가 사람보다 잘한다. 반복이 매우 흔한 일이어서, R에서 반복 작업을 쉽게 하도록 몇가지 언어적 기능을 제공한다.

R에서 반복의 한 형태가 `while`문이다. 다음은 5 에서부터 거꾸로 세어서 마지막에 "Blastoff(발사)!"를 출력하는 간단한 프로그램이다.

```
n <- 5
while(n > 0) {
  print(n)
  n <- n -1
}
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
```

```
print("Blastoff( )!")
```

```
## [1] "Blastoff( )!"
```

마치 영어를 읽듯이 `while`을 읽어 내려갈 수 있다. `n`이 0 보다 큰 동안에 `n`의 값을 출력하고 `n` 값에서 1만큼 뺀다. 0 에 도달했을 때, `while`문을 빠져나가 Blastoff(발사)!"를 화면에 출력한다.

좀더 형식을 갖춰 정리하면, 다음이 `while`문에 대한 실행 흐름에 대한 정리다.

1. 조건을 평가해서 **참(TRUE)** 혹은 **거짓(FALSE)**를 산출한다.
2. 만약 조건이 거짓이면, `while`문을 빠져나가 다음 문장을 계속 실행한다.
3. 만약 조건이 참이면, 몸통 부문의 문장을 실행하고 다시 처음 1번 단계로 돌아간다.

3번째 단계에서 처음으로 다시 돌아가는 반복을 하기 때문에 이런 종류의 흐름을 **루프(loop)**라고 부른다. 매번 루프 몸통 부문을 실행할 때마다, 이것을 **반복(iteration)**이라고 한다. 상기 루프에 대해서 "5번 반복했다"고 말한다. 즉, 루프 몸통 부문이 5번 수행되었다는 의미가 된다.

루프 몸통 부문은 필히 하나 혹은 그 이상의 변수값을 바꾸어서 종국에는 조건식이 **거짓(FALSE)**이 되어 루프가 종료되게 만들어야 한다. 매번 루프가 실행될 때마다 상태를 변경하고 언제 루프가 끝날지 제어하는 변수를 **반복 변수(iteration variable)**라고 한다. 만약 반복 변수가 없다면, 루프는 영원히 반복될 것이고, 결국 **무한 루프(infinite loop)**에 빠질 것이다.

## 6.3 무한 루프

프로그래머에게 무한한 즐거움의 원천은 아마도 "거품내고, 헹구고, 반복" 이렇게 적혀있는 샴프 사용법 문구가 무한루프라는 것을 알아차릴 때일 것이다. 왜냐하면, 얼마나 많이 루프를 실행해야 하는지 말해주는 반복 변수(iteration variable)가 없어서 무한 반복하기 때문입니다.

숫자를 꺼꾸로 세는 (countdown) 예제는 루프가 끝나는 것을 증명할 수 있다. 왜냐하면 `n`값이 유한하고, `n`이 매번 루프를 돌 때마다 작아져서 결국 0 에 도달할

것이기 때문이다. 다른 경우 반복 변수가 전혀 없어서 루프가 명백하게 무한 반복한다.

## 6.4 무한 반복과 break

가끔 몸통 부문을 절반 진행할 때까지 루프를 종료해야하는 시점인지 확신하지 못한다. 이런 경우 의도적으로 무한 루프를 작성하고 break 문을 사용하여 루프를 빠져 나온다.

다음 루프는 명백하게 무한 루프(infinite loop)가 되는데 이유는 while문 논리 표현식이 단순히 논리 상수 참(TRUE)으로 되어 있기 때문이다.

```
n <- 10

while(TRUE) {
  print(n)
  n <- n - 1
}

print('Done!')
```

실수하여 상기 프로그램을 실행한다면, 폭주하는 R 프로세스를 어떻게 멈추는지 빨리 배우거나, 컴퓨터의 전원 버튼이 어디에 있는지 찾아야 할 것이다. 표현식 상수 값이 참(TRUE)이라는 사실로 루프 상단 논리 연산식이 항상 참 값이여서 프로그램이 영원히 혹은 배터리가 모두 소진될 때까지 실행된다.

이것이 역기능 무한 루프라는 것은 사실이지만, 유용한 루프를 작성하기 위해는 이 패턴을 여전히 이용할 것이다. 이를 위해서 루프 몸통 부문에 break문을 사용하여 루프를 빠져나가는 조건에 도달했을 때, 루프를 명시적으로 빠져나갈 수 있도록 주의깊게 코드를 추가해야 한다.

예를 들어, 사용자가 done을 입력하기 전까지 사용자로부터 입력값을 받는다고 가정해서 프로그램 코드를 다음과 같이 작성한다.

```
while(TRUE) {
  line <- readline(prompt = '> ')
  if(line == 'done') {
    break
  }
  print(line)
}
```

루프 조건이 항상 참(TRUE)이여서 break문이 호출될 때까지 루프는 반복적으로 실행된다.

매번 프로그램이 꺾쇠 괄호로 사용자에게 명령문을 받을 준비를 한다. 사용자가 done을 타이핑하면, break문이 실행되어 루프를 빠져나온다. 그렇지 않은 경우

프로그램은 사용자가 무엇을 입력하든 메아리처럼 입력한 것을 그대로 출력하고 다시 루프 처음으로 되돌아 간다. 다음 예제로 실행한 결과가 있다.

```
> hello there
hello there
> finished
finished
> done
> done
Error: object 'done' not found
```

while 루프를 이와 같은 방식으로 작성하는 것이 흔한데 프로그램 상단에서 뿐만 아니라 루프 어디에서나 조건을 확인할 수 있고 피동적으로 "이벤트가 발생할 때까지 계속 실행" 대신에, 적극적으로 "이벤트가 생겼을 때 중지"로 멈춤 조건을 표현할 수 있다.

## 6.5 next로 반복 종료

때때로 루프를 반복하는 중간에서 현재 반복을 끝내고, 다음 반복으로 즉시 점프하여 이동하고 싶을 때가 있다. 현재 반복 루프 몸통 부분 전체를 끝내지 않고 다음 반복으로 건너뛰기 위해서 next문을 사용한다.

사용자가 "done"을 입력할 때까지 입력값을 그대로 복사하여 출력하는 루프 예제가 있다. 하지만 R 주석문처럼 해쉬(#)로 시작하는 줄은 출력하지 않느다.

```
while(TRUE) {
  line <- readline(prompt = '> ')
  if(substr(line,1,1) == "#") {
    next
  }
  if(line == 'done') {
    break
  }
  print(line)
}
```

next문이 추가된 새로운 프로그램을 샘플로 실행했다.

```
> hello there
[1] hello there
> # don't print this
> print this!
[2] print this!
> done
```

해쉬 기호(#)로 시작하는 줄을 제외하고 모든 줄을 출력한다. 왜냐하면, next문이 실행될 때,현재 반복을 종료하고 while문 처음으로 돌아가서 다음 반복을 실행하게 되어서 print문을 건너뛴다.

## 6.6 for문 사용 명확한 루프

때때로, 단어 리스트나, 파일의 줄, 숫자 리스트 같은 사물의 집합에 대해 루프를 반복할 때가 있다. 루프를 반복할 사물 리스트가 있을 때, for문을 사용해서 **확정 루프(definite loop)**를 구성한다.

while문을 **불확정 루프(indefinite loop)**라고 하는데, 왜냐하면 어떤 조건이 거짓(FALSE)가 될 때까지 루프가 단순히 계혹해서 돌기 때문이다. 하지만, for루프는 확정된 항목의 집합에 대해서 루프가 돌게 되어서 집합에 있는 항목만큼만 실행이 된다.

for문이 있고, 루프 몸통 부문으로 구성된다는 점에서 for루프 구문은 while루프 구문과 비슷하다.

```
friends <- c('Joseph', 'Glenn', 'Sally')

for(friend in friends) {
  cat('Happy New Year:', friend, "\n")
}
```

R 용어로, 변수 friends는 3개의 문자열을 가지는 벡터이며, for 루프는 벡터내 원소를 하나씩 하나씩 찾아서 벡터에 있는 3개 문자열 각각에 대해 출력을 실행하여 다음 결과를 얻게 된다.

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
```

for 루프를 영어로 번역하는 것이 while문을 번역하는 것과 같이 직접적이지는 않다. 하지만, 만약 friends를 집합(set)으로 생각한다면 다음과 같다. friends라고 명명된 집합에서 friend 각각에 대해서 한번씩 for 루프 몸통 부문에 있는 문장을 실행하라.

for 루프를 살펴보면, for와 in은 R 예약어이고 friend와 friends는 변수이다.

```
for(friend in friends) {
  cat('Happy New Year:', friend, "\n")
}
```

특히, friend는 for 루프의 **반복 변수(iteration variable)**다. 변수 friend는 루프가 매번 반복할 때마다 변하고, 언제 for 루프가 완료되는지 제어한다. 반복 변수는 friends 변수에 저장된 3개 문자열을 순차적으로 훑고 간다.

## 6.7 루프 패턴

종종 for문과 while문을 사용하여, 벡터나 리스트 항목, 파일 콘텐츠를 훑어 자료에 있는 가장 큰 값이나 작은 값 같은 것을 찾는다.

for나 while 루프는 일반적으로 다음과 같이 구축된다.

1. 루프가 시작하기 전에 하나 혹은 그 이상의 변수를 초기화
2. 루프 몸통부분에 각 항목에 대해 연산을 수행하고, 루프 몸통 부분의 변수 상태를 변경
3. 루프가 완료되면 결과 변수의 상태 확인

루프 패턴의 개념과 작성을 시연하기 위해서 숫자 벡터를 사용한다.

### 6.7.1 계수와 합산 루프

예를 들어, 벡터의 항목을 세기(counting) 위해서 다음과 같이 `for` 루프를 작성한다.

```
count <- 0
for(itervar in c(3, 41, 12, 9, 74, 15)){
  count <- count + 1
}

cat('Count: ', count)
```

```
## Count:  6
```

루프가 시작하기 전에 변수 `count`를 0 으로 설정하고, 숫자 목록을 훑어 갈 `for` 루프를 작성한다. 반복(iteration) 변수는 `itervar`라고 하고, 루프에서 `itervar`을 사용되지 않지만, `itervar`는 루프를 제어하고 루프 몸통 부문 리스트의 각 값에 대해서 한번만 실행되게 한다.

루프 몸통 부문에 리스트의 각 값에 대해서 변수 `count` 값에 1을 더한다. 루프가 실행될 때, `count` 값은 "지금까지" 살펴본 값의 횟수가 된다.

루프가 종료되면, `count` 값은 총 항목 숫자가 된다. 총 숫자는 루프 맨마지막에 얻어진다. 루프를 구성해서, 루프가 끝났을 때 기대했던 바를 얻었다.

숫자 집합의 갯수를 세는 또 다른 비슷한 루프는 다음과 같다.

```
total <- 0
for(itervar in c(3, 41, 12, 9, 74, 15)){
  total <- total + itervar
}

cat('Total: ', total)
```

```
## Total:  154
```

상기 루프에서, 반복 변수(iteration variable)가 사용되었다. 앞선 루프에서처럼 변수 `count`에 1을 단순히 더하는 대신에, 각 루프가 반복을 수행하는 동안 실제 숫자 (3, 41, 12, 등)를 작업중인 합계에 덧셈을 했다. 변수 `total`을 생각해보면, total은 "지금까지 값의 총계다." 루프가 시작하기 전에 `total`은 어떤 값도 살펴본 적이 없어서 0 이다. 루프가 도는 중에는 `total`은 작업중인 총계가 된다. 루프의 마지막 단계에서 `total`은 리스트에 있는 모든 값의 총계가 된다.

루프가 실행됨에 따라, `total`은 각 요소의 합계로 누적된다. 이 방식으로 사용되는 변수를 **누산기(accumulator)**라고 한다.

계수(counting) 루프나 합산 루프는 특히 실무에서 유용하지는 않다. 왜냐하면 리스트에서 항목의 개수와 총계를 계산하는 `length()`과 `sum()`가 각각 내장 함수로 있기 때문이다.

### 6.7.2 최대값과 최소값 루프

리스트, 벡터나 열(sequence)에서 가장 큰 값을 찾기 위해서, 다음과 같이 루프를 작성한다.

```
largest <- NA

cat('Before:', largest, "\n")

for(itervar in c(3, 41, 12, 9, 74, 15)){
  if(is.na(largest) || itervar > largest){
    largest <- itervar
    cat('Loop:', itervar, largest, "\n")
    cat('Largest:', largest, "\n")
  }
}
```

프로그램을 실행하면, 출력은 다음과 같다.

```
Loop: 3 3
Largest: 3
Loop: 41 41
Largest: 41
Loop: 74 74
Largest: 74
```

변수 `largest`는 "지금까지 본 가장 큰 수"로 생각할 수 있다. 루프 시작 전에 `largest` 값은 상수 `NA`이다. `NA`은 "빈(empty)" 변수를 표기하기 위해서 변수에 저장하는 특별한 상수 값이다.

루프 시작 전에 지금까지 본 가장 큰 수는 `NA`이다. 왜냐하면 아직 어떤 값도 보지 않았기 때문이다. 루프가 실행되는 동안에, `largest` 값이 `NA` 이면, 첫 번째 본 값이 지금까지 본 가장 큰 값이 된다. 첫번째 반복에서 `itervar`는 3 이 되는데 `largest` 값이 `NA`이여서 즉시, `largest`값을 3 으로 갱신한다.

첫번째 반복 후에 `largest`는 더 이상 `NA`가 아니다. `itervar > largest`인지를 확인하는 복합 논리 표현식의 두 번째 부분은 "지금까지 본" 값 보다 더 큰 값을 찾게 될 때 자동으로 동작한다. "심지어 더 큰" 값을 찾게 되면 변수 `largest`에 새로운 값으로 대체한다. `largest`가 3에서 41, 41에서 74로 변경되어 출력되어 나가는 것을 확인할 수 있다.

루프의 끝에서 모든 값을 훑어서 변수 `largest`는 리스트의 가장 큰 값을 담고 있다.

최소값을 계산하기 위해서는 코드가 매우 유사하지만 작은 변화가 있다.

```
smallest <- NA

cat('Before:', smallest, "\n")

for(itervar in c(3, 41, 12, 9, 74, 15)){
  if(is.na(smallest) || itervar < smallest){
    smallest <- itervar
    cat('Loop:', itervar, smallest, "\n")
    cat('Largest:', smallest, "\n")
  }
}
```

변수 smallest는 루프 실행 전에, 중에, 완료 후에 "지금까지 본 가장 작은" 값이 된다. 루프 실행이 완료되면, smallest는 벡터의 최소값을 담게 된다.

계수(counting)과 합산에서와 마찬가지로 R 내장함수 max()와 min()은 이런 루프문 작성을 불필요하게 만든다.

다음은 R 내장 min() 함수의 간략 버전이다. getAnywhere(min), .Primitive("min")을 입력해도 원소스코드를 볼 수는 없다. names(methods:::.BasicFunsList) 명령어를 통해 .Primitive() 함수를 파악할 수 있다.

```
min <- function(values) {
  smallest <- NA
  for(value in values){
    if(is.na(smallest) || value < smallest){
      smallest <- value
  return(smallest)
  }
}
```

가장 적은 코드로 작성한 함수 버전은 R에 이미 내장된 min 함수와 동등하게 만들기 위해서 모든 print문을 삭제했다.

## 6.8 디버깅

좀더 큰 프로그램을 작성할 때, 좀더 많은 시간을 디버깅에 보내는 자신을 발견할 것이다. 좀더 많은 코드는 버그가 숨을 수 있는 좀더 많은 장소와 오류가 발생할 기회가 있다는 것을 의미한다.

디버깅 시간을 줄이는 한 방법은 **"이분법에 따라 디버깅(debugging by bisection)"** 하는 것이다. 예를 들어, 프로그램에 100 줄이 있고 한번에 하나씩 확인한다면, 100 번 단계가 필요하다.

대신에 문제를 반으로 나눈다. 프로그램 정확히 중간이나, 중간부분에서 점검한다. print문이나, 검증 효과를 갖는 상응하는 대용물을 넣고 프로그램을 실행한다.

중간지점 점검 결과 잘못 되었다면 문제는 양분한 프로그램 앞부분에 틀림없이 있다. 만약 정확하다면, 문제는 프로그램 뒷부분에 있다.

이와 같은 방식으로 점검하게 되면, 검토 해야하는 코드의 줄수를 절반으로 계속 줄일 수 있다. 단계가 100 번 걸리는 것에 비해 6번 단계 후에 이론적으로 1 혹은 2 줄로 문제 코드의 범위를 좁힐 수 있다.

실무에서, "프로그램의 중간"이 무엇인지는 명확하지 않고, 확인하는 것도 가능하지 않다. 프로그램 코드 라인을 세서 정확히 가운데를 찾는 것은 의미가 없다. 대신에 프로그램 오류가 생길 수 있는 곳과 오류를 확인하기 쉬운 장소를 생각하세요. 점검 지점 앞뒤로 버그가 있을 곳과 동일하게 생각하는 곳을 중간지점으로 고르세요.

## 6.9 용어정의

- **누산기(accumulator)**: 더하거나 결과를 누적하기 위해 루프에서 사용되는 변수
- **계수(counter)**: 루프에서 어떤 것이 일어나는 횟수를 기록하는데 사용되는 변수. 카운터를 0 으로 초기화하고, 어떤 것의 "횟수"를 셀 때 카운터를 증가시킨다.
- **감소(decrement)**: 변수 값을 감소하여 갱신
- **초기화(initialize)**: 갱신될 변수의 값을 초기 값으로 대입
- **증가(increment)**: 변수 값을 증가시켜 갱신 (통상 1씩)
- **무한 루프(infinite loop)**: 종료 조건이 결코 만족되지 않거나 종료 조건이 없는 루프
- **반복(iteration)**: 재귀함수 호출이나 루프를 사용하여 명령문을 반복 실행

## 6.10 연습문제

1. 사용자가 "done"을 입력할 때까지 반복적으로 숫자를 읽는 프로그램을 작성하세요. "done"이 입력되면, 총계, 갯수, 평균을 출력하세요. 만약 숫자가 아닌 다른 것을 입력하게되면, `tryCatch`를 사용하여 사용자 실수를 탐지해서 오류 메시지를 출력하고 다음 숫자로 건너 뛰게 하세요.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.33333333333
```

2. 위에서처럼 숫자 목록을 사용자로부터 입력받는 프로그램을 작성하세요. 평균값 대신에 숫자 목록 최대값과 최소값을 출력하세요.

# 자료구조

# Chapter 7

# 문자열

## 7.1 문자열은 순열이다.

문자열은 여러 문자들의 순열(sequence)이다. 꺾쇠 연산자로 한번에 하나씩 문자에 접근한다.

```
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")

letter <- fruit_letter[[1]][1]
```

두 번째 문장은 변수 `fruit_letter`에서 1번 위치 문자를 추출하여 변수 `letter`에 대입한다. 꺾쇠 표현식을 인덱스(index)라고 부른다. 인덱스는 순서(sequence)에서 사용자가 어떤 문자를 원하는지 표시한다.

하지만, 여러분이 기대한 것은 출력됨이 확인된다.

```
letter
```

```
## [1] "b"
```

파이썬 사용자에게 'banana'의 첫 분자는 a가 아니라 b다. 하지만, 파이썬 인텍스는 문자열 처음부터 오프셋(offset)[1]이다. 첫 글자 오프셋은 0 이다.

하지만, R은 사람 친화적이기 때문에 b가 'banana'의 첫번째 문자가 되고 a가 두번째, n이 세번째 문자가 된다.

인덱스로 문자와 연산자를 포함하는 어떤 표현식도 사용가능지만, 인덱스 값은 정수일 필요는 없다. 정수가 아닌 경우 다음과 같은 결과를 얻게 된다. 문제는 R에서 `1.5`를 내려서 `1`로 처리한다는 점이다. 경우에 따라서는 반올림으로 판단해서 2가 될 수도 있어 오해의 소지가 있기 때문에 무조건 정수로 표현한다.

[1] 컴퓨터에서 어떤 주소로부터 간격을 두고 떨어진 주소와의 거리. 기억 장치가 페이지 혹은 세그먼트 단위로 나누어져 있을 때 하나의 시작 주소로부터 오프셋만큼 떨어진 위치를 나타내는 것이다. 출처: 네이버 지식백과 - 오프셋 [offset] (IT용어사전, 한국정보통신기술협회)

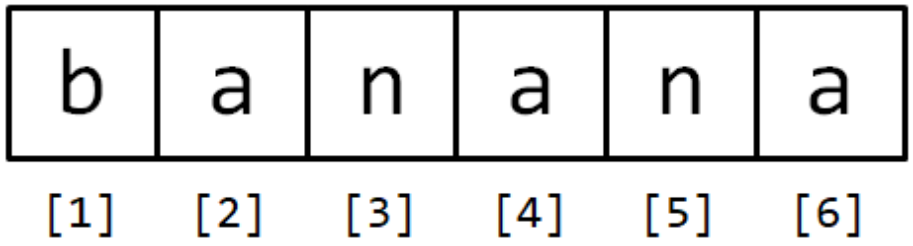



Figure 7.1: 바나나 문자열

```
fruit_letter[[1]][1.5]
```

```
## [1] "b"
```

## 7.2 length() 함수 사용 문자열 길이

length() 함수는 문자열의 문자 갯수를 반환하는 내장함수다.

```
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")

length(fruit_letter[[1]])
```

```
## [1] 6
```

문자열의 가장 마지막 문자를 얻기 위해서, 아래와 같이 시도하려 싶을 것이다.

```
len <- length(fruit_letter[[1]])
fruit_letter[[1]][len]
```

```
## [1] "a"
```

파이썬에서는 인덱스 오류 (IndexError)가 발생하는데 이유는 'banana' 에 6번 인텍스 문자가 없기 때문이다. 0 에서부터 시작했기 때문에 6개 문자는 0 에서부터 5 까지 번호가 매겨졌다. 마지막 문자를 얻기 위해서 length에서 1을 빼야 한다. 하지만, R에서는 사람이 생각하는 방식대로 마지막 문자가 얻어진다.

## 7.3 루프를 사용한 문자열 순회

연산의 많은 경우에 문자열을 한번에 한 문자씩 처리한다. 종종 처음에서 시작해서, 차례로 각 문자를 선택하고, 선택된 문자에 임의 연산을 수행하고, 끝까지 계속한다. 이런 처리 패턴을 **순회법(traversal)**라고 한다. 순회법을 작성하는 한 방법이 while 루프다.

```
index <- 1
while(index <= length(fruit_letter[[1]])){
  letter <- fruit_letter[[1]][index]
  print(letter)
  index <- index + 1
}
```

```
## [1] "b"
## [1] "a"
## [1] "n"
## [1] "a"
## [1] "n"
## [1] "a"
```

while 루프가 문자열을 운행하여 문자열을 한줄에 한 글자씩 화면에 출력한다. 루프 조건이 `index <= length(fruit_letter[[1]])`이여서, `index`가 문자열 길이와 같을 때, 조건은 거짓이 되고, 루프의 몸통 부문은 실행이 되지 않는다. R이 접근한 마지막 `length(fruit_letter[[1]])` 인텍스 문자로, 문자열의 마지막 문자다.

**연습문제:** 문자열의 마지막 문자에서 시작해서, 문자열 처음으로 역진행하면서 한줄에 한자씩 화면에 출력하는 `while` 루프를 작성하세요.

운행법을 작성하는 또 다른 방법은 `for` 루프다.

```
for(char in fruit_letter[[1]]) {
  print(char)
}
```

```
## [1] "b"
## [1] "a"
## [1] "n"
## [1] "a"
## [1] "n"
## [1] "a"
```

루프를 매번 반복할 때, 문자열 다음 문자가 변수 `char`에 대입된다. 루프는 더 이상 남겨진 문자가 없을 때까지 계속 실행된다.

## 7.4 문자열 슬라이스

문자열의 일부분을 **슬라이스(slice)**라고 한다. 문자열 슬라이스를 선택하는 것은 문자를 선택하는 것과 유사하다.

```
s <- strsplit('Monty Python', "")
paste(s[[1]][1:5], collapse="")
```

```
## [1] "Monty"
```

```
paste(s[[1]][7:12], collapse="")
```

```
## [1] "Python"
```

[n:m] 연산자는 n번째 문자부터 m번째 문자까지의 문자열 부분을 반환한다.

파이썬 fruit[:3]와 같이 콜론 앞 첫 인덱스를 생략하면, 문자열 슬라이스는 문자열 처음부터 시작한다. 파이썬 fruit[3:]와 같이 두 번째 인덱스를 생략하면, 문자열 슬라이스는 문자열 끝까지 간다.

이와 동일한 역할을 수행하는 방법은 head(fruit_letter[[1]], 3), tail(fruit_letter[[1]], 3)와 같이 head(), tail() 함수를 활용한다.

```
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")

head(fruit_letter[[1]], 3)
```

```
## [1] "b" "a" "n"
```

```
tail(fruit_letter[[1]], 3)
```

```
## [1] "a" "n" "a"
```

만약 첫번째 인덱스가 두번째보다 크거나 같은 경우 파이썬에는 결과가 인용부호로 표현되는 빈 문자열(empty string)이 된다. 하지만, R에서는 해당 인덱스에 해당되는 문자가 추출된다.

```
fruit_letter[[1]][3:3]
```

```
## [1] "n"
```

```
fruit_letter[[1]][2:1]
```

```
## [1] "a" "b"
```

빈 문자열은 어떤 문자도 포함하지 않아서 길이가 0 이 되지만, 이것을 제외하고 다른 문자열과 동일하다.

## 7.5 루프 돌기 계수

다음 프로그램은 문자열에 문자 a가 나타나는 횟수를 계수(counting)한다.

```
word <- strsplit('banana', "")
count <- 0
for(letter in word[[1]]) {
  if(letter == 'a'){
    count <- count + 1
  }
}
```

```
count
```

```
## [1] 3
```

상기 프로그램은 **계수기(counter)**라고 부르는 또다른 연산 패턴을 보여준다. 변수 count는 0 으로 초기화 되고, 매번 a를 찾을 때마다 증가한다. 루프를 빠져나갔을 때, count는 결과 값 즉, a가 나타난 총 횟수를 담고 있다.

**연습문제** : 문자열과 문자를 인자(argument)로 받도록 상기 코드를 count라는 함수로 **캡슐화(encapsulation)**하고 일반화하세요.

## 7.6 %in% 연산자

연산자 in 은 부울 연산자로 두 개의 문자열을 받아, 첫 번째 문자열이 두 번째 문자열의 일부이면 참(TRUE)을 반환한다.

```
'a' %in% strsplit('banana', "")[[1]]
```

```
## [1] TRUE
```

```
'c' %in% strsplit('banana', "")[[1]]
```

```
## [1] FALSE
```

## 7.7 문자열 비교

비교 연산자도 문자열에서 동작한다. 두 문자열이 같은지를 살펴보다.

```
word <- 'banana'

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

```
## [1] "All right, bananas."
```

다른 비교 연산자는 단어를 알파벳 순서로 정렬하는데 유용하다.

```
word <- 'Pineapple'

if(word < 'banana') {
  cat('Your word',  word, ' comes before banana.')
} else if (word > 'banana') {
  cat('Your word', word, ' comes after banana.')
} else {
  cat('All right, bananas.')
}
```

```
## Your word Pineapple  comes after banana.
```

이러한 문제를 다루는 일반적인 방식은 비교 연산을 수행하기 전에 문자열을 표준 포맷으로 예를 들어 모두 소문자, 변환하는 것입니다. 경우에 따라서 “Pineapple”로 무장한 사람들로부터 여러분을 보호해야하는 것을 명심하세요.

## 7.8 문자열 함수

R은 객체지향언어의 특성을 갖고 있지만 함수형 프로그래밍 언어의 특성도 갖고 있다. 문자열을 R 객체(objects)로 객체를 데이터(실제 문자열 자체)와 메쏘드(methods)를 담고 있는 것으로 바라볼 수도 있다. 메쏘드는 객체에 내장되고 어떤 객체의 인스턴스(instance)에도 사용되는 사실상 함수다.

하지만, 함수형 프로그래밍 패러다임으로 문자열을 객체로 두고 함수를 적용시켜 다양한 작업을 하는 것이 일반적이다. `tidyverse` 팩키지를 설치하게 되면 `stringr` 팩키지가 구성요소로 포함되어 있다. `str_`로 시작되는 다양한 함수가 지원된다.

예를 들어, `stringr` 팩키지 `str_to_upper()` 함수는 문자열을 받아 모두 대문자로 변환된 새로운 문자열을 반환한다.

```
library(stringr)
word <- 'banana'
new_word <- stringr::str_to_upper(word)
new_word
```

```
## [1] "BANANA"
```

동일한 작업을 함수형 패러다임으로 `str_to_upper(word)`와 같이 표현하는데 반해, 객체지향으로 구현하면 파이썬 같은 경우 `word.upper()` 메쏘드 구문이 사용된다.

예를 들어, 문자열안에 문자열의 위치를 찾는 `str_locate()`, `str_locate_all()`라는 문자열 함수가 있다. `str_locate()`는 매칭되는 첫번째만 반환하는 반면에 `str_locate_all()`는 매칭되는 전부를 반환하는 차이가 있다.

```
str_locate(word, 'a')
```

```
##      start end
## [1,]     2   2
```

상기 예제에서, word 문자열에 `str_locate_all()` 함수를 호출하여 매개 변수로 찾고자 하는 문자를 넘긴다.

`str_locate_all()` 함수로 문자뿐만 아니라 부속 문자열(substring)도 찾을 수 있다.

```
str_locate_all(word, 'na')
```

```
## [[1]]
##      start end
## [1,]     3   4
## [2,]     5   6
```

한 가지 자주 있는 작업은 `str_trim()` 함수를 사용해서 문자열 시작과 끝의 공백(공백 여러개, 탭, 새줄)을 제거하는 것이다.

```
line <- '     Here we go '
str_trim(line)
```

```
## [1] "Here we go"
```

`str_detect()` 함수와 나중에 다룰 정규표현식을 섞어 표현하게 되면 참, 거짓 같은 부울 값(boolean value)을 반환한다. `'^Please'`에서 `^`은 문자열 시작을 지정한다.

```
line <- 'Please have a nice day'
str_detect(line, '^Please')
```

```
## [1] TRUE
```

대소문자를 구별하는 것을 요구하기 때문에 `str_to_lower()` 함수를 사용해서 검증을 수행하기 전에, 한 줄을 입력받아 모두 소문자로 변환하는 것이 필요하다.

```
line <- 'Please have a nice day'
str_detect(line, '^p')
```

```
## [1] FALSE
```

```
str_to_lower(line)
```

```
## [1] "please have a nice day"
```

```
str_detect(str_to_lower(line), '^p')
```

```
## [1] TRUE
```

마지막 예제에서 문자열이 문자 "p"로 시작하는지를 검증하기 위해서, `str_to_lower()` 함수를 호출하고 나서 바로 `str_detect()` 함수를 사용한다. 주의깊게 순서만 다룬다면, 한 줄에 다수 함수를 괄호에 넣어 호출할 수 있다.

**연습문제:** 앞선 예제와 유사한 함수인 `str_count()`로 불리는 문자열 메쏘드가 `stringr` 팩키지 내부에 있다. `? str_count()` 도움말로 `str_count()` 함수에 대한 문서를 읽고, 문자열 'banana'의 문자가 몇 개인지 계수하는 메쏘드 호출 프로그램을 작성하세요.

## 7.9 문자열 파싱

종종, 문자열을 들여다 보고 특정 부속 문자열(substring)을 찾고 싶다. 예를 들어, 아래와 같은 형식으로 작성된 일련의 라인이 주어졌다고 가정하면,

```
From stephen.marquard@uct.ac.zaSat Jan  5 09:14:16 2008
```

각 라인마다 뒤쪽 전자우편 주소(즉, uct.ac.za)만 뽑아내고 싶을 것이다. `str_locate()` 함수와 문자열 슬라이싱(string sliceing)을 사용해서 작업을 수행할 수 있다.

우선, 문자열에서 골뱅이(@, at-sign) 기호의 위치를 찾는다. 그리고, 골뱅이 기호 뒤 첫 공백 위치를 찾는다. 그리고 나서, 찾고자 하는 부속 문자열을 뽑아내기 위해서 문자열 슬라이싱을 사용한다.

```
data <- 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
atpos <- str_locate(data, '@')
atpos
```

```
##      start end
## [1,]    22  22
```

```
sppos <- str_locate_all(data, ' ')
sppos
```

```
## [[1]]
##      start end
## [1,]     5   5
## [2,]    32  32
## [3,]    36  36
## [4,]    40  40
## [5,]    42  42
## [6,]    51  51
```

```
str_sub(data, start=atpos[1,1]+1, end=sppos[[1]][2,2]-1)
```

```
## [1] "uct.ac.za"
```

`str_locate()` 함수를 사용해서 찾고자 하는 문자열의 시작 위치를 명세한다. 문자열 슬라이싱(slicing)할 때, 골뱅기 기호 뒤부터 빈 공백을 포함하지 않는 위치까지 문자열을 뽑아낸다.

## 7.10 서식 연산자

**서식 연산자(format operator)** Base R의 `sprintf()` 함수에 C언어 스타일로 %를 사용하기도 하지만 glue: Interpreted String Literals 팩키지도 최근에 많이 사용된다. `glue` 팩키지 {}는 문자열 일부를 변수에 저장된 값으로 바꿔 문자열을 구성한다. 정수에 서식 연산자가 적용될 때, {}는 나머지 연산자가 된다. 하지만 첫 피연산자가 문자열이면, {}은 서식 연산자가 된다.

첫 피연산자는 서식 문자열(format string)로 두번째 피연산자가 어떤 형식으로 표현되는지를 명세하는 하나 혹은 그 이상의 서식 순서(format sequence)를 담고 있다. 결과값은 문자열이다.

예를 들어, 형식 순서 '%d'의 의미는 두번째 피연산자가 정수 형식으로 표현됨을 뜻한다. (d는 "decimal"을 나타낸다.)

```
camels <- 42
sprintf('%d', camels)
```

```
## [1] "42"
```

```
glue::glue("{camels}")
```

```
## 42
```

결과는 문자열 '42'로 정수 42와 혼동하면 안 된다.

서식 순서는 문자열 어디에도 나타날 수 있어서 문장 중간에 값을 임베드(embed)할 수 있다.

```
camels <- 42
sprintf('I have spotted %d camels.', camels)
```

```
## [1] "I have spotted 42 camels."
```

```
glue::glue('I have spotted {camels} camels.')
```

```
## I have spotted 42 camels.
```

만약 문자열 서식 순서가 하나 이상이라면, 두번째 인자는 튜플(tuple)이 된다. 서식 순서 각각은 순서대로 튜플 요소와 매칭된다.

다음 예제는 정수 형식을 표현하기 위해서 '%d', 부동 소수점 형식을 표현하기 위해서 '%g', 문자열 형식을 표현하기 위해서 '%s'을 사용한 사례다. 여기서 왜 부동 소수점 형식이'%f'대신에 '%g'인지는 질문하지 말아주세요.

```
> camels <- 42
> glue('I have spotted {camels} camels.')
I have spotted 42 camels.
> sprintf('I have spotted %d camels & %d camels', camels)
Error in sprintf("I have spotted %d camels & %d camels", camels) :
  too few arguments
> camels <- "king"
> sprintf('I have spotted %d camels.', camels)
Error in sprintf("I have spotted %d camels.", camels) :
  invalid format '%d'; use format %s for character objects
```

문자열 서식 순서와 갯수는 일치해야 하고, 요소의 자료형(type)도 서식 순서와 일치해야 한다.

상기 첫 예제는 충분한 요소 개수가 되지 않고, 두 번째 예제는 자료형이 맞지 않는다. 서식 연산자는 강력하지만, 사용하기가 까다로운 점이 있으니, glue를 사용하는 것도 권장된다.

## 7.11 디버깅

프로그램을 작성하면서 배양해야 하는 기술은 항상 자신에게 질문을 하는 것이다. "여기서 무엇이 잘못 될 수 있을까?" 혹은 "내가 작성한 완벽한 프로그램을 망가뜨리기 위해 사용자는 무슨 엄청난 일을 할 것인가?"

예를 들어 앞장의 반복 while 루프를 시연하기 위해 사용한 프로그램을 살펴봅시다.

```
while(TRUE) {
  line <- readline(prompt = '> ')
  if(substr(line,1,1) == "#") {
    next
  }
  if(line == 'done') {
    break
  }
  print(line)
}
```

사용자가 입력값으로 빈 공백 줄을 입력하게 될 때 무엇이 발생하는지 살펴봅시다.

```
> hello there
[1] hello there
> # don't print this
> print this!
[2] print this!
>
[1] ""
> done
```

빈 공백줄이 입력될 때까지 코드는 잘 작동합니다. 그리고 나서, 파이썬의 경우 0번째 문자가 없어서 트레이스백(traceback)이 발생한다. R의 경우 정상 실행되지만 원하는 바는 아니다. 입력줄이 비어있을 때, 코드 3번째 줄을 "안전"하게 만드는 두 가지 방법이 있다.

하나는 빈 문자열이면 거짓(FALSE)을 반환하도록 `str_detect()` 함수를 사용하는 것이다.

```
if(str_detect(line, '^#'))
```

**가디언 패턴(guardian pattern)**을 사용한 `if`문으로 문자열에 적어도 하나의 문자가 있는 경우만 두번째 논리 표현식이 평가되도록 코드를 작성한다.

```
if(str_length(line) > 0 & str_detect(line, '^#'))
```

## 7.12 용어정의

**계수기(counter)**: 무언가를 계수하기 위해서 사용되는 변수로 일반적으로 0으로 초기화하고 나서 증가한다. **빈 문자열(empty string)**: 두 인용부호로 표현되고, 어떤 문자도 없고 길이가 0 인 문자열. **서식 연산자(format operator)**: 서식 문자열과 튜플을 받아, 서식 문자열에 지정된 서식으로 튜플 요소를 포함하는 문자열을 생성하는 연산자, . **서식 순서(format sequence)**: d처럼 어떤 값의 서식으로 표현되어야 하는지를 명세하는 "서식 문자열" 문자 순서. **서식 문자열(format string)**: 서식 순서를 포함하는 서식 연산자와 함께

사용되는 문자열. **플래그(flag)**: 조건이 참인지를 표기하기 위해 사용하는 불 변수(boolean variable) **호출(invocation)**: 메쏘드를 호출하는 명령문. **불변(immutable)**: 순서의 항목에 대입할 수 없는 특성. **인덱스(index)**: 문자열의 문자처럼 순서(sequence)에 항목을 선택하기 위해 사용되는 정수 값. **항목(item)**: 순서에 있는 값의 하나. **메쏘드(method)**: 객체와 연관되어 점 표기법을 사용하여 호출되는 함수. **객체(object)**: 변수가 참조하는 무엇. 지금은 "객체"와 "값"을 구별없이 사용한다. **검색(search)**: 찾고자 하는 것을 찾았을 때 멈추는 운행법 패턴. **순서(sequence)**: 정돈된 집합. 즉, 정수 인텍스로 각각의 값이 확인되는 값의 집합. **슬라이스(slice)**: 인텍스 범위로 지정되는 문자열 부분. **운행법(traverse)**: 순서(sequence)의 항목을 반복적으로 훑기, 각각에 대해서는 동일한 연산을 수행.

## 7.13 연습문제

1. 다음 문자열에서 숫자를 뽑아내는 R 코드로 작성하세요.

```
str <- 'X-DSPAM-Confidence: 0.8475'
```

`str_locate()` 함수와 문자열 슬라이싱을 사용하여 `str_sub()` 문자 뒤 문자열을 뽑아내고 `as.numeric()` 함수를 사용하여 뽑아낸 문자열을 부동 소수점 숫자로 변환하세요.

# Chapter 8

# 파일

## 8.1 영속성(Persistence)

지금까지, 프로그램을 어떻게 작성하고 조건문, 함수, 반복을 사용하여 중앙처리장치(CPU, Central Processing Unit)에 프로그래머의 의도를 커뮤니케이션하는지 학습했다. 주기억장치(Main Memory)에 어떻게 자료구조를 생성하고 사용하는지도 배웠다. CPU와 주기억장치는 소프트웨어가 동작하고 실행되는 곳이고, 모든 "생각(thinking)"이 발생하는 장소다.

하지만, 앞서 하드웨어 아키텍처를 논의했던 기억을 되살린다면, 전원이 꺼지게 되면, CPU와 주기억장치에 저장된 모든 것이 지워진다. 지금까지 작성한 프로그램은 R을 배우기 위한 일시적으로 재미로 연습한 것에 불과하다.

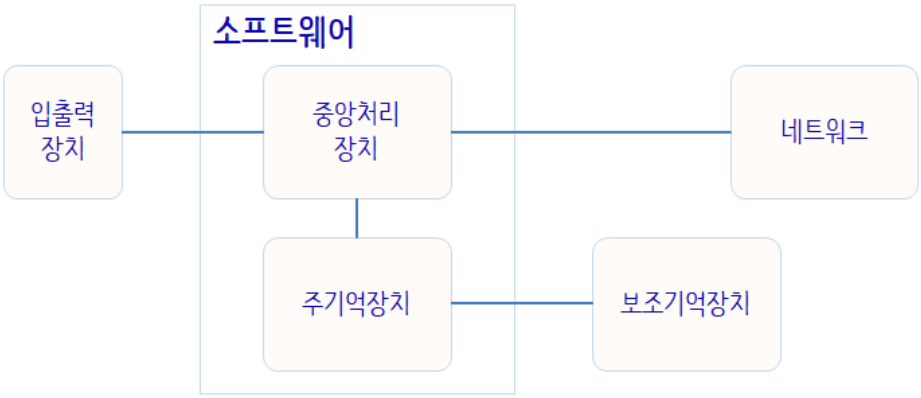



Figure 8.1: 소프트웨어 아키텍처

이번 장에서 보조 기억장치(Secondary Memory) 혹은 파일을 가지고 작업을 시작한다. 보조 기억장치는 전원이 꺼져도 지워지지 않는다. 혹은, USB 플래쉬

드라이브를 사용한 경우에는 프로그램으로부터 작성한 데이터는 시스템에서 제거되어 다른 시스템으로 전송될 수 있다.

우선 텍스트 편집기로 작성한 텍스트 파일을 읽고 쓰는 것에 초점을 맞출 것이다. 나중에 데이터베이스 소프트웨어를 통해서 읽고 쓰도록 설계된 바이너리 파일 데이터베이스를 가지고 어떻게 작업하는지를 살펴볼 것이다.

## 8.2 파일 열기

하드 디스크 파일을 읽거나 쓸려고 할 때, 파일을 열어야(open) 한다. 파일을 열 때 각 파일 데이터가 어디에 저장되었는지를 알고 있는 운영체제와 커뮤니케이션 한다. 파일을 열 때, 운영체제에 파일이 존재하는지 확인하고 이름으로 파일을 찾도록 요청한다.
이번 예제에서, <www.py4inf.com/code/mbox.txt> 에서 파일을 다운로드한 후 R을 시작한 동일한 폴더에 저장된 `mbox.txt` 파일을 연다.

`download.file("http://www.py4inf.com/code/mbox.txt", destfile = "data/mbox.txt")` 명령어를 사용하여 코딩을 시작하는 디렉토리 아래 `data` 디렉토리를 만들고 동일한 `mbox.txt` 이름으로 저장한다.

```
# download.file("http://www.py4inf.com/code/mbox.txt", destfile = "data/mbox.txt")

fhand <- file("data/mbox.txt", open = "r")
fhand
```

```
## A connection with
## description "data/mbox.txt"
## class       "file"
## mode        "r"
## text        "text"
## opened      "opened"
## can read    "yes"
## can write   "no"
```

open이 성공하면, 운영체제는 **파일 핸들(file handle)**을 반환한다. 파일 핸들(file handle)은 파일에 담겨진 실제 데이터는 아니고, 대신에 데이터를 읽을 수 있도록 사용할 수 있는 "핸들(handle)"이다. 요청한 파일이 존재하고, 파일을 읽을 수 있는 적절한 권한이 있다면 이제 핸들이 여러분에게 주어졌다.

파일이 존재하지 않는다면, open은 역추적(traceback) 파일 열기 오류로 실패하고, 파일 콘텐츠에 접근할 핸들도 얻지 못한다.

```
> fhand <- file("data/stuff.txt", "r")
Error in file("data/stuff.txt", "r") : cannot open the connection
In addition: Warning message:
In file("data/stuff.txt", "r") :
  cannot open file 'data/stuff.txt': No such file or directory
```

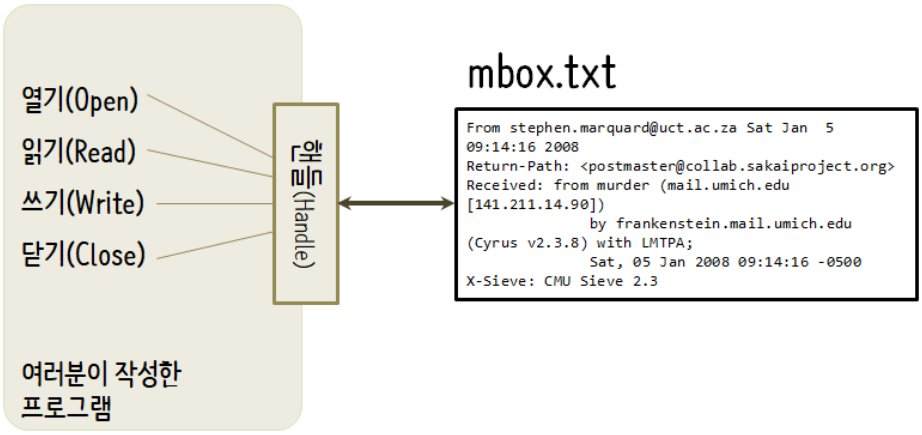



Figure 8.2: 파일 핸들

나중에 `tryCatch()`를 가지고, 존재하지 않는 파일을 열려고 하는 상황을 좀더 우아하게 처리할 것이다.

최근에 사용자 중심으로 R에 다양한 기능이 추가되어 `tidyverse` 팩키지 일부를 구성하는 `readr` 팩키지의 `read_lines()` 함수를 통해 인터넷 웹사이트에서 바로 불러오는 것도 가능하다.

```
# library(tidyverse)

txt_file <- readr::read_lines("http://www.py4inf.com/code/mbox.txt")

head(txt_file)
```

```
## [1] "From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008"
## [2] "Return-Path: <postmaster@collab.sakaiproject.org>"
## [3] "Received: from murder (mail.umich.edu [141.211.14.90])"
## [4] "\t by frankenstein.mail.umich.edu (Cyrus v2.3.8) with LMTPA;"
## [5] "\t Sat, 05 Jan 2008 09:14:16 -0500"
## [6] "X-Sieve: CMU Sieve 2.3"
```

## 8.3 텍스트 파일과 라인

R 문자열이 문자 순서(sequence)로 간주 되듯이 마찬가지로 텍스트 파일은 줄(라인, line) 순서(sequence)로 생각될 수 있다. 예를 들어, 다음은 오픈 소스 프로젝트 개발팀에서 다양한 참여자들의 전자우편 활동을 기록한 텍스트 파일 샘플이다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
```

```
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=revrev=39772
...
```

상호 의사소통한 전자우편 전체 파일은 〈www.py4inf.com/code/mbox.txt〉에서 접근 가능하고, 간략한 버젼 파일은 〈www.py4inf.com/code/mbox-short.txt〉 에서 얻을 수 있다. 이들 파일은 다수 전자우편 메시지를 담고 있는 파일로 표준 포맷으로 되어 있다. "From"으로 시작하는 라인은 메시지 본문과 구별되고, "From:"으로 시작하는 라인은 본문 메시지의 일부다. 더 자세한 정보는 http://en.wikipedia.org/wiki/Mbox에서 찾을 수 있다.

파일을 라인으로 쪼개기 위해서, 새줄(newline) 문자로 불리는 "줄의 끝(end of the line)"을 표시하는 특수 문자가 있다.

R에서, 문자열 상수 역슬래쉬-n(\n)으로 새줄(newline) 문자를 표현한다. 두 문자처럼 보이지만, 사실은 단일 문자다. 인터프리터에 "stuff"에 입력한 후 변수를 살펴보면, 문자열에 \n가 있다. 하지만, cat문을 사용하여 문자열을 출력하면, 문자열이 새줄 문자에 의해서 두 줄로 쪼개지는 것을 볼 수 있다.

```
stuff <- 'Hello\nWorld!'
stuff
```

```
## [1] "Hello\nWorld!"
```

```
cat(stuff)
```

```
## Hello
## World!
```

```
stuff <- 'X\nY'
cat(stuff)
```

```
## X
## Y
```

```
str_length(stuff)
```

```
## [1] 3
```

문자열 'X\nY'의 길이는 stringr::str_length("X\nY") 명령어를 통해 확인이 가능한데 3 이다. 왜냐하면 새줄(newline) 문자도 한 문자이기 때문이다.

그래서, 파일 라인을 볼 때, 라인 끝을 표시하는 새줄(newline)로 불리는 눈에 보이지 않는 특수 문자가 각 줄의 끝에 있다고 상상할 필요가 있다.

그래서, 새줄(newline) 문자는 파일에 있는 문자를 라인으로 분리한다.

## 8.4 파일 읽어오기

파일 핸들(file handle)이 파일 자료를 담고 있지 않지만, for 루프를 사용하여 파일 각 라인을 읽고 라인수를 세는 것을 쉽게 구축할 수 있다.

```
fhand <- file('data/mbox.txt', open = "r")
count <- 0
for(line in readLines(fhand)) {
  count <- count + 1
}

cat('Line Count:', count, "\n")
```

```
## Line Count: 132045
```

```
close(fhand)
```

파일 핸들을 `for` 루프 순서(sequence)로 사용할 수 있다. `for` 루프는 단순히 파일 라인 수를 세고 전체 라인수를 출력한다. `for` 루프를 대략 일반어로 풀어 말하면, "파일 핸들로 표현되는 파일 각 라인마다, `count` 변수에 1 씩 더한다"

`file` 함수가 전체 파일을 바로 읽지 못하는 이유는 파일이 수 기가 바이트 파일 크기를 가질 수도 있기 때문이다. `file` 문장은 파일 크기에 관계없이 파일을 여는데 시간이 동일하게 걸린다. 실질적으로 `for` 루프가 파일로부터 자료를 읽어오는 역할을 한다.

`for` 루프를 사용해서 이 같은 방식으로 파일을 읽어올 때, 새줄(newline) 문자를 사용해서 파일 자료를 라인 단위로 쪼갠다. 파이썬에서 새줄(newline) 문자까지 각 라인 단위로 읽고, `for` 루프가 매번 반복할 때마다 line 변수에 새줄(newline)을 마지막 문자로 포함한다.

`for` 루프가 데이터를 한번에 한줄씩 읽어오기 때문에, 데이터를 저장할 주기억장치 저장공간을 소진하지 않고, 매우 큰 파일을 효과적으로 읽어서 라인을 셀 수 있다. 각 라인별로 읽고, 세고, 그리고 나서 폐기되기 때문에, 매우 적은 저장공간을 사용해서 어떤 크기의 파일도 상기 프로그램을 사용하여 라인을 셀 수 있다.

만약 주기억장치 크기에 비해서 상대적으로 작은 크기의 파일이라는 것을 안다면, 전체 파일을 파일 핸들로 `readLines()` 함수를 사용해서 문자열로 읽어올 수 있다.

```
fhand <- file("data/mbox-short.txt", open = "r")
inp <- readChar(fhand, nchars=1e6)
str_length(inp)
```

```
## [1] 94626
```

```
close(fhand)

str_sub(inp, 1, 20)
```

```
## [1] "From stephen.marquar"
```

상기 예제에서, `mbox-short.txt` 전체 파일 콘텐츠(96,536 문자)를 변수 `inp`로 바로 읽었다. 문자열 슬라이싱을 사용해서 `inp`에 저장된 문자열 자료 첫 20 문자를 출력한다.

파일이 이런 방식으로 읽혀질 때, 모든 라인과 새줄(newline)문자를 포함한 모든 문자는 변수 `inp`에 대입된 매우 큰 문자열이다. 파일 데이터가 컴퓨터 주기억장치가 안정적으로 감당해 낼 수 있을때만, 이런 형식의 `readChar()` 함수가 사용될 수 있다는 것을 기억하라.

만약 주기억장치가 감당해 낼 수 없는 매우 파일 크기가 크다면, `for`나 `while` 루프를 사용해서 파일을 쪼개서 읽는 프로그램을 작성해야 한다.

## 8.5 파일 검색

파일 데이터를 검색할 때, 흔한 패턴은 파일을 읽고, 대부분 라인은 건너뛰고, 특정 기준을 만족하는 라인만 처리하는 것이다. 간단한 검색 메카니즘을 구현하기 위해서 파일을 읽는 패턴과 문자열 메쏘드를 조합한다.

예를 들어, 파일을 읽고, "From:"으로 시작하는 라인만 출력하고자 한다면, `stringr` 팩키지에 포함된 `str_detect()` 문자열 탐지 함수를 사용해서 원하는 접두사(From:)로 시작하는 라인만을 선택한다.

```
fhand <- file('data/mbox-short.txt', open = "r")

for(line in readLines(fhand)) {
  if(str_detect(line, "^From:")) {
    print(line)
  }
}
```

이 프로그램이 실행하면 다음 출력값을 얻는다.

```
[1] "From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008"
[1] "From: stephen.marquard@uct.ac.za"
[1] "From louis@media.berkeley.edu Fri Jan  4 18:10:48 2008"
[1] "From: louis@media.berkeley.edu"
[1] "From zqian@umich.edu Fri Jan  4 16:10:39 2008"
[1] "From: zqian@umich.edu"
[1] "From rjlowe@iupui.edu Fri Jan  4 15:46:24 2008"
```

"From:"으로만 시작하는 라인만 출력하기 때문에 출력값은 훌륭해 보인다.

파일 처리 프로그램이 점점 더 복잡해짐에 따라 `next`를 사용해서 검색 루프(search loop)를 구조화할 필요가 있다. 검색 루프의 기본 아이디어는 "흥미로운" 라인을 집중적으로 찾고, "흥미롭지 않은" 라인은 효과적으로 건너뛰는 것이다. 그리고 나서 흥미로운 라인을 찾게되면, 그 라인에서 특정 연산을 수행하는 것이다.

다음과 같이 루프를 구성해서 흥미롭지 않은 라인은 건떠뛰는 패턴을 따르게 한다.

```
fhand <- file('data/mbox-short.txt', open = "r")

for(line in readLines(fhand)) {
  if(!str_detect(line, "^From:")) {
    next
  }
  print(line)
}
```

프로그램의 출력값은 동일하다. 흥미롭지 않는 라인은 “From:”으로 시작하지 않는 라인이라 next문을 사용해서 건너뛴다. “흥미로운” 라인 (즉, “From:”으로 시작하는 라인)에 대해서는 연산처리를 수행한다.

str_detect() 문자열 함수를 사용해서 검색 문자열이 라인 어디에 있는지를 찾아주는 텍스트 편집기 검색기능을 모사(simulation)할 수 있다. str_detect() 문자열 함수는 다른 문자열 내부에 검색하는 문자열이 있는지 찾고, 존재하는 경우 참(TRUE) 만약 문자열이 없다면 거짓(FALSE)을 반환하기 때문에, “(**uct.ac.za?**)”(남아프리카 케이프 타운 대학으로부터 왔다) 문자열을 포함하는 라인을 검색하기 위해 다음과 같이 루프를 작성한다.

```
fhand <- file('data/mbox-short.txt', open = "r")

for(line in readLines(fhand)) {
  if(!str_detect(line, "@uct.ac.za")) {
    next
  }
  print(line)
}
```

출력결과는 다음과 같다.

```
[1] "From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008"
[1] "X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to stephen.marquard@uct
[1] "From: stephen.marquard@uct.ac.za"
[1] "Author: stephen.marquard@uct.ac.za"
[1] "From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008"
[1] "X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to david.horwitz@uct.ac
[1] "From: david.horwitz@uct.ac.za"
[1] "Author: david.horwitz@uct.ac.za"
[1] "r39753 | david.horwitz@uct.ac.za | 2008-01-04 13:05:51 +0200 (Fri, 04 Jan 2008) | 1 line"
[1] "From david.horwitz@uct.ac.za Fri Jan  4 06:08:27 2008"
[1] "X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to david.horwitz@uct.ac
```

## 8.6 사용자가 파일명 선택

매번 다른 파일을 처리할 때마다 R 코드를 편집하고 싶지는 않다. 매번 프로그램이 실행될 때마다, 파일명을 사용자가 입력하도록 만드는 것이 좀더 유용할 것이다. 그래서 R 코드를 바꾸지 않고, 다른 파일에 대해서도 동일한 프로그램을 사용하도록 만들자.

다음과 같이 commandArgs을 사용해서 사용자로부터 파일명을 읽어 프로그램을 실행하는 것이 단순하다. file-user-input.R 파일에 다음과 같이 R 스크립트를 작성한다. 자세한 사항은 R 병렬 프로그래밍을 참조한다. [1] 그리고, 사용자의 입력을 받도록 하는 프롬프트를 생략하고 바로 쉘에서 인자를 넘기는 것으로 프로그램을 작성했다.

```
#!/usr/bin/env Rscript
library(stringr)

args <- commandArgs(trailingOnly=TRUE)
fname <- args[1]

fhand <- file(fname, open = "r")

count <- 0
for(line in readLines(fhand)) {
  if(str_detect(line, "Subject:")) {
    count <- count + 1
  }
}

cat('There were', count, 'subject lines in', fname)
```

사용자로부터 파일명을 읽고 변수 fname에 저장하고, 그 파일을 연다. 이제 다른 파일에 대해서도 반복적으로 프로그램을 실행할 수 있다. RStudio Terminal (Console 패널 아님)을 열고 다음과 같이 인자를 넘겨 실행하면 된다.

```
D:\docs\r4inf\code>Rscript file-user-input.R "data/mbox-short.txt"
There were 27 subject lines in ../data/mbox-short.txt
```

다음 절을 엿보기 전에, 상기 프로그램을 살펴보고 자신에게 다음을 질문해 보자. "여기서 어디가 잘못될 수 있는가?" 혹은 "이 작고 멋진 프로그램에 트레이스백(traceback)을 남기고 바로 끝나게 하여, 결국 사용자 눈에는 좋지 않은 프로그램이라는 인상을 남길 수 있도록 우리의 친절한 사용자는 무엇을 할 수 있을까?"

[1] .R 스크립트를 인자와 함께 실행

## 8.7 tryCatch 사용하기

제가 여러분에게 엿보지 말라고 말씀드렸습니다. 이번이 마지막 기회입니다. 사용자가 파일명이 아닌 뭔가 다른 것을 입력하면 어떻게 될까요?

```
D:\docs\r4inf\code>Rscript file-user-input.R "data/missing.txt"
Error in file(fname, open = "r") :
: ():
In file(fname, open = "r") :
   'data/missing.txt'  : No such file or directory

D:\docs\r4inf\code>Rscript file-user-input.R "na na boo boo"
Error in file(fname, open = "r") :
: ():
In file(fname, open = "r") :
   'na na boo boo'  : No such file or directory
```

웃지마시구요, 사용자는 결국 여러분이 작성한 프로그램을 망가뜨리기 위해 고의든 악의를 가지든 가능한 모든 수단을 강구할 것입니다. 사실, 소프트웨어 개발팀의 중요한 부분은 **품질 보증(Quality Assurance, QA)**이라는 조직이다. 품질보증 조직은 프로그래머가 만든 소프트웨어를 망가뜨리기 위해 가능한 말도 안 되는 것을 합니다.

사용자가 소프트웨어를 제품으로 구매하거나, 주문형으로 개발하는 프로그램에 대해 월급을 지급하던지 관계없이 품질보증 조직은 프로그램이 사용자에게 전달되기 전까지 프로그램 오류를 발견할 책임이 있다. 그래서 품질보증 조직은 프로그래머의 최고의 친구다.

프로그램 오류를 찾았기 때문에, `tryCatch` 구조를 사용해서 오류를 우아하게 고쳐봅시다. 파일 열기 `file()` 호출이 잘못될 수 있다고 가정하고, `file()` 호출이 실패할 때를 대비해서 다음과 같이 복구 코드를 추가한다.

```
#!/usr/bin/env Rscript
library(stringr)

args <- commandArgs(trailingOnly=TRUE)
fname <- args[1]

tryCatch({
  fhand <- file(fname, open = "r")},
  error = function(err) print(paste("ERROR: ", err))
)

count <- 0
for(line in readLines(fhand)) {
  if(str_detect(line, "Subject:")) {
    count <- count + 1
  }
```

```
}

cat('There were', count, 'subject lines in', fname)
```

이제 사용자 혹은 품질 보증 조직에서 올바르지 않거나 어처구니 없는 파일명을 입력했을 때, 버그를 "tryCatch()" 함수로 잡아서 우아하게 복구한다.

```
D:\docs\r4inf\code> Rscript file-user-input-try.R "na na boo boo"
[1] "ERROR:  Error in file(fname, open = \"r\"):    \n"
():
In file(fname, open = "r") :
   'na na boo boo'  : No such file or directory
Error in readLines(fhand) :  'fhand'


D:\docs\r4inf\code>Rscript file-user-input-try.R "../data/mbox-short.txt"
There were 27 subject lines in ../data/mbox-short.txt
D:\docs\r4inf\code>Rscript file-user-input-try.R "../data/mbox.txt"
There were 1797 subject lines in ../data/mbox.txt
```

R 프로그램을 작성할 때 `file()` 파일 열기 호출을 보호하는 것은 `tryCath()`의 적절한 사용 예제가 된다. "R 방식(R way)"으로 무언가를 작성할 때, "알스러운"이라는 용어를 사용한다. 상기 파일을 여는 예제는 알스러운 방식의 좋은 예가 된다고 말한다.

R에 좀더 자신감이 생기게 되면, 다른 R 프로그래머와 동일한 문제에 대해 두 가지 동치하는 해답을 가지고 어떤 접근법이 좀더 "알스러운지"에 대한 현답을 찾는데도 관여하게 된다.

"좀더 알스럽게" 되는 이유는 프로그래밍이 엔지니어링적인 면과 예술적인 면을 동시에 가지고 있기 때문이다. 항상 무언가를 단지 작동하는 것에만 관심이 있지 않고, 프로그램으로 작성한 해결책이 좀더 우아하고, 다른 동료에 의해서 우아한 것으로 인정되기를 또한 원합니다.

## 8.8 파일에 쓰기

파일에 쓰기 위해서는 두 번째 매개 변수로 'w' 모드로 파일을 열어야 한다.

```
fhand <- file("data/output.txt", open = "w")
fhand
```

파일이 이미 존재하는데 쓰기 모드에서 파일을 여는 것은 이전 데이터를 모두 지워버리고, 깨끗한 파일 상태에서 다시 시작되니 주의가 필요하다. 만약 파일이 존재하지 않는다면, 새로운 파일이 생성된다.

파일 핸들 객체의 `writeLines()` 함수는 데이터를 파일에 저장한다.

```
writeLines("This here's the wattle,", fhand)
```

다시 한번, 파일 객체는 마지막 포인터가 어디에 있는지 위치를 추적해서, 만약 write 메쏘드를 다시 호출하게 되면, 새로운 데이터를 파일 끝에 추가한다.

라인을 끝내고 싶을 때, 명시적으로 새줄(newline) 문자를 삽입해서 파일에 쓰도록 라인 끝을 필히 관리해야 한다.

print문이 자동적으로 새줄(newline)을 추가하듯이 writeLines() 함수도 자동적으로 새줄(newline)을 추가한다.

```
writeLines("the emblem of our land.'", fhand)
```

파일 쓰기가 끝났을 때, 파일을 필히 닫아야 한다. 파일을 닫는 것은 데이터 마지막 비트까지 디스크에 물리적으로 쓰여져서, 전원이 나가더라도 자료가 유실되지 않는 역할을 한다.

```
close(fhand)
```

파일 읽기로 연 파일을 닫을 수 있지만, 몇개 파일을 열어 놓았다면 약간 단정치 못하게 끝날 수 있습니다. 왜냐하면 프로그램이 종료될 때 열린 모든 파일이 닫혀졌는지 파이썬이 확인하기 때문이다. 파일에 쓰기를 할 때는, 파일을 명시적으로 닫아서 예기치 못한 일이 발생할 여지를 없애야 한다.

파일에 두 문장을 써 넣은 결과는 다음과 같다.

```
D:\docs\r4inf> cat data/output.txt
This here's the wattle,
the emblem of our land.
```

## 8.9 디버깅

파일을 읽고 쓸 때, 공백 때문에 종종 문제에 봉착한다. 이런 종류의 오류는 공백, 탭, 새줄(newline)이 눈에 보이지 않기 때문에 디버그하기도 쉽지 않다.

```
s <- '1 2\t 3\n 4'
cat(s)
```

```
## 1 2   3
##  4
```

우선 RStudio IDE의 상단 메뉴에서 Tools -> Global Options -> Code -> Display -> “Show whitespace characters” 를 통해 공백문자(whitespace)에 대해 확인할 수 있다.

문자열 공백문자는 역슬래쉬 순서(sequence)로 나타냅니다.

여러분이 봉착하는 또 다른 문제는 다른 시스템에서는 라인 끝을 표기하기 위해서 다른 문자를 사용한다는 점이다. 어떤 시스템은 \n 으로 새줄(newline)을 표기하고, 다른 시스템은 \r으로 반환 문자(return character)를 사용한다. 둘다 모두

사용하는 시스템도 있다. 파일을 다른 시스템으로 이식한다면, 이러한 불일치가 문제를 야기한다.

대부분의 시스템에는 A 포맷에서 B 포멧으로 변환하는 응용프로그램이 있다. https://en.wikipedia.org/wiki/Newline 에서 응용프로그램을 찾을 수 있고, 좀더 많은 것을 읽을 수 있다. 물론, 여러분이 직접 프로그램을 작성할 수도 있다.

## 8.10 용어정의

- **잡기(catch)**: try와 except 문을 사용해서 프로그램이 끝나는 예외 상황을 방지하는 것.
- **새줄(newline)**: 라인의 끝을 표기 위한 파일이나 문자열에 사용되는 특수 문자.
- **파이썬스러운(Pythonic)**: 파이썬에서 우아하게 작동하는 기술. "try와 catch를 사용하는 것은 파일이 없는 경우를 복구하는 파이썬스러운 방식이다."
- **품질 보증(Quality Assurance, QA)**: 소프트웨어 제품의 전반적인 품질을 보중하는데 집중하는 사람이나 조직. 품질 보증은 소프트웨어 제품을 시험하고, 제품이 시장에 출시되기 전에 문제를 확인하는데 관여한다.
- **텍스트 파일(text file)**: 하드디스크 같은 영구 저장소에 저장된 일련의 문자 집합.

## 8.11 연습문제

1. 파일을 읽고 한줄씩 파일의 내용을 모두 대문자로 출력하는 프로그램을 작성하세요. 프로그램을 실행하면 다음과 같이 보일 것입니다.

```
$ Rscript shout.R "mbox-short.txt"

FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
SAT, 05 JAN 2008 09:14:16 -0500
```

http://www.py4inf.com/code/mbox-short.txt에서 파일을 다운로드 받으세요.

2. 파일명을 입력받아, 파일을 읽고, 다음 형식의 라인을 찾는 프로그램을 작성하세요.

   X-DSPAM-Confidence: **0.8475**

"X-DSPAM-Confidence:"로 시작하는 라인을 만나게 되면, 부동 소수점 숫자를 뽑아내기 위해 해당 라인을 별도로 보관하세요. 라인 수를 세고, 라인으로부터 스팸 신뢰값의 총계를 계산하세요. 파일의 끝에 도달할 했을 때, 평균 스팸 신뢰도를 출력하세요.

```
$ Rscript calc.R "mbox-short.txt"
Average spam confidence: 0.750718518519

$ Rscript calc.R "mbox.txt"
Average spam confidence: 0.894128046745
```

mbox.txt와 mbox-short.txt 파일에 작성한 프로그램을 시험하세요.

3. 때때로, 프로그래머가 지루해지거나, 약간 재미를 목적으로, 프로그램에 무해한 부활절 달걀(Easter Egg, https://en.wikipedia.org/wiki/Easter_egg_(media))을 넣습니다. 사용자가 파일명을 입력하는 프로그램을 변형시켜, 'na na boo boo'로 파일명을 정확하게 입력했을 때, 재미있는 메시지를 출력하는 프로그램을 작성하세요. 파일이 존재하거나, 존재하지 않는 다른 모든 파일에 대해서도 정상적으로 작동해야 합니다. 여기 프로그램을 실행한 견본이 있습니다.

```
$ Rscript egg.R "mbox.txt"
There were 1797 subject lines in mbox.txt

$ Rscript egg.R "missing.tyxt"
File cannot be opened: missing.tyxt

$ Rscript egg.R "na na boo boo"
NA NA BOO BOO TO YOU - You have been punk'd!
```

프로그램에 부활절 달걀을 넣도록 격려하지는 않습니다. 단지 연습입니다.

# Chapter 9

# 리스트 (List)

## 9.1 리스트는 순서(sequence)다.

문자열처럼, 리스트(list)는 값의 순서(sequence)다. 문자열에서, 값은 문자지만, 리스트에서는 임의 자료형(type)도 될 수 있다. 리스트 값은 요소(elements)나 때때로 항목(items)으로 불린다.

신규 리스틀 생성하는 방법은 여러 가지가 있다. 가장 간단한 방법은 `list()`함수로 요소를 감싸는 것이다.

```
list(10, 20, 30, 40)
list('crunchy frog', 'ram bladder', 'lark vomit')
```

첫번째 예제는 4개 정수 리스트다. 두번째 예제는 3개 문자열 리스트다. 문자열 요소가 동일한 자료형(type)일 필요는 없다. 다음 리스트는 문자열, 부동 소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

```
list('spam', 2.0, 5, list(10, 20))
```

또 다른 리스트 내부에 리스트가 **중첩(nested)**되어 있다.

어떤 요소도 담고 있지 않는 리스트를 빈 리스트(empty list)라고 부르고, `list()`로 생성한다.

예상했듯이, 리스트 값을 변수에 대입할 수 있다.

```
(cheeses <- list('Cheddar', 'Edam', 'Gouda'))
```

```
## [[1]]
## [1] "Cheddar"
##
## [[2]]
## [1] "Edam"
##
```

```
## [[3]]
## [1] "Gouda"
```

```
(numbers <- list(17, 123))
```

```
## [[1]]
## [1] 17
##
## [[2]]
## [1] 123
```

```
(empty <- list())
```

```
## list()
```

## 9.2 리스트는 변경가능하다.

리스트 요소에 접근하는 구문은 문자열 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺾쇠 괄호 내부 표현식은 인덱스를 명세한다. 기억할 것은 인덱스는 1에서부터 시작한다는 것이다.

```
cheeses[1]
```

```
## [[1]]
## [1] "Cheddar"
```

문자열과 마찬가지로, 리스트 항목 순서를 바꾸거나, 리스트에 새로운 항목을 다시 대입할 수 있기 때문에 리스트는 변경가능하다. 꺾쇠 괄호 연산자가 대입문 왼쪽편에 나타날 때, 새로 대입될 리스트 요소를 나타낸다.

```
numbers <- list(17, 123)

(numbers[1] <- 5)
```

```
## [1] 5
```

리스트 numbers 첫번째 요소는 123 값을 가지고 있었으나, 이제 5 값을 가진다.

리스트를 인덱스와 요소의 관계로 생각할 수 있다. 이 관계를 **매핑(mapping)**이라고 부른다. 각각의 인덱스는 요소 중 하나에 **대응("maps to")**된다.

리스트 인덱스는 문자열 인덱스와 동일한 방식으로 동작한다.

- 어떠한 정수 표현식도 인덱스로 사용할 수 있다.
- 존재하지 않는 요소를 읽거나 쓰려고 하면, 일종의 인덱스 오류 (IndexError)로 `NULL`이 반환된다.
- 인덱스가 음의 값이면, 해당 리스트 원소가 누락된다.

`%in%` 연산자도 또한 리스트에서 동작하니 리스트 원소로 존재하는지 여부를 판별하는데 사용할 수 있다.

```
cheeses <- list('Cheddar', 'Edam', 'Gouda')

'Edam' %in% cheeses
```

```
## [1] TRUE
```

```
'Brie' %in% cheeses
```

```
## [1] FALSE
```

## 9.3 리스트 순행법

리스트 요소를 운행하는 가장 흔한 방법은 for문을 사용하는 것이다. 문자열에서 사용한 것과 구문은 동일하다.

```
for(cheese in cheeses) {
  print(cheese)
}
```

```
## [1] "Cheddar"
## [1] "Edam"
## [1] "Gouda"
```

리스트 요소를 읽기만 한다면 이것만으로도 잘 동작한다. 하지만, 리스트 요소를 쓰거나, 갱신하는 경우, 인덱스가 필요하다. 리스트 요소를 쓰거나 갱신하는 일반적인 방법은 seq_along() 함수를 조합하는 것이다.

```
numbers <- list(1, 2, 3, 4, 5)

for(i in seq_along(numbers)){
  cat("before:", numbers[[i]], "\n")
  numbers[[i]] <- numbers[[i]] * 2
  cat("after:", numbers[[i]], "\n")
}
```

```
## before: 1
## after: 2
## before: 2
## after: 4
## before: 3
## after: 6
## before: 4
## after: 8
## before: 5
## after: 10
```

상기 루프는 리스트를 운행하고 각 요소를 갱신한다. seq_along() 함수는 1 에서 $n$ 까지 리스트 인덱스를 반환한다. 여기서, $n$은 리스트 길이가 된다. 매번 루프가

반복될 때마다, i는 다음 요소 인덱스를 얻는다. 몸통 부문 대입문은 i를 사용해서 요소의 이전 값을 읽고 새 값을 대입한다.

빈 리스트(list())에 대해서 for문은 결코 몸통 부문을 실행하지 않는다.

```
for(x in list()) {
  print('This never happens.')
}
```

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 요소 하나로 간주된다. 다음 리스트 길이는 4 이다.

```
list('spam', 1, list('Brie', 'Roquefort', 'Pol le Veq'), list(1, 2, 3))
```

```
## [[1]]
## [1] "spam"
##
## [[2]]
## [1] 1
##
## [[3]]
## [[3]][[1]]
## [1] "Brie"
##
## [[3]][[2]]
## [1] "Roquefort"
##
## [[3]][[3]]
## [1] "Pol le Veq"
##
##
## [[4]]
## [[4]][[1]]
## [1] 1
##
## [[4]][[2]]
## [1] 2
##
## [[4]][[3]]
## [1] 3
```

## 9.4 리스트 연산자

append() 함수 연산자는 리스트를 추가하여 결합시킨다.

```
a <- list(1, 2, 3)
b <- list(4, 5, 6)
```

```
c <- append(a, b)
```

유사하게 `rep()` 함수를 활용하면 주어진 횟수만큼 리스트를 반복한다.

```
rep(list(0), 4)
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 0
##
## [[3]]
## [1] 0
##
## [[4]]
## [1] 0
```

```
rep(list(1, 2, 3), 3)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 1
##
## [[5]]
## [1] 2
##
## [[6]]
## [1] 3
##
## [[7]]
## [1] 1
##
## [[8]]
## [1] 2
##
## [[9]]
## [1] 3
```

첫 예제는 `list(0)`을 4회 반복한다. 두 번째 예제는 `list(1, 2, 3)` 리스트를 3회 반복한다.

## 9.5 리스트 슬라이스

**슬라이스(slice)** 연산자는 리스트에도 또한 동작한다.

```
t <- list('a', 'b', 'c', 'd', 'e', 'f')

t[2:3]
```

```
## [[1]]
## [1] "b"
##
## [[2]]
## [1] "c"
```

```
t[1:4]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
```

```
t[4:length(t)]
```

```
## [[1]]
## [1] "d"
##
## [[2]]
## [1] "e"
##
## [[3]]
## [1] "f"
```

```
t[]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
```

```
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
##
## [[5]]
## [1] "e"
##
## [[6]]
## [1] "f"
```

첫 번째 인덱스를 1로 지정하면, 슬라이스는 처음부터 시작한다. 두 번째 인덱스를 `length()` 함수로 리스트 길이를 지정하면, 슬라이스는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, `t[]`같이 지정하면 슬라이스 결과는 전체 리스트를 복사한 것이 된다.

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연산을 수행하기 전에 복사본을 만들어 두는 것이 유용하다.

대입문 왼편의 슬라이스 연산자로 복수의 요소를 갱신할 수 있다.

```
t <- list('a', 'b', 'c', 'd', 'e', 'f')
t[2:3] = list('x', 'y')

t
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "x"
##
## [[3]]
## [1] "y"
##
## [[4]]
## [1] "d"
##
## [[5]]
## [1] "e"
##
## [[6]]
## [1] "f"
```

## 9.6 리스트 함수

R은 리스트 자료형에 연산할수 있는 함수를 제공한다. 예를 들어, 덧붙이기 (append) 함수는 리스트 끝에 신규 요소를 추가한다.

```
t <- list('a', 'b', 'c')
append(t, 'd')
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
```

정렬 (order) 함수는 낮음에서 높음으로 리스트 요소를 정렬한다. 리스트에서 `sapply()` 혹은 `unlist()` 함수로 값으로 변환시키고 `order()` 함수를 통해 내림차순 혹은 오름차순으로 정렬 인덱스를 뽑아 리스트내 원소를 정렬시킨다.

```
t <- list('d', 'c', 'e', 'b', 'a')

t[order(sapply(t,'[[',1))]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
##
## [[5]]
## [1] "e"
```

```
t[order(unlist(t), decreasing=TRUE)]
```

```
## [[1]]
## [1] "e"
##
```

```
## [[2]]
## [1] "d"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "b"
##
## [[5]]
## [1] "a"
```

## 9.7 리스트 요소 삭제

리스트 요소를 삭제하는 방법이 몇 가지 있다. 리스트 요소 명칭을 알고 있다면, 리스트 요소에 NULL을 대입하여 삭제시킨다.

```
t <- list(a='a', b='b', c = 'c')
t[["c"]] <- NULL
t$c <- NULL
t
```

```
## $a
## [1] "a"
##
## $b
## [1] "b"
```

NULL을 대입하여 삭제시킨 리스트는 제거된 요소를 반환한다.

```
t[[1]] <- NULL
t
```

```
## $b
## [1] "b"
```

t[[1]] 리스트 인덱스를 통해 요소에 접근하고 NULL을 대입하여 삭제한다.

(인덱스 혹인 리스트 요인 이름이 아닌) 제거할 요소값을 알고 있다면, 리스트 요인값을 활용해서 제거하는 것도 가능하다.

```
t <- list(a='x', b='y', c = 'z')

t[t != "y"]
```

```
## $a
## [1] "x"
##
## $c
```

```
## [1] "z"
```

하나 이상의 요소를 제거하기 위해서, 슬라이스 인덱스(slice index)를 사용하는 것도 가능하다.

```
t <- list('a', 'b', 'c', 'd', 'e', 'f')
t[-c(2:5)]
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "f"
```

슬라이스는 첫번째 인덱스를 포함하고 두 번째 인덱스까지 모든 요소를 선택한다.

## 9.8 리스트와 함수

루프를 작성하지 않고도 리스트를 빠르게 살펴볼 수 있는 리스트에 적용할 수 있는 내장함수를 활용하는 것도 방법이지만, 깔끔한 세상(tidyverse) 생태계의 일원인 purrr 함수형 프로그래밍 팩키지에 내장된 함수를 활용하는 것도 권장된다.

하지만, 다음과 같이 1차원 리스트는 `unlist()` 함수를 활용하여 벡터로 변환해서 사용하는 것이 편리한 경우가 많다.

```
nums <- list(3, 41, 12, 9, 74, 15) %>% unlist

length(nums)
```

```
## [1] 6
```

```
max(nums)
```

```
## [1] 74
```

```
min(nums)
```

```
## [1] 3
```

```
sum(nums)
```

```
## [1] 154
```

```
sum(nums) / length(nums)
```

```
## [1] 25.7
```

`sum()`, `max()`, `length()` 등 함수는 입력 자료형이 무엇이냐에 따라 다르게 동작할 수 있고, 입력 자료형에 결측값 등 특이값이 들어있는 경우 기대했던 결과가 나올 수 없으니 필히 자료형을 사전에 점검하고 활용하도록 한다.

리스트를 사용해서, 앞서 작성한 프로그램을 다시 작성해서 사용자가 입력한 숫자 목록 평균을 계산한다.

우선 리스트 없이 평균을 계산하는 프로그램:

```
##     : `list_average.R`
total <- 0
count <- 0

while (TRUE) {
  inp <- readline('Enter a number: ')
  if(inp == 'done') break

  value <- as.numeric(inp)
  total <- total + value
  count <- count + 1
}

average <- total / count
cat('Average:', average)
```

상기 프로그램에서, `count` 와 `sum` 변수를 사용해서 반복적으로 사용자가 숫자를 입력하면 값을 저장하고, 지금까지 사용자가 입력한 누적 합계를 계산한다. R 콘솔에서 `source()` 함수를 사용해서 실행한 결과는 다음과 같다.

```
> source("code/list_average.R")
Enter a number: 10
Enter a number: 20
Enter a number: 30
Enter a number: done
Average: 20
```

단순하게, 사용자가 입력한 각 숫자를 기억하고 내장함수를 사용해서 프로그램 마지막에 합계와 갯수를 계산한다.

```
##     : `list_datatype.R`
numlist <- list()

while (TRUE) {
  inp <- readline('Enter a number: ')
  if(inp == 'done') break

  value <- as.numeric(inp)
  numlist <- append(numlist, value)
}

num_vector <- unlist(numlist)
```

```
average <- sum(num_vector) / length(num_vector)
cat('List Average:', average)
```

루프가 시작되기 전 빈 리스트를 생성하고, 매번 숫자를 입력할 때마다 숫자를 리스트에 추가한다. 프로그램 마지막에 간단하게 리스트 총합을 계산하고, 평균을 산출하기 위해서 입력한 숫자 개수로 나눈다. R 콘솔에서 source() 함수를 사용해서 실행한 결과는 다음과 같다.

```
> source("code/list_datatype-average.R")
Enter a number: 40
Enter a number: 50
Enter a number: 60
Enter a number: done
List Average: 50
```

## 9.9 리스트와 문자열

문자열(string)은 **문자 순서(sequence)**이고, 리스트는 **값 순서(sequence)**이다. 하지만 리스트 문자는 문자열과 같지는 않다. 문자열을 리스트 문자로 변환하기 위해서, strsplit() 함수를 사용한다.

```
s <- 'spam'
t <- strsplit(s, NULL)[[1]] %>% strsplit(NULL)
t
```

```
## [[1]]
## [1] "s"
##
## [[2]]
## [1] "p"
##
## [[3]]
## [1] "a"
##
## [[4]]
## [1] "m"
```

list는 내장함수 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. l을 사용하면 1 처럼 보이기 때문에 가능하면 피한다. 그래서, t를 사용했다.

strsplit() 함수는 문자열을 구분자(이번 경우에는 NULL)를 사용해서 문자 각각으로 쪼갠다. 문자열 단어로 쪼개려면, 구분자를 바꿔 예를 들어 공백을 기준으로 쪼갠다.

```
s <- list('pining for the fjords')
t <- strsplit(s[[1]], " ")[[1]] %>% strsplit(" ")
t
```

```
## [[1]]
## [1] "pining"
##
## [[2]]
## [1] "for"
##
## [[3]]
## [1] "the"
##
## [[4]]
## [1] "fjords"
```

분할 함수를 사용해서 문자열을 리스트 토큰으로 쪼개면, 인덱스 연산자('[]')를 사용하여 리스트의 특정 단어를 볼 수 있다.

옵션 인자로 단어 경계로 어떤 문자를 사용할 것인지 지정하는데 사용되는 구분자(delimiter)를 활용하여 분할 `strsplit()` 함수를 호출한다. 다음 예제는 구분자로 하이픈('-')을 사용한 사례다.

```
s <- list('spam-spam-spam')
delimiter <- '-'
strsplit(s[[1]], delimiter)[[1]] %>% strsplit(" ")
```

```
## [[1]]
## [1] "spam"
##
## [[2]]
## [1] "spam"
##
## [[3]]
## [1] "spam"
```

합병 (paste) 함수는 분할 (strsplit) 함수의 역이다. 문자열 리스트를 받아 리스트 요소를 연결한다.

```
t <- list('pining', 'for', 'the', 'fjords')
delimiter <- ' '
paste0(t, delimiter, collapse = "")
```

```
## [1] "pining for the fjords "
```

상기의 경우, 구분자가 공백 문자여서 결합 (paste) 함수가 단어 사이에 공백을 넣는다. 공백없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

## 9.10 라인 파싱하기

파일을 읽을 때 통상, 단지 전체 라인을 출력하는 것 말고 뭔가 다른 것을 하고자 한다. 종종 "흥미로운 라인을" 찾아서 라인을 **파싱(parse)**하여 흥미로운 부분을 찾고자

한다. “From”으로 시작하는 라인에서 요일을 찾고자 하면 어떨까?

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

이런 종류의 문제에 직면했을 때, `stringr` 팩키지 분할 `str_split()` 함수가 매우 효과적이다. 작은 프로그램을 작성하여 “From”으로 시작하는 라인을 찾고 `str_split()` 함수로 파싱하고 라인의 흥미로운 부분을 출력한다.

```
fhand <- file('data/mbox-short.txt', "r")
for(line in readLines(fhand)) {
  line <- stringr::str_trim(line)
  if(!stringr::str_detect(line, "^From ")) next

  words <- stringr::str_split(line, " ")[[1]]
  cat(words[3], "\n")
}
```

`if` 문의 축약 형태를 사용하여 `next` 문을 `if`문과 동일한 라인에 놓았다. `if` 문 축약 형태는 `next` 문을 들여쓰기를 다음 라인에 한 것과 동일하다.

프로그램은 다음을 출력한다.

```
Sat
Fri
Fri
Fri
...
```

나중에, 매우 정교한 기술에 대해서 학습해서 정확하게 검색하는 비트(bit) 수준 정보를 찾아 내기 위해서 작업할 라인을 선택하고, 어떻게 해당 라인을 뽑아낼 것이다.

## 9.11 객체와 값(value)

다음 대입문을 실행하면,

```
a <- 'banana'
b <- 'banana'
```

`a` 와 `b` 모두 문자열을 참조하지만, 두 변수가 동일한 문자열을 참조하는지 알 수 없다. 두 가지 가능한 상태가 있다.

한 가지 경우는 `a` 와 `b`가 같은 값을 가지는 다른 두 객체를 참조하는 것이다. 두 번째 경우는 같은 객체를 참조하는 것이다.

두 변수가 동일한 객체를 참조하는지를 확인하기 위해서, 파이썬에서는 `is` 연산자가 사용된다.

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

```
>>> a is b
True
```

이 경우, 파이썬은 하나의 문자열 객체를 생성하고 a 와 b 모두 동일한 객체를 참조한다.

하지만, 리스트 두 개를 생성할 때, 객체가 두 개다.

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

상기의 경우, 두 개의 리스트는 동등하다고 말할 수 있다. 왜냐하면 동일한 요소를 가지고 있기 때문이다. 하지만, 같은 객체는 아니기 때문에 동일하지는 않다. 두 개의 객체가 동일하다면, 두 객체는 또한 등등하다. 하지만, 동등하다고 해서 반듯이 동일하지는 않다.

지금까지 "객체(object)"와 "값(value)"을 구분 없이 사용했지만, 객체가 값을 가진다라고 말하는 것이 좀더 정확하다. a = [1,2,3] 을 실행하면, a 는 특별한 순서 요소값을 갖는 리스트 객체로 참조한다. 만약 또 다른 리스트가 동일한 요소를 가진다면, 그 리스트는 같은 값을 가진다고 말한다.

R도 이와 유사하게 참조와 복사에 대해 동일한 개념을 활용하여 처리한다.

## 9.12 에일리어싱(Aliasing)

a가 객체를 참조하고, b <- a 대입하다면, 두 변수는 동일한 객체를 참조한다.

```
a <- list(1, 2, 3)
b <- a

a <- list(4,5,6)

identical(a, b)
```

```
## [1] FALSE
```

객체와 변수의 연관짓는 것을 **참조(reference)**라고 한다. 상기의 경우 동일한 객체에 두 개의 참조가 있다.

하나 이상의 참조를 가진 객체는 한개 이상의 이름을 갖게 되어서, 객체가 **에일리어스(aliased)** 되었다고 한다.

만약 에일리어스된 객체가 변경 가능하면, 변화의 여파는 다른 객체에도 파급된다.

이와 같은 행동이 유용하기도 하지만, 오류를 발생시키기도 쉽다. 일반적으로, **변경가능한 객체(mutable object)**로 작업할 때 에일리어싱을 피하는 것이 안전하다.

파이썬 문자열 같이 변경 불가능한 객체에 에일리어싱은 그렇게 문제가 되지 않는다.

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

상기 예제에서, a 와 b가 동일한 문자열을 참조하든 참조하지 않든 거의 차이가 없다.

## 9.13 디버깅

부주의한 리스트 사용이나 변경가능한 객체를 사용하는 경우 디버깅을 오래 할 수 있다. 다음에 일반적인 함정 유형과 회피하는 방법을 소개한다.

1. **관용구를 선택하고 고수하라.**

리스트와 관련된 문제 일부는 리스트를 가지고 할 수 있는 것이 너무 많다는 것이다.

2. **에일리어싱을 회피하기 위해서 사본 만들기.**

인자를 변경하는 정렬 (sort)같은 메쏘드를 사용하지만, 원 리스트도 보관되길 원한다면, 사본을 만든다.

```
orig <- t
t[t(unlist(t), decreasing=TRUE)]
```

상기 예제에서 원 리스트는 그대로 둔 상태로 새로 정렬된 리스트를 반환된 결과는 t에 저장한다. 하지만 이 경우에는, 변수명으로 sorted를 사용하는 것을 피해야 한다!

3. **리스트, 분할 (split), 파일**

파일을 읽고 파싱할 때, 프로그램이 중단될 수 있는 입력값을 마주할 수많은 기회가 있다. 그래서 파일을 훑어 "건초더미에서 바늘"을 찾는 프로그램을 작성할 때 사용한 가디언 패턴(guardian pattern)을 다시 살펴보는 것은 좋은 생각이다.

파일 라인에서 요일을 찾는 프로그램을 다시 살펴보자.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

각 라인을 단어로 나누었기 때문에, startswith를 사용하지 않고, 라인에 관심있는 단어가 있는지 살펴보기 위해서 단순하게 각 라인의 첫 단어를 살펴본다. 다음과 같이 continue 문을 사용해서 "From"이 없는 라인을 건너 뛴다.

```
fhand <- file('data/mbox-short.txt', "r")
for(line in readLines(fhand)) {
  if(!stringr::str_detect(line, "^From ")) next

  words <- stringr::str_split(line, " ")[[1]]
  cat(words[3], "\n")
}
```

프로그램이 훨씬 간단하고, 파일 끝에 있는 새줄(newline)을 제거하기 위해 str_trim() 함수를 사용할 필요도 없다. 하지만, 더 좋아졌는가?

작동하는 것 같지만, 경우에 따라서 첫줄에 Sat 를 출력하고 나서 오류로 프로그램이 정상 동작에 실패하는 경우도 있다. 무엇이 잘못되었을까? 어딘가 엉망이 된 데이터가 있어 우아하고, 총명하며, 매우 R스러운 프로그램을 망가뜨린건가?

오랜 동안 프로그램을 응시하고 머리를 짜내거나, 다른 사람에게 도움을 요청할 수 있지만, 빠르고 현명한 접근법은 `print()`문을 추가하는 것이다. `print()`문을 넣는 가장 좋은 장소는 프로그램이 동작하지 않는 라인 앞이 적절하고, 프로그램 실패를 야기할 것 같은 데이터를 출력한다.

이 접근법이 많은 라인을 출력하지만, 즉석에서 문제에 대해서 손에 잡히는 단서는 최소한 준다. 그래서 words를 출력하는 출력문을 5번째 라인 앞에 추가한다. "Debug:"를 접두어로 라인에 추가하여, 정상적인 출력과 디버그 출력을 구분한다.

```
fhand <- file('data/mbox-short.txt', "r")
for(line in readLines(fhand)) {
  line <- stringr::str_trim(line)
  cat("Debug", line, "\n")
  if(!stringr::str_detect(line, "^From ")) next

  words <- stringr::str_split(line, " ")[[1]]
  cat(words[3], "\n")
}
```

프로그램을 실행할 때, 많은 출력결과가 스크롤되어 화면 위로 지나간다. 마지막에 디버그 결과물과 역추적(traceback)을 보고 역추적(traceback) 바로 앞에서 무슨 일이 생겼는지 알 수 있다.

```
Debug: 'X-DSPAM-Confidence:', '0.8475'
Debug: 'X-DSPAM-Probability:', '0.0000'
Debug:
```

각 디버그 라인은 리스트 단어를 출력하는데, 라인을 분할 `str_split()` 함수를 활용하여 단어로 만들 때 얻어진다. 프로그램이 실패할 때 리스트 단어는 비었다 ''. 텍스트 편집기로 파일을 열어 살펴보면 그 지점은 다음과 같다.

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

프로그램이 빈 라인을 만났을 때, 오류가 발생한다. 물론, 빈 라인은 '0' 단어 ("zero words")다. 프로그램을 작성할 때, 왜 이것을 생각하지 못했을까? 첫 단어(word[1])가 "From"과 일치하는지 코드가 점검할 때, "인덱스 범위 오류(index out of range)"가 발생한다.

물론, 첫 단어가 없다면 첫 단어 점검을 회피하는 **가디언 코드(guardian code)**를 삽입하기 최적 장소이기는 하다. 코드를 보호하는 방법은 많다. 첫 단어를 살펴보기

전에 단어의 갯수를 확인하는 방법을 택한다.

```
fhand <- file('data/mbox-short.txt', "r")
for(line in readLines(fhand)) {
  line <- stringr::str_trim(line)
  # cat("Debug", line, "\n")
  if(length(words) == 0) next
  if(!stringr::str_detect(line, "^From ")) next

  words <- stringr::str_split(line, " ")[[1]]
  cat(words[3], "\n")
}
```

변경한 코드가 실패해서 다시 디버그할 경우를 대비해서, `print`문을 제거하는 대신에 `print`문을 주석 처리한다. 그리고 나서, 단어가 '0' 인지를 살펴보고 만약 '0' 이면, 파일 다음 라인으로 건너뛰도록 `next`문을 사용하는 가디언 문장(guardian statement)을 추가한다.

두 개 `next`문이 "흥미롭고" 좀더 처리가 필요한 라인 집합을 정제하도록 돕는 것으로 생각할 수 있다. 단어가 없는 라인은 "흥미 없어서" 다음 라인으로 건너뛴다. 첫 단어에 "From"이 없는 라인도 "흥미 없어서" 건너뛴다.

변경된 프로그램이 성공적으로 실행되어서, 아마도 올바르게 작성된 것으로 보인다. 가디언 문장(guardian statement)이 `words[1]`가 정상작동할 것이라는 것을 확인해 주지만, 충분하지 않을 수도 있다. 프로그램을 작성할 때, "무엇이 잘못 될 수 있을까?"를 항상 생각해야만 한다.

**연습문제:** 상기 프로그램의 어느 라인이 여전히 적절하게 보호되지 않은지를 생각해 보세요. 텍스트 파일을 구성해서 프로그램이 실패하도록 만들 수 있는지 살펴보세요. 그리고 나서, 프로그램을 변경해서 라인이 적절하게 보호되게 하고, 새로운 텍스트 파일을 잘 다룰 수 있도록 시험하세요.

**연습문제:** 두 `if` 문 없이, 상기 예제의 가디언 코드(guardian code)를 다시 작성하세요. 대신에 단일 `if`문과 `&` 논리 연산자를 사용하는 복합 논리 표현식을 사용하세요.

## 9.14 용어정의

- **에일리어싱(aliasing)**: 하나 혹은 그 이상의 변수가 동일한 객체를 참조하는 상황.
- **구분자(delimiter)**: 문자열이 어디서 분할되어져야 할지를 표기하기 위해서 사용되는 문자나 문자열.
- **요소(element)**: 리스트 혹은 다른 순서(sequence) 값의 하나로 항목(item)이라고도 한다.
- **동등한(equivalent)**: 같은 값을 가짐.
- **인덱스(index)**: 리스트의 요소를 지칭하는 정수 값.
- **동일한(identical)**: 동등을 함축하는 같은 객체임.
- **리스트(list)**: 순서(sequence) 값.

- **리스트 운행법(list traversal)**: 리스트의 각 요소를 순차적으로 접근함.
- **중첩 리스트(nested list)**: 또 다른 리스트의 요소인 리스트.
- **객체(object)**: 변수가 참조할 수 있는 무엇. 객체는 자료형(type)과 값(value)을 가진다.
- **참조(reference)**: 변수와 값의 연관.

## 9.15 연습문제

1. http://www.py4inf.com/code/romeo.txt에서 파일 사본을 다운로드 받으세요. `romeo.txt` 파일을 열어, 한 줄씩 읽어들이는 프로그램을 작성하세요. 각 라인마다 stringr 팩키지에서 분할 `str_split()` 함수를 사용하여 라인을 단어 리스트로 쪼개세요.

각 단어마다, 단어가 이미 리스트에 존재하는지를 확인하세요. 만약 단어가 리스트에 없다면, 리스트에 새 단어로 추가하세요.

프로그램이 완료되면, 알파벳 순으로 결과 단어를 정렬하고 출력하세요.

```
Enter file: romeo.txt
[1] 'Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder'
```

2. 전자우편 데이터를 읽어 들이는 프로그램을 작성하세요. "From"으로 시작하는 라인을 발견했을 때, stringr 팩키지에서 분할 `str_split()` 함수를 사용하여 라인을 단어로 쪼개세요. "From" 라인의 두번째 단어, 누가 메시지를 보냈는지에 관심이 있다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

"From" 라인을 파싱하여 각 "From"라인의 두번째 단어를 출력한다. 그리고 나서, "From:"이 아닌 "From"라인 갯수를 세고, 끝에 갯수를 출력한다.

여기 몇 줄을 삭제한 출력 예시가 있다.

```
Rscript fromcount.R
Enter a file name: mbox-short.txt

stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
```

```
cwen@iupui.edu
cwen@iupui.edu

There were 27 lines in the file with From as the first word
```

3. 사용자가 숫자 리스트를 입력하고, 입력한 숫자 중에 최대값과 최소값을 출력하고 사용자가 “done”을 입력할 때 종료하는 프로그램을 다시 작성하세요. 사용자가 입력한 숫자를 리스트에 저장하고, `max()` 과 `min()` 함수를 사용하여 루프가 끝나면, 최대값과 최소값을 출력하는 프로그램을 작성하세요.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

# Chapter 10

# 딕셔너리

**명칭을 갖는 리스트(named list)**는 딕셔너리로 더 잘 알려져 있고, **딕셔너리(dictionary)**는 리스트 같지만 좀더 일반적이다. 리스트에서 위치(인텍스)는 정수이지만, 딕셔너리에서는 인덱스는 임의 자료형(type)이 될 수 있다.

딕셔너리를 인덱스 집합(**키(keys)**라고 부름)에서 값(value) 집합으로 사상(mapping)하는 것으로 생각할 수 있다. 각각의 키는 값에 대응한다. 키와 값을 연관시키는 것을 **키-값 페어(key-value pair)**라고 부르고, 종종 항목(item)으로도 부른다.

한 예제로, 영어 단어에서 스페인 단어에 매핑되는 사전을 만들 것이다. 키와 값은 모두 문자열이다.

`list` 함수는 항목이 전혀 없는 리스트를 신규로 생성한다. `list`는 내장함수명이어서, 변수명으로 사용하는 것을 피해야 한다.

```
eng2kr <- list()
eng2kr
```

```
## list()
```

`list()`는 빈 리스트임을 나타낸다. 리스트에 신규 요소를 추가하기 위해서 `list()` 함수 내부에 `' '='  '`과 같이 명칭과 값을 지정한다.

```
eng2kr <- list('one' = ' ')
```

상기 코드는 키(명칭) `'one'`에서 값 `' '`를 매핑하는 항목을 생성한다. 명칭을 갖는 리스를 다시 출력하면, 키-값 페어(key-value pair)를 볼 수 있다.

```
eng2kr
```

```
## $one
## [1] " "
```

다수 키-값을 갖는 명칭을 갖는 리스트를 제작할 경우 순차적으로 작성하고 `list()`로 감싼다.예를 들어, 세개 항목을 가진 명칭을 갖는 리스트를 생성할 수

있다.

```
eng2kr <- list('one' = ' ',
               'two' = ' ',
               'three' = ' ')
```

eng2sp을 출력하면 다음과 같다.

```
eng2kr
```

```
## $one
## [1] " "
##
## $two
## [1] " "
##
## $three
## [1] " "
```

명칭을 갖는 리스트에서 키를 사용해서 상응하는 값을 찾을 수 있다.

```
eng2kr['one']
```

```
## $one
## [1] " "
```

'two' 키는 항상 값 ' '에 상응되어서 명칭을 갖는 리스트 항목 순서는 문제가 되지 않는다.

만약 키가 리스트에 존재하지 않으면, NULL 값을 반환한다.

```
eng2kr['four']
```

```
## $<NA>
## NULL
```

length() 함수를 리스트에 사용하여 키-값 페어(key-value pair) 항목 개수를 파악할 수 있다.

```
length(eng2kr)
```

```
## [1] 3
```

%in% 연산자는 명칭을 갖는 리스트에 키(Key, 명칭)이 있는지 알려준다. %in% 연산자는 각 항목마다 키(명칭)가 있는지 참/거짓으로 알려주기 때문에 any()와 결합해서 사용하게 되면 리스트에 키가 있는지 없는지만 확인할 때 요긴하다.

```
names(eng2kr) %in% "one" %>% any()
```

```
## [1] TRUE
```

비교를 위해서 리스트에 없는 키(명칭) five를 확인해보자.

```
names(eng2kr) %in% "five" %>% any()
```

```
## [1] FALSE
```

이번에는 리스트에 값이 있는지를 확인해보자.

```
eng2kr %in% " "
```

```
## [1] FALSE  TRUE FALSE
```

"둘" 값을 갖는 항목이 있는지를 `%in%` 연산자를 사용해서 확인했다.

```
eng2kr %in% c(" ", " ")
```

```
## [1] FALSE  TRUE  TRUE
```

조금 확장해서 "둘", "셋"이 있는지도 없는지도 쉽게 확인할 수 있다.

### 10.0.1 연습문제

`words.txt` 단어를 읽어서 명칭을 갖는 리스트에 키로 저장하는 프로그램을 작성하세요. 값이 무엇이든지 상관없습니다. 리스트에 문자열을 확인하는 가장 빠른 방법으로 명칭을 확인할 경우 `names()` 함수와 값을 확인할 경우 그냥 `%in%` 연산자와 조합하여 사용할 수 있습니다.

## 10.1 계수기 집합으로 리스트

문자열이 주어진 상태에서, 각 문자가 얼마나 나타나는지를 센다고 가정합시다. 몇 가지 방법이 아래에 있습니다.

1. 26개 변수를 알파벳 문자 각각에 대해 생성한다. 그리고 나서 아마도 연쇄 조건문을 사용하여 문자열을 훑고 해당 계수기(counter)를 하나씩 증가한다.
2. 26개 요소를 가진 리스트를 생성한다. 리스트 안에 인덱스로 숫자를 사용해서 적당한 계수기(counter)를 증가한다.
3. 키(key)로 문자, 계수기(counter)로 해당 값(value)을 갖는 리스트를 생성한다. 처음 문자를 만나면, 딕셔너리에 항목으로 추가한다. 추가한 후에는 기존 항목 값을 증가한다.

상기 3개 옵션은 동일한 연산을 수행하지만, 각각은 다른 방식으로 연산을 구현한다.

**구현(implementation)**은 연산(computation)을 수행하는 방법이다. 어떤 구현 방법이 다른 것 보다 낫다. 다음에 명칭을 갖는 리스트로 구현한 코드가 있다.

```
word <- 'brontosaurus'
word_split <- strsplit(word, "")[[1]]

n_list <- list()
```

```
for(char in 1:length(word_split)) {

  if( word_split[char]  %in% names(n_list) ) { #       ( )
    n_list[[ word_split[char] ]] <- n_list[[ word_split[char] ]] + 1
  } else { #       ( )
    temp_list <- list()
    temp_list[[word_split[char]]] <- 1
    n_list <- append(n_list, temp_list)
  }
}

# n_list
```

계수기(counter) 혹은 빈도에 대한 통계 용어인 **히스토그램(histogram)**을 효과적으로 산출할 수 있다.

`for` 루프는 문자열을 훑는다. 매번 루프를 반복할 때마다 리스트에 문자 `c`가 없다면, 키 `c`와 초기값 1을 가진 새로운 항목을 생성한다. 문자 `c`가 이미 리스트에 존재한다면, `n_list[['c']]`을 증가한다.

다음 프로그램 실행 결과가 있다.

```
n_list
```

```
## $b
## [1] 1
##
## $r
## [1] 2
##
## $o
## [1] 2
##
## $n
## [1] 1
##
## $t
## [1] 1
##
## $s
## [1] 2
##
## $a
## [1] 1
##
## $u
## [1] 2
```

히스토그램은 문자 'a', 'b'는 1회, 'o'는 2회 등등 나타남을 보여준다.

R은 태생에 통계를 근간으로 하기 때문에 빈도수를 구하거나 하는 문제를 아주 쉽고 간결하게 작성할 수 있다. 앞선 for, if 문을 명칭이 있는 리스트 자료구조를 이용해서 길게 작성했지만, table() 함수를 사용하면 훨씬 간결하게 동일한 효과를 낼 수 있다.

```
word <- 'brontosaurus'
word_split <- strsplit(word, "")[[1]]

table(word_split) %>%
  as.list()
```

```
## $a
## [1] 1
##
## $b
## [1] 1
##
## $n
## [1] 1
##
## $o
## [1] 2
##
## $r
## [1] 2
##
## $s
## [1] 2
##
## $t
## [1] 1
##
## $u
## [1] 2
```

시간을 가지고서 잠시 if 문과 in 연산자를 사용한 루프와 적절한 전처리 과정을 거쳐 자료형을 맞추고 나서 table() 함수를 사용한 방식을 비교해 보세요. 동일한 연산을 수행하지만, 하나가 더 간결한다.

## 10.2 리스트와 파일

딕셔너리의 흔한 사용법 중의 하나는 텍스트로 작성된 파일에 단어 빈도를 세는 것이다. http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html 사이트 덕분에 로미오와 쥴리엣(Romeo and Juliet) 텍스트 파일에서 시작합시다.

처음 연습으로 구두점이 없는 짧고 간략한 텍스트 버젼을 사용한다. 나중에 구두점이 포함된 전체 텍스트로 작업할 것이다.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

파일 라인을 읽고, 각 라인을 단어 리스트로 쪼개고, 루프를 돌려 사전을 이용하여 각 단어의 빈도수를 세는 R 프로그램을 작성한다.

두 개의 `for` 루프를 사용한다. 외곽 루프는 파일 라인을 읽고, 내부 루프는 특정 라인의 단어 각각에 대해 반복한다. 하나의 루프는 외곽 루프가 되고, 또 다른 루프는 내부 루프가 되어서 **중첩 루프(nested loops)**라고 불리는 패턴 사례다.

외곽 루프가 한번 반복을 할 때마다 내부 루프는 모든 반복을 수행하기 때문에 내부 루프는 "좀더 빨리" 반복을 수행하고 외곽 루프는 좀더 천천히 반복을 수행하는 것으로 생각할 수 있다.

두 중첩 루프의 조합이 입력 파일의 모든 라인에 있는 모든 단어의 빈도를 계수(count)하는 것을 보증한다.

중첩루프를 돌려 단어 빈도수를 계산하는 것도 가능하지만 R의 강력한 내장함수를 활용하여 간결하게 다음과 같이 작성할 수도 있다.

```
romeo_text <- "But soft what light through yonder window breaks It is the east and Juli

romeo_split <- stringr::str_split(romeo_text, " ")[[1]]

romeo_freq <- romeo_split %>%
  table() %>%
  as.list()
```

프로그램을 실행하면, 정렬되지 않은 해쉬 순서로 모든 단어의 빈도수를 출력합니다. `romeo.txt` 파일은 www.py4inf.com/code/romeo.txt에서 다운로드 가능하다. 다운로드 받은 `romeo.txt` 파일을 로컬 파일에 저장한 후에 파일명을 읽어 실행하는 코드를 작성하여 실행하면 다음과 같은 결과를 확인할 수 있다.

이를 위해서 앞서 작성한 코드를 다음과 같이 사용자 입력을 받아 처리할 수 있도록 `count1.R` 파일에 저장시킨다.

```
## script/count1.R

library(tidyverse)

cat("      ?")

input_filename <- readLines("stdin", n=1)

romeo_text <- readr::read_lines(input_filename)
```

```
romeo_split <- stringr::str_split(romeo_text, " ")

romeo_freq <- romeo_split %>% unlist() %>%
  table() %>%
  as.list()

romeo_freq
```

상기 코드를 쉘에서 Rscript 명령어로 실행하게 되면 `romeo.txt` 파일에 담긴 단어 빈도수를 계산할 수 있게 된다.

```
$ Rscript --vanilla script/count1.R

      ? data/romeo.txt
$already
[1] 1

$and
[1] 3

$Arise
[1] 1

$breaks
[1] 1

$But
[1] 1

$east
[1] 1

$envious
[1] 1

$fair
[1] 1

$grief
[1] 1

$is
[1] 3

$It
```

```
[1] 1

$Juliet
[1] 1

$kill
[1] 1

$light
[1] 1

$moon
[1] 1

$pale
[1] 1

$sick
[1] 1

$soft
[1] 1

$sun
[1] 2

$the
[1] 3

$through
[1] 1

$what
[1] 1

$Who
[1] 1

$window
[1] 1

$with
[1] 1

$yonder
```

```
[1] 1
```

가장 높은 빈도 단어와 빈도수를 찾기 위해서 리스트를 훑는 것이 불편하다. 좀더 도움이 되는 출력결과를 만들려고 코드를 바꿔보자.

## 10.3 반복과 리스트

for문에 순서(sequence)로서 리스트를 사용한다면, 리스트 키를 훑는다. 루프는 각 키와 해당 값을 출력한다.

```
counts <- list('chuck' = 1 ,
               'annie' = 42,
               'jan'   = 100)

for( count in seq_along(counts) ) {
  cat( names(counts)[count], ":", counts[[names(counts)[count]]], "\n")
}
```

출력은 다음과 같다.

```
## chuck : 1
## annie : 42
## jan : 100
```

이 패턴을 사용해서 앞서 기술한 다양한 루프 숙어를 구현한다. 예를 들어, 리스트에서 10 보다 큰 값을 가진 항목을 모두 찾고자 한다면, 다음과 같이 코드를 작성한다.

```
counts <- list('chuck' = 1 ,
               'annie' = 42,
               'jan'   = 100)

for( count in seq_along(counts) ) {
  if( counts[[names(counts)[count]]] > 10 ) {
    cat( names(counts)[count], ":", counts[[names(counts)[count]]], "\n")
  }
}
```

for 루프는 딕셔너리 키(keys)를 반복한다. 그래서, 인덱스 연산자를 사용해서 각 키에 상응하는 값(value)을 가져와야 한다. 여기 출력값이 있다.

```
## annie : 42
## jan : 100
```

10 이상 값만 가진 항목만 볼 수 있다.

알파벳 순으로 키를 출력하고자 한다면, 리스트 객체에서 이름을 따로 추출해서 알파벳순서로 정렬한다. 그리고 이를 리스트 객체에 반영하여 정렬된 명칭이

있는 리스트를 준비한다.아래와 같이 정렬된 순서로 키/값 페어(key/value pair)를 출력한다.

```
counts <- list('chuck' = 1 ,
               'annie' = 42,
               'jan'   = 100)

name_sorted <- sort(names(counts))
counts <- counts[name_sorted]

for( count in seq_along(counts) ) {
    cat( names(counts)[count], ":", counts[[names(counts)[count]]], "\n")
}
```

다음에 출력결과가 있다.

```
## annie : 42
## chuck : 1
## jan : 100
```

첫 명칭이 있는 리스트는 정렬되지 않은 키 리스트였다면, `for` 루프로 정렬된 키/값 페어(key/value pair)를 확인할 수 있다.

## 10.4 고급 텍스트 파싱{named-list-advanced}

`romeo.txt` 파일을 사용한 상기 예제에서, 수작업으로 모든 구두점을 제거해서 가능한 단순하게 만들었다. 실제 텍스트는 아래 보여지는 것처럼 많은 구두점이 있다.

```
    But, soft! what light through yonder window breaks?
    It is the east, and Juliet is the sun.
    Arise, fair sun, and kill the envious moon,
    Who is already sick and pale with grief,
```

R `stringr` 패키지 `str_split()` 함수는 공백을 찾고 공백으로 구분되는 토큰으로 단어를 처리하기 때문에, "soft!" 와 "soft"는 다른 단어로 처리되고 각 단어에 대해서 별도 딕셔너리 항목을 생성한다.

파일에 대문자가 있어서, "who"와 "Who"를 다른 단어, 다른 빈도수를 가진 것으로 처리한다.

`stringr` 패키지 `str_to_lower`, `str_squish`, `str_replace_all`, 문자열 함수를 사용해서 상기 문제를 해결할 수 있다. `str_replace_all` 함수가 가장 적합하다. `str_replace_all` 함수에 대한 문서가 다음에 있다.

```
str_replace_all(string, pattern, replacement)
```

`pattern` 매개변수를 사용해서 모든 구두점을 삭제할 수 있다. "구두점"으로 간주되는 문자 리스트는 `[[:punct:]]`에 정의되어 있어 별도 `~!@#$%^&*(){}_+:\"<>?,./;'[]-=`와

같이 지정하지 않아도 된다. replacement에는 삭제 혹은 교체 문자를 지정하면 된다.

프로그램을 다음과 같은 수정을 한다.

```
library(tidyverse)

cat("      ?")

input_filename <- readLines("stdin", n=1)

romeo_text <- readr::read_lines(input_filename)

romeo_split <- stringr::str_split(romeo_text, " ") %>%
  unlist() %>%
  stringr::str_to_lower() %>%
  stringr::str_replace_all(., pattern = "[[:punct:]]",
                               replacement = "")

romeo_freq <- romeo_split %>% unlist() %>%
  table() %>%
  as.list()

romeo_freq
```

str_replace_all 함수를 사용해서 모든 구두점을 제거했고, str_to_lower 함수를 사용해서 라인을 소문자로 수정했다. 나머지 프로그램은 변경된 것이 없다.

상기 프로그램을 실행한 출력결과는 다음과 같다.

```
$ $ Rscript.exe script/count2.R

      ? data/romeo.txt
$already
[1] 1

$and
[1] 3

$arise
[1] 1

$breaks
[1] 1

$but
[1] 1
```

```
$east
[1] 1

$envious
[1] 1

$fair
[1] 1

$grief
[1] 1

$is
[1] 3

$it
[1] 1

$juliet
[1] 1

$kill
[1] 1

$light
[1] 1

$moon
[1] 1

$pale
[1] 1

$sick
[1] 1

$soft
[1] 1

$sun
[1] 2

$the
[1] 3
```

```
$through
[1] 1

$what
[1] 1

$who
[1] 1

$window
[1] 1

$with
[1] 1

$yonder
[1] 1
```

출력결과는 여전히 다루기 힘들어 보입니다. R 프로그래밍을 통해 정확히 찾고자는 하는 것을 찾았으나 R **튜플(tuples)**(다양한 자료형을 갖는 리스트)에 대해서 학습할 필요성을 느껴진다. 튜플을 학습할 때, 다시 이 예제를 살펴볼 것이다.

## 10.5 디버깅 {r-dicionaries-debugging}

점점 더 큰 데이터로 작업함에 따라, 수작업으로 데이터를 확인하거나 출력을 통해서 디버깅을 하는 것이 어려울 수 있다. 큰 데이터를 디버깅하는 몇가지 방법이 있다.

1. **입력값을 줄여라(Scale down the input)**

   가능하면, 데이터 크기를 줄여라. 예를 들어, 프로그램이 텍스트 파일을 읽는다면, 첫 10줄로 시작하거나, 찾을 수 있는 작은 예제로 시작하라. 데이터 파일을 편집하거나, 프로그램을 수정해서 첫 `n` 라인만 읽도록 프로그램을 변경하라.

   오류가 있다면, `n`을 줄여서 오류를 재현하는 가장 작은 값으로 만들어라. 그리고 나서, 오류를 찾고 수정해 나감에 따라 점진적으로 늘려나가라.

2. **요약값과 자료형을 확인하라(Check summaries and types)**

   전체 데이터를 출력하고 검증하는 대신에 데이터를 요약하여 출력하는 것을 생각하라: 예를 들어, 딕셔너리 항목의 숫자 혹은 리스트 숫자의 총계

   실행 오류(runtime errors)의 일반적인 원인은 올바른 자료형(right type)이 아니기 때문이다. 이런 종류의 오류를 디버깅하기 위해서, 종종 값의 자료형을 출력하는 것으로 충분하다.

3. **자가 진단 작성(Write self-checks)**

종종 오류를 자동적으로 검출하는 코드를 작성한다. 예를 들어, 리스트 숫자의 평균을 계산한다면, 결과값은 리스트의 가장 큰 값보다 클 수 없고, 가장 작은 값보다 작을 수 없다는 것을 확인할 수 있다. "완전히 비상식적인" 결과를 탐지하기 때문에 "건전성 검사(sanity check)"라고 부른다.

또 다른 검사법은 두가지 다른 연산의 결과를 비교해서 일치하는지 살펴보는 것이다. "일치성 검사(consistency check)"라고 부른다.

4. **고급 출력(Pretty print the output)**

   디버깅 출력결과를 서식화하는 것은 오류 발견을 용이하게 한다.

다시 한번, 발판(scaffolding)을 만드는데 들인 시간이 디버깅에 소비되는 시간을 줄일 수 있다.

## 10.6 용어정의

- **명칭있는 리스트/딕셔너리(dictionary)**: 키(key)에서 해당 값으로 매핑(mapping)
- **해쉬테이블(hashtable)**: 파이썬 딕셔너리를 구현하기 위해 사용된 알고리즘
- **해쉬 함수(hash function)**: 키에 대한 위치를 계산하기 위해서 해쉬테이블에서 사용되는 함수
- **히스토그램(histogram)**: 계수기(counter) 집합.
- **구현(implementation)**: 연산(computation)을 수행하는 방법
- **항목(item)**: 키-값 페어(key-value pair)에 대한 또 다른 이름.
- **키(key)**: 키-값 페어(key-value pair)의 첫번째 부분으로 딕셔너리에 나타나는 객체.
- **키-값 페어(key-value pair)**: 키에서 값으로 매핑 표현.
- **룩업(lookup)**: 키를 가지고 해당 값을 찾는 딕셔너리 연산.
- **중첩 루프(nested loops)**: 루프 "내부"에 하나 혹은 그 이상의 루프가 있음. 외곽 루프가 1회 실행될 때, 내부 루프는 전체 반복을 완료함.
- **값(value)**:키-값 페어(key-value pair)의 두번째 부분으로 딕셔너리에 나타나는 객체. 앞에서 사용한 단어 "값(value)" 보다 더 구체적이다.

## 10.7 연습문제

1. 커밋(commit)이 무슨 요일에 수행되었는지에 따라 전자우편 메세지를 구분하는 프로그램을 작성하세요. "From"으로 시작하는 라인을 찾고, 3번째 단어를 찾아서 요일별 횟수를 계수(count)하여 저장하세요. 프로그램 끝에 딕셔너리 내용을 출력하세요. (순서는 문제가 되지 않습니다.)

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Sample Execution:
Rscript --vanilla dow.R
Enter a file name: mbox-short.txt
```

```
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

2. 전자우편 로그(log)를 읽고, 히스토그램을 생성하는 프로그램을 작성하세요. 딕셔너리를 사용해서 전자우편 주소별로 얼마나 많은 전자우편이 왔는지를 계수(count)하고 딕셔너리를 출력합니다.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

4. 상기 프로그램에 누가 가장 많은 전자우편 메시지를 가졌는지 알아내는 코드를 추가하세요.

결국, 모든 데이터를 읽고, 딕셔너리를 생성한다. 최대 루프를 사용해서 딕셔너리를 훑어서 누가 가장 많은 전자우편 메시지를 갖는지, 그리고 그 사람이 얼마나 많은 메시지를 갖는지 출력한다.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

5. 다음 프로그램은 주소 대신에 도메인 이름을 기록한다. 누가 메일을 보냈는지 대신(즉, 전체 전자우편 주소) 메시지가 어디에서부터 왔는지 출처를 기록한다. 프로그램 마지막에 딕셔너리 내용을 출력한다.

```
Rscript --vanilla schoolcount.R
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

# Chapter 11

# 튜플(Tuples)

파이썬 튜플은 다양한 자료형을 갖는 리스트에 대응된다. 튜플이 갖는 불변성이 R에서 데이터 분석 등의 작업에 꼭 필요한 기능은 아니라서 불변성을 활용한 튜플 개념이 많이 사용되지는 않고 있다.

하지만, 다양한 자료형을 갖는 리스트는 데이터프레임을 확장한 개념으로 고급 데이터 분석과 처리에 필히 이해를 가져야하는 내용이다.

추후 다양한 자료형을 갖는 리스트는 별도 장을 따로 만들어 풀어나갈 예정이다.

## 11.1 튜플은 불변이다

다양한 자료형을 갖는 리스트가 **튜플(tuple)**이다. **튜플(tuple)**[^about-tuple]은 리스트와 마찬가지로 순서(sequence) 값이다. 튜플에 저장된 값은 임의 자료형(type)이 될 수 있고, 정수로 색인 된다. 일반적으로 알려진 튜플은 **불변(immutable**하다는 것이다. 튜플은 **비교 가능(comparable)**하고 **해쉬형(hashable)**이다. 따라서, 리스트 값을 정렬할 수 있고, 키 값으로 튜플을 사용할 수 있다.

[^about-tuple] : 재미난 사실: 단어 "튜플(tuple)"은 가변 길이 (한배, 두배, 세배, 네배, 다섯배, 여섯배, 일곱배 등) 숫자열에 붙여진 이름에서 유래한다

구문론적으로, 튜플은 콤마로 구분되는 서로 다른 자료형을 갖는 리스트 값이다.

```
t <- list('a', 'b', 'c', 'd', 'e')
```

`(tuple)`이 생성자 이름이기 `list`이기 때문에 변수명으로 리스트(list) 사용을 피해야 한다.

대부분의 리스트 연산자는 튜플에서도 사용 가능하다. 꺾쇠 연산자가 요소를 색인한다.

```
t[1]
```

```
## [[1]]
## [1] "a"
```

그리고, 슬라이스 연산자(slice operator)는 요소 범위를 선택한다.

```
t[2:3]
```

```
## [[1]]
## [1] "b"
##
## [[2]]
## [1] "c"
```

하지만, 파이썬 튜플 요소 중 하나를 변경하고 하면 오류가 발생하지만, R에서 리스트는 다양한 자료형을 담을 수 있고 변경도 가능하다.

```
t[1] <- 'A'
t
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] "c"
##
## [[4]]
## [1] "d"
##
## [[5]]
## [1] "e"
```

## 11.2 가장 빈도수 높은 단어

로미오와 쥴리엣 2장 2막 텍스트 파일(romeo-full.txt)로 다시 돌아와서, 텍스트에서 가장 빈도수가 높은 단어를 10개를 출력하는 프로그램을 작성한다.

```
library(tidyverse)

romeo_text <- readr::read_lines("data/romeo-full.txt")

romeo_split <- stringr::str_split(romeo_text, " ") %>%
  unlist() %>%
  stringr::str_to_lower() %>%
```

```
  stringr::str_replace_all(., pattern = "[[:punct:]]",
                           replacement = "")

romeo_split <- romeo_split[romeo_split != ""]

romeo_full_freq <- romeo_split %>% unlist() %>%
  table() %>%
  as_tibble() %>%
  set_names(c("key", "value")) %>%
  arrange(desc(value)) %>%
  slice_head(n=10)

# romeo_full_freq
```

파일을 읽고 각 단어를 문서의 단어 빈도수는 전처리 로직으로 구두점을 제거하고 모두 소문자로 변환한 후에 `table()` 함수로 단어별 빈도수를 계산하고 데이터프레임으로 변환 후 고빈도 단어 10개를 추출한다.

이제 단어 빈도 분석을 위해서 작성한 프로그램의 마지막 출력결과는 원하는 바를 완수한 것처럼 보인다.

```
## # A tibble: 10 x 2
##   key    value
##   <chr>  <int>
## 1 i         61
## 2 and       42
## 3 romeo     40
## 4 the       34
## 5 to        34
## 6 juliet    32
## # ... with 4 more rows
```

## 11.3 순열: 문자열, 리스트, 튜플

여기서 리스트 튜플에 초점을 맞추었지만, 이장의 거의 모든 예제가 또한 리스트의 리스트, 튜플의 튜플, 리스트 튜플에도 동작한다. 가능한 모든 조합을 열거하는 것을 피하기 위해서, 순열의 순열(sequences of sequences)에 대해서 논의하는 것이 때로는 쉽다.

대부분의 문맥에서 다른 종류의 순열(문자열, 리스트, 튜플)은 상호 호환해서 사용될 수 있다. 그런데 왜 그리고 어떻게 다른 것보다 이것을 선택해야 될까?

서로 다른 자료형이 갖는 특징을 잘 파악하고 주어진 문제를 가장 쉽게 풀수 있는 방향의 자료형을 선택한다.

## 11.4 디버깅

리스트, 딕셔너리, 튜플은 **자료 구조(data structures)**로 일반적으로 알려져 있다. 이번장에서 리스트 튜플, 키로 튜플, 값으로 리스트를 담고 있는 딕셔너리 같은 복합 자료 구조를 보기 시작했다. 복합 자료 구조는 유용하지만, 저자가 작명한 **형태 오류(shape errors)**라고 불리는 오류에 노출되어 있다. 즉, 자료 구조가 잘못된 자료형(type), 크기, 구성일 경우 오류가 발생한다. 혹은 코드를 작성하고, 자료의 모양이 생각나지 않는 경우도 오류의 원인이 된다.

예를 들어, 정수 하나를 가진 리스트를 기대하고, (리스트가 아닌) 일반 정수를 넘긴다면, 작동하지 않는다.

프로그램을 디버깅할 때, 정말 어려운 버그를 잡으려고 작업을 한다면, 다음 네가지를 시도할 수 있다.

1. **코드 읽기(reading)**: 코드를 면밀히 조사하고, 스스로에게 다시 읽어 주고, 코드가 자신이 작성한 의도를 담고 있는지 점검하라.
2. **실행(running)**: 변경해서 다른 버젼을 실행해서 실험하라. 종종, 프로그램이 적절한 곳에 적절한 것을 보여준다면, 문제가 명확하다. 발판(scaffolding)을 만들기 위해서 때때로 시간을 들일 필요도 있다.
3. **반추(ruminating)**:생각의 시간을 갖자. 어떤 종류의 오류인가: 구문, 실행, 의미론(semantic). 오류 메시지로부터 혹은 프로그램 출력결과로부터 무슨 정보를 얻을 수 있는가? 어떤 종류 오류가 지금 보고 있는 문제를 만들었을까? 문제가 나타나기 전에, 마지막으로 변경한 것은 무엇인가?
4. **퇴각(retreating)**: 어느 시점에선가, 최선은 물러서서, 최근의 변경을 다시 원복하는 것이다. 잘 동작하고 이해하는 프로그램으로 다시 돌아가서, 다시 프로그램을 작성한다.

초보 프로그래머는 종종 이들 활동 중 하나에 사로잡혀 다른 것을 잊곤 한다. 활동 각각은 고유한 실패 방식과 함께 온다.

예를 들어, 프로그램을 정독하는 것은 문제가 인쇄상의 오류에 있다면 도움이 되지만, 문제가 개념상 오해에 뿌리를 두고 있다면 그다지 도움이 되지 못한다. 만약 작성한 프로그램을 이해하지 못한다면, 100번 읽을 수는 있지만, 오류를 발견할 수는 없다. 왜냐하면, 오류는 여러분 머리에 있기 때문입니다.

만약 작고 간단한 테스트를 진행한다면, 실험을 수행하는 것이 도움이 될 수 있다. 하지만, 코드를 읽지 않거나, 생각없이 실험을 수행한다면, 프로그램이 작동될 때까지 무작위 변경하여 개발하는 "랜덤 워크 프로그램(random walk programming)" 패턴에 빠질 수 있다. 말할 필요없이 랜덤 워크 프로그래밍은 시간이 오래 걸린다.

생각할 시간을 가져야 한다. 디버깅은 실험 과학 같은 것이다. 문제가 무엇인지에 대한 최소한 한 가지 가설을 가져야 한다. 만약 두개 혹은 그 이상의 가능성이 있다면, 이러한 가능성 중에서 하나라도 줄일 수 있는 테스트를 생각해야 한다.

휴식 시간을 가지는 것은 생각하는데 도움이 된다. 대화를 하는 것도 도움이 된다. 문제를 다른 사람 혹은 자신에게도 설명할 수 있다면, 질문을 마치기도 전에 답을

종종 발견할 수 있다.

하지만, 오류가 너무 많고 수정하려는 코드가 매우 크고, 복잡하다면 최고의 디버깅 기술도 무용지물이다. 가끔, 최선의 선택은 퇴각하는 것이다. 작동하고 이해하는 곳까지 후퇴해서 프로그램을 간략화하라.

초보 프로그래머는 종종 퇴각하기를 꺼려한다. 왜냐하면, 설사 잘못되었지만, 한줄 코드를 지울 수 없기 때문이다. 삭제하지 않는 것이 기분을 좋게 한다면, 다시 작성하기 전에 프로그램을 다른 파일에 복사하라. 그리고 나서, 한번에 조금씩 붙여넣어라.

정말 어려운 버그(hard bug)를 발견하고 고치는 것은 코드 읽기, 실행, 반추, 때때로 퇴각을 요구한다. 만약 이들 활동 중 하나도 먹히지 않는다면, 다른 것들을 시도해 보세요.

## 11.5 용어정의

- **비교가능한(comparable)**:동일한 자료형의 다른 값과 비교하여 큰지, 작은지, 혹은 같은지를 확인하기 위해서 확인할 수 있는 자료형(type). 비교가능한(comparable) 자료형은 리스트에 넣어서 정렬할 수 있다.
- **자료 구조(data structure)**:연관된 값의 집합, 종종 리스트, 딕셔너리, 튜플 등으로 조직화된다.
- **DSU**: "decorate-sort-undecorate,"의 약어로 리스트 튜플을 생성, 정렬, 결과 일부 추출을 포함하는 패턴.
- **모음(gather)**: 가변-길이 인자 튜플을 조합하는 연산.
- **해쉬형(hashable)**: 해쉬 함수를 가진 자료형(type). 정수, 소수점, 문자열 같은 불변형은 해쉬형이다. 리스트나 딕셔너리 처럼 변경가능한 형은 해쉬형이 아니다.
- **스캐터(scatter)**: 순서(sequence)를 리스트 인자로 다루는 연산.
- **(자료 구조의) 형태**: 자료 구조의 자료형(type), 크기, 구성을 요약.
- **싱글톤(singleton)**: 단일 요소를 가진 리스트 (혹은 다른 순서(sequence)).
- **튜플(tuple)**: 불변 요소들의 순서 (sequence).
- **튜플 대입(tuple assignment)**: 오른편 순서(sequence)와 왼편 튜플 변수를 대입. 오른편이 평가되고나서 각 요소들은 왼편의 변수에 대입된다.

## 11.6 연습문제

1. 앞서 작성한 프로그램을 다음과 같이 수정하세요. "From"라인을 읽고 파싱하여 라인에서 주소를 뽑아내세요. 딕셔너리를 사용하여 각 사람으로부터 메시지 숫자를 계수(count)한다.

2. 모든 데이터를 읽은 후에 가장 많은 커밋(commit)을 한 사람을 출력하세요. 딕셔너리로부터 리스트 (count, email) 튜플을 생성하고 역순으로 리스트를 정렬한 후에 가장 많은 커밋을 한 사람을 출력하세요.

```
Sample Line:
```

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

3. 이번 프로그램은 각 메시지에 대한 하루 중 시간의 분포를 계수(count)한다. “From” 라인으로부터 시간 문자열을 찾고 콜론(:) 문자를 사용하여 문자열을 쪼개서 시간을 추출합니다. 각 시간별로 계수(count)를 누적하고 아래에 보여지듯이 시간 단위로 정렬하여 한 라인에 한시간씩 계수(count)를 출력합니다.

```
Sample Execution:
Rscript timeofday.R
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

4. 파일을 읽고, 빈도(frequencey)에 따라 내림차순으로 문자(letters)를 출력하는 프로그램을 작성하세요. 작성한 프로그램은 모든 입력을 소문자로 변환하고 a-z 문자만 계수(count)한다. 공백, 숫자, 문장기호 a-z를 제외한 다른 어떤 것도 계수하지 않습니다. 다른 언어로 구성된 텍스트 샘플을 구해서 언어마다 문자 빈도가 어떻게 변하는지 살펴보세요. 결과를 wikipedia.org/wiki/Letter_frequencies 표와 비교하세요.

# Chapter 12

# 데이터프레임

다른 프로그래밍 언어에서 다루지 않는 독특한 자료구조가 데이터프레임(Dataframe)이다. R도 프로그래밍 언어이기 때문에 다른 언어에서 갖고 있는 자료구조를 대부분 갖추고 있지만, 데이터분석, 시각화, 모형개발 등에 꼭 필요한 기본 자료구조가 데이터프레임이다. 데이터프레임을 본격적으로 살펴보기 전에 통계학에서 다루는 측정에 대해 살펴보고, 측정 척도(scale)에 대한 이론적 배경을 이해한다. 척도의 개념에 대응되는 R 자료구조를 통해 데이터프레임과 추후 프로그래밍 과정에서 많이 다뤄지는 리스트에 대한 차별점도 살펴보자.

## 12.1 측정 변수의 구분 [1] [2] [3]

자료가 갖는 고유한 특성을 숫자료 표현한 측정 척도에 따라, 명목형, 순서형, 구간형, 비율형 네가지로 구분된다. 측정 척도에 따라 유의미한 통계량도 함께 정해진다. 자세한 사항은 Stevens, Stanley Smith. “On the theory of scales of measurement.” (1946). 논문을 참조한다.

- 명목척도(Nominal): 단순히 개체 특성 분류를 위해 숫자나 부호를 부여한 척도로 숫자는 의미가 없음.
  - 남자: M, 여자: F 혹은 월: 1, 화: 2, … 일:7 혹은 갑:1, 을:2, 병:3, …
- 서열척도(Ordinal): 명목척도에 부가적으로 “순서(서열)” 정보가 추가된 척도로 측정대상 간 차이는 정보가 없음.
  - 군대계급: 사병, 장교, 장군 등
  - 소득계층: 1분위, 2분위, 3분위 등
- 등간척도(Interval): 서열척도에 부가적으로 “등간격” 정보가 추가된 척도
  - 온도에서 0도는 상대적인 위치로 수학에서 다루는 개념과 차이가 있음.
  - 온도가 서울 10도, 제주 20도는 제주가 서울보다 온도가 2배 높지 않음.

[1] Stevens, Stanley Smith. “On the theory of scales of measurement.” (1946).

[2] Wiener, Norbert. “A new theory of measurement: a study in the logic of mathematics.” Proceedings of the London Mathematical Society 2.1 (1921): 181-205.

[3] 이경화 외 (2020), “고등학교 실용통계”, 통계청 통계교육원

  - 온도, 시력, IQ 지수, 물가지수 등
- 비율척도(Ratio): 구간척도에 “비율” 비교특성이 추가된 척도로 “비율 등간격” 특성이 포함됨.
  - 키나 몸무게에서 0은 수학적 의미 0을 의미함.
  - 100m는 200m의 절반 의미.
  - 절대 ’0’을 가지고 사칙연산이 가능함.
  - 연령, 월소득, TV 시청률 등.

## 12.2 자료구조 기본

R에서 기본으로 사용하는 벡터 자료형은 **원자 벡터(Atomic Vector)** 와 **리스트(List)** 로 나눠진다. 원자 벡터에는 6가지 자료형이 있고, logical, integer, double, character, complex, raw, 총 6 개가 있으며 주로, 논리형, 정수형, 부동소수점형, 문자형, 4가지를 많이 사용한다. 리스트는 재귀 벡터(recursive vector)라고도 불리는데 리스트는 다른 리스트를 포함할 수 있기 때문이다. [4]

| 자료형(Type) | 모드(Mode) | 저장모드(Storage Mode) |
|---|---|---|
| logical | logical | logical |
| **integer** | numeric | integer |
| **double** | numeric | double |
| complex | complex | complex |
| character | character | character |
| raw | raw | raw |

따라서, 원자벡터는 동질적(homogeneous)이고, 리스트는 상대적으로 이질적(heterogeneous)이다.

모든 벡터는 두가지 성질(Property)을 갖는데, 자료형과 길이로 이를 확인하는데 `typeof()`와 `length()` 함수를 사용해서 확인한다.

```
library(tidyverse)

a <- list(a = 1:3,
          b = "a string",
          c = pi,
          d = list(-1, -5) )

typeof(a)
```

```
## [1] "list"
```

```
length(a)
```

```
## [1] 4
```

[4] Garrett Grolemund & Hadley Wickham, R for Data Science

모드 함수는 객체의 **모드**를 반환하고, 클래스 함수는 **클래스**를 반환한다. 가장 흔하게 만나는 객체 모드는 숫자, 문자, 논리 모드다. 때때로 리스트와 데이터프레임과 같이 하나의 객체안에 여러 모드를 포함하기도 한다.
리스트(List)는 데이터를 저장하는 유연하며 강력한 방법으로 과거 리스트 자료구조를 처리하는 sapply 함수와 함께 가장 빈번하게 사용되는 자료형이다. 현재는 purrr 팩키지 map_*()함수를 사용한다. 리스트형 자료 a를 세가지 숫자형, 문자형, 숫자형, 리스트 네가지 자료형을 포함하게 작성한다. sapply(), map_chr() 함수를 이용하여 mode와 class 인자를 넣어줌으로써, 각각 자료형의 모드와 자료형을 확인한다.

```
map_chr(a, mode)   # sapply(a, mode)
```

```
##           a           b           c           d
##   "numeric" "character"   "numeric"      "list"
```

```
map_chr(a, class)  # sapply(a, class)
```

```
##           a           b           c           d
##   "integer" "character"   "numeric"      "list"
```

리스트에서 원소를 뽑아내는 의미를 살펴보자. 시각적으로 표현하면 다음과 같다. 리스트는 이질적인 객체를 담을 수 있다는 점에서 동질적인 것만 담을 수 있어 한계가 있는 원자벡터보다 쓰임새가 다르다. 회귀분석 결과 산출되는 lm 결과값은 다양한 정보를 담을 수 있는 리스트로 표현된다.

- 리스트 생성 : list()
- 하위 리스트 추출 : [
- 리스트에 담긴 원소값 추출 : [[, $ → 연산작업을 통해 위계를 갖는 구조를 제거한다.

**리스트 원소 1개**

```
str(a[4])
```

```
## List of 1
##  $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

**리스트 원소 2개**

```
str(a[[4]])
```

```
## List of 2
##  $ : num -1
##  $ : num -5
```

범주형 자료를 R에 저장하기 위해서 요인(Factor) 클래스를 사용하며 요인 클래스를 사용하여 자료를 저장할 경우 저장공간을 절약할 수 있다. 요인은 내부적으로 숫자(value)로 저장을 하고 레이블(value label)을 사용하여 표시하여 저장공간을 절약한다.

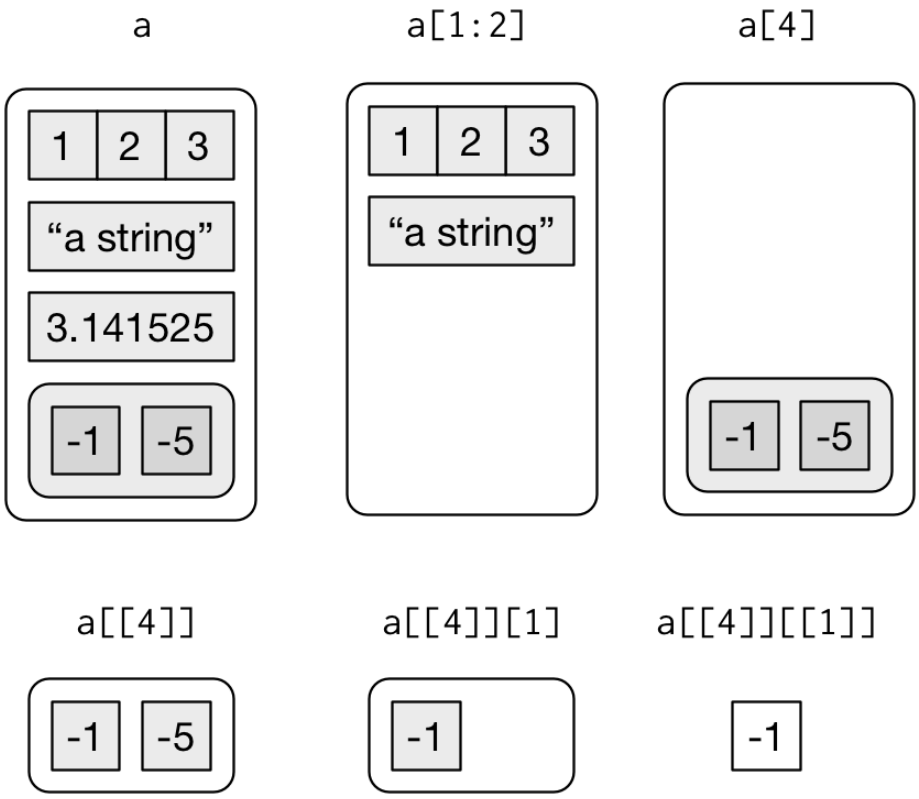



Figure 12.1: 리스트에서 하위 리스트 뽑아내기 - 출처: 해들리 위컴

**자료형 확인**

각각의 데이터 형식에 맞는지를 다양한 테스트 함수(`is.`)를 이용하여 데이터 형식을 확인할 수 있다.

- `is.list` : 리스트 형식인 확인
- `is.factor` : 팩터 형식인지 확인
- `is.numeric` : 숫자형인지 확인
- `is.data.frame` : 데이터 프레임형인지 확인
- `is.character` : 문자형인지 확인

## 12.3 자료형 확장

요인, 텍스트, 날짜와 시간도 중요한 R에서 자주 사용되는 중요한 데이터 자료형으로 별도로 다뤄진다. 이를 위해서 `stringr`, `lubridate`, `forcats` 팩키지를 사용해서 데이터 정제작업은 물론 기계학습 예측모형 개발에 활용한다.

| R 자료형 | 자료형 | 예제 |
|---|---|---|
| `logical` | 부울 | 부도여부(Y/N), 남여 |
| `integer` | 정수 | 코로나19 감염자수 |
| `factor` | 범주 | 정당, 색상 |
| `numeric` | 실수 | 키, 몸무게, 주가, 환율 |
| `character` | 텍스트 | 주소, 이름, 책제목 |
| `Date` | 날짜 | 생일, 투표일 |

## 12.4 벡터, 행렬, 배열, 데이터프레임

가장 많이 사용되는 논리형, 문자형, 숫자형을 통해 자료분석 및 모형개발을 진행하게 되고, 경우에 따라서 동일한 자료형을 모은 경우 이를 행렬로 표현할 수 있고, 행렬을 모아 RGB 시각 데이터를 위한 배열(Array)로 표현한다. 데이터프레임은 서로 다른 자료형을 모아 넣은 것이다.

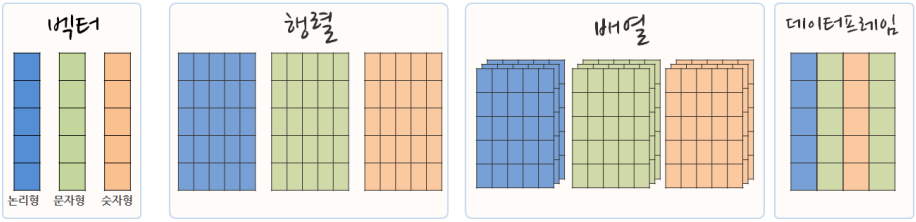



Figure 12.2: R 자료구조 - 벡터, 행렬, 배열, 데이터프레임

## 12.5 데이터프레임

R 은 6가지 기본 벡터로 자료를 저장하지만, 이외에 행렬(matrix), 데이터프레임(data.frame), 리스트(list) 자료구조가 있다. 하지만, 자료분석을 위해서 데이터를 데이터셋의 형태로 구성을 해야한다. 데이터셋이 중요한 이유는 자료를 분석하기 위해서 다양한 형태의 개별 자료를 통합적으로 분석하기 위해서다. 이를 위해서 리스트 자료구조로 일단 모으게 된다. 예를 들어 개인 신용분석을 위해서는 개인의 소득, 부채, 성별, 학력 등등의 숫자형, 문자형, 요인(Factor)형 등의 자료를 데이터셋에 담아야 한다. 특히 변수와-관측값 (Variable-Observation) 형식의 자료를 분석하기 위해서는 데이터프레임(`data.frame`)을 사용한다. 데이터프레임은 모든 변수에 대해서 관측값이 같은 길이를 갖도록 만들어 놓은 것이다.

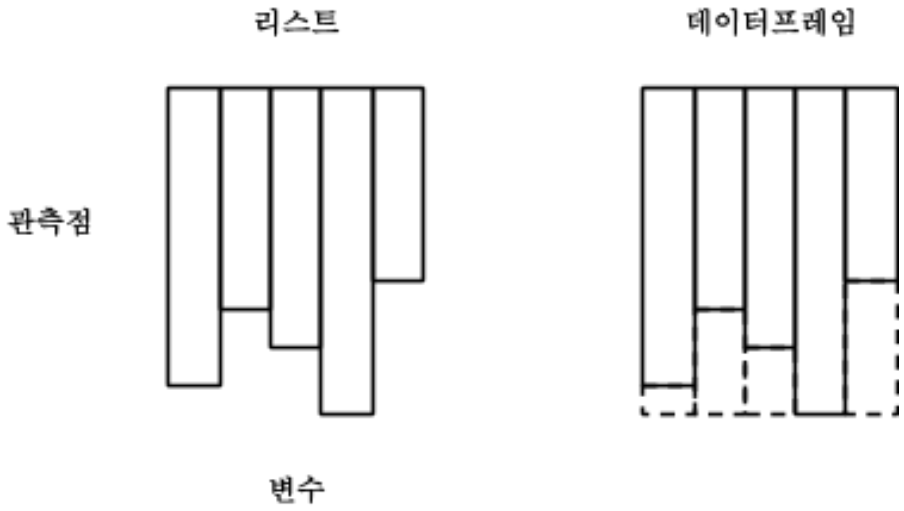



Figure 12.3: 리스트와 데이터프레임

데이터프레임은 `data.frame()` 함수를 사용해서 생성한다. R 객체 구조 파악을 위해서는 간단한 자료의 경우 데이터 형식을 확인할 수 있는 1-2줄 정도의 간단한 스크립트와 명령어를 통해서 확인이 가능하지만, 복잡한 데이터의 구조를 파악하기 위해서는 `summary` 함수와 `str` 함수를 통해서 확인해야 한다.

```
#      .
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
          "Terrestrial planet", "Gas giant", "Gas giant", "Gas giant", "Gas giant")
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rotation <- c(58.64, -243.02, 1, 1.03, 0.41, 0.43, -0.72, 0.67)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

#
planets_df <-data.frame(name, type, diameter, rotation, rings)
```

## 12.6 명목척도, 서열척도 범주 자료형

명목척도 범주형, 서열척도 범주 자료형을 생성하는 경우 주의를 기울여야 한다. `factor` 함수를 사용해서 요인형 자료형을 생성하는데, 내부적으로 저장공간을 효율적으로 사용하고 속도를 빠르게 하는데 유용한다. 순서를 갖는 범주형의 경우 `factor` 함수 내부에 `levels` 인자를 넣어 정의하면 순서 정보가 유지된다.

```
#    -
animals_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")
factor_animals_vector <- factor(animals_vector)
factor_animals_vector
```

```
## [1] Elephant Giraffe  Donkey   Horse
## Levels: Donkey Elephant Giraffe Horse
```

```
#    -
temperature_vector <- c("High", "Low", "High","Low", "Medium")
factor_temperature_vector <- factor(temperature_vector, order = TRUE, levels = c("Low", "Medium",
factor_temperature_vector
```

```
## [1] High   Low    High   Low    Medium
## Levels: Low < Medium < High
```

범주형 자료의 경우 범주가 갖는 척도 가독성을 높이기 위해 `levels()` 함수를 사용하기도 한다.

```
# "M", "F"
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector)
```

```
## [1] "F" "M"
```

```
# "Female", "Male"
levels(factor_survey_vector) <- c("Female", "Male")
levels(factor_survey_vector)
```

```
## [1] "Female" "Male"
```

통계 처리와 자료분석에 문자형 벡터와 요인 범주형 벡터를 다른 의미를 갖는 점에 유의한다. 동일한 `summary()` 함수지만 입력 자료형에 따라 R은 적절한 후속 작업을 자동 수행한다.

```
#
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)

#
summary(survey_vector)
```

```
##    Length     Class      Mode
##         5 character character
```

```
#
summary(factor_survey_vector)
```

```
## F M
## 2 3
```

## 12.7 NULL과 NA 결측값

결측되었다는 없다는 것을 표시하는 방법이 두가지 필요하다. 하나는 벡터가 없다는 NULL이고, 벡터 내부에 값이 결측되었다는 NA 다. dataframe$variable <- NULL 명령문을 사용하면 데이터프레임(dataframe)에 변수(variable)를 날려보내는 효과가 있다. 예를 들어 책장이 아예 없다는 의미(NULL)와 책장에 책이 없다(NA)는 다른 개념을 지칭하고 쓰임새가 다르다.

**NULL**

```
# NULL
typeof(NULL)
```

```
## [1] "NULL"
```

```
length(NULL)
```

```
## [1] 0
```

**NA**

```
# NA
typeof(NA)
```

```
## [1] "logical"
```

```
length(NA)
```

```
## [1] 1
```

NA의 중요한 특징은 전염된다는 것이다. 즉, NA에 연산을 가하면 연산결과는 무조건 NA가 된다. NA가 7보다 큰지, 7을 더하고 빼고, 부울 연산을 하든 NA와 연산결과는 무조건 NA가 된다.

```
NA + 7
```

```
## [1] NA
```

```
NA / 7
```

```
## [1] NA
```

```
NA > 7
```

```
## [1] NA
```

```
7 == NA
```

```
## [1] NA
```

```
NA == NA
```

```
## [1] NA
```

## 12.8 리스트 칼럼 [5] [6]

레고를 통해 살펴본 R 자료구조는 계산가능한 원자 자료형(논리형, 숫자형, 요인형)으로 크게 볼 수 있다. R에서 정수형과 부동소수점은 그다지 크게 구분을 하지 않는다. 동일 길이를 갖는 벡터를 쭉 붙여넣으면 자료구조형이 **데이터프레임**으로 되고, 길이가 갖지 않는 벡터를 한 곳에 모아넣은 자료구조가 **리스트**다.

데이터프레임이 굳이 모두 원자벡터만을 갖출 필요는 없다. 리스트를 데이터프레임 내부에 갖는 것도 데이터프레임인데 굳이 구별하자면 티블(tibble)이고, 이런 자료구조를 **리스트-칼럼(list-column)**이라고 부른다.


Figure 12.4: 리스트 칼럼

리스트-칼럼 자료구조가 빈번히 마주하는 경우가 흔한데… 대표적으로 다음 사례를 들 수 있다.

- 정규표현식을 통한 텍스트 문자열 처리

[5] List columns
[6] Photos that depict R data structures and operations via Lego

- 웹 API로 추출된 JSON, XML 데이터
- 분할-적용-병합(Split-Apply-Combine) 전략

데이터프레임이 티블(tibble) 형태로 되어 있으면 다음 작업을 나름 수월하게 추진할 수 있다.

- **들여다보기(Inspect)**: 데이터프레임에 무엇이 들었는지 확인.
- **인덱싱(Indexing)**: 명칭 혹은 위치를 확인해서 필요한 원소를 추출.
- **연산(Compute)**: 리스트-칼럼에 연산 작업을 수행해서 또다른 벡터나 리스트-칼럼을 생성.
- **간략화(Simplify)**: 리스트-칼럼을 익숙한 데이터프레임으로 변환.

# 분야별 도구

# Chapter 13

# 정규 표현식

지금까지 파일을 훑어서 패턴을 찾고, 관심있는 라인에서 다양한 비트(bits)를 뽑아냈다. `stringr` 패키지 `str_split`, `str_detect` 같은 문자열 함수를 사용하였고, 라인에서 일정 부분을 뽑아내기 위해서 리스트와 문자열 슬라이싱(slicing)을 사용했다.

검색하고 추출하는 작업은 너무 자주 있는 일이어서 파이썬은 상기와 같은 작업을 매우 우아하게 처리하는 **정규 표현식(regular expressions)**으로 불리는 매우 강력한 라이브러리를 제공한다. 정규 표현식을 책의 앞부분에 소개하지 않은 이유는 정규 표현식 라이브러리가 매우 강력하지만, 약간 복잡하고, 구문에 익숙해지는데 시간이 필요하기 때문이다.

정규표현식은 문자열을 검색하고 파싱하는데 그 자체가 작은 프로그래밍 언어다. 사실, 책 전체가 정규 표현식을 주제로 쓰여진 책이 몇권 있다. 이번 장에서는 정규 표현식의 기초만을 다룰 것이다. 정규 표현식의 좀더 자세한 사항은 다음을 참조하라.

- http://en.wikipedia.org/wiki/Regular_expression
- https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html

R에서 정규표현식을 지원하는 패키지는 많지만, 대표적으로 `stringr` 패키지가 활용사례도 많고 문서화도 충실하다. 정규 표현식 패키지를 사용하기 전에 패키지를 가져오기해야 한다. 정규 표현식 패키지의 가장 간단한 쓰임은 `str_detect()` 검색 함수다. 다음 프로그램은 검색 함수의 사소한 사용례를 보여준다.

```
library(tidyverse)
library(stringr)

hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {
  line <- stringr::str_squish(hand[line])
```

```
  if(stringr::str_detect(line, "From:")) {
    print(line)
  }
}
```

```
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: wagnermr@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: antranig@caret.cam.ac.uk"
## [1] "From: gopal.ramasammycook@gmail.com"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: ray@media.berkeley.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
```

파일을 열고, 각 라인을 루프로 반복해서 정규 표현식 `stringr::str_detect()` 함수를 호출하여 문자열 “From”이 포함된 라인만 출력한다. 상기 프로그램에는 진정으로 강력한 정규 표현식 기능이 사용되지 않았다. 왜냐하면, 다른 함수를 가지고도 동일한 결과를 쉽게 구현할 수 있기 때문이다.

정규 표현식의 강력한 기능은 문자열에 해당하는 라인을 좀더 정확하게 제어하기 위해서 검색 문자열에 특수문자를 추가할 때 확인될 수 있다. 매우 적은 코드를 작성할지라도, 정규 표현식에 특수 문자를 추가하는 것만으로도 정교한 일치(matching)와 추출이 가능하게 한다.

예를 들어, 탈자 기호(caret)는 라인의 “시작”과 일치하는 정규 표현식에 사용된다. 다음과 같이 “From:”으로 시작하는 라인만 일치하도록 응용프로그램을 변경할 수 있다.

```
for( line in seq_along(hand)) {
  line <- stringr::str_squish(hand[line])
  if(stringr::str_detect(line, "^From:")) {
    print(line)
  }
}
```

```
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: wagnermr@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: antranig@caret.cam.ac.uk"
## [1] "From: gopal.ramasammycook@gmail.com"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: ray@media.berkeley.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
```

“From:” 문자열로 시작하는 라인만 일치할 수 있다. 여전히 매우 간단한 프로그램으로 다른 패키지에서도 다양한 함수로 동일하게 수행할 수 있다. 하지만, 무엇을 정규 표현식과 매칭하는가에 대해서 특수 액션 문자(^)를 담아 강력한 제어를 수행하는 정규 표현식 개념을 소개하기에는 충분하다.

## 13.1 정규 표현식 문자 매칭

좀더 강력한 정규 표현식을 작성할 수 있는 다른 특수문자는 많이 있다. 가장 자주 사용되는 특수 문자는 임의 문자를 매칭하는 마침표다.

다음 예제에서 정규 표현식 “F..m:”은 “From:”, “Fxxm:”, “F12m:”, “F!@m:’ 같은

임의 문자열을 매칭한다. 왜냐하면 정규 표현식 마침표 문자가 임의의 문자와 매칭되기 때문이다.

```
for( line in seq_along(hand)) {
  line <- stringr::str_squish(hand[line])
  if(stringr::str_detect(line, "^F..m:")) {
    print(line)
  }
}
```

```
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: wagnermr@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: antranig@caret.cam.ac.uk"
## [1] "From: gopal.ramasammycook@gmail.com"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: ray@media.berkeley.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
```

정규 표현식에 “*”, “+’’ 문자를 사용하여 문자가 원하는만큼 반복을 나타내는 기능과 결합되었을 때는 더욱 강력해진다.”*“,”+’’ 특수 문자가 검색 문자열에 문자 하나만을 매칭하는 대신에 별표 기호인 경우 0 혹은 그 이상의 매칭, 더하기 기호인 경우 1 혹은 그 이상의 문자의 매칭을 의미한다.

다음 예제에서 반복 **와일드 카드(wild card)** 문자를 사용하여 매칭하는 라인을 좀더 좁힐 수 있다.

```
for( line in seq_along(hand)) {
  line <- stringr::str_squish(hand[line])
```

```
  if(stringr::str_detect(line, "^From:.+@")) {
    print(line)
  }
}
```

```
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: rjlowe@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: gsilver@umich.edu"
## [1] "From: wagnermr@iupui.edu"
## [1] "From: zqian@umich.edu"
## [1] "From: antranig@caret.cam.ac.uk"
## [1] "From: gopal.ramasammycook@gmail.com"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: david.horwitz@uct.ac.za"
## [1] "From: stephen.marquard@uct.ac.za"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: louis@media.berkeley.edu"
## [1] "From: ray@media.berkeley.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
## [1] "From: cwen@iupui.edu"
```

검색 문자열 "^From:.+@" 은 "From:" 으로 시작하고, ".+" 하나 혹은 그 이상의 문자들, 그리고 `@` 기호와 매칭되는 라인을 성공적으로 찾아낸다. 그래서 다음 라인은 매칭이 될 것이다.

```
From: stephen.marquard@uct.ac.za
```

콜론(:)과 `@` 기호 사이의 모든 문자들을 매칭하도록 확장하는 것으로 ".+" 와이드카드를 간주할 수 있다.

```
From:.+ @
```

더하기와 별표 기호를 "밀어내기(pushy)" 문자로 생각하는 것이 좋다. 예를 들어, 다음 문자열은 ".+" 특수문자가 다음에 보여주듯이 밖으로 밀어내는 것처럼 문자열 마지막 `@` 기호를 매칭한다.

```
From:stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

다른 특수문자를 추가함으로서 별표나 더하기 기호가 너무 “탐욕(greedy)”스럽지 않게 만들 수 있다. 와일드 카드 특수문자의 탐욕스러운 기능을 끄는 것에 대해서는 자세한 정보를 참조바란다.

## 13.2 정규 표현식 데이터 추출

R stringr 패키지로 문자열에서 데이터를 추출하려면, str_extract_all() 함수를 사용해서 정규 표현식과 매칭되는 모든 부속 문자열을 추출할 수 있다. 형식에 관계없이 임의 라인에서 전자우편 주소 같은 문자열을 추출하는 예제를 사용하자. 예를 들어, 다음 각 라인에서 전자우편 주소를 뽑아내고자 한다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

각각의 라인에 대해서 다르게 쪼개고, 슬라이싱하면서 라인 각각의 형식에 맞추어 코드를 작성하고는 싶지는 않다. 다음 프로그램은 str_extract_all() 함수를 사용하여 전자우편 주소가 있는 라인을 찾아내고 하나 혹은 그 이상의 주소를 뽑아낸다.

```
library(stringr)

s <- 'Hello from csev@umich.edu to cwen@iupui.edu about the meeting @2PM'

lst <- str_extract_all(s, '\\S+@\\S+')

# lst
```

str_extract_all() 함수는는 두번째 인자 패턴을 갖는 문자열을 찾아서 전자우편 주소처럼 보이는 모든 문자열을 리스트로 반환한다. 공백이 아닌 문자 (\\S)와 매칭되는 가운데 @을 갖는 두 문자열 시퀀스(sequence)를 매칭한다.

프로그램의 출력은 다음과 같다.

```
lst
```

```
## [[1]]
## [1] "csev@umich.edu" "cwen@iupui.edu"
```

정규 표현식을 해석하면, 적어도 하나의 공백이 아닌 문자, @과 적어도 하나 이상의 공백이 아닌 문자를 가진 부속 문자열을 찾는다. 또한, “\\S+” 특수 문자는 가능한 많이 공백이 아닌 문자를 매칭한다. (정규 표현식에서 **“탐욕(greedy)”** 매칭이라고 부른다.)

정규 표현식은 두번 매칭(csev@umich.edu, cwen@iupui.edu)하지만, 문자열 “@2PM”은 매칭을 하지 않는다. 왜냐하면, @ 기호 앞에 공백이 아닌 문자가 하나도

없기 때문이다. 프로그램의 정규 표현식을 사용해서 파일에 모든 라인을 읽고 다음과 같이 전자우편 주소처럼 보이는 모든 문자열을 출력한다.

```
hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_extract(line, "\\S+@\\S+")

  if(! is.na(str_length(x)) ) {
    print(x)
  }
}
```

각 라인을 읽어 들이고, 정규 표현식과 매칭되는 모든 부속 문자열을 추출한다. `str_extract()` 함수는 문자 벡터를 반환하기 때문에, 전자우편 처럼 보이는 부속 문자열을 적어도 하나 찾아서 출력하기 위해서 반환 리스트 요소 숫자가 `NA` 여부를 간단히 확인한다.

`mbox.txt` 파일에 프로그램을 실행하면, 다음 출력을 얻는다.

```
[1] "stephen.marquard@uct.ac.za"
[1] "<postmaster@collab.sakaiproject.org>"
[1] "<200801051412.m05ECIaH010327@nakamura.uits.iupui.edu>"
[1] "<source@collab.sakaiproject.org>;"
[1] "<source@collab.sakaiproject.org>;"
[1] "<source@collab.sakaiproject.org>;"
[1] "apache@localhost)"
[1] "source@collab.sakaiproject.org;"
[1] "stephen.marquard@uct.ac.za"
[1] "source@collab.sakaiproject.org"
[1] "stephen.marquard@uct.ac.za"
[1] "stephen.marquard@uct.ac.za"
[1] "louis@media.berkeley.edu"
...
```

전자우편 주소 몇몇은 "<", ";" 같은 잘못된 문자가 앞과 뒤에 붙어있다. 문자나 숫자로 시작하고 끝나는 문자열 부분만 관심있다고 하자.

그러기 위해서, 정규 표현식의 또 다른 기능을 사용한다. 매칭하려는 다중 허용 문자 집합을 표기하기 위해서 꺾쇠 괄호를 사용한다. 그런 의미에서 "\S"은 공백이 아닌 문자 집합을 매칭하게 한다. 이제 매칭하려는 문자에 관해서 좀더 명확해졌다.

여기 새로운 정규 표현식이 있다.

```
[a-zA-Z0-9]\\S*@\\S*[a-zA-Z]
```

약간 복잡해졌다. 왜 정규 표현식이 자신만의 언어인가에 대해서 이해할 수 있다. 이 정규 표현식을 해석하면, 0 혹은 그 이상의 공백이 아닌 문자("\\S*")로 하나의

소문자, 대문자 혹은 숫자("[a-zA-Z0-9]")를 가지며, @ 다음에 0 혹은 그 이상의 공백이 아닌 문자("\\S*")로 하나의 소문자, 대문자 혹은 숫자("[a-zA-Z0-9]")로 된 부속 문자열을 찾는다. 0 혹은 그 이상의 공백이 아닌 문자를 나타내기 위해서 "+"에서 ""으로 바꿨다. 왜냐하면 "[a-zA-Z0-9]" 자체가 이미 하나의 공백이 아닌 문자이기 때문이다. "", "+"는 단일 문자에 별표, 더하기 기호 왼편에 즉시 적용됨을 기억하세요.

프로그램에 정규 표현식을 사용하면, 데이터가 훨씬 깔끔해진다.

```
hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_extract(line, "[a-zA-Z0-9]\\S*@\\S*[a-zA-Z]")

  if(! is.na(str_length(x)) ) {
    print(x)
  }
}
```

```
[1] "stephen.marquard@uct.ac.za"
[1] "postmaster@collab.sakaiproject.org"
[1] "200801051412.m05ECIaH010327@nakamura.uits.iupui.edu"
[1] "source@collab.sakaiproject.org"
[1] "source@collab.sakaiproject.org"
[1] "source@collab.sakaiproject.org"
[1] "apache@localhost"
[1] "source@collab.sakaiproject.org"
[1] "stephen.marquard@uct.ac.za"
[1] "source@collab.sakaiproject.org"
[1] "stephen.marquard@uct.ac.za"
[1] "stephen.marquard@uct.ac.za"
[1] "louis@media.berkeley.edu"
[1] "postmaster@collab.sakaiproject.org"
[1] "200801042308.m04N8v6O008125@nakamura.uits.iupui.edu"
[1] "source@collab.sakaiproject.org"
[1] "source@collab.sakaiproject.org"
[1] "source@collab.sakaiproject.org"
[1] "apache@localhost"
...
```

"source@collab.sakaiproject.org" 라인에서 문자열 끝에 ">" 문자를 정규 표현식으로 제거한 것을 주목한다. 정규 표현식 끝에 "[a-zA-Z]"을 추가하여서 정규 표현식 파서가 찾는 임의 문자열은 문자로만 끝나야 되기 때문이다. 그래서, "sakaiproject.org>;"에서 ">"을 봤을 때, "g"가 마지막 맞는 매칭이 되고, 거기서 마지막 매칭을 마치고 중단한다.

str_extract_all() 프로그램의 출력은 리스트의 단일 요소를 가진 문자열로 R 리스트이다.

## 13.3 검색과 추출 조합

다음과 같은 "X-" 문자열로 시작하는 라인의 숫자를 찾고자 한다면,

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

임의의 라인에서 임의 부동 소수점 숫자가 아니라 상기 구문을 가진 라인에서만 숫자를 추출하고자 한다.

라인을 선택하기 위해서 다음과 같이 정규 표현식을 구성한다.

```
^X-.*: [0-9.]+
```

정규 표현식을 해석하면, ^에서 "X-"으로 시작하고, ".*"에서 0 혹은 그이상의 문자를 가지며, 콜론(":")이 나오고 나서 공백을 만족하는 라인을 찾는다. 공백 뒤에"[0-9.]+"에서 숫자 (0-9) 혹은 점을 가진 하나 혹은 그 이상의 문자가 있여야 한다. 꺽쇠 기호 사이에 마침표는 실제 마침표만 매칭함을 주목하기 바란다. (즉, 꺾쇠 기호 사이는 와일드 카드 문자가 아니다.)

관심을 가지고 있는 특정한 라인과 매우 정확하게 매칭이되는 매우 빠듯한 정규 표현식으로 다음과 같다.

```
library(stringr)

hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_detect(line, "^X\\S*: [0-9.]+")
  if(x == TRUE) {
    print(line[x])
  }
}
```

프로그램을 실행하면, 잘 걸러져서 찾고자 하는 라인만 볼 수 있다.

```
[1] "X-DSPAM-Confidence: 0.8475"
[1] "X-DSPAM-Probability: 0.0000"
[1] "X-DSPAM-Confidence: 0.6178"
[1] "X-DSPAM-Probability: 0.0000"
[1] "X-DSPAM-Confidence: 0.6961"
...
```

하지만, 이제 str_split() 함수를 사용해서 숫자를 뽑아내는 문제를 해결해야 한다.

str_split()을 사용하는 것이 간단해 보이지만, 동시에 라인을 검색하고 파싱하기 위해서 정규 표현식의 또 다른 기능을 사용할 수 있다.

괄호는 정규 표현식의 또 다른 특수 문자다. 정규 표현식에 괄호를 추가한다면, 문자열이 매칭될 때, 무시된다. 하지만, str_extract()을 사용할 때, 매칭할 전체 정규 표현식을 원할지라도, 정규 표현식을 매칭하는 부속 문자열의 부분만을 뽑아낸다는 것을 괄호가 표시한다.

그래서, 프로그램을 다음과 같이 수정한다.

```
library(stringr)

hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_extract(line, "^X\\S*: ([0-9.]+)")
  prob <- stringr::str_extract(x, "([0-9.]+)")

  if(!is.na(prob)) {
    print(prob)
  }
}
```

str_detect()을 호출하는 대신에, 매칭 문자열의 부동 소수점 숫자만 뽑아내는데 str_extract()에 원하는 부동 소수점 숫자를 표현하는 정규 표현식 부분에 괄호를 추가한다.

프로그램의 출력은 다음과 같다.

```
[1] "0.8475"
[1] "0.0000"
[1] "0.6178"
[1] "0.0000"
[1] "0.6961"
[1] "0.0000"
[1] "0.7565"
[1] "0.0000"
[1] "0.7626"
...
```

숫자가 여전히 리스트에 있어서 문자열에서 부동 소수점으로 변환할 필요가 있지만, 흥미로운 정보를 찾아 뽑아내기 위해서 정규 표현식의 강력한 힘을 사용했다.

이 기술을 활용한 또 다른 예제로, 파일을 살펴보면, 폼(form)을 가진 라인이 많다.

```
    Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

상기 언급한 동일한 기법을 사용하여 모든 변경 번호(라인의 끝에 정수 숫자)를

추출하고자 한다만, 다음과 같이 프로그램을 작성할 수 있다.

```
library(stringr)

hand <- read_lines("data/mbox-short.txt")

for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_extract(line, "^Details:.*rev=([0-9]+)")
  rev_num <- stringr::str_extract(x, "([0-9]+)")
  # rev_num <- stringr::str_extract(x, "([:digit:]+)")

  if(!is.na(rev_num)) {
    print(rev_num)
  }
}
```

```
## [1] "39772"
## [1] "39771"
## [1] "39770"
## [1] "39769"
## [1] "39766"
## [1] "39765"
## [1] "39764"
## [1] "39763"
## [1] "39762"
## [1] "39761"
## [1] "39760"
## [1] "39759"
## [1] "39758"
## [1] "39757"
## [1] "39756"
## [1] "39755"
## [1] "39754"
## [1] "39753"
## [1] "39752"
## [1] "39751"
## [1] "39750"
## [1] "39749"
## [1] "39746"
## [1] "39745"
## [1] "39744"
## [1] "39743"
## [1] "39742"
```

작성한 정규 표현식을 해석하면, “Details:”로 시작하는 “.*“에 임의의 문자들로,”rev=“을

포함하고 나서, 하나 혹은 그 이상의 숫자를 가진 라인을 찾는다. 전체 정규 표현식을 만족하는 라인을 찾고자 하지만, 라인 끝에 정수만을 추출하기 위해서"[0-9]+"을 괄호로 감쌌다.

프로그램을 실행하면, 다음 출력을 얻는다.

```
[1] "39772"
[1] "39771"
[1] "39770"
[1] "39769"
[1] "39766"
[1] "39765"
[1] "39764"
...
```

"[0-9]+"은 "탐욕(greedy)"스러워서, 숫자를 추출하기 전에 가능한 큰 문자열 숫자를 만들려고 한다는 것을 기억하라. 이런 "탐욕(greedy)"스러운 행동으로 인해서 왜 각 숫자로 모두 5자리 숫자를 얻은 이유가 된다. 정규 표현식 라이브러리는 양방향으로 파일 처음이나 끝에 숫자가 아닌 것을 마주칠 때까지 뻗어 나간다.

이제 정규 표현식을 사용해서 각 전자우편 메시지의 요일에 관심이 있었던 책 앞의 연습 프로그램을 다시 작성한다. 다음 형식의 라인을 찾는다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

그리고 나서, 각 라인의 요일의 시간을 추출하고자 한다. 앞에서 `str_split`를 두번 호출하여 작업을 수행했다. 첫번째는 라인을 단어로 쪼개고, 다섯번째 단어를 뽑아내서, 관심있는 두 문자를 뽑아내기 위해서 콜론 문자에서 다시 쪼갰다.

작동을 할지 모르지만, 실질적으로 정말 부서지기 쉬운 코드로 라인이 잘 짜여져 있다고 가정하에 가능하다. 잘못된 형식의 라인이 나타날 때도 결코 망가지지 않는 프로그램을 담보하기 위해서 충분한 오류 검사기능을 추가하거나 커다란 try/except 블록을 넣으면, 참 읽기 힘든 10-15 라인 코드로 커질 것이다.

다음 정규 표현식으로 훨씬 간결하게 작성할 수 있다.

```
^From .* [0-9][0-9]:
```

상기 정규 표현식을 해석하면, 공백을 포함한 "From"으로 시작해서, ".*"에 임의 갯수의 문자, 그리고 공백, 두 개의 숫자"[0-9][0-9]" 뒤에 콜론(:) 문자를 가진 라인을 찾는다. 일종의 찾고 있는 라인에 대한 정의다.

`str_extract()` 함수를 사용해서 단지 시간만 뽑아내기 위해서, 두 숫자에 괄호를 다음과 같이 추가한다.

```
^From .* ([0-9][0-9]):
```

작업 결과는 다음과 같이 프로그램에 반영한다.

```
library(stringr)

hand <- read_lines("data/mbox-short.txt")
```

```
for( line in seq_along(hand)) {

  line <- stringr::str_squish(hand[line])
  x <- stringr::str_extract(line, "^From .* ([0-9][0-9]):")
  days <- stringr::str_extract(x, "([0-9][0-9])")

  if(!is.na(days)) {
    print(days)
  }
}
```

```
## [1] "09"
## [1] "18"
## [1] "16"
## [1] "15"
## [1] "15"
## [1] "14"
## [1] "11"
## [1] "11"
## [1] "11"
## [1] "11"
## [1] "11"
## [1] "11"
## [1] "10"
## [1] "10"
## [1] "10"
## [1] "09"
## [1] "07"
## [1] "06"
## [1] "04"
## [1] "04"
## [1] "04"
## [1] "19"
## [1] "17"
## [1] "17"
## [1] "16"
## [1] "16"
## [1] "16"
```

프로그램을 실행하면, 다음 출력 결과가 나온다.

```
[1] "09"
[1] "18"
[1] "16"
[1] "15"
[1] "15"
```

```
[1] "14"
[1] "11"
[1] "11"
...
```

## 13.4 이스케이프(Escape) 문자

라인의 처음과 끝을 매칭하거나, 와일드 카드를 명세하기 위해서 정규 표현식의 특수 문자를 사용했기 때문에, 정규 표현식에 사용된 문자가 "정상(normal)"적인 문자임을 표기할 방법이 필요하고 달러 기호와 탈자 기호(^) 같은 실제 문자를 매칭하고자 한다.

역슬래쉬('⊠ ⊠ 를 가진 문자를 앞에 덮붙여서 문자를 단순히 매칭하고자 한다고 나타낼 수 있다. 예를 들어, 다음 정규표현식으로 금액을 찾을 수 있다.

```
library(stringr)

x <- 'We just received $10.00 for cookies.'

str_extract(x, pattern = '\\$[0-9.]+')
```

```
## [1] "$10.00"
```

역슬래쉬 달러 기호를 앞에 덮붙여서(\\$), 실제로 "라인 끝(end of line)" 매칭 대신에 입력 문자열의 달러 기호와 매칭한다. 정규 표현식 나머지 부분은 하나 혹은 그 이상의 숫자 혹은 소수점 문자를 매칭한다. 주목: 꺾쇠 괄호 내부에 문자는 "특수 문자"가 아니다. 그래서 "[0-9.]"은 실제 숫자 혹은 점을 의미한다. 꺾쇠 괄호 외부에 점은 "와일드 카드(wild-card)" 문자이고 임의의 문자와 매칭한다. 꺾쇠 괄호 내부에서 점은 점일 뿐이다.

## 13.5 요약

지금까지 정규 표현식의 표면을 긁은 정도지만, 정규 표현식 언어에 대해서 조금 학습했다. 정규 표현식은 특수 문자로 구성된 검색 문자열로 "매칭(matching)" 정의하고 매칭된 문자열로부터 추출된 결과물을 정규 표현식 시스템과 프로그래머가 의도한 바를 의사소통하는 것이다. 다음에 특수 문자 및 문자 시퀀스의 일부가 있다.

- ^ 라인의 처음을 매칭.
- $ 라인의 끝을 매칭.
- . 임의의 문자를 매칭(와일드 카드)
- s 공백 문자를 매칭.
- S 공백이 아닌 문자를 매칭.(s 의 반대).
- * 바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선 문자와 매칭을 표기함.
- *? 바로 앞선 문자에 적용되고 0 혹은 그 이상의 앞선 문자와 매칭을 "탐욕적이지 않은(non-greedy) 방식"으로 표기함.
- + 바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선 문자와 매칭을 표기함.

- +? 바로 앞선 문자에 적용되고 1 혹은 그 이상의 앞선 문자와 매칭을 “탐욕적이지 않은(non-greedy) 방식”으로 표기함.
- [aeiou] 명세된 집합 문자에 존재하는 단일 문자와 매칭. 다른 문자는 안되고, “a”, “e”, “i”, “o”, “u” 문자만 매칭되는 예제.
- [a-z0-9] 음수 기호로 문자 범위를 명세할 수 있다. 소문자이거나 숫자인 단일 문자만 매칭되는 예제.
- [^A-Za-z] 집합 표기의 첫문자가 ^인 경우, 로직을 거꾸로 적용한다. 대문자나 혹은 소문자가 아닌 임의 단일 문자만 매칭하는 예제.
- ( ) 괄호가 정규표현식에 추가될 때, 매칭을 무시한다. 하지만 str_extract()을 사용할 때 전체 문자열보다 매칭된 문자열의 상세한 부속 문자열을 추출할 수 있게 한다.
- \b 빈 문자열을 매칭하지만, 단어의 시작과 끝에만 사용된다.
- \B 빈 문자열을 매칭하지만, 단어의 시작과 끝이 아닌 곳에 사용된다.
- \d 임의 숫자와 매칭하여 [0-9] 집합에 상응함.
- \D 임의 숫자가 아닌 문자와 매칭하여 [^0-9] 집합에 상응함.

## 13.6 유닉스 사용자 보너스

정규 표현식을 사용하여 파일을 검색 기능은 1960년대 이래로 유닉스 운영 시스템에 내장되어 여러가지 형태로 거의 모든 프로그래밍 언어에서 이용가능하다.

사실, str_detect() 예제에서와 거의 동일한 기능을 하는 **grep** (Generalized Regular Expression Parser)으로 불리는 유닉스 내장 명령어 프로그램이 있다. 그래서, 맥킨토시나 리눅스 운영 시스템을 가지고 있다면, 명령어 창에서 다음 명령어를 시도할 수 있다.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

grep을 사용하여, mbox-short.txt 파일 내부에 “From:” 문자열로 시작하는 라인을 보여준다. grep 명령어를 가지고 약간 실험을 하고 grep에 대한 문서를 읽는다면, 파이썬에서 지원하는 정규 표현식과 grep에서 지원되는 정규 표현식과 차이를 발견할 것이다. 예를 들어, grep 공백이 아닌 문자 “\S”을 지원하지 않는다. 그래서 약간 더 복잡한 집합 표기 “[^ ]”을 사용해야 한다. “[^ ]”은 간단히 정리하면, 공복을 제외한 임의의 문자와 매칭한다.

## 13.7 디버깅

R에 대해서 도움을 어떻게 받을 수 있을까요?

만약 특정 함수의 정확한 이름을 기억해 내기 위해서 빠르게 생각나게 하는 것이 필요하다면 도움이 많이 될 수 있는 간단하고 초보적인 내장 문서가 R에 포함되어 있다. 내장 문서 도움말은 인터랙티브 모드의 R 인터프리터에서 볼 수 있다.

### 13.7.1 도움말 파일 읽어오기

R과 모든 팩키지는 함수에 대한 도움말 파일을 제공한다. 네임스페이스(인터랙티브 R 세션)에 적재된 팩키지에 있는 특정 함수에 대한 도움말은 다음과 같이 찾는다:

```
?function_name
help(function_name)
```

RStudio에 도움말 페이지에 도움말이 표시된다. (혹은 R 자체로 일반 텍스트로 표시된다)

각 도움말 페이지는 절(section)로 구분된다:

- 기술(Description): 함수가 어떤 작업을 수행하는가에 대한 충분한 기술
- 사용법(Usage): 함수 인자와 기본디폴트 설정값
- 인자(Arguments): 각 인자가 예상하는 데이터 설명
- 상세 설명(Details): 알고 있어야 되는 중요한 구체적인 설명
- 값(Value): 함수가 반환하는 데이터
- 함께 보기(See Also): 유용할 수 있는 연관된 함수.
- 예제(Examples): 함수 사용법에 대한 예제들.

함수마다 상이한 절을 갖추고 있다. 하지만, 상기 항목이 알고 있어야 하는 핵심 내용이다.

**꿀팁: 도움말 파일 불러 읽어오기**

R에 대해 가장 기죽게 되는 한 측면이 엄청난 함수 갯수다. 모든 함수에 대한 올바른 사용법을 기억하지 못하면, 엄두가 나지 않을 것이다. 운좋게도, 도움말 파일로 인해 기억할 필요가 없다!

### 13.7.2 특수 연산자

특수 연산자에 대한 도움말을 찾으려면, 인용부호를 사용한다:

```
?"<-"
```

### 13.7.3 팩키지 도움말 얻기

많은 팩키지에 “소품문(vignettes)”이 따라온다: 활용법과 풍부한 예제를 담은 문서. 어떤 인자도 없이, `vignette()` 명령어를 입력하면 설치된 모든 팩키지에 대한 모든 소품문 목록이 출력된다; `vignette(package="package-name")` 명령어는 `package-name` 팩키지명에 대한 이용가능한 모든 소품문 목록을 출력하고, `vignette("vignette-name")` 명령어는 특정된 소품문을 연다.

팩키지에 어떤 소품문도 포함되지 않는다면, 일반적으로 `help("package-name")` 명령어를 타이핑해서 도움말을 얻는다.

### 13.7.4 함수가 기억나지 않을 때

함수가 어느 팩키지에 있는지 확신을 못하거나, 구체적인 철자법을 모르는 경우, 퍼지 검색(fuzzy search)을 실행한다:

```
??function_name
```

### 13.7.5 시작조차 난감할 때

어떤 함수 혹은 팩키지가 필요한지 모르는 경우, CRAN Task Views 사이트가 좋은 시작점이 된다. 유지관리되는 팩키지 목록이 필드로 묶여 잘 정리되어 있다.

### 13.7.6 코드가 동작하지 않을 때

동료로부터 도움을 구해 코드가 동작하지 않는 이슈를 해결한다. 함수 사용에 어려움이 있는 경우, 10 에 9 경우에 찾는 정답이 이미 Stack Overflow에 답글이 달려 있다. 검색할 때 `[r]` 태그를 사용한다:

원하는 답을 찾지 못한 경우, 동료에게 질문을 만드는데 몇가지 유용한 함수가 있다:

```
?dput
```

`dput()` 함수는 작업하고 있는 데이터를 텍스트 파일 형식으로 덤프해서 저장한다. 그래서 다른 사람 R 세션으로 복사해서 붙여넣기 좋게 돕는다.

```
sessionInfo()
```

```
## R version 4.1.3 (2022-03-10)
## Platform: x86_64-apple-darwin20.6.0 (64-bit)
## Running under: macOS Big Sur 11.7
##
## Matrix products: default
## BLAS:   /usr/local/Cellar/openblas/0.3.20/lib/libopenblasp-r0.3.20.dylib
## LAPACK: /usr/local/Cellar/r/4.1.3/lib/R/lib/libRlapack.dylib
##
## locale:
## [1] ko_KR.UTF-8/ko_KR.UTF-8/ko_KR.UTF-8/C/ko_KR.UTF-8/C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] forcats_0.5.1   stringr_1.4.0   dplyr_1.0.9     purrr_0.3.4
##  [5] readr_2.1.1     tidyr_1.2.0     tibble_3.1.8    ggplot2_3.3.5
##  [9] tidyverse_1.3.1 DBI_1.1.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.9       lubridate_1.8.0  assertthat_0.2.1 digest_0.6.29
```

```
##  [5] utf8_1.2.2       R6_2.5.1         cellranger_1.1.0 backports_1.4.0
##  [9] reprex_2.0.1     RSQLite_2.2.12   evaluate_0.14    httr_1.4.2
## [13] pillar_1.8.1     rlang_1.0.4      readxl_1.3.1     rstudioapi_0.13
## [17] blob_1.2.2       rmarkdown_2.11   bit_4.0.4        munsell_0.5.0
## [21] broom_1.0.0      compiler_4.1.3   modelr_0.1.8     xfun_0.28
## [25] pkgconfig_2.0.3  htmltools_0.5.3  tidyselect_1.1.2 bookdown_0.24
## [29] fansi_1.0.3      crayon_1.5.1     tzdb_0.2.0       dbplyr_2.1.1
## [33] withr_2.5.0      grid_4.1.3       jsonlite_1.8.0   gtable_0.3.0
## [37] lifecycle_1.0.1  magrittr_2.0.3   scales_1.1.1     cli_3.3.0
## [41] stringi_1.7.6    vroom_1.5.7      cachem_1.0.6     fs_1.5.2
## [45] xml2_1.3.3       ellipsis_0.3.2   generics_0.1.2   vctrs_0.4.1
## [49] tools_4.1.3      bit64_4.0.5      glue_1.6.2       hms_1.1.1
## [53] parallel_4.1.3   fastmap_1.1.0    yaml_2.2.1       colorspace_2.0-2
## [57] rvest_1.0.2      memoise_2.0.1    knitr_1.36       haven_2.4.3
```

`sessionInfo()`는 R 현재 버젼 정보과 함께 적재된 팩키지 정보를 출력한다. 이 정보가 다른 사람이 여러분 문제를 재현하고 디버그하는데 유용할 수 있다.

## 13.8 용어정의

- **부서지기 쉬운 코드(brittle code)**:입력 데이터가 특정한 형식일 경우에만 작동하는 코드. 하지만 올바른 형식에서 약간이도 벗아나게 되면 깨지기 쉽니다. 쉽게 부서지기 때문에 "부서지기 쉬운 코드(brittle code)"라고 부른다.
- **욕심쟁이 매칭(greedy matching)**:정규 표현식의 "+", "*" 문자는 가능한 큰 문자열을 매칭하기 위해서 밖으로 확장하는 개념.
- **grep**: 정규 표현식에 매칭되는 파일을 탐색하여 라인을 찾는데 대부분의 유닉스 시스템에서 사용가능한 명령어. "Generalized Regular Expression Parser"의 약자.
- **정규 표현식(regular expression)**: 좀더 복잡한 검색 문자열을 표현하는 언어. 정규 표현식은 특수 문자를 포함해서 검색 라인의 처음 혹은 끝만 매칭하거나 많은 비슷한 것을 매칭한다.
- **와일드 카드(wild card)**: 임의 문자를 매칭하는 특수 문자. 정규 표현식에서 와이드 카드 문자는 마침표 문자다.

## 13.9 연습문제

1. 유닉스의 `grep` 명령어를 모사하는 간단한 프로그램을 작성하세요. 사용자가 정규 표현식을 입력하고 정규 표현식에 매칭되는 라인수를 셈하는 프로그램입니다.

```
$ Rscript grep.R
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author
```

```
$ Rscript grep.R
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ Rscript grep.R
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

2. 다음 형식의 라인만을 찾는 프로그램을 작성하세요.

`New Revision: 39772`

그리고, 정규 표현식과 `str_extract()` 함수를 사용하여 각 라인으로부터 숫자를 추출하세요. 숫자들의 평균을 구하고 출력하세요.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```

# Chapter 14

# 네트워크 프로그램

지금까지 책의 많은 예제는 파일을 읽고 파일의 정보를 찾는데 집중했지만, 다양한 많은 정보의 원천이 인터넷에 있다.

이번 장에서는 웹브라우져로 가장하고 HTTP 프로토콜(HyperText Transport Protocol,HTTP)을 사용하여 웹페이지를 검색할 것이다. 웹페이지 데이터를 읽고 파싱할 것이다.

파이썬 튜플과 마찬가지로 R에서 네트워크 프로그래밍을 위해 소켓까지 내려가서 프로그래밍을 할 경우는 많지 않다. 하지만 네트워크 프로그램에 대한 이해와 개념을 잡기 위해 파이썬 소켓 네트워크 프로그래밍을 먼저 살펴본다.

## 14.1 하이퍼 텍스트 전송 프로토콜

웹에 동력을 공급하는 네트워크 프로토콜(HyperText Transport Protocol - HTTP)은 실제로 매우 단순하다. 파이썬에는 `(sockets)`이라고 불리는 내장 지원 모듈이 있다. 파이썬 프로그램에서 소켓 모듈을 통해서 네트워크 연결을 하고, 데이터 검색을 매우 용이하게 한다.

**소켓(socket)**은 단일 소켓으로 두 프로그램 사이에 양방향 연결을 제공한다는 점을 제외하고 파일과 매우 유사하다. 동일한 소켓에 읽거나 쓸 수 있다. 소켓에 무언가를 쓰게 되면, 소켓의 다른 끝에 있는 응용프로그램에 전송된다. 소켓으로부터 읽게 되면, 다른 응용 프로그램이 전송한 데이터를 받게 된다.

하지만, 소켓의 다른쪽 끝에 프로그램이 어떠한 데이터도 전송하지 않았는데 소켓을 읽으려고 하면, 단지 앉아서 기다리기만 한다. 만약 어떠한 것도 보내지 않고 양쪽 소켓 끝의 프로그램 모두 기다리기만 한다면, 모두 매우 오랜 시간동안 기다리게 될 것이다.

인터넷으로 통신하는 프로그램의 중요한 부분은 특정 종류의 프로토콜을 공유하는 것이다. 프로토콜(protocol)은 정교한 규칙의 집합으로 누가 메시지를 먼저 보내고, 메세지로 무엇을 하며, 메시지에 대한 응답은 무엇이고, 다음에 누가 메세지를

보내고 등등을 포함한다. 이런 관점에서 소켓 끝의 두 응용프로그램이 함께 춤을 추고 있으니, 다른 사람 발을 밟지 않도록 확인해야 한다.

네트워크 프로토콜을 기술하는 문서가 많이 있다. 하이퍼텍스트 전송 프로토콜(HyperText Transport Protocol)은 다음 문서에 기술되어 있다.

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

매우 상세한 176 페이지나 되는 장문의 복잡한 문서다. 흥미롭다면 시간을 가지고 읽어보기 바란다. RFC2616에 36 페이지를 읽어보면, GET 요청(request)에 대한 구문을 발견하게 된다. 꼼꼼히 읽게 되면, 웹서버에 문서를 요청하기 하기 위해서, 80 포트로 `www.py4inf.com` 서버에 연결을 하고 나서 다음 양식 한 라인을 전송한다.

```
GET http://www.py4inf.com/code/romeo.txt HTTP/1.0
```

두번째 매개변수는 요청하는 웹페이지가 된다. 그리고 또한 빈 라인도 전송한다. 웹서버는 문서에 대한 헤더 정보와 빈 라인 그리고 문서 본문으로 응답한다.

## 14.2 가장 간단한 웹 브라우져

아마도 HTTP 프로토콜이 어떻게 작동하는지 알아보는 가장 간단한 방법은 매우 간단한 파이썬 프로그램을 작성하는 것이다. 웹서버에 접속하고 HTTP 프로토콜 규칙에 따라 문서를 요청하고 서버가 다시 보내주는 결과를 보여주는 것이다.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()
```

처음에 프로그램은 www.py4e.com 서버에 80 포트로 연결한다. "웹 브라우져" 역할로 작성된 프로그램이 하기 때문에 HTTP 프로토콜은 GET 명령어를 공백 라인과 함께 보낸다.

공백 라인을 보내자 마자, 512 문자 덩어리의 데이터를 소켓에서 받아 더 이상 읽을 데이터가 없을 때까지(즉, recv()이 빈 문자열을 반환한다.) 데이터를 출력하는 루프를 작성한다.

프로그램 실행결과 다음을 얻을 수 있다.

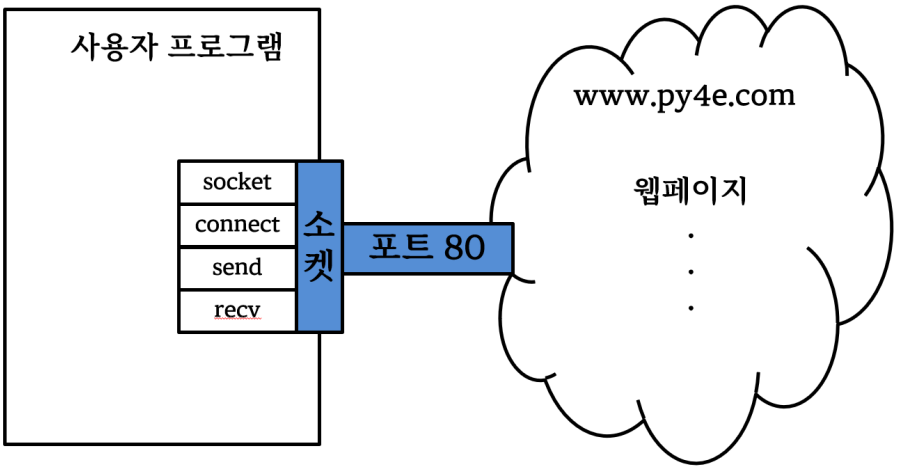



Figure 14.1: 소켓 개념도

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

출력결과는 웹서버가 문서를 기술하기 위해서 보내는 헤더(header)로 시작한다. 예를 들어, `Content-Type` 헤더는 문서가 일반 텍스트 문서(`text/plain`)임을 표기한다.

서버가 헤더를 보낸 후에, 빈 라인을 추가해서 헤더 끝임을 표기하고 나서 실제 파일`romeo.txt`을 보낸다.

이 예제를 통해서 소켓을 통해서 저수준(low-level) 네트워크 연결을 어떻게 하는지 확인할 수 있다. 소켓을 사용해서 웹서버, 메일 서버 혹은 다른 종류의 서버와 통신할 수 있다. 필요한 것은 프로토콜을 기술하는 문서를 찾고 프로토콜에 따라 데이터를

주고 받는 코드를 작성하는 것이다.

하지만, 가장 흔히 사용하는 프로토콜은 HTTP (즉, 웹) 프로토콜이기 때문에, 파이썬에는 HTTP 프로토컬을 지원하기 위해 특별히 설계된 라이브러리가 있다. 이것을 통해서 웹상에서 데이터나 문서를 검색을 쉽게 할 수 있다.

## 14.3 HTTP 경유 이미지 가져오기

상기 예제에서는 파일에 새줄(newline)이 있는 일반 텍스트 파일을 가져왔다. 그리고 나서, 프로그램을 실행해서 데이터를 단순히 화면에 복사했다. HTTP를 사용하여 이미지를 가져오도록 비슷하게 프로그램을 작성할 수 있다. 프로그램 실행 시에 화면에 데이터를 복사하는 대신에, 데이터를 문자열로 누적하고, 다음과 같이 헤더를 잘라내고 나서 파일에 이미지 데이터를 저장한다.

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()
```

```
# Code: http://www.py4e.com/code3/urljpeg.py
```

프로그램을 실행하면, 다음과 같은 출력을 생성한다.

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

상기 url에 대해서, `Content-Type` 헤더가 문서 본문이 이미지(`image/jpeg`)를 나타내는 것을 볼 수 있다. 프로그램이 완료되면, 이미지 뷰어로 `stuff.jpg` 파일을 열어서 이미지 데이터를 볼 수 있다.

프로그램을 실행하면, `recv()` 메쏘드를 호출할 때 마다 5120 문자는 전달받지 못하는 것을 볼 수 있다. `recv()` 호출하는 순간마다 웹서버에서 네트워크로 전송되는 가능한 많은 문자를 받을 뿐이다. 매번 5120 문자까지 요청하지만, 1460 혹은 2920 문자만 전송받는다.

결과값은 네트워크 속도에 따라 달라질 수 있다. `recv()` 메쏘드 마지막 호출에는 스트림 마지막인 1681 바이트만 받았고, `recv()` 다음 호출에는 0 길이 문자열을 전송받아서, 서버가 소켓 마지막에 `close()` 메쏘드를 호출하고 더이상의 데이터가 없다는 신호를 준다.

주석 처리한 `time.sleep()`을 풀어줌으로써 `recv()` 연속 호출을 늦출 수 있다. 이런 방식으로 매번 호출 후에 0.25초 기다리게 한다. 그래서, 사용자가 `recv()` 메쏘드를 호출하기 전에 서버가 먼저 도착할 수 있어서 더 많은 데이터를 보낼 수가 있다. 정지 시간을 넣어서 프로그램을 다시 실행하면 다음과 같다.

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

recv() 메쏘드 호출의 처음과 마지막을 제외하고, 매번 새로운 데이터를 요청할 때마다 이제 5120 문자가 전송된다.

서버 send() 요청과 응용프로그램 recv() 요청 사이에 버퍼가 있다. 프로그램에 지연을 넣어 실행하게 될 때, 어느 지점엔가 서버가 소켓 버퍼를 채우고 응용프로그램이 버퍼를 비울 때까지 잠시 멈춰야 된다. 송신하는 응용프로그램 혹은 수신하는 응용프로그램을 멈추게 하는 행위를 "흐름 제어(flow control)"이라고 한다.

## 14.4 httr 웹페이지 가져오기

수작업으로 소켓 라이브러리를 사용하여 HTTP로 데이터를 주고 받을 수 있지만, httr 패키지를 사용하여 R에서 동일한 작업을 수행하는 좀더 간편한 방식이 있다.

httr을 사용하여 파일처럼 웹페이지를 다룰 수가 있다. 단순하게 어느 웹페이지를 가져올 것인지만 지정하면 httr 라이브러리가 모든 HTTP 프로토콜과 헤더 관련 사항을 처리해 준다.

웹에서 romeo.txt 파일을 읽도록 urllib를 사용하여 작성한 상응 코드는 다음과 같다.

```
library(httr)

fhand <- httr::GET('http://www.py4inf.com/code/romeo.txt')
```

```
httr::content(fhand, encoding = "UTF-8")
```

GET() 함수를 사용하여 웹페이지를 열게 되면, 파일처럼 다룰 수 있고 content() 함수를 사용해서 데이터를 읽을 수 있다.

프로그램을 실행하면, 파일 내용 출력결과만을 볼 수 있다. 헤더정보는 여전히 전송되었지만, GET() 함수가 헤더를 받아 내부적으로 처리하고, 사용자에게는 단지 데이터만 반환한다.

```
## [1] "But soft what light through yonder window breaks\nIt is the east and Juliet is the sun\nA
```

예제로, romeo.txt 데이터를 가져와서 파일의 각 단어 빈도를 계산하는 프로그램을 다음과 같이 작성할 수 있다. 웹페이지를 로컬 텍스트로 변환시킨 후에 공백과 \n 으로 텍스트를 잘게 쪼개고, 단어 빈도수를 table()로 계산한 후 데이터 문법 dplyr을 사용해서 최빈 단어를 상위 3개 뽑는다.

```
## # A tibble: 3 x 2
##
##   <chr>  <int>
## 1 and        3
## 2 is         3
## 3 the        3
```

다시 한번, 웹페이지를 열게 되면, 로컬 파일처럼 웹페이지를 읽을 수 있다.

## 14.5 HTML 파싱과 웹 스크래핑

R httr 패키지를 활용하는 일반적인 사례는 웹 **스크래핑(scraping)**이다. 웹 스크래핑은 웹브라우저를 가장한 프로그램을 작성하는 것이다. 웹페이지를 가져와서, 패턴을 찾아 페이지 내부의 데이터를 꼼꼼히 조사한다. 예로, 구글같은 검색엔진은 웹 페이지의 소스를 조사해서 다른 페이지로 가는 링크를 추출하고, 그 해당 페이지를 가져와서 링크 추출하는 작업을 반복한다. 이러한 기법으로 구글은 웹상의 거의 모든 페이지를 **거미(spiders)**줄처럼 연결한다.

구글은 또한 발견한 웹페이지에서 특정 페이지로 연결되는 링크 빈도를 사용하여 얼마나 중요한 페이지인지를 측정하고 검색결과에 페이지가 얼마나 높은 순위로 노출되어야 하는지 평가한다.

## 14.6 정규 표현식 사용 HTML 파싱하기

HTML을 파싱하는 간단한 방식은 정규 표현식을 사용하여 특정한 패턴과 매칭되는 부속 문자열을 반복적으로 찾아 추출하는 것이다.

여기 간단한 웹페이지가 있다.

```
<h1>The First Page</h1>
<p>
```

```
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

모양 좋은 정규표현식을 구성해서 다음과 같이 상기 웹페이지에서 링크를 매칭하고 추출할 수 있다.

```
href="http://.+?"
```

작성된 정규 표현식은 “href=http://%22로 시작하고, 하나 이상의 문자를”.+?” 가지고 큰 따옴표를 가진 문자열을 찾는다. “.+?”에 물음표가 갖는 의미는 매칭이 “욕심쟁이(greedy)” 방식보다 “비욕심쟁이(non-greedy)” 방식으로 수행됨을 나타낸다. 비욕심쟁이(non-greedy) 매칭방식은 가능한 가장 적게 매칭되는 문자열을 찾는 방식이고, 욕심 방식은 가능한 가장 크게 매칭되는 문자열을 찾는 방식이다.

추출하고자 하는 문자열이 매칭된 문자열의 어느 부분인지를 표기하기 위해서 정규표현식에 괄호를 추가하여 다음과 같이 프로그램을 작성한다.

```
library(tidyverse)
library(httr)

url <- readline(prompt = "Enter - ")

html <- httr::GET(url)

url_link <- str_extract_all(html, pattern = 'href="(http://.*?)"')

print(url_link)
```

str_extract_all 정규 표현식 함수는 정규 표현식과 매칭되는 모든 문자열 리스트를 추출하여 큰 따옴표 사이에 링크 텍스트만을 반환한다.

쉘에서 Rscript.exe 프로그램으로 실행하기 위해 다음과 같이 사용자 입력을 받는 부분을 다음과 같이 수정한다.

```
# urlregex.R

library(httr)

typeline <- function(msg = "Enter - ") {
  if (interactive() ) {
    url <- readline(msg)
  } else {
    cat(msg);
    url <- readLines("stdin",n=1);
  }
  return(url)
```

```
}

url <- typeline()

html <- httr::GET(url)

url_link <- stringr::str_extract_all(html, pattern = 'href="(http://.*?)"')

print(url_link)
```

프로그램을 실행하면, 다음 출력을 얻게된다.

```
$ Rscript.exe ./script/urlregex.R
During startup - Warning message:
Setting LC_CTYPE=ko_KR.UTF-8@cjknarrow failed
Enter - http://www.dr-chuck.com/page1.htm
No encoding supplied: defaulting to UTF-8.
Warning message:
In stri_extract_all_regex(string, pattern, simplify = simplify,  :
  argument is not an atomic vector; coercing
[[1]]
[1] "href=\"http://www.dr-chuck.com/page2.htm\""

$ Rscript.exe ./script/urlregex.R
During startup - Warning message:
Setting LC_CTYPE=ko_KR.UTF-8@cjknarrow failed
Enter - http://www.py4inf.com/book.htm
No encoding supplied: defaulting to UTF-8.
Warning message:
In stri_extract_all_regex(string, pattern, simplify = simplify,  :
  argument is not an atomic vector; coercing
[[1]]
 [1] "href=\"http://amzn.to/1KkULF3\""
 [2] "href=\"http://www.py4e.com/book\""
 [3] "href=\"http://amzn.to/1KkULF3\""
 [4] "href=\"http://amzn.to/1hLcoBy\""
 [5] "href=\"http://amzn.to/1KkV42z\""
 [6] "href=\"http://amzn.to/1fNOnbd\""
 [7] "href=\"http://amzn.to/1N74xLt\""
 [8] "href=\"http://do1.dr-chuck.net/py4inf/EN-us/book.pdf\""
 [9] "href=\"http://do1.dr-chuck.net/py4inf/ES-es/book.pdf\""
[10] "href=\"http://do1.dr-chuck.net/py4inf/PT-br/book.pdf\""
[11] "href=\"http://www.xwmooc.net/python/\""
[12] "href=\"http://fanwscu.gitbooks.io/py4inf-zh-cn/\""
[13] "href=\"http://itunes.apple.com/us/book/python-for-informatics/id554638579?mt=13\""
[14] "href=\"http://www-personal.umich.edu/~csev/books/py4inf/ibooks//python_for_informatics.iboo
```

```
[15] "href=\"http://www.py4inf.com/code\""
[16] "href=\"http://www.greenteapress.com/thinkpython/thinkCSpy/\""
[17] "href=\"http://allendowney.com/\""
```

정규 표현식은 HTML이 예측가능하고 잘 구성된 경우에 멋지게 작동한다. 하지만, "망가진" HTML 페이지가 많아서, 정규 표현식만을 사용하는 솔류션은 유효한 링크를 놓치거나 잘못된 데이터만 찾고 끝날 수 있다.

이 문제는 강건한 HTML 파싱 라이브러리를 사용해서 해결될 수 있다.

## 14.7 'rvest" 사용한 HTML 파싱

HTML을 파싱하여 페이지에서 데이터를 추출할 수 있는 R 패키지는 많이 있다. 패키지 각각은 강점과 약점이 있어서 사용자 필요에 따라 취사선택한다.

예로, 간단하게 HTML 입력을 파싱하여 **rvest** 라이브러리를 사용하여 링크를 추출할 것이다.

HTML이 XML 처럼 보이고 몇몇 페이지는 XML로 되도록 꼼꼼하게 구축되었지만, 일반적으로 대부분의 HTML이 깨져서 XML 파서가 HTML 전체 페이지를 잘못 구성된 것으로 간주하고 받아들이지 않는다. 'rvest" 패키지는 결점 많은 HTML 페이지에 내성이 있어서 사용자가 필요로하는 데이터를 쉽게 추출할 수 있게 한다.

`httr` 패키지를 사용하여 페이지를 읽어들이고, `rvest`를 사용해서 앵커 태그(a)로부터 `href` 속성을 추출한다.

```
library(httr)
library(rvest)

html <- httr::GET('http://www.dr-chuck.com/page1.htm')
webpage <- rvest::read_html(html)

a_href <- webpage %>%
  html_elements("a") %>%
  html_attr("href")

print(a_href)
```

프로그램이 웹 주소를 입력받고, 웹페이지를 열고, 데이터를 읽어서 BeautifulSoup 파서에 전달하고, 그리고 나서 모든 앵커 태그를 불러와서 각 태그별로 `href` 속성을 출력한다.

쉘에서 `Rscript.exe` 프로그램으로 실행하기 위해 다음과 같이 사용자 입력을 받는 부분을 다음과 같이 수정한다.

```
## urllinks.R

library(httr)
library(rvest)
```

```
typeline <- function(msg = "Enter - ") {
  if (interactive() ) {
    url <- readline(msg)
  } else {
    cat(msg);
    url <- readLines("stdin",n=1);
  }
  return(url)
}

url <- typeline()

html <- httr::GET(url)
webpage <- rvest::read_html(html)

a_href <- webpage %>%
  html_elements("a") %>%
  html_attr("href")

print(a_href)
```

프로그램을 실행하면, 아래와 같다.

```
$ Rscript.exe ./script/urllinks.R
During startup - Warning message:
Setting LC_CTYPE=ko_KR.UTF-8@cjknarrow failed
Enter - http://www.dr-chuck.com/page1.htm
[1] "http://www.dr-chuck.com/page2.htm"

$ Rscript.exe ./script/urllinks.R
During startup - Warning message:
Setting LC_CTYPE=ko_KR.UTF-8@cjknarrow failed
Enter - http://www.py4inf.com/book.htm
 [1] "http://amzn.to/1KkULF3"
 [2] "http://www.py4e.com/book"
 [3] "http://amzn.to/1KkULF3"
 [4] "http://amzn.to/1hLcoBy"
 [5] "http://amzn.to/1KkV42z"
 [6] "http://amzn.to/1fNOnbd"
 [7] "http://amzn.to/1N74xLt"
 [8] "http://do1.dr-chuck.net/py4inf/EN-us/book.pdf"
 [9] "http://do1.dr-chuck.net/py4inf/ES-es/book.pdf"
[10] "https://twitter.com/fertardio"
[11] "http://do1.dr-chuck.net/py4inf/PT-br/book.pdf"
[12] "https://twitter.com/victorjabur"
```

```
[13] "translations/KO/book_009_ko.pdf"
[14] "http://www.xwmooc.net/python/"
[15] "http://fanwscu.gitbooks.io/py4inf-zh-cn/"
[16] "book_270.epub"
[17] "translations/ES/book_272_es4.epub"
[18] "https://www.gitbook.com/download/epub/book/fanwscu/py4inf-zh-cn"
[19] "html-270/"
[20] "html_270.zip"
[21] "http://itunes.apple.com/us/book/python-for-informatics/id554638579?mt=13"
[22] "http://www-personal.umich.edu/~csev/books/py4inf/ibooks//python_for_informatics.
[23] "http://www.py4inf.com/code"
[24] "http://www.greenteapress.com/thinkpython/thinkCSpy/"
[25] "http://allendowney.com/"
```

'rvest"을 사용하여 다음과 같이 각 태그별로 다양한 부분을 뽑아낼 수 있다.

```
library(httr)
library(rvest)

typeline <- function(msg = "Enter - ") {
  if (interactive() ) {
    url <- readline(msg)
  } else {
    cat(msg);
    url <- readLines("stdin",n=1);
  }
  return(url)
}

url <- typeline()

html <- httr::GET(url)
webpage <- rvest::read_html(html)

a_href <- webpage %>%
  html_elements("a") %>%
  html_attr("href")

a_text <- webpage %>%
  html_elements("a") %>%
  html_text() %>%
  stringr::str_remove("\n")

cat("URL: ", a_href, "\n")
cat("Content:", a_text)
```

상기 프로그램은 다음을 출력합니다.

```
$ Rscript.exe ./script/urllinks2.R
During startup - Warning message:
Setting LC_CTYPE=ko_KR.UTF-8@cjknarrow failed
Enter - http://www.dr-chuck.com/page1.htm
URL:  http://www.dr-chuck.com/page2.htm
Content: Second Page
```

HTML을 파싱하는데 `rvest`가 가진 강력한 기능을 예제로 보여줬다. 좀더 자세한 사항은 https://rvest.tidyverse.org/ 에서 문서와 예제를 살펴보세요.

## 14.8 바이너리 파일 읽기

이미지나 비디오 같은 텍스트가 아닌 (혹은 바이너리) 파일을 가져올 때가 종종 있다. 일반적으로 이런 파일 데이터를 출력하는 것은 유용하지 않다. 하지만, `download.file()` 함수를 사용하여, 하드 디스크 로컬 파일에 URL 사본을 쉽게 만들 수 있다.

`download.file()` 함수 내부에 인터넷에 공개된 이미지 `url`을 적고, `destfile`에는 로컬파일에 저장할 파일명을 적어준다. 중요한 것은 텍스트가 아니라 바이너리 이미지라 `mode = "wb"`를 지정한다.

```
download.file(url = "http://www.py4inf.com/cover.jpg",
              destfile = "assets/images/fox.jpeg", mode="wb")
```

작성된 프로그램은 네트워크로 모든 데이터를 한번에 읽어서 컴퓨터 `cover.jpg` 파일을 다운로드 받아 로컬 디스크에 지정한 디렉토리 파일명으로 데이터를 쓴다. 이 방식은 파일 크기가 사용자 컴퓨터의 메모리 크기보다 작다면 정상적으로 작동한다.

UNIX 혹은 매킨토시 컴퓨터를 가지고 있다면, 다음과 같이 상기 동작을 수행하는 명령어가 운영체제 자체에 내장되어 있다.

```
curl -O http://www.py4inf.com/cover.jpg
```

## 14.9 용어정의

- **BeautifulSoup**: 파이썬 라이브러리로 HTML 문서를 파싱하고 브라우저가 일반적으로 생략하는 HTML의 불완전한 부분을 보정하여 HTML 문서에서 데이터를 추출한다. www.crummy.com 사이트에서 BeautifulSoup 코드를 다운로드 받을 수 있다.
- **rvest**: 파이썬 BeautifulSoup에 대응되는 R 크롤링 패키징

- **포트(port)**: 서버에 소켓 연결을 만들 때, 사용자가 무슨 응용프로그램을 연결하는지 나타내는 숫자. 예로, 웹 트래픽은 통상 80 포트, 전자우편은 25 포트를 사용한다.
- **스크래핑(scraping)**: 프로그램이 웹브라우저를 가장하여 웹페이지를 가져와서 웹 페이지의 내용을 검색한다. 종종 프로그램이 한 페이지의

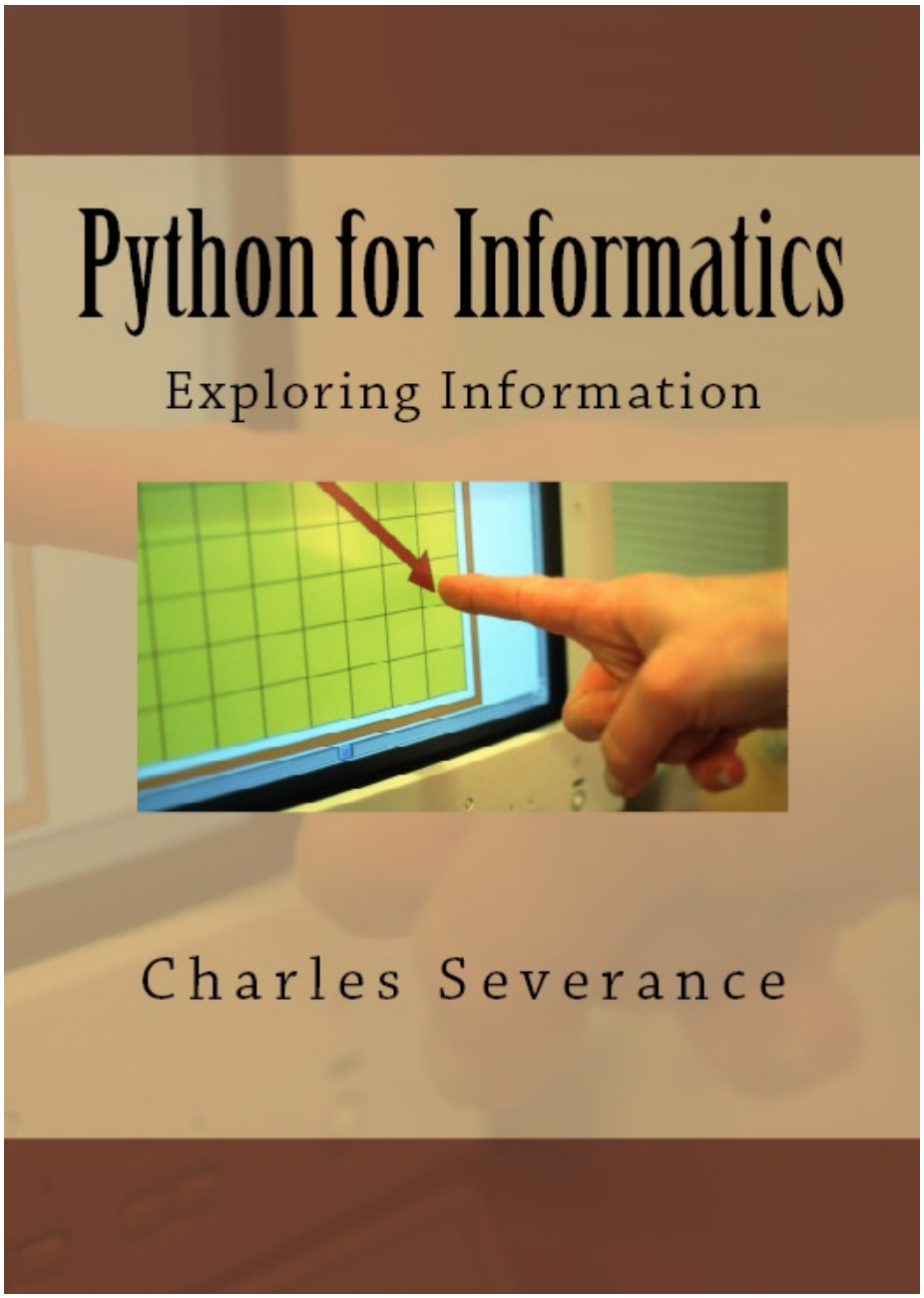



Figure 14.2: 바이너리 파일 다운로드

링크를 따라 다른 페이지를 찾게 된다. 그래서, 웹페이지 네트워크 혹은 소셜 네트워크 전체를 훑을 수 있다.

- **소켓(socket)**: 두 응용프로그램 사이 네트워크 연결. 두 응용프로그램은 양방향으로 데이터를 주고 받는다.
- **스파이더(spider)**:검색 색인을 구축하기 위해서 한 웹페이지를 검색하고, 그 웹페이지에 링크된 모든 페이지 검색을 반복하여 인터넷에 있는 거의 모든 웹페이지를 가져오기 위해서 사용되는 검색엔진 행동.

## 14.10 연습문제

1. 소켓 프로그램 `socket1.py`을 변경하여 임의 웹페이지를 읽을 수 있도록 URL을 사용자가 입력하도록 바꾸세요. `split('/')`을 사용하여 URL을 컴포턴트로 쪼개서 소켓 `connect` 호출에 대해 호스트 명을 추출할 수 있다. 사용자가 적절하지 못한 형식 혹은 존재하지 않는 URL을 입력하는 경우를 처리할 있도록 `try`, `except`를 사용하여 오류 검사기능을 추가하세요.

2. 소켓 프로그램을 변경하여 전송받은 문자를 계수(count)하고 3000 문자를 출력한 후에 그이상 텍스트 출력을 멈추게 하세요. 프로그램은 전체 문서를 가져와야 하고, 전체 문자를 계수(count)하고, 문서 마지막에 문자 계수(count)결과를 출력해야 합니다.

3. `httr` 패키지를 사용하여 이전 예제를 반복하세요. (1) 사용자가 입력한 URL에서 문서 가져오기 (2) 3000 문자까지 화면에 보여주기 (3) 문서의 전체 문자 계수(count)하기. 이 연습문제에서 헤더에 대해서는 걱정하지 말고, 단지 문서 본문에서 첫 3000 문자만 화면에 출력하세요.

4. `urllinks.R` 프로그램을 변경하여 가져온 HTML 문서에서 문단 (p) 태그를 추출하고 프로그램의 출력물로 문단을 계수(count)하고 화면에 출력하세요. 문단 텍스트를 화면에 출력하지 말고 단지 숫자만 셉니다. 작성한 프로그램을 작은 웹페이지 뿐만 아니라 조금 큰 웹 페이지에도 테스트해 보세요.

5. (고급) 소켓 프로그램을 변경하여 헤더와 빈 라인 다음에 데이터만 보여지게 하세요. `recv`는 라인이 아니라 문자(새줄(newline)과 모든 문자)를 전송받는다는 것을 기억하세요.

# Chapter 15

# 웹서비스 사용하기

프로그램을 사용하여 HTTP상에서 문서를 가져와서 파싱하는 것이 익숙해지면, 다른 프로그램(즉, 브라우져에서 HTML로 보여지지 않는 것)에서 활용되도록 특별히 설계된 문서를 생성하는 것은 그다지 오래 걸리지 않는다.

웹상에서 데이터를 교환할 때 두 가지 형식이 많이 사용된다. XML("eXtensible Markup Language")은 오랜 기간 사용되어져 왔고 문서-형식(document-style) 데이터를 교환하는데 가장 적합하다. 딕셔너리, 리스트 혹은 다른 내부 정보를 프로그램으로 서로 교환할 때, JSON(JavaScript Object Notation, www.json.org)을 사용한다. 두 가지 형식에 대해 모두 살펴볼 것이다.

## 15.1 XML

XML(eXtensible Markup Language)은 HTML과 매우 유사하지만, XML이 좀더 HTML보다 구조화되었다. 여기 XML 문서 샘플이 있다.

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

종종 XML문서를 나무 구조(tree structure)로 생각하는 것이 도움이 된다. 최상단 `person` 태그가 있고, `phone` 같은 다른 태그는 부모 노드의 자식(children) 노드로 표현된다.

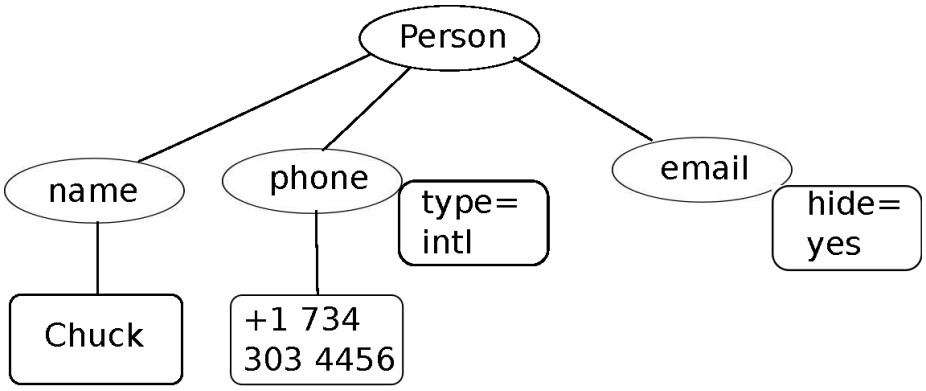



Figure 15.1: XML 나무구조 도식화

## 15.2 XML 파싱

다음은 XML을 파싱하고 XML에서 데이터 요소를 추출하는 간단한 응용프로그램이다.

```
library(xml2)

data <- '
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>'

tree <- read_xml(data)

person_name <- xml_find_all(tree, ".//name") %>%
  xml_text()

person_email <- xml_find_all(tree, ".//email") %>%
  xml_attr('hide')

cat("Name: ", person_name, "\n",
    "Attr: ", person_email, "\n")
```

`xml2` 패키지 `read_xml()` 함수를 사용하여 XML 문자열 표현을 XML 노드 '나무(tree)'로 변환한다. XML이 나무구조로 되었을 때, XML에서 데이터 일부분을 추출하기 위해서 호출하는 함수가 있다.

`xml_find_all()` 함수는 XML 나무를 훑어서 특정한 태그와 매칭되는 **노드(node)**를

검색한다. 각 노드는 텍스트, 속성(즉, hide 같은), 그리고 "자식(child)" 노드로 구성된다. 각 노드는 노드 나무의 최상단이 될 수 있다.

```
Name:  Chuck
 Attr:  yes
```

`xml2`같은 XML 패키지를 사용하는 것은 장점이 있다. 상기 예제의 XML은 매우 간단하지만, 적합한 XML에 관해서 규칙이 많이 있고, XML 구문 규칙에 얽매이지 않고 `xml2`를 사용해서 XML에서 데이터를 추출할 수 있다.

## 15.3 노드 반복하기

종종 XML이 다중 노드를 가지고 있어서 모든 노드를 처리하는 루프를 작성할 필요가 있다. 다음 프로그램에서 모든 `user` 노드를 루프로 반복한다.

```
library(xml2)

input <- '
    <stuff>
        <users>
            <user x="2">
                <id>001</id>
                <name>Chuck</name>
            </user>
            <user x="7">
                <id>009</id>
                <name>Brent</name>
            </user>
        </users>
    </stuff>'

stuff <- read_xml(input)

lst <- xml_find_all(stuff, "//users/user")

cat("User count:", length(lst), "\n")
```

```
## User count: 2
```

```
for(i in 1:length(lst)) {

  lst_name <- xml_find_all(lst[[i]], ".//name") %>% xml_text()
  lst_id <- xml_find_all(lst[[i]], ".//id") %>% xml_text()
  lst_attr <- xml_attr(lst[[i]], "x")

  cat("Name: ", lst_name, "\n")
  cat("Id: ", lst_id, "\n")
```

```
  cat("Attribute: ", lst_attr, "\n")
}
```

```
## Name:  Chuck
## Id:  001
## Attribute:  2
## Name:  Brent
## Id:  009
## Attribute:  7
```

`xml_find_all()` 함수는는 R 리스트의 하위 나무를 가져온다. 리스트는 XML 나무에서 `user` 구조를 표현한다. 그리고 나서, `for` 루프를 작성해서 각 `user` 노드 값을 확인하고 `name`, `id` 텍스트 요소와 `user` 노드에서 `x` 속성도 가져와서 출력한다.

```
Name:  Chuck
Id:  001
Attribute:  2
Name:  Brent
Id:  009
Attribute:  7
```

## 15.4 JSON

JSON(JavaScript Object Notation) 형식은 자바스크립트 언어에서 사용되는 객체와 배열 형식에서 영감을 얻었다. 하지만 파이썬이 자바스크립트 이전에 개발되어서 딕셔너리와 리스트의 파이썬 구문이 JSON 구문에 영향을 주었다. 그래서 JSON 포맷이 거의 파이썬 리스트와 딕셔너리의 조합과 일치한다.

상기 간단한 XML에 대략 상응하는 JSON으로 작성한 것이 다음에 있다.

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

몇가지 차이점에 주목하세요. 첫째로 XML에서는 “phone” 태그에 “intl”같은 속성을 추가할 수 있다. JSON 에서는 단지 키-값 페어(key-value pair)다. 또한 XML “person” 태그는 사라지고 외부 중괄호 세트로 대체되었다.

일반적으로 JSON 구조가 XML 보다 간단하다. 왜냐하면, JSON 이 XML보다 적은 역량을 보유하기 때문이다. 하지만 JSON 이 딕셔너리와 리스트의 조합에 직접

매핑된다는 장점이 있다. 그리고, 거의 모든 프로그래밍 언어가 파이썬 딕셔너리와 리스트에 상응하는 것을 갖고 있어서, JSON 이 협업하는 두 프로그램 사이에서 데이터를 교환하는 매우 자연스러운 형식이 된다.

XML에 비해서 상대적으로 단순하기 때문에, JSON이 응용프로그램 간 거의 모든 데이터를 교환하는데 있어 빠르게 선택되고 있다.

## 15.5 JSON 파싱하기

딕셔너리(객체)와 리스트를 중첩함으로써 JSON을 생성한다. 이번 예제에서, user 리스트를 표현하는데, 각 user가 키-값 페어(key-value pair, 즉, 딕셔너리)다. 그래서 리스트 딕셔너리가 있다.

다음 프로그램에서 내장된 **json** 라이브러리를 사용하여 JSON을 파싱하여 데이터를 읽어온다. 이것을 상응하는 XML 데이터, 코드와 비교해 보세요. JSON은 조금 덜 정교해서 사전에 미리 리스트를 가져오고, 리스트가 사용자이고, 각 사용자가 키-값 페어 집합임을 알고 있어야 한다. JSON은 좀더 간략(장점)하고 하지만 좀더 덜 서술적(단점)이다.

```
library(jsonlite)

input <- '
  [
    { "id" : "001",
      "x" : "2",
      "name" : "Chuck"
    } ,
    { "id" : "009",
      "x" : "7",
      "name" : "Chuck"
    }
  ]'

input_df <- jsonlite::fromJSON(input)
```

JSON과 XML에서 데이터를 추출하는 코드를 비교하면, `jsonlite` 패키지 `fromJSON()` 함수는 JSON 파일을 즉시 정형 데이터프레임을 변환시킨다.

프로그램 출력은 정확하게 상기 XML 버젼과 동일한 정보를 데이터프레임으로 표현하고 있어 후속 작업에 훨씬 유연하게 대응할 수 있다.

```
input_df
```

```
##    id x  name
## 1 001 2 Chuck
## 2 009 7 Chuck
```

일반적으로 웹서비스에 대해서 XML에서 JSON으로 옮겨가는 산업 경향이

뚜렷하다. JSON이 프로그래밍 언어에서 이미 갖고 있는 네이티브 자료 구조와 좀더 직접적이며 간단히 매핑되기 때문에, JSON을 사용할 때 파싱하고 데이터 추출하는 코드가 더욱 간단하고 직접적이다. 하지만 XML 이 JSON 보다 좀더 자기 서술적이고 XML 이 강점을 가지는 몇몇 응용프로그램 분야가 있다. 예를 들어, 대부분의 워드 프로세서는 JSON보다는 XML을 사용하여 내부적으로 문서를 저장한다.

## 15.6 API(응용 프로그램 인터페이스)

이제 HTTP를 사용하여 응용프로그램간에 데이터를 교환할 수 있게 되었다. 또한, XML 혹은 JSON을 사용하여 응용프로그램간에도 복잡한 데이터를 주고 받을 수 있는 방법을 습득했다.

다음 단계는 상기 학습한 기법을 사용하여 응용프로그램 간에 "계약(contract)"을 정의하고 문서화한다. 응용프로그램-대-응용프로그램 계약에 대한 일반적 명칭은 API **응용 프로그램 인터페이스(Application Program Interface)** 다. API를 사용할 때, 일반적으로 하나의 프로그램이 다른 응용 프로그램에서 사용할 수 있는 가능한 서비스 집합을 생성한다. 또한, 다른 프로그램이 서비스에 접근하여 사용할 때 지켜야하는 API (즉, "규칙")도 게시한다.

다른 프로그램에서 제공되는 서비스에 접근을 포함하여 프로그램 기능을 개발할 때, 이러한 개발법을 SOA, **Service-Oriented Architecture(서비스 지향 아키텍처)**라고 부른다. SOA 개발 방식은 전반적인 응용 프로그램이 다른 응용 프로그램 서비스를 사용하는 것이다. 반대로, SOA가 아닌 개발방식은 응용 프로그램이 하나의 독립된 응용 프로그램으로 구현에 필요한 모든 코드를 담고 있다.

웹을 사용할 때 SOA 사례를 많이 찾아 볼 수 있다. 웹사이트 하나를 방문해서 비행기표, 호텔, 자동차를 단일 사이트에서 예약완료한다. 호텔관련 데이터는 물론 항공사 컴퓨터에 저장되어 있지 않다. 대신에 항공사 컴퓨터는 호텔 컴퓨터와 계약을 맺어 호텔 데이터를 가져와서 사용자에게 보여준다. 항공사 사이트를 통해서 사용자가 호텔 예약을 동의할 경우, 항공사 사이트에서 호텔 시스템의 또다른 웹서비스를 통해서 실제 예약을 한다. 전체 거래(transaction)를 완료하고 카드 결재를 진행할 때, 다른 컴퓨터가 프로세스에 관여하여 처리한다.

서비스 지향 아키텍쳐는 많은 장점이 있다. (1) 항상 단 하나의 데이터만 유지관리한다. 이중으로 중복 예약을 원치 않는 호텔 같은 경우에 매우 중요하다. (2) 데이터 소유자가 데이터 사용에 대한 규칙을 정한다. 이러한 장점으로, SOA 시스템은 좋은 성능과 사용자 요구를 모두 만족하기 위해서 신중하게 설계되어야 한다.

응응프로그램이 웹상에 이용가능한 API로 서비스 집합을 만들 때, **웹서비스(web services)**라고 부른다.

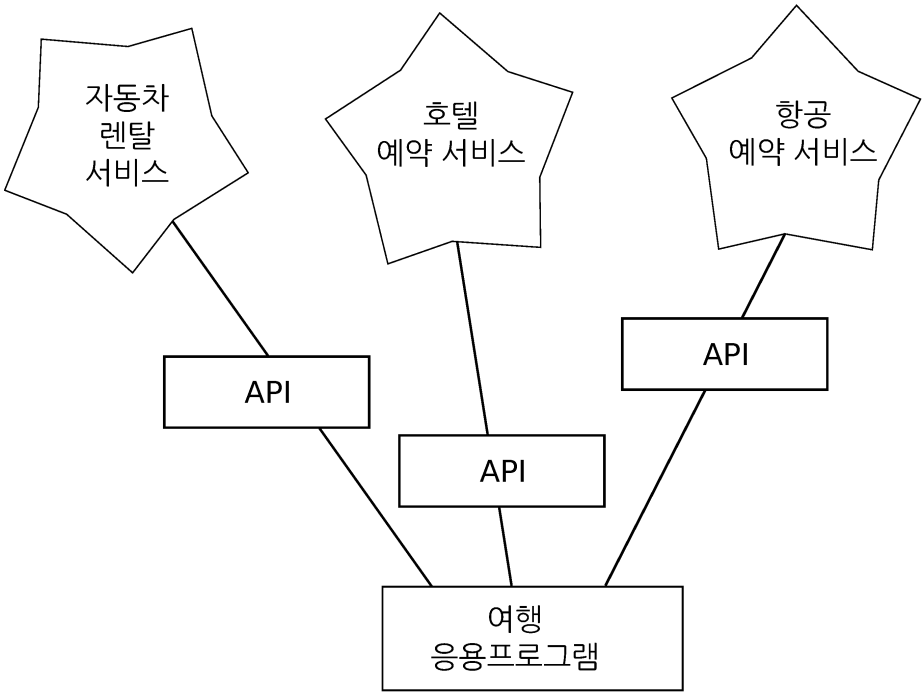



Figure 15.2: 서비스 지향 아키텍쳐

## 15.7 지오코딩 웹서비스

구글이 자체적으로 구축한 대용량 지리 정보 데이터베이스를 누구나 이용할 수 있게 하는 훌륭한 웹서비스가 있다. “Ann Arbor, MI” 같은 지리 검색 문자열을 지오코딩 API 에 넣으면, 검색 문자열이 의미하는 지도상에 위치와 근처 주요 지형지물 정보를 나름 최선을 다해서 예측 제공한다.

지오코딩 서비스는 무료지만 사용량이 제한되어 있어서, 상업적 응용프로그램에 API를 무제한 사용할 수는 없다. 하지만, 최종 사용자가 자유형식 입력 박스에 위치정보를 입력하는 설문 데이터가 있다면, 구글 API를 사용하여 데이터를 깔끔하게 정리하는데는 유용하다.

*구글 지오코딩 API 같은 무료 API를 사용할 때, 자원 사용에 대한 지침을 준수해야 한다. 너무나 많은 사람이 서비스를 남용하게 되면, 구글은 무료 서비스를 중단하거나, 상당부분 줄일 수 있다. (When you are using a free API like Google’s geocoding API, you need to be respectful in your use of these resources. If too many people abuse the service, Google might drop or significantly curtail its free service.)

서비스에 대해서 자세한 사항을 온라인 문서를 정독할 수 있지만, 무척 간단해서 브라우저에 다음 URL을 입력해서 테스트까지 할 수 있다.

http://maps.googleapis.com/maps/api/geocode/json?sensor=false &address=Ann+Arbor%2C+MI

웹프라우저에 붙여넣기 전에, URL만 뽑아냈고 URL에서 모든 공백을 제거했는지 확인하세요.

다음은 간단한 응용 프로그램이다. 사용자가 검색 문자열을 입력하고 구글 지오코딩 API를 호출하여 반환된 JSON에서 정보를 추출한다.

구글 지리정보 API는 상용으로 전환되었기에 다음카카오 지도 API를 대체하여 동일한 개발작업을 수행한다.

```
library(tidyverse)
library(httr)
library(jsonlite)

# usethis::edit_r_environ()

#       .
kpmg_addr <- '         737'

# HTTP       .
kpmg_res <- GET(url = 'https://dapi.kakao.com/v2/local/search/address.json',
                query = list(query = kpmg_addr),
                add_headers(Authorization = paste0("KakaoAK ", DAUM_MAP_API_KEY)))

# KPMG
```

```
kpmg_list <- kpmg_res %>%
  content(as = 'text') %>%
  fromJSON()

##
kpmg_list$documents$road_address %>%
  select(  = address_name,   = x,  =y)
```

```
1         152 127.036508620542 37.5000242405515
```

프로그램이 사용자로부터 검색 문자열을 받는다. 적절히 인코딩된 매개 변수로 검색문자열을 변환하여 URL을 만든다. 그리고 나서 **httr** 패키지를 사용하여 카카오 지오코딩 API에서 텍스트를 가져온다. 고정된 웹페이지와 달리, 반환되는 데이터는 전송한 매개변수와 카카오 서버에 저장된 지리정보 데이터에 따라 달라진다.

JSON 데이터를 가져오면, **jsonlite** 패키지로 파싱하고 전송받은 데이터가 올바른지 확인하는 몇가지 절차를 거친 후에 찾고자 하는 정보를 추출한다.

`Rscript` 프로그램 실행을 위해서 사용자 입력과 API키 외부 유출 방지를 위한 조치를 취한 후에 일반화를 위해 코드를 일부 수정한다.

```
library(tidyverse)
library(httr)
library(jsonlite)

#       .

typeline <- function(msg = "       - ") {
  if (interactive() ) {
    url <- readline(msg)
  } else {
    cat(msg);
    url <- readLines("stdin",n=1);
  }
  return(url)
}

address <- typeline() # '         '737'

# HTTP       .
address_res <- GET(url = 'https://dapi.kakao.com/v2/local/search/address.json',
                   query = list(query = address),
                   add_headers(Authorization = paste0("KakaoAK ", Sys.getenv("DAUM_MAP_API_KEY"))) )

#
address_list <- address_res %>%
  content(as = 'text') %>%
```

```
  fromJSON()

##
address_list$documents$road_address %>%
  select(  = address_name,   = x,  =y)
```

프로그램 출력결과는 다음과 같다.

```
$ Rscript script/geocoding.R

      - '         737'

1         152 127.036508620542 37.5000242405515
```

다음카카오 지도 API 외 다른 지오코딩 관련 자세한 사항은 공간통계를 위한 데이터 사이언스 - 지리정보 API - 주소와 위도경도 웹사이트를 참조한다.

## 15.8 보안과 API 사용

상용업체 API를 사용하기 위해서는 일종의 “API키(API key)”가 일반적으로 필요하다. 서비스 제공자 입장에서 누가 서비스를 사용하고 있으며 각 사용자가 얼마나 사용하고 있는를 알고자 한다. 상용 API 제공업체는 서비스에 대한 무료 사용자와 유료 사용자에 대한 구분을 두고 있다. 특정 기간 동안 한 개인 사용자가 사용할 수 있는 요청수에 대해 제한을 두는 정책을 두고 있다.

때때로 API키를 얻게 되면, API를 호출할 때 POST 데이터의 일부로 포함하거나 URL의 매개변수로 키를 포함시킨다.

또 다른 경우에는 업체가 서비스 요청에 대한 보증을 강화해서 공유키와 비밀번호를 암호화된 메시지 형식으로 보내도록 요구한다. 인터넷을 통해서 서비스 요청을 암호화하는 일반적인 기술을 **OAuth**라고 한다. http://www.oauth.net 사이트에서 OAuth 프로토콜에 대해 더 많은 정보를 만날 수 있다.

트위터 API가 점차적으로 가치있게 됨에 따라 트위터가 공개된 API에서 API를 매번 호출할 때마다 OAuth 인증을 거치도록 API를 바뀌었다. 다행스럽게도 편리한 OAuth 라이브러리가 많이 있다.

그래서 명세서를 읽고 아무것도 없는 상태에서 OAuth 구현하는 것을 필할 수 있게 되었다. 이용 가능한 라이브러리는 복잡성도 다양한만틈 기능적으로도 다양하다. OAuth 웹사이트에서 다양한 OAuth 라이브러리 정보를 확인할 수 있다.

OAuth 보안 요구사항을 충족하기 위해 추가된 다양한 매개 변수 의미를 좀더 자세히 알고자 한다면, OAuth 명세서를 읽어보기 바란다.

이와 같은 보안 API키는 누가 트위터 API를 사용하고 어느 정도 수준으로 트위터를 사용하는지에 대해서 트위트가 확고한 신뢰를 갖게 한다. 사용량에 한계를 두고 서비스를 제공하는 방식은 단순히 개인적인 목적으로 데이터 검색을 할 수는 있지만,

하루에 수백만 API 호출로 데이터를 추출하여 제품을 개발 못하게 제한하는 기능도 동시에 한다.

## 15.9 용어정의

- **API**: 응용 프로그램 인터페이스(Application Program Interface) - 두 응용 프로그램 컴포넌트 간에 상호작용하는 패턴을 정의하는 응용 프로그램 간의 계약.
- **ElementTree**: XML데이터를 파싱하는데 사용되는 파이썬 내장 라이브러리.
- **JSON**: JavaScript Object Notation- 자바스크립트 객체(JavaScript Objects) 구문을 기반으로 구조화된 데이터 마크업(markup)을 허용하는 형식.
- **REST**: REpresentational State Transfer - HTTP 프로토콜을 사용하여 응용 프로그램 내부에 자원에 접근을 제공하는 일종의 웹서비스 스타일.
- **SOA**: 서비스 지향 아키텍처(Service Oriented Architecture) - 응용 프로그램이 네트워크에 연결된 컴포넌트로 구성될 때.
- **XML**: 확장 마크업 언어(eXtensible Markup Language) - 구조화된 데이터의 마크업을 허용하는 형식.

# Chapter 16

# 데이터베이스와 SQL

## 16.1 데이터베이스가 뭔가요?

**데이터베이스(database)**는 데이터를 저장하기 위한 목적으로 조직된 파일이다. 대부분의 데이터베이스는 **키(key)**와 **값(value)**를 매핑한다는 의미에서 딕셔너리처럼 조직되었다. 가장 큰 차이점은 데이터베이스는 디스크(혹은 다른 영구 저장소)에 위치하고 있어서, 프로그램 종료 후에도 정보가 계속 저장된다. 데이터베이스가 영구 저장소에 저장되어서, 컴퓨터 주기억장치(memory) 크기에 제한받는 딕셔너리보다 훨씬 더 많은 정보를 저장할 수 있다.

딕셔너리처럼, 데이터베이스 소프트웨어는 엄청난 양의 데이터 조차도 매우 빠르게 삽입하고 접근하도록 설계되었다. 컴퓨터가 특정 항목으로 빠르게 찾아갈 수 있도록 데이터베이스에 **인덱스(indexes)**를 추가한다. 데이터베이스 소프트웨어는 인덱스를 구축하여 성능을 보장한다.

다양한 목적에 맞춰 서로 다른 많은 데이터베이스 시스템이 개발되어 사용되고 있다. Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite이 여기에 포함된다. 이 책에서는 SQLite를 집중해서 살펴볼 것이다. 왜냐하면 매우 일반적인 데이터베이스이며 파이썬에 이미 내장되어 있기 때문이다. 응용프로그램 내부에서 데이터베이스 기능을 제공하도록 SQLite가 다른 응용프로그램 내부에 내장(embedded)되도록 설계되었다. 예를 들어, 다른 많은 소프트웨어 제품이 그렇듯이, 파이어폭스 브라우져도 SQLite를 사용한다.

- http://sqlite.org/

이번 장에서 기술하는 트위터 스파이더링 응용프로그램처럼 정보과학(Informatics)에서 마주치는 몇몇 데이터 조작 문제에 SQLite가 적합하다.

## 16.2 데이터베이스 개념

처음 데이터베이스를 볼때 드는 생각은 마치 엑셀같은 다중 시트를 지닌 스프레드쉬트(spreadsheet)같다는 것이다. 데이터베이스에서 주요 데이터 구조물은 테이블(tables), 행(rows), and 열(columns)이 된다.

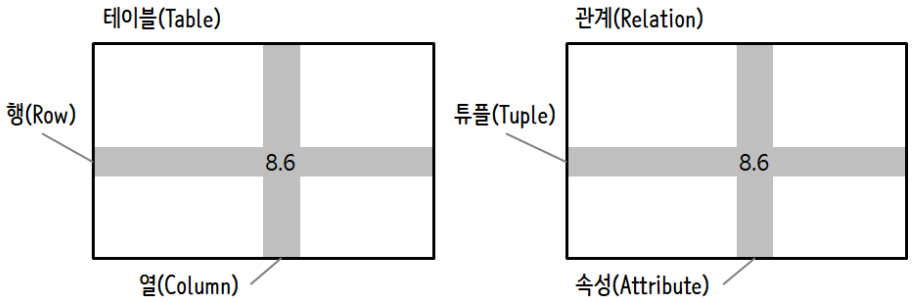



Figure 16.1: 데이터베이스 개념

관계형 데이터베이스의 기술적인 면을 설명하면 테이블, 행, 열의 개념은 **관계(relation)**, **튜플(tuple)**, **속성(attribute)** 각각 형식적으로 매칭된다. 이번 장에서는 조금 덜 형식 용어를 사용한다.

## 16.3 파이어폭스 SQLite 관리자

SQLite 데이터베이스 파일에 있는 데이터를 다루기 위해서 이번장에서 주로 R 사용에 집중을 하지만, 다음 웹사이트에서 무료로 이용 가능한 SQLite 데이터베이스 매니저(SQLite Database Manager)로 불리는 **파이어폭스 애드온(add-on)**을 사용해서 좀더 쉽게 많은 작업을 수행할 수 있다. 파이어폭스 애드온은 크롬 확장 프로그램과 유사한 개념으로 파이어폭스는 개발자들이 많이 사용하는 웹브라우져 중 하나다.

- https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/

브라우져를 사용해서 쉽게 테이블을 생성하고, 데이터를 삽입, 편집하고 데이터베이스 데이터에 대해 간단한 SQL 질의를 실행할 수 있다.

이러한 점에서 데이터베이스 매니저는 텍스트 파일을 작업할 때 사용하는 텍스트 편집기와 유사하다. 텍스트 파일에 하나 혹은 몇개 작업만 수행하고자 하면, 텍스트 편집기에서 파일을 열어 필요한 수정작업을 하고 닫으면 된다. 텍스트 파일에 작업할 사항이 많은 경우는 종종 간단한 R 프로그램을 작성하여 수행한다. 데이터베이스로 작업할 때도 동일한 패턴이 발견된다. 간단한 작업은 데이터베이스 매니저를 통해서 수행하고, 좀더 복잡한 작업은 R로 수행하는 것이 더 편리하다.

## 16.4 데이터베이스 테이블 생성

데이터베이스는 R 리스트 혹은 딕셔너리보다 좀더 명확히 정의된 구조를 요구한다.[1]

데이터베이스에 테이블(table)을 생성할 때, 열(column)의 명칭과 각 열(column)에 저장하는 테이터 형식을 사전에 정의해야 한다. 데이터베이스 소프트웨어가 각 열의 데이터 형식을 인식하게 되면, 데이터 형식에 따라 데이터를 저장하고 찾아오는 방법을 가장 효율적인 방식을 선택할 수 있다.

다음 url에서 SQLite에서 지원되는 다양한 데이터 형식을 살펴볼 수 있다.

- http://www.sqlite.org/datatypes.html

처음에는 데이터 구조를 사전에 정의하는 것이 불편하게 보이지만, 대용량의 데이터가 데이터베이스에 포함되더라도 데이터의 빠른 접근을 보장하는 잇점이 있다.

데이터베이스 파일과 데이터베이스에 두개의 열을 가진 `Tracks` 이름의 테이블을 생성하는 코드는 다음과 같다.

```
library(RSQLite)

music_db  <- "data/music.sqlite"
conn <- dbConnect(drv = SQLite(), dbname= music_db)

dbSendQuery(conn, "INSERT INTO Tracks (title, plays) VALUES ( ?, ? )", c('Thunderstruck', 20))
dbSendQuery(conn, "INSERT INTO Tracks (title, plays) VALUES ( ?, ? )", c('My Way', 15))

dbDisconnect(conn)
```

**연결 (connect)** 연산은 현재 디렉토리 `data/music.sqlite3` 파일에 저장된 데이터베이스에 "연결(connection)"한다. 파일이 존재하지 않으면, 자동 생성된다. "연결(connection)"이라고 부르는 이유는 때때로 데이터베이스가 응용프로그램이 실행되는 서버로부터 분리된 "데이터베이스 서버(database server)"에 저장되기 때문이다. 지금 간단한 예제 파일의 경우에 데이터베이스가 로컬 파일 형태로 R 코드 마찬가지로 동일한 디렉토리에 있다.

파일을 다루는 **파일 핸들(file handle)**처럼 데이터베이스에 저장된 파일에 연산을 수행하기 위해서 **커서(cursor)**를 사용한다. `cursor()`를 호출하는 것은 개념적으로 텍스트 파일을 다룰 때 `readLines()`을 호출하는 것과 개념적으로 매우 유사하다.

커서가 생성되면, `dbGetQuery()` 함수를 사용하여 데이터베이스 콘텐츠에 명령어 실행을 할 수 있다.

데이터베이스 명령어는 특별한 언어로 표현된다. 단일 데이터베이스 언어를 학습하도록 서로 다른 많은 데이터베이스 업체 사이에서 표준화되었다.

[1] 실질적으로 SQLite는 열에 저장되는 데이터 형식에 대해서 좀더 많은 유연성을 부여하지만, 이번 장에서는 데이터 형식을 엄격하게 적용해서 MySQL 같은 다른 관계형 데이터베이스 시스템에도 동일한 개념이 적용되게 한다.

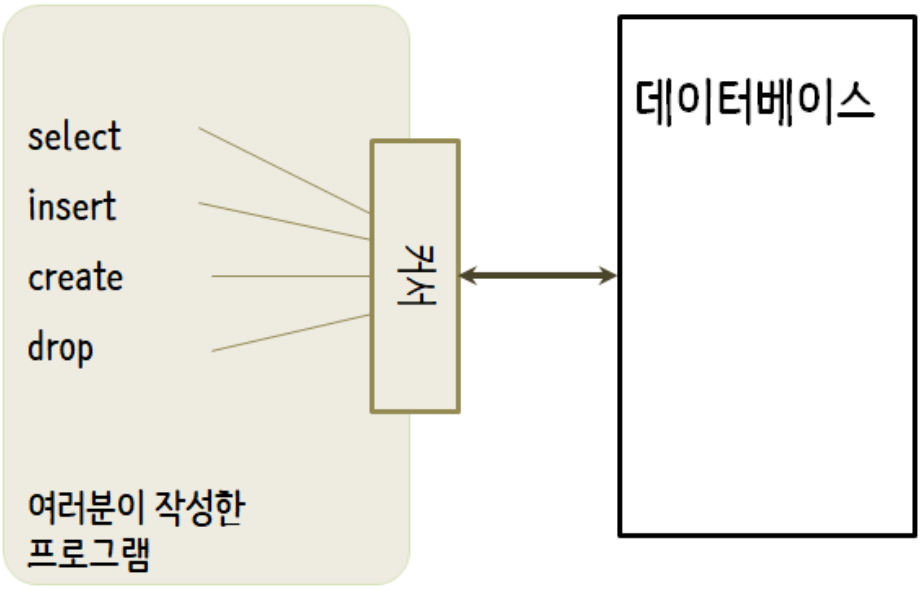



Figure 16.2: 데이터베이스 커서

데이터베이스 언어를 **SQL(Structured Query Language 구조적 질의 언어)**로 부른다.

- http://en.wikipedia.org/wiki/SQL

상기 예제에서, 데이터베이스에 두개의 SQL 명령어를 실행했다. 관습적으로 데이터베이스 키워드는 대문자로 표기한다. 테이블명이나 열의 명칭처럼 사용자가 추가한 명령어 부분은 소문자로 표기한다.

첫 SQL 명령어는 만약 존재한다면 데이터베이스에서 `Tracks` 테이블을 삭제한다. 동일한 프로그램을 실행해서 오류 없이 반복적으로 `Tracks` 테이블을 생성하도록하는 패턴이다. `DROP TABLE` 명령어는 데이터베이스 테이블 및 테이블 콘텐츠 전부를 삭제하니 주의한다. (즉, "실행취소(undo)"가 없다.)

```
`dbGetQuery(conn, 'DROP TABLE IF EXISTS Tracks ') `
```

두번째 명령어는 title 문자형 열과 plays 정수형 열을 가진 Tracks으로 명명된 테이블을 생성한다.

```
`dbGetQuery(conn, 'CREATE TABLE Tracks (title TEXT, plays INTEGER)')`
```

이제 Tracks으로 명명된 테이블을 생성했으니, SQL INSERT 연산을 통해 테이블에 데이터를 넣을 수 있다. 다시 한번, 데이터베이스에 연결하여 커서(cursor)를 얻어 작업을 시작한다. 그리고 나서 커서를 사용해서 SQL 명령어를 수행한다.

SQL `INSERT` 명령어는 어느 테이블을 사용할지 특정한다. 그리고 나서 `(title, plays)` 포함할 필드 목록과 테이블 새로운 행에 저장될 VALUES 나열해서 신규 행을

정의를 마친다. 실제 값이 execute() 호출의 두번째 매개변수로 튜플 ('My Way', 15) 로 넘겨는 것을 표기하기 위해서 값을 물음표 (?, ?)로 명기한다.

```
library(RSQLite)

music_db  <- "data/music.sqlite"
conn <- dbConnect(drv = SQLite(), dbname= music_db)

dbSendQuery(conn, "INSERT INTO Tracks (title, plays) VALUES ( ?, ? )",
            c('Thunderstruck', 20))
dbSendQuery(conn, "INSERT INTO Tracks (title, plays) VALUES ( ?, ? )",
            c('My Way', 15))

print('Tracks:')

dbGetQuery(conn, 'SELECT title, plays FROM Tracks')

dbSendQuery(conn, "DELETE FROM Tracks WHERE plays < 100")

dbDisconnect(conn)
```

먼저 테이블에 두개 열을 삽입(INSERT)하여 데이터를 데이터베이스에 저장되도록 했다. 그리고 나서, SELECT 명령어를 사용하여 테이블에 방금 전에 삽입한 행을 불러왔다. SELECT 명령어에서 데이터를 어느 열(title, plays)에서, 어느 테이블Tracks에서 가져올지 명세한다. 프로그램 실행결과는 다음과 같다.

```
> dbGetQuery(conn, 'SELECT title, plays FROM Tracks')
          title plays
1 Thunderstruck    20
2        My Way    15
```

프로그램 마지막에 SQL 명령어를 실행 사용해서 방금전에 생성한 행을 모두 삭제(DELETE)했기 때문에 프로그램을 반복해서 실행할 수 있다. 삭제(DELETE) 명령어는 WHERE 문을 사용하여 선택 조건을 표현할 수 있다. 따라서 명령문에 조건을 충족하는 행에만 명령문을 적용한다. 이번 예제에서 기준이 모든 행에 적용되어 테이블에 아무 것도 없게 된다. 따라서 프로그램을 반복적으로 실행할 수 있다. 삭제(DELETE)를 실행한 후에 데이터베이스에서 데이터를 완전히 제거했다.

## 16.5 SQL 요약

지금까지, R 예제를 통해서 SQL(Structured Query Language)을 사용했고, SQL 명령어에 대한 기본을 다루었다. 이번 장에서는 SQL 언어를 보고 SQL 구문 개요를 살펴본다.

대단히 많은 데이터베이스 업체가 존재하기 때문에 호환성의 문제로 SQL(Structured Query Language)이 표준화되었다. 그래서, 여러 업체가 개발한 데이터베이스 시스템 사이에 호환하는 방식으로 커뮤니케이션 가능하다.

관계형 데이터베이스는 테이블, 행과 열로 구성된다. 열(column)은 일반적으로 텍스트, 숫자, 혹은 날짜 자료형을 갖는다. 테이블을 생성할 때, 열의 명칭과 자료형을 지정한다.

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

테이블에 행을 삽입하기 위해서 SQL INSERT 명령어를 사용한다.

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

INSERT 문장은 테이블 이름을 명기한다. 그리고 나서 새로운 행에 놓고자 하는 열/필드 리스트를 명시한다. 그리고 나서 키워드 VALUES와 각 필드 별로 해당하는 값을 넣는다.

SQL SELECT 명령어는 데이터베이스에서 행과 열을 가져오기 위해 사용된다. SELECT 명령문은 가져오고자 하는 행과 WHERE절을 사용하여 어느 행을 가져올지 지정한다. 선택 사항으로 ORDER BY 절을 이용하여 반환되는 행을 정렬할 수도 있다.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

* 을 사용하여 WHERE 절에 매칭되는 각 행의 모든 열을 데이터베이스에서 가져온다.

주목할 점은 R과 달리 SQL WHERE 절은 등식을 시험하기 위해서 두개의 등치 기호 대신에 단일 등치 기호를 사용한다. WHERE에서 인정되는 다른 논리 연산자는 <,>,<=,>=,!= 이고, 논리 표현식을 생성하는데 AND, OR, 괄호를 사용한다.

다음과 같이 반환되는 행이 필드값 중 하나에 따라 정렬할 수도 있다.

```
SELECT title,plays FROM Tracks ORDER BY title
```

행을 제거하기 위해서, SQL DELETE 문장에 WHERE 절이 필요하다. WHERE 절이 어느 행을 삭제할지 결정한다.

```
SELECT title,plays FROM Tracks ORDER BY title
```

다음과 같이 SQL UPDATE 문장을 사용해서 테이블에 하나 이상의 행 내에 있는 하나 이상의 열을 갱신(UPDATE)할 수 있다.

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

UPDATE 문장은 먼저 테이블을 명시한다. 그리고 나서, SET 키워드 다음에 변경할 필드 리스트 와 값을 명시한다. 그리고 선택사항으로 갱신될 행을 WHERE절에 지정한다. 단일 UPDATE 문장은 WHERE절에서 매칭되는 모든 행을 갱신한다. 혹은 만약 WHERE절이 지정되지 않으면,테이블 모든 행에 대해서 갱신(UPDATE)을 한다.

네가지 기본 SQL 명령문(**INSERT**, **SELECT**, **UPDATE**, **DELETE**)은 데이터를 생성하고 유지 관리하는데 필요한 기본적인 4가지 작업을 가능케 한다.

## 16.6 데이터 모델링 기초

관계형 데이터베이스의 진정한 힘은 다중 테이블과 테이블 사이의 관계를 생성할 때 생긴다. 응용프로그램 데이터를 쪼개서 다중 테이블과 두 테이블 간에 관계를

설정하는 것을 **데이터 모델링(data modeling)**이라고 한다. 테이블 정보와 테이블 관계를 표현하는 설계 문서를 **데이터 모델(data model)**이라고 한다.

데이터 모델링(data modeling)은 상대적으로 고급 기술이여서 이번 장에서는 관계형 데이터 모델링의 가장 기본적인 개념만을 소개한다. 데이터 모델링에 대한 좀더 자세한 사항은 다음 링크에서 시작해 볼 수 있다.

- http://en.wikipedia.org/wiki/Relational_model

트위터 스파이더 응용프로그램으로 단순히 한 사람의 친구가 몇명인지 세는 대신에, 모든 관계 리스트를 가지고서 특정 계정에 팔로잉하는 모든 사람을 찾는다.

모두 팔로잉하는 계정을 많이 가지고 있어서, (Twitter) 테이블에 단순히 하나의 열만을 추가해서는 해결할 수 없다. 그래서 친구를 짝으로 추적할 수 있는 새로운 테이블을 생성한다. 다음이 간단하게 상기 테이블을 생성하는 방식이다.

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

drchuck을 팔로잉하는 사람을 마주칠 때마다, 다음과 같은 형식의 행을 삽입한다.

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

drchuck 트위터 피드에서 친구 20명을 처리하면서, "drchuck"을 첫 매개변수로 가지는 20개 레코드를 삽입해서 데이터베이스에 중복되는 많은 문자열이 생길 것이다.

문자열 데이터 중복은 **데이터베이스 정규화(database normalization)** 모범 사례(berst practice)를 위반하게 만든다. 기본적으로 데이터베이스 정규화는 데이터베이스에 결코 한번 이상 동일한 문자열을 저장하지 않는다. 만약 한번 이상 데이터가 필요하다면, 그 특정 데이터에 대한 숫자 **키(key)**를 생성하고, 그 키를 사용하여 실제 데이터를 참조한다.

실무에서, 문자열이 컴퓨터 주기억장치나 디스크에 저장되는 정수형 자료보다 훨씬 많은 공간을 차지하고 더 많은 처리시간이 비교나 정렬에 소요된다. 항목이 단지 수백개라면, 저장소나 처리 시간이 그다지 문제되지 않는다. 하지만, 데이터베이스에 수백만명의 사람 정보와 1억건 이상의 링크가 있다면, 가능한 빨리 데이터를 스캔하는 것이 정말 중요하다.

앞선 예제에서 사용된 Twitter 테이블 대신에 People로 명명된 테이블에 트위커 계정을 저장한다. People 테이블은 트위터 사용자에 대한 행과 관련된 숫자키를 저장하는 추가 열(column)이 있다. SQLite는 데이터 열의 특별한 자료형(INTEGER PRIMARY KEY)을 이용하여 테이블에 삽입할 임의 행에 대해서 자동적으로 키값을 추가하는 기능이 있다.

다음과 같이 추가적인 id 열을 가진 People 테이블을 생성할 수 있다.

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

People 테이블의 각 행에서 친구 숫자를 더 이상 유지관리하고 있지 않음을 주목하세요. id 열 자료형으로 INTEGER PRIMARY KEY 선택할 때 함축되는 의미는 다음과 같다., 사용자가 삽입하는 각 행에 대해서 SQLite가 자동으로 유일한 숫자

키를 할당하고 관리하게 한다. UNIQUE 키워드를 추가해서 SQLite에 name에 동일한 값을 가진 두 행을 삽입하지 못하게 한다.

상기 Pals 테이블을 생성하는 대신에, 데이터베이스에 from_id, to_id 두 정수 자료형 열을 지닌 Follows 테이블을 생성한다. Follows 테이블은 from_id과 to_id의 조합으로 테이블이 유일하다는 제약사항도 가진다. (즉, 중복된 행을 삽입할 수 없다.)

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

테이블에 UNIQUE절을 추가한다는 의미는 레코드를 삽입할 때 데이터베이스에서 지켜야하는 규칙 집합을 의사소통하는 것이다. 잠시 후에 보겠지만, 프로그램상에 편리하게 이러한 규칙을 생성한다. 이러한 규칙 집합은 실수를 방지하게 하고 코드를 작성을 간결하게 한다.

본질적으로 Follows 테이블을 생성할 때, "관계(relationship)"를 모델링하여 한 사람이 다른 사람을 "팔로우(follow)"하고 이것을 (a) 사람이 연결되어 있고, (b) 관계을 방향성이 나타나도록 숫자를 짝지어 표현한다.

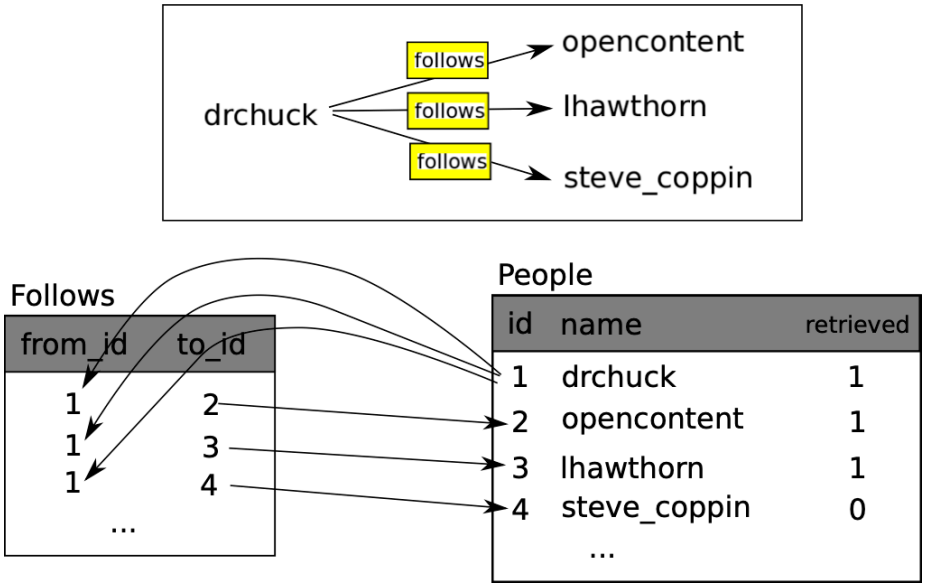



Figure 16.3: 트위터 관계 데이터베이스 모델링

### 16.6.1 테이블 제약사항

테이블 구조를 설계할 때, 데이터베이스 시스템에 몇 가지 규칙을 설정할 수 있다. 이러한 규칙은 실수를 방지하고 잘못된 데이터가 테이블에 들어가는 것을 막는다. 테이블을 생성할 때:

```
dbSendQuery( conn, 'CREATE TABLE IF NOT EXISTS People (id INTEGER PRIMARY KEY, name TE
```

```
dbSendQuery( conn, 'CREATE TABLE IF NOT EXISTS Follows (from_id INTEGER, to_id INTEGER, UNIQUE(fr
```

People 테이블에 name 칼럼이 (UNIQUE)함을 나타낸다. Follows 테이블의 각 행에서 두 숫자 조합은 유일하다는 것도 나타낸다. 하나 이상의 동일한 관계를 추가하는 것 같은 실수를 이러한 제약 사항을 통해서 방지한다.

다음 코드에서 이런 제약사항의 장점을 확인할 수 있다.

```
dbSendQuery( conn, 'INSERT OR IGNORE INTO People (name, retrieved) VALUES ( ?, 0)', c( 'friend',
```

INSERT 문에 OR IGNORE 절을 추가해서 만약 특정 INSERT가 "name이 유일(unique)해야 한다"를 위반하게 되면, 데이터베이스 시스템은 INSERT를 무시한다. 데이터베이스 제약 사항을 안전망으로 사용해서 무언가가 우연히 잘못되지 않게 방지한다.

마찬가지로, 다음 코드는 정확히 동일 Follows관계를 두번 추가하지 않는다.

```
dbSendQuery( conn, 'INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)', c(id, friend_i
```

다시 한번, Follows 행에 대해 지정한 유일한 제약사항을 위반하게 되면 INSERT 시도를 무시하도록 데이터베이스에 지시한다.

## 16.7 세 종류 키

지금까지 데이터를 다중 연결된 테이블에 넣고 **키(keys)**를 사용하여 행을 연결하는 방식으로 데이터 모델을 생성했는데, 키와 관련된 몇몇 용어를 살펴볼 필요가 있다. 일반적으로 데이터베이스 모델에서 세가지 종류의 키가 사용된다.

- **논리 키(logical key)**는 "실제 세상"이 행을 찾기 위해서 사용하는 키다. 데이터 모델 예제에서, name 필드는 논리키다. 사용자에 대해서 screen_name이고, name 필드를 사용하여 프로그램에서 여러번 사용자 행을 찾을 수 있다. 논리 키에 UNIQUE 제약 사항을 추가하는 것이 의미있다는 것을 종종 이해하게 된다. 논리 키는 어떻게 바깥 세상에서 행을 찾는지 다루기 때문에, 테이블에 동일한 값을 가진 다중 행이 존재한다는 것은 의미가 없다.

- **주키(primary key)**는 통상적으로 데이터베이스에서 자동 대입되는 숫자다. 프로그램 밖에서는 일반적으로 의미가 없고, 단지 서로 다른 테이블에서 행을 열결할 때만 사용된다. 테이블에 행을 찾을 때, 통상적으로 주키를 사용해서 행을 찾는 것이 가장 빠르게 행을 찾는 방법이다. 주키는 정수형이어서, 매우 적은 저장공간을 차지하고 매우 빨리 비교 혹은 정렬할 수 있다. 이번에 사용된 데이터 모델에서 id 필드가 주키의 한 예가 된다.

- **외부 키(foreign key)**는 일반적으로 다른 테이블에 연관된 행의 주키를 가리키는 숫자다. 이번에 사용된 데이터 모델의 외부 키의 사례는 from_id다.

주키 id필드명을 호출하고, 항상 외부키에 임의 필드명에 접미사로 _id 붙이는 명명규칙을 사용한다.

## 16.8 DVD 대여 데이터베이스

### 16.8.1 PostgreSQL [2] [3]

PostgreSQL은 확장 가능성 및 표준 준수를 강조하는 객체-관계형 데이터베이스 관리 시스템(ORDBMS)의 하나로 BSD 라이선스로 배포되며 오픈소스 개발자 및 관련 회사들이 개발에 참여하고 있다. 소규모의 단일 머신 애플리케이션에서부터 수많은 동시 접속 사용자가 있는 대형의 인터넷 애플리케이션(또는 데이터웨어하우스용)에 이르기까지 여러 부하를 관리할 수 있으며 macOS 서버의 경우 PostgreSQL은 기본 데이터베이스로 상용 오라클 데이터베이스를 대체하는 오픈소스 데이터베이스로 알려져 있다.

### 16.8.2 PostgreSQL 설치 [4]

PostgreSQL: The World's Most Advanced Open Source Relational Database 웹사이트에서 PostgreSQL 다운로드 한다. 윈도우에 설치하는 경우 다음을 참고한다. 설치과정에서 나중에 도움이 될만한 정보는 다음과 같다.

- 설치 디렉토리: `C:\Program Files\PostgreSQL\11`
- 포트: `5432`
- 사용자명: `postgres`

`PostgreSQL 11` → `SQL Shell (psql)`을 클릭한 후에 postgreSQL 헬로월드를 찍어본다. 설치과정에서 등록한 비번만 넣어주고 나머지는 로컬호스트와 기본 디폴트 설정된 데이터베이스를 사용할 것이라 `postgres` 사용자 비밀번호만 넣어준다. 그리고 나서 `postgre=#` 쉘에 `SELECT version()` 명령어를 넣어준다.

### 16.8.3 예제 데이터베이스 - `pagila` [5]

PostgreSQL Sample Database를 Github에서 구해서 설치하거나, PostgreSQL Sample Database, Load PostgreSQL Sample Database을 참조하여 DVD 대여 데이터베이스를 설치한다.

1. `SQL Shell (psql)` 쉘을 실행하여 `dvd` 데이터베이스를 생성한다.

```
Database [postgres]:
Port [5432]:
Username [postgres]:
postgres     :
psql (11.5)
      "help"     .
```

[2] 위키백과, "PostgreSQL"
[3] NAVER D2 한눈에 살펴보는 PostgreSQL
[4] Install PostgreSQL
[5] PostgreSQL Sample Database

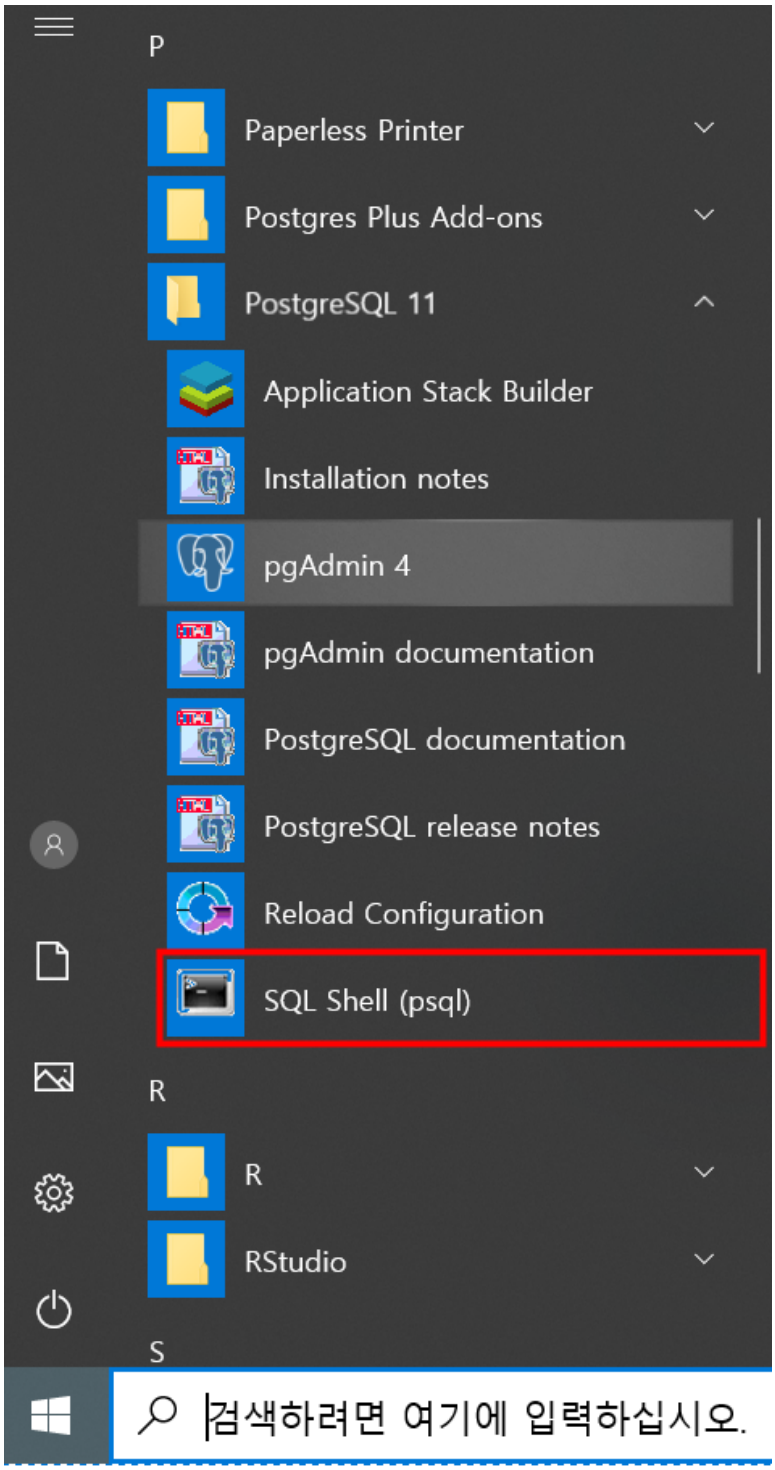



Figure 16.4: postgreSQL 설치

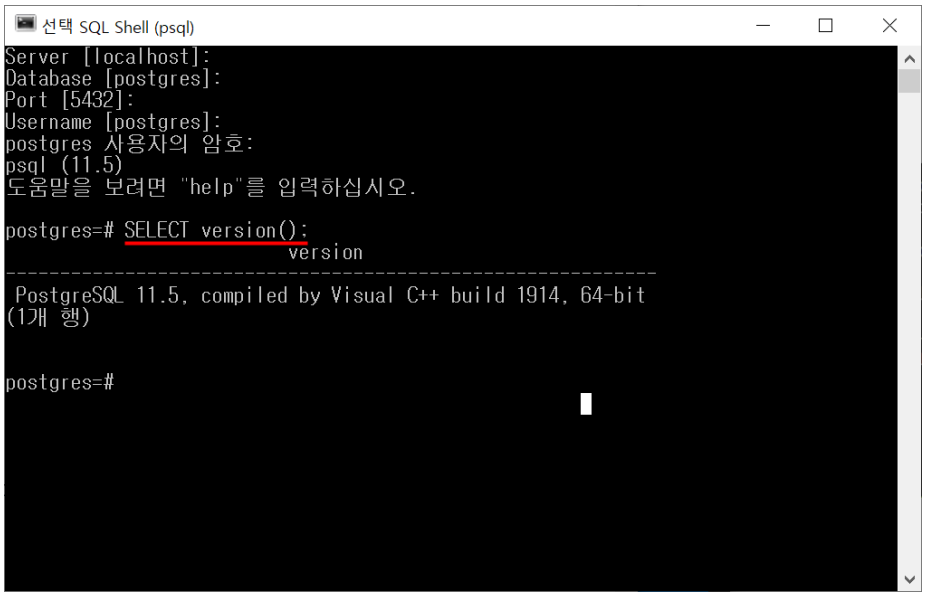



Figure 16.5: postgreSQL 헬로월드

```
postgres=# CREATE DATABASE dvd;
CREATE DATABASE
```

2. Windows + R 단축키를 실행시켜 cmd를 입력하여 윈도우 쉘을 구동시킨다. 그리고 postgreSQL을 설치한 윈도우 디렉토리로 이동한다. C:\Program Files\PostgreSQL\11\bin 디렉토리가 된다. 그리고 나서 다운받은 dvdrental.zip 파일 압축을 풀어 dvdrental.tar을 지정한다.

- pg_restore 명령어는 데이터베이스를 생성시키는 역할을 한다.
- -U postgres 인자는 사용자를 지정한다.
- -d dvd 인자는 데이터베이스를 지정한다.
- C:\dvdrental\dvdrental.tar 인자는 파일로 저장된 데이터베이스 정보를 담고 있다.

```
C:\Program Files\PostgreSQL\11\bin> pg_restore -U postgres -d dvd C:\dvdrental\dvdrenta
 :
C:\Program Files\PostgreSQL\11\bin>
```

### 16.8.4 DVD 대여 질의문 작성 [6]

dvd 데이터베이스가 설치되었기 때문에 쿼리를 던지기 위해서는 postgreSQL 데이터베이스에 접속을 해야한다. 이를 위해서 pgAdmin 4를 실행시키게 되면 웹브라우져에 웹인터페이스가 생기게 된다. 데이터베이스를 dvd로 지정하고 하고

[6] Connect To a PostgreSQL Database Server

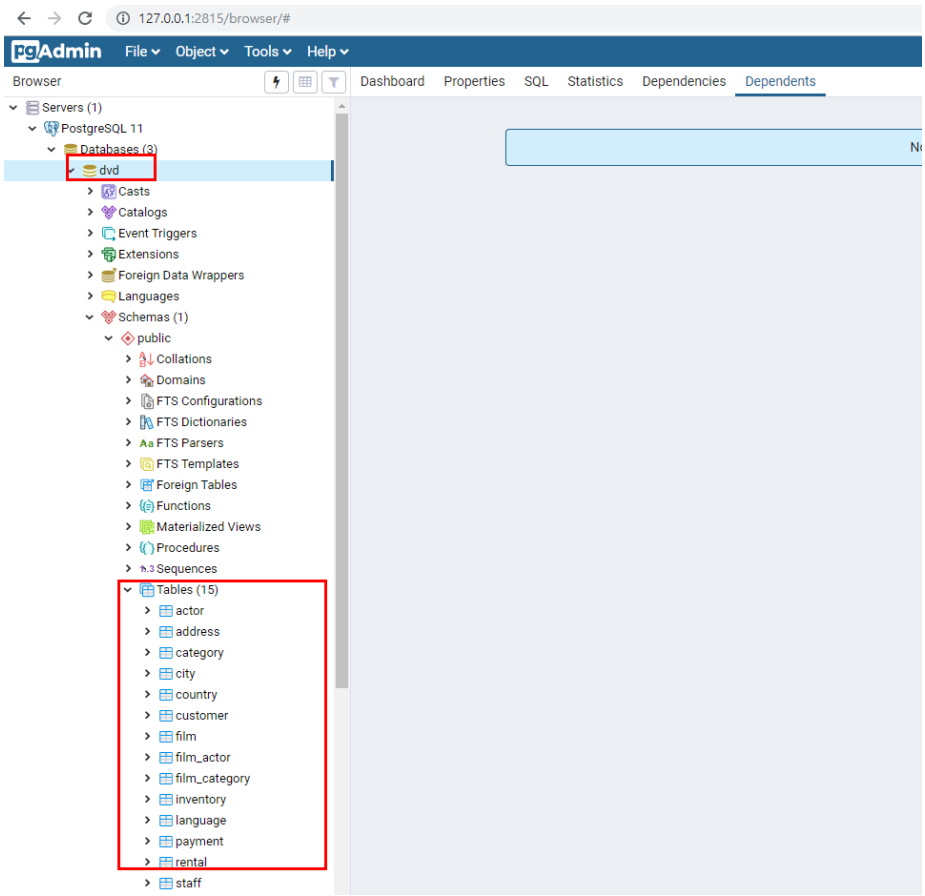



Figure 16.6: postgreSQL DVD 데이터베이스

나서, `Tools` → `Query Tool`을 클릭하게 되면 해당 데이터베이스 테이블에 쿼리를 던질 수가 있게 된다.

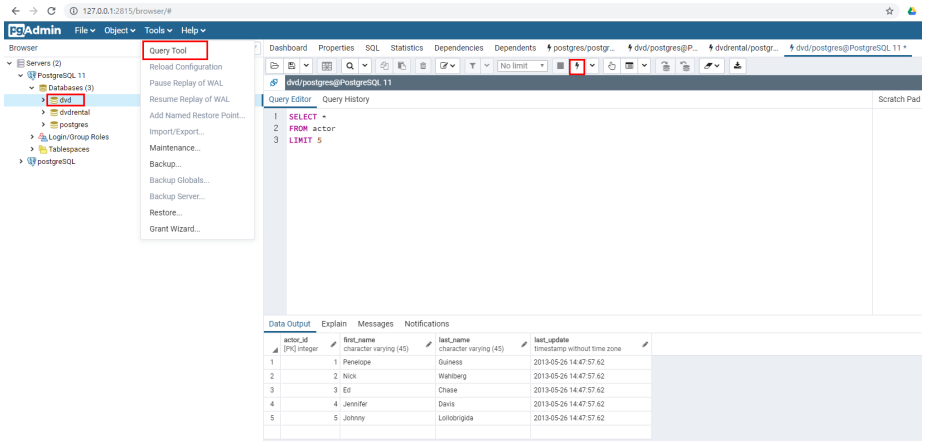



Figure 16.7: postgreSQL select 쿼리문 예시

#### 16.8.4.1 데이터베이스 사용자 추가

`postgres` 사용자는 이미 존재하기 때문에 별도로 `tidyverse` 사용자를 추가하고 권한을 부여한다. `\du` 명령어로 사용자가 정상 등록되었는지 확인한다.

```
postgres=# create user tidyverse with encrypted password '1234';
CREATE ROLE
postgres=# grant all privileges on database dvd to tidyverse;
GRANT
postgres=# \du

       |                                                    |     :
-----------+------------------------------------------------------+------------
 postgres  |    ,     , DB    ,   , RLS    | {}
 tidyverse |                                                      | {}
```

#### 16.8.4.2 R에서 postgreSQL 연결

`postgreSQL` DBMS 내부에 `dvd` 데이터베이스가 생성되었다. 이를 R에서 작업하기 위해서 `RPostgreSQL`, `DBI` 팩키지를 도입한다. `dbConnect()` 함수에 데이터베이스와 연결에 필요한 모든 정보를 저장시킨다. 그리고 나서 `dbGetQuery()` 함수로 쿼리를 던져 원하는 결과를 받아온다.

```
library(DBI)
library(RPostgres)
```

```
con <- dbConnect(RPostgres::Postgres(), dbname="dvd",
                 host="localhost",
                 port="5432",
                 user="postgres",
                 password="1234")

actor <- dbGetQuery(con, "SELECT * FROM actor LIMIT 5")

DBI::dbDisconnect(con)
```

dbGetQuery()로 가져온 데이터프레임을 dplyr 동사로 후속작업을 진행한다.

```
library(tidyverse)

actor %>%
  filter(actor_id ==1)
```

```
  actor_id first_name last_name         last_update
1        1   Penelope   Guiness 2013-05-26 14:47:57
```

#### 16.8.4.3 작업에 필요한 테이블 찾기

데이터베이스에서 쿼리 작업을 수행할 때 가장 먼저 해야 되는 일중의 하나가 적합한 테이블을 찾는 것이다. 이를 위해서 각 DBMS마다 나름대로 정리를 해둔 메타테이블이 존재한다. postgreSQL의 경우는 pg_catalog.pg_tables가 된다. 가장 많이 사용되는 SQL 데이터베이스별로 동일한 사안에 대해서 찾아보자.

- postgreSQL: SELECT * FROM pg_catalog.pg_tables;
- sqlite3: .tables
- MS SQL - Transact-SQL: SELECT * FROM INFORMATION_SCHEMA.TABLES;
- MySQL: SHOW TABLES;

```
qry <- "SELECT *
        FROM pg_catalog.pg_tables"

dbGetQuery(con, qry) %>%
  filter(schemaname == 'public')
```

```
   schemaname     tablename tableowner tablespace hasindexes hasrules hastriggers rowsecurity
1      public         actor   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
2      public         store   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
3      public       address   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
4      public      category   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
5      public          city   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
6      public       country   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
7      public      customer   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
8      public    film_actor   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
9      public film_category   postgres       <NA>       TRUE    FALSE        TRUE       FALSE
```

```
10     public     inventory   postgres       <NA>       TRUE    FALSE        TRUE
11     public      language   postgres       <NA>       TRUE    FALSE        TRUE
12     public        rental   postgres       <NA>       TRUE    FALSE        TRUE
13     public         staff   postgres       <NA>       TRUE    FALSE        TRUE
14     public       payment   postgres       <NA>       TRUE    FALSE        TRUE
15     public          film   postgres       <NA>       TRUE    FALSE        TRUE
```

#### 16.8.4.4 테이블 별 칼럼명

다음으로 테이블을 찾았다고 하면, 해당되는 칼럼명을 찾을 수 있어야 한다. 이를 통해서 유의미한 의미를 찾아낼 수 있는데 칼럼명을 통해 영감을 받아 다가설 수 있게 된다.

```
col_qry <- "SELECT table_name,
                   STRING_AGG(column_name, ', ') AS columns
            FROM information_schema.columns
            WHERE table_schema = 'public'
            GROUP BY table_name;"

dbGetQuery(con, col_qry) %>%
  filter(table_name %in% c( "actor", "rental", "store"))

DBI::dbDisconnect(con)
```

```
  table_name
1      actor                                          actor_id, last_update, first_name
2     rental rental_id, rental_date, inventory_id, customer_id, return_date, staff_id,
3      store                                    store_id, manager_staff_id, address_id,
```

#### 16.8.4.5 DVD ER 다이어그램

후속 쿼리 분석 작업을 위해서 도움이 되는 ER 다이어그램은 다음과 같다.

## 16.9 DVD DB 인사이트

DVT 대여 데이터베이스를 설치했다면 다음 단계로 다양한 SQL 쿼리문을 던져 뭔가 가치 있는 정보를 추출해야만 한다. 데이터 과학: "postgreSQL - DVD 대여 데이터베이스"에서 데이터베이스 설치와 접속에 대한 사항은 확인한다.

### 16.9.1 DB 접속 헬로월드 [7]

먼저 `DBI::dbConnect()`를 통해 접속하고 SQL 쿼리 헬로월드를 던져보자.

[7] Okoh Anita (2018-12-20), "How I analyzed DVD rental data with SQL", freeCodeCamp

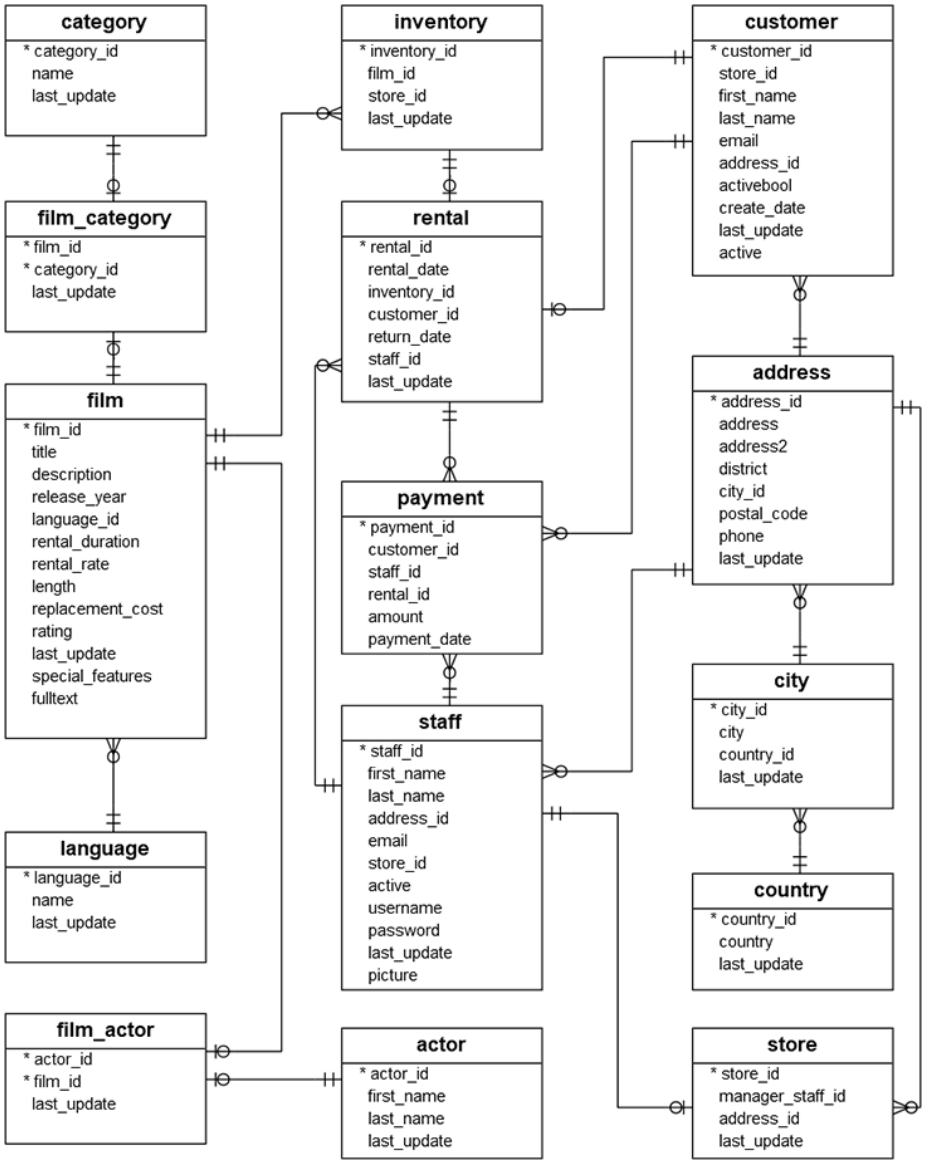



Figure 16.8: DVD ER 다이어그램

```
library(RPostgres)

con <- dbConnect(RPostgres::Postgres(), dbname="dvd",
                 host="localhost",
                 port="5432",
                 user="postgres",
                 password="1234")

actor <- dbGetQuery(con, "SELECT * FROM actor LIMIT 5")

actor
```

```
  actor_id first_name    last_name         last_update
1        1   Penelope      Guiness 2013-05-26 14:47:57
2        2       Nick     Wahlberg 2013-05-26 14:47:57
3        3         Ed        Chase 2013-05-26 14:47:57
4        4   Jennifer        Davis 2013-05-26 14:47:57
5        5     Johnny Lollobrigida 2013-05-26 14:47:57
```

### 16.9.2 이탈/잔존고객 구매금액

customer 테이블에는 active 칼럼을 통해 잔존고객과 이탈고객을 파악할 수 있다. 이를 통해서 잔존고객과 이탈고객이 몇명이고 구매금액을 파악할 수 있다. 먼저 datamodelr 팩키지를 통해 해당 테이블을 뽑아내서 이를 시각화해보자.

```
library(tidyverse)
library(datamodelr)

payment <- tbl(con, "payment") %>% collect()
customer <- tbl(con, "customer") %>% collect()

payment_customer_model <- dm_from_data_frames(payment, customer)

payment_customer_model <- dm_add_references(
  payment_customer_model,
  customer$customer_id ==  payment$customer_id
)

payment_customer_graph <- dm_create_graph(payment_customer_model, rankdir = "LR", col_a
dm_render_graph(payment_customer_graph)
```

con을 통해 DVD 대여 데이터베이스에 접속이 이루어진 상태다. 이탈고객과 잔존고객별로 구매금액에 대한 평균, 최소, 최대, 총합계를 구하려면 두 테이블을 INNER JOIN으로 customer_id를 키값으로 합치고 나서 기술통계를 산출한다.

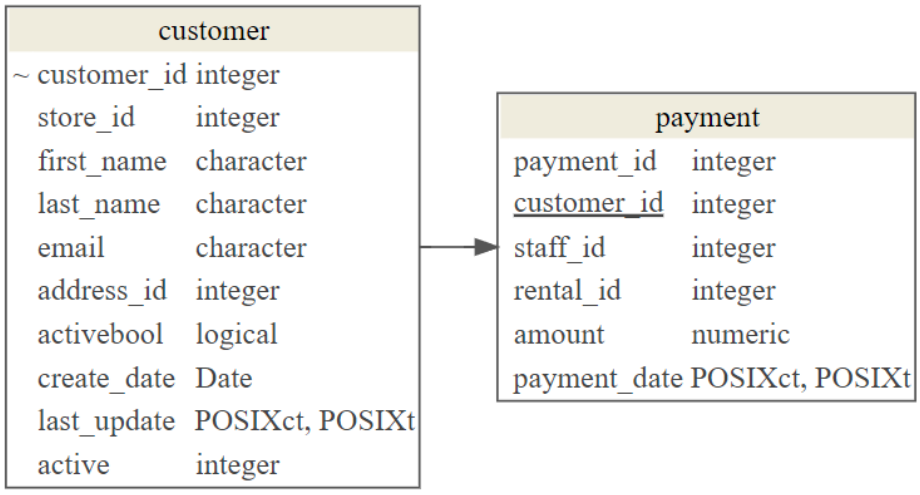



Figure 16.9: 테이블 구조 시각화 - 구매금액

```
sql_query <-
"SELECT active,
       COUNT(*) AS num_active,
       MIN(amount) AS min_amt,
       AVG(amount) AS avg_amt,
       MAX(amount) AS max_amt,
       SUM(amount) AS total_amt
FROM payment AS p
INNER JOIN customer AS c
  ON p.customer_id = c.customer_id
GROUP BY c.active;"

dbGetQuery(con, sql_query)
```

```
  active num_active min_amt  avg_amt max_amt total_amt
1      0        369    0.99 4.092981   11.99   1510.31
2      1      14227    0.00 4.203397   11.99  59801.73
```

### 16.9.3 장르별 평균 대여평점

앞서와 마찬가지로 장르별 평균 대여평점을 계산할 수 있는 테이블을 쭉 뽑아본다. 이를 통해서 3개 테이블, 즉 `category`, `film_category`, `film`을 뽑아놓고 각 해당 키값을 사용하여 결합시킨다.

```
category <- tbl(con, "category") %>% collect()
film_category <- tbl(con, "film_category") %>% collect()
film <- tbl(con, "film") %>% collect()
```

```
rental_rating_model <- dm_from_data_frames(category, film_category, film)

rental_rating_model <- dm_add_references(
  rental_rating_model,
  category$category_id == film_category$category_id,
  film_category$film_id == film$film_id
)

rental_rating_graph <- dm_create_graph(rental_rating_model, rankdir = "LR", col_attr =
dm_render_graph(rental_rating_graph)
```

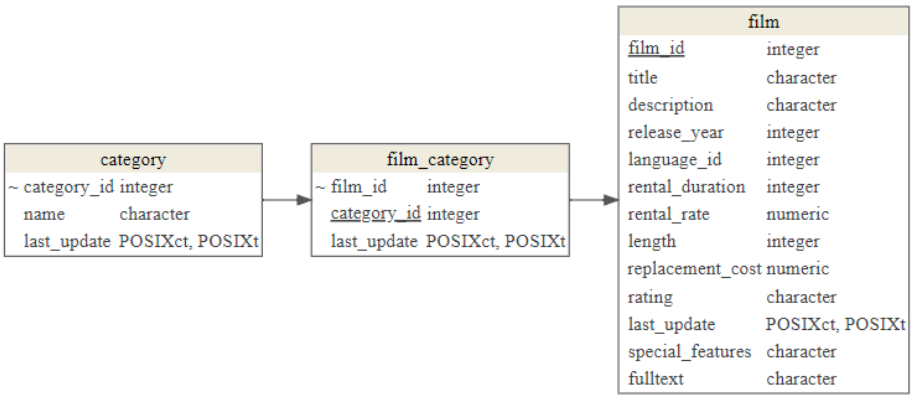



Figure 16.10: 테이블 구조 시각화 - 장르별 대여평점

먼저 `film_category`와 `category`를 결합시켜 영화(`film`)가 속한 장르(`category`)를 파악한다.

```
rate_qry <-
"SELECT *
FROM category AS c
INNER JOIN film_category AS fc
  ON c.category_id = fc.category_id
LIMIT 5;"

dbGetQuery(con, rate_qry)
```

```
  category_id        name         last_update film_id category_id..5      last_update.
1           6 Documentary 2006-02-15 09:46:27       1              6 2006-02-15 10:07:0
2          11      Horror 2006-02-15 09:46:27       2             11 2006-02-15 10:07:0
3           6 Documentary 2006-02-15 09:46:27       3              6 2006-02-15 10:07:0
4          11      Horror 2006-02-15 09:46:27       4             11 2006-02-15 10:07:0
5           8      Family 2006-02-15 09:46:27       5              8 2006-02-15 10:07:0
```

다음으로 film 테이블을 조인하여 rental_rate를 결합하고 장르(category) 별로 평균평점을 구하고 이를 ORDER BY ... DESC를 사용해서 내림차순으로 정렬한다.

```
rate_qry <-
"SELECT c.name,
        AVG(rental_rate) AS avg_rental_rate
FROM category AS c
INNER JOIN film_category AS fc
  ON c.category_id = fc.category_id
INNER JOIN film AS f
  ON fc.film_id = f.film_id
GROUP BY c.category_id
ORDER BY avg_rental_rate DESC;"

dbGetQuery(con, rate_qry)
```

```
          name avg_rental_rate
1        Games        3.252295
2       Travel        3.235614
3       Sci-Fi        3.219508
4       Comedy        3.162414
5       Sports        3.125135
6          New        3.116984
7      Foreign        3.099589
8       Horror        3.025714
9        Drama        3.022258
10       Music        2.950784
11    Children        2.890000
12   Animation        2.808182
13      Family        2.758116
14    Classics        2.744386
15 Documentary        2.666471
16      Action        2.646250
```

### 16.9.4 Top 10 DVD 영화

가장 많이 대여된 Top 10 DVD 영화를 찾아내기 위해서 이에 해당되는 연관 테이블을 검색하여 찾아낸다. film, inventory, rental 테이블을 특정하고 서로 연결시킬 수 있는 키값을 찾아 연결시킨다.

```
film <- tbl(con, "film") %>% collect()
inventory <- tbl(con, "inventory") %>% collect()
rental <- tbl(con, "rental") %>% collect()

top_10_model <- dm_from_data_frames(film, inventory, rental)

top_10_model <- dm_add_references(
```

```
  top_10_model,
  film$film_id == inventory$film_id,
  inventory$inventory_id == rental$inventory_id
)

top_10_graph <- dm_create_graph(top_10_model, rankdir = "LR", col_attr = c("column", "t
dm_render_graph(top_10_graph)
```

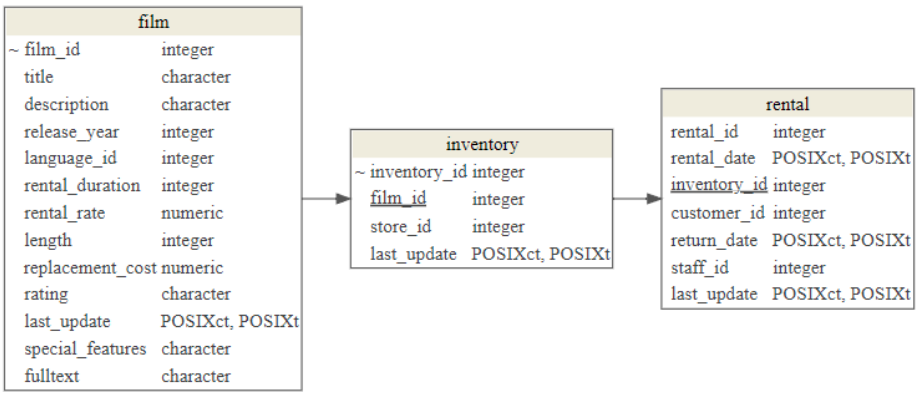



Figure 16.11: 테이블 구조 시각화 - Top 10 DVD 영화

`film` → `inventory` → `rental` 테이블을 순차적으로 `film_id`, `inventory_id`를 키값으로 삼아 결합시킨다. 그리고 나서 가장 많이 대여된 영화를 찾기 위해서 `COUNT()` 함수로 개수하고 나서 이를 내림차순 정리한다.

```
top_query <-
"SELECT f.title AS movie_title,
        COUNT(f.title) AS num_rentals
FROM film AS f
INNER JOIN inventory AS i
  ON f.film_id = i.film_id
INNER JOIN rental AS r
  ON i.inventory_id = r.inventory_id
GROUP BY f.title
ORDER BY num_rentals DESC;"

dbGetQuery(con, top_query) %>%
  slice_max(n=10, order_by = num_rentals)
```

```
           movie_title num_rentals
1   Bucket Brotherhood          34
2     Rocketeer Mother          33
3       Juggler Hardly          32
4  Ridgemont Submarine          32
```

```
5         Scalawag Duck          32
6        Grit Clockwork          32
7        Forward Temple          32
8        Timberland Sky          31
9             Zorro Ark          31
10          Robbers Joon          31
11          Hobbit Alien          31
12          Network Peak          31
13        Apache Divine           31
14      Rush Goodfellas           31
15            Wife Turn           31
16   Goodfellas Salute            31
```

## 16.10 요약

이번 장은 파이썬에서 데이터베이스 사용 기본적인 개요에 대해 폭넓게 다루었다. 데이터를 저장하기 위해서 파이썬 딕셔너리나 일반적인 파일보다 데이터베이스를 사용하여 코드를 작성하는 것이 훨씬 복잡하다. 그래서, 만약 작성하는 응용프로그램이 실질적으로 데이터베이스 역량을 필요하지 않는다면 굳이 데이터베이스를 사용할 이유는 없다. 데이터베이스가 특히 유용한 상황은 (1) 큰 데이터셋에서 작은 임의적인 갱신이 많이 필요한 응용프로그램을 작성할 때 (2) 데이터가 너무 커서 딕셔너리에 담을 수 없고 반복적으로 정보를 검색할 때, (3) 한번 실행에서 다음 실행 때까지 데이터를 보관하고, 멈추고, 재시작하는데 매우 긴 실행 프로세스를 갖는 경우다.

많은 응용프로그램 요구사항을 충족시키기 위해서 단일 테이블로 간단한 데이터베이스를 구축할 수 있다. 하지만, 대부분의 문제는 몇개의 테이블과 서로 다른 테이블간에 행이 연결된 관계를 요구한다. 테이블 사이 연결을 만들 때, 좀더 사려깊은 설계와 데이터베이스의 역량을 가장 잘 사용할 수 있는 데이터베이스 정규화 규칙을 따르는 것이 중요하다. 데이터베이스를 사용하는 주요 동기는 처리할 데이터의 양이 많기 때문에, 데이터를 효과적으로 모델링해서 프로그램이 가능하면 빠르게 실행되게 만드는 것이 중요하다.

## 16.11 디버깅

SQLite 데이터베이스에 연결하는 파이썬 프로그램을 개발할 때 하나의 일반적인 패턴은 파이썬 프로그램을 실행하고 SQLite 데이터베이스 브라우저를 통해서 결과를 확인하는 것이다. 브라우저를 통해서 빠르게 프로그램이 정상적으로 작동하는지를 확인할 수 있다.

SQLite에서 두 프로그램이 동시에 동일한 데이터를 변경하지 못하기 때문에 주의가 필요하다. 예를 들어, 브라우저에서 데이터베이스를 열고 데이터베이스에 변경을 하고 “저장(save)”버튼을 누르지 않는다면, 브라우져는 데이터베이스 파일에 “락(lock)”을 걸구, 다른 프로그램이 파일에 접근하는 것을 막는다. 특히, 파일이 잠겨져 있으면 작성하고 있는 파이썬 프로그램이 파일에 접근할 수 없다.

해결책은 데이터베이스가 잠겨져 있어서 파이썬 코드가 작동하지 않는 문제를 피하도록 파이썬에서 데이터베이스에 접근하려 시도하기 전에 데이터베이스 브라우져를 닫거나 혹은 **File** 메뉴를 사용해서 브라우져 데이터베이스를 닫는 것이다.

## 16.12 용어정의

- **속성(attribute)**: 튜플 내부에 값의 하나. 좀더 일반적으로 “열”, “칼럼”, “필드”로 불린다.
- **제약(constraint)**: 데이터베이스가 테이블의 필드나 행에 규칙을 강제하는 것. 일반적인 제약은 특정 필드에 중복된 값이 없도록 하는 것(즉, 모든 값이 유일해야 한다.)
- **커서(cursor)**: 커서를 사용해서 데이터베이스에서 SQL 명령어를 수행하고 데이터베이스에서 데이터를 가져온다. 커서는 네트워크 연결을 위한 소켓이나 파일의 파일 핸들러와 유사하다.
- **데이터베이스 브라우져(database browser)**: 프로그램을 작성하지 않고 직접적으로 데이터베이스에 연결하거나 데이터베이스를 조작할 수 있는 소프트웨어.
- **외부 키(foreign key)**: 다른 테이블에 있는 행의 주키를 가리키는 숫자 키. 외부 키는 다른 테이블에 저장된 행사이에 관계를 설정한다.
- **인텍스(index)**: 테이블에 행이 추가될 때 정보 검색하는 것을 빠르게 하기 위해서 데이터베이스 소프트웨어가 유지관리하는 추가 데이터.
- **논리 키(logical key)**: “외부 세계”가 특정 행의 정보를 찾기 위해서 사용하는 키. 사용자 계정 테이블의 예로, 사람의 전자우편 주소는 사용자 데이터에 대한 논리 키의 좋은 후보자가 될 수 있다.
- **정규화(normalization)**: 어떠한 데이터도 중복이 없도록 데이터 모델을 설계하는 것. 데이터베이스 한 장소에 데이터 각 항목 정보를 저장하고 외부키를 이용하여 다른 곳에서 참조한다.
- **주키(primary key)**: 다른 테이블에서 테이블의 한 행을 참조하기 위해서 각 행에 대입되는 숫자 키. 종종 데이터베이스는 행이 삽입될 때 주키를 자동 삽입하도록 설정되었다.
- **관계(relation)**: 튜플과 속성을 담고 있는 데이터베이스 내부 영역. 좀더 일반적으로 “테이블(table)”이라고 한다.
- **튜플(tuple)**:데이터베이스 테이블에 단일 항목으로 속성 집합이다. 좀더 일반적으로 “행(row)”이라고 한다.

# Chapter 17

# 작업 자동화

파일, 네트워크, 서비스, 그리고 데이터베이스에서 데이터를 읽어왔다. R은 또한 여러분의 로컬 컴퓨터 디렉토리와 폴더를 훑어서 파일도 읽어온다.

이번 장에서, 여러분의 로컬 컴퓨터를 스캔하고 각 파일에 대해서 연산을 수행하는 프로그램을 작성한다. 파일은 디렉토리(또한 "폴더"라고도 부른다.)에 정렬되어 보관된다. 간단한 R 스크립트로 전체 로컬 컴퓨터나 디렉토리 여기저기 뒤져야 찾아지는 수백 수천개 파일에 대한 단순한 작업을 짧게 수행한다.

트리상의 디렉토리나 파일을 여기저기 돌아다니기 위해서 `os.walk`과 `for` 루프를 사용한다. `open`이 파일 콘텐츠를 읽는 루프를 작성하는 것과 비슷하게, `socket`은 네트워크 연결된 콘텐츠를 읽는 루프를 작성하고, `urllib`는 웹문서를 열어 콘텐츠를 루프를 통해서 읽어오게 한다.

## 17.1 파일 이름과 경로

모든 실행 프로그램은 "현재 디렉토리(current directory)"를 가지고 있는데 작업 대부분을 수행하는 디폴트 디렉토리가 된다. 예를 들어, 읽기 위해서 파일을 연다면, R은 현재 디렉토리에서 파일을 찾는다.

`libuv` C 라이브러리에 기반한 `fs` 패키지는 파일과 디렉토리를 작업하는 함수를 제공한다. `path_wd()` 함수는 현재 디렉토리 이름을 반환한다.

```
library(fs)

cwd <- fs::path_wd()
# C:/swc/book_programming
```

`cwd` 는 current working directory의 약자로 현재 작업 디렉토리다. 예제의 결과는 `C:/swc/book_programming`인데 현재 작성중인 전자책 `book_programming` 의 홈 디렉토리가 된다.

파일을 식별하는 `cwd` 같은 문자열을 경로(path)라고 부른다. **상대경로(relative path)**는 현재 디렉토리에서 시작하고, **절대경로(absolute path)**는 파일 시스템의 가장 최상단의 디렉토리에서 시작한다.

지금까지 살펴본 경로는 간단한 파일 이름이여서, 현재 디렉토리에서 상대적이다. 파일의 절대 경로를 알아내기 위해서 `fs::path_abs()` 함수를 사용한다.

```
fs::path_abs("index.Rmd")
```

```
C:/swc/book_programming/index.Rmd
```

`dir_exists`는 디렉토리, `file_exists`는 파일이 존재하는지 검사한다.

```
fs::dir_exists("data")
```

```
data
TRUE
```

```
fs::file_exists("index.Rmd")
```

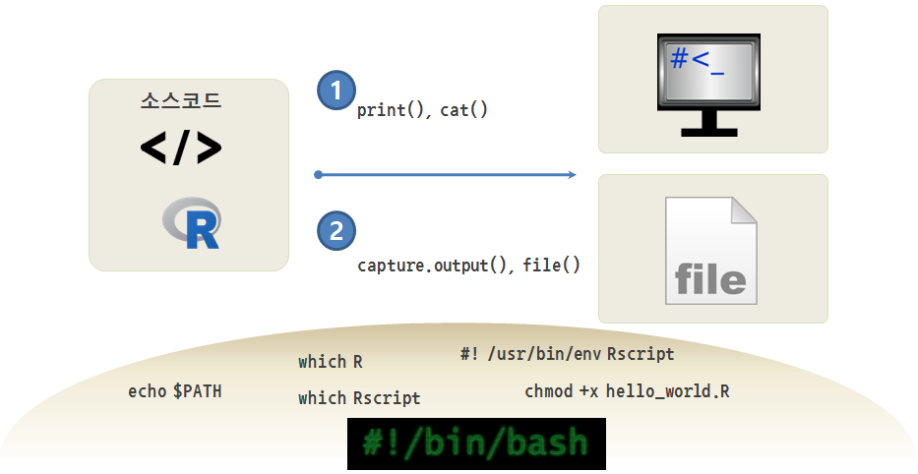
```
index.Rmd
TRUE
```

`dir_ls()` 함수는 은 주어진 디렉토리에 파일 리스트(그리고 다른 디렉토리)를 반환한다.

```
fs::dir_ls()
```

```
00-why.Rmd          01-intro.Rmd        02-var.Rmd          03-cont.Rmd
05-loop.Rmd         06-string.Rmd       07-file.Rmd         08-list.Rmd
10-tuples.Rmd       11-regex.Rmd        12-database.Rmd     13-web.Rmd
assets              book.bib            book_programming.rds book_programming.l
DESCRIPTION         Dockerfile          docs                index.Rmd
now.json            packages.bib        preamble.tex        README.md
script              style.css           toc.css             _bookdown.yml
_build.sh           _common.R           _deploy.sh          _output.yml
```

## 17.2 명령 줄 인자 [1] [2]




명령라인 인터페이스에서 R 스크립트를 실행하고 다양한 R 스크립트 실행방법을 살펴보자. 먼저, 유닉스/리눅스/윈도우 운영체제가 준비되었다면 R스크립트 실행환경을 준비한다.

### 17.2.1 R 설치

R을 스크립트 형태(`.R`) 파일로 실행할 경우 가장 먼저 `r-base-core`를 설치한다. 그래픽 사용자 인터페이스가 없는 형태의 R이 설치되며 R 스크립트 실행에 필요한 연관된 프로그램도 더불어 설치한다.

```
$ sudo apt-get install -y r-base-core
```

### 17.2.2 R 스크립트 실행환경

R과 스크립트를 실행할 `Rscript` 실행프로그램이 위치한 디렉토리를 확인한다. 여기에 사용되는 명령어는 `which`다. `which R` 명령어를 통해 R 실행파일이 `/usr/bin/` 디렉토리에 위치한 것을 확인할 수 있다.

```
$ which R
/usr/bin/R
```

`which Rscript` 명령어를 통해 Rcript 실행파일도 `/usr/bin/` 디렉토리에 위치한 것을 확인할 수 있다.

```
$ which Rscript
/usr/bin/Rscript
```

[1] R scripts
[2] .R 스크립트를 인자와 함께 실행

echo $PATH 명령어를 통해 /usr/bin, 경우에 따라서는 /usr/local/bin 디렉토리에 R과 Rscript 실행파일이 존재하는 것을 확인한다.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/game
```

### 17.2.3 R 스크립트 파일 실행

.R 스크립트 파일을 명령라인 인터페이스로 실행하는 방법은 다양한다. 먼저 hello_world.R 스크립트 파일을 생성한다.

```
#! /usr/bin/env Rscript

print("Hello World!!!")
```

쉬뱅(shebang, #!) 다음에 스크립트를 실행할 프로그램을 지정한다. Rscript로 지정하여 R스크립트를 실행하는데 사용한다.

```
#! /usr/bin/env Rscript
```

```
$ chmod +x hello_world.R
$ ./hello_world.R
```

chmod +x 명령어를 통해서 일반 텍스트 파일을 실행가능한 파일 형식으로 지정한다. hello_world.R 파일이 실행가능한 형태가 되었기 때문에 ./hello_world.R 명령어로 R스크립트를 실행시킨다.

```
Hello World!!!
```

### 17.2.4 다른 R 스크립트 파일 실행법

R스크립트를 실행하는 방법은 다양하다.

```
$ R --slave -f hello_world.R
$ Rscript hello_world.R
```

Rscript 명령어로 실행을 시켜도 동일한 산출 결과가 출력된다.

```
$ R CMD BATCH hello_world.R hello_world_output.txt
```

R CMD BATCH 명령어로 실행시키면 실행결과가 hello_world_output.txt 파일에 저장된다. hello_world_output.txt 파일명을 지정하지 않으면 hello_world.Rout 파일에 저장된다.

```
$ R --no-save << RSCRIPT
     print("Hello World")
  RSCRIPT
```

R --no-save << 사용법도 가능하다.

## 17.3 실습 사례 [3]

### 17.3.1 R스크립트 작성

`Rscript r_session_info.R` 명령어를 실행해서 실제로 RStudio나 R 콘솔을 열지 않고도 R 세션정보를 명령라인 인터페이스에서 처리하는 R 스크립트를 작성한다.

텍스트 편집기를 열고, `sessionInfo()`를 적고 파일명을 `r_session_info.R`로 저장한다.

```
sessionInfo()
```

배쉬쉘에서 R스크립트를 실행해서 R 세션정보를 받아확인한다.

```
$ Rscript r_session_info.R

R version 3.0.2 (2013-09-25)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base
```

### 17.3.2 R스크립트 출력결과 파일저장

> 파이프 연산자를 사용해서 R 스크립트 출력결과를 텍스트 파일로 저장한다.

```
$ Rscript r_session_info.R > r_session_info_pipe_output.txt
```

또다른 방법은 R 스크립트 내부에서 출력결과를 파일에 저장하고 프로그램을 종료하는 방법도 있다.

`caputre.output` 함수를 `cat`과 함께 사용하는데, 한글도 적용이 가능하도록, `encoding="UTF-8"`도 추가한다.

```
output <- capture.output(sessionInfo())
cat("R   ", output, file="./r_session_info_rscript.txt", sep="\n", encoding="UTF-8")
```

`Rscript r_session_info.R` 명령어를 실행시키면 다음과 같이 실행결과가 텍스트 파일 `r_session_info_rscript.txt`로 떨어진다.

[3] R프로그래밍 - 명령-라인 프로그램

```
R
R version 3.0.2 (2013-09-25)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  base
UTF-8
```

## 17.4 자동화 사례: 국가별 통계

.R 스크립트를 유닉스/리눅스상에서 유연하게 동작시킨다. R 스크립트를 수정하지 않고, 인자를 바꿔 작업을 수행하는 방법을 살펴보자. 국가별 통계를 데이터 파일을 달리하여 R스크립트 통계량을 산출하는 방법을 사례로 구현해보자.

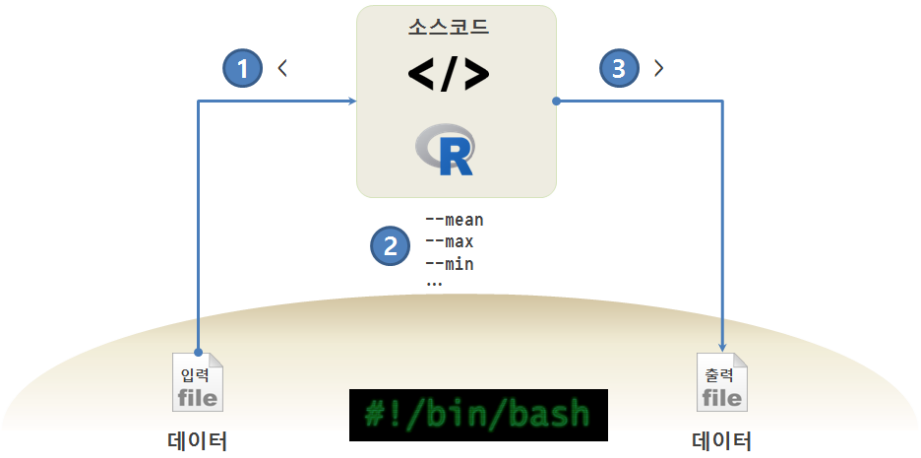



Figure 17.1: R 스크립트 실행

### 17.4.1 인자를 받는 R 스크립트

유닉스/GNU리눅스 환경에서 통계모형, 기계학습, 딥러닝 작업을 하게 되면 텍스트 형태 데이터, R스크립트, 출력 산출물을 갖게 되고, 입력 데이터를 출력산출물로 변환시키는데 R이 역할을 하게 된다.

R 스크립트를 유연하게 만들게 되면 데이터만 바꿔도 산출물을 생성해 내고, 경우에 따라 인자값을 달리하면 원하는 다른 결과를 얻게 된다.

### 17.4.2 데이터 변경 R 스크립트

SOME TIME SERIES DATA SETS에서 Per capita annual GDP for several countries during 1950-1983 (first row is 1950, last is 1983) 데이터를 사용한다. 데이터를 `austria.csv`, `canada.csv`, `france.csv`와 같이 구분하여 저장한다.

#### 17.4.2.1 R 스크립트 시제품 제작

먼저 `austria.csv` 파일을 불러와서 평균을 계산하는 R스크립트를 작성한다.

```
#
gdp.df <- read.csv("austria.csv", sep=",", head=FALSE)
#
gdp.mean <- mean(gdp.df$V1)
#
cat(" : ", gdp.mean, "\n", encoding="utf-8")
```

`Rscript r-args-ex01.R` 실행결과 예상대로 평균 GDP가 계산되었다.

```
$ Rscript r-args-ex01.R
 :  0.06553276
```

#### 17.4.2.2 입력파일 변경 R스크립트 실행

국가가 더 많을 수도 있지만, `austria.csv`, `france.csv`, `canada.csv` 3개 국가가 csv 파일로 데이터가 저장되어 있으니, 입력파일을 달리하여 평균을 계산하도록 R스크립트를 작성한다.

`commandArgs` 명령어를 통해 명령라인에서 인자를 받아 온다. 인자가 순서대로 들어오기 때문에 첫번째 인자로 들어오는 국가에 대한 GDP 평균을 구하고, 이를 화면에 출력하는 R스크립트다. `strsplit` 함수를 사용해서 파일명 앞쪽 - `.csv` 확장자 제거 - 만을 뽑아내어 국가명을 명기했다.

```
#!/usr/bin/env Rscript

args = commandArgs(trailingOnly=TRUE)
country <- args[1]

#
gdp.df <- read.csv(country, sep=",", head=FALSE)
#
gdp.mean <- mean(gdp.df$V1)
#
cat(strsplit(country, '\\.')[[1]][1], " : ", gdp.mean, "\n", encoding="utf-8")
```

상기 R스크립트를 쉘에서 실행한 결과는 다음과 같다.

```
$ Rscript r-args-ex02.R  austria.csv
austria  :  0.06553276
$ Rscript r-args-ex02.R france.csv
france  :  20.95751
$ Rscript r-args-ex02.R canada.csv
canada  :  5.817088
```

#### 17.4.2.3 국가별 기본통계 계산

국가 데이터를 바꾸는 것에 더해서 최소, 평균, 최대 GDP를 계산하는 로직을 추가한다. `commandArgs` 함수로 인자를 받는데, 최종 인자는 -1로 지정되기 때문에 그런 특성을 이용하여 R스크립트를 작성한다. 따라서, 첫번째 인자에 최소, 평균, 최소를 구할 것인지 정보를 받고, 마지막 인자로 파일명을 받는다.

```
#!/usr/bin/env Rscript

args = commandArgs(trailingOnly=TRUE)
action = args[1]
country <- args[-1]

#
gdp.df <- read.csv(country, sep=",", head=FALSE)
#

if(action == "--min") {
    gdp.min <- min(gdp.df$V1)
    cat(strsplit(country, '\\.')[[1]][1], " : ", gdp.min, "\n", encoding="utf-8")
}else if(action == "--mean") {
    gdp.mean <- mean(gdp.df$V1)
    cat(strsplit(country, '\\.')[[1]][1], " : ", gdp.mean, "\n", encoding="utf-8")
}else if (action=="--max"){
    gdp.max <- max(gdp.df$V1)
    cat(strsplit(country, '\\.')[[1]][1], " : ", gdp.max, "\n", encoding="utf-8")
}
```

`Rscript r-args-ex03.R --min canada.csv` 명령라인을 살펴보면, `r-args-ex03.R` R 스크립트를 실행하고, `--min` 인자로 최소값을 계산하는데, `canada.csv` 데이터 파일을 이용한다.

```
$ Rscript r-args-ex03.R --min canada.csv
canada  :  3.651109
$ Rscript r-args-ex03.R --max canada.csv
canada  :  8.382785
$ Rscript r-args-ex03.R --mean canada.csv
canada  :  5.817088
```

**R 스크립트에 인자를 넘기는 패키지**

- commandArgs
- optparse - Command Line Option Parser
- argparse - Command line optional and positional argument parser
- getopt - C-like getopt behavior

## 17.5 파이프(Pipes)

대부분의 운영 시스템은 **쉘(shell)**로 알려진 명령어 기반 인터페이스를 지원한다. 일반적으로 쉘은 파일 시스템을 탐색하거나 응용 프로그램을 실행하는 명령어를 지원한다. 예를 들어, 유닉스에서 `cd` 명령어로 디렉토리를 변경하고 `ls` 명령어로 디렉토리의 콘텐츠를 보여주고, `firefox`를 타이핑해서 웹브라우져를 실행한다.

쉘에서 실행시킬 수 있는 어떤 프로그램이나 **파이프(pipe)**를 사용하여 파이썬에서도 실행시킬 수 있다. 파이프는 작동중인 프로세스를 표현하는 객체다.

예를 들어, 유닉스 명령어[4] `ls -l`는 정상적으로 현재 디렉토리의 콘텐츠(긴 형식으로)를 보여준다. `system2()` 내장함수, 혹은 `processx`, `sys` 패키지를 가지고 `ls` 를 실행시킬 수 있다.

```
system2("ls", "-l", stdout = TRUE, stderr = TRUE)
```

```
 [1] "total 960"
 [2] "-rw-r--r--   1 tidyverse  staff  16280  5 18 10:26 00-why.Rmd"
 [3] "-rw-r--r--   1 tidyverse  staff  44481  5 18 10:26 01-intro.Rmd"
 [4] "-rw-r--r--   1 tidyverse  staff  25092  5 18 10:26 02-var.Rmd"
 [5] "-rw-r--r--   1 tidyverse  staff  20041  5 18 10:26 03-cont.Rmd"
 [6] "-rw-r--r--   1 tidyverse  staff  24862  5 18 10:26 04-func.Rmd"
 [7] "-rw-r--r--   1 tidyverse  staff  19254  5 18 10:26 05-loop.Rmd"
 [8] "-rw-r--r--   1 tidyverse  staff  19007  5 18 10:26 06-string.Rmd"
 [9] "-rw-r--r--   1 tidyverse  staff  26301  5 18 10:26 07-file.Rmd"
[10] "-rw-r--r--   1 tidyverse  staff  27648  5 18 10:26 08-list.Rmd"
[11] "-rw-r--r--   1 tidyverse  staff  22463  5 18 10:26 09-dictionaries.Rmd"
[12] "-rw-r--r--   1 tidyverse  staff  12085  5 18 10:26 10-tuples.Rmd"
[13] "-rw-r--r--   1 tidyverse  staff  30744  5 18 12:42 11-regex.Rmd"
[14] "-rw-r--r--   1 tidyverse  staff  36390  5 19 11:15 12-database.Rmd"
[15] "-rw-r--r--   1 tidyverse  staff  18931  5 19 11:15 13-web.Rmd"
[16] "-rw-r--r--   1 tidyverse  staff  32936  5 19 12:19 14-tasks.Rmd"
[17] "-rw-r--r--   1 tidyverse  staff    115  5 18 10:26 DESCRIPTION"
[18] "-rw-r--r--   1 tidyverse  staff    150  5 18 10:26 Dockerfile"
[19] "-rw-r--r--   1 tidyverse  staff   6556  5 18 10:26 LICENSE"
[20] "-rw-r--r--   1 tidyverse  staff    741  5 18 10:26 README.md"
[21] "-rw-r--r--   1 tidyverse  staff    671  5 19 11:15 _bookdown.yml"
[22] "-rw-r--r--   1 tidyverse  staff    235  5 18 10:26 _build.sh"
```

[4] 파이프를 사용하여 `ls` 같은 운영 시스템 명령어로 대화할 때, 무슨 운영 시스템을 사용하는지 알고 운영 시스템에서 지원되는 명령어로 파이프를 열수 있다는 것이 중요하다.

```
[23] "-rw-r--r--   1 tidyverse  staff    842  5 18 10:26 _common.R"
[24] "-rwxr-xr-x   1 tidyverse  staff    398  5 18 10:26 _deploy.sh"
[25] "-rw-r--r--   1 tidyverse  staff    360  5 18 10:26 _output.yml"
[26] "-rw-r--r--   1 tidyverse  staff   1427  5 18 10:26 _publish.R"
[27] "drwxr-xr-x   4 tidyverse  staff    128  5 18 14:45 assets"
[28] "-rw-r--r--   1 tidyverse  staff   3203  5 18 10:26 book.bib"
[29] "-rw-r--r--   1 tidyverse  staff    277  5 18 10:26 book_programming.Rproj"
[30] "-rw-r--r--   1 tidyverse  staff    208  5 19 12:10 book_programming.rds"
[31] "drwxr-xr-x  10 tidyverse  staff    320  5 18 14:04 data"
[32] "drwxr-xr-x  48 tidyverse  staff   1536  5 19 12:10 docs"
[33] "-rw-r--r--   1 tidyverse  staff   6272  5 18 10:26 index.Rmd"
[34] "-rw-r--r--   1 tidyverse  staff     41  5 18 10:26 now.json"
[35] "-rw-r--r--   1 tidyverse  staff   2655  5 18 10:26 packages.bib"
[36] "-rw-r--r--   1 tidyverse  staff    161  5 18 10:26 preamble.tex"
[37] "-rw-r--r--   1 tidyverse  staff     55  5 18 10:26 references.Rmd"
[38] "drwxr-xr-x   4 tidyverse  staff    128  5 18 10:26 script"
[39] "-rw-r--r--   1 tidyverse  staff    172  5 18 10:26 style.css"
[40] "-rw-r--r--   1 tidyverse  staff   2443  5 18 10:26 toc.css"
```

인자는 쉘 명령어를 포함하는 문자열이다.

### 17.5.1 유닉스 철학과 파이프

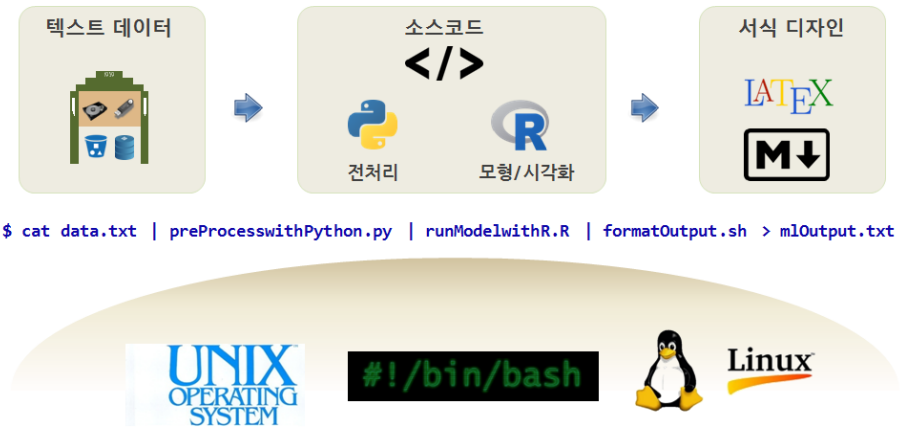



Figure 17.2: R 파이프라인

통계 예측모형, 기계학습, 딥러닝 시스템을 개발할 경우 유닉스/리눅스 운영체제로 환경을 통일하고 텍스트 파일을 모든 프로그램과 시스템이 의사소통하는 기본 인터페이스로 잡고, 이를 파이프로 연결한다.

1. 텍스트 데이터로 분석에 사용될 혹은 훈련데이터로 준비한다.
2. 파이썬 혹은 쉘스크립트, R스크립트를 활용하여 전처리한다.

3. R `tidymodels` 혹은 파이썬 `Scikit-learn` 예측모형을 적합, 기계학습 훈련, 시각화를 수행한다.
4. 마크다운(웹), LaTeX(조판) 출력형식에 맞춰 서식을 결정한다.
5. 최종 결과를 텍스트, 이미지 파일, pdf, html로 출력한다.

```
$ cat data.txt | preProcesswithPython.py | runModelwithR.R | formatOutput.sh > mlOutput.txt
```

**유닉스 철학**

[원문] Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface. - Doug McIlroy

- 한가지 작업만 매우 잘하는 프로그램을 작성한다(Write programs that do one thing and do it well)
- 프로그램이 함께 동작하도록 작성한다(Write programs to work together)
- 텍스트를 다루는 프로그램을 작성한다. 이유는 어디서나 사용되는 인터페이스가 되기 때문이다(Write programs to handle text streams, because that is a universal interface)

## 17.6 용어정의

- **절대경로(absolute path)**: 파일이나 디렉토리가 어디에 저장되어 있는지를 저장하는 문자열로 "최상단의 디렉토리"에서 시작해서, 현재 작업 디렉토리에 관계없이 파일이나 디렉토리를 접근하는데 사용할 수 있다.
- **체크썸(checksum)**: **해싱(hashing)**을 참조하세요. "체크썸(checksum)"단어는 네트워크로 데이터가 보내지거나 백업 매체에 쓰여지고 다시 읽어올 때, 데이터가 왜곡되었는지를 검증하는 필요에서 생겨났다. 데이터가 쓰여지거나 보내질 때, 송신 시스템은 체크썸을 계산하고 또한 체크썸도 보낸다. 데이터가 읽혀지거나 받았을 때, 수신 시스템을 수신된 데이터의 체크썸을 다시 계산하고 받은 체크썸과 비교한다. 만약 체크썸이 매칭되지 않으면, 전송 시에 데이터가 왜곡된 것으로 판단해야 한다.
- **명령 줄 인자(command line argument)**: 파이썬 파일 이름 뒤에 명령 줄에 매개 변수.
- **현재 작업 디렉토리(current working directory)**: 여러분이 "작업하고 있는" 현재 디렉토리. 명령-줄 인터페이스에서 대부분의 시스템에 `cd` 명령어를 사용하여 작업 디렉토리를 변경할 수 있다. 경로 정보 없이 파일만을 사용하여 파이썬에서 파일을 열게 될 때, 파일은 프로그램을 실행하고 있는현재 작업 디렉토리에 있어야 한다.
- **해싱(hashing)**: 가능한 큰 데이터를 읽고 그 데이터에 대해서 유일한 체크썸을 생성하는 것. 최고의 해쉬 함수는 거의 "충돌(collision)"을 만들지 않는다. 여기서 충돌은 서로 다른 두 데이터 스트림에 해쉬 함수를 줄 때 동일한 해쉬값을 돌려받는 것이다. MD5, SHA1, SHA256 는 가장 많이 사용되는 해쉬 함수의 사례다.
- **파이프(pipe)**: 파이프는 실행하는 프로그램에 연결이다. 파이프를 사용해서, 데이터를 다른 프로그램에 보내거나 그 프로그램에서 데이터를 받는

프로그램을 작성할 수 있다. 파이프는 **소켓(socket)**과 매우 유사하다. 차이점은 파이프는 동일한 컴퓨터에서 실행되는 프로그램을 연결하는데만 사용된다는 것이다. (즉, 네트워크를 통해서는 사용할 수 없다.)

- **상대경로(relative path)**: 파일 혹은 디렉토리가 어디에 저장되었는지를 현재 작업 디렉토리에 상대적으로 표현하는 문자열.
- **쉘(shell)**: 운영 시스템에 명령줄 인터페이스. 다른 시스템에서는 또한 "터미널 프로그램(terminal program)"이라고 부른다. 이런 인터페이스에서 라인에 명령어와 매개 변수를 타입하고 명령을 실행하기 위해서 "엔터(enter)"를 누른다.
- **워크(walk)**: 모든 디렉토리를 방문할 때까지 디렉토리, 하위 디렉토리, 하위의 하위 디렉토리 전체 트리를 방문하는 개념을 나타내기 위해서 사용된 용어. 여기서 이것을 "디렉토리 트리를 워크"한다고 부른다.

## 17.7 연습문제

MP3 파일이 대규모로 수집되어 있은 곳에는 같은 노래의 복사본 하나 이상이 다른 디렉토리 혹은 다른 파일 이름으로 저장되어 있을 수 있다. 이번 연습문제의 목표는 이런 중복 파일을 찾는 것이다.

1. `.mp3`같은 확장자를 가진 파일을 모든 디렉토리와 하위 디렉토리를 검색해서 동일한 크기를 가진 파일짝을 목록으로 보여주는 프로그램을 작성하세요.

2. **체크썸(checksum)** 알고리즘이나 해싱을 사용하여 중복 콘텐츠를 가진 파일을 찾는 이전의 프로그램을 개작하세요. 예를 들어, MD5 (Message-Digest algorithm 5)는 임의적으로 긴 "메시지"를 가지고 128비트 "체크썸"을 반환한다. 다른 콘텐츠를 가진 두 파일이 같은 체크썸을 반환할 확률은 매우 적다.

   wikipedia.org/wiki/Md5에서 MD5에 대해서 더 배울 수 있다. 다음 코드 조각은 파일을 열고, 읽고, 체크썸을 계산한다.

   체크섬을 계산하고 키로 이미 딕셔너리에 있게 되면, 중복 콘텐츠인 두 파일 있어서 딕셔너리에 파일과 방금전에 읽은 파일을 출력한다. 사진 파일 폴더에서 실행한 샘플 출력물이 다음에 있다.

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jpg
```

   .

# Chapter 18

# 함수형 프로그래밍

데이터 과학 맥락에서 함수형 프로그래밍을 이해하고, 수치해석 예제를 통해 코드를 작성하고, 함수형 프로그래밍과 짝꿍인 단위 테스트에 대해서 살펴본다.

**함수형 프로그래밍(functional programming)**은 코드를 작성하는 한 방식으로 특정 연산을 수행하는 함수를 먼저 작성하고 나서, 사용자가 함수를 호출해서 작업을 수행하는 방식이다. 순수 함수형 언어, 예를 들어 하스케(Haskell)은 루프가 없다. 루푸없이 어떻게 프로그램을 작성할 수 있을까? 루프는 재귀(recursion)로 대체된다. 이런 이유로 아래에서 뉴톤 방법을 통해 근을 구하는 방식을 R코드로 두가지 방법을 보여준다. R은 아직 꼬리 호출(tail-call recurssion) 기능을 제공하지 않기 때문에 루프를 사용하는 것이 더 낫다.

수학 함수는 멋진 특성이 있는데, 즉 해당 입력에 항상 동일한 결과를 갖는다. 이 특성을 **참조 투명성(referential transparaency)** 이라고 부른다. 참조 투명성은 **부수효과(side effect)** 없음을 표현하는 속성인데, 함수가 결과값 외에 다른 상태를 변경시킬 때 부수효과(side effect)가 있다고 한다. 부수 효과는 프로그램 버그를 발생시키는 온상으로, 부수 효과를 없애면 디버깅이 용이해진다. 따라서, 부수 효과를 제거하고 참조 투명성을 유지함으로써 데이터 분석 수행 결과를 예측 가능한 상태로 유지시켜 **재현가능한 과학**이 가능하게 된다.

**함수형 프로그래밍**

R 함수형 프로그래밍을 이해하기 위해서는 먼저 자료구조에 대한 이해가 선행되어야 한다. 그리고 나서, 함수를 작성하는 이유와 더불어 작성법에 대한 이해도 확고히 해야만 한다.

객체(object)가 함수를 갖는 데이터라면, 클러져는 데이터를 갖는 함수다.

> "An object is data with functions. A closure is a function with data." - John D. Cook

명령형 언어(Imperative Language) 방식으로 R코드를 쭉 작성하게 되면, 각 단계별로 상태가 변경되는 것에 대해 신경을 쓰고 관리를 해나가야 된다. 그렇지

않으면 예기치 않은 부수 효과가 발생하여 데이터 분석 및 모형을 잘못 해석하게 된다.

그렇다고, 부수효과가 없는 순수 함수가 반듯이 좋은 것은 아니다. 예를 들어 `rnorm()` 함수를 통해 평균 0, 분산 1인 난수를 생성시키는데, 항상 동일한 값만 뽑아내면 사용자에게 의미있는 함수는 아니다.

결국, 함수형 프로그래밍을 통해 **테스팅(testing)**과 **디버깅(debugging)**을 수월하게 하는 것이 추구하는 바이다. 이를 위해 다음 3가지 요인이 중요하다.

- 한번에 한가지 작업을 수행하는 함수
- 부수효과(side effect) 회피
- 참조 투명성(Referential transparaency)

## 18.1 왜 함수형 프로그래밍인가?

데이터 분석을 아주 추상화해서 간략하게 얘기한다면 데이터프레임을 함수에 넣어 새로운 데이터프레임으로 만들어 내는 것이다.

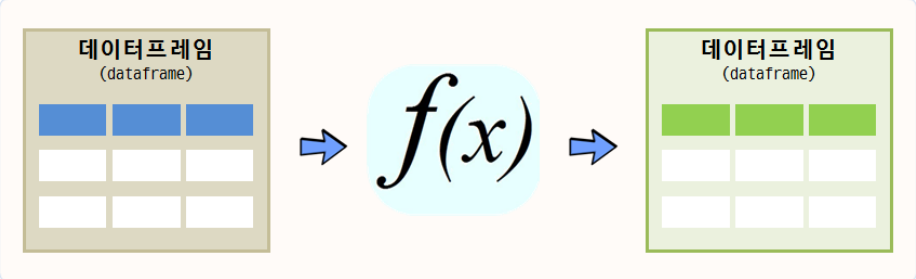



Figure 18.1: 데이터 분석 과정 추상화

데이터 분석, 데이터 전처리, 변수 선택, 모형 개발이 한번에 해결되는 것이 아니라서, 데이터프레임을 함수에 넣어 상태가 변경된 데이터프레임이 생성되고, 이를 다시 함수에 넣어 또다른 변경된 상태 데이터프레임을 얻게 되는 과정을 쭉 반복해 나간다.

따라서… 데이터 분석에는 함수형 프로그래밍 패러다임을 활용하고, 툴/패키지 개발에는 객체지향 프로그래밍(OOP) 패러다임 사용이 권장된다.

## 18.2 사례: 뉴튼 방법(Newton's Method)

뉴튼-랩슨 알고리즘으로도 알려진 뉴튼(Newton Method) 방법은 컴퓨터를 사용해서 수치해석 방법으로 실함수의 근을 찾아내는 방법이다.

특정 함수 $f$ 의 근을 찾을 경우, 함수 미분값 $f'$, 초기값 $x_0$가 주어지면 근사적 근에 가까운 값은 다음과 같이 정의된다.

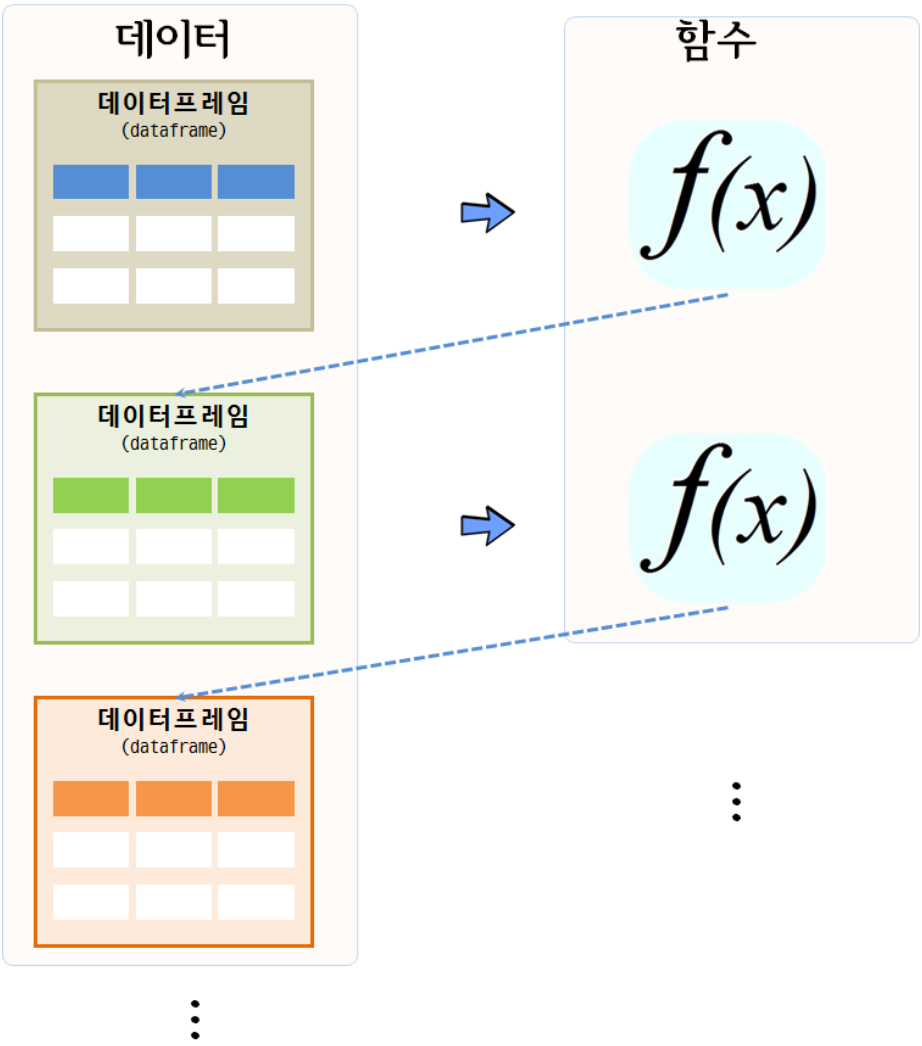



Figure 18.2: 데이터 분석 작업흐름


Figure 18.3: 데이터 분석은 함수형 프로그래밍, 툴/패키지 개발에는 OOP

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

이 과정을 반복하게 되면 오차가 매우 적게 근의 값에 도달하게 된다.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

기하적으로 보면, 파란 선은 함수 $f$ 이고, $f$를 미분한 $f'$ 빨간 선은 뉴턴방법을 활용하여 근을 구해가는 과정을 시각적으로 보여주고 있다. $x_{n-1}$ 보다 $x_n$이, $x_n$ 보다 $x_{n+1}$이 함수 $f$ 근에 더 가깝게 접근해 나가는 것이 확인된다.

### 18.2.1 뉴튼 방법 R 코드 [1]

뉴튼 방법을 R코들 구현하면 다음과 같이 612의 제곱근 값을 수치적으로 컴퓨터를 활용하여 구할 수 있다. `while`같은 루프를 활용하여 반복적으로 해를 구하는 것도 가능하지만 재귀를 활용하여 해를 구하는 방법이 코드를 작성하고 읽는 개발자 관점에서는 훨씬 더 편리하고 권장된다.

하지만, 속도는 `while` 루프를 사용하는 것이 R에서는 득이 많다. 이유는 오랜 세월에 걸쳐 최적화 과정을 거쳐 진화했기 때문이다.

#### while 루프를 사용한 방법

```
find_root <- function(guess, init, eps = 10^(-10)){
    while(abs(init**2 - guess) > eps){
```

[1] Bruno Rodrigues(2016), "Functional programming and unit testing for data munging with R", LeanPub, 2016-12-23

```
        init <- 1/2 *(init + guess/init)
        cat("   : ", init, "\n")
    }
    return(init)
}

find_root(612, 10)
```

```
##    :  35.6
##    :  26.4
##    :  24.8
##    :  24.7
##    :  24.7
##    :  24.7
```

```
## [1] 24.7
```

**재귀를 사용한 방법**

```
find_root_recur <- function(guess, init, eps = 10^(-10)){
    if(abs(init**2 - guess) < eps){
        return(init)
    } else{
        init <- 1/2 *(init + guess/init)
        cat("      : ", init, "\n")
        return(find_root_recur(guess, init, eps))
    }
}

find_root_recur(612, 10)
```

```
##       :  35.6
##       :  26.4
##       :  24.8
##       :  24.7
##       :  24.7
##       :  24.7
```

```
## [1] 24.7
```

### 18.2.2 Map(), Reduce() 함수와 *apply() 함수 가족 [2] [3]

함수를 인자로 받는 함수를 고차함수(High-order function)라고 부른다. 대표적으로 Map(), Reduce()가 있다. 숫자 하나가 아닌 벡터에 대한 제곱근을 구하기 위해서 Map 함수를 사용한다.

[2] purrr tutorial: Lessons and Examples
[3] purrr tutorial GitHub Webpage

```
#      ----------------------------------------

find_root_recur <- function(guess, init, eps = 10^(-10)){
    if(abs(init**2 - guess) < eps){
        return(init)
    } else{
        init <- 1/2 *(init + guess/init)
        return(find_root_recur(guess, init, eps))
    }
}

#

numbers <- c(16, 25, 36, 49, 64, 81)
Map(find_root_recur, numbers, init=1, eps = 10^-10)
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 8
##
## [[6]]
## [1] 9
```

숫자 하나를 받는 함수가 아니라, 벡터를 인자로 받아 제곱근을 계산하는 함수를 작성할 경우 함수 내부에서 함수를 인자로 받을 수 있도록 `Map` 함수를 활용한다.

```
# `Map`

find_vec_root_recur <- function(numbers, init, eps = 10^(-10)){
    return(Map(find_root_recur, numbers, init, eps))
}

numbers_z <- c(9, 16, 25, 49, 121)
find_vec_root_recur(numbers_z, init=1, eps=10^(-10))
```

```
## [[1]]
```

```
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 11
```

이러한 패턴이 많이 활용되어 *apply 함수가 있어, 이전에 많이 사용했을 것이다. 벡터를 인자로 먼저 넣고, 함수명을 두번째 인자로 넣고, 함수에 들어갈 매개변수를 순서대로 쭉 나열하여 lapply, sapply 함수에 넣는다.

```
# `lapply`

lapply(numbers_z, find_root_recur, init=1, eps=10^(-10))
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 11
```

```
sapply(numbers_z, find_root_recur, init=1, eps=10^(-10))
```

```
## [1]  3  4  5  7 11
```

Reduce 함수도 삶을 편안하게 할 수 있는, 루프를 회피하는 또다른 방법이다. 이름에서 알 수 있듯이 numbers_z 벡터 원소 각각에 대해 해당 연산작업 +, %%을 수행시킨다. %%는 나머지 연산자로 기본디폴트 설정으로 $\frac{10}{7}$로 몫 대신에 나머지 3을 우선 계산하고, 그 다음으로 $\frac{3}{5}$로 최종 나머지 3을 순차적으로 계산하여 결과를 도출한다.

```
# Reduce ------------------------------------------------
numbers_z
```

```
## [1]   9  16  25  49 121
```

```
Reduce(`+`, numbers_z)
```

```
## [1] 220
```

```
numbers_z <- c(10,7,5)
Reduce(`%%`, numbers_z)
```

```
## [1] 3
```

### 18.2.3 purrr 뉴톤 방법 코드

초창기 for나 while 루프를 사용해서 뉴톤이 고안한 방식에 따라 해를 구했다면 base 내장 패키지 apply 계열 함수를 사용하여 간결하게 코드 작성이 가능했다면 현재는 purrr 패키지를 통해 동일한 개념을 훨씬 더 깔끔하게 작성하게 되었다.

*apply 계열 함수는 각각의 자료형에 맞춰 기억하기가 쉽지 않아, 매번 도움말을 찾아 확인하고 코딩을 해야하는 번거러움이 많다. 데이터 분석을 함수형 프로그래밍 패러다임으로 실행하도록 purrr 팩키지가 개발되었다. 이를 통해 데이터 분석 작업이 수월하게 되어 저녁이 있는 삶이 길어질 것으로 기대된다.

purrr 팩키지를 불러와서 map_dbl() 함수에 구문에 맞게 작성하면 동일한 결과를 깔끔하게 얻을 수 있다. 즉,

- map_dbl(): 벡터, 데이터프레임, 리스트에 대해 함수를 원소별로 적용시켜 결과를 double 숫자형으로 출력시킨다.
- numbers: 함수를 각 원소별로 적용시킬 벡터 입력값
- find_root_recur: 앞서 작성한 뉴톤 방법으로 제곱근을 수치적으로 구하는 사용자 정의함수.
- init=1, eps = 10^-10: 뉴톤 방법을 구현한 사용자 정의함수에 필요한 초기값.

```
# `purrr`

library(purrr)
numbers <- c(16, 25, 36, 49, 64, 81)
map_dbl(numbers, find_root_recur, init=1, eps = 10^-10)
```

```
## [1] 4 5 6 7 8 9
```

## 18.3 사례: 붓꽃 데이터

구글 검색을 통해서 쉽게 iris(붓꽃) 데이터를 구할 수 있다. 이를 불러와서 각 종별로 setosa versicolor, virginica로 나눠 로컬 .csv 파일로 저장하고 나서 이를 다시

불러오는 사례를 함수형 프로그래밍으로 구현해본다.

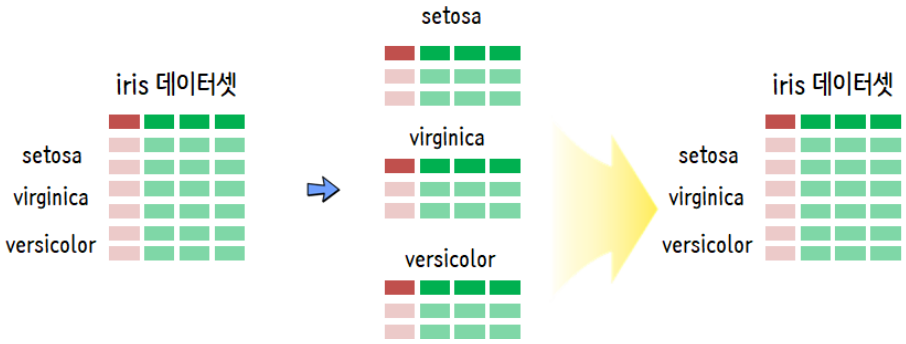



Figure 18.4: 붓꽃 데이터 품종별 불러오기

먼저 iris.csv 파일을 R로 불러와서 각 종별로 나눠서 iris_ .csv 파일형식으로 저장시킨다.

```
library(tidyverse)
iris_df <- read_csv("https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/d546eaee7

iris_species <- iris_df %>%
  count(species) %>% pull(species)

for(i in 1:nrow(iris_df)) {
  tmp_df <- iris_df %>%
    filter(species == iris_species[i])
  species_name <- iris_species[i]
  tmp_df %>% write_csv(paste0("data/iris_", species_name, ".csv"))
}

Sys.glob("data/iris_*.csv")
```

```
## [1] "data/iris_NA.csv"         "data/iris_setosa.csv"
## [3] "data/iris_versicolor.csv" "data/iris_virginica.csv"
```

로컬 파일 iris_ .csv 형식으로 저장된 데이터를 함수형 프로그래밍을 통해 불러와서 분석작업을 수행해보자. map() 함수를 사용해서 각 종별로 데이터를 깔끔하게 불러왔다.

iris_filename 벡터에 iris_ .csv과 경로명이 포함된 문자열을 저장시켜 놓고 read_csv() 함수를 각 벡터 원소에 적용시켜 출력값으로 리스트 iris_list 객체를 생성시켰다.

```
iris_filename <- c("data/iris_setosa.csv", "data/iris_versicolor.csv", "data/iris_virginica.csv")

iris_list <- map(iris_filename, read_csv) %>%
```

```
  set_names(iris_species)

iris_list %>%
  enframe()
```

```
## # A tibble: 3 x 2
##   name       value
##   <chr>      <list>
## 1 setosa     <spec_tbl_df [50 x 5]>
## 2 versicolor <spec_tbl_df [50 x 5]>
## 3 virginica  <spec_tbl_df [50 x 5]>
```

## 18.4 사례: 데이터 분석

iris_list 각 원소는 데이터프레임이라 summary 함수를 사용해서 기술 통계량을 구할 수도 있다. 물론 cor() 함수를 사용해서 iris_list의 각 원소를 지정하는 .x 여기서는 종별 데이터프레임에서 변수 두개를 추출하여 sepal_length, sepal_width 이 둘간의 스피커만 상관계수를 계산하는데 출력값이 double 연속형이라 map_dbl로 저정하여 작업시킨다.

```
map(iris_list, summary)
```

```
## $setosa
##   sepal_length   sepal_width    petal_length   petal_width
##  Min.   :4.30   Min.   :2.30   Min.   :1.00   Min.   :0.100
##  1st Qu.:4.80   1st Qu.:3.12   1st Qu.:1.40   1st Qu.:0.200
##  Median :5.00   Median :3.40   Median :1.50   Median :0.200
##  Mean   :5.01   Mean   :3.42   Mean   :1.46   Mean   :0.244
##  3rd Qu.:5.20   3rd Qu.:3.67   3rd Qu.:1.57   3rd Qu.:0.300
##  Max.   :5.80   Max.   :4.40   Max.   :1.90   Max.   :0.600
##    species
##  Length:50
##  Class :character
##  Mode  :character
##
##
##
##
## $versicolor
##   sepal_length   sepal_width    petal_length   petal_width     species
##  Min.   :4.90   Min.   :2.00   Min.   :3.00   Min.   :1.00   Length:50
##  1st Qu.:5.60   1st Qu.:2.52   1st Qu.:4.00   1st Qu.:1.20   Class :character
##  Median :5.90   Median :2.80   Median :4.35   Median :1.30   Mode  :character
##  Mean   :5.94   Mean   :2.77   Mean   :4.26   Mean   :1.33
##  3rd Qu.:6.30   3rd Qu.:3.00   3rd Qu.:4.60   3rd Qu.:1.50
```

```
##  Max.   :7.00   Max.   :3.40   Max.   :5.10   Max.   :1.80
##
## $virginica
##   sepal_length   sepal_width    petal_length   petal_width     species
##  Min.   :4.90   Min.   :2.20   Min.   :4.50   Min.   :1.40   Length:50
##  1st Qu.:6.22   1st Qu.:2.80   1st Qu.:5.10   1st Qu.:1.80   Class :character
##  Median :6.50   Median :3.00   Median :5.55   Median :2.00   Mode  :character
##  Mean   :6.59   Mean   :2.97   Mean   :5.55   Mean   :2.03
##  3rd Qu.:6.90   3rd Qu.:3.17   3rd Qu.:5.88   3rd Qu.:2.30
##  Max.   :7.90   Max.   :3.80   Max.   :6.90   Max.   :2.50
```

```
map_dbl(iris_list, ~cor(.x$sepal_length, .x$sepal_width, method = "spearman"))
```

```
##     setosa versicolor  virginica
##      0.769      0.518      0.427
```

## 18.5 사례: ggplot 시각화 [4]

list-column을 활용하여 티블(tibble) 데이터프레임에 담아서 시각화를 진행해도 되고, 다른 방법으로 리스트에 담아서 이를 한장에 찍는 것도 가능하다.

```
library(gapminder)

##    -----
three_country <-  c("Korea, Rep.", "Japan", "China")

gapminder_tbl <- gapminder %>%
  # filter(str_detect(continent, "Asia")) %>%
  group_by(continent, country) %>%
  nest() %>%
  filter(country %in% three_country ) %>%
  ungroup()

##         -----
gapminder_plot_tbl <- gapminder_tbl %>%
  select(-continent) %>%
  mutate(graph = map2(data, three_country,
                      ~ggplot(.x, aes(x=year, y=gdpPercap)) +
                        geom_line() +
                        labs(title=.y)))

gapminder_plot_tbl
```

```
## # A tibble: 3 x 3
##   country     data              graph
```

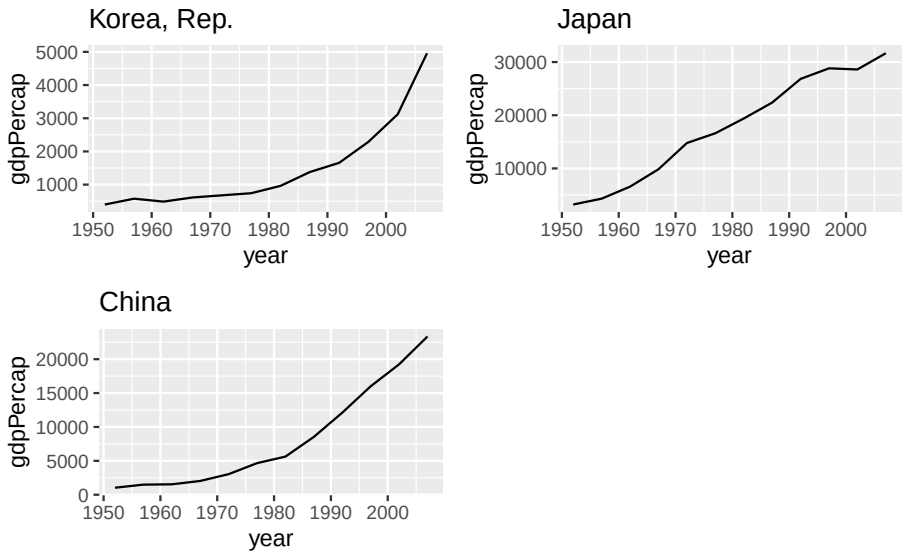
[4] Very statisticious (August 20, 2018), "Automating exploratory plots with ggplot2 and purrr"

```
##   <fct>       <list>            <list>
## 1 China       <tibble [12 x 4]> <gg>
## 2 Japan       <tibble [12 x 4]> <gg>
## 3 Korea, Rep. <tibble [12 x 4]> <gg>
```

```
##         -----
gapminder_plot <- map2(gapminder_tbl$data , three_country,
                       ~ggplot(.x, aes(x=year, y=gdpPercap)) +
                         geom_line() +
                         labs(title=.y))

# walk(gapminder_plot, print)

##         -      -----
cowplot::plot_grid(plotlist = gapminder_plot)
```




## 18.6 FP 이론과 실제

함수는 다음과 같이 될 수도 있어 함수형 프로그래밍(Functional Programming, FP) 언어가 된다. [5]

- 함수의 인자
- 함수로 반환
- 리스트에 저장
- 변수에 저장

[5] Advanced R, “Introduction”

- 무명함수
- 조작할 수 있다.

John Chambers 창시자가 말하는 R 계산의 기본원칙

- 존재하는 모든 것은 객체다. (Everything that exists is an object.)
- 일어나는 모든 것은 함수호출이다. (Everything that happens is a function call.)

```
library(tidyverse)
class(`%>%`)
```

```
## [1] "function"
```

```
class(`$`)
```

```
## [1] "function"
```

```
class(`<-`)
```

```
## [1] "function"
```

```
class(`+`)
```

```
## [1] "function"
```

### 18.6.1 (비)순수한 함수

**순수한 함수(pure function)**는 입력값에만 출력값이 의존하게 되는 특성과 부수효과(side-effect)를 갖지 않는 반면 **순수하지 않은 함수(impure function)**는 환경에 의존하며 부수효과도 갖는다.

#### 순수한 함수(pure function)

```
min(1:100)
```

```
## [1] 1
```

```
mean(1:100)
```

```
## [1] 50.5
```

#### 순수하지 않은 함수(impure function)

```
Sys.time()
```

```
## [1] "2022-10-17 19:24:05 KST"
```

```
rnorm(10)
```

```
##  [1]  0.492 -0.410 -0.931 -0.722  0.724  0.246  1.674  0.313 -0.224 -1.144
```

```
# write_csv("data/sample.csv")
```

### 18.6.2 무명함수와 매퍼

$\lambda$ (람다) 함수는 무명(anonymous) 함수는 함수명을 갖는 일반적인 함수와 비교하여 함수의 좋은 점은 그대로 누리면서 함수가 많아 함수명으로 메모리가 난잡하게 지져분해지는 것을 막을 수 있다.

무명함수로 기능르 구현한 후에 매퍼(mapper)를 사용해서 `as_mapper()` 명칭을 부여하여 함수처럼 사용하는 것도 가능하다. 매퍼(mapper)를 사용하는 이유를 다음과 같이 정리할 수 있다.

- 간결함(Concise)
- 가독성(Easy to read)
- 재사용성(Reusable)

정치인 페이스북 페이지에서 팬수를 추출한다. 그리고 이를 이름이 부은 리스트(named list)로 일자별 팬수 추이를 리스트로 준비한다. 그리고 나서 안철수, 문재인, 심상정 세 후보에 대한 최고 팬수증가를 무명함수로 계산한다.

```
library(tidyverse)
##
ahn_df  <- read_csv("data/fb_ahn.csv")  %>% rename(fans = ahn_fans) %>%
  mutate(fans_lag = lag(fans),
         fans_diff = fans - fans_lag) %>%
  select(fdate, fans = fans_diff) %>%
  filter(!is.na(fans))
moon_df <- read_csv("data/fb_moon.csv") %>% rename(fans = moon_fans) %>%
  mutate(fans_lag = lag(fans),
         fans_diff = fans - fans_lag) %>%
  select(fdate, fans = fans_diff) %>%
  filter(!is.na(fans))
sim_df  <- read_csv("data/fb_sim.csv")  %>% rename(fans = sim_fans) %>%
  mutate(fans_lag = lag(fans),
         fans_diff = fans - fans_lag) %>%
  select(fdate, fans = fans_diff) %>%
  filter(!is.na(fans))

convert_to_list <- function(df) {
  df_fans_v <- df$fans %>%
    set_names(df$fdate)
  return(df_fans_v)
}

ahn_v  <- convert_to_list(ahn_df)
moon_v <- convert_to_list(moon_df)
sim_v  <- convert_to_list(sim_df)

fans_lst <- list(ahn_fans  = ahn_v,
                 moon_fans = moon_v,
```

```
                sim_fans  = sim_v)

fans_lst %>% enframe()
```

```
## # A tibble: 3 x 2
##   name      value
##   <chr>     <list>
## 1 ahn_fans  <dbl [114]>
## 2 moon_fans <dbl [114]>
## 3 sim_fans  <dbl [114]>
```

```
##
map_dbl(fans_lst, ~max(.x))
```

```
##  ahn_fans moon_fans  sim_fans
##       796      1464      2029
```

rlang_lambda_function 무명함수로 increase_1000_fans 작성해서 일별 팬수 증가가 1000명 이상인 경우 keep() 함수를 사용해서 각 후보별로 추출할 수 있다. discard() 함수를 사용해서 반대로 버려버릴 수도 있다.

```
increase_1000_fans <- as_mapper( ~.x > 1000)

map(fans_lst, ~keep(.x, increase_1000_fans))
```

```
## $ahn_fans
## named numeric(0)
##
## $moon_fans
## 2017-03-28 2017-04-18 2017-04-20
##       1464       1310       1093
##
## $sim_fans
## 2017-03-12 2017-03-13 2017-04-14 2017-04-19 2017-04-20 2017-04-21 2017-04-24
##       1301       1079       1070       1441       1190       1025       1948
## 2017-04-25
##       2029
```

술어논리(predicate logic)은 조건을 테스트하여 참(TRUE), 거짓(FALSE)을 반환시킨다. every, some을 사용하여 팬수가 증가한 날이 매일 1,000명이 증가했는지, 전부는 아니고 일부 특정한 날에 1,000명이 증가했는지 파악할 수 있다.

```
##          1000      ?
map(fans_lst, ~every(.x, increase_1000_fans))
```

```
## $ahn_fans
## [1] FALSE
##
```

```
## $moon_fans
## [1] FALSE
##
## $sim_fans
## [1] FALSE
```

```
##                  1000      ?
map(fans_lst, ~some(.x, increase_1000_fans))
```

```
## $ahn_fans
## [1] FALSE
##
## $moon_fans
## [1] TRUE
##
## $sim_fans
## [1] TRUE
```

### 18.6.3 고차 함수(High order function)

고차 함수(High order function)는 함수의 인자로 함수를 받아 함수로 반환시키는 함수를 지칭한다. `high_order_fun` 함수는 함수를 인자(`func`)로 받아 함수를 반환시키는 고차함수다. 평균 함수(`mean`)를 인자로 넣어 출력값으로 `mean_na()` 함수를 새롭게 생성시킨다. `NA`가 포함된 벡터를 넣어 평균값을 계산하게 된다.

```
high_order_fun <- function(func){
  function(...){
    func(..., na.rm = TRUE)
  }
}

mean_na <- high_order_fun(mean)
mean_na( c(NA, 1:10) )
```

```
## [1] 5.5
```

벡터가 입력값으로 들어가서 벡터가 출력값으로 나오는 보통 함수(Regular Function)외에 고차함수는 3가지 유형이 있다.

- 벡터 → 함수: 함수공장(Function Factory)
- 함수 → 벡터: Functional - `for`루프를 `purrr` 팩키지 `map()` 함수로 대체
- 함수 → 함수: 함수연산자(Function Operator) - `Functional`과 함께 사용될 경우 `adverbs`로서 강력한 기능을 발휘

### 18.6.4 부사(adverbs)

`purrr` 팩키지의 대표적인 부사(adverbs)에는 `possibly()`와 `safely()`가 있다. 그외에도 `silently()`, `surely()` 등 다른 부사도 있으니 필요한 경우 `purrr` 팩키지

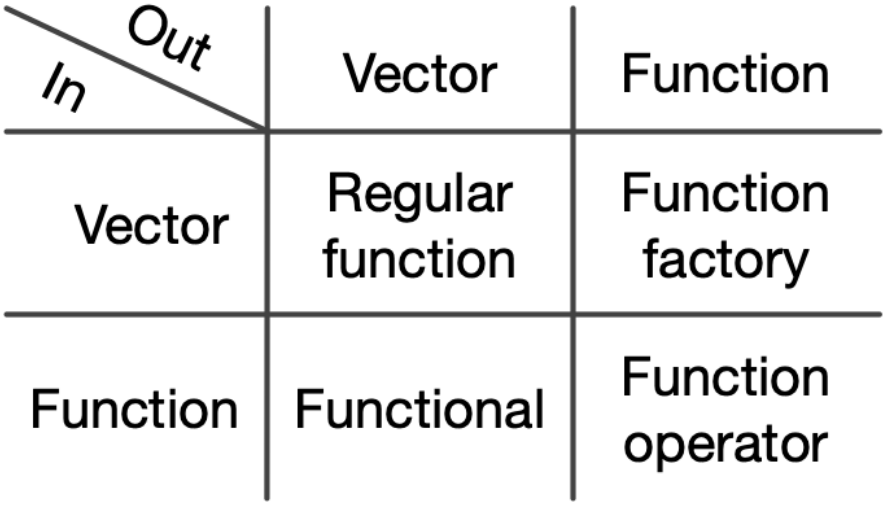



Figure 18.5: 고차함수 유형

문서를 참조한다.

`safely(mean)`은 동사 함수(`mean()`)를 받아 부사 `safely()`로 "부사 + 동사"로 기능이 추가된 부사 동사를 반환시킨다. 따라서, `NA`가 추가된 벡터를 넣을 경우 `$result`와 `$error`를 원소로 갖는 리스트를 반환시킨다.

```
mean_safe <- safely(mean)
class(mean_safe)
```

```
## [1] "function"
```

```
mean_safe(c(NA, 1:10))
```

```
## $result
## [1] NA
##
## $error
## NULL
```

이를 활용하여 오류처리작업을 간결하게 수행시킬 수 있다. `$result`와 `$error`을 원소로 갖는 리스트를 반환시키기 때문에 오류와 결과값을 추출하여 후속작업을 수행하여 디버깅하는데 유용하게 활용할 수 있다.

```
test_lst <- list("NA", 1,2,3,4,5)
log_safe <- safely(log)

map(test_lst, log_safe) %>%
  map("result")
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] 0
##
## [[3]]
## [1] 0.693
##
## [[4]]
## [1] 1.1
##
## [[5]]
## [1] 1.39
##
## [[6]]
## [1] 1.61
```

```
map(test_lst, log_safe) %>%
  map("error")
```

```
## [[1]]
## <simpleError in .Primitive("log")(x, base):                >
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
```

반면에 possibly()는 결과와 otherwise 값을 반환시켜서 오류가 발생되면 중단되는 것이 아니라 오류가 있다는 사실을 알고 예외처리시킨 후에 쭉 정상진행시킨다.

```
max_possibly <- possibly(sum, otherwise = "watch out")

max_possibly(c(1:10))
```

```
## [1] 55
```

```
max_possibly(c(NA, 1:10))
```

```
## [1] NA
```

```
max_possibly(c("NA", 1:10))
```

```
## [1] "watch out"
```

possibly()는 부울 논리값, NA, 문자열, 숫자를 반환시킬 수 있다.

transpose()와 결합하여 safely(), possibly() 결과를 변형시킬 수도 있다.

```
map(test_lst, log_safe) %>% length()
```

```
## [1] 6
```

```
map(test_lst, log_safe) %>% transpose() %>% length()
```

```
## [1] 2
```

compact()를 사용해서 NULL을 제거하는데, 앞서 possibly()의 인자로 otherwise=를 지정하는 경우 otherwise=NULL와 같이 정의해서 예외처리로 NULL을 만들어 내고 compact()로 정상처리된 데이터만 얻는 작업흐름을 갖춘다.

```
null_lst <- list(1, NULL, 3, 4, NULL, 6, 7, NA)
compact(null_lst)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 6
##
## [[5]]
## [1] 7
##
## [[6]]
## [1] NA
```

```
possibly_log <- possibly(log, otherwise = NULL)
map(null_lst, possibly_log) %>% compact()
```

```
## [[1]]
## [1] 0
##
## [[2]]
## [1] 1.1
##
## [[3]]
## [1] 1.39
##
## [[4]]
## [1] 1.79
##
## [[5]]
## [1] 1.95
##
## [[6]]
## [1] NA
```

## 18.7 깨끗한 코드(clean code) [6]

round_mean() 함수를 compose() 함수를 사용해서 mean() 함수로 평균을 구한 후에 round()함수로 반올림하는 코드를 다음과 같이 쉽게 작성할 수 있다.

```
round_mean <- compose(round, mean)
round_mean(1:10)
```

```
## [1] 6
```

두번째 사례로 전형적인 데이터 분석 사례로 lm() → anova() → tidy()를 통해 한방에 선형회귀 모형 산출물을 깨끗한 코드로 작성하는 사례를 살펴보자.

mtcars 데이터셋에서 연비 예측에 변수 두개를 넣고 일반적인 lm() 선형예측모형 제작방식과 동일하게 인자를 넣는다.

```
clean_lm <- compose(broom::tidy, anova, lm)
clean_lm(mpg ~ hp + wt, data=mtcars)
```

```
## # A tibble: 3 x 6
##   term         df sumsq meansq statistic   p.value
##   <chr>     <int> <dbl>  <dbl>     <dbl>     <dbl>
## 1 hp            1  678. 678.       101.   5.99e-11
## 2 wt            1  253. 253.        37.6  1.12e- 6
## 3 Residuals    29  195.   6.73       NA   NA
```

compose()를 통해 함수를 조합하는 경우 함수의 인자를 함께 전달해야될 경우가 있다. 이와 같은 경우 partial()을 사용해서 인자를 넘기는 함수를 제작하여

[6] Colin Fay, "A Crazy Little Thing Called {purrr} - Part 5: code optimization"

compose()에 넣어준다.

```
robust_round_mean <- compose(
  partial(round, digits=1),
  partial(mean, na.rm=TRUE))
robust_round_mean(c(NA, 1:10))
```

```
## [1] 5.5
```

리스트 칼럼(list-column)과 결합하여 모형에서 나온 데이터 분석결과를 깔끔하게 코드로 제작해보자. 먼저 lm을 돌려 모형 요약하는 함수 summary를 통해 r.squared값을 추출하는 함수를 summary_lm으로 제작한다.

그리고 나서 nest() 함수로 리스트 칼럼(list-column)을 만들고 두개의 집단 수동/자동을 나타내는 am 변수를 그룹으로 삼아 두 집단에 속한 수동/자동 데이터에 대한 선형 회귀모형을 적합시키고 나서 "r.squared"값을 추출하여 이를 티블 데이터프레임에 저장시킨다.

```
summary_lm <- compose(summary, lm)

mtcars %>%
  group_by(am) %>%
  nest() %>%
  mutate(lm_mod = map(data, ~ summary_lm(mpg ~ hp + wt, data = .x)),
         r_squared = map(lm_mod, "r.squared")) %>%
  unnest(r_squared)
```

```
## # A tibble: 2 x 4
## # Groups:   am [2]
##      am data               lm_mod     r_squared
##   <dbl> <list>             <list>         <dbl>
## 1     1 <tibble [13 x 10]> <smmry.lm>     0.837
## 2     0 <tibble [19 x 10]> <smmry.lm>     0.768
```

Fox, John. 2005. "The R Commander: A Basic Statistics Graphical User Interface to R." Journal of Statistical Software 14 (9): 1-42. https://doi.org/10.18637/jss.v014.i09.

———. 2016. Using the r Commander: A Point-and-Click Interface for r. Chapman; Hall/CRC.

Fox, John, and Milan Bouchet-Valat. 2021. Rcmdr: R Commander. https://socialsciences.mcmaster.ca/jfox/Misc/Rcmdr/.