재귀와 반복 - 검색

Recursion & Iteration - Searching

검색searching 문제는 순서열과 같은 데이터구조에 모여 있는 데이터 중에서 특정데이터(키key 라고 함)를 찾는 문제이다. Python으로 코딩하는 경우, 표준 라이브러리에서 제공하는 순서열 공통 메소드인 `index`를 사용하면 손쉽게 순서열 검색을 할 수 있다. `index` 메소드를 아래 표에서와 같이 호출하여 문자열 전체 또는 지정한 검색 범위 내에서 가장 앞에 있는 키의 위치번호를 알아낼 수 있다.

연산	의미
`s.index(x)`	s에서 가장 앞에 있는 키 x의 위치번호
`s.index(x,i)`	s의 i 위치에서 시작하여 가장 앞에 있는 키 x의 위치번호
`s.index(x,i,j)`	s의 i 위치와 j 위치 범위 내에서 가장 앞에 있는 키 x의 위치번호 (i 위치는 검색범위에 포함하고, j 위치는 검색범위에 포함하지 않음)

이 검색 메소드를 본격적으로 테스트해보려면 어느 정도 규모가 있는 검색 대상이 필요하다. 그러니 테스트 용으로 무작위 값으로 구성된 검색 대상 리스트를 만들어보자. 표준 라이브러리의 `random` 모듈에서 제공하는 다음 메소드를 써서 주어진 시퀀스에서 원하는 만큼 중복없이 무작위로 골라서 리스트를 만들 수 있다.

연산	의미
`random.sample(population,k)`	population 시퀀스에서 중복없이 k개를 무작위로 골라 리스트로 모아서 내준다.

예를 들어, 0에서 9,999사이 10,000개의 정수 중에서 무작위로 1,000개를 중복없이 샘플링하여 모아 리스트로 만드는 표현식은 다음과 같이 표현한다.

```
random.sample(range(10000),1000)
```

python3 실행기에서 리스트가 주문한대로 만들어지는지 실제로 확인해보자.

그럼 이제 검색 대상을 만들 수 있으니 하나 만들어 `index` 메소드로 검색이 잘 되는지 다음 코드를 실행기에서 실행하여 확인해보록 하자.

```
>>> import random
>>> db = random.sample(range(10000),1000)
>>> key = db[109]
>>> print(db.index(key))
```

무작위로 생성된 1,000개 값으로 구성된 리스트인 db의 위치번호 109에 해당하는 값을 key로 하여 검색하니 바로 109가 프린트 됨을 확인할 수 있을 것이다. 만약 리스트에 없는 데이터를 검색하면 어떻게 될까? 다음과 같이 `ValueError`가 발생한다.

```
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print(data.index(9999))
ValueError: 9999 is not in list
```

검색 오류가 발생하는 상황을 만들어 실행기로 오류메시지가 발생하는지 확인해보자. 오류 발생을 미연에 방지하기 위해서는 검색을 시도하기 전에 값이 있는지 다음과 같이 확인하여 있는 경우에만 index 메소드를 호출하게하면 프로그램이 비정상적으로 종료하는 경우가 발생하지 않는다.

```
if key in data:
    data.index(key)
```

오류메시지가 발생한 같은 키로 위와 같이 수정하면 발생하지 않음을 실행기로 직접 확인해보자.

이와 같이 Python에서는 표준 라이브러리 메소드로 간단히 검색할 수 있지만, 검색 문제는 컴퓨터과학에서 가장 중요한 문제 중의 하나이니 검색 알고리즘도 공부할겸 프로그램 짜는 훈련도 할겸 검색 함수를 직접 만들어보자.

1. 리스트 검색

1.1. 검색 : OX문제

"키 key가 리스트 s에 있는가?"라는 질문에 대답할 수 있는 함수를 만들어보자. 이 함수의 입출력 요구사항은 다음과 같다.

- 입력 : 리스트 s와 키 key
- 출력 : key가 s에 있으면 True, 없으면 False

순차검색 sequential search

순차검색 알고리즘은 앞에서 부터 차례로 하나씩 검색하여 찾는 방법이다. 이 알고리즘도 리스트의 구조를 기반으로 반복조건과 종료조건으로 나누어 다음과 같이 설계할 수 있다.

s에서 key를 찾으려면	
(반복조건) s != []	• s의 선두원소 s[0]가 key와 같으면, 찾았으므로 True를 내줌 • 그렇지 않으면, s의 후미리스트 **s[1:]에서 key를 재귀로 찾음**
(종료조건) s == []	• 검색 대상이 없으므로 False

이 알고리즘을 Python 코드로 작성하면 다음과 같다.

```
1  def seq_search_ox(s,key):
2      if s != []:
3          if s[0] == key:
4              return True
5          else:
6              return seq_search_ox(s[1:],key)
7      else:
8          return False
```

이 프로그램을 두 사례를 가지고 다음과 같이 실행 추적하여 이 함수의 실행의미를 이해해보자.

```
seq_search_ox([3,5,4,2],4)
=> seq_search_ox([5,4,2],4)
=> seq_search_ox([4,2],4)
=> True
```

```
seq_search_ox([3,5,4],6)
=> seq_search_ox([5,4],6)
=> seq_search_ox([4],6)
=> seq_search_ox([],6)
=> False
```

위 재귀함수는 꼬리재귀이므로 다음과 같이 while 문으로 바로 재작성하여 마무리할 수 있다.

```
1  def seq_search_ox(s,key):
2      while s != []:
3          if s[0] == key:
4              return True
5          else:
6              s = s[1:]
7      return False
```

그런데 s는 순서열이므로 for 문을 사용하여 다음과 같이 작성할 수도 있다.

```
1  def seq_search_ox(s,key):
2      for x in s:
3          if x == key:
4              return True
5      return False
```

순차검색 함수의 경우 마지막으로 작성한 for 문 버전이 가장 간결하고 읽기 쉬워보인다. 재귀와 반복의 개념을 제대로 이해하기 위해서 다양한 형태의 재귀와 반복을 공부해야 했지만, 경험이 쌓여 숙달이 되면 위와 같은 방식으로 바로 코딩이 가능해져야 한다. 현대 프로그래밍 언어에서는 이와 같이 반복문을 재귀없이 간결하게 작성할 수 있도록 효율적인 for 문을 제공하고 있다.

이분검색 binary search

순차검색 알고리즘으로 검색하는 경우, 키를 찾는 순간 바로 검색을 종료할 수 있다. 하지만 키가 맨 뒤에 있거나, 아예 없는 경우 리스트 전체를 다 검사해야 최종 판단을 할 수 있다. 그런데 리스트를 정렬해 놓으면, 리스트 전체를 다 검사하지 않고도 키의 존재유무를 판단할 수 있는 방안이 있다. 정렬된 리스트를 반으로 나누어, 가운데 원소를 키와 비교하여 그 결과에 따라 바로 검색을 끝내거나 좌우 반쪽 중

하나만 검색하면 되기 때문이다. 리스트의 구조를 기반으로 반복조건과 종료조건으로 나누어 알고리즘을 기술하면 다음과 같다.

정렬된 ss에서 key를 찾으려면	
(반복조건) ss != []	• ss의 정 가운데 원소의 위치번호를 mid라고 함 • key가 ss[mid]와 같으면, 찾았으므로 True를 내줌 • key가 ss[mid]보다 작으면, **ss[:mid]에서 key를 재귀로 찾음** • 그렇지 않으면, **ss[mid+1:]에서 key를 재귀로 찾음**
(종료조건) ss == []	• 검색 대상이 없으므로 False

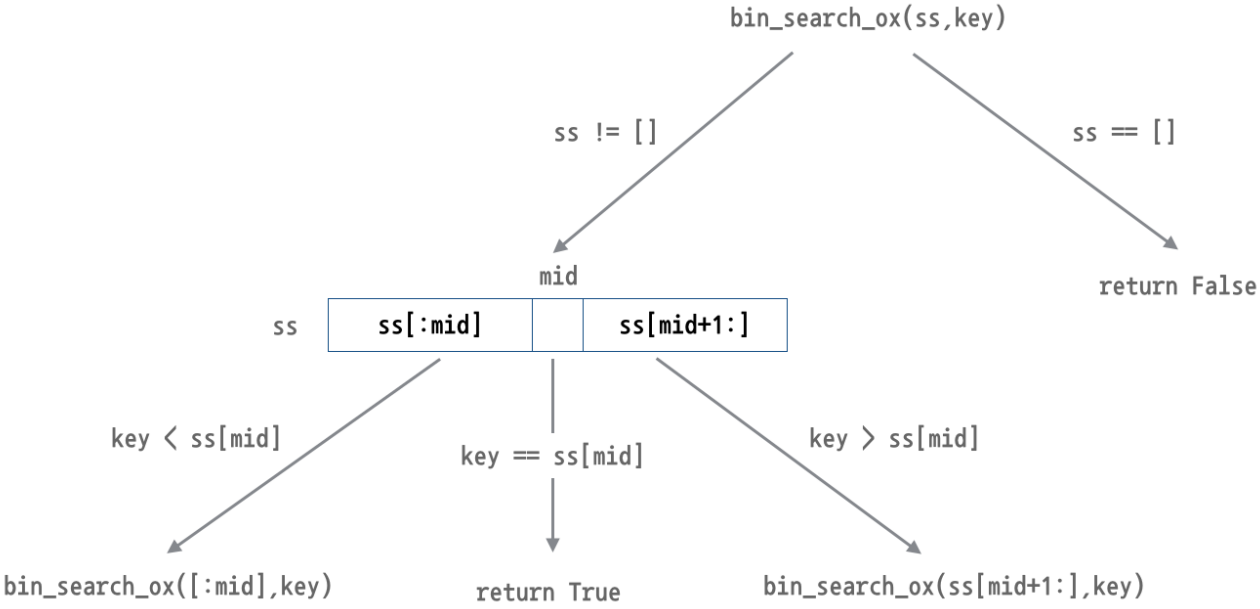
정렬된 리스트 ss를 반으로 나누어 가운데 값을 검사하여 key와 같으면 찾았으므로 검색을 끝내고, key 보다 작으면 오른쪽에는 절대 없을 것이므로 검색 대상에서 제외하고 왼쪽 반쪽만 검사하고, key 보다 크면 왼쪽에는 절대 없을 것이므로 검색 대상에서 제외하고 오른쪽 반쪽만 검사하면 충분하다. 즉, 비교를 한 번 할 때마다 검색 범위가 반으로 줄어든다. 어느 시점에서 key와 같은 값을 찾으면 검색을 바로 종료해야겠지만, 더 이상 검색할 원소가 없으면 key는 리스트 ss에 없음이 확실하다. 이와 같이 정렬된 리스트를 가운데 원소를 찾아 반으로 나우어 검색하는 방법을 **이분검색**binary search이라고 한다.

코딩을 하기 전에 이해를 돕기 위하여 이 알고리즘을 그림으로 표현하고, 이를 Python 재귀함수로 코딩하면 다음과 같다.

```
1  def bin_search_ox(ss,key):
2      if ss != []:
3          mid = len(ss) // 2
4          if key == ss[mid]:
5              return True
6          elif key < ss[mid]:
7              return bin_search_ox(ss[:mid],key)
8          else:
9              return bin_search_ox(ss[mid+1:],key)
10     else:
11         return False
```

이 재귀함수도 꼬리재귀이므로 다음과 같이 while 문으로 바로 재작성하여 마무리할 수 있다.

```
1  def bin_search_ox(ss,key):
2      while ss != []:
3          mid = len(ss) // 2
4          if key == ss[mid]:
5              return True
6          elif key < ss[mid]:
7              ss = ss[:mid]
8          else:
9              ss = ss[mid+1:]
10     return False
```



성능비교

이분검색이 선형검색과 비교하여 얼마나 빨리 검색하는지 알아둘 필요가 있다. key와 비교 횟수를 기준으로 비교해보면 다음 표와 같이 검색 대상 리스트의 크기가 아무리 커져도 선형검색과는 달리 이분검색의 비교회수는 그리 많아지지 않음을 관찰할 수 있다. 사실 이 두 검색 방법의 비교횟수 차이는 엄청나다. 데이터를 정렬해두면 좋은 이유가 바로 여기에 있다.

리스트 길이	선형검색의 비교횟수	이분검색의 비교횟수
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

1.2. 검색 : 찾은 위치를 알려주기

키가 단순히 있다는 사실 뿐만 아니라 어디에 있는 알려주도록 검색문제를 개선해보자.
"리스트 s에 키 key가 처음 나타나는 위치번호는?"라는 질문에 대답할 수 있는 함수를 만들어보자. 이는 True 또는 False 로 대답하는 OX문제와는 달리 주관식 문제이다. 이 검색문제 입출력 요구사항은 다음과 같다.

- 입력 : 리스트 s와 키 key
- 출력 : key가 s에서 처음 나타나는 위치번호, key가 s에 없으면 None

순차검색 sequential search

순차검색 알고리즘은 앞에서 완성한 다음 버전을 찾은 원소의 위치번호를 내줄 수 있도록 보완해보자.

```
1  def seq_search_ox(s,key):
2      for x in s:
3          if x == key:
4              return True
5      return False
```

for 반복문은 순서열 s를 앞에서부터 하나씩 차례로 검사한다. 따라서 검사하는 원소의 위치번호를 기억하는 변수(i)를 다음과 같이 하나 만들어 두고 키를 찾았을 때 해당 위치번호로 쓸 수 있도록 한다.

```
1  def seq_search(s,key):
2      i = 0
3      for x in s:
4          if x == key:
5              return i
6          i += 1
7      return None
```

키가 리스트에 없는 경우에는 None 값을 내준다.

연습문제 1

0부터 9,999 사이의 자연수 10,000개 중에서 1,000개를 중복없이 무작위로 뽑아서 나열한 리스트를 만들고, 같은 범위 내의 임의의 수를 무작위로 뽑아서 그 리스트에 있는지 확인하게 하여, 위의 순차검색 함수 seq_search를 테스트하는 함수를 다음과 같이 작성하였다.

```
1  def test_seq_search():
2      print("Sequential search test")
3      db = random.sample(range(10000),1000) # 무작위 리스트 생성
4      for i in range(10):
5          key = random.randrange(10000) # 키 무작위 생성
6          index = seq_search(db,key)
7          print(key,"found at",index)
```

이 코드에서 사용하는 random 모듈의 randrange 메소드의 의미는 다음과 같다.

연산	의미
random.randrange(n)	range(n) 에 있는 정수범위 내에서 무작위로 정수 하나 골라서 내준다.

예를 들어, 0에서 9,999 사이의 10,000개의 정수 중에서 무작위로 하나를 뽑으려면, 다음과 같이 호출하면 된다.

```
random.randrange(10000)
```

이 코드를 실행하여 seq_search 함수가 잘 작동하는지 확인해보자. 직접 테스트해보면 느끼겠지만 10,000개의 수에서 무작위로 뽑은 1,000개의 수 중에서 찾으니 찾을 확률이 대략 1/10이라 찾는 수가 없는 경우가 훨씬 많다. 그래서 대안으로, 키가 없는 경우에 키 값과 크기가 가장 비슷한 수를 찾아달라는 요청을 받았다고 하자.

찾으려는 key가 리스트 s에 없더라도 key에 가장 가까운 수를 찾아서 그 위치번호를 내주도록 함수 seq_search를 확장하여 seq_search_closest 함수를 아래 형식으로 작성하자.

```
1  def linear_search_closest(s,key):
..
..
```

특정 수 x가 key와 얼마나 가까운지를 알기 위해서는 두 수 사이의 거리를 구하면 된다. 두 수 사이의 거리는 한 수에서 다른 수를 뺀 절대값이다. 절대값을 구하는 함수는 표준라이브러리에서 제공하고 있으며 abs이다. 즉, key와 x의 거리는 abs(key-x)로 구할 수 있다. 이 경우 유일하게 None 값이 나오는 경우는 s가 비어있는 경우뿐임을 유의하여 코딩하자. 검색하는 동안 항상 그 시점에서 가장 가까운 수의 거리와 위치를 기억하고 있어야 하고, 더 가까운 수를 찾은 경우 거리와 위치를 갱신해야 한다. 거리의 초기값은 다음과 같이 최대값으로 설정하도록 하자.

abs(key-max(s))

완성이 되면, 다음 테스트 함수를 호출하여 잘 작동하는지 확인해보자.

```
1  def test_seq_search_closest():
2      print("Sequential search test")
3      db = random.sample(range(10000),1000)
4      for i in range(10):
5          key = random.randrange(10000)
6          index = seq_search_closest(db,key)
7          print("The closest value to",key,":",db[index],"at index",index)
8      key = random.randrange(10000)
9      index = seq_search_closest([],key)
10     print(key,"found at",index) # None
```

이분검색 binary search

앞서 작성한 아래의 binary_search_ox 함수는 찾았는지 여부만 알려주지 위치번호를 알려주지는 않는다.

```
1  def bin_search_ox(ss,key):
2      while ss != []:
3          mid = len(ss) // 2
4          if key == ss[mid]:
5              return True
6          elif key < ss[mid]:
7              ss = ss[:mid]
8          else:
9              ss = ss[mid+1:]
10     return False
```

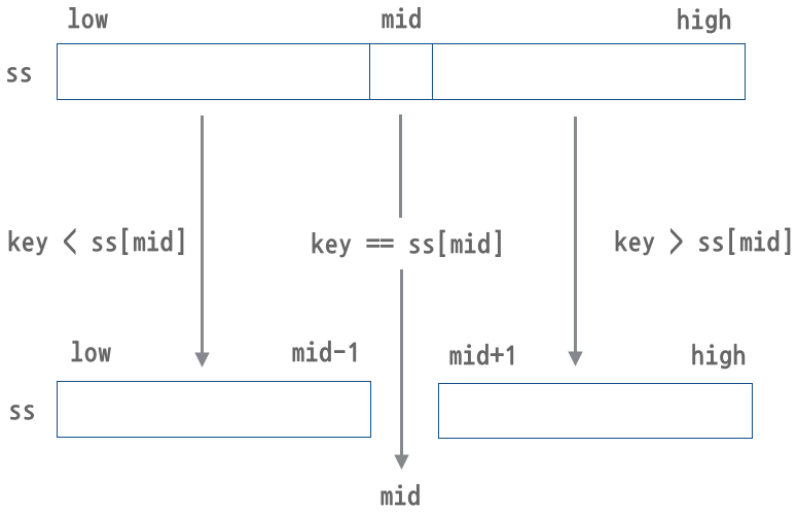
이 함수를 위치번호를 내주는 함수로 수정한 bin_search 함수를 작성해보자.

위치번호를 내주어야 하는데 위 코드에서와 같이 리스트를 잘라버리면 검색대상 리스트의 오리지날 위치번호를 잃어버린다. 따라서 리스트를 자르는 대신 검색범위의 시작과 끝을 알려주는 위치번호를 각각 low와 high 변수에 기억하여 검색범위를 좁혀가며 검색을 하면 위치번호를 그대로 유지하며 검색할 수 있다. 초기에 검색대상 리스트인 다음 그림과 같은 ss의 시작과 끝의 위치번호 low와 high는 각각 다음과 같이 놓을 수 있다.

```
low = 0
high = len(ss) - 1
```

그리고 가운데 위치번호인 mid는 다음 식으로 구할 수 있다.

```
mid = (low + high) // 2
```



반복조건은 검색할 대상이 하나라도 있는 경우인 ss != [] 였는데, 이제는 ss를 자르는 대신 시작과 끝 위치번호로 검색범위를 나타내므로 검색을 계속해야 할 반복조건은 다음과 같다.

```
low <= high
```

이 조건을 만족하는 한 최소한 하나의 원소는 검색 범위 내에 있게 된다. low > high 가 되면 검색할 대상도 없고 키를 찾지도 못했음이 확실하다. 이 전략으로 함수를 완성하면 다음과 같다.

```
1  def bin_search(ss,key):
2      low = 0
3      high = len(ss) - 1
4      while low <= high:
5          mid = (high + low) // 2
6          if key == ss[mid]:
7              return mid
8          elif key < ss[mid]:
9              high = mid - 1
10         else:
11             low = mid + 1
12     return None
```

이 함수를 이해하고 다음의 테스트 함수로 잘 작동하는지 확인해보자.

```
def test_bin_search():
    print("Binary search test")
    db = random.sample(range(10000),1000)
    db.sort()
    for i in range(10):
        key = random.randrange(10000)
        index = bin_search(db,key)
        print(key,"found at",index)
```

연습문제 2 ———————————————————————————————————————

이분검색 함수 bin_search도 연습문제 1에서 작성한 검색함수와 같이 찾으려는 key가 리스트 ss에 없더라도 key에 가장 가까운 수를 찾아서 그 위치번호를 내주도록 bin_search를 확장하여 bin_search_closest 함수를 작성하자. 테스드 함수도 유사한 방식으로 작성하여 잘 작동하는지 확인해 보자.

———-

2. 문자열 검색

2.1. 파일 입출력

표준입출력으로는 키보드와 실행창을 통한 상호작용으로 사용자와 입력-계산-출력을 하여 결과를 바로 보여주지만 계산 결과는 프로그램이 끝나면 사라진다. 오래 보존해야 하는 정보는 **파일**file을 매개체로 하여 영구히 저장하고 필요한 경우 재사용할 수 있다.

텍스트 파일textfile이란 키보드로 입력가능한 ASCII 문자로만 구성된 파일이다. 텍스트 파일은 ASCII 문자로만 구성되어 있기 때문에 특정 플랫폼을 가리지 않고 어디에서나 사용할 수 있고 편집기로 쉽게 작성할 수 있어서 정보를 영구히 저장해두고 필요시 재사용해야 하는 경우 요긴하게 쓸 수 있다. 이 절에서는 텍스트 파일을 사용하여 정보를 읽고 쓰는 방법에 대해서 알아본다.

파일 열기와 닫기

읽거나 쓰려면 먼저 파일을 열어야 한다. 파일은 open 함수로 여는데, 대상 파일 이름(필요에 따라 경로 포함)과 용도인 **접근 모드**access mode를 인수로 전달해주어야 한다. 예를 들어, 현재 디렉토리의 input.txt 파일을 읽을 용도로 열고 싶으면 다음과 같이 한다.

```
t = open("input.txt", "r")
```

여기서 두번째 인수인 "r"은 접근모드를 나타내며, 파일을 여는 용도가 **읽기** 위함임을 알려주기 위한 것이다. 어떤 용도로 파일을 여는지 알려주는 접근 모드를 중요한 것 몇개만 나열해보면 다음과 같다.

위에서 open 함수로 연 input.txt 파일은 변수 t로 지정하였으므로, 이 변수로 파일을 읽을 수 있다.

일단 연 파일은 사용 후 다음과 같은 형식으로 닫아야 한다. (연 파일은 반드시 닫는 습관 필수)

```
t.close()
```

파일을 열었으면 이제 읽을 수 있다.

텍스트 파일에서 파일 단위로 문자열 읽기

input.txt 파일에 다음과 같은 내용이 저장되어 있다고 하자.

```
새 나라의 대학생은 일찍 일어납니다
잠꾸러기 없는 나라
우리나라 좋은 나라
```

파일에서 n개의 문자를 읽어오려면 read(n) 메소드를 호출하면 된다. 예를 들어, 파일을 연 다음 한 문자만 읽어서 프린트하고 싶으면 다음과 같이 한다.

```
t = open("input.txt", "r")
print(t.read(1))
```

'새'를 프린트할 것이다. 그 다음 9자를 더 읽어서 프린트 하고 싶으면 계속해서 다음과 같이 한다.

```
print(t.read(9))
```

' 나라의 대학생은'을 프린트할 것이다. 읽을 때 어디까지 읽었는지 기억해 두고, 다음에 읽을 때 전에 기억해 둔 곳에서 읽기 시작한다. 파일 끝에 도달하면 그 이후에 읽은 결과는 무조건 빈 문자열이 된다. 처음부터 다시 읽고 싶으면 다음과 같이 해당 파일을 닫고 다시 열어야 한다.

```
t.close()
t = open("input.txt", "r")
```

읽을 문자의 개수를 명시하지 않으면 남은 파일 전체를 끝까지 다 읽어버린다.

```
print(t.read())
```

그러면 파일 t에서 읽어온 문자열 '새 나라의 대학생은 일찍 일어납니다.\n잠꾸러기 없는 나라\n우리나라 좋은 나라'를 실행창에 프린트할 것이다. 그러면 다음과 같이 보일 것이다.

```
새 나라의 대학생은 일찍 일어납니다
잠꾸러기 없는 나라
우리나라 좋은 나라
```

연 파일은 반드시 닫아야 한다. 닫지 않고 두면 다른 프로그램이 파일을 보거나 고칠 수 있으므로 연 파일을 반드시 닫는 습관은 절대 잊지 말아야 할 중요한 코딩습관이다.

```
t.close()
```

텍스트 파일에서 줄 단위로 문자열 읽기

파일에서 줄 단위로 n개의 문자를 읽어오려면 readline(n) 메소드를 쓰면 된다.

```
t = open("input.txt", "r")
print(t.readline(1))
print(t.readline(9))
t.close()
```

이 코드를 실행하면 다음과 같이 실행창에 프린트한다.

```
새
 나라의 대학생은
```

이 사례만 보면 read(n)과 차이가 없어 보인다. 그러나 차이가 있다. read(n)은 파일 전체에서 문자 n개를 읽어오는 반면, readline(n)은 현재 줄에서만 문자 n개를 읽어온다.

readline()은 파일을 한 줄씩 읽을 때 편리하게 사용할 수 있다. 예를 들어, input.txt 파일을 한 줄씩 읽어서 프린트하려면 다음과 같이 하면 된다.

```
t = open("input.txt", "r")
print(t.readline())
print(t.readline())
print(t.readline())
t.close()
```

이 코드를 실행하면 다음과 같이 실행창에 프린트 한다.

```
새 나라의 대학생은 일찍 일어납니다

잠꾸러기 없는 나라

우리나라 좋은 나라
```

여기서 가운데 빈칸이 한 줄씩 생겼을까? input.txt 파일의 각 줄 끝 부분에 보이지는 않지만 줄바꿈문자("\n")가 있다. 따라서 줄바꿈 문자를 프린트하니 한 줄을 띄게되고, print 명령을 실행하니 또 한 줄을 띄게 된 것이다.

텍스트 파일에 문자열 쓰기

텍스트 파일에 쓸때도 쓰기 전에 먼저 해당 파일을 열어야 한다. output.txt 파일에 문자열을 쓰고 싶으면 다음과 같이 "w" 모드로 파일을 열어야 한다.

```
t = open("output.txt", "w")
```

그러면 내용이 비어있는 output.txt 파일이 새로 생겨나며 파일에 문자열을 써주기를 기다리게 된다. 만약 같은 이름의 파일이 이미 있다면, 그 파일은 지워지고 내용이 '빈' 새로운 파일로 대체된다.

파일에 문자열을 쓰려면 write() 메소드를 다음과 같이 호출한다.

```
t.write("새 나라의 대학생은 일찍 일어납니다")
```

그런데 write() 메소드는 쓰는 문자열의 맨끝에 줄바꿈 문자를 넣지 않는다. 따라서 줄을 바꾸고 싶으면 원하는 위치에 "\n" 를 명시해야 한다.

앞 절에서 본 input.txt 파일과 똑같은 내용이 든 output.txt를 다음과 같이 만들 수 있다.

```
t = open("output.txt", "w")
t.write("새 나라의 대학생은 일찍 일어납니다\n")
t.write("잠꾸러기 없는 나라\n")
t.write("우리나라 좋은 나라")
t.close()
```

또는 다음과 같이 해도 똑같은 결과를 얻는다.

```
t = open("output.txt", "w")
t.write("새 나라의 대학생은 일찍 일어납니다\n잠
꾸러기 없는 나라\n우리나라 좋은 나라")
t.close()
```

파일 메소드 요약

메소드	의미
close()	파일을 닫는다. 일단 닫힌 파일은 다시 열기 전에는 읽거나 쓸 수 없다.
read(n)	파일에서 문자 n개를 읽어서 문자열로 내준다.
read()	파일의 현재 위치에서 그 파일의 맨끝까지 문자를 모두 내준다.
readline(n)	파일의 현재 위치에서 그 줄의 문자 n개를 읽어서 문자열로 내준다.
readline()	파일의 현재 위치에서 그 줄의 맨끝까지 문자를 모두 내준다.
readlines()	파일을 줄 별로 모두 읽어서 줄의 리스트로 내준다.
write(s)	문자열 s를 파일에 쓴다.
writelines(ss)	문자열 리스트 ss에 있는 문자열을 모두 파일에 쓴다.

텍스트 파일을 읽고 문자열 메소드를 사용하여 계산한 뒤 결과를 텍스트 파일에 쓰는 프로그램을 만드는 훈련을 해보자.

2.2. 텍스트 파일에서 문자열 검색

텍스트 파일은 하나의 문자열이다. Python 표준라이브러리의 문자열 메소드를 활용하여 텍스트 파일에서 특정 문자열을 검색하는 방법을 실습을 통하여 공부하고 외부파일에 데이터를 기록하고 읽어오는 방법을 알아보자.

문자열 메소드

다음은 문자열 검색에 유용하게 사용할 수 있는 연산자를 일부 모아놓은 것이다. 하나씩 의미를 이해해 보자. 여기서 `str, sub, prefix, suffix`는 문자열을 나타낸다.

연산	의미
str.find(sub)	str에서 맨 앞에 나오는 sub의 위치번호를 내줌, 없으면 -1을 내줌
str.index(sub)	str에서 맨 앞에 나오는 sub의 위치번호를 내줌, 없으면 ValueError 오류
str.rfind(sub)	str에서 맨 뒤에 나오는 sub의 위치번호를 내줌, 없으면 -1을 내줌

연산	의미
str.startswith(prefix)	str이 prefix로 시작하면 True 를, 그렇지 않으면 False 를 내줌
str.endswith(suffix)	str이 suffix로 끝나면 True 를, 그렇지 않으면 False 를 내줌

여기서 find는 문자열 전용이고 index는 모든 순서열에 공통으로 사용할 수 있다. sub가 str에 있는 경우에는 정확히 같은 결과를 내주지만, sub가 str에 없으면 결과가 달라짐을 유의하자.

다음 사례를 실행하여 연산의 의미를 정확히 파악해보자.

```
sentence = "Your time is limited, so don't waste it living someone else's life."
print(sentence.find("me")))
print(sentence.find("li"))
print(sentence.find( “ live"))
print(sentence.rfind("me")))
print(sentence.rfind("li"))
```

그리고 startswith와 endswith 연산도 각각 True와 False 결과를 내주는 사례를 하나씩 만들어보면서 연산의 의미를 이해하도록 하자.

대상 문자열의 검색 범위를 위치번호를 지정하여 제한할 수도 있다. 시작 위치번호와 끝 위치번호를 문자열 조각내기와 동일한 요령으로 인수로 추가하면 된다. 예를 들어, 다음을 편집창에서 한줄 씩 실행하여 결과를 확인해보자.

```
sentence = "Your time is limited, so don't waste it living someone else's life."
print(sentence.find("me"))       # sentence에서 "me"가 처음 나오는 위치번호
print(sentence.find("me",8))     # sentence[8:]에서 "me"가 처음 나오는 위치번호
print(sentence.find("me",8,49))  # sentence[8:49]에서 "me"가 처음 나오는 위치번호
```

나머지 연산도 똑같은 방법으로 검색 범위를 지정해줄 수 있다.

첫째로 나타나는 문자열 하나만 찾기

파일이름 filename과 찾을 문자열 key을 받아서 파일에서 문자열이 처음 나타나는 위치번호를 "result.txt" 파일을 만들어 쓰는 프로시저 find_first를 만들어보자. 즉, 텍스트 파일이름이 article.txt이고 찾으려는 문자열이 "컴퓨터"이면, find_first("article.txt","컴퓨터")를 호출하고, "컴퓨터"가 "article.txt"에서 처음 나타나는 위치번호를 "result.txt" 파일에 쓰고, 없으면 not found 를 쓴다.

이 작업 절차는 다음과 같다.

1. 읽고 쓰는 파일을 각각 연다.
2. 읽을 파일 전체를 문자열로 읽어온다.
3. find 메소드를 사용하여 key가 있는 위치를 찾는다.
4. 쓰는 파일에 위치번호를 쓴다.
5. 연 파일을 모두 닫는다.

이 작업을 하는 프로시저는 다음과 같이 작성한다.

```
1  def find_first(filename,key) :
2      infile = open(filename,"r")
3      outfile = open("result.txt","w")
4      text = infile.read()
5      pos = text.find(key)
6      if pos == -1:
7          outfile.write(key + " is not found.\n")
8      else:
9          outfile.write(key + " is at " + str(pos) + ".\n")
10     outfile.close()
11     infile.close()
12     print("done")
```

이 프로그램을 이해하고, article.txt 파일을 다운받아 실행해보자.

```
find_first("article.txt","컴퓨터") # 컴퓨터의 위치번호는 1982
find_first("article.txt","한양대") # 한양대는 not found
```

둘째로 나타나는 문자열 하나만 찾기

파일이름 filename과 찾을 문자열 key을 받아서 파일에서 문자열이 두번째 나타나는 위치번호를 "result.txt" 파일에 쓰는 프로시저 find_second를 만들어보자. 즉, 텍스트 파일이름이 article.txt 이고 찾으려는 문자열이 "컴퓨터"이면, find_second("article.txt","컴퓨터")를 호출하고, "컴퓨터"가 "article.txt"에서 두번째 나타나는 위치번호를 "result.txt" 파일에 쓰고, 없거나 한번만 나타나면 not found를 쓴다.

이 작업 절차는 다음과 같다.

1. 읽고 쓰는 파일을 각각 연다.
2. 읽을 파일 전체를 문자열로 읽어온다.
3. find 메소드로 key가 있는 위치를 찾는다.
4. 찾은 바로 다음 위치번호에서 시작하여 find 메소드로 key가 있는 위치를 한번 더 찾는다.
5. 쓰는 파일에 위치번호를 쓴다.
6. 연 파일을 모두 닫는다.

이 작업을 하는 프로시저는 다음과 같이 작성한다.

```
1  def find_second(filename,key) :
2      infile = open(filename,"r")
3      outfile = open("result.txt","w")
4      text = infile.read()
5      pos = text.find(key)
6      pos = text.find(key,pos+1)
7      if pos == -1:
8          outfile.write(key + " is not found.\n")
9      else:
10         outfile.write(key + " is at " + str(pos) + " the 2nd time.\n")
11     outfile.close()
12     infile.close()
13     print("done")
```

이 프로그램을 이해하고 실행해보자.

```
find_second('article.txt','컴퓨터')    # 컴퓨터의 위치번호는 2016
find_first('article.txt','데스크탑')   # 데스크탑의 위치번호는 6360
find_second('article.txt','데스크탑')  # 데스크탑은 not found
```