챗GPT 코딩

인공지능(LLM)과 동료(Git/GitHub)와 함께 하는 프로그래밍

이광춘

2024년 03월 14일

목차

서문

코딩은 챗GPT로 대표되는 인공지능 시대에 필수적인 기술이다. 본 책 "챗GPT 코딩"은 프로그래밍의 기본 개념부터 실무에서 활용할 수 있는 다양한 도구와 기술까지 폭넓게 다룬다. 특히 챗GPT와 같은 최신 AI 기술을 자유자재로 다루기 위해 꼭 필요한 기본기를 상세히 설명한다.

책의 앞부분에서는 프로그래밍의 기초인 변수, 조건문, 반복문, 함수 등의 핵심 개념과 함께 데이터 과학, 기계 학습, 딥러닝, 나아가 생성형 AI의 기반이 되는 문자열, 파일, 리스트, 딕셔너리, 데이터프레임 등의 자료구조를 기초부터 다루고 있다.

이어서 프로그래밍과 불가분의 관계인 버전 제어에 대해 다룬다. 버전 제어를 대중화시킨 Git을 소개하고, Git의 기본 사용법부터 GitHub를 통한 원격 작업, 협업, 충돌 해결 등 실무에 필요한 기술을 자세히 안내한다. 공개 과학 기술 및 협업 과정에서 필수적으로 따라오게 되는 공개 과학, 라이선싱, 호스팅, 출처 표시, 통합 개발 환경 등도 소개하고 있다.

후반부에서는 정규표현식, 네트워크 프로그래밍, 웹서비스 활용, 작업 자동화 등 실무에서 유용한 도구와 기술을 소개하고, 마지막으로 챗GPT를 프로그래밍에 어떻게 활용할 수 있는지 설명하며, 코딩에 대한 새로운 시각을 제시한다.

본 책을 통해 프로그래밍 전반에 대한 이해를 높이고, AI 기술을 활용한 프로그래밍 작업의 생산성을 극대화할 수 있을 것이다. 지금부터 새로운 코딩 여정을 시작해보길 바란다.

책의 구성

이 책은 통계 및 데이터 사이언스를 염두에 두고 프로그래밍을 처음 접하는 독자들을 위해 기획되었다. 특히 디지털 전환 시대 갈수록 중요성을 더하고 있는 통계 및 데이터 사이언스 분야에서 코딩에 중점을 두고 집필을 진행하면서 저자가 그동안 깊숙이 관여한 "소프트웨어 카펜트리"[**gonzalez2019software**], "정보과학을 위한 파이썬"[**severance2015python**], "챗GPT 데이터 사이언스"[**lee2024sql**, **leeshin2023**,

lee2024quarto]를 기반으로 작성되었음을 밝혀둔다. 이 책은 총 4부로 나누어져 있으며, 각 부는 프로그래밍의 기초부터 실제 활용까지 단계적으로 설명하고 있다.

1부 "프로그래밍"에서는 프로그래밍을 학습해야 하는 이유와 함께 변수, 표현식, 문장, 조건부 실행, 함수, 반복 등 프로그래밍의 기본 개념을 다루고 있다. 이를 통해 독자들은 프로그래밍의 기초를 탄탄히 다질 수 있고 실제 R과 파이썬 프로그램을 직접 작성하고 테스트할 수 있다.

2부 "자료구조"에서는 문자열, 파일, 리스트, 딕셔너리, 데이터프레임 등 다양한 자료구조를 소개하고 있다. 자료구조는 데이터를 효율적으로 저장하고 관리하는 방법으로, 프로그래밍에서 매우 중요한 역할을 한다. 모든 자료구조를 다루지 않고 꼭 필요한 것만 선택하여 설명하였다.

3부 "버전제어와 협업"에서는 Git을 이용한 버전 관리와 GitHub을 통한 협업 방법을 설명한다. 이를 통해 독자들은 효과적으로 프로젝트를 관리하고, 다른 동료 개발자, 과학기술 연구자들과 협업하는 방법을 배울 수 있다. 코딩을 혼자서만 하지 않고, 디지털 전환 시대를 넘어 AI 시대 다른 동료와 함께 협업하여 프로그램을 작성하는 방법과 연관된 지식을 학습하게 된다.

4부 "분야별 코딩"에서는 1~3부에서 학습한 기초를 바탕으로 정규 표현식, 네트워크 프로그래밍, 웹서비스 API 사용, 데이터베이스와 SQL, 작업 자동화, 시각화 등 다양한 분야에서의 프로그래밍 활용 방법을 살펴본다. 또한 최근 주목받고 있는 챗GPT를 활용한 코딩 방법도 다루고 있다.

이 책을 통해 독자들은 프로그래밍의 기초를 다지고, 챗GTP로 대표되는 AI시대 실제 활용 방법을 배움으로써 프로그래밍 실력을 향상시킬 수 있을 것으로 기대되고, 또한 버전 관리와 협업 방법을 학습함으로써 챗GPT와 같은 AI와 함께 동료 개발자들과 협업하여 다양한 방식으로 프로그램을 작성하는 방법을 배울 수 있다.

감사의 글

이 책이 탄생할 수 있도록 도움을 주신 여러분께 깊은 감사의 마음을 표합니다.

공익법인 한국 R 사용자회가 없었다면 데이터 과학분야 챗GPT 시리즈가 세상에 나오지 못했을 것입니다. 한국 R 사용자회의 유충현 회장님, 신종화 사무처장님, 홍성학 감사님, 올해부터 새롭게 공익법인 한국 R 사용자를 이끌어주실 형환희 회장님께 감사드립니다.

또한 이 책은 2014년 처음 몸담게 된 소프트웨어 카펜트리 그렉 윌슨 박사님과 Python for Informatics 저자인 미시건 대학 찰스 세브란스 교수님을 비롯한 전세계 수많은 익명의 기여자들의 노력과 지원이 있었고, 서울 R 미트업에서 발표해주시고 참여해주신 수많은 분들이 격려와 영감을 주셨기에 가능했습니다.

이 책이 출간되는데 있어 이들 모든 분들의 도움 없이는 어려웠을 것입니다. 그동안의 관심과 지원에 깊은 감사를 드리며, 이 책이 데이터 과학의 발전과 독자들에게 도움이 될 수 있기를 바라는 마음으로 마무리하겠습니다.

2024년 3월 속초 청초호

이광춘

제 I 편

2부 자료구조

제 1 장

문자열

1.1 문자열은 시퀀스다

문자열은 여러 문자들의 시퀀스(sequence)다. 꺾쇠 연산자로 한 번에 하나씩 문자에 접근한다. `substr()` 함수를 사용해서 바로 특정 문자를 추출할 수도 있지만, `strsplit()` 함수로 문자열을 문자의 벡터로 다루는 방법도 있다.

1.1.1 R

```
#¦ label: r-string-seq
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")[[1]]

letter <- fruit_letter[1]
```

1.1.2 파이썬

```
#¦ label: r-string-seq
fruit = 'banana'
letter = fruit[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.

1.1.3 R

```
#¦ label: r-string-seq-output
letter
```

1.1.4 파이썬

```
#¦ label: py-string-seq-output
letter
```

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

하지만, R은 사람 친화적이기 때문에 b가 'banana'의 첫 번째 문자가 되고 a가 두 번째, n이 세 번째 문자가 된다.

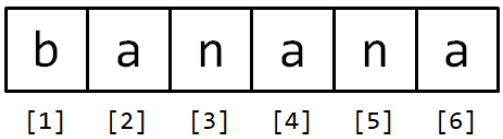

그림 1.1: 바나나 문자열

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

1.1.5 R

```
#¦ label: r-string-integer
fruit_letter[1.5]
```

[1]컴퓨터에서 어떤 주소로부터 간격을 두고 떨어진 주소와의 거리를 말한다. 기억 장치가 페이지 혹은 세그먼트 단위로 나누어져 있을 때 하나의 시작 주소로부터 오프셋만큼 떨어진 위치를 나타낸다. 네이버 지식백과(IT용어사전, 한국정보통신기술협회)

1.1.6 파이썬

```
#¦ label: py-string-integer
letter = fruit[1.5]
#> TypeError: string indices must be integers
```

1.2 length() 함수 사용 문자열 길이 구하기

length() 함수는 문자열의 문자 갯수를 반환하는 내장함수다.

1.2.1 R

```
#¦ label: r-string-length
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")[[1]]

length(fruit_letter)
```

1.2.2 파이썬

```
#¦ label: py-string-length
fruit = 'banana'
len(fruit)
#> 6
```

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

1.2.3 R

```
#¦ label: r-string-last
len <- length(fruit_letter)
fruit_letter[len]
```

1.2.4 파이썬

```
#¦ label: py-string-last
length = len(fruit)
last = fruit[length]
#> IndexError: string index out of range
```

파이썬에서는 인덱스 오류(IndexError)가 발생하는데 이유는 'banana'에 6번 인덱스 문자가 없기 때문이다. 0에서부터 시작했기 때문에 6개 문자는 0에서부터 5까지 번호가 매겨졌다. 마지막 문자를 얻기 위해서 length에서 1을 빼야 한다. `fruit[-1]`은 마지막 문자를, `fruit[-2]`는 끝에서 두 번째 문자를 가리킨다. 하지만, R에서는 사람이 생각하는 방식대로 마지막 문자를 얻는다.

1.3 루프를 사용한 문자열 순회

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

1.3.1 R

```
#¦ label: r-string-traversal
index <- 1

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

1.3.2 파이썬

```
#¦ label: py-string-traversal
index = 0

while index < len(fruit):
    letter = fruit[index]
```

```
    print(letter)
    index = index + 1
```

while 루프가 문자열을 순회하여 문자열을 한 줄에 한 글자씩 화면에 출력한다. 루프 조건이 index <= length(fruit_letter)이어서, index가 문자열 길이와 같을 때, 조건은 거짓이 되고, 루프의 몸통 부문은 실행되지 않는다. R이 접근한 마지막 length(fruit_letter) 인덱스 문자로, 문자열의 마지막 문자다.

> ⚠ 연습문제
>
> 문자열의 마지막 문자에서 시작해서, 문자열 처음으로 역진행하면서 한 줄에 한 자씩 화면에 출력하는 while 루프를 작성하세요.

순회를 작성하는 또 다른 방법은 for 루프다.

1.3.3 R

```
#| label: r-string-banana-for
for(char in fruit_letter) {
  print(char)
}
```

1.3.4 파이썬

```
#| label: py-string-banana-for}
for char in fruit:
    print(char)
```

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

1.4 문자열 슬라이스

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

1.4.1 R

```
#¦ label: r-string-slice}
s <- strsplit('Monty Python', "")[[1]]
paste(s[1:5], collapse="")
#> [1] "Monty"
paste(s[7:12], collapse="")
#> [1] "Python"
```

1.4.2 파이썬

```
#¦ label: py-string-slice}
s = 'Monty Python'
print(s[0:5])
#> [1] "Monty"
print(s[6:13])
#> [1] "Python"
```

`[n:m]` 연산자는 `n`번째 문자부터 `m`번째 문자까지의 문자열 부분을 반환한다.

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

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

1.4.3 R

```
#¦ label: r-string-slice-banana
fruit <- 'banana'
fruit_letter <- strsplit(fruit, "")[[1]]

paste(head(fruit_letter, 3), collapse="")
paste(tail(fruit_letter, 3), collapse="")
```

1.4.4 파이썬

```
#¦ label: py-string-slice-banana
fruit = 'banana'

fruit[:3]
fruit[3:]
```

만약 첫 번째 인덱스가 두 번째보다 크거나 같은 경우 파이썬에서는 결과가 인용부호로 표현되는 빈 문자열(empty string)이 된다. 하지만, R에서는 해당 인덱스에 해당되는 문자가 추출된다.

1.4.5 R

```
#¦ label: r-string-slice-empty
fruit_letter[3:3]
```

1.4.6 파이썬

```
#¦ label: py-string-slice-empty
fruit = 'banana'
fruit[3:3]
#> ''
```

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

> ⚠ 연습문제 (파이썬)
>
> `fruit`이 문자열로 주어졌을 때, `fruit[:]`의 의미는 무엇인가요?

1.5 문자열은 불변이다 (파이썬)

문자열 내부에 있는 문자를 변경하려고 대입문 왼쪽편에 `[]` 연산자를 사용하고 싶은 유혹이 있을 것이다. 예를 들어 다음과 같다.

1.5.1 R

```
#¦ label: r-string-immutable
greeting <- strsplit('Hello, world!', "")[[1]]
greeting[1] <- 'J'

paste0(greeting, collapse="")
```

1.5.2 파이썬

```
#¦ label: py-string-immutable
greeting = 'Hello, world!'
greeting[0] = 'J'
#> TypeError: 'str' object does not support item assignment

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

파이썬에서 "TypeError: 'str' object does not support item assignment" 오류가 발생하는데 파이썬 문자열이 불변(immutable)하기 때문이다. 즉, 파이썬에서 문자열의 특정 문자를 직접 변경하려고 할 때 이 오류가 발생한다. 반면에, R에서는 문자열 자체가 불변 객체로 취급되지 않기 때문에 `strsplit` 함수를 사용하여 문자열을 문자의 벡터로 변환하면, 벡터의 각 요소는 별도의 문자열로 취급되어 개별적으로 변경할 수 있다. R은 파이썬과 달리 불변 문자열에 대한 제약이 없기 때문에 오류를 발생시키지않는다.

파이썬에서 "객체(object)"는 문자열이고, 대입하고자 하는 문자는 "항목(item)"이다. 지금으로서 객체는 값과 동일하지만, 나중에 객체 정의를 좀 더 상세화할 것이다. 항목은 순서 값 중의 하나다. 최선의 방법은 원래 문자열을 변형한 새로운 문자열을 생성하는 것이다.

새로운 첫 문자에 `greeting` 문자열 슬라이스를 연결한다. 원래 문자열에는 어떤 영향도 주지 않는 새로운 문자열이 생성되었다.

1.6 루프 사용 문자 계수하기

다음 프로그램은 문자열에 문자 `a`가 나타나는 횟수를 계수(counting)한다.

1.6.1 R

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

count
```

1.6.2 파이썬

```
#¦ label: py-string-a-count
word = 'banana'
count = 0

for letter in word:
    if letter == 'a':
        count = count + 1

print(count)
```

상기 프로그램은 **계수기(counter)**라고 부르는 또 다른 연산 패턴을 보여준다. 변수 count는 0으로 초기화되고, 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.

1.7 %in% 연산자

\index{%in% 연산자} \index{연산자!%in%}

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

1.7.1 R

```
#¦ label: r-string-in-op
'a' %in% strsplit('banana', "")[[1]]
'seed' %in% strsplit('banana', "")[[1]]
```

1.7.2 파이썬

```
#¦ label: py-string-in-op
'a' in 'banana'
#> True
'seed' in 'banana'
#> False
```

1.8 문자열 비교

비교 연산자도 문자열에서 동작한다. 두 문자열이 같은지를 살펴보자.

1.8.1 R

```
#¦ label: r-string-comparison
word <- 'banana'

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

1.8.2 파이썬

```
#¦ label: py-string-comparison
word = 'banana'

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

다른 비교 연산자는 단어를 알파벳 순서로 정렬하는 데 유용하다.

1.8.3 R

```
#¦ label: r-string-pineapple
word <- 'Pineapple'

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

1.8.4 파이썬

```
#¦ label: py-string-pineapple
word = 'Pineapple'

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

R과 파이썬 같은 프로그래밍 언어는 사람과 동일한 방식으로 대문자와 소문자를 다루지 않는다. 모든 대문자는 소문자 앞에 온다. 프로그래밍 언어에서 대문자와 소문자를 다루는 방식은 ASCII 코드 값을 기반으로 한다. ASCII 코드에서 대문자(A-Z)는 65부터 90까지의 값을, 소문자(a-z)는 97부터 122까지의 값을 갖기 때문에 대문자가 숫자적으로 소문자보다 먼저 나오기 때문에 문자열을 정렬하거나 비교할 때, 대문자가 소문자 앞에 위치하게 된다.

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

이러한 문제를 다루는 일반적인 방식은 비교 연산을 수행하기 전에 문자열을 표준 포맷으로 예를 들어 모두 소문자로 변환하는 것이다. 경우에 따라서 “Pineapple”로 무장한 사람들로부터 여러분을 보호해야 하는 것도 명심한다.

1.9 문자열 함수

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

> **파이썬 dir 함수**
>
> 객체에 대해 이용 가능한 메서드를 보여주는 dir 함수가 파이썬에 있다. type 함수는 객체의 자료형(type)을 보여주고, dir은 객체에 사용될 수 있는 메서드를 보여준다.

```
#¦ label: py-string-dir
#¦
stuff = 'Hello world'
type(stuff)
#> <type 'str'>
methods = [method for method in dir(stuff) if not method.startswith('__') and
 ↪  not method.endswith('__')]
methods
#> ['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 ↪  'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 ↪  'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 ↪  'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
 ↪  'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
 ↪  'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 ↪  'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 ↪  'swapcase', 'title', 'translate', 'upper', 'zfill']
```

> dir 함수가 메서드 목록을 보여주고, 메서드에 대한 간단한 문서 정보는 help를 사용할 수 있지만, 문자열 메서드에 대한 좀 더 좋은 문서 정보는 https://docs.python.org/3/library/string.html에서 찾을 수 있다.
> 인자를 받고 값을 반환한다는 점에서 메서드(method)를 호출하는 것은 함수를 호출하는 것과 유사하지만, 구문은 다르다. 구분자로 점을 사용해서 변수명에 메서드명을 붙여 메서드를 호출한다.
> 예를 들어, upper 메서드는 문자열을 받아 모두 대문자로 변환된 새로운 문자열을 반환한다. 함수 구문 upper(word) 대신에, word.upper() 메서드 구문을 사용한다.

하지만, 함수형 프로그래밍 패러다임으로 문자열을 객체로 두고 함수를 적용시켜 다양한 작업을 하는 것이 일반적이다. tidyverse 패키지를 설치하게 되면 stringr 패키지가 구성요소로 포함되어 있다. str_로 시작되는 다양한 함수가 지원된다.

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

1.9.1 R

```
#¦ label: r-string-upper
library(stringr)

word <- 'banana'
new_word <- stringr::str_to_upper(word)
new_word
#> [1] "BANANA"
```

1.9.2 파이썬

```
#¦ label: r-string-upper
word = 'banana'
new_word = word.upper()
print(new_word)
#> BANANA
```

동일한 작업을 함수형 패러다임으로 str_to_upper(word)와 같이 표현하는 데 반해, 객체지향으로 구현하면 파이썬 같은 경우 word.upper() 메서드 구문이 사용된다.

예를 들어, 문자열 안에 문자열의 위치를 찾는 str_locate(), str_locate_all()이라는 문자열 함수가 있다. str_locate()는 매칭되는 첫 번째만 반환하는 반면에 str_locate_all()은 매칭되는 전부를 반환하는 차이가 있다.

1.9.3 R

```
#¦ label: r-string-locate
str_locate(word, 'a')
#>      start end
#> [1,]     2   2
```

1.9.4 파이썬

```
#¦ label: py-string-locate
word = 'banana'
index = word.find('a')
print(index)
#> 1
```

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.

1.9.5 R

```
#¦ label: r-string-locate-substring
str_locate_all(word, 'na')[[1]]
#>      start end
#> [1,]     3   4
#> [2,]     5   6
```

1.9.6 파이썬

```
#¦ label: py-string-locate-substring
word.find('na')
#> 2
word.find('na', 3)
#> 4
```

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

1.9.7 R

```
#¦ label: r-string-strip
line <- '     Here we go '
str_trim(line)
#> [1] "Here we go"
```

1.9.8 파이썬

```
#¦ label: py-string-strip
line = '     Here we go '
line.strip()
#> '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.

1.9.9 R

```
#¦ label: r-string-startwith
line <- '좋은 하루되세요!'
str_detect(line, '^좋은')
#> [1] TRUE
```

1.9.10 파이썬

```
#¦ label: py-string-startwith
line = '좋은 하루되세요!'
line.startswith('좋은')
#> True
line.startswith('조은')
#> False
```

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

1.9.11 R

```
#¦ label: r-string-startwith-to-lower
line <- 'Please have a nice day'
str_detect(line, '^p')

str_to_lower(line)

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

1.9.12 파이썬

```
#¦ label: py-string-startwith-to-lower
line = 'Please have a nice day'
line.startswith('p')
#> False
line.lower()
#> 'please have a nice day'
line.lower().startswith('p')
#> True
```

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

> ⚠ 연습문제
>
> 앞선 예제와 유사한 함수인 `str_count()`로 불리는 문자열 메서드가 `stringr` 패키지 내부에 있다. `? str_count()` 도움말로 `str_count()` 함수에 대한 문서를 읽고, 문자열 ’banana’의 문자가 몇 개인지 계수하는 메서드 호출 프로그램을 작성하세요.

1.10 문자열 파싱

종종, 문자열을 들여다보고 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.

1.10.1 R

```
#¦ label: r-string-email-parsing
data <- 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
atpos <- str_locate(data, '@')
```

```
atpos[1,1]
#> start
#>    5
sppos <- str_locate_all(data, ' ')[[1]]
sppos
#>      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[2,2] - 1)
#> [1] "uct.ac.za"
```

1.10.2 파이썬

```
#¦ label: py-string-email-parsing
data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
atpos = data.find('@')
print(atpos)
#> 21
sppos = data.find(' ',atpos)
print(sppos)
#> 31
host = data[atpos+1:sppos]
print(host)
#> uct.ac.za
```

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

1.11 서식 연산자

서식 연산자(format operator) Base R의 `sprintf()` 함수에 C언어 스타일로 `%`를 사용하기도 하지만 glue: Interpreted String Literals 패키지도 최근에 많이 사용된다. `glue` 패키지 `{}`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. 동일한 기능을 `stringr` 패키지 `str_glue()` 함수로 수행할 수 있다.

첫 피연산자는 서식 문자열(format string)로 두 번째 피연산자가 어떤 형식으로 표현되

는지를 명세하는 하나 혹은 그 이상의 서식 순서(format sequence)를 담고 있다. 결과값은 문자열이다.

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

1.11.1 R

```
#¦ label: r-string-format
camels <- 42
sprintf('%d', camels)
#> [1] "42"
str_glue("{camels}")
#> 42
```

1.11.2 파이썬

```
#¦ label: py-string-format
camels = 42
'%d' % 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.

1.11.3 R

```
#¦ label: r-string-format-camels
camels <- 42
sprintf('I have spotted %d camels.', camels)
#> [1] "I have spotted 42 camels."
str_glue('I have spotted {camels} camels.')
#> I have spotted 42 camels.
```

1.11.4 파이썬

```
#¦ label: py-string-format-camels
camels = 42
'I have spotted %d camels.' % camels
#> 'I have spotted 42 camels.'
```

만약 문자열 서식 순서가 하나 이상이라면, 두 번째 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.

1.11.5 R

```
#¦ label: r-string-matching
sprintf('In %d years I have spotted %g %s', 3, 0.1, 'camels')
#> [1] "In 3 years I have spotted 0.1 camels"

str_glue('In {3} years I have spotted {0.1} {"camels"}')
#> In 3 years I have spotted 0.1 camels
```

1.11.6 파이썬

```
#¦ label: py-string-matching
# `format` 메서드
result = 'In {} years I have spotted {} {}'.format(3, 0.1, 'camels')
print(result)
#> In 3 years I have spotted 0.1 camels

# `f-string`
years = 3
camels = 0.1
result = f'In {years} years I have spotted {camels} {"camels"}'
print(result)
#> In 3 years I have spotted 0.1 camels
```

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

1.11.7 R

```
#¦ label: r-string-type-error
sprintf('%d %d %d', 1, 2)
#> sprintf("%d %d %d", 1, 2)에서 다음과 같은 에러가 발생했습니다:인자들의 수가 너무
 ↪  적습니다
sprintf('%d', 'dollars')
#> sprintf("%d", "dollars")에서 다음과 같은 에러가 발생했습니다: '%d'는 유효하지 않
 ↪  은 포맷입니다; 문자형 객체들에는 포맷 %s를 사용해주세요
```

1.11.8 파이썬

```
#¦ label: py-string-type-error
'%d %d %d' % (1, 2)
#> TypeError: not enough arguments for format string
'%d' % 'dollars'
#> TypeError: %d format: a number is required, not str
```

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

1.12 디버깅

프로그램을 작성하면서 배양해야 하는 기술은 항상 자신에게 질문을 하는 것이다. "여기서 무엇이 잘못될 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.

1.12.1 R

```
#¦ label: r-string-debug
while(TRUE) {
  line <- readline(prompt = '> ')
  if(substr(line,1,1) == "#") {
    next
  }
  if(line == 'done') {
```

```
    break
  }
  print(line)
}
```

1.12.2 파이썬

```
#¦ label: py-string-debug
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

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

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

R

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

파이썬

```
if line.startswith('#') :
```

가디언 패턴(guardian pattern)을 사용한 if문으로 문자열에 적어도 하나의 문자가 있는 경우만 두 번째 논리 표현식이 평가되도록 코드를 작성한다.

R

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

파이썬

```
if len(line) > 0 and line[0] == '#' :
```

1.13 용어 정의

- **계수기(counter)**: 무언가를 계수하기 위해서 사용되는 변수로 일반적으로 0으로 초기화하고 나서 증가한다.
- **빈 문자열(empty string)**: 두 인용부호로 표현되고, 어떤 문자도 없고 길이가 0인 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)**: 메서드를 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.
- **메서드(method)**: 객체와 연관되어 점 표기법을 사용하여 호출되는 함수.
- **객체(object)**: 변수가 참조하는 무엇. 지금은 "객체"와 "값"을 구별 없이 사용한다.

- **검색(search)**: 찾고자 하는 것을 찾았을 때 멈추는 순회 패턴.
- **순서(sequence)**: 정돈된 집합. 즉, 정수 인덱스로 각각의 값이 확인되는 값의 집합.

- **슬라이스(slice)**: 인덱스 범위로 지정되는 문자열 부분.
- **순회(traverse)**: 순서(sequence)의 항목을 반복적으로 훑기, 각각에 대해서는 동일한 연산을 수행.

연습문제

1. 다음 문자열에서 숫자를 뽑아내는 R 코드를 작성하라.

```
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 변환하라.

제 2 장

파일

2.1 영속성

지금까지, 프로그램을 어떻게 작성하고 조건문, 함수, 반복을 사용하여 중앙처리장치(CPU, Central Processing Unit)에 프로그래머의 의도를 전달하는지 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.

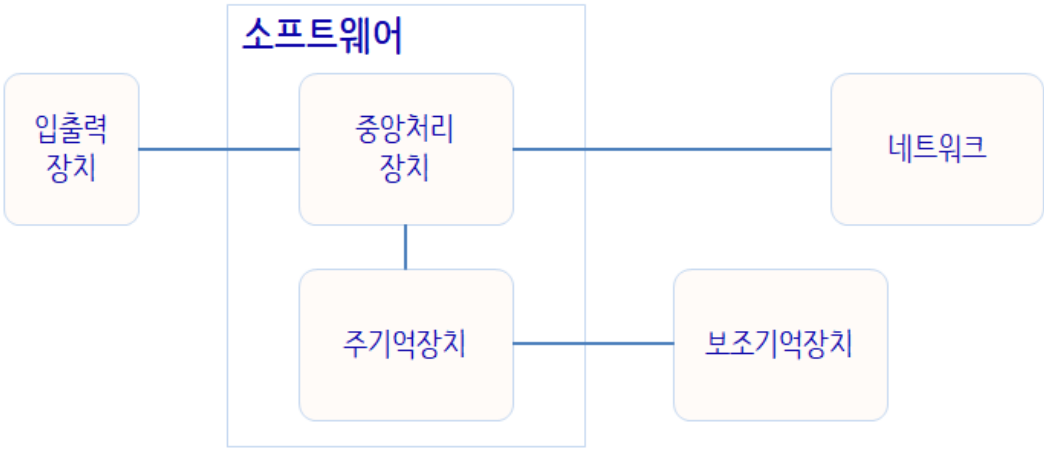

그림 2.1: 소프트웨어 아키텍처

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

2.2 파일 열기

하드 디스크 파일을 읽거나 쓰려고 할 때, 파일을 열어야(open) 한다. 파일을 열 때 각 파일 데이터가 어디에 저장되었는지를 알고 있는 운영체제와 커뮤니케이션한다. 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("https:// www.dr-chuck.com/ py4inf/ code/ mbox.txt", destfile = "mbox.txt")` 명령어를 사용하여 코딩을 시작하는 디렉토리에 `mbox.txt` 이름으로 저장한다.

2.2.1 R

```
#¦ label: r-file-open
download.file("https://www.dr-chuck.com/py4inf/code/mbox.txt",
              destfile = "mbox.txt")
fhand <- file("mbox.txt", open = "r")
fhand
#> A connection with
#> description "mbox.txt"
#> class       "file"
#> mode        "r"
#> text        "text"
#> opened      "opened"
#> can read    "yes"
#> can write   "no"
```

2.2.2 파이썬

```
#¦ label: py-file-open
fhand = open('mbox.txt')
print(fhand)
#> <_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp949'>
```

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.

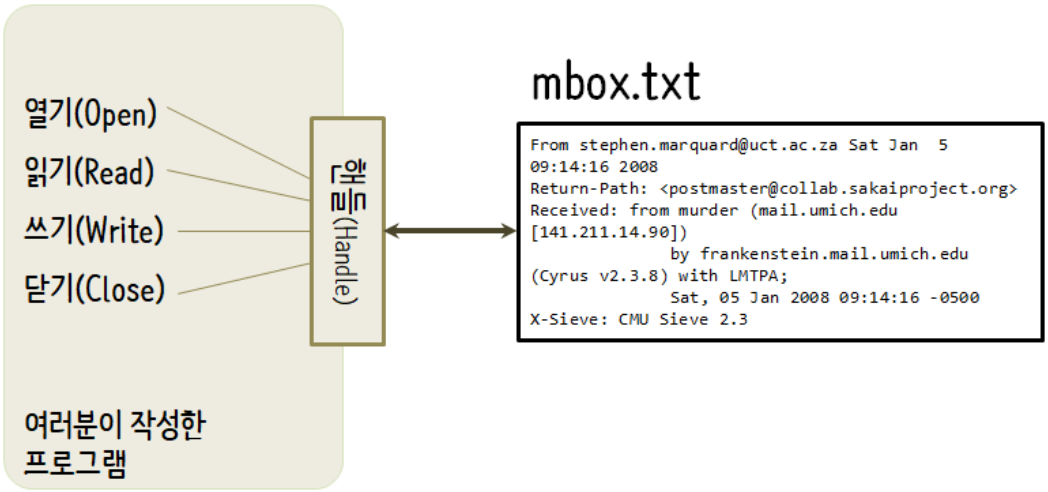

그림 2.2: 파일 핸들

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

2.2.3 R

```
#¦ label: r-file-open-error
fhand <- file("stuff.txt", "r")
#> file("stuff.txt", "r")에서 다음과 같은 에러가 발생했습니다:커넥션을 열 수 없습니
 ↪ 다
#> 추가정보: 경고메시지(들):
#> file("stuff.txt", "r")에서:
#>   파일 'stuff.txt'를 여는데 실패했습니다: No such file or directory
```

2.2.4 파이썬

```
#¦ label: py-file-open-error
fhand = open('stuff.txt')
#> Traceback (most recent call last):
#>   File "D:\tcs\gpt-coding\file.py", line 1, in <module>
#>     fhand = open('stuff.txt')
#> FileNotFoundError: [Errno 2] No such file or directory: 'data/stuff.txt'
```

나중에 `tryCatch()`를 가지고, 존재하지 않는 파일을 열려고 하는 상황을 좀 더 우아하게 처리할 것이다. 최근에 사용자 중심으로 R에 다양한 기능이 추가되어 `tidyverse` 패키지 일부를 구성하는 `readr` 패키지의 `read_lines()` 함수를 통해 인터넷 웹사이트에서 바로 불러오는 것도 가능하다. 하지만, `readr::read_lines()` 함수는 줄바꿈 문자를 가정하고 동작하기 때문에 제대로 파일을 못 읽어오는 경우도 종종 있다.

2.2.5 R

```
#¦ label: r-file-open-readLines
txt_file <- readLines("mbox.txt")

head(txt_file)
```

2.2.6 파이썬

```
#¦ label: py-file-open-tidyverse
with open("mbox.txt", "r") as f:
    txt_file = f.readlines()

for line in txt_file[:6]:
    print(line, end='')
```

2.3 텍스트 파일과 라인

R 문자열이 문자 순서(sequence)로 간주되듯이 마찬가지로 텍스트 파일은 줄(라인, line) 순서(sequence)로 생각될 수 있다. 예를 들어, 다음은 오픈 소스 프로젝트 개발 팀에서 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 버전 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, 줄바꿈 문자로 불리는 "줄의 끝(end of the line)"을 표시하는 특수 문자가 있다.

R에서, 문자열 상수 역슬래시-n(\n)으로 줄바꿈 문자를 표현한다. 두 문자처럼 보이지만, 사실은 단일 문자이다. 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.

2.3.1 R

```
#¦ label: r-file-newline
stuff <- 'Hello\nWorld!'
stuff
#> [1] "Hello\nWorld!"
print(stuff)
#> [1] "Hello\nWorld!"
stuff <- 'X\nY'
print(stuff)
#> [1] "X\nY"
str_length(stuff)
#> [1] 3
```

2.3.2 파이썬

```
#¦ label: py-file-newline
stuff = 'Hello\nWorld!'
stuff
#> 'Hello\nWorld!'
print(stuff)
#> Hello
#> World!
stuff = 'X\nY'
print(stuff)
#> X
#> Y
len(stuff)
#> 3
```

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

그래서, 파일 라인을 볼 때, 라인 끝을 표시하는 줄바꿈으로 불리는 눈에 보이지 않는 특수 문자가 각 줄의 끝에 있다고 상상할 필요가 있다.

그래서, 줄바꿈 문자는 파일에 있는 문자를 라인으로 분리한다.

2.4 파일 읽어오기

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

2.4.1 R

```
#¦ label: r-file-open-count
fhand <- file('mbox.txt', open = "r")
count <- 0

for(line in readLines(fhand)) {
  count <- count + 1
}

message('행수:', count, "\n")
#> 행수:132045
close(fhand)
```

2.4.2 파이썬

```
#¦ label: py-file-open-count
fhand = open('mbox.txt')
count = 0

for line in fhand:
    count = count + 1

print('행수:', count)
#> 행수: 132045
```

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

file 함수가 전체 파일을 바로 읽지 못하는 이유는 파일이 수 기가바이트(GB) 파일 크기를 가질 수도 있기 때문이다. file 문장은 파일 크기에 관계없이 파일을 여는 데 시간이 동일하게 걸린다. 실질적으로 for 루프가 파일로부터 자료를 읽어오는 역할을 한다.

for 루프를 사용해서 이같은 방식으로 파일을 읽어올 때, 줄바꿈 문자를 사용해서 파일 자료를 라인 단위로 쪼갠다. 파이썬에서 줄바꿈 문자까지 각 라인 단위로 읽고, for 루프가 매번 반복할 때마다 line변수에 줄바꿈을 마지막 문자로 포함한다.

for 루프가 데이터를 한 번에 한 줄씩 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.

2.4.3 R

```
#¦ label: r-file-input
download.file("https://www.dr-chuck.com/py4inf/code/mbox-short.txt",
              destfile = "mbox-short.txt")

fhand <- file("mbox-short.txt", open = "r")
inp <- readLines(fhand)
inp_str <- paste(inp, collapse = "\n")
close(fhand)
```

```
print(nchar(inp_str))
#> [1] 94626
print(substr(inp_str, 1, 20))
#> [1] "From stephen.marquar"
```

2.4.4 파이썬

```
#¦ label: py-file-input
fhand = open('mbox-short.txt')
inp = fhand.read()
print(len(inp))
#> 94626
print(inp[:20])
#> From stephen.marquar
```

상기 예제에서, `mbox-short.txt` 전체 파일 콘텐츠(94,626 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 줄바꿈 문자를 포함한 모든 문자는 변수 `inp`에 대입된 매우 큰 문자열이다. 파일 데이터가 컴퓨터 주기억장치가 안정적으로 감당해 낼 수 있을 때만, 이런 형식의 `nchar()` 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.

2.5 파일 검색

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

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

2.5.1 R

```
#¦ label: r-file-print-from
fhand <- readLines("mbox-short.txt")

for(line in fhand) {
  if(startsWith(line, "From:")) {
    print(line)
  }
}
```

2.5.2 파이썬

```
#¦ label: py-file-print-from
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print(line)
```

이 프로그램이 실행하면 다음 출력값을 얻는다.

```
[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"
...
```

"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 찾게 되면, 그 라인에서 특정 연산을 수행하는 것이다.

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

2.5.3 R

```
#¦ label: r-file-print-from-skip
fhand <- readLines("mbox-short.txt")
```

```
for (line in fhand) {

  line <- trimws(line, which = "right")

  # 관심 없는 라인 건너뛰기
  if (!startsWith(line, "From:")) {
    next
  }
  # 관심 있는 라인 작업
  print(line)
}
```

2.5.4 파이썬

```
#¦ label: py-file-print-from-skip
fhand = open('mbox-short.txt')

for line in fhand:
    line = line.rstrip()
    # 관심 없는 라인 건너뛰기
    if not line.startswith('From:'):
        continue
    # 관심 있는 라인 작업
    print(line)
```

프로그램의 출력값은 동일하다. 흥미롭지 않는 라인은 "From:"으로 시작하지 않는 라인이라 `next`문을 사용해서 건너뛴다. "흥미로운" 라인(즉, "From:"으로 시작하는 라인)에 대해서는 연산 처리를 수행한다.

`str_detect()` 문자열 함수를 사용해서 검색 문자열이 라인 어디에 있는지를 찾아주는 텍스트 편집기 검색 기능을 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. `stringr` 패키지 의존성 대신 `grepl()` 함수를 사용할 수도 있다. `if`문 `!str_detect(line, "@uct.ac.za")` 대신 `grepl("@uct.ac.za", line) == FALSE`로 대체한다.

2.5.5 R

```
#¦ label: r-file-find-email
library(stringr)

fhand <- readLines("mbox-short.txt")

for (line in fhand) {
  line <- trimws(line, which = "right")
  if (!str_detect(line, "@uct.ac.za")) {
    next
  }
  print(line)
}
```

2.5.6 파이썬

```
#¦ label: py-file-find-email
fhand = open('mbox-short.txt')

for line in fhand:
    line = line.rstrip()
    if '@uct.ac.za' not in line:
        continue
    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.ac.za using -f"
[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.za using -f"
[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.za using -f"
```

2.6 사용자가 파일명 선택

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

다음과 같이 `commandArgs`을 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.

R

```r
cat("Enter the file name: ")
fname <- readLines(file("stdin"), 1)

fhand <- readLines(fname)

count <- 0

for (line in fhand) {
  if (startsWith(line, "Subject:")) {
    count <- count + 1
  }
}

print(paste('There were', count, 'subject lines in', fname))
```

파이썬

```python
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
```

[1] R 스크립트를 인자와 함께 실행

```
        count += 1
print('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.

```
$ rscript file-user-input.R
Enter the file name: mbox.txt
경고메시지(들):
사용되지 않는 커넥션 3 (stdin)를 닫습니다
[1] "There were 1797 subject lines in mbox.txt"

$ rscript file-user-input.R
Enter the file name: mbox-short.txt
경고메시지(들):
사용되지 않는 커넥션 3 (stdin)를 닫습니다
[1] "There were 27 subject lines in mbox-short.txt"
There were 27 subject lines in ../data/mbox-short.txt
```

다음 절을 살펴보기 전에, 이 프로그램을 검토하면서 자신에게 다음을 질문해보자. "여기서 무엇이 잘못될 수 있을까?" 또는 "이 간결하고 멋진 프로그램이 오류를 발생시키고 바로 종료되어 사용자에게 나쁜 인상을 남길 수 있게 만드는 것은 무엇일까?"

2.7 tryCatch 사용하기

여러분에게 엿보지 말라고 말씀드렸다. 이번이 마지막 기회다. 사용자가 파일명이 아닌 뭔가 다른 것을 입력하면 어떻게 될까?

```
$ rscript file-user-input.R "missing.txt"
file(con, "r")에서 다음과 같은 에러가 발생했습니다:커넥션을 열 수 없습니다
호출: readLines -> file
추가정보: 경고메시지(들):
file(con, "r")에서:
  파일 'missing.txt'를 여는데 실패했습니다: No such file or directory
실행이 정지되었습니다

$ rscript file-user-input.R "na na boo boo"
file(con, "r")에서 다음과 같은 에러가 발생했습니다:커넥션을 열 수 없습니다
호출: readLines -> file
추가정보: 경고메시지(들):
file(con, "r")에서:
```

```
  파일 'na na boo boo'를 여는데 실패했습니다: No such file or directory
실행이 정지되었습니다
```

웃을 일은 절대 아니다. 사용자는 결국 여러분이 작성한 프로그램을 망가뜨리기 위해 고의든 악의를 가지든 가능한 모든 수단을 강구할 것이다. 사실, 소프트웨어 개발팀의 중요한 부분은 **품질 보증(Quality Assurance, QA)**이라는 조직이다. 품질보증 조직은 프로그래머가 만든 소프트웨어를 망가뜨리기 위해 가능한 말도 안 되는 것을 수행한다.

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 고쳐본다. 파일 열기 `file()` 호출이 잘못될 수 있다고 가정하고, `file()` 호출이 실패할 때를 대비해서 다음과 같이 복구 코드를 추가한다.

2.7.1 R 파일 `file-user-input-try.R`

```
cat("Enter the file name: ")
fname <- readLines(file("stdin"), 1)

fileOpened <- FALSE

result <- try({
  fhand <- readLines(fname)
  fileOpened <- TRUE
}, silent = TRUE)

if (!fileOpened) {
  cat("File cannot be opened:", fname, "\n")
  q("no")
}

count <- 0

for (line in fhand) {
  if (startsWith(line, "Subject:")) {
    count <- count + 1
  }
}

cat("There were", count, "subject lines in", fname, "\n")
```

2.7.2 파이썬 file-user-input-try.py

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count += 1

print('There were', count, 'subject lines in', fname)
```

이제 사용자 혹은 품질 보증 조직에서 올바르지 않거나 어처구니없는 파일명을 입력했을 때, 버그를 try() 함수로 잡아서 우아하게 복구한다.

```
$ rscript file-user-input-try.R
Enter the file name: mbox.txt
경고메시지(들):
사용되지 않는 커넥션 3 (stdin)를 닫습니다
There were 1797 subject lines in mbox.txt

$ rscript file-user-input-try.R
Enter the file name: no no no
경고메시지(들):
file(con, "r")에서:
  파일 'no no no'를 여는데 실패했습니다: No such file or directory
File cannot be opened: no no no
```

R 프로그램을 작성할 때 readLines() 파일 열기 호출을 보호하는 것은 try()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 좀 더 자신감이 생기게 되면, 다른 R 프로그래머와 동일한 문제에 대해 두 가지 동치하는 해답을 가지고 어떤 접근법이 좀 더 "알스러운지"에 대한 현답을 찾는 데도 관여하게 된다.

"좀 더 알스럽게" 되는 이유는 프로그래밍이 엔지니어링적인 면과 예술적인 면을 동시에 가지고 있기 때문이다. 항상 무언가를 단지 작동하는 것에만 관심이 있지 않고, 프로그램으로 작성한 해결책이 좀 더 우아하고, 다른 동료에 의해서 우아한 것으로 인정되

기를 또한 원한다.

2.8 파일에 쓰기

파일에 쓰기 위해서는 두 번째 매개 변수로 'w' 모드로 파일을 열어야 한다.

R

```
fout <- file("output.txt", "w")
print(fout)
close(fout)
```

파이썬

```
fout = open('output.txt', 'w')
print(fout)
```

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. 라인을 끝내고 싶을 때, 명시적으로 줄바꿈 문자를 삽입해서 파일에 쓰도록 라인 끝을 필히 관리해야 한다. `print`문이 자동적으로 줄바꿈을 추가하듯이 `writeLines()` 함수도 자동적으로 줄바꿈을 추가한다.

R

```
fout <- file("output.txt", "w")
line1 <- "This here's the wattle,\n"
writeLines(line1, fout)
line2 <- 'the emblem of our land.\n'
writeLines(line2, fout)
close(fout)
```

파이썬

```
fout = open("output.txt", "w")
line1 = "This here's the wattle,\n"
fout.write(line1)
line2 = 'the emblem of our land.\n'
fout.write(line2)
fout.close()
```

파일 쓰기가 끝났을 때, 파일을 필히 닫아야 한다. 파일을 닫는 것은 데이터 마지막 비트까지 디스크에 물리적으로 쓰여져서, 전원이 나가더라도 자료가 유실되지 않는 역할을 한다.

파일 읽기로 연 파일을 닫을 수 있지만, 몇 개 파일을 열어놓았다면 약간 단정치 못하게 끝날 수 있다. 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 써넣은 결과는 다음과 같다.

```
$ cat output.txt
This here's the wattle,

the emblem of our land.
```

2.9 디버깅

파일을 읽고 쓸 때, 공백 때문에 종종 문제에 봉착한다. 이런 종류의 오류는 공백, 탭, 줄바꿈이 눈에 보이지 않기 때문에 디버그하기도 쉽지 않다.

2.9.1 R

```
#| label: r-file-debug
s <- '1 2\t 3\n 4'
print(cat(s))
#> 1 2   3
#>  4NULL
```

2.9.2 파이썬

```
#¦ label: py-file-debug
s = '1 2\t 3\n 4'
print(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.

내장함수 `dput`이 도움이 될 수 있다. 인자로 임의 객체를 잡아 객체 문자열 표현식으로 반환한다. 문자열 공백문자는 역슬래시 시퀀스로 표현된다.

2.9.3 R

```
#¦ label: r-file-debug
s <- '1 2\t 3\n 4'
dput(s)
#> "1 2\t 3\n 4"
```

2.9.4 파이썬

```
#¦ label: py-file-debug
s = '1 2\t 3\n 4'
print(repr(s))
#> '1 2\t 3\n 4'
```

여러분이 봉착하는 또 다른 문제는 다른 시스템에서는 라인 끝을 표기하기 위해서 다른 문자를 사용한다는 점이다. 어떤 시스템은 `\n`으로 줄바꿈을 표기하고, 다른 시스템은 `\r`으로 반환 문자(return character)를 사용한다. 둘 다 모두 사용하는 시스템도 있다. 파일을 다른 시스템으로 이식한다면, 이러한 불일치가 문제를 야기한다.

대부분의 시스템에는 A 포맷에서 B 포맷으로 변환하는 응용프로그램이 있다. https://en.wikipedia.org/wiki/Newline에서 응용프로그램을 찾을 수 있고, 좀 더 많은 것을 읽을 수 있다. 물론, 여러분이 직접 프로그램을 작성할 수도 있다.

2.10 용어 정의

- **잡기(catch)**: tryCatch 함수를 사용하여 프로그램이 예외 상황으로 인해 종료되는 것을 방지하는 과정으로 예외가 발생할 때 실행할 코드를 지정하고,정상적으로 코드를 계속 실행할 수 있게 한다.
- **줄바꿈**: 라인 끝을 표기하기 위해 파일이나 문자열에 사용되는 특수 문자.
- **파이썬다움(Pythonic)**: 파이썬에서 우아하게 작동하는 기술. "try와 catch를 사용하는 것은 파일이 없는 경우를 복구하는 파이썬스러운 방식이다."
- **품질 보증(Quality Assurance, QA)**: 소프트웨어 제품의 전반적인 품질을 보증하는 데 집중하는 사람이나 조직. 품질 보증은 소프트웨어 제품을 시험하고, 제품이 시장에 출시되기 전에 문제를 확인하는 데 관여한다.
- **텍스트 파일**: 하드디스크 같은 영구 저장소에 저장된 일련의 문자 집합.

연습문제

1. 파일을 읽고 한 줄씩 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 도달했을 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.

제 3 장

리스트

3.1 리스트는 시퀀스다

문자열처럼, 리스트(list)는 값의 시퀀스다. 문자열에서, 값은 문자지만, 리스트에서는 임의 자료형(type)도 될 수 있다. 리스트 값은 요소(elements)나 때때로 항목(items)으로 불린다.

신규 리스트를 생성하는 방법은 여러 가지가 있다. 가장 간단한 방법은 `list()` 함수로 요소를 감싸는 것이다.

R

```
list(10, 20, 30, 40)
list('crunchy frog', 'ram bladder', 'lark vomit')
list('고양이', '호랑이', '사자')
```

파이썬

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

첫 번째 예제는 4개 정수 리스트다. 두번째 예제는 3개 문자열 리스트다. 세 번째 예제는 한글 문자열 리스트다. 리스트 문자열 요소가 동일한 자료형(type)일 필요는 없다. 다

음 리스트는 문자열, 부동 소수점 숫자, 정수, (아!) 또 다른 리스트를 담고 있다.

R

```
list('spam', 2.0, 5, list(10, 20))
```

파이썬

```
['spam', 2.0, 5, [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.

3.1.1 R

```
#¦ label: r-list-data
cheeses <- list('체다', '브리', '까망베르')
numbers <- list(17, 123)
empty <- list()
message(cheeses, numbers, empty)
#> 체다브리까망베르17123
```

3.1.2 파이썬

```
#¦ label: py-list-data
cheeses = ['체다', '브리', '까망베르']
numbers = [17, 123]
empty = []
print(cheeses, numbers, empty)
#> ['체다', '브리', '까망베르'] [17, 123] []
```

3.2 리스트는 변경 가능하다

리스트 요소에 접근하는 구문은 문자열 문자에 접근하는 것과 동일한 꺾쇠 괄호 연산자다. 꺾쇠 괄호 내부 표현식은 인덱스를 명세한다. 기억할 것은 인덱스는 1에서부터 시작한다는 것이다.

3.2.1 R

```
#¦ label: r-list-access}
cheeses[1]
#> [[1]]
#> [1] "체다"
```

3.2.2 파이썬

```
#¦ label: py-list-access}
cheeses[0]
#> '체다'
```

문자열과 마찬가지로, 리스트 항목 순서를 바꾸거나, 리스트에 새로운 항목을 다시 대입할 수 있기 때문에 리스트는 변경 가능하다. 꺾쇠 괄호 연산자가 대입문 왼쪽편에 나타날 때, 새로 대입될 리스트 요소를 나타낸다.

3.2.3 R

```
#¦ label: r-list-mutable
numbers <- list(17, 123)

numbers[1] <- 5
print(numbers)
#> [[1]]
#> [1] 5
#>
#> [[2]]
#> [1] 123
```

3.2.4 파이썬

```
#¦ label: py-list-mutable
numbers = [17, 123]
numbers[1] = 5
print(numbers)
#> [17, 5]
```

리스트 numbers의 첫 번째 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 NULL이 반환된다.
- 인덱스가 음의 값이면, 해당 리스트 원소가 누락된다.

`%in%` 연산자도 또한 리스트에서 동작하니 리스트 원소로 존재하는지 여부를 판별하는데 사용할 수 있다.

\index{%in% 연산자} \index{연산자!%in%}

3.2.5 R

```
#¦ label: r-list-in
cheeses <- list('체다', '브리', '까망베르')
'체다' %in% cheeses
#> [1] TRUE
'고르곤졸라' %in% cheeses
#> [1] FALSE
```

3.2.6 파이썬

```
#¦ label: py-list-in
cheeses = ['체다', '브리', '까망베르']
'체다' in cheeses
True
```

```
'고르곤졸라' in cheeses
False
```

3.3 리스트 순회법

리스트 요소를 순회하는 가장 흔한 방법은 for 문을 사용하는 것이다. 문자열에서 사용한 것과 구문은 동일하다.

R

```
for(cheese in cheeses) {
  print(cheese)
}
```

파이썬

```
for cheese in cheeses:
    print cheese
```

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.

3.3.1 R

```
#¦ label: r-list-seq_along
numbers <- list(1, 2, 3, 4, 5)

for(i in seq_along(numbers)){
  numbers[[i]] <- numbers[[i]] * 2
  message("결과:", numbers[[i]])
}
```

3.3.2 파이썬

```
#¦ label: py-list-seq_along
numbers = [1, 2, 3, 4, 5]

for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
    print("결과:", numbers[i])
```

상기 루프는 리스트를 순회하고 각 요소를 갱신한다. `seq_along()` 함수는 1에서 n까지 리스트 인덱스를 반환한다. 여기서, n은 리스트 길이가 된다. 매번 루프가 반복될 때마다, `i`는 다음 요소 인덱스를 얻는다. 몸통 부문 대입문은 `i`를 사용해서 요소의 이전 값을 읽고 새 값을 대입한다.

빈 리스트(`list()`)에 대해서 `for` 문은 결코 몸통 부문을 실행하지 않는다.

R

```
for(x in list()) {
  print('이런 일은 절대 발생하지 않는다.')
}
```

파이썬

```
for x in empty:
    print('이런 일은 절대 발생하지 않는다.')
```

리스트가 또 다른 리스트를 담을 수 있지만, 중첩된 리스트는 여전히 요소 하나로 간주된다. 다음 리스트 길이는 4다.

R

```
list('spam', 1, list('브리', '체다', '까멩베르'), list(1, 2, 3))
```

파이썬

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

3.4 리스트 연산자

c() 함수 연산자는 리스트를 추가하여 결합시킨다.

3.4.1 R

```
#¦ label: r-list-append
a <- list(1, 2, 3)
b <- list(4, 5, 6)
c <- c(a, b)
print(c)
```

3.4.2 파이썬

```
#¦ label: py-list-append
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
#> [1, 2, 3, 4, 5, 6]
```

유사하게 rep() 함수를 활용하면 주어진 횟수만큼 리스트를 반복한다.

3.4.3 R

```
#¦ label: r-list-repeat
rep(list(0), 4)

rep(list(1, 2, 3), 3)
```

3.4.4 파이썬

```
#¦ label: py-list-repeat
[0] * 4
#> [0, 0, 0, 0]
[1, 2, 3] * 3
#> [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

첫 예제는 `list(0)`을 4회 반복한다. 두 번째 예제는 `list(1, 2, 3)` 리스트를 3회 반복한다.

3.5 리스트 슬라이스

슬라이스(slice) 연산자는 리스트에도 또한 동작한다.

3.5.1 R

```
#¦ label: r-list-slice}
t <- list('a', 'b', 'c', 'd', 'e', 'f')

t[2:3]

t[1:4]

t[4:length(t)]

t
```

3.5.2 파이썬

```
#¦ label: py-list-slice
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3]
#> ['b', 'c']
 t[:4]
#> ['a', 'b', 'c', 'd']
t[3:]
#> ['d', 'e', 'f']
t[:]
```

```
#> ['a', 'b', 'c', 'd', 'e', 'f']
```

첫 번째 인덱스를 1로 지정하면, 슬라이스는 처음부터 시작한다. 두 번째 인덱스를 length() 함수로 리스트 길이를 지정하면, 슬라이스는 끝까지 간다. 그래서 양쪽의 인덱스를 생략하면, t같이 지정하면 슬라이스 결과는 전체 리스트를 복사한 것이 된다.

리스트는 변경이 가능하기 때문에 리스트를 접고, 돌리고, 훼손하는 연산을 수행하기 전에 복사본을 만들어두는 것이 유용하다.

대입문 왼편의 슬라이스 연산자로 복수의 요소를 갱신할 수 있다.

3.5.3 R

```
#¦ label: r-list-update
t <- list('a', 'b', 'c', 'd', 'e', 'f')
t[2:3] <- list(1, 2)
print(t)
```

3.5.4 파이썬

```
#¦ label: py-list-update
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3] = [1, 2]
print(t)
#> ['a', 1, 2, 'd', 'e', 'f']
```

3.6 리스트 함수

R은 리스트 자료형에 연산할 수 있는 함수를 제공한다. 예를 들어, 덧붙이기(append) 함수는 리스트 끝에 신규 요소를 추가한다.

3.6.1 R

```
#¦ label: r-list-function-append
t <- list('a', 'b', 'c')
append(t, 'd')
```

3.6.2 파이썬

```
#¦ label: py-list-function-append}
t = ['a', 'b', 'c']
t.append('d')
print(t)
#> ['a', 'b', 'c', '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 리스트 내 원소를 정렬시킨다.

3.6.3 R

```
#¦ label: r-list-function-sort
t <- list('d', 'c', 'e', 'b', 'a')

t[order(unlist(t), decreasing=FALSE)]
```

3.6.4 파이썬

```
#¦ label: py-list-function-sort
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
#> ['a', 'b', 'c', 'd', 'e']
```

3.7 리스트 요소 삭제

리스트 요소를 삭제하는 방법이 몇 가지 있다. 리스트 요소 인덱스를 알고 있다면, 숫자 인덱스 앞에 - 기호를 붙여 사용한다.

3.7.1 R

```
#¦ label: r-list-delete-by-index
t <- list('a', 'b', 'c')
```

```
x <- t[2]
t <- t[-2]
print(x)
print(t)
```

3.7.2 파이썬

```
#¦ label: py-list-delete-by-index
t = ['a', 'b', 'c']
x = t.pop(1)
print(t)
#> ['a', 'c']
print(x)
#> b
```

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]] 리스트 인덱스를 통해 요소에 접근하고 NULL을 대입하여 삭제한다.

3.7.3 R

```
#¦ label: r-list-delete-by-index
t = list('a', 'b', 'c')
t[[1]] <- NULL
t
```

3.7.4 파이썬

```
#¦ label: py-list-delete-by-index
t = ['a', 'b', 'c']
del t[1]
print(t)
#> ['a', 'c']
```

(인덱스 혹은 리스트 요소 이름이 아닌) 제거할 요소값을 알고 있다면, 리스트 요소 값을 활용해서 제거하는 것도 가능하다.

3.7.5 R

```
#¦ label: r-list-delete-by-value
t <- list('x', 'y', 'z')

t[t != "y"]
```

3.7.6 파이썬

```
#¦ label: py-list-delete-by-value
t = ['a', 'b', 'c']
t.remove('b')
print(t)
#> ['a', 'c']
```

하나 이상의 요소를 제거하기 위해서, 슬라이스 인덱스(slice index)를 사용하는 것도 가능하다.

3.7.7 R

```
#¦ label: r-list-delete-slice
t <- list('a', 'b', 'c', 'd', 'e', 'f')
t[-c(2:5)]
```

3.7.8 파이썬

```
#¦ label: py-list-delete-slice
t = ['a', 'b', 'c', 'd', 'e', 'f']
del t[1:5]
print(t)
#> ['a', 'f']
```

위의 예제에서 2에서 5까지 모든 요소를 선택하고 선택된 모든 요소를 제거한다.

3.8 리스트와 함수

루프를 작성하지 않고도 리스트를 빠르게 살펴볼 수 있는 리스트에 적용할 수 있는 내장 함수를 활용하는 것도 방법이지만, 깔끔한 세상(tidyverse) 생태계의 일원인 `purrr` 함수형 프로그래밍 패키지에 내장된 함수를 활용하는 것도 권장된다.

하지만, 다음과 같이 1차원 리스트는 `unlist()` 함수를 활용하여 벡터로 변환해서 사용하는 것이 편리한 경우가 많다.

3.8.1 R

```
#¦ label: r-list-unlist
nums <- list(3, 41, 12, 9, 74, 15) %>% unlist

length(nums)
#> 6
max(nums)
#> 74
min(nums)
#> 3
sum(nums)
#> 154
sum(nums) / length(nums)
#> 25.66667
```

3.8.2 파이썬

```
#¦ label: py-list-unlist
nums = [3, 41, 12, 9, 74, 15]
```

```
print(len(nums))
#> 6
print(max(nums))
#> 74
print(min(nums))
#> 3
print(sum(nums))
#> 154
print(sum(nums) / len(nums))
#> 25.666666666666668
```

`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 반드시 자료형을 사전에 점검하고 활용하도록 한다.

리스트를 사용해서, 앞서 작성한 프로그램을 다시 작성해서 사용자가 입력한 숫자 목록 평균을 계산한다.

우선 리스트 없이 평균을 계산하는 프로그램:

3.8.3 R

```
#¦ label: r-list-user-input-average
## 프로그램 명칭: `list_average.R`
total <- 0
count <- 0

while(TRUE) {
  cat("숫자를 입력하세요: ")
  inp <- readLines(file("stdin"), 1)
  if(inp == 'done') {
    break
  }
  value <- as.numeric(inp)
  total <- total + value
  count <- count + 1
}

average <- total / count
message('평균:', average)
#> 숫자를 입력하세요: 10
#> 숫자를 입력하세요: 20
#> 숫자를 입력하세요: 30
```

```
#> 숫자를 입력하세요: done
#> 평균: 20
```

3.8.4 파이썬

```
#| label: py-list-user-input-average
total = 0
count = 0

while True:
    inp = input('숫자를 입력하세요: ')
    if inp == 'done':
        break
    value = float(inp)
    total += value
    count += 1

average = total / count
print('평균:', average)

#> 숫자를 입력하세요: 10
#> 숫자를 입력하세요: 20
#> 숫자를 입력하세요: 30
#> 숫자를 입력하세요: done
#> 평균: 20.0
```

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")
숫자를 입력하세요: 10
숫자를 입력하세요: 20
숫자를 입력하세요: 30
숫자를 입력하세요: done
평균: 20
```

단순하게, 사용자가 입력한 각 숫자를 리스트로 기억하고 내장 함수를 사용해서 프로그램 마지막에 합계와 갯수를 통해 평균을 계산한다.

3.8.5 R

```r
#¦ label: r-list-user-input-average
# list_average2.R ... 자료구조 사용
numlist <- list()

while(TRUE) {
  cat("숫자를 입력하세요: ")
  inp <- readLines(file("stdin"), 1)
  if(inp == 'done') {
    break
  }
  value <- as.numeric(inp)
  numlist <- append(numlist, value)
}

average <- sum(unlist(numlist)) / length(numlist)
message('평균:', average)
```

3.8.6 파이썬

```python
#¦ label: py-list-user-input-average
numlist = list()

while True:
    inp = input('숫자를 입력하세요: ')
    if inp == 'done':
        break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('평균:', 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_average2.R")
숫자를 입력하세요: 10
숫자를 입력하세요: 20
```

```
숫자를 입력하세요: 30
숫자를 입력하세요: done
평균: 20
```

3.9 리스트와 문자열

문자열(string)은 **문자 시퀀스**이고, 리스트는 **값 시퀀스**이다. 하지만 리스트 문자는 문자열과 같지는 않다. 문자열을 리스트 문자로 변환하기 위해서, `strsplit()` 함수를 사용한다.

3.9.1 R

```
#¦ label: py-list-strsplit
s <- 'spam'
t <- strsplit(s, "")[[1]] ¦>  strsplit(NULL)
print(t)
```

3.9.2 파이썬

```
#¦ label: r-list-strsplit
s = 'spam'
t = list(s)
print(t)
#> ['s', 'p', 'a', 'm']
```

`list`는 내장 함수 이름이기 때문에, 변수명으로 사용하는 것을 피해야 한다. `l`을 사용하면 1처럼 보이기 때문에 가능하면 피한다. 그래서, `t`를 사용했다.

`strsplit()` 함수는 문자열을 구분자(이번 경우에는 `NULL`)를 사용해서 문자 각각으로 쪼갠다. 문자열 단어로 쪼개려면, 구분자를 바꿔 예를 들어 공백을 기준으로 쪼갠다.

3.9.3 R

```
#¦ label: r-list-split-separator
s <- list('이제는 디지털 글쓰기가 대세다.')
t <- strsplit(s[[1]], " ")[[1]] %>% strsplit(" ")
t
```

3.9.4 파이썬

```
#¦ label: py-list-split-separator
s = '이제는 디지털 글쓰기가 대세다.'
t = s.split()
print(t)
#> ['이제는', '디지털', '글쓰기가', '대세다.']
```

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 지정하는 데 사용되는 구분자(delimiter)를 활용하여 분할 `strsplit()` 함수를 호출한다. 다음 예제는 구분자로 하이픈('-')을 사용한 사례이다.

3.9.5 R

```
#¦ label: r-list-delimiter
s <- list('spam-spam-spam')
delimiter <- '-'
strsplit(s[[1]], delimiter)[[1]] %>% strsplit(" ")
```

3.9.6 파이썬

```
#¦ label: py-list-delimiter
s = 'spam-spam-spam'
delimiter = '-'
print(s.split(delimiter))
#> ['spam', 'spam', 'spam']
```

합병(`paste`) 함수는 분할(`strsplit`) 함수의 역이다. 문자열 리스트를 받아 리스트 요소를 연결한다.

3.9.7 R

```
#¦ label: r-list-paste
t <- list('이제는', '디지털', '글쓰기가', '대세다.')
delimiter <- ' '
```

```
paste0(t, delimiter, collapse = "")
#> [1] "이제는 디지털 글쓰기가 대세다. "
```

3.9.8 파이썬

```
#¦ label: py-list-paste
t = ['이제는', '디지털', '글쓰기가', '대세다.']
delimiter = ' '
print(delimiter.join(t))
#> 이제는 디지털 글쓰기가 대세다.
```

상기의 경우, 구분자가 공백 문자여서 결합(paste) 함수가 단어 사이에 공백을 넣는다. 공백 없이 문자열을 결합하기 위해서, 구분자로 빈 문자열 ''을 사용한다.

3.10 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` 패키지 분할 `str_split()` 함수가 매우 효과적이다. 작은 프로그램을 작성하여 "From"으로 시작하는 라인을 찾고 `str_split()` 함수로 파싱하고 라인의 흥미로운 부분을 출력한다.

3.10.1 R

```
#¦ label: r-list-email-parsing
download.file("https://www.dr-chuck.com/py4inf/code/mbox-short.txt",
              destfile = "mbox-short.txt")

fhand <- readLines('mbox-short.txt')

for (line in fhand) {
    line <- trimws(line)  # 공백 제거
    if (length(grep("^From ", line)) == 0) {
        next
    }
    words <- unlist(strsplit(line, " "))
```

```
    if (length(words) >= 3) {
        print(words[3])
    }
}
```

3.10.2 파이썬

```
#| label: py-list-email-parsing
fhand = open('mbox-short.txt')

for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '):
        continue
    words = line.split()
    print(words[2])
```

`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 찾아내기 위해서 작업할 라인을 선택하고, 어떻게 해당 라인을 뽑아낼 것이다.

3.11 객체와 값

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.

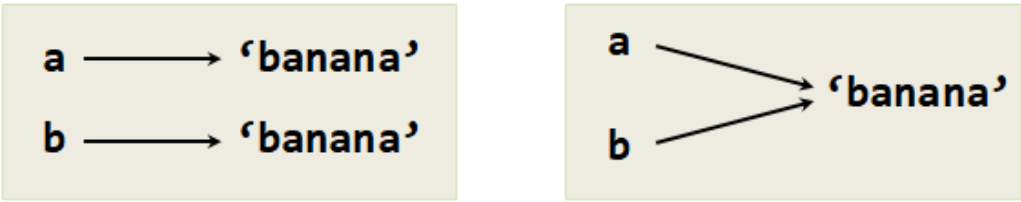

그림 3.1: 문자열 참조

한 가지 경우는 a와 b가 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.

3.11.1 R

```
#¦ label: r-a-b-reference
a <- 'banana'
b <- 'banana'
print(identical(a, b))
#> [1] TRUE
```

3.11.2 파이썬

```
#¦ label: py-a-b-reference
a = 'banana'
b = 'banana'
print(a is b)
#> True
```

이 경우, 파이썬은 하나의 문자열 객체를 생성하고 a와 b 모두 동일한 객체를 참조한다.

하지만, 리스트 두 개를 생성할 때, 객체가 두 개다.

3.11.3 R

```
#¦ label: r-a-b-list
a <- list(1, 2, 3)
```

```
b <- list(1, 2, 3)
print(identical(a, b))
```

3.11.4 파이썬

```
#¦ label: py-a-b-list
a = [1, 2, 3]
b = [1, 2, 3]
print(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 동등하다. 하지만, 동등하다고 해서 반드시 동일하지는 않다.

R과 파이썬이 다른 결과를 출력하는 이유는 무엇일까? R `identical()` 함수는 객체의 값과 구조가 완전히 동일한지를 확인한다. `a <- list(1, 2, 3)`와 `b <- list(1, 2, 3)`의 경우, `a`와 `b`는 별도의 객체이지만, 그 내용(값과 구조)이 완전히 동일하기 때문에 `identical(a, b)`는 `TRUE`를 반환한다. 파이썬 `is` 연산자는 객체가 메모리 상에서 동일한 위치를 가리키는지(즉, 같은 객체인지)를 확인한다. `a = [1, 2, 3]`와 `b = [1, 2, 3]`의 경우, `a`와 `b`는 값은 동일하지만 서로 다른 메모리 위치에 할당된 별개의 리스트 객체라서 `a is b`는 `False`를 반환한다.

지금까지 "객체(object)"와 "값(value)"을 구분 없이 사용했지만, "객체가 값을 가진다."라고 말하는 것이 좀 더 정확하다. `a = [1,2,3]`을 실행하면, `a`는 특별한 순서 요소 값을 갖는 리스트 객체로 참조한다. 만약 다른 리스트가 같은 요소를 가지고 있다면, 그 리스트는 같은 값을 가진다고 말할 수 있다.

3.12 에일리어싱

에일리어싱(별칭 부여, Aliasing) 용어는 두 개 이상의 변수가 동일한 객체를 참조할 때 사용된다. `a`가 객체를 참조하고, `b <- a` 대입한다면, 두 변수는 동일한 객체를 참조한다.

3.12.1 R

```
#¦ label: r-list-alias
a <- list(1, 2, 3)
```

```
b <- a
identical(a, b)
#> [1] TRUE

a <- list(4,5,6)
identical(a, b)
#> [1] FALSE
```

3.12.2 파이썬

```
#¦ label: py-list-alias
a = [1, 2, 3]
b = a
print(a is b)
#> True
a = [4, 5, 6]
print(a is b)
#> False
```

객체와 변수의 연관 짓는 것을 **참조(reference)**라고 한다. 상기의 경우 동일한 객체에 두 개의 참조가 있다.

하나 이상의 참조를 가진 객체는 한 개 이상의 이름을 갖게 되어서, 객체가 **에일리어스(aliased)** 되었다고 한다. 만약 에일리어스된 객체가 변경 가능하면, 변화의 여파는 다른 객체에도 파급된다.

R에는 파이썬 문자열(`string`), 튜플(`tuple`)과 같은 변경 불가능한 자료구조가 없다. 대신 객체 복사본을 생성하여 불변성을 유사하게 구현할 수 있다. `dplyr`과 같은 패키지에서 데이터 조작을 위해 내부적으로 데이터 복사본을 만들어 작업하는 경우가 많다. 다음 예제를 통해 R과 파이썬 차이를 확인할 수 있다.

3.12.3 R

```
#¦ label: r-mutable-alias
a <-  list(1, 2, 3)
b <-  a
b[1] <-  17
print(a)
#> [[1]]
#> [1] 1
#>
```

```
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

3.12.4 파이썬

```
#¦ label: py-immutable-alias
a = [1, 2, 3]
b = a
b[0] = 17
print(a)
#> [17, 2, 3]
```

이와 같은 행동이 유용하기도 하지만, 오류를 발생시키기도 쉽다. 일반적으로, **변경 가능한 객체(mutable object)**로 작업할 때 에일리어싱을 피하는 것이 안전하다.

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

파이썬 문자열 같이 변경 불가능한 객체에 에일리어싱은 그렇게 문제가되지 않는다. 상기 파이썬 예제에서, a와 b가 동일한 문자열을 참조하든 참조하지 않든 거의 차이가 없다.

3.13 디버깅

부주의한 리스트 사용이나 변경 가능한 객체를 사용하는 경우 디버깅에 시간이 오래 걸릴 수 있다. 다음에 일반적인 함정 유형과 회피하는 방법을 소개한다.

1. R에서 리스트 함수가 파이썬과는 다르게 동작한다. R 리스트는 기본적으로 변경 가능(mutable)하지만, 리스트를 변형하는 함수 대부분은 새로운 리스트를 반환하고 원본 리스트는 변경하지 않는다. 파이썬에서 문자열이 작동하는 방식과 유사하지만 차이가 있다. 예를 들어, R에서 `sort` 함수는 `t` 리스트의 원본 내용을 변경하지 않고, 정렬된 결과를 새로운 벡터 `t_sorted`에 저장한다. R에서 리스트나 벡터를 다룰 때는 이러한 특성을 기억하는 것이 중요하다. 또한 R에서는 인터랙티브 환경에서 함수의 동작을 테스트하고, 관련 문서나 도움말(`?function_name`)을 참조하여 함수가 어떻게 작동하는지 먼저 이해할 것을 권장한다. 하지만, 파이썬에서 `word = word.strip()`과 같은 코드를 사용하여 문자열에서 공백을 제거하고, `t =`

t.sort()처럼 리스트를 정렬할 수 있다. 하지만 정렬(sort) 메서드는 None을 반환하기 때문에 주의가 필요하다. 따라서, 리스트 관련 함수, 메서드, 연산자를 사용하기 전에, 문서를 주의 깊게 읽고, 인터랙티브 모드에서 시험하는 것을 권장한다.

3.13.1 R

```
#¦ label: r-sort-debug
t <- list(4, 2, 3)
t_sorted <- sort(unlist(t))
print(as.list(t_sorted))
```

3.13.2 파이썬

```
#¦ label: py-sort-debug
t = [4,2,3]
t = t.sort() # word = word.strip() 방식과 다름
print(t)
#> None
```

1. **관용구를 선택하고 고수하라.**

리스트와 관련된 문제 일부는 리스트를 가지고 할 수 있는 것이 너무 많다는 것이다. 예를 들어, 리스트에서 요소를 제거하기 위해서는 list 객체에서 직접 요소를 제거하는 대신, 새로운 리스트를 생성하거나 벡터로 작업을 대신한다. R에는 파이썬 pop, remove, del에 해당하는 직접적인 함수가 없으며, 대신 리스트의 특정 요소를 제외한 새로운 리스트를 생성한다. 요소를 추가하기 위해서 append 함수를 사용하거나, c() 함수로 리스트 또는 벡터에 요소를 추가한다.

3.13.3 R

```
#¦ label: r-list-append-debug
t <- list(1, 2, 3, 4)
t <- t[-2]  # 두 번째 요소 제거
t <- append(t, 5) # 요소 추가

# append(t, list(x)) # 신규 x 리스트 t에 추가 안됨
t <- append(t, list(x))
# t + list(x) # 신규 x 리스트 t에 추가 안됨
t <- c(t, list(x))
```

3.13.4 파이썬

```
#¦ label: py-list-append-debug
t = [4,2,3]
x = 1
t.append(x)
t = t + [x]

t.append([x])     # 틀림(WRONG)!
t = t.append(x)   # 틀림(WRONG)!
t + [x]           # 틀림(WRONG)!
t = t + x         # 틀림(WRONG)!
```

2. **에일리어싱을 회피하기 위해 사본 만들기.**

R에서 리스트나 벡터를 정렬하면서 원본 데이터를 보존하고 싶은 경우, 정렬 함수가 원본 객체를 변경하지 않고 새로운 객체를 반환하기 때문에 별도의 사본을 만들 필요가 없다. 하지만, 데이터 분석에서처럼 원본 데이터를 보존하여 만일의 사태에 대비하는 것이 좋다. 파이썬에서는 원본 리스트를 유지하면서, 변경을 가하는 `sort`와 같은 메서드를 사용하고자 한다면, 사본을 만들어야 한다.

3.13.5 R

```
#¦ label: r-list-debug-aliasing
t <- list(3, 1, 5, 2)
orig <- t  # 원본 리스트 복사
t <- sort(unlist(t))
```

3.13.6 파이썬

```
#¦ label: py-list-debug-aliasing
orig = t[:]
t.sort()
```

상기 예제에서 원 리스트는 그대로 둔 상태로 새로 정렬된 리스트를 반환된 결과는 `t`에 저장한다. 하지만 이 경우에는, 변수명으로 `sorted`를 사용하는 것을 피해야 한다!

3. **리스트, 분할(split), 파일**

파일을 읽고 파싱할 때, 프로그램이 중단될 수 있는 입력값을 마주할 수많은 기회가 있

다. 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 관심 있는 단어가 있는지 살펴보기 위해서 단순하게 각 라인의 첫 단어를 살펴본다. 다음과 같이 continue 문을 사용해서 “From”이 없는 라인을 건너뛴다.

3.13.7 R

```
#¦ label: r-list-email-parsing-ex
download.file("https://www.dr-chuck.com/py4inf/code/mbox-short.txt",
              destfile = "mbox-short.txt")

fhand <- readLines('mbox-short.txt')

for (line in fhand) {
    line <- trimws(line)  # 공백 제거
    if (length(grep("^From ", line)) == 0) {
        next
    }
    words <- unlist(strsplit(line, " "))
    if (length(words) >= 3) {
        print(words[3])
    }
}
```

3.13.8 파이썬

```
#¦ label: py-list-email-parsing-ex
fhand = open('mbox-short.txt')

for line in fhand:
    words = line.split()
    if words[0] != 'From':
        continue
    print(words[2])
```

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 첫 줄에 Sat를 출력하고 나서 오류로 프로그램이 정상 동작에 실패하는 경우도 있다. 무엇이 잘못되었을까? 어딘가 엉망이 된 데이터가 있어 우아하고, 총명하며, 매우 R스러운 프로그램을 망가뜨린 건가?

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.

3.13.9 R

```
#¦ label: r-list-email-parsing-ex-print
fhand <- readLines('mbox-short.txt')

for (line in fhand) {
    line <- trimws(line)  # 공백 제거
    message("Debug: ", line)
    if(!stringr::str_detect(line, "^From ")) next
    words <- unlist(strsplit(line, " "))
    print(words[3])
}
```

3.13.10 파이썬

```
#¦ label: py-list-email-parsing-ex-print
fhand = open('mbox-short.txt')

for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From':
        continue
    print(words[2])
```

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.

R

```
Debug: 'X-DSPAM-Confidence:', '0.8475'
Debug: 'X-DSPAM-Probability:', '0.0000'
Debug:
```

파이썬

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "D:/tcs/gpt-coding/data/mbox_debug.py", line 5, in <module>
    if words[0] != 'From':
IndexError: list index out of range
```

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 장소이다. 코드를 보호하는 방법은 많다. 첫 단어를 살펴보기 전에 단어의 개수를 확인하는 방법을 택한다.

3.13.11 R

```
#¦ label: r-list-email-parsing-ex-guardian
fhand <- readLines('mbox-short.txt')
```

```
for (line in fhand) {
    line <- trimws(line)  # 공백 제거
    # message("Debug: ", line)
    if(length(words) == 0) next
    if(length(grep("^From ", line)) == 0) next

    words <- unlist(strsplit(line, " "))
    print(words[3])
}
```

3.13.12 파이썬

```
#¦ label: py-list-email-parsing-ex-guardian
fhand = open('mbox-short.txt')

for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From':
        continue
    print(words[2])
```

{r

변경한 코드가 실패해서 다시 디버그할 경우를 대비해서, print문을 제거하는 대신에 print문을 주석 처리한다. 그리고 나서, 단어가 '0' 인지를 살펴보고 만약 '0' 이면, 파일 다음 라인으로 건너뛰도록 next문을 사용하는 가디언 문장(guardian statement)을 추가한다.

두 개의 next문이 "흥미롭고" 좀 더 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 정상 작동할 것이라는 것을 확인해 주지만, 충분하지 않을 수도 있다. 프로그램을 작성할 때, "무엇이 잘못될 수 있을까?"를 항상 생각해야 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 파일

을 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.

3.14 용어 정의

- **에일리어싱(aliasing)**: 하나 혹은 그 이상의 변수가 동일한 객체를 참조하는 상황.
- **구분자(delimiter)**: 문자열이 어디서 분할되어야 할지를 표기하기 위해서 사용되는 문자나 문자열.
- **요소(element)**: 리스트 혹은 다른 시퀀스 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)**: 시퀀스 값.
- **리스트 순회(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.

연습문제

1. http://www.py4inf.com/code/romeo.txt에서 파일 사본을 다운로드 받는다. `romeo.txt` 파일을 열어, 한 줄씩 읽어들이는 프로그램을 작성하세요. 각 라인마다 stringr 팩키지에서 분할 `str_split()` 함수를 사용하여 라인을 단어 리스트로 쪼갠다.

각 단어마다, 단어가 이미 리스트에 존재하는지를 확인하세요. 만약 단어가 리스트에 없다면, 리스트에 새 단어로 추가한다.

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 작성한다. "From"으로 시작하는 라인을 발견했을 때, stringr 팩키지에서 분할 `str_split()` 함수를 사용하여 라인을 단어로 쪼갠다. "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 두 번째 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 작성한다. 사용자가 입력한 숫자를 리스트에 저장하고, `max()` 과 `min()` 함수를 사용하여 루프가 끝나면, 최대값과 최소값을 출력하는 프로그램을 작성한다.

```
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
```

제 4 장

딕셔너리

4.1 명칭을 갖는 리스트

명칭을 갖는 리스트(named list)는 딕셔너리로 더 잘 알려져 있고, **딕셔너리(dictionary)**는 리스트와 유사하지만 좀 더 일반적이다. 리스트에서 위치(인덱스)가 정수이지만, 딕셔너리에서 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 한국어 단어로 매핑되는 사전을 만들 것이다. 키와 값은 모두 문자열이다.

`list` 함수는 항목이 전혀 없는 리스트를 신규로 생성한다. `list`는 내장함수명이기 때문에 변수명으로 사용하는 것을 피해야 한다.

4.1.1 R

```
#| label: r-named-list
eng2kr <- list()
eng2kr
#> list()
```

4.1.2 파이썬

```
#¦ label: py-named-list
eng2kr = dict()
print(eng2kr)
#> {}
```

`list()`는 빈 리스트임을 나타낸다. 리스트에 신규 요소를 추가하기 위해서 `list()` 함수 내부에 `'명칭'='값'`과 같이 명칭과 값을 지정한다. 상기 코드는 키(명칭) `'one'`에서 값 `'하나'`를 매핑하는 항목을 생성한다. 명칭을 갖는 리스트를 다시 출력하면, 키-값 페어(key-value pair)를 볼 수 있다.

4.1.3 R

```
#¦ label: r-named-list-one
eng2kr <- list('one' = '하나')
eng2kr
#> $one
#> [1] "하나"
```

4.1.4 파이썬

```
#¦ label: py-named-list-one
eng2kr['one'] = '하나'
print(eng2kr)
#> {'one': '하나'}
```

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`을 출력하면 다음과 같다.

4.1.5 R

```
#¦ label: r-named-list-many
eng2kr <- list('one' = '하나',
               'two' = '둘',
               'three' = '셋')
```

```
eng2kr
#> $one
#> [1] "하나"
#>
#> $two
#> [1] "둘"
#>
#> $three
#> [1] "셋"
```

4.1.6 파이썬

```
#| label: py-named-list-many
eng2kr = {'one': '하나', 'two': '둘', 'three': '셋'}
print(eng2kr)
#> {'one': '하나', 'two': '둘', 'three': '셋'}
```

> ⚠ 파이썬 딕셔너리
>
> 파이썬 3.7 버전 이전에는 키-값 페어(key-value pair)의 순서가 같지 않다. 사실 동일한 사례를 여러분의 컴퓨터에서 입력하면, 다른 결과를 얻게 된다. 일반적으로, 딕셔너리 항목 순서는 예측 가능하지 않았다. 파이썬 3.7 버전 이후부터는 딕셔너리 항목 순서가 입력한 순서대로 유지된다. 딕셔너리 요소가 정수 인덱스로 색인되지 않아서 문제되지는 않는다. 대신에, 키를 사용해서 항상 상응하는 값을 찾을 수 있다. R 네임드 리스트(named list)는 항상 정의한 순서대로 요소를 유지한다. 리스트에 요소를 추가하면 추가된 순서대로 요소가 유지되며, 이 순서는 변경되지 않는다.

명칭을 갖는 리스트에서 키를 사용해서 상응하는 값을 찾을 수 있다.

4.1.7 R

```
#| label: r-named-list-extraction
eng2kr['one']
#> $one
#> [1] "하나"
```