< 문자열(string) >

* 문자열
* 패턴 매칭
* 문자열 암호화
* 문자열 압축
* **ASCII**

: 7bit 인코딩으로 128문자를 표현하며 33개의 출력 불가능한 제어 문자들과 공백을 비롯한 95개의 출력 가능한 문자들로 이루어져 있다



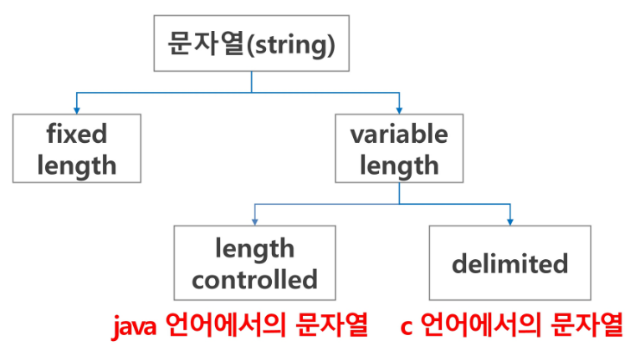
* **유니코드**

: 다국어 처리를 위해 마련한 표준

* 유티코드도 다시 Character Set으로 분류된다.
* UCS-2(Universal Character Set 2)
* UCs-4
* 유니코드를 저장하는 변수의 크기를 정의
* 그러나, 바이트 순서에 대해서 표준화하지 못했음
* 다시말해 파일을 인식 시 이 파일이 UCS-2, UCS-4인지 인식하고 각 경우를 구분해서 모두 다르게 구현해야 하는 문제 발생
* 그래서 유니 코드의 적당한 외부 인코딩이 필요하게 되었다.

**[문자열]**

* 문자열의 분류



* **Python에서의 문자열 처리**
* Char 타입 없음
* 텍스트 데이터의 취급방법이 통일되어 있음
* 문자열 기호

: ‘(홀따옴표), “(쌍따옴표), ‘’’(홑따옴표 3개), “””(쌍따옴표 3개)

: + 연결(Concatenation)

* **문자열 + 문자열** : 이어 붙여주는 역할

: \* 반복

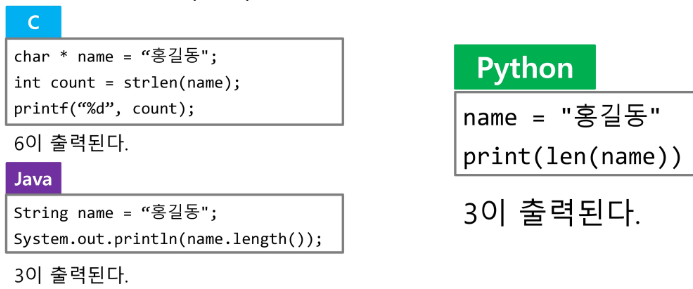
* **문자열 \* 수**: 수만큼 문자열이 반복
* 문자열은 시퀀스 자료형으로 분류되고, 시퀀스 자료형에서 사용할 수 있는 인덱싱, 슬라이싱 연산들을 사용할 수 있음
* 문자열 클래스에서 제공되는 메소드

: replace( ), split( ), isalpha( ), find( )

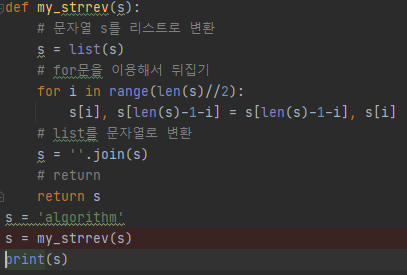
* 문자열은 튜플과 같이 요소값을 변경 할 수 없음

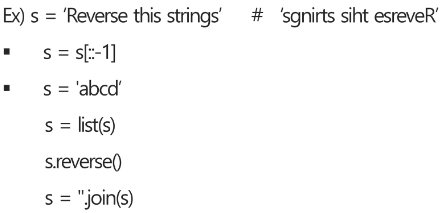
(immutable)(리스트로 바꿔서 값 변경하고 join써야함)

* C와 Java의 String 처리의 기본적인 차이점
* C는 아스키 코드로 저장
* Java는 유니코드(UTF16, 2byte)로 저장
* 파이썬은 유니코드(UTF8)로 저장



* **문자열 뒤집기**
* 자기 문자열에서 뒤집는 방법이 있고 새로운 빈 문자열을 만들어 소스의 뒤에서부터 읽어서 타겟에 쓰는 방법이 있다.
* 자기 문자열을 이용할 경우는 swap을 위한 임시 변수가 필요하며 반복 수행을 문자열 길이의 반만을 수행해야 함





* **문자열 비교**

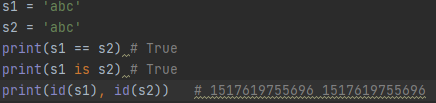
: **==** 연산자(값이 같은지)와 **is** 연산자(주소가 같은지) 제공

* == 연산자는 내부적으로 특수메서드 \_\_eq\_\_() 호출

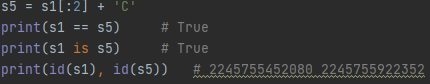
\*C : strcmp() 함수를 제공

\*Jave: equals() 메소드를 제공

-> 문자열 비교에서 ==연산은 메모리 참조가 같은지 묻는 것



**# 파이썬은 재활용 함**



* **문자열 숫자를 정수로 변환하기**

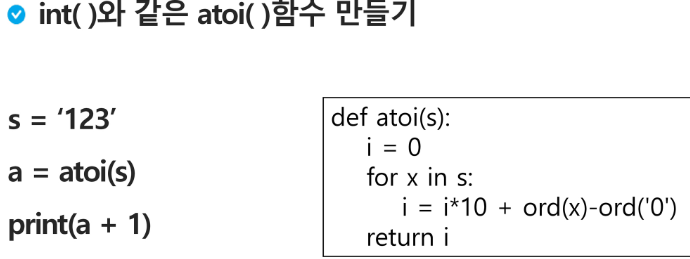
: 숫자와 문자 변환 함수를 제공

* C언어

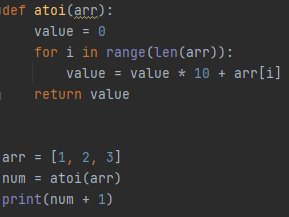
: atoi()함수(asci to int / arr to int) 제공. 역 함수로는 itoa()

* Java

: 숫자 클래스의 parse 메소드 제공



**2진수로 바꾸려면 \*2로 바꾸기**



**[패턴 매칭]**

* 패턴 매칭에 사용되는 알고리즘
* 고지식한 패턴 검색 알고리즘
* 카프-라빈 알고리즘
* KMP 알고리즘
* 보이어-무어 알고리즘

# 해시 함수

: 임의의 길이를 갖는 메시지를 입력 받아 고정된 길이의 해시 값을 출력하는 함수

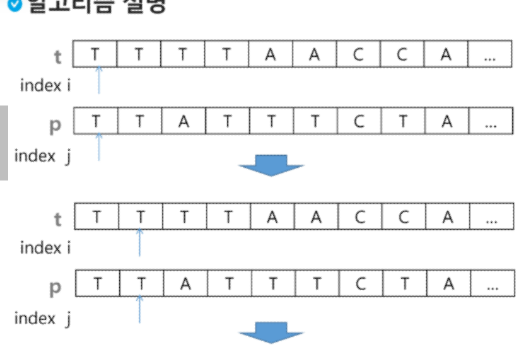
목적: 메시지의 오류나 변조를 탐지할 수 있는 무결성 제공

* 어떤 입력 값에도 항상 고정된 길이의 해시 값을 출력
* 눈사태 효과: 입력 값의 아주 일부만 변경되어도 전혀 다른 결과 값을 출력
* 출력된 결과 값을 토대로 입력 값을 유추할 수 없다.

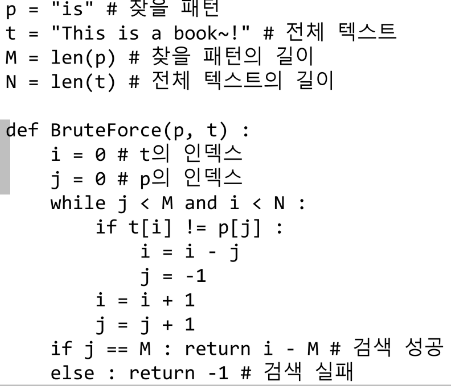
1. **고지식한 알고리즘(Brute Force)**

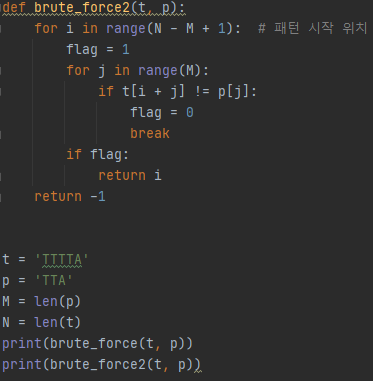
: 본문 문자열을 처음부터 끝까지 차례대로 순회하면서 패턴 내의 문자들을 일일이 비교하는 방식으로 동작











* 고지식한 패턴 검색 알고리즘의 시간 복잡도
* 최악의 경우 시간 복잡도는 텍스트의 모든 위치에서 패턴을 비교해야 하므로 O(MN)이 됨
* 비교횟수 줄이는 방법은?

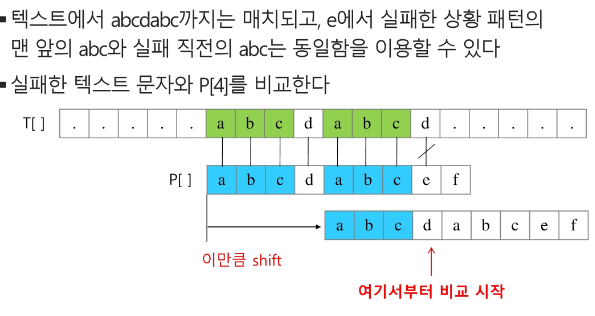
1. **KMP 알고리즘**

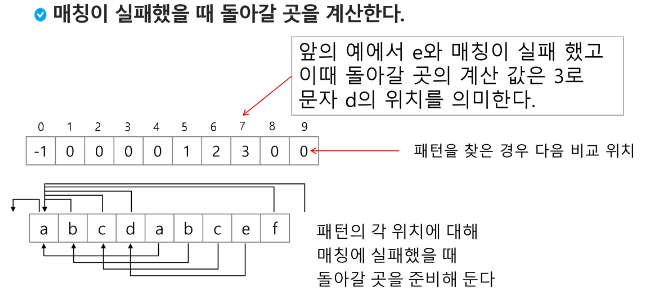
: 불일치가 발생한 텍스트 스트링의 앞 부분에 어떤 문자가 있는지를 미리 알고 있으므로, 불일치가 발생한 앞 부분에 대하여 다시 비교하지 않고 매칭을 수행

: **패턴을 전처리**하여 배열 next[M]을 구해서 잘못된 시작을 최소화함

\*next[M]: 불일치가 발생했을 경우 이동할 다음 위치

: 시간 복잡도 : O(M+N)

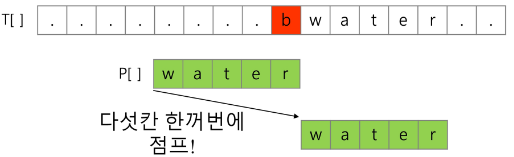




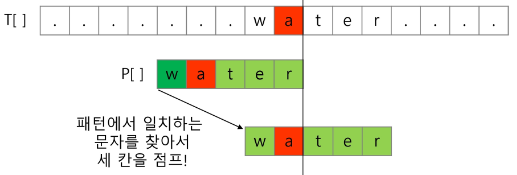
**접두어와 접미어 1개 같음**

1. **보이어-무어 알고리즘**

* **오른쪽에서 왼쪽으로 비교**
* 대부분의 상용 소프트웨어에서 채택하고 있는 알고리즘
* 보이어-무어 알고리즘은 패턴에 오른쪽 끝에 있는 문자가 불일치 하고 이 문자가 패턴 내에 존재하지 않는 경우, **이동 거리는 무려 패턴의 길이 만큼**이 된다.



* 오른쪽 끝에 있는 문자가 불일치 하고 이 문자가 패턴 내에 존재할 경우





* 문자열 매칭 알고리즘 비교
* 찾고자 하는 문자열 패턴의 길이 m, 총 문자열 길이 n
* 고지식한 패턴 검색 알고리즘: 수행시간 O(mn)
* 카프-라빈 알고리즘: 수행시간 O(n)
* KMP 알고리즘: 수행시간 O(n)

