Week 9

프로그램은 일단 기본적으로 순서대로 명령을 수행한다.

이를 이용하여 어떤 업무가 주어졌을 때 우리는 코드를 순서에 맞게 끔  그 단계를 계획하고 입력한다. 이 때 이 순서를 배정한 단계들의 집합을 알고리즘이라고 한다.

프로그래밍에서 알고리즘은 매우 중요한 항목이다 . 왜냐하면 이 알고리즘에 따라 프로그램이 동작하는 속도를 빠르게 , 크기를 작게 만들어 더 최적된 상태에서 효율적으로 프로그램을

실행할 수 있기 때문이다.

알고리즘 을 짠 이후 리팩토링도 중요하다.

리팩토링은 프로그램을 작성한 후 프로그램의 기능은 유지하면서 코드의 구조를 간결하게 코드의 재사용성을 높이고 코드의 가독성을 높이는 행위이다.

이런 과정을 거쳐야 우리의 프로그램이 남들이 보기에도 좋고 용량도 적어지며 한번 만든 코드를 재활용 할 수있으니 생산성과 디자인, 그리고 효율성에서 모두 필요한 과정이다.

일단 알고리즘 부터 알아보자

Week 9에서는 여러가지 알고리즘을 제시하고 직접 그 작업을 수행하는 시간을 비교해본다 . 따라서 코드를 작성해봐야겠다.

책은 정렬되지 않은 요소들의 리스트에서 가장 작은 두 요소를 찾는 작업을 제시했고

find, remove, find 방법

sort, identify minimums, get indices 방법

walk through list를 먼저 제시했다.

하나씩 천천히 알아보자

1.Find, remove, find

순서 :

리스트의 가장 작은 요소를 찾아  그 요소의 index를 first index 에 저장 -> 이후 가장 작은 요소를 제거 -> 가장 작은 요소를 제거한 리스트에서 다시 가장 작은 요소(두번째로 작은 요소)를 찾음

->처음에 찾은 가장 작은 요소(원래 리스트에서 가장 작은 요소)를 다시 원래대로 집어넣는다

-> 그리고 원래 리스트로 복구 된 상태에서 두번째로 작은 요소의 index를 second index에 저장(필요하다면 second index 수정 가능)

->저장한 first index와 second index를 통해 가장 작은 두가지 요소의 index를 출력

2. sort, identify minimums, get indices

순서  :

리스트 요소를 복사(copying)해 다른 변수 리스트에 할당 한다

-> 그리고 카피한 리스트를 정렬해 가장 작은 요소의 값을 알아낸다.(index 0, 1 번호가 가장 작을 것. sort 하면 가장 작은 값 부터 배열됨)

->이후 그 값을 통해 원래 리스트에서 해당 index를 출력

3. walk through list

min1과 min2를 만들어  가장 작은 요소의 index와 그 다음으로 작은 요소의 index를 가리키게 한다.

-> 리스트의 첫번 째 요소 (index 0) 부터 시작해서 지나쳐온 값 들과 비교한다.

-> min1과 min2를 계속 업데이트(수정) 시킨다.(작은 값이 나오면)

->요소의 끝까지 찾은 후 min1 과 min2 를 출력한다.

이후 time 모듈에서 perf\_counter 함수를 통해  끝시간-시작시간을 해 작동시간을 알아본다.

요소의 개수가 1400개인 예제는 그렇게 시간 차이가 나지않아서 간단하고 명확한 것이 더 중요하다.

하지만 요소의 개수가 백만개 , 천만개인 경우에는 어떨 것 같냐고 물어본다.

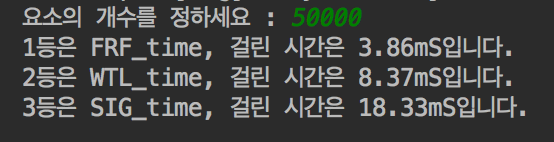
시간 차이가 많이 날 것이고 그러면 시간 이 적게 걸리는 것이 더 좋은 알고리즘이라고 생각할 수 있다.

좋은 프로그램 이란 빠르고, 메모리 사용량이 적은 프로그램이 좋은 프로그램이다.

Searching for the smallest value

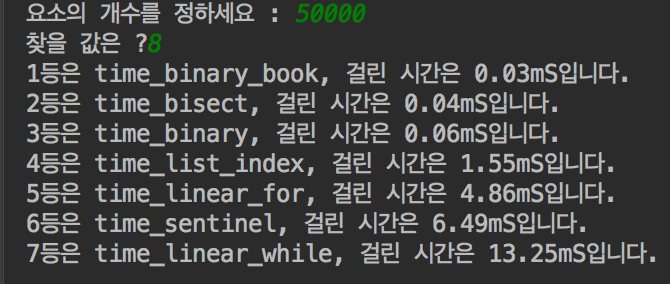
속도는 Find,Remove,Find  > Walk Through List > Sort Idetify minimum, Get indicies

순이라는 것을 알았다.



책에 나와있는 linear while , linear for, sentinel, list.index, binary search(내가 만든 것) binary search in the book, bisect 이 7가지의 속도를 비교해 보았다

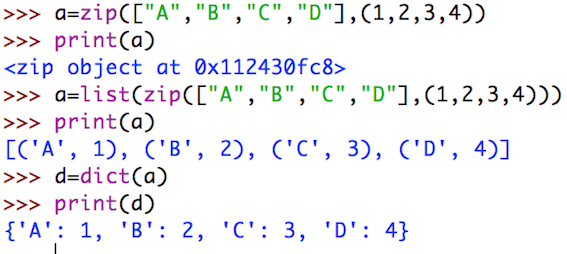
책에 나와있는 결과 그대로 나온 것을 확인할 수있고 정말 신기한건 확실히 책에 나와있는 예제가 가장 속도 가 빠르게 나왔다는 것이다



하다가 알게 된 내장함수가 있는데 zip이라는 내장함수이다.

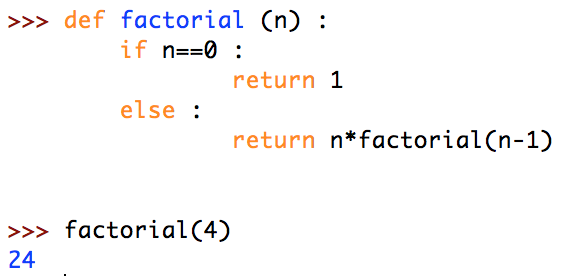
zip(\*iterable) (\*iterable은 Week 2에서 봤듯이 가변형 인자를 받을 수 있어 입력값을 여러개를 받을 수있다.)

요소의 개수가 같은 반복가능한 자료형을 하나하나씩 묶어 준다.



합병정렬 코드를 짜다가 재귀함수라는 것을 알았다.

일종의 루프를 만들 수 있는데 마지막에 리턴에 자기 자신의 함수를 호출을 하는 것이다.



대표적으로 팩토리얼을 구하는 함수를 구현할 때 쓰인는데 factorial(4)를 입력하면 factorial(3)이 호출되어 다시 함수의 코드를 거치다가 n=0이 되었을 시 1이 나와서 루프가 끝나고

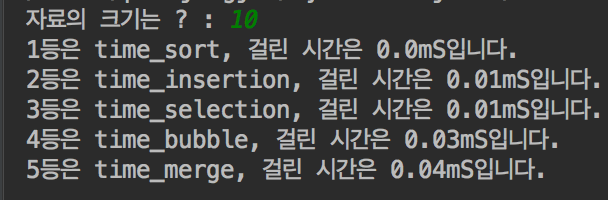
최종적으로 24라는 결과를 도출한다.

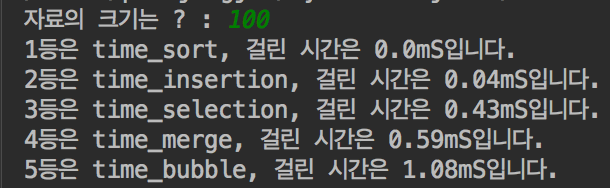
합병정렬을 할 때 리스트를 쪼개고 다시 합치는 과정에서 필요한 것 같다.

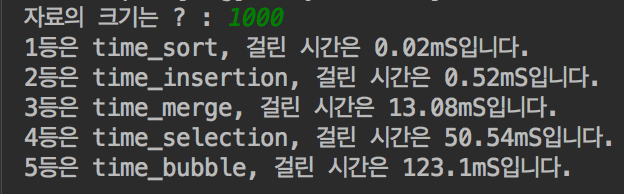
재귀함수가 이해가 안가 [Memory visualization](http://pythontutor.com/visualize.html#mode=display) 을 통해 이해를 했다. 처음에 저장공간과 변수에 할당된 객체가 바뀌는데 기존에 있던 정보는 어떻게 되는건지 아예 이해가 안갔었는데

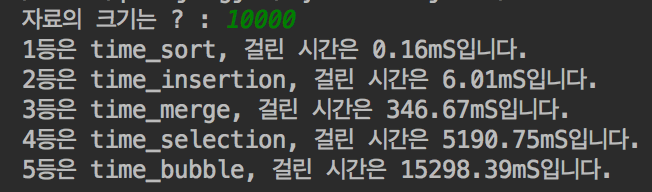
처리가 되지 않은 상태로 남겨지는 것이다. 다시 올라가 함수를 호출하고 다시 올라가 호출하다 자기 자신을 호출하지 않는 리턴을 만나면 이제 계속 펼쳐진 리턴값들이 역순으로 처리되어 역순으로 올라가 처음에 리턴되었던 값을 처리하는 것인 것 같다.(내가 썼는데도 정리가 안된다.)

이 정렬 알고리즘과 파이썬 리스트 매서드 sort()의 시간을 비교해보았다









자료의 크기 가 커질수록 내장함수 sort()의 위력은 강력하고 버블은 최악의 효율은 보여준다.

* linear search

index 0번부터 0,1,2,3,....순으로 차례로 값을 비교해서 그 값을 찾았을 때 멈추고 그 값을 반환하는 알고리즘

* sentinel(보초값) search

맨 뒤에 찾는 값을 추가(.append())하고 찾는 값을 sentinel로 정해 데이터의 종료를 알린다. 그후  index를 0번부터 0,1,2,3,... 순으로 차례로 결과를 비교하고 결과를 찾으면 while문을 멈추고

.pop() 매서드를 통해 맨 뒤에 저장되있던 찾을 값을 꺼낸다.

pop으로 꺼내는 이유 : 처음에 찾을 값을 리스트의 마지막에 위치한 것을 알고있는 상태이고 그러면 while문으로 찾는 행위를 반복하다가 보면 마지막에 무조건 찾게 되어 while문을 종료한다.

그 러면 우리가 지정한 i(1씩 증가하도록 만든 변수)는 리스트의 마지막 index번호가 된 상태이다.

안꺼내면  if 에서 index번호가 ==len(list)라고 해 못찾으면 -1로 반환하려고하는데 뒤에 하나를 추가하고 안 꺼내면

index번호는 len(list)와 무조건 같을 수가 없으므로 (len(list)가 무조건 index보다 1 이 크다(파이썬은 index가 0부터 시작하므로))

결국 출력하는 index번호는 -1은 절대 안나오고 끝난 리스트의 마지막 index번호를 리턴한다.

그런데 이 index번호는 기존의 리스트에서 벗어난 index이므로 우리가 원하는 값이 나오지 않게 되는 것이다.

linear\_search와 다른 점은 while의 조건이 하나이므로 while문으로 반복을 할지 말지 판단 할 때 한번의 연산만 거치면 된다는 것이다.

이런 linear search는 리스트의 길이가 길면 비효율적이지만 단순하여 구현이 쉽고 정렬되지 않은 리스트에도 작동할 수 있는 장점이 있다.

시간 복잡도는 O(n)으로 리스트의 길이만큼 계산 복잡도가 n만큼 늘어난다.

* binary search(정렬된 경우 매우 좋은 알고리즘)

시간복잡도가 O(logn)인 알고리즘

리스트가 정렬되있다는 것만을 알고 있는 상태에서 매우 좋은 알고리즘

리스트의 가운데 점을 비교하고 그 이상인지 그 이하인지 비교해서 그 이상이면 앞 의 반 쪽을 잘라버리고 그 이하이면 뒤의 반쪽 을 잘라버리고

남은 반쪽 리스트로 또 가운데점을 비교해 같은 행위를 계속하는 재귀함수인 알고리즘 , 찾을 때까지 반복한다.

binary search는 정확하게 O(log_2n)의 시간복잡도를 가지는데 예를들어  2^{21}=2097152의 리스트의 개수를 갖고있을 때

linear search는 최악의 경우(bad case) 2097152회의 탐색을 가지는데 binary search는 21회가 최악의 경우이다.

이처럼 숫자가 커지면 커질수록 둘의 시간 복잡도의 차이는 극심하게 날 것이다.

파이썬 모듈중 bisect라는 모듈이 있다 그안에 bisect(list, num, beg, end)함수가 있는데 이 함수를 이용하면 binary search를 할 수있다.

첫번째 인자에 리스트를 넣고 두번째에 찾을 값, 시작지점(생략가능) , 끝지점(생략가능) 이렇게 넣을 수있다. 함수를 사용하면 리스트를 정렬해주고 binary search를 해준다.

bisect\_left( ) 는 같은 값이 있을 경우 앞에 있는 값을 중심으로 search를 해주고 bisect\_right()는 뒤에 있는 값을 중심으로 search를 해준다.

* 시간 복잡도

시간복잡도의 대표적 유형 다섯가지를 좋은 순서대로 나열해보면

O(1) > O(logn) > O(n) > O(nlogn) > O(n^2) > O(C^n) 순이다

* O(1) (상수시간) : 어떤 과정을 할 때 입력값이 얼마가 되었든 딱 일정한 시간(상수)만을 필요로 한다.

-> 입력값이 n인 어떤 문제를 해결할 때(예를 들어 searching) 단 한 단계의 과정(따라서 시간이 상수 값을 갖는다)만을 거친다.

(우리가 리스트가 정렬되있고 찾는 값의 index값을 안다면 그 값을 한 단계만을 거쳐 그 값을 찾을 수 있을 것이다.)

* O(logn) (로그시간) : 어떤 과정을 할 때 입력값이 n이면 단계의 과정이 logn의 비율만큼 적용되는 복잡도

->  입력값이 n인 문제를 해결할 때 연산을 할 때 마다 특정 요인에 의해 단계가 줄어든다. (binary search 처럼 가운데 지점 비교후 반 쪽을 버리는 행위)

* O(n) (선형시간) : 어떤 과정을 할 때 입력값이 n이면 단계의 과정이 an+b 만큼 적용되는 복잡도

->입력값이 n인 문제를 해결할  때 n과 1:1비율로 단계 수가 적용된다.(linear search같은 경우 리스트 길이가 n이면 최악의 경우 과정이 n번 일어나야한다.)

* O(nlogn) (선형로그시간):  어떤 과정을 할 때 입력값이 n이면 단계의 과정이 anlogn+b 비율로 적용되는 복잡도

->입력값이 n인 문제를 해결할 때 n과 nlogn비율로 단계 수가 적용된다.

1.Merge sort(합병 정렬)

<https://youtu.be/XaqR3G_NVoo>

개발자는 [존 폰 노이만](https://namu.wiki/w/%EC%A1%B4%20%ED%8F%B0%20%EB%85%B8%EC%9D%B4%EB%A7%8C)이라고 한다.

요소의 개수가 1 또는 0이 될 때까지 두 부분으로 계속 나눈다.

-> 자른 순서의 역순(?)으로 크기를 비교해 부분들을 합친다 (자르는 순서는 앞 쪽에서부터 차례차례 자른다)

(나무위키 gif파일을 확실히 이해해보려 직접 해보았다.)

([6,5,3,1,8,7,2,4] -> [6,5,3,1],[8,7,2,4] -> [6,5],[3,1],[8,7],[2,4] -> [6],[5],[3],[1],[8],[7],[2],[4]

->1.[6],[5] 비교 [5,6] 생성   2. [3],[1] 비교 [1,3] 생성   3. [8],[7] 비교 [7,8] 생성   4. [2],[4] 비교 [2,4] 생성

->1. [5,6]의 첫 번째 요소 5와 [1,3]의 첫 번째 요소 1 비교 새로운 부분 []에 1 삽입   2. [5,6]의 첫 번째 요소 5와 [1,3]의 두 번째 요소 3 비교 새로운 부분에 [1]에 맨 뒤에 3 대입

    3. 남은 5대입 이후  6대입(여기서는 비교할 필요가 없는 것이 먼저 정렬이 된 상태(정렬을 했기 때문에)임을 알기 때문에 차례로 삽입하면 된다) [1,3,5,6] 생성

->1,[7,8]의 첫 번 째 요소 7과 [2,4]의 첫 번째 요소 2 비교 새로운 부분에 []에 2 삽입   2. [7,8]의 첫 번 째 요소 7과 [2,4]의 두 번째 요소 4 비교 새로운 부분에 [2]에 4 삽입

    3. 남은 7대입 이후 8대입(여기서도 마찬가지로 비교할 필요가 없다(이미 정렬을 한 부분이기 때문에)) [2,4,7,8] 생성

->1. [1,3,5,6]의 첫 번째 요소 1과 [2,4,7,8]의 첫 번째 요소 2를 비교 새로운 부분 [] 에 1 삽입   2. [1,3,5,6]의 두 번 째 요소 3과 [2,4,7,8]의 첫 번째 요소 2를 비교 새로운 부분[1] 에 2 삽입

3. [1,3,5,6]의 두 번 째 요소 3과 [2,4,7,8]의 두 번째 요소 4를 비교 새로운 부분[1,2] 에 3 삽입 , 4. [1,3,5,6]의 세 번 째 요소 5과 [2,4,7,8]의 두 번째 요소 4를 비교 새로운 부분[1,2,3] 에 4 삽입

4. ... 진행  5.[1,2,3,4,5,6,7,8] 생성)

합칠 때 보면 요소를 남겨두고 복사하는 것이 아니라 그 요소를 꺼내서 새로운 부분에 대입하는 것인 것 같다.

자료 하나를 통으로 새로 만드는 것이므로 데이터 메모리 가 두배(?) 정도 필요 할 것 같다.(기존 리스트와 똑같은 크기의 데이터를 새로 만드는 것이므로)

 장점은 자료의 정렬 상태는 거의 상관 없다는 점과 안정적이라는 점라고 한다.

힙과 퀵이라는 정렬방법에서는 같은 값을 갖는 요소가  기존 위치에서 변경될 수 있지만 이 합병 정렬은 그런 경우가 없다고 한다.

* O(n^2) (2차시간): 어떤 과정을 할 때 입력값이 n이면 단계의 과정이 k\*n^2 만큼 적용되는 복잡도

->입력값이 n인 문제를 해결할  때 단계의 수는 n^2의 비율만큼 적용된다.

1. bubble sort  (버블정렬)

<https://youtu.be/lyZQPjUT5B4>

 첫 번째와 두 번째를 비교하여 정렬하고 두번 째와 세 번째를 비교하여 정렬하고 ...n-1번째와 n번째를 비교하여 정렬한 뒤

다시 처음으로 돌아가 첫 번째와 두  번째를 비교하여 정렬하고 .....  해서 최대 \frac{n(n-1)}{2}번 반복한다.

이미 정렬된 자료에서는 1번의 과정만 거치면 되지만 그 이외의 경우에서는  효율적인 면에서 정말 안좋은 것 같다.

장점으로는 간단히 구현할 수 있고 직관적이다(교재에서 나온 clarity and simplicity). 입력값이 적은 경우에서는 괜찮은데 입력값이 크면 속도가 너무 느리다.

각 프로그램마다 내장되어있는 정렬은 고효율의 알고리즘을 갖고있고 오픈소스로 좋은 정렬 알고리즘을 손쉽게 구할 수 있기 때문에  실제에서 쓰이는 경우는 거의 없다고 한다.

2.selection sort(선택 정렬)

<https://youtu.be/Ns4TPTC8whw>

선택정렬은 첫 번째부터 n까지 훑어서 가장 작은 게 첫 번째,  두 번째부터 n까지 훑어서 가장 작은 게  두 번째 ....  마지막으로  n-1번 째부터 n까지 훑어서 가장 작은게 n-1번째 까지 해서  n-1번 반복한다.

어떻게 정렬이 되어 있든 일관성 있게 \frac{n(n-1)}{2}에 비례하는 시간이 걸린다. 속도는 버블 정렬보다 두 배 정도 빠르다고 한다.

3.insertion sort(삽입 정렬)

<https://youtu.be/ROalU379l3U>

임의의 k번째 요소를 1부터 k-1까지와 비교해 적절한 위치(k-1 부터 시작해서 비교하다 어느 지점에서 k번째 요소 값이 더 크면  그 지점에 넣는다)에 끼워넣고 그 뒤의 자료를 한 칸씩 뒤로 밀어내는 방식이다.

평균적으론 O(n^2)중 빠른 편이라고 한다.

하지만  자료구조에 따라선 뒤로 밀어내는데 걸리는 시간이 크며, 앞의 예시처럼 작은 게 뒤쪽에 몰려있으면(내림차순의 경우 큰 게 뒤쪽에 몰려있으면) 효율이 매우 안 좋아진다.  
다만 이미 정렬되어 있는 자료 리스트에 어떤 자료를 하나씩  삽입이나 제거 하는 경우에는 효율이 매우 좋은 정렬 알고리즘이라고 한다.

* O(C^n)(지수시간) :어떤 과정을 할 때 입력값이 n이면 단계의 과정이 C^n 비율만큼 적용되는 복잡도

->입력값이 n인 문제를 해결할  때 단계의 수는 C^n의 비율만큼 적용된다.(ex. C=는 상수이므로 n이 100만 되어도 C^100만큼의 단계를 가진다.)

class and inheritance

class (쉽게말해서 내가 하나의 타입(클래스)를 만드는 것.)

인스턴스(instance)

 클래스 안에서 객체를 인스턴스라고 한다. 객체나 인스턴스나 같은 말인데 표현 방법이 다른 것 뿐이다.

인스턴스를 만드는 방법은

인스턴스명 = 클래스명() 으로 만든다.

* isinstance(객체명, 클래스명)

 객체가 클래스의 인스턴스인지 boolean 형태로 리턴하는 매서드이다.

클래스안의 def(함수)는 매서드라고 한다.

\_\_init\_\_ (Constructor , 생성자)

\_\_init\_\_메서드는  객체가 생성될 때 자동으로 호출되는 메서드이다.

객체를 생성할 때 객체의 정보?를 초기화 한다고 생각하면 된다.

따라서 \_\_init\_\_ 을 쓰면 객체를 생성할 때 뒤에 \_\_init\_\_ 에 설정한 변수 숫자 에 맞게끔 인자 를 넣어줘야한다.

인스턴스 명 = 클래스명 ( self를 제외한 인자 )

그리고 클래스 파트를 읽다보면

\_\_init\_\_ 밑에

self.title =title

이런 형태가 있는데 의미는

객체를 생성할 때 받은 title의 정보를 인스턴스의  title로 만드는 것이다

즉  a라는 인스턴스가 있으면

a.title은 a 인스턴스 를 생성 했을 때 설정한 title이 나오는 것이다.

이처럼 인스턴스의 정보를 리스트로 받을 수 있는데

리스트를 init으로 받을 때 aliasing 방법보다 copying을 해서 받아야한다.

def \_\_init\_\_ (self, a,b, ...) :

클래스의 메서드를 생성할 때는 항상 맨 앞에 self를 넣어 줘야하는데 그 이유는 self를 지정하면 자연스레 self가 인스턴스가 되기 위함이다.

인스턴스가 메서드를 호출 할 때 호출한 객체 자신이 self 가 되는 것이다.

그래서 메서드를 호출 할 때 방법은

인스턴스.메서드(self를 제외한 변수)

클래스.메서드(인스턴스,  self를 제외한 변수)

이렇게 두가지 방법이 있다.

string 클래스 메서드를 기억해보면

str.count(객체, ....)

객체.count(....)

이렇게 두가지 방법으로 불렀던 것을 기억이 난다.

\_\_str\_\_

인스턴스를 출력하면 형식을 지정해주는 메서드

\_\_eq\_\_

인스턴스 == 인스턴스를 정의 하는 메서드

\_\_이 붙은 메서드를 매직 메서드 Double Under Method라 하여 DUNDER 메서드라고 도 한다.

여러가지 내장함수를 재정의 함으로써 인스턴스 관리를 더욱더 편하게 해준다.

종류는 정말 많고 사용하면 좋은 메서드들이 많으니  한번 정리해봐야 겠다.

Override vs Overload

Override : 부모클래스에서 상속된 자식 클래스에서 부모클래스에게 받았던 메서드를 다시 재정의 하는 것.

기존 부모 클래스에 적용되었던 메서드는 지워지고 자식 메서드에서 새로 정의하는  것.

Overload : 같은 클래스 안에 같은 이름의 메서드를 정의하지만 각 메서드 들 마다 받는 parameter가 다른 것을 overload라 한다.

inheritance( 상속)

클래스는 상속할 수 있다는 말이 기존 클래스가 갖고있던 매서드,프로퍼티 모두 상속한 클래스에서도 지정하지 않아도 똑같이 사용할 수 있다는 것이다.

모든 클래스는 object클래스가 부모 클래스이다. 조상? 느낌이어서 클래스를 만들면 따로 지정을 안해도 부모 클래스는 당연히 object 클래스인 것이다.

class 클래스 명(부모클래스) :

이런 식으로 부모클래스의 내용을 상속 받을 수 있다.

super() :  자식클래스에서 부모클래스의 내용을 사용하고 싶을 때 사용하는 메서드

자식클래스에서 오버라이드 해서 재정의 할 때 사용

super() 를 쓰면 다시 쓰지 않아도 부모클래스의 내용을 그대로 가져와 사용할 수있다.

super().부모클래스 내용