Python Start

- 소민호

01. 파이썬 알아보기

기초부터 차근차근

- 파이썬 설치하기
- 다른언어와 파이썬의 차이점

Python 3.x 다운로드 및 설치하기

1. https://www.python.org/downloads/windows 접속 후 (귀찮으면 그냥 구글에서 python 검색을..) Download Python 3.6.4 클릭하기

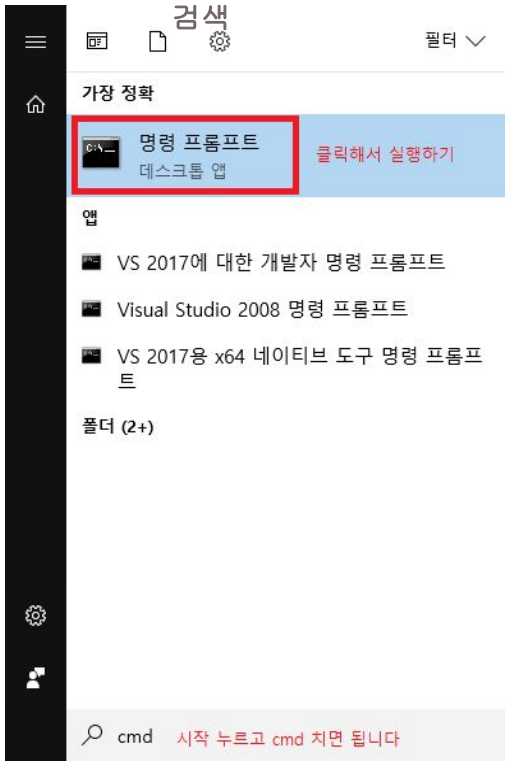
2. 다운로드 된 파일을 실행 해서 Next만 눌러주면 돼요

3. 설치가 잘 완료 됐나요?

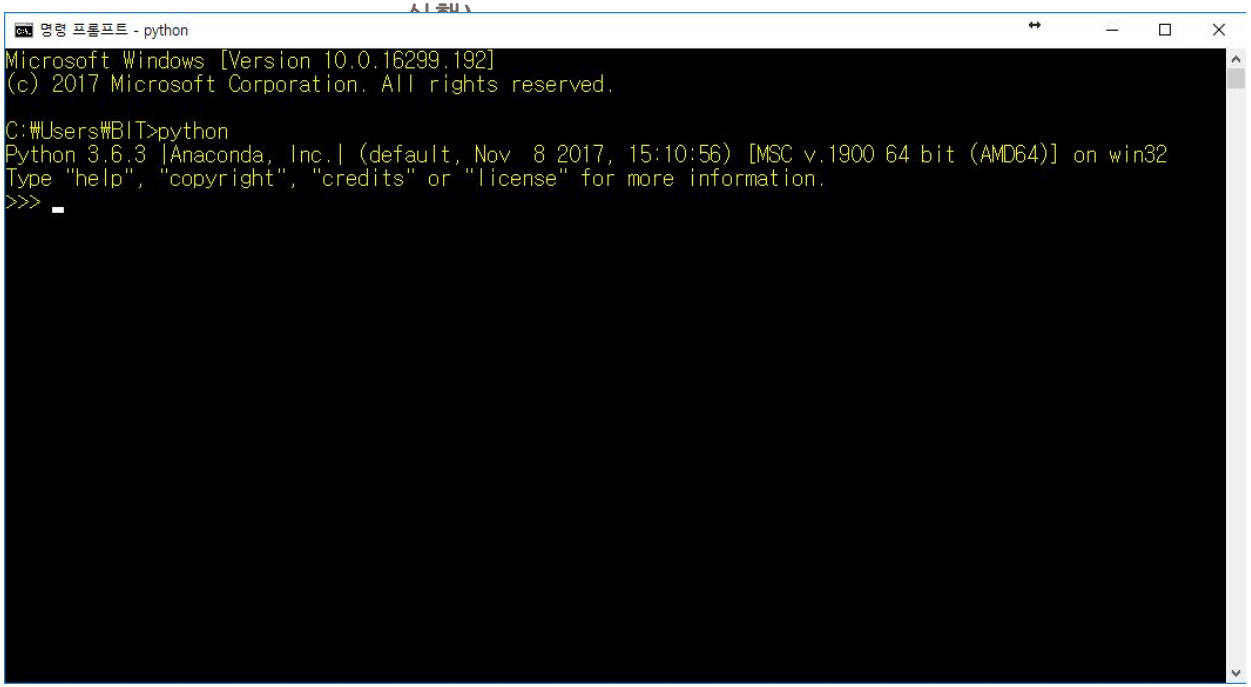
Python 3.x 정상 설치 확인하기

혹시 모르니까 설치가 잘 됐는지 확인해 봅시다

1) 시작 -> cmd



2) python 입력하고 실행 (인터프리터



간단하게 **Python** 인터프리터를 실행한

모습

Python과 다른 언어의 차이점은 뭘까요

Hello World를 자바와 파이썬으로 각각 출력 해보면?

Java에서 Hello World를 출력 하기

```
* HelloWorld.java
 public class HelloWorld {
   public static void main(String[] args){
            String sayHello = "Hello World!";
            System.out.println(sayHello);
   }
 }
```

Python에서 Hello World를 출력 하기

```
* HelloWorld.py
 sayHello = 'Hello World!'
 print(sayHello);
```

자바는 변수의 자료형이 고정 되어 있는 정적 언어(static language)
파이썬은 변수의 자료형이 고정되어 있지 않은 동적 언어 (dynamic language)

Python은 이런 점이 좋습니다

쓰기(작성)도 쉽고 읽기도 쉽고 범용성도 좋아요

읽고 쓰기가 좋다는 이야기는

① 배우기 쉽다는 이야기 라고 할 수 있으며,

② 배우기 쉬우면 코드 작성도 쉽다는 이야기 입니다.

Hello World 예제를 다시 한번 봅시다.

자바로 작성 했었던 코드보다 훨씬 코드의 량이 줄어 듭니다!

즉 상대적으로 같은 동작을 하는 코드를 작성하면 파이썬이 훨씬 생산성이 좋다 라고 할 수 있네요

범용성이 좋다는 이야기는

① 거의 모든 곳에서 실행 됩니다. **– Windows던 Mac이던 Linux던 어디에서든!(이식성)**

② 많은 사람이 오픈소스로 만들어 놓은 라이브러리들을 제공합니다. **– 이미 내장 되어 있는 라이브러리들도 있습니다.**

Python 에서는 개발 할 때 필요한 라이브러리를 검색 후 바로 다운로드 받아서 설치 하는 플랫폼을 제공하고 있습니다.

많은 개발자들이 만들어 놓은 여러 라이브러리를 설치하고 필요한 기능을 수행하면 끝!

인터프리터가 뭔가요?

Compiler & Interpreter

기계가 알아 들을 수 있는 말로 변환 해주는 컴파일러(Compiler)와 인터프리터 (Interpreter)

① 컴파일러는 소스코드를 통째로 기계어로 번역합니다.

② 인터프리터는 개발자가 입력하는 **소스코드 한 줄을 실시간으로 번역**합니다.

그렇다면 인터프리터는 컴파일러보다 느리지 않을까요?

안타깝게도 인터프리터는 컴파일러보다 속도는 느립니다.만!

현재 많은 개선이 이루어 지고 있다고 합니다.(이미 충분히 됐습니다)

보통 동적 언어(dynamic language)에서 인터프리터를 많이 사용합니다.

① 정적 언어(static language)의 데이터 공급을 돕는 짧은 프로그램 (스크립트)을 만들기 위해 사용해 왔습니다.

② 이러한 정적 언어로 만들어진 프로그램과 프로그램을 이어주기 위해 작성되는 프로그램들을 **접착제 코드(Glue 코드)** 라고 합니다.

③ 인터프리터의 성능 향상이 매우 좋아져서 아주 큰 업무(웹 서버, 데이터 분석, 딥 러닝, 머신 러닝, 등등)도 잘 수행합니다.

인터프리터를 왜 쓸까요?

인터프리터의 용도는 테스트를 위한 환경입니다.

프로그램을 전부다 만들고 나서 실행을 해야 하는 방식은 컴파일러 방식 입니다. 하지만 빠르게 개발하고, 방금 만든 모듈을 테스트 하거나 **커맨드 라인에 코드를 입력 하고 바로 그 결과를 볼 수 있도록 제공 하는 것이 인터프리터** 입니다.

그렇다면 복잡한 코드는 어떻게 작성해야 할까요?

복잡한 코드를 꼭 인터프리터에 전부 입력 해 가면서 개발 할 필요는 당연히 없겠죠! **파이썬 소스파일(.py)**을 만들어서 파이썬 코드를 입력 한 후에 인터프리터에서 확인 해 볼 수 있습니다. **물론 편리 하게 개발 할 수 있는 파이썬 IDE가 많이 존재 합니다!(이클립스 pyDev, pyCharm, 각종 문서편집기 등등...)**

정리하자면

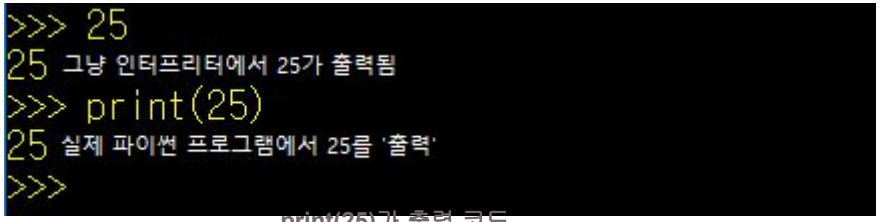
인터프리터는 여러분들이 만든 파이썬 소스코드 파일(.py)를 테스트 해볼 용도로 많이 사용 될 것입니다. 함수(function) 또는 클래스 및 모듈 등등을 만들고 간단하게 테스트 할 용도로 인터프리터를 사용 할 것입니다.

인터프리터에 코드 입력하기

>>> 에 값을 입력할 때 자동으로 출력 되는 부분을 제외 하고는 대부분 파일(.py)로 실행되는 파이썬 프로그램과 거의 동일하게 동작합니다.

25를 입력하면 그냥 단순히 시간을 절약 하기 위해 인터프리터에서 그냥 출력을 하게 됩니다. (실제 프로그램상에서 25가 '출력' 되는 것은 아님!)

print(25)를 입력 하면 '출력' 입니다. (자바의 System.out.println(25) 와 같습니다.)



print(25)가 출력 코드 입니다!

실시간으로 연산 및 코드의 결과가 나타납니다. 따라서 파이썬의 인터프리터를 *대화형 인터프리터* 라고도 합니다.

'대화형' 이라고 이름이 붙은 자체가 개발자가 코드를 한 줄 칠 때마다 인터프리터에서 곧바로 응답을 하기 때문에 붙은 이름 입니다. 다음 코드를 입력 해 보세요!

```
>>> 25 + 5
>>> 30 * 3
>>> 30 // 7
>>> 30 / 7
```

인터프리터를 종료 할 때는 CTRL + Z 를 누르고 엔터 키를 눌러주면 됩니다.

.py 파일을 작성 해 볼까요? 앞으로 작성할 예제들을 저장 하는 방법입니다.

먼저 예제를 작성할 폴더 부터 만들어 보겠습니다. 원하는 위치에 폴더부터 만들어 주시면 됩니다. 여기서는 D:\pythonexample\chap01 에 작성 하지만

어디에 만드셔도 별 상관은 없습니다. 최대한 접근하기 쉬운 위치에 만들어 주세요! **인터프리터가 아닌 일반 명령 프롬포트 에서 실행합니다.**

C:\Users\mhso > **D:** □ *현재 위치에서 D 드라이브로 이동*

D:\> **mkdir pythonexample** □ *pythonexample 폴더 만들기*

D:\> **cd pythonexample** □ *D:\의 pythonexample 폴더로 이동하기*

D:\pythonexample> **mkdir chap01** □ *chap01 폴더 만들기*

D:\pythonexample> **cd chap01** □ *chap01 폴더로 이동하기*

D:\pythonexample\chap01> **notepad.exe** □ *메모장 실행하기*

메모장이 실행되면 다음 코드를 입력 해주세요. 파이썬 코드 앞쪽에 #이 붙으면 주석입니다. 저장 할 때 chap01 폴더에 저장 하는 것을 잊지 마세요!

```
#Hello.py
sayHello = 'Hello Python'
print(sayHello)
```

저장 할 때 **반드시 파일 형식을 모든 파일(*.*)로** 바꿔 주고 **Hello.py 를 파일 이름**으로 지정하여 저장 해주세요

만약 **한글이 코드에 존재 한다면 인코딩 방식도 UTF-8**로 지정 해야 합니다

이제 인터프리터에서 .py를 실행 해 보시죠

앞서 만든 폴더에 *Hello.py* 파일이 잘 만들어 졌나요' 

아이콘 이 보이시면 제대로 저장 된 겁니다.

다시 명령 프롬포트에서 .py 파일을 실행 해 보겠습니다.

```
D:\pythonexample\chap01>  python   Hello.py
Hello Python
D:\pythonexample\chap01>
```

잘 실행 되시나요?

02. 파이썬의 데이터

파이썬을 활용하자

- 간단히 사용하는 기본 자료형
- 파이썬에서 문자열 다루기

프로그램에 필요한 재료들을 모아봅시다

<재료>라는 건 무엇일까요

요리 할 때 제일 먼저 무엇을 하시나요? 보통은 요리에 필요한 재료부터 준비합니다.

① 돼지고기, 양념, 각종 채소를 각각 그릇에 담아 놓자.

② 이름과 나이를 각각 변수에 담아놓고 이 사람의 정보를 담아 놓자.

요리를 할 때 필요한 재료들을 각각 그릇에 담아 놓듯이,
프로그램에 필요한 재료들을 각각 담아 놓아야 합니다.

재료들의 종류를 ***자료형****,*

재료 자체는 ***데이터(값)***

재료들을 저장할 그릇은 ***변수***

라고 할 수 있겠네요

요리 할 때 재료를 먼저 담아 놓는 것처럼,
프로그램을 만들 때도 재료(데이터)를 먼저 생각합시다.

그렇다면, 파이썬에서 사용 할 수 있는 재료는?

파이썬의 데이터 타입

기본적으로 파이썬은 다음과 같은 데이터 타입을 제공하고 있습니다.

① 부울**(bool)** □ True 혹은 False (논리 자료형)

② 정수**(int)** □ -1,-2, 0, 3, 100,200,300 같은 수

③ 실수**(float)** □ 3.14 같은 소수점이 있는 수, 또는 1.0e3 같은 지수

④ 문자열**(str)** □ 'Hello World!' 같은 문자들의 나열

인터프리터에서 확인해 보시죠. 다음 코드는 type이라는 기능을 사용하는 코드입니다. **기능(function - 함수)**에 대해서는 다음에 자세히 이야기 해보겠습니다

```
>>> type(True)
<class 'bool'>
>>> type(10)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('Hello Python!')
<class 'str'>
```

파이썬의 변수와 이름과 객체

파이썬에서 활용 할 수 있는 모든 것들은 객체로 되어 있습니다.

파이썬의 모든 것(데이터, 자료구조, 함수, 프로그램) 들은 전부다 객체로 구현 되어 있습니다.

객체란 무엇일까요?

1) 객체란 **실제 사용 할 수 있는 것**
2) 데이터와 기능을 갖는 것
3) 프로그램에서 실제 사용 할 수 있는 **부품**
4) 등등… 많은 정의가 있습니다

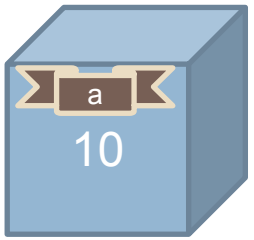
일단은 단순히 값을 담아 놓은 박스(Box)라고 이해 하겠습니다.

이 박스에는 우리가 **인식 할 수 있는 이름**이 붙어야 하는데, 이것을 **변수** 라고 합니다.

오른쪽 코드는 10(정수 □ int) 이 들어있는 박스에 a라는 이름을 붙이는 코드 입니다.

```
>>> a = 10 #변수 선언과 할당
>>> print(a) # a의 참조 값 출력
10
```

이제 변수 a는 정수를 담아 놓은 박스를 지정 해 놓았기 때문에 자료형이 int 입니다



a 라는 변수를 만드는 과정을 **선언**

10이 들어있는 객체를 a라는 이름을 붙이는(=) 과정을 **할당**

a라는 변수가 선언되어서 10이 들어있는 객체가 할당된 것을 **참조한다** 라고 표현합니다.

언제나 변수는 왼쪽에, 데이터는 오른쪽에 위치하며 할당 시에는 = 기호를 사용합니다.

덧뺄곱나를 해보자. 숫자 자료형과 연산자

파이썬은 정수와 실수를 다룰 수 있는 자료형를 제공합니다.

파이썬에는 기본적으로 정수와 실수를 다룰 수 있는 자료형인 int와 float을 제공합니다. 각종 **연산자**를 이용해 계산 할 수 있습니다.

인터프리터에서 각각 테스트 해보세요!

기호	의미	예시	결과
+	더하기	**5 + 5**	10
-	빼기	**10 – 5**	5
*	곱하기	**10*5**	50
/	나누기	**10 / 3**	3.3333333...
//	몫 구하기	**10 // 3**	3
**	지수계산	**3**4**	81

더하기, 빼기 등등 어떠한 연산을 하기 위해 사용하는 기호를 **연산자** 라고 합니다.

연산자에는 우선순위가 있으며, 우선순위가 높은 연산자가 먼저 작동합니다. 다음을 인터프리터에서 실행 해 보세요

```
>>> 2 + 2 * 2
>>> ( 2 + 2 ) * 2
```

원하는 자료형으로 바꿔주는 형 변환(Type Casting)

우리는 파이썬을 사용하면서 여러 가지 자료형으로 되어 있는 데이터를 입맛에 맞게 바꿀 수 있습니다.

예를 들어 실수를 정수로 바꾸고 싶다던가…

```
>>> int(10.5)
10
```

논리값(bool) 값을 정수로 바꿔보고 싶다던가…

```
>>> int(True)
1
>>> int(False)
0
```

심지어 문자열 형태의 숫자도 가능합니다

```
>>> a = int( '10' ) #변환 하고 나서 변수에 집어 넣을 수도 있어요
>>> print(a)
10
```

하지만 문자열의 경우 변환 하고자 하는 자료형이 아니면 오류가 납니다

```
>>> a = int( '10.5' )
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '10.5'
```

Say Something… 문자열은 왜 쓸까

프로그래머는 숫자로 이야기 하는 사람?

물론 수학적으로 계산 식을 세우고, 컴퓨터에 데이터를 적절하게 입력해서 원하는 값을 얻어내고, 처리 하는 것은 프로그래머의 역할 입니다.
자료구조와 알고리즘 등 최대한 효율적으로 컴퓨터에게 일을 시키기 위해서는 *수학이 필수적* 이죠.

하지만 이렇게 얻어낸 값을 의미 있는 컨텐츠로써 활용 할 수 있게 해주는 것 또한 프로그래머의 역할입니다.
대부분의 프로그래머는 의미 있는 컨텐츠를 사용자에게 보여주기 위해, 자료로써 활용 하기 위해 문자열과 많은 씨름을 합니다!

ㄱ,ㄴ,ㄷ,ㄹ, A, B, C, D … “안녕하세요!”, “Greeting~” 문자와 문자열의 차이점은 뭘까요?

문자열이란 단순하게 문자들을 순서대로 나열 한 것을 문자열 이라고 합니다. **a**는 그냥 문자 **a** 이지만 이러한 문자들을 나열하면 **Hello**가 됩니다.
파이썬 에서는 **문자는 지원하지 않고 문자열만 지원 합니다.**

따옴표를 활용하여 문자열을 만들자

이미 여러 번 봤지만, 문자열에는 따옴표 aka.인용부호 가 붙습니다.

큰따옴표 (“ Hello ”) 또는 작은 따옴표 (‘ Hello ’) 그리고 3개의 따옴표를 연속해서 (‘‘‘ Hello ’’’ 또는 “““ Hello ”””)를 사용합니다.

문자열을 표현하는 방식이 생각보다 많죠?

문자열을 최대한 쉽게 다루기 위함 입니다. 먼저 기본적인 문자열을 만들어 보겠습니다.

```
>>> sayHello = ‘ Hello ’
>>> print(sayHello)
Hello
>>> sayGoodBye = “ Good Bye ”
>>> print(sayGoodBye)
Good Bye
```

따옴표를 사용한 아주 간단한 문자열 만들어보기

문자열 안쪽에 작은 따옴표를 넣고 싶으면 다음과 같이 하면 됩니다.

```
>>> introMessage = "Hi! My name is 'Minho'"
>>> print(introMessage)
Hi! My name is 'Minho'
```

큰 따옴표는 어떻게 넣을까요? 작은 따옴표 안쪽에 큰 따옴표를 넣게 되면 표현 됩니다! **‘ Hi! My name is “Minho” ’** 를 입력 하고 실행 해보세요!

따옴표의 종류가 많은 이유가 보이시나요?

여러 라인을 한꺼번에 표현하기

개행(여러 줄) 표현하기.

문자열을 여러 줄로 표현 하고 싶을 때도 있을 겁니다. 이럴 때는 개행 문자(\n)를 활용 할 수도 있습니다.

```
>>> print('Hello\nWorld')
Hello
World
```

긴 문자열을 개행 문자를 사용해서 코드를 짜면 어떻게 될까요? 진짜 복잡해 보입니다.

```
>>> print('Bewhy – Forever\n래퍼 딱지를 떼는 중 이젠 MC로\n품 잡고 걸어가고 싶어 예술가의 길로\n확실히 단단해져 버린 내 신념과 Ego')
Bewhy – Forever
래퍼 딱지를 떼는 중 이젠 MC로
품 잡고 걸어가고 싶어 예술가의 길로
확실히 단단해져 버린 내 신념과 Ego
```

이럴 때 따옴표 3개를 연속으로 사용하면 개행이 쉽게 됩니다.

```
>>> print('''Bewhy - Forever
... 래퍼 딱지를 떼는 중 이젠 MC로
... 품 잡고 걸어가고 싶어 예술가의 길로
... 확실히 단단해져 버린 내 신념과 Ego''')
```

결과를 확인 해 보면 \n를 쓴 코드와 동일하게 작동하지만, 코드는 훨씬 더 깔끔하게 되는 것을 확인 할 수 있습니다. 각 줄 앞의 ...은 아직 입력 중이라는 뜻입니다.

'''를 사용하면 개행 문자 \n이 엔터를 입력해서 넘어간 곳마다 자동으로 들어갑니다

문자열에 특정한 효과주기! 이스케이프 문자

\ (백 슬래시) 문자를 이용해 문자열에 특정 효과를 줄 수 있습니다.

이전에 살펴 보았던 개행문자 \n도 이스케이프 문자 입니다. 문자열에 \n이 있으면 **문자열을 개행** 시킵니다.

```
>>>  print('Hello\nWorld')
Hello
World
```

\t는 tab 문자로써 tab을 입력 한 효과를 냅니다.

```
>>> print('\tHelloWorld')
        HelloWorld
>>> print('Hello\tWorld')
Hello   World
```

\를 이용해서 따옴표들을 따로 표시 할 수도 있습니다.

```
>>> print('bewhy : \"어나더어나더레벨얍얍얍 \"')
bewhy :  " 어나더어나더레벨얍얍얍 "
```

문자열 합치기, 복제하기, 문자 추출하기

+ 기호 는 문자열과 문자열을 이어주는 (append) 역할을 합니다.

```
>>> str1 = 'Hello' + ' World'
>>> print(str1)
Hello World
```

*** 기호 는 문자열을 특정 횟수 만큼 복제 합니다.**

```
>>> str1 = 'Hello' * 2
>>> print(str1)
HelloHello
```

*** [*n*] 기호는 *n* 번째 문자를 하나 추출 할 수 있습니다. 이 때 n을 오프셋(offset) 이라고 합니다.**

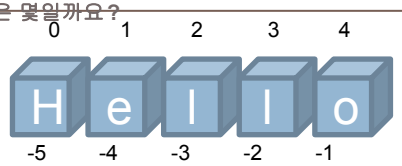
```
>>> str2 = 'Hello'
>>> print(str2[1])
e
>>> print(str2[-1])
o
```

[1] 이면 **e**가 아니고 **H** 아닌가요? 라는 생각을 가지 실 수도 있겠지만, 문자열 안의 문자들은 각각 **순서를 갖습니다.**

이 순서가 바로 오프셋 이 되는데, **가장 왼쪽의 오프셋은 0으로 시작하고, 가장 오른쪽의 오프셋은 -1입니다.**

따라서 str2[1] 은 두 번째 문자인 **e** 를 추출 하란 이야기가 되고, str2[-1]은 제일 오른쪽에 있는 문자인 **o**를 추출 하라는 이야기가 됩니다.

~~그렇다면 str[-3] 은 몇일까요?~~



원하는 부분에 있는 문자열만 추출하는 slice 기법 [*start* : *end* : *step*] □ [여기부터 : 여기까지 : 건너뛰면서]

slice는 자르다 라는 뜻입니다. *지정한 숫자(오프셋)만큼 잘라 낸다* 라고 생각하면 편합니다. 문법 부터 보시죠!

문자열에 사용 할 수 있는 [ ] 기호는 오프셋을 사용해서 특정 위치에 있는 **문자열**을 추출 하겠다는 이야기 입니다.

우리는 슬라이스 기법을 이용해서 **추출할 범위를 지정** 할 수 있습니다. 다음 예제는 처음부터 3번 오프셋 이전까지의 문자열을 추출합니다.

```
>>> str = '확실한 어나더 어나더 레벨 얍얍얍'
>>> print( str[:3] )
'확실한'
```

[:] □ 처음 부터 끝까지 추출하기 (원본 문자열의 형태 그대로를 사용하는 것과 같습니다)

[*start* :] □ **start**로 지정한 곳부터 전체 문자를 추출합니다.

[: *end*] □ 0번 오프셋(문자열의 처음) 부터 **end – 1** 까지 추출합니다.

[*start* : *end*] □ **start** 부터 **end – 1**까지 추출합니다.

[*start* : *end* : *step*] □ **start** 부터 **end -1** 까지 추출하는데, **step** 마다 출력합니다.

각각의 값은 생략 될 수 있으며, 생략 될 경우 다음과 같습니다.

각각의 오프셋 값은 생략 될 수 있으며, 생략 될 시 갖는 값은 다음과 같습니다.

start 값이 없으면? **start의 기본값은 0으로**

end 값이 없으면? **end의 기본값은 문자열의 총 길이(len)로,**

step값이 없으면? **step의 기본값은 1으로** 자동 지정 됩니다.

뭔가 많죠? 예제로 확인 해 볼게요!

start 만 있는 경우 [*start* : :] 또는 [*start* :]

```
>>> str = 'Well you done done me and you bet I felt it'
>>> str[5:]
'you done done me and you bet I felt it'
```

오프셋 5번 부터 출력 하기 'Well '다음부터
출력합니다

end만 있는 경우[: *end*] 또는 [: *end* :]

```
>>> str[ : 4]
'Well'
```

오프셋 3번까지 출력 하기 때문에(end – 1) 'Well' 이
나옵니다.

step만 있는 경우[: : *step*]

```
>>> str[ : : 4]
'W enedutf ' □ Well_you_done  done  me and you bet I felt_it
```

step으로 지정한 4번 째 문자마다
출력합니다.

뭔가 많죠? 예제로 확인 해 볼게요!

start와 end를 같이 사용해 보기

```
>>> str = 'Well you done done me and you bet I felt it'
>>> str[5:8]
'you'
```

오프셋 5번 부터 7번(8 – 1)까지 출력한 값이 'you' 가 출력 됩니다.

start와 end와 step을 같이 사용해 보기

```
>>> str[5:8:2]
'yu'
```

오프셋 5번 부터 7번(8 – 1)까지 출력한 값인 'you' 에서 2개 마다 출력 하기 때문에 'yu'만 출력 됩니다.

음수 오프셋을 사용하면?

```
>>> str[ :-5 :]
'Well you done done me and you bet I fe'
>>> str[-5:-1:]
'It i'
```

오프셋 지정은 언제나 작은 값에서 큰 값으로!

문자열에 사용 할 수 있는 각종 기능들

- 기능은 함수(function)라고 합니다

특정한 작업을 수행 하기 위한 코드를 함수(function) 라고 합니다. 파이썬은 문자열을 다루기 위한 여러 가지 함수들을 제공하고 있습니다.

len(*string*)	문자열의 길이를 세줍니다.
string.split(seq)	문자열을 seq(구분자)를 기준으로 쪼개서 리스트에 넣습니다.
string.join(*list*)	split과 정 반대로, 리스트에 있는 문자열을 합칩니다.
string.startswith(*word*)	문자열이 word로 시작하는지 검사합니다.(True, False)
string.endswith(*word*)	문자열이 word로 끝나는지 검사합니다.(True, False)
string.find(*word*)	문자열에 word가 있는 곳을 앞에서 부터 찾아서 해당 위치(오프셋)를 나타내 줍니다.
string.rfind(*word*)	문자열에 word가 있는 곳을 뒤에서 부터 찾아서 해당 위치(오프셋)를 나타내 줍니다.
string.count(*word*)	문자열에 word가 몇 회 등장 했는지 나타내 줍니다.
string.isalnum()	문자열이 모두 숫자로 이루어져 있는지 나타내 줍니다.

엄청 많지만, 자주 쓰다 보면 익숙해 집니다
- 연습을 많이 해야 해요

문자열을 다루기 위한 함수들이 정말 많습니다. 외워서 사용한다기 보다 익숙해 질 수 있도록 많은 연습이 필요합니다.

원하는 노래가사를 각종 함수를 이용해 실습해 봅시다.

```
>>> str = ""이러지도 못하는데 저러지도 못하네
... 그저 바라보며 ba-ba-ba-baby
... 매일 상상만 해 이름과 함께
... 쓱 말을 놨네 baby
        …생략…
... I'm like TT
... Just like TT
... Tell me that you'd be my baby"'
>>>str.count('TT')    # 'TT'가 등장한 횟수 세기
12
>>> str.startswith('이러지도')    # '이러지도' 로 시작하는지 검사하기
True
>>> str.endswith('your baby')    # 'your baby'로 끝나는지 검사하기
False
>>> str.endswith('my baby')    #'my baby'로 끝나는지 검사하기
True
>>> str.find('아직')    #'아직'이 처음 등장한 오프셋을 앞에서 부터 찾기
74
>>> str.rfind('Just like TT')    #'Just like TT'가 제일 마지막으로 등장한 오프셋 찾기
1118
>>> len(str)   # str의 전체 길이
1160
```

문자열 자르고 붙이기! split과 join!

split과 join 입니다.

split은 문자열을 특정 기준(seq)을 기준으로 문자열을 **잘라내서 list로 만들어** 주는 역할을 하고요,

join은 리스트로 되어 있는 문자열들을 특정 기준으로 **하나의 문자열로 만들어** 주는(합쳐 주는) 역할을 합니다.

리스트에 대한 이야기는 다음 장에서 자세히 진행 해 볼게요

```
>>> str = '''참 많은 시간이 흘러가고
... 넌 어떻게 사는지 참 궁금해
... 날 걱정하는 사람들에게
... 다 잊었단 거짓말하는
... 내가 참 미운 날'''
>>> myList = str.split('\n')      개행문자를 기준(sequence)로 하여 문자열을 잘라냅니다. 잘라낸 결과는 리스트로 반환됩니다.
>>> print(myList)
['참 많은 시간이 흘러가고', '넌 어떻게 사는지 참 궁금해', '날 걱정하는 사람들에게', '다 잊었단 거짓말 하는 ', '내가 참 미운 날']  개행문자를 기준으로 잘려진 문자열이 리스트에
    들어갑니다.
>>> '*'.join(myList)    '*' 를 기준으로 리스트에 있는 내용을 하나의 문자열로 만들어 줍니다.
'참 많은 시간이 흘러가고*넌 어떻게 사는지 참 궁금해*날 걱정하는 사람들에게*다 잊었단 거짓말하는 *내가 참 미운 날'
```

문자열을 효과적으로 다뤄 보겠습니다.

파이썬에는 보다시피 많은 문자열 함수가 존재합니다. 일반적인 문자열 함수를 이용해서 어떻게 동작하는지 살펴볼게요.

```
>>> like_it= '"이제 괜찮니 너무 힘들었잖아
... 우리 그 마무리가 고작 이별뿐인 건데
... 우린 참 어려웠어
... 잘 지낸다고 전해 들었어 가끔
... 벌써 참 좋은 사람
... 만나 잘 지내고 있어
... 굳이 내게 전하더라'"
>>> like_it[:15]  # 처음부터 15번째 문자까지 출력 합니다.
'이제 괜찮니 너무 힘들었잖아'
>>> like_it.startswith('이제')
True
>>> like_it.endswith('전하더라')
True
>>> word = '좋은'
>>> like_it.find(word)  # '좋은'이 최초로 등장한 위치
70
>>> like_it.rfind('우린 ' ) # '우린'이 마지막으로 등장한 위치
38
>>> like_it.replace('이제 괜찮니', '이젠 괜찮니 ') #"이제
    괜찮니"를 "이젠 괜찮니"로 변경하기
```

```
>>> zico_artist = 'We are we are we artist baby!!!!'
>>> zico_artist.strip('!') # 양쪽의 ! 문자열 삭제하기
'We are we are we artist baby'
>>> zico_artist.capitalize() # 제일 앞문자를 대문자로 만들기
'We are we are we artist baby!!!!'
>>> zico_artist.title() # 모든 단어의 첫 글자를 대문자로
    만들기
'We Are We Are We Artist Baby!!!!'
>>> zico_artist.upper() # 모든 글자들을 매누자로 만들기
'WE ARE WE ARE WE ARTIST BABY!!!!'
>>> zico_artist.lower() # 모든 글자들을 소문자로 만들기
'we are we are we artist baby!!!!'
>>> zico_artist.swapcase() # 모든 글자의 대 소문자 바꾸기
'wE ARE WE ARE WE ARTIST BABY!!!!'
>>> zico_artist.center(50) # 지정한 공간(50) 기준으로
    가운데 정렬
'        We are we are we artist baby!!!!         '
>>> zico_artist.ljust(50) # 지정한 공간 기준으로 왼쪽 정렬
'We are we are we artist baby!!!!                 '
>>> zico_artist.rjust(50) # 지장한 공간 기준으로 오른쪽 정렬
'                 We are we are we artist baby!!!!'
```

03. 데이터를 모아보자

파이썬의 재료 모으기

- 데이터를 효과적으로 모아주는 자료구조
- 순환에 대한 이야기들

2장에서는 재료에 대한 이야기를,
3장에서는 재료를 모으는 방법에 대한 이야기를

<2장> 에서 파이썬의 재료의 종류를 알아보았죠?

재료를 한가지만 쓰라는 법은 없습니다. 재료를 모아보는 방법에 대해서 이야기 해볼게요

① 재료란 단순히 데이터들 (자료형) 이고,

② 이러한 **재료들을 모으는 것**을 ***자료구조*** 라 합니다.

즉, 기본 타입으로 만들어진 변수나 데이터를 자르고 붙이면서 ~~복잡한 형태로~~ 결합 된다는 것을 의미합니다.

이 세상 모든 것들의 기본 단위는 원자(atom)과 분자(molecule)로 이루어져 있습니다.

원자들이 복잡한 형태로 모여서(결합 되어서) 분자가 되듯이, 이러한 분자들이 모여서 세상을 구성합니다.

즉, 원자와 분자는 이 세상을 구성하는 '기초' 라고 생각 할 수 있겠네요

마찬가지로 2장에서 배운 원초적인 **데이터 타입**들을 **원자**로 생각하고

3장에서 배울 **자료구조**를 **분자**로 생각합시다.

좌우지간 세상을 놓고 이야기 하던, 프로그램을 놓고 이야기 하던 **기초재료**라는 사실은 변함이 없네요.

파이썬의 존재하는 모든 것을 모아보자
- 리스트(List)와 튜플(Tuple)

문자열은 문자를 '모은 것' 이라는 이야기, 기억하시나요?

문자를 모아서 문자열을 만듭니다. 이 때 **문자열을 문자의 시퀀스(Sequence)** 라고 합니다. 시퀀스란 연속적으로 자료들이 모아진 것을 의미합니다

따라서 문자열에서 특정한 문자에 접근 하기 위해서 숫자(정수)를 사용하여 문자열의 항목에 접근 할 때는 **오프셋(Offset)**을 사용 하였죠.

지금부터 이야기 해볼 **리스트와 튜플은 파이썬에 존재하는 모든 객체들의 시퀀스**가 될 수 있습니다. (파이썬에 존재하는 어떤 객체든 모아 낼 수 있습니다.)

문자열에서와 마찬가지로 오프셋과 슬라이스 기법 등등을 사용 할 수 있을 것 같네요.

먼저 리스트와 튜플의 차이점에 대해서 이야기 해 보자면

리스트는?

모아놓은 자료의 변경이 가능 합니다. 모아놓은 자료들에 대해 변경, 추가, 삭제 등등이 가능 하다는 이야기가 됩니다.

이러한 특징을 **변경 가능(mutable)** 이라고 이야기 합니다.

튜플은?

리스트와 다르게 **모아놓은 자료의 변경이 불가능** 합니다. 한번 튜플에 들어간 자료들은 변경, 추가, 삭제 등등이 불가능 합니다.

이러한 특징을 **불변(immutable)** 이라고 합니다.

데이터를 마음대로 추가하고 삭제하자!
- 리스트(list)

먼저 리스트에 대해 이야기 해 봅시다.

리스트는 데이터를 순차적으로 파악하는데 굉장히 좋습니다. 특히 내용의 순서 자체가 바뀔 수 있다는 특징을 가지고 있습니다.
또한 문자열과 달리 리스트는 변경 가능한 자료형이기 때문에 **새로운 요소를 추가 하거나, 삭제하거나, 수정 할 수 있습니다.**
그리고 중요한 특징 중에 하나는 중복된 형태의 데이터를 저장 할 수 있습니다

먼저 리스트를 만드는 것부터 시작 해 보시죠.
[ ] 기호를 이용하거나 **list( )** 리스트 함수 를 이용하거나, 추후에 배울 **리스트 컴프리헨션(list comprehension)**을 이용해 만들어 낼 수 있습니다.
리스트 안에 추가된 데이터들을 **원소 또는 요소** 라고 합니다. 원소를 추가한 상태로 리스트를 만들고 싶으면 추가되는 원소를 쉼표(,) 로 구분 해 줍니다.

```
>>> empty_list = []     # 원소가 없는 비어있는 리스트 만들기
>>> pokemons = ['Picachu', 'Raichu‘, 'Picachu']   # ‘Picachu’ 2개 와 ‘Raichu’ 를 담아낸 리스트 만들기. 원소의 개수는? 3개!
>>> print(empty_list)
[]
>>> print(pokemons)
['Picachu', 'Raichu', 'Picachu']    # list는 중복된 원소를 저장 할 수 있습니다.
>>> another_empty_list = list()    # list 함수를 이용해서 비어있는 리스트 만들기
>>> print(another_empty_list)
>>> []
```

다른 데이터 타입을 list로 변환 할 수 있습니다.

list() 를 사용하면 다른 데이터 타입을 리스트 형태로 변환 시켜 줄 수 있습니다.

list() 는 함수, 객체 생성 등 여러 가지 의미로 생각 할 수 있는데, 함수나 객체에 대해서는 다음에 자세히 알아 봅니다. 지금은 간단히 '기능' 이라고만 알고 넘어가 보겠습니다. list()의 기능은 소괄호 ( ) 에 들어오는 내용을 리스트의 형태로 변환하는 기능을 합니다.

```
>>> list('cat')    #문자열 'cat'을 리스트로 바꾸기
['c', 'a', 't']    # 단순 문자열을 리스트로 바꾸면 한 글자씩 원소로 배치됩니다.

>>> my_tuple = ('hello','나는','튜플입니다')    # 튜플을 만들 때는 소괄호를 사용합니다. 다음 장에서 튜플에 대해 알아봅니다.
>>> list(my_tuple)    # 튜플인 my_tuple을 리스트로 바꾸기
['hello', '나는', '튜플입니다']    # 튜플이 리스트로 변환 되었네요

>>> list(10)    # 정수도 리스트로 바꿀 수 있을까? NO!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

정수는 리스트로 바꿀 수 없네요! 정수 뿐만 아니라 실수(float), 논리(Boolean) 자료형은 리스트로 바꿀 수 없습니다.
그 이유는 list는 iterable(연속적) 특징을 가진 자료들만 변환 시킬 수 있기 때문입니다. 즉, 연속적으로 데이터들이 나열 되어 있는 **Sequence** 만 변환이 가능합니다.

문자열 나눌 때 사용했던 split 기억하시죠?

문자열을 특정 기준(seperator)를 이용해 나누는 split은 문자열을 리스트에 담습니다.

이번 장에서는 split을 조금 더 알아보겠습니다.

```
>>> today = '2018-02-23'   # 문자열 ‘ 2018-02-23 ’을
>>> today.split('-')     # ‘-’ 를 기준으로 나눠서 리스트로 만들기
['2018', '02', '23']

>>> str = 'a/b//c/d///e'     # 구분자가 여러 개가 있다면
>>> str.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']     # 비어있는 문자열이 리스트에 들어갑니다.

>>> str.split(‘//')     #구분자를 두개 이상 쓰게 되면
['a/b', 'c/d', '/e']     # ‘ // ‘ 가 구분자가 되기 때문에 / 로는 나눠지지 않습니다.
```

리스트에서도 오프셋(Offset) 사용이 가능합니다.

문자열에서 사용했었던 오프셋을 리스트에서도 사용 가능합니다.

```
>>> singer=['Twice', 'Zion T', 'Tei']
>>> singer[0]
'Twice'
>>> singer[1]
'Zion T'
>>> singer[2]
'Tei'
>>> singer[-1]
'Tei'
```

리스트 안쪽에 리스트도 포함 될 수 있다?

다시 말하자면, 리스트는 다른 리스트까지 추가 시킬 수 있습니다.

파이썬에 존재하는 모든 것이 들어 갈 수 있기 때문에 리스트라고 해서 예외는 아니겠죠?

```
>>> boyGroup = ['빅뱅', 'BTS', 'iKon']
>>> girlGroup = ['Twice', '블랙핑크', '헬로 비너스']
>>> band = ['버즈', 'FT 아일랜드', '볼 빨간 사춘기']
>>> favoriteSinger = [ boyGroup, girlGroup, band, '제와피']    # favoriteSinger 리스트에 boyGroup, girlGroup, band, '제와피'(문자열)이 추가 되었습니다.
>>> print(favoriteSinger)
[['빅뱅', 'BTS', 'iKon'], ['Twice', '블랙핑크', '헬로 비너스'], ['버즈', 'FT 아일랜드', '볼 빨간 사춘기'], '제와피']
```

문제. favoriteSinger 에서 ' 블랙핑크 ' 를 추출하고 싶습니다. 오프셋 값을 어떠한 식으로 줘야 할까요?

리스트는 변경 가능한 mutable 자료구조

앞에서 설명했던 내용 중 리스트는 내부 원소가 변경 될 수 있는 mutable, 튜플은 변경이 불가능한 immutable 이라고 했었죠?

리스트에 오프셋을 이용하면 데이터를 추출 할 수도 있지만, 데이터를 변경 시킬 수도 있습니다.

또한 문자열에서 배웠었던 slice 기법도 사용 가능합니다.

```
>>> boyGroup = ['빅뱅',  'BTS', '철이와 미애']     # 철이와 미애를 좀 바꾸고 싶다... 철이와 미애의 현재 오프셋은 2
>>> print(boyGroup[2])
철이와 미애
>>> boyGroup[2] = 'WANNA ONE'     # 철이와 미애를 WANNA ONE 으로 바꿔주기
>>> print(boyGroup)
['빅뱅', 'BTS', 'WANNA ONE']    # 철이와 미애가 WANNA ONE으로 바뀌었네요
>>> boyGroup[3] = '하이라이트'    # 오프셋을 벗어난 곳에 값을 넣으면 오류가 발생 됩니다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> boyGroup[1:3]    # slice 기법도 사용 가능합니다.
['BTS', 'WANNA ONE']
>>> print(boyGroup[ : : -1 ])    # step 값을 -1로 주면 리스트가 뒤집힙니다.
['WANNA ONE', 'BTS', '빅뱅']
```

리스트에 자료 더하기
- append와 extend(+=)그리고 insert

이번엔 리스트에 아이템을 추가 하는 세가지 방법에 대해 알아봅니다.

1) append : 리스트의 끝에 원하는 원소를 추가한다.
2) extend : 다른 iterable 자료형 (list, tuple, str ...). 즉 Sequence형태의 자료를 덧붙여 확장(병합 – merge) 시킵니다.
3) insert : 원하는 오프셋에 원하는 항목을 추가 할 수 있습니다.

```
>>> girlGroup1=['핑클','SES','베이비복스']
>>> girlGroup1.append('샤크라')     # girlGroup1 맨 뒤에 '샤크라' 추가
>>> print(girlGroup1)
['핑클', 'SES', '베이비복스', '샤크라' ]

>>> girlGroup2=['소녀시대', '트와이스', 'I.O.I']
>>> girlGroup1.extend(girlGroup2)          # ※ 주의 ! girlGroup1과 girlGroup2를 병합한 결과를 내는 것이 아닌 girlGroup1에 girlGroup2를 병합합니다.
>>> print(girlGroup1)    # girlGroup2와 병합된 girlGroup1를 출력
['핑클', 'SES', '베이비복스', '샤크라', '소녀시대', '트와이스', 'I.O.I' ]

>>> girlGroup1 += ['A Pink', 'Miss A'] # += 기호를 이용해서도 병합(extend)를 할 수 있습니다.
>>> print(girlGroup1)
['핑클', 'SES', '베이비복스', '샤크라', '소녀시대', '트와이스', 'I.O.I', 'A Pink', 'Miss A' ]

>>> girlGroup1.insert(3, '시스타')
>>> print(girlGroup1)
['핑클', 'SES', '베이비복스', '시스타', '샤크라', '소녀시대', '트와이스', 'I.O.I', 'A Pink', 'Miss A']
```

리스트에서 원소 삭제 하기
- del 과 remove와 pop

이번엔 리스트에 서 원소을 제거 하는 세가지 방법에 대해 알아봅니다.

1) del : 오프셋을 이용해 해당하는 오프셋의 원소를 제거합니다. ※ del은 함수가 아닌 키워드 (구문) 입니다!!
2) remove : 원하는 원소 값을 넣어서 삭제 합니다.
3) pop : 원하는 오프셋에 있는 원소를 추출한 후 삭제합니다.

```
>>> idol = ['빅뱅','소녀시대','트와이스','iKon', '육각수','에이핑크' ,'철이와 미애']    #육각수랑 철이와 미애는... 아이돌이 아니죠
>>> del  idol[  -1 ] # 제일 뒤에 있는 '철이와 미애'를 삭제 합니다. 삭제 할 원소의 오프셋을 알고 있는 경우 유용합니다.
>>> print( idol )
['빅뱅', '소녀시대', '트와이스', 'iKon', '육각수', '에이핑크'

>>> idol.remove('육각수')   # 원소 값을 직접 집어 넣어 삭제 시킵니다. 삭제 할 원소의 데이터 값을 알고 있는 경우 유용합니다.
>>> print( idol )
['빅뱅', '소녀시대', '트와이스', 'iKon', '에이핑크']

>>> item1 = idol.pop() # pop에 인자가 없으면 제일 뒤에 있는 원소를 추출하고 삭제합니다
>>> print(item1)
에이핑크
>>> print(idol)
['빅뱅', '소녀시대', '트와이스', 'iKon']
```

실습. pop(1)과 pop(2) pop(-1)로 원소를 추출하고, 리스트가 어떻게 변화되었는지 확인하세요

만들어진 객체에 이름표 붙이기. 할당!

변수란 객체의 이름표에 불과 하다는 이야기 기억 하시나요?

객체에 변수의 이름을 지어주는 과정을 **할당** 이라고 한 것도 기억하시나요?

먼저 간단하게 리스트를 변수에 할당 해 보겠습니다. 1장에 나온 것 처럼 리스트 객체가 만들어 지고 myList라는 이름이 객체에 할당 될 것입니다.

```
>>> myList1 = [ 10, 20, 30
]
```

위의 그림에서 myList에 10,20,30 원소를 가진 list에 myList라는 이름이 붙어 있는 것을 확인 할 수 있습니다.
다음 코드를 보겠습니다. myList라고 이름을 붙인 객체에 myList2라는 이름이 또 붙을 수도 있습니다.

```
>>> myList2 = myList1
```

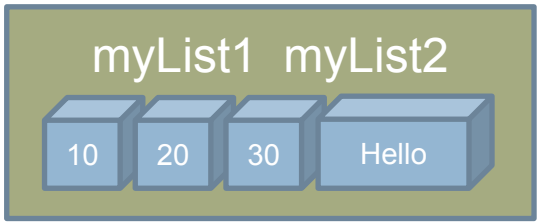
하나의 리스트 객체에 이름이 두 개가 붙었습니다! 그렇다면 myList1, myList2는 같은 객체가 할당 되었다 라고 볼 수 있겠네요

만들어진 객체에 이름표 붙이기. 할당!

하나의 리스트에 두 개의 이름표가 붙었습니다!

그렇다면 myList2를 이용해 리스트의 원소를 변경 하면 어떻게 될까요?

```
>>> myList2.append('Hello')
```



myList2에 문자열 Hello를 붙여보았습니다. 10,20,30 에 이어 Hello가 잘 추가 되는 것을 볼 수 있는데요, myList2에서 원소를 추가했지만, myList1을 출력 하면 어떻게 될까요?

```
>>> print(myList1)
```

```
[10,20,30,'Hello']
```

myList2 = myList1 코드가 일반적으로 생각하기엔 복사(copy) 라고 생각 할 수 있지만 실제는 myList1이란 이름이 붙은 list를 단순히 myList2 라고도
부를 수 있도록 해 주는 할당 과정입니다. **= 은 언제나 할당 이라는 사실을 잊지 마세요!**

할당과 복사는 다릅니다.

이전 예제는 할당이었죠? 이번에는 복사(copy) 입니다.

할당은 객체에 이름을 하나 더 붙여 주는 것이었죠? 복사는 말 그대로 **똑같은 객체를 하나 더** 만드는 것입니다!

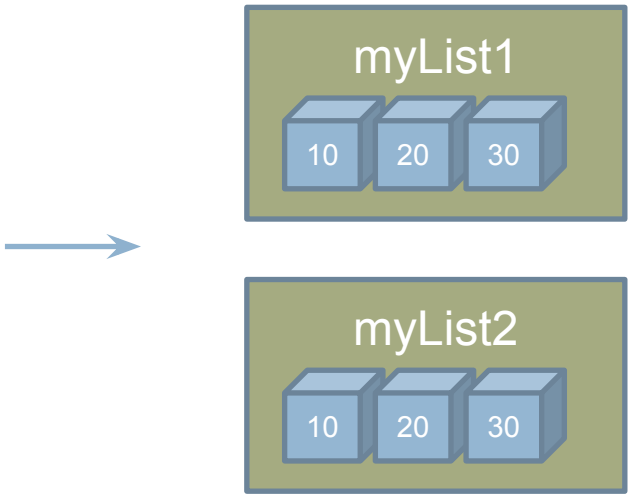
파이썬에서는 리스트를 복사하기 위해서 3가지 방식을 제공합니다. copy() 함수, [:] (슬라이스), list( ) 함수가 대표적입니다.

copy() 또는 list() 함수를 사용하면 리스트를 그대로 복사시켜서 새로운 리스트를 만들어 냅니다.

```
>>> myList1 = [10, 20, 30]
>>> myList2 =
        myList1.copy()
```

또는

```
>>> myList1 = [10, 20, 30]
>>> myList2 =
        list(myList1)
```



할당과 복사는 다릅니다.

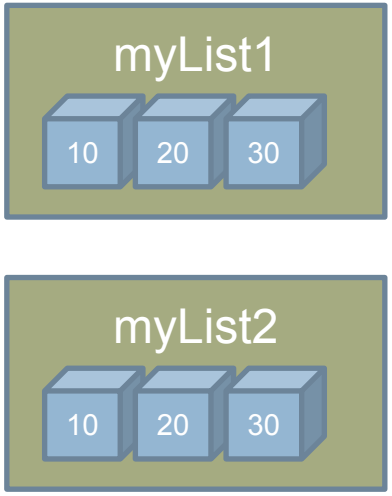
slice기법도 마찬가지 입니다. 새로운 리스트를 만들어 냅니다.

오프셋을 이용하여 잘라내는 역할을 하는 슬라이스 기법도 마찬가지로 새로운 리스트를 만들어 냅니다.

start, end, step 값을 지정하지 않으면 그대로 복사 되는 것을 확인 할 수 있습니다.

슬라이스 기법을 사용 할 때 start, end를 지정하지 않으면 똑같은 객체를 만들어 냅니다.
물론 start, end, step 값을 지정하면 원하는 대로 잘려진 새로운 리스트가 만들어 집니다.

```
>>> myList1 = [10, 20, 30]
>>> myList2 = myList1[ : ]
```



리스트와 튜플에 대한 이야기

리스트와 튜플은 단 하나의 차이점만 가지고 있습니다.

바로 리스트는 내부 원소의 변경이 가능하지만, 튜플은 불가능 하다는 것입니다.

예를 들어 여러분이 리스트에 어떠한 원소들을 저장 하고 있었는데, 변경의 위험이 없도록 안전하게 보호 하고 싶으면 튜플을 선택 할 수 있다는 뜻입니다.

튜플은 그럼 언제 쓸까요?

단순히 변경이 불가능한 (immutable) 형태의 자료구조를 만들고 싶을 때 사용 하면 됩니다. 예를 들어 함수의 인자로써 전달을 해야 한다거나, 리턴값을 여러개를 동시에 리턴 한다거나 하는 등등 어떠한 기능을 실행 함에 따라, 값을 다뤄야 할 때 우리는 튜플을 고민 해 볼 수있을 것입니다.

그럼 일반적으로 튜플을 많이 사용하나요?

튜플은 생각보다 제한적인 상황에서 사용됩니다. 바로 위에서 이야기 했지만, 튜플은 보통 함수의 매개변수 전달 또는 동시에 여러개의 값을 리턴 받고 싶을 때 많이 사용합니다..

키 : 값 쌍으로 저장하여 관리하는 딕셔너리

키 : 값 쌍으로 데이터를 관리 합니다

딕셔너리는 저장되는 항목의 순서를 따지지 않습니다. 즉, 입력한 순서가 딕셔너리에 순차적으로 저장 되어지지는 않는다는 이야기가 됩니다.

대신 딕셔너리에는 key 라는 것이 주어집니다. 이 때 key로써 활용 될 수 있는 자료형은 대부분 문자열 이지만 사실상 어떠한 자료형을 사용 하더라도 별로 상관은 없습니다.

또한 딕셔너리는 언제라도 key값을 이용해 데이터를 조회하거나 변경(삭제, 수정)이 가능합니다.

다른 언어에서는 파이썬의 딕셔너리를 연관배열, 해시, 해시맵 등등 이라고 하고 있습니다.

딕셔너리를 만들 때는 { }, dict()를 사용합니다.

{ key1 : value1, key2 : value2 ,.... } 같은 형태로 딕셔너리가 만들어 집니다.

```
>>> person_info = {  # 중괄호를 열고 닫아서 딕셔너리를 만들 수 있다!
... "name" : "mhso",   # key :  'name',  value : 'mhso'
... "age"  : 31
... }
>>> person_info  # 딕셔너리의 이름을 입력하면 모든 키와 값을 출력 할 수 있습니다.
{'name': 'mhso', 'age': 31}
>>>
```

다른 자료형을 딕셔녀리로 변환 시켜 볼 수도 있습니다. 단, 변환 가능한 형태는 정해져 있습니다.

리스트 또는 튜플을 딕셔너리로 바꾸고자 할 때는 dict() 함수를 사용 할 수 있습니다. 단, 키 : 값 형태를 유지 해야 하기에

리스트 또는 튜플의 원소가 두개 일 때만 가능 합니다.

```
>>> lol = [['a','b'],['c','d'],['e','f']]
>>> lol
[['a', 'b'], ['c', 'd'], ['e', 'f']]
>>> dict(lol)  # lol 리스트를 dict로 변환 시키기
{'a': 'b', 'c': 'd', 'e': 'f'}
>>>
```

딕셔너리 항목 변경은 자연스럽게!

딕셔너리의 가장 큰 특징 중 하나는 '키'의 개념을 이용 한다는 것입니다.

키가 있으면 수정, 키가 없으면 추가 합니다.

```
>>> test_dict = {'a':'hello'}   # test_dict에는 최초에 a : hello 만 존재 했지만
>>> test_dict['b'] = 'bye '    # 'b' 라는 키값이 추가되면서 bye라는 문자열이 값으로 등록 되었습니다. b : bye 데이터가 등록됩니다.
>>> test_dict['a'] = 'greeting~ '    # 원래 존재 했었는 a라는 키의 값을 greeting~ 으로 바꿔줍니다. 이제 키 a 의 값은 더 이상 hello가 아닌 greeting~ 이 될 것입니다.
>>> test_dict
{'a': 'greeting~', 'b': 'bye'}
>>>
```

딕셔너리의 결합을 할 때는 update()를 사용합니다.

딕셔너리를 결합 하는 방법에 대해 알아보겠습니다. 정말 쉽습니다.

그냥 update함수만 사용하면 딕셔너리와 딕셔너리를 결합할 수 있습니다.

```
>>> dict_a = { 'a':'b','c':'d'}
>>> dict_b = { 'e':'f', 'g':'h' }
>>> dict_a.update(dict_b)
>>> dict_a
{'a': 'b', 'c': 'd', 'e': 'f', 'g': 'h'}
>>>
```

만약 키값이 중복 되어 있다면 어떻게 될까요?

합쳐지는 딕셔너리가 승리합니다. (두 번째 딕셔너리)

```
>>> dict_a = { 'a':'b', 'c':'d' }
>>> dict_b = { 'a':'hello', 'e':'f' }
>>> dict_a.update(dict_b)
>>> dict_a
{'a': 'hello', 'c': 'd', 'e': 'f'}
>>>
```

딕셔너리의 여러가지 기능들...

in 키워드를 활용하면 키가 존재 하는지 알아 낼 수 있습니다.

```
>>> myDict = { 'a' : 'b', 'c' : 'd' }
>>> 'a' in myDict
True
>>> 'f' in myDict
False
```

가장 많이 딕셔너리가 사용되는 일반적인 용도 입니다. **key** 를 이용해 값을 참조(조회) 합니다.

```
>>> myDict['a']
'b'
```

모든 **key**를 얻어 내고 싶을 때는 **keys()**를, 모든 **value**를 얻어 내고 싶을 때는 **values()** 를 활용 할 수 있습니다.

참고로 **items()**는 딕셔너리 내의 모든 키, 값을 튜플 리스트로 만들어 냅니다. 모든 결과는 **list()**를 이용해 리스트로 만들 수 있습니다.

```
>>> myDict.keys()
dict_keys(['a', 'c'])
>>> myDict.values()
dict_values(['b', 'd'])
>>> myDict.items()
dict_items([('a', 'b'), ('c', 'd')])
```

딕셔너리 삭제도 알아 보죠

del 키워드를 활용하면 손쉽게 딕셔너리의 원소를 삭제 할 수 있습니다.

```
>>> del dict_a['a']
>>> dict_a
{'c': 'd', 'e': 'f'}
>>>
```

전체 삭제는 **clear**를 사용합니다.

```
>>> dict_a.clear()
>>> dict_a
{}
```

집합 표현하기. set에 대해 알아보겠습니다.

집합의 특징 중 하나는 값이 중복되지 않는 다는 점입니다. 또한 값을 입력 할 때의 순서도 유지 되지 않죠

따라서 set을 사용 하겠다는 이야기는 중복되지 않는 유일한 데이터셋을 사용 하겠다는 이야기 입니다.

물론 딕셔너리의 key값도 중복을 허용 하지 않기 때문에 유일한 key : value를 가진 다고 생각 할 수 있겠네요.

셋은 딕셔너리에서 value가 없는 형태로 만들어 주면 됩니다

```
>>> mySet = {1,2,3,4,4,5,6}  # 4 가 두번 들어갔지만 한번만 추가 됩니다
>>> mySet
{1, 2, 3, 4, 5, 6}
```

set() 함수를 이용해 시퀀스를 set으로 만들어 낼 수도 있습니다.

```
>>> set('letters')
{'s', 'r', 't', 'l', 'e'}
>>> set([1,1,2,3,4,4,5])
{1, 2, 3, 4, 5}
>>>
```

04. 파이썬의 코드 작성

간단한 파이썬 코드 작성 방법

- 컴퓨터에게 말 하는 방법 들
- 파이썬 코드를 파이썬 답게 만들어 주기

다른언어와는 다른 파이썬의 코드구조

파이썬엔 중괄호 ({ } – 코드블럭) 및 세미콜론(;)이 없다?

파이썬 코드의 철학은 간결한 코드입니다

① 다른언어들은 중괄호 및 세미콜론을 활용하여 코드의 시작과 끝을 나타냅니다

이러다 보니 코딩하기 싫어지죠... 예외처리 나쁜놈

② 중괄호와 세미콜론이 없고 **들여쓰기(indent)로 코드를 구분**하게 됩니다.

java를 사용한 코드 구조

```
int a = 10;
if ( a == 10 ) {
    System.out.println("a는 10입니다");
}else if( a < 10 ){
    System.out.println("a는 10보다 작아요);
}else{
    System.out.println("a는 10보다 작거나 같지 않습니다.");
}
```

python을 사용한 코드 구조

```
a = 10
if a is 10 :
    print("a는 10입니다.")
elif a < 10 :
    print("a는 10보다 작습니다.")
else :
    print("a는 10보다 작거나 같지 않습니다.")
```

컴퓨터에게 문장으로 이야기 하자.

마치 컴퓨터에게 말을 하듯이 코딩하자. 좀 더 복잡하지만 쉽게

나. 먹는다. 밥. 맛있는. 저녁. 소고기. 오늘 밤에

위의 문장을 읽으면 말은 됩니다. 조금 이상할 뿐이지만요. 단순하고 쉬운 단어와 문장들의 나열이지만 말은 됩니다.
우리가 코드구조를 잘 모르고 코딩 하는것은 컴퓨터에게 위 처럼 이야기 하는 것과 마찬 가지 입니다.

나 오늘 밤에 저녁밥으로 맛있는 소고기 먹는다.

당연히 위 문장 처럼 말하는게 훨씬 낫겠죠? 여러가지 단어와 문장들을 유려하고 효율적으로 코딩을 하는 것이 당연히 당연히 좋습니다.

지금부터 효과적인 코드 구조를 작성 하기 위해서 **조건** 이라는 것 부터 이야기 해보겠습니다.

조건을 만들어내는 연산자들

조건을 만들어 내기 위해서 우리는 몇가지 연산자를 사용 할 수 있습니다!

간단한 초등학교 때 배웠던 지식을 이용하여 우리는 여러가지 조건들을 만들어서 활용 할 수 있습니다.

예를 들어 변수 a이 10이냐? a 가 10보다 작냐? a가 음수냐? 등등 같은 것들이죠. **당연히 결과는 bool 형태로 등장 하겠죠?**

연산자	의미	예시 (a = 1, b = 2 라고 가정)
== (is)	같다.	a==b 또는 a is b # a 와 b 가 같다 □ False
!= (is not)	다르다(같지 않다)	a!=b 또는 a is not b # a와 b 가 다르다 □ True
<	작다	a < b # a가 b 보다 작다 □ True
>	크다	a > b # a가 b 보다 크다 □ False
<=	작거나 같다	a <= b # a가 b 보다 작거나 같다 □ True
>=	크거나 같다	a >= b # a가 b 보다 크거나 같다 □ False
and	그리고	a == 1 and b <= 2 # a가 1과 같고 b가 2보다 작거나 같은가? □ True
or	또는	a > 2 or b == 2 # a가 2보다 크거나 b가 2와 같은가? □ True
in	포함한다 (자료구조 전용)	1 in [a, b] # a, b 를 원소로 가진 리스트에 1이 포함된다. □ True

데이터의 존재 여부도 조건이다?

파이썬의 조건은 꼭 bool 일 필요는 없습니다!

컴퓨터에게 질문을 하기 위해서 우리는 bool 값을 활용한 여러가지 조건을 만들어 낼 수 있습니다만

특이하게 파이썬에서는 데이터의 존재 유무도 조건이 될 수 있습니다. 데이터가 존재하면 True, 데이터가 존재하지 않으면 False 입니다.

다음 표는 자료가 False 로 인식 되는지에 대한 표 입니다.

요소	False
null	None
int	0
float	0.0
string	''
list	[ ]
tuple	( )
dict	{ }
set	set()

만약에 ~하면 ~하고, 그게 아니고 ~하면 ~해 그것도 아니면 ~해

조건과 간단한 조건문

~하면, ~라면, ~할 때까지 등등... 조건을 의미합니다.

먼저 조건이란?

① 컴퓨터에게 질문을 한다 라고 생각합시다. 예를 들어 변수 str에 들어가 있는 값이 'Hello'야? 또는 str 에 제대로 값이 들어있니? 등등

② 컴퓨터에게 물어볼 질문이 맞다면 참(True), 틀리면 거짓(False)이 됩니다. 즉 bool 변수를 사용하는 것이죠.

사람이 뭔가 컴퓨터에게 질문을 하면 항상 맞다! 아니다! 로 대답이 들어옵니다.

애매 하게 ~할 수도 있고 아닐 수도 있어요. 라고 이야기 하지는 않는 다는 것이죠. 여기서 이 질문을 우리는 **조건** 이라고 이야기 할 것입니다.

이런 식으로 어떠한 **조건이 참(True) 일 때 특정한 문장을 실행 하게 하는 것을 조건문** 이라고 합니다.

본격적인 조건문에 대해 알아보겠습니다. 파이썬은 들여쓰기로 실행할 문장을 따로 구분 합니다.

```
>>>a = 5
>>>if a == 5:
...   print("Hello")
... elif a>5:
...   print("Bye")
... else:
...   print("Greeting~")
Hello
```

코드를 한글로 해석하면

```
a 변수에 5를 대입해줘
만약에 a가 5라면 □ 조건
    Hello 를 출력해
그게 아니고 a가 5보다 크다면 □ 조건
    Bye를 출력해
그게 아니면
    Gretting~ 을 출력해
Hello
```

조건문을 파헤쳐 봅시다. if, elif , else

1) if : ~ 하면

if 문장은 조건문의 시작을 알립니다. 어떠한 데이터를 이용해서 조건문을 구성 하고자 할 때 사용하면 됩니다.
뜬금없이 그게 아니고 ~라면 ~~해! 라고 이야기 할 순 없겠죠???

2) elif (else if) : 그게 아니고 ~ 하면

elif는 else if 의 줄임말 입니다. if 에서 조건이 맞지 않고 계속 이어서 조건을 검사 하고 싶을 때 elif를 사용하면 됩니다.
if에서 이야기 했지만 갑자기 뜬금없이 그게 아니고 ~ 로 문장이 구성되면 조금은 이상해지겠죠??
또한 elif는 여러번 사용 될 수 있습니다. 조건검사를 여러번 해야 하는 경우라면요!
계속 그게 아니고 ~ 면, 그게 아니고 ~면 ... 이라고 계속 이야기 하는 것과 같습니다.

3) else : 그게 아니면

특이하게 else 에는 조건이 없습니다. 사실 없어야 된다고 표현해야 더 적절한 표현이 되겠네요.
'그게 아니면' 이라는 문장 자체가 위에 있었던 if나 elif 의 조건이 모두 맞지 않았을 때 사용하기 때문입니다.
위에 존재하는 모든 조건이 맞지 않으면 실행됩니다.

어떤 일이 일어날 때 까지 특정 코드를 반복 해줘
반복(loop)할 때 사용하는 while

if, elif, else는 위에서 아래로 한번만 실행 합니다. 여러번 반복 하고 싶을 때는 반복문(loop)가 필요합니다.

이번에 배워 볼 반복문은 **~ 할 때 까지 반복해줘** 라고 이야기 하는 while 문입니다. while 문의 구성은 다음과 같습니다.

while 조건 :

  # 실행 문장

while문의 조건이 True 일 때 아래에 있는 실행 문장을 실행 합니다. 실행 문장이 모두 실행 되면 다시 조건 검사를 진행 하게 됩니다.

이렇게 조건이 True 일 때 반복을 하다가 조건이 False가 되면 반복을 중지합니다.

```
>>> count = 1
>>> while count<= 5:
...     print(count)
...     count += 1

1
2
3
4
5
```

코드를 한글로 해석하면

```
count에 1을 대입해줘
count 가 5이하면 반복하자 □ count  가 5보다 커지면 (6이상) 반복이 안되겠죠?
    count를 출력해줘
    count의 값을 1 증가시켜줘
1
2
3
4
5
```

실습 하기 : count 가 짝수면 *짝수 : count*로 출력, 홀수면 *홀수 : count*로 출력하기

반복을 중단합니다.

어떤 일이 일어 날 때 까지 반복 할 수 있지만, 어떠한 조건일 때는 멈추고 싶을 때도 있을 것 같습니다.

input() 함수를 사용하면 키보드에서 엔터를 칠 때 까지 입력한 내용을 읽어 올 수 있습니다.

사용자가 q를 입력 하면 더 이상 반복을 하지 않고 중단 하는 코드 입니다.

파이썬 에서는 반복을 중지 하고 싶을 때 **break** 키워드를 사용합니다.

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
...     if stuff is 'q':
...             break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
```

코드를 한글로 해석하면

```
계속 반복 하자
    stuff 변수에 키보드를 통해서 입력 받을 거야
    만약에 stuff가 q 라면
        반복을 중지해
    stuff의 맨 앞글자를 대문자화 시켜서 출력해줘

String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
```

break 키워드를 사용 할 때는 조건문과 함께 사용 하는 것이 좋습니다.

주로 **~ 할 때 반복을 멈추겠다!** 라고 이야기 하기 때문이죠.

break와 다음에 할 continue는 항상 **반복문에서만 사용** 할 수 있습니다

이번 반복은 건너 뛰고 계속 이어서 하시죠

반복을 중단 하기는 싫지만 어떠한 이유로 다음 반복을 이어서 진행 하고 싶을 때도 있을 것입니다.

break 예제와 마찬가지로 input() 함수를 이용해 사용자가 q를 입력하면 반복을 중지 하긴 하지만,
사용자가 홀수를 입력하면 제곱을 하고, 짝수를 입력하면 무시하고 다음 입력을 받을 수 있는 코드 입니다.
현재 반복을 중단 하고 다시 처음으로 돌아가 조건 검사부터 이어서 반복을 하고 싶을 때는 **continue** 키워드를 사용합니다.

```
>>> while True:
...     value = input("Integer, please [ q to quit ]: ")
...     if value is 'q':   # 종료
...             break
...     number = int(value)
...     if number % 2 == 0:    # 짝수검사
...             continue    # 조건 검사부터 다시 반복 시작하기
...     print(number, "Squared is", number * number)
...
Integer, please [ q to quit ]: 1
1 Squared is 1
Integer, please [ q to quit ]: 2
Integer, please [ q to quit ]: 3
3 Squared is 9
Integer, please [ q to quit ]: 4
Integer, please [ q to quit ]: 5
5 Squared is 25
Integer, please [ q to quit ]: q
>>>
```

코드를 한글로 해석하면

```
계속 반복 하자
    value 변수에 키보드를 통해서 입력 받을 거야
    만약에 value가 q 라면
        반복을 중지해
    입력 받은 value 를 정수로 만들어서 number에 대입해줘
    만약에 number와 2를 나눈 나머지가 0 이라면
        이번 반복은 중지 하고 다시 처음부터( 조건 검사부터) 시작하자
    입력 받은 숫자와 제곱수를 출력해줘

Integer, please [ q to quit ]: 1
1 Squared is 1
Integer, please [ q to quit ]: 2
Integer, please [ q to quit ]: 3
3 Squared is 9
Integer, please [ q to quit ]: 4
Integer, please [ q to quit ]: 5
5 Squared is 25
Integer, please [ q to quit ]: q
>>>
```

데이터를 하나씩 꺼내봅시다.

모여있는 자료를 하나씩 참조 가능하다라는 뜻의 *순회 가능 하다*는 말인 iterable. 기억 나시나요?

파이썬의 자료구조인 문자열, 리스트, 튜플, 셋, 딕셔너리는 데이터를 모아놓는 자료형입니다. 공통점은 전부 *시퀀스* 라는 사실!

Iterator란, 자료구조에 있는 데이터를 하나씩 꺼낼 수 있는 파이썬의 도구라고 생각 해 보시면 됩니다. – 사실 코드에 등장하진 않습니다. 어려워서..

지금부터 알아볼 for 구문은 내부적으로 이터레이터를 사용해 더 이상 **자료구조에서 꺼낼 데이터가 없을 때까지 반복** 합니다.

```
for item in iterable_object :
    # 실행 코드
```

while 문보다 조금 복잡한데요, for 문에서 사용하는 내용은 다음과 같습니다.

1) iterable_object : 순회 가능한 객체로써 문자열, 리스트, 튜플, 셋, 딕셔너리 등 자료구조를 의미한다.
2) item : iterator 에 의해 자료구조에서 하나씩 꺼내어진 원소를 의미한다.

결론을 내자면, 자료구조인 iterable_object 안쪽에 있는 원소들을 하나씩 빼서 item 변수로 활용 한다고 생각하시면 쉽습니다.

for문은 당연히 더 이상 꺼낼 데이터(원소)가 없으면 반복을 중단하겠죠?

for문은 이렇게 사용하시면 됩니다. #1
- list, tuple, string, set 에서 사용하기

예제로 for문의 다양한 사용 방법을 알아 보겠습니다.

리스트 또는 튜플 순회하기

(예제는 리스트지만, 튜플, 셋도 똑같이 작동합니다.)

```
>>> coffee_list = ['아메리카노','카페라떼','헤이즐넛']
>>> for coffee in coffee_list:
...     print(coffee)
...
아메리카노
카페라떼
헤이즐넛
```

코드를 한글로 해석하면

```
coffee_list 에 아메리카노, 카페라떼, 헤이즐넛이 들어있는 리스트를 할당해줘
coffee_list에 있는 커피를 하나씩 꺼내서 coffee에 넣어줘, 없을 때까지 반복
    할거야
    coffee를 출력 해줘


아메리카노
카페라떼
헤이즐넛
```

문자열 순회도 가능합니다!

```
for a in 'coffee':
...     print(a)
...
c
o
f
f
e
e
```

set도 마찬가지로 같은 원리 입니다

```
>>> singer_set = set()
>>> singer_set.add('비와이')
>>> singer_set.add('트와이스')
>>> singer_set.add('Bruno Mars')
>>> singer_set.add('Bruno Mars') # 두 번 추가 되었지만 set이니까 한번만 추가
    됐겠죠?
>>> for singer in singer_set:
...     print(singer)
Bruno Mars
트와이스
비와이
```

for문은 이렇게 사용하시면 됩니다. #2
- dict 에서 사용하기

dict의 순회 입니다. 다른 자료구조에 비해 조금 복잡하지만...

1) 딕셔녀리의 key를 순회합니다.

.keys()로 키만 가져 올 수 있습니다.

```
>>> singer_dict = {'BrunoMars' : 'Just The Way You
      Are',
                    'Json Mraz':'I\'m Yours‘}
>>> key_singer = singer_dict.keys()
>>> for singer in key_singer:
...     print(singer)
...
BrunoMars
Json Mraz
```

2) 딕셔녀리의 value를 순회합니다.

.values()로 밸류만 가져 올 수 있습니다.

```
>>> value_songs = singer_dict.values()
>>> for song in value_songs:
...     print(song)
...
Just The Way You Are
I'm Yours
```

3) 딕셔너리의 원소를 각각 튜플화 해서 가져 올 수도 있습니다.

items()로 key와 value를 각각 튜플화 할 수 있습니다.

```
>>> all_items = singer_dict.items()
>>> for item in all_items:
...     print(item)
...
('BrunoMars', 'Just The Way You Are')
('Json Mraz', "I'm Yours")
```

4) 튜플화 시킨 key, value를 언패킹 할 수 있겠죠?

```
>>> for singer, song in all_items:
...     print('가수 :', singer, ‘ / 노래 :', song)
...
가수 : BrunoMars / 노래 : Just The Way You Are
가수 : Json Mraz / 노래 : I'm Yours
```

여러 시퀀스 순회 하기 – zip 사용하기

여러 개의 시퀀스를 동시에 순회 해야 할 일도 있을 겁니다. 이 때는 **zip()** 을 사용 할 수 있습니다.

```
>>> meals = ['아침밥','점심밥','저녁밥','야식']  # meals 는 길이가 4
>>> foods = ['샌드위치','김가네','강남불백']    # foods는 길이가 3
>>> for meal, food in zip(meals, foods): # meals, foods 를 하나로 묶습니다.
...     print('오늘', meal,'은', food, '먹었어요')
...
오늘 아침밥 은 샌드위치 먹었어요
오늘 점심밥 은 김가네 먹었어요
오늘 저녁밥 은 강남불백 먹었어요
```

결과를 확인 해보면 **meals** 의 마지막 원소인 '야식'에 대한 순환은 진행 하지 않습니다.

zip() 으로 합쳐서 순환 시킬 때 주의 할 점은 제일 작은 길이(**len**)를 가진 시퀀스가 기준이 된다는 것입니다.

또한 **zip** 함수를 사용하면 서로 다른 시퀀스를 쉽게 합쳐낼 수 있습니다.

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> korean = '월요일', '화요일', '수요일'
>>> list(zip(english,korean))          # 두 개의 튜플을 묶어서 리스트의 원소로 담기
[('Monday', '월요일'), ('Tuesday', '화요일'), ('Wednesday', '수요일')]
>>> dict(zip(english, korean))         # 두 개의 튜플을 묶어 각각 딕셔너리의 key와 value로 묶어주기
{'Monday': '월요일', 'Tuesday': '화요일', 'Wednesday': '수요일'}
```

숫자로만 이루어진 시퀀스 생성하기 – range()

지금까지는 일반적인 자료구조 시퀀스를 활용했지만, 특정 범위의 숫자에 대한 시퀀스 또한 range로 만들어 낼 수 있습니다.

range() 함수는 슬라이스 기법을 사용하는 것과 비슷합니다. *range(start, stop, step)* 으로 이루어져 있습니다.

시퀀스와 똑같이 생각하면 됩니다! start 부터 시작하고, stop-1에서 끝나고, step 만큼 건너 뛰면서 숫자 시퀀스를 만들어 냅니다.

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> for x in range(2, -1, -1):
...     print(x)
...
2
1
0
>>> list( range (2, -1, -1) ) # range로 숫자 리스트 만들기
[2, 1, 0]
>>> list( range(0,11,2))
[0, 2, 4, 6, 8, 10]
```

컴프리헨션(comprehension) 사용하기 - 리스트 컴프리헨션 사용하기

조금 더 파이썬답게, 그리고 콤팩트 하게 간지 나게 **자료구조를 생성하는 방법입니다.**

리스트, 튜플, 딕셔너리, 셋 등에 들어가는 원소들을 구성 할 때 for 문법을 사용해 for 문의 결과로 자료구조의 원소를 채웁니다.

1~5 까지의 숫자 리스트를 만들고 싶으면 *list(range(1,6))* 같은 코드를 이용해서도 만들 수가 있지만, 컴프리헨션을 활용하면 다음과 같이 됩니다.

```
>>> number_list = [ number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

먼저, 컴프리헨션 작성문법은 **[*expr* for *item* in *iterable_object*]** 입니다.

1) expr : 리스트의 원소에 넣을 원소에 대한 표현식
2) item : iterable_object 에서 꺼낸 아이템
3) iterable_object : 순환 가능한 시퀀스

그냥 list(range(1,6)) 쓰는 게 낫지 않을까요? 더 어려워 보이는데…

네, 맞습니다. 적어도 위의 예제 같은 경우는요. 컴프리헨션의 강력함은 **조건을 추가 할 수 있다** 라는 것입니다!

조건을 이용한 컴프리헨션의 문법은 다음과 같습니다.

[***expr*** **for** ***item*** **in** ***iterable_object*** **if** ***condition***]

if가 추가 됐네요. condition이 True 일 때만 순회중인 item을 리스트에 추가 시킬 수 있습니다.

사실 컴프리헨션을 사용하는 가장 큰 이유중의 하나가 조건을 넣을 수 있기 때문이라고도 할 수 있습니다.

다음 예제 부터 컴프리헨션에 각종 조건을 넣어서 사용해 보겠습니다.

컴프리헨션에 조건 추가하기!
- 원하는 아이템만 추가하자.

먼저, 1~100 에서 3의 배수를 구해서 리스트에 집어 넣는 예제로 가정해 보겠습니다.

3의 배수이기 때문에 *3으로 나눴을 때의 나머지가 0* 조건이 되겠죠? 이 될 것 같네요. 컴프리헨션을 사용하지 않고 만든 코드 입니다.

```
>>> divisor_three = []
>>> for x in range(1,101):
...     if x % 3 is 0:
...         divisor_three.append(x)
...
>>> divisor_three
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

이번엔 컴프리헨션을 사용한 방법입니다.

```
>>> divisor_three = [ x for x in range(1,101) if x % 3 is 0] # 1~100을 순회하며 변수 x에 할당하고, 이 x 가 3으로 나눴을 때 나머지가 0이면 리스트에 추가한다.
>>> divisor_three
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

컴프리헨션을 사용하면 여러가지 **조건에 따라서** 원하는 자료구조를 손쉽게 만들어 낼 수 있습니다.

컴프리헨션 문법 순서를 다음과 같이 생각하고 만드시면 조금 쉽습니다. 위의 예를 한번 들어 보겠습니다.

[x for x in ①**range(1,101)** if x % 3 is 0] □ **① 범위 생각하기. 여기서는 "1~100을 순회 하며 " 가 되겠네요**
[x for ②**x** in range(1,101) if x % 3 is 0] □ **② 순회 생각하기. ① 에서 순회중인 숫자가 x에 들어 갈 것입니다.**
[x for x in range(1,101) ③**if x % 3 is 0**] □ **③ 조건 생각하기. ② 원소가 리스트에 포함 될지를 결정 해 줍니다.**
[④**x** for x in range(1,101) if x % 3 is 0] □ **④ 원소 생각하기. ③ 조건에 의해 리스트의 원소가 추가될 것입니다.**

① 1~100을 순회하며 ②변수 x에 할당하고, ③이 x 가 3으로 나눴을 때 나머지가 0이면 ④리스트에 추가한다.

중첩된 반복문을 가진 컴프리헨션

일반적으로 중첩된 루프를 이용해서 다수의 시퀀스를 차례로 순회 할 수 있습니다. 다음 그림의 전체 좌표를 찍어보는 예제를 만들어 봅시다.

	1	2
1	(1,1)	(1,2)
2	(2,1)	(2,2)
3	(3,1)	(3,2)

컴프리헨션 미사용

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...             print(row,col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

컴프리헨션 사용 + 튜플로 만들기

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col  in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

컴프리헨션 반복을 중첩되게 사용 하였습니다. rows가 한번 순회 하면서 안쪽에 있는 cols가 안쪽에서 순회 한다고 생각하면 쉽습니다.

딕셔너리 컴프리헨션

리스트 컴프리헨션은 대괄호 [ ] 를 사용해서 컴프리헨션을 진행 했습니다. 딕셔너리도 마찬가지로 중괄호 { } 를 이용해서 만들어 낼 수 있어요!

{ *key expr* : *value expr* for *expr* in *iterable_object* if *condition* }

리스트 컴프리헨션과 달라진 점은 딕셔너리 컴프리헨션은 제일 앞에 키 표현식(key expr) 이 있는 것 뿐입니다.

문자열 알파벳이 각각 몇 개가 있는지 세어보고 그 결과를 저장하는 딕셔너리를 만들어 보겠습니다. Mark Ronson – Uptown Funk 가사입니다.

```
>>> uptown_funk = ''' This hit, that ice cold
      Michelle Pfeiffer, that white gold
      This one, for them hood girls
      Them good girls, straight masterpieces
            … 생략 …
      Uptown Funk you up, Uptown Funk you up
      Uptown Funk you up, Uptown Funk you up (say whaa?!)
      Uptown Funk you up'''

#방법 1
>>> letter_counts = { letter : uptown_funk.count(letter) for letter in uptown_funk }
#방법2
>>> letter_counts = { letter: uptown_funk.count(letter) for letter in set(uptown_funk )}
```

방법 1은 uptown_func_coun에 word 자체가 가지고 있는 알파벳을 세어줍니다. 하지만 uptown_funk 문자열을 이루고 있는 중복된 알파벳이 많기 때문에

방법 2를 사용해서 알파벳 집합(set)을 미리 만들어 두고 하는 것이 약간 더 효과적입니다

셋 컴프리헨션, 제너레이터 컴프리헨션

셋 컴프리헨션은 컴프리헨션 구문을 이용해 set 아이템들을 구성합니다! 기본적인 문법은 list와 매우 흡사합니다.

{ *value expr* for *expr* in *iterable_object* }

set을 의미하는 중괄호 { }가 붙었습니다. 이외 다른 특징은 없습니다.

```
>>> a_set = { number for number in range(1,6) if number % 3 is 1}
>>> a_set
{1, 4}
```

튜플 컴프리헨션은 없습니다! 이미 리스트 컴프리헨션을 할 때 값 표현식에 소괄호 ( ) 를 넣으면 튜플을 만들어 봤었기 때문입니다.

하지만 소괄호를 집어 넣으면 튜플 컴프리헨션을 사용 하는 것이다 라고 착각 할 수 있는데, 다음 코드로 한번 보겠습니다.

(*value expr* for *expr* in *iterable_object* if *condition*)

위 처럼 컴프리헨션을 지정하면 마치 튜플 컴프리헨션을 사용 하는 것 처럼 보이긴 하지만, 사실은 **제너레이터 컴프리헨션**을 만들었다라고 이야기 할 수 있습니다. **제너레이터는 순회를 한번 하고 나면 다시 순회 할 수 없다는 특징이 있습니다.**

```
>>> number_thing = ( number for number in range(1,6))
>>> for num in number_thing:
...     print(num)
...
1
2
3
4
5
>>> for num in number_thing:
        print(num)
```

드디어 여러분만의 기능을 만들 수 있습니다 - 함수

지금까지의 모든 예제는 작은 코드 조각일 뿐이었습니다. 이 코드 조각들을 필요 할 때마다 계속 입력 하는 것은 옳은 방법이 아닙니다.

우리가 만들어 놓은 기능(function)을 반복해서 사용 할 수 있는 첫 번째 방법이 지금부터 알아볼 함수(function) 입니다.
함수는 첫 번째로 정의define의 단계를 가집니다. 정의란 ~한 기능을 가진 코드 조각을 만들겠다! 라는 뜻이 됩니다.
두 번째로 호출call 입니다. 호출은 정의된 함수를 사용 하는 것입니다. 호출을 함과 동시에 정의된 기능이 곧바로 실행됩니다.

함수는 매개변수parameter와 반환return 이라는 개념이 있습니다.
매개변수란, 함수를 호출 하면서 필요한 데이터들을 받아내는 변수입니다.
반환이란, 함수를 호출 하고 호출에 대한 결과 데이터를 받아 낼 수 있는 것을 반환 이라고 합니다.

실제 함수를 만들어 보고 호출하면서 자세히 이야기 해보죠

함수 만들기 기본

먼저 파이썬에서 함수를 만드는 방법은 다음과 같습니다.

#대괄호 [ ] 는 생략이 가능 하다는 뜻입니다.

def *function_name*(*[parameter1, parameter2, …]*) :

    #do something…

    [return data1,data2,data3 …]

파이썬 함수를 정의하기 위해서는 def와 함수이름(function_name) 및 매개변수([parameter…])를 설정 할 수 있습니다.

일단 제일 간단한 아무 일도 하지 않는 함수를 만들어 보겠습니다.

```
>>> def do_nothing():        □ 함수 정의 (define)
...     pass
...
>>> do_nothing()                    □ 함수 호출 (call)
>>>
```

먼저 간단히 정리하면, 위의 do_nothing 함수는 매개변수가 없는 함수입니다.

매개변수가 없더라도 괄호는 반드시 적어 주어야 합니다. 마찬가지로 콜론(:)을 이용하여 함수로써 해야 할 일을 적어 줍니다.

pass는 아무것도 하지 않을 때 사용 합니다.

위의 do_nothing 함수는 결론적으로 매개변수도 없고, 하는 일도 없으며, 뭔가 데이터를 반환 하지도 않습니다.

진짜 이제 뭔가 해봅시다

보통 do_nothing() 함수처럼 아무것도 하지 않는 함수는 아직 뭔가 함수가 해야 할 작업이 무엇인지 확실 하지 않을 때 많이 사용합니다.

이제는 뭔가 일을 해 보는 함수를 만들어 보도록 하죠!
개소리를 내는 함수를 만들어 보겠습니다.

```
>>> def dog_sound():
...     print('월월우러우렁뤙월')
...
>>> dog_sound()
월월우러우렁뤙월
```

dog_sound() 함수는 호출되면 print 문장이 실행 되면서 한 문장이 출력 됩니다.
dog_sound() 함수에서 뭔가 문장이 출력 되는 것도 괜찮지만, 함수의 결과로써 뭔가 받아 낼 수 있습니다.
동물의 이름과 동물 소리를 반환 받을 수 있는 함수를 만들어 보죠 아래의 예제는 매개변수와 반환을 사용하는 코드입니다.

```
>>> def animal_sound(animal):     animal_sound 함수를 정의하면서 animal 이라는 이름의 매개변수를 정의함
...     if animal is 'duck':     animal 변수에 들어있는 값을 조사 하는 if~ elif ~ else문장
...             return 'quack‘     animal 변수에 들어있는 값이 ‘duck’ 이라면 ‘quack’을 호출한 곳에 반환 하기
...     elif animal is 'dog':
...             return '월월멍멍월월월'
...     else:
...             return 'duck 또는 dog만 입력하세요'
...
>>> animal_sound('duck')             animal_sound를 호출하면서 매개변수 animal에 데이터 ‘duck’ 전달함. 이 때 전달되는 데이터를 인자
    (Argument)
'quack‘                  라고 합니다
>>> animal_sound('dog')
'월월멍멍월월월'
```

매개변수에 대한 여러가지 이야기들

파이썬 뿐만 아닌 여러 언어에서 함수를 만들 때 가장 익숙한 것은 위치 인자positional argument 입니다.
매개변수에 인자를 차례로 넣고 차례로 함수 안에서 사용 하는 것이죠.

```
>>> def today_menu(breakfast, lunch, dinner):
...     print('아침', breakfast)
...     print('점심', lunch)
...     print('저녁', dinner)
...
>>> today_menu('샌드위치','갈비탕','소고기')
아침 샌드위치
점심 갈비탕
저녁 소고기
```

가장 일반적인 인자 사용 방법이지만, 단점은 각 **위치 별 인자의 의미를 개발자가 알고 있어야 한다**는 것입니다.
만약에 다음과 같이 함수를 호출하면 약간은 이상해 질 것입니다.

```
>>> today_menu('소고기','샌드위치','갈비탕')
아침 소고기
점심 샌드위치
저녁 갈비탕
```

소고기가 맛있긴 하지만 아침에 먹기엔 조금 부담스러워 보이네요

소고기는 저녁에 먹자...

이전 예제를 다시 한번 상기시켜 보겠습니다. 함수를 호출 할 때가 문제가 되는데요, 왜냐 하면 위치 별 인자가 어떠한 의미를 가지고 있는 제대로 파악을 하지 못해서 벌어지는 일일 수도 있습니다. 따라서 인자의 순서에 맞게 신경 써서 넣어 주는 것이 아니라 내가 넣고 싶은 인자를 알맞은 매개변수에 넣어 주는 것이 좋다! 라고 생각을 하는게 조금 더 나은 생각 같습니다.

```
>>> today_menu(dinner='소고기', lunch='갈비탕', breakfast='샌드위치')
아침 샌드위치
점심 갈비탕
저녁 소고기
```

아주 간단하죠? 대입 하고 싶은 인자를 명시적인 매개변수를 이용해 넣어주는 방법입니다. 이를 ***키워드 인자*** 라고 합니다.

다이어트를 하기 위해 저녁은 샐러드를 먹기로 했습니다. 가끔은 다른 식사를 하겠지만요
아무튼 우리가 먹을 저녁의 기본은 *샐러드* 입니다. 파이썬 함수를 만들 때 *기본 매개변수 값을 지정* 할 수 있습니다.

```
>>> def today_menu(breakfast, lunch, dinner='샐러드'):
...     print('아침', breakfast)
...     print('점심', lunch)
...     print('저녁', dinner)
...
>>> today_menu('샌드위치', '백반')                  □ 인자를 넣어주지 않아도
아침 샌드위치
점심 백반
저녁 샐러드                                  □ 매개변수 dinner의 값이 '샐러드'로 고정 되어 있습니다.
>>> today_menu('햄버거','제육덮밥', '피자')      □ 물론 dinner에 인자를 전달하면
아침 햄버거
점심 제육덮밥
저녁 피자                                    □ 전달된 인자로 값이 전달 됩니다.
```

주의! 기본 매개변수 값을 지정 할 때는 항상 오른쪽부터 채워야 합니다. 왼쪽부터 채우거나 중간을 건너뛰면 안됩니다!

기본 인자값에 대한 진실

기본 인자는 언제 만들어 질까요? 보통 일반적인 매개변수 및 인자는 함수를 호출 할 때 만들어 집니다.

하지만 기본 인자값은 함수를 정의 할 때 이미 만들어져 있는 상태 입니다!

```
>>> def temp_func(arg, default_list = []):      temp_func의 매개변수인 default_list는 함수 정의 시 이미 만들어져 있는 상태입니다.
...     default_list.append(arg)
...     print(default_list)
...
>>> temp_func(1)    default_list를 다시 만들지 않고 함수 정의시에 만들어진 리스트를 계속 사용하는 중입니다.
[1]
>>> temp_func(2)                 따라서 default_list 에는 1, 2 가 들어있겠죠?
[1, 2]
```

인자를 모아서 관리하기 애스터리스크 (*)

우리한테 별 표시 라고 익숙한 * 기호의 명칭은 애스터리스크 asterisk 입니다.

파이썬에서는 애스터리스크를 사용해 함수의 위치 인자들을 하나로 묶어 줄 수 있습니다.

```
>>> def print_args(*args):
...     print("위치인자 튜플 : ", args)
...
>>> print_args(1,2,'hello','bye')      □인자를 개수의 순서에 상관 없이 받아 낼 수 있다.
위치인자 튜플 : (1, 2, 'hello', 'bye')  □ 튜플로 변환되어 출력된다.
>>> print_args()                □ 아무것도 전달 하지 않으면
위치인자 튜플 : ()                      □ 그냥 비어있는 튜플이 된다.
```

일반적인 위치 인자와도 같이 사용 할 수 있습니다.

```
>>> def print_more(arg1, arg2, *args):
...     print('arg1', arg1)
...     print('arg2', arg2)
...     print('args', args)
...
>>> print_more('a','b',1,2,3,4,5,'hello')
arg1 a
arg2 b
args (1, 2, 3, 4, 5, 'hello')
```

매개변수 기본 인자값 지정 처럼 애스터리스크를 사용해서 인자를 모아 관리 하기 위해 애스터리스크 매개변수는

제일 오른쪽에 위치 해야 합니다!

키워드 인자 모아서 관리하기 : **

위치 인자만 모으기 위해서는 애스터리스크를 한 개만 사용하면 됩니다.

키워드 인자도 같이 모아서 관리 하기 위해서는 애스터리스크를 두개 (**) 를 붙여 주면 됩니다!

애스터리스크를 한 개 붙여서 위치 인자를 모으면 튜플이지만, **키워드 인자를 모아서 관리하면 딕셔너리**가 됩니다.

```
>>> def print_kwargs(**kwargs):                    □ 애스터리스크를 두개 붙이면 키워드를 받아 낼 수 있습니다~
...     print('keyword arguments:', kwargs)
...
>>> print_kwargs(breakfast='샌드위치', lunch='소고기', dinner='갈비탕')
keyword arguments: {'breakfast': '샌드위치', 'lunch': '소고기', 'dinner': '갈비탕'}    □ 전달된 키워드 인자들이 딕셔너리 형태로 함수 내에서 활용됩니다.
```

위치 인자 모으기와 키워드 인자 모으기는 같이 쓸 수 있습니다.

단, 순서를 반드시 위치인자 모으기 □ 키워드 인자 모으기 순으로 해주셔야 합니다

```
>>> def goo(*args, **kwargs):
...     print('args :', args)
...     print('kwargs : ', kwargs)
...
>>> goo(1,2,3,4,keyword1='hi', keyword2='bye')
args : (1, 2, 3, 4)
kwargs :  {'keyword1': 'hi', 'keyword2': 'bye'}
```

일등 시민: 함수

함수는 단순히 기능만 실행 하는 것이다 라고 생각 하시나요?

실은 함수도 객체 입니다. 즉, 변수에 함수를 할당 할 수 있다는 이야기 입니다

```
>>> def foo(arg1):
...     print(arg1)
...
>>> foo_var = foo                # 함수 foo를 foo_var에 할당하기
>>> foo_var(10)       # foo_var로 함수 실행하기
10
```

일반 변수에 잘 들어가는 것이 확인 됩니다. 마찬가지로 함수 자체가 다른 함수의 인자로써 활용 되는 것도 가능하겠군요!

```
>>> def goo(a,b,func):
...     print("goo's a :", a)
...     func(b)        매개변수로 들어온 함수를 실행 시킵니다.
...
>>> goo('hello','hihi',foo)
goo's a : hello
hihi
```

함수 안에 또 다른 함수를 만들자

사실 함수 자체는 어디에서든 다 만들어 낼 수 있습니다.

하나의 함수가 호출 되면서 그 함수 안에 있는 또다른 함수를 만들어서 호출 하거나 반환 하게 할 수 있습니다.

```
>>> def outer(a,b):
...     def inner(c,d):
...         return c+d
...     return inner(a,b)
...
>>>
>>> outer(4,7)
11
```

위 코드의 동작 방식은 다음과 같습니다.

1) 먼저 outer의 매개변수에 인자 4,7이 전달되고 inner 함수를 정의 합니다.

2) 그리고 나서 outer의 맨 마지막 코드를 확인 해 보면 inner를 호출 하고 있네요.

3) 호출 되면서 outer를 호출하면서 받아온 a,b를 inner에 넘기는 것이 확인 됩니다.

4) 그렇게 되면 inner에서는 return c+d 코드를 통해 4와 7을 더한 값인 11이 outer의 최종 결과가 됩니다.

내부함수는 함수 안쪽에서 반복문 및 어떤 복잡한 작업이 한번 이상 이루어 져야 할 때 유용하게 사용됩니다.

클로져 Closure

클로져란 하나의 현상입니다. 다른 함수에 의해 동적으로 함수가 호출 될 때 일어나는 현상이라고 보시면 됩니다.

클로져의 가장 큰 특징은 바깥 함수로부터 생성된 변수값을 변경하고, 저장 할 수 있다는 것입니다.

```
>>> def outer(a,b):
...     def inner(c):
...             print('inner :', a,b)
...             return a+c, b*c
...     return inner
...
>>> func = outer(10,20)
>>> func(10)
inner : 10 20
(20, 200)
```

클로저의 핵심 코드는 outer 함수의 맨 마지막 부분인 return inner 입니다.

왜냐하면 아직 호출되지 않은 inner의 특별한 복사본(클로져)를 반환 하기 때문입니다.

따라서 outer 함수를 호출 할 때마다 새로운 inner 함수가 계속 생성 된다 라고 생각하면 됩니다.

익명 함수 : lambda()

파이썬의 람다 함수는 아주 간단한 함수를 만들어야 할 때 def 까지 사용해서 사용 하는 것이 부담 스럽다면,
그리고 코드 어디에라도 정의와 동시에 실행 되는 함수를 만들고 싶다면 람다 함수를 생각 해 볼 수도 있습니다.
여기에서는 아주 간단한 람다 함수만 살펴 보겠지만, 앞으로 예제를 진행 하면서 다양한 람다 함수를 만날 수 있을 것입니다.
먼저, 람다 함수 사용 방법은 다음과 같습니다. 람다는 expr의 결과를 리턴합니다. 만약 값이 없으면 리턴 하지 않습니다.

lambda *[parameter1, parameter2...]*: *expr*

```
>>> def f(x):  아주 간단한 연산을 하는 함수
...     return x**2
...
>>> f(3)
9
>>> g = lambda x: x**2     람다 표현식을 활용해서 아주 간단하게 만들어 주자.
>>> g(3)
9
>>> m = lambda x,y : x*y   매개변수가 2개
>>> m(10,20)
200
>>> n = lambda x,y : print(x,y)  값의 리턴이 아닌 단순히 출력만 해주기
>>> n(10,20)
10 20
```

어디에 있는거니... 네임스페이스와 스코프

우리가 알고 있는 파이썬에서의 name(이름)은 여러가지가 있습니다.

변수이름, 함수이름, 나중에 해볼 클래스의 이름 등등을 일컫습니다.

그렇다면 네임스페이스(namespace)는 무엇일까요? 이름 공간 이란 뜻이 됩니다.

파이썬의 네임스페이스는 현재 작성 중인 코드에서 접근 가능한 공간 이란 뜻을 가집니다.

네임스페이스는 두 가지 종류가 있습니다.

코드 어디에서든 사용 할 수 있는 전역 네임스페이스 (global namespace)와,

지정된 함수나 메소드 또는 클래스 에서만 사용 할 수 있는 지역 네임스페이스(local namespace)

예제를 보시죠!

```
>>> str = 'hello'   □ 함수 밖에 있으면 전역 네임스페이스가 됩니다.
>>> def foo():      □ 함수 내부에 있으면 지역 네임스페이스가 됩니다.
...     print(str)  □ 지역 네임스페이스 에서는 전역 네임스페이스 사용이 가능합니다.
...     hi = 'hihi' □ foo() 함수의 영역이 끝나면 변수 hi는 사라지게 됩니다.
>>> foo()
hello
>>> print(hi)       □ 전역 네임스페이스 에서는 지역 네임스페이스에 있는 요소(변수 또는 내부 함수 등)를 사용 할 수 없습니다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hi' is not defined
```

함축적인 것 보다 명확한 것이 낫다

먼저 예제로 바로 뭔가를 확인 해 보겠습니다.

```
>>> def foo():
...     print(str)
...     str = 'hihi'    전역 변수를 바꾸는 건지 지역 변수 str을 새로 만드는건지 명확하지가 않습니다.
...
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'str' referenced before assignment
```

파이썬의 철학 중 하나는, 함축적으로 뭔가 쓰기보다 명확한 것이 낫다 입니다.

위의 코드에서 문제가 되는 것은 무엇일까요? 바로 foo 함수 안에서 str = 'hihi' 코드가 문제가 됩니다.

왜냐하면 위에 있는 print(str) 코드 때문인데요, print(str) 코드는 전역변수 str을 출력 하는 코드가 되지만, 아래에 있는 str = 'hihi' 코드는 전역변수 str의 값을 바꾸는지, 아니면 새로운 지역 변수를 만들어 내는지 잘 모르고 있기 때문입니다.

우리는 명확하게 str이 전역 변수다 라는 것을 명시 해 줘야 합니다. foo() 함수의 코드를 다음과 같이 바꿔주세요

```
>>> def foo():
...     global str    global 키워드를 활용하여 str이 전역 네임스페이스에 위치한 전역 변수라는 것을 명시적으로 알려줍니다.
...     print(str)
...     str = 'hihi'
>>> foo()
hello
>>> print(str)
hihi
```

언제 어디서 무슨 오류가 발생할지?

의도하던, 의도하지 않았던, 오류 때문에 프로그램이 꺼지는 것은 막아야 할 것 같습니다.

리스트나 튜플에 잘못된 인덱스를 이용해서 접근한다거나, 딕셔너리의 존재하지 않는 키와 같은 몇몇 잘못된 상황에서의 예외(에러)를 처리 해야 합니다. 이를 예외 처리 라고 합니다.

사용자에게 무슨 일이 일어났는지, 뭘 잘못 건드렸는지 알려주는 것도 개발자의 역할입니다. 먼저 파이썬의 예외처리도 다른 언어와 비슷한데, 예외를 처리 할 때 까지 계속 예외 처리 구문을 찾다가 끝까지 (전역 네임스페이스) 예외 처리가 되지 않으면 프로그램을 강제로 종료 시켜 버립니다. 간단하게 파이썬의 예외처리에 대해 이야기 해보겠습니다.

```
>>> short_list = [1,2,3]    #short_list의 마지막 오프셋은 2 입니다.
>>> position = 5            #여기서는 5를 지정해서
>>> short_list[position]    # 6번째 인자를 빼오겠네요... short_list의 길이는 3입니다
Traceback (most recent call last):  따라서 예외가 나게 되겠죠?
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

위의 예제는 예외 처리가 되지 않아서 프로그램이 꺼져버리고 맙니다.

본격적으로 예외 처리 예제를 보기 전에 예외처리 문법 부터 보고 가겠습니다.

try:

#do something…

except: 또는 except *에러 타입*

try: 구문에는 예외가 발생 할 가능성이 있는 코드가 들어가고, except: 구문에는 발생된 예외를 안전하게 처리하는 구문이 들어갑니다.

각종 예외 처리 방법

예외 처리에는 두 가지 방법이 있습니다.

1) 모든 예외를 하나로 통으로 처리 하는 방법

2) 예외 타입을 기록하여 특정 예외에 대한 처리만 하는 방법

```
>>> short_list = [1,2,3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('position이 이상해요')
position이 이상해요    예외처리 됨.
```

위의 예제에는 예외 타입이 없는 첫 번째 예외처리 방식입니다. 위와 같이 try와 except: 만 적어주면 포괄적인 모든 예외에 대한 처리를 한다 라고 생각 해 볼 수 있습니다.

위의 방법도 괜찮긴 하지만, 조금 더 세세하게 예외를 처리 하기 위해서는 예외 타입을 직접적으로 기록 해 두는 것이 좋습니다.

```
>>> try:
...     short_list[position]
... except IndexError as err:          명시적으로 예외를 직접 기록함
...     print('position이 이상해요', err)
position이 이상해요 list index out of range
```

05. 데이터 가공하기

문자열을 가공하여 필요한 데이터로

- 문자열 포매팅(Formatting)
- 정규식 사용하기(Regular Expression)

지금까지는 텍스트 포매팅이 없었습니다.

단순히 지금까지는 print 함수 같은 것들을 이용해서 인터프리터가 값을 표시하도록 놔뒀습니다.

이제부터는 값이 바뀔 때마다 문자열이 같이 바뀌어서 사용하는 것이 아닌, 기준이 되는 문자열(format)을 만들어 놓고, 값을 대입하는 식으로 만들어 내겠습니다. 문자열의 **format** 함수를 활용합니다.

```
>>> n = 42
>>> f = 7.03
>>> s = 'hello!'
>>> '{} {} {}'.format(n,f,s) # 중괄호의 위치에 알맞게 n,f,s 가 각각 입력 됩니다.
'42 7.03 hello!'
```

format 함수를 활용해 간단하게 문자열에 중괄호를 집어 넣어서 우리가 원하는 값을 타입에 상관 없이 간단히 집어 넣을 수 있습니다.

format의 강점은 순서와 변수의 이름까지도 지정 할 수 있다는 데에 있습니다.

```
>>> '{2} {0} {1}'.format(n,f,s)  # format의 매개변수를 표현할 순서를 중괄호 안쪽에 지정 할 수도 있습니다.
'hello! 42 7.03'
>>> '{str1} {str2} {str3}'.format(str1= ' 비트코인',str2='영차영차', str3='가즈아 ' )
 ' 비트코인 영차영차 가즈아'
```

딕셔너리를 사용해서도 문자열 포매팅이 가능 합니다

```
>>> d = {'n' : 42, 'f':7.03, 's':'hello!'}
>>> '{0[n]} {0[f]} {0[s]} {1}'.format(d, 'other') # 매개변수 오프셋 0번에 위치한 딕셔너리 객체에서 키값을 이용해 하나씩 빼기, 'other' 문자열은
      매개변수상 1
'42 7.03 hello! other'
```

정규식 활용하기.. re 모듈 활용하기

정규식은 원하는 문자열 패턴을 직접 사용자가 정의하여 소스 문자열과 일치하는지 비교 합니다.

먼저 파이썬에서 정규식을 활용 하기 위해서는 re 모듈을 임포트 해여 사용해야 합니다.

정규식은 패턴과 문자열을 개발자가 직접 지정해서 원하는 결과를 얻어 냅니다. 간단한 정규식 입니다. match 함수를 이용해 검사합니다.

```
>>> import re # 정규식 사용을 위한 re 모듈 import 하기
>>> result = re.match('You', 'You can do it! ' ) #  'You can do it!' 문자열이 ' You'  패턴에 매칭되는지 검사한 결과를 result 변수에 넣습니다.
```

match 함수의 첫 번재 매개변수에는 **검사를 수행할 패턴**이 들어가고, 두 번째 매개변수에는 **검사 대상 문자열**이 들어갑니다.

패턴을 **컴파일** 시켜 놓고 조금 더 빠르게 정규식 검사를 수행 할 수도 있습니다. 미리 컴파일 했기 때문에 재사용이 가능합니다.

```
>>> youpattern = re.compile('You')
>>> result = youpattern.match('You can do it!')
```

match만 사용 할 수 있는 것은 아닙니다.

match()는 문자열의 첫 번째 부터 매치되는 패턴을 검색하여 변환합니다.

search() 는 위치에 상관 없이 첫 번째 일치하는 객체를 반환합니다.

findall()은 중첩에 상관 없이 모두 일치하는 문자열 리스트를 반환합니다.

split() 은 패턴에 맞게 검사대상 문자열을 쪼갠 후 문자열 조각의 리스트를 반환 합니다.

sub() 는 대체 시킬 문자열 인자를 하나 더 받아서, 패턴과 일치하는 모든 검사 대상 문자열을 대체 인자로 변경합니다.

본격적으로 살펴 보겠습니다.
- match부터 시작할게요

‘You can do it!’ 문자열은 과연 ‘You’라는 단어로 시작하는가? 에 대한 내용입니다.

```
>>> import re # 정규식 사용을 위한 re 모듈 import 하기
>>> source = 'You can do it!'
>>> m = re.match('You', source) # source 변수에 들어있는 문자열이 ‘You’와 매칭되는지에 대한 결과물을 리턴합니다.
>>> if m:
...     print(m.group()) # 패턴 ‘you’와 매칭되는 문자열을 출력 합니다.
...
You
>>> m = re.match('Hey', source) # source에 들어있는 문자열은 ‘Hey’로 시작 하지 않죠!
>>> m    # 출력 시에 아무 것도 나오지 않습니다. 탐색 할 내용이 없단 뜻이 되겠네요.
>>> if m:              # 당연히 해당 조건문은 실행 되지 않습니다.
...     print(m.group())
...
>>>
```

위의 소스코드를 해석 해 보겠습니다.

생각보다 정규식이라는게 별게 없습니다. 단순히 match 함수를 이용 했을 때 source 문자열이 ‘You’로 시작하는 검사 합니다.
검사 후엔 group() 함수를 이용해 매칭 되는 곳에 있는 문자열을 찾아 냅니다.

문자열 You c 까지 등장 하려면 어떤 식으로 패턴을 마련 해야 할까요?

어디에 있던 작동하는 search()

이전에 살펴 보았던 match 같은 경우는 항상 문자열이 시작 할 때의 패턴을 매칭 시키지만, search는 어디에 있던 작동 합니다.

```
>>> source = '이런 내맘 모르고 너무해 너무해'
>>> m = re.match('너', source) # source에 들어있는 문자열이 '너'로 시작 하지 않기 때문에 아무것도 나오지 않겠죠?
>>> if m:
...     print(m.group())
...
>>> m = re.search('너', source) # search는 '너' 라는 문자열이 어디에 있던 패턴 검색이 작동됩니다.
>>> if m:
...     print(m.group())
...
너
>>> m = re.search('너.*', source) # 문자열 '너'로 시작하고, 그 뒤에 문자가 아무거나 오는 패턴을 매칭 시킵니다.( 잠시 후에 자세한 설명이 있습니다.)
>>> if m :
...     print(m.group())
...
너무해 너무해
>>>
```

search는 match와 다르게 어디에 있던 작동 됩니다. 위의 소스에 대해 간단히 설명을 덧붙여 보자면

1) . 은 공백을 포함한 모든 문자 한 문자를 의미 합니다.

2) *은 이전 패턴(여기에서는 .)이 여러 개 올 수 있다는 것을 의미 합니다. 0개 이상이라는 뜻이 되겠네요.

3) '너' 는 반드시 매칭 되어야 할 문구를 의미 합니다.

일치하는 모든 패턴을 찾아봅시다. findall()

findall은 전체 문자열 대상으로 패턴에 매칭되는 문자열들을 리스트로 리턴합니다. 모모랜드-뿜뿜 가사를 가져오겠습니다.

```
>>> bboombboom = '''설레어 너와 나의 랑데뷰
... (랑데뷰)
... 내 마음을 들었다 놨다 해 맘대루
... (맘대루)
    ... 중간 생략 ...
...
... Just feel it 뿜뿜
... 네 앞에서 난 뿜뿜
... 네게 줄게 뿜뿜 뿜뿜 어'''
>>> m = re.findall('뿜', bboombboom) # bboombboom에 '뿜'과 매칭되는 모든 문자열을 찾아냅니다.
>>> m
['뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜',
    '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜', '뿜']
```

모든 곳에 있는 뿜을 찾아냅니다. '뿜' 다음에 어떤 문자가 오는지 검색 하려면 뒤에 .?를 붙여 주면 됩니다.

```
>>> m = re.findall('뿜.?', bboombboom) # ?의 의미는 앞에 있는 패턴이 0개 또는 1개 매칭 될 때 입니다.
>>> m
['뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜어', '뿜어', '뿜어', '뿜어', '뿜 ', '뿜 ', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜', '뿜뿜',
    '뿜뿜', '뿜뿜', '뿜뿜']
```

새로운 패턴인 ?이 등장 했습니다. ?의 뜻은 앞에 있는 패턴이 0개 또는 1개 인 것을 의미 합니다.

실습 : 위의 결과에 나온 요소들이 각각 모모랜드-뿜뿜에서 몇번씩 등장하는지 딕셔너리 컴프리헨션으로 만들어서 표현해주세요

원하는 패턴으로 문자열을 나누기 - split()

문자열에도 **split** 함수가 있어서 문자열을 나눠주지만, 정규식을 활용하면 정규식 패턴을 이용한 문자열 분리가 가능합니다.

```
>>> twice_tt = '''I'm like TT
... Just like TT
... 이런 내 맘 모르고 너무해 너무해
... I'm like TT
... Just like TT
... Tell me that you'd be my baby
... '''
>>> m = re.split('T.?', twice_tt) # T 다음으로 나오는 한 글자를 기준으로 twice_tt 문자열 나누기
>>> m
['I'm like ', '\nJust like ', '\n이런 내 맘 모르고 너무해 너무해\nI'm like ', '\nJust like ', '\n', 'll me that you'd be my baby\n']
```

이전에 사용 했던 패턴인 **?** 까지 같이 써 보았습니다. **T** 다음한 글자로 자르니까 **Tell** 의 **Te** 또한 나누기의 기준이 되는군요

패턴으로 문자열 치환하기 : sub()

마찬가지로 문자열의 **replace** 와 비슷한 역할을 합니다. 단지 패턴을 지정 할 수 있다는 점이 조금 다르겠죠?

```
>>> m = re.sub('li.{2}', 'love', twice_tt) # { 2 } 는 앞서 등장한 패턴이 2회 등장 할 때의 매칭 입니다.
>>> print(m)
I'm love TT
Just love TT
이런 내 맘 모르고 너무해 너무해
I'm love TT
Just love TT
Tell me that you'd be my baby
```

새로운 패턴이 또 등장합니다. 추후에 한꺼번에 정리 하겠지만, 미리미리 하나씩 맛만 본다고 생각 해보시면 될 것 같습니다.

단순히 설명해서 **{ n }**은 앞에 등장한 패턴이 최대 **n**회 반복 되는 패턴을 의미 합니다.

메타문자 활용하기

정규식을 표현 할 때 사용 할 수 있는 메타문자들 입니다.

메타문자란, 원래 그 문자가 가진 뜻이 아닌, 특별한 용도로 사용되는 문자를 말합니다. expr은 표현식을 의미합니다.

메타문자	뜻
.	줄바꿈 문자 \n을 제외한 모든 문자와 매칭됩니다.
[*expr*]	대괄호 내에 표시된 문자 중 한 글자만 매칭됩니다.
expr *	바로 앞에 있는 문자가 0개 이상 시에 매칭 됩니다.
expr +	바로 앞에 있는 문자가 1개 이상 시에 매칭 됩니다.
expr ?	표현식이 0개 또는 1개일 때 매칭 됩니다.
expr{m,n}	표현식이 m ~ n 회 반복 되면 매칭 됩니다.
expr1 | *expr2*	or로써 expr1 또는 expr2에 매칭됩니다.
^*expr*	expr에 해당하는 문자열로 시작할 때 매칭됩니다.
expr$	expr로 끝날 때 매칭됩니다.

메타문자 활용하기

- 개행문자 \n을 제외한 모든 문자를 표현 가능한 .
- [ ] 을 이용해 한 글자만 매칭 하고 싶을 때, 그리고 범위 지정 -

메타문자를 활용해서 정규식을 살펴보도록 하겠습니다.

```
>>> str = 'cat'
>>> re.findall('[a-c]', str) # a와 c 사이에 있는 문자열에 대해 매칭 검사를 수행 합니다. a,b,c가 되겠네요
['c', 'a']
>>> re.findall('[cat]', str) # c 또는 a 또는 b와 매칭되는 문자열을 검사합니다
['c', 'a', 't']
>>> re.findall('[a-c]', str)
['c', 'a']
>>> re.findall('a.', str) # a 다음 하나의 어떤 문자 든지 매칭 합니다.
['at']
```

메타문자 활용하기

- 0번 이상 반복 할 때 사용 할 수 있는 *
- 1번 이상 반복 할 때 사용 할 수 있는 +

반복 메타문자는 단순히 0번 이상이냐, 1번 이상 이냐의 차이 밖에 없습니다.

```
>>> pattern = re.compile('ca*t') #a가 0번 이상이면 매칭 됩니다.
>>> pattern.findall('ct')
['ct']
>>> pattern.findall('cat')
['cat']
>>> pattern.findall('caaat')
['caaat']

>>> pattern = re.compile('ca+t') # a가 한번 이상일 때만 매칭 됩니다.
>>> pattern.findall('ct')
[]
>>> pattern.findall('cat')
['cat']
>>> pattern.findall('caaat')
['caaat']

>>> re.findall('[a-c]*','caca') # caca를 이미 소모한 상태에서 마지막 한번 을 더 검사 하기 때문에 (0회 일 때의 검사)
['caca', '']                #결과물에 아무것도 없는 문자열이 포함됩니다.
>>> re.findall('[a-c]+','caca') # 무조건 앞에 a-c까자의 문자열이 하나는 있어야 하기 때문에  caca 이후의 문자열은 검사 하지
    않습니다.
['caca']
```

메타문자 활용하기

- 횟수를 지정 할 수 있는 {m, n } □ m번 ~ n번 반복 시

반복 횟수 패턴을 지정 해 줄 수도 있습니다.

```
>>> re.findall('a{2,4}', 'cat')
[]
>>> re.findall('a{2,4}', 'caat')
['aa']
>>> re.findall('a{2,4}', 'caaat')
['aaa']
>>> re.findall('a{2,4}', 'caaaat')
['aaaa']
>>> re.findall('a{2,4}', 'caaaaat')
['aaaa']
```

메타문자 활용하기
- 이거 아님 저거 OR(|) 메타문자

메타문자 | 입니다. 보통 OR 조건을 구현 하기 위해 많이 사용하는데, 정규식에서도 마찬가지 입니다. 사용 방법을 바로 보시죠

```
>>> import re
>>> source ='''Oh her eyes her eyes
... Make the stars look like they're not shining
... Her hair her hair
... Falls perfectly without her trying'''
>>> m = re.findall('eyes|hair', source) # eyes 혹은 hair와 매칭 시킵니다.
>>> m
['eyes', 'eyes', 'hair', 'hair']
>>> m = re.findall('e\w+|h\w+', source) # e로 시작하는 알파벳으로 이루어져 있거나, h로 시작하는 알파벳으로 이루어진 단어
>>> m
['her', 'eyes', 'her', 'eyes', 'he', 'hey', 'hining', 'er', 'hair', 'her', 'hair', 'erfectly', 'hout', 'her']
```

메타문자 활용하기

- 특정 패턴으로 시작하는 매칭(^)
- 특정 패턴으로 끝나는 매칭 ($)

이번엔 시작 패턴을 지정하여 확인 해 보는 방법입니다.

시작 패턴을 지정 할 때는 ^ 를 사용 합니다. 추가적으로 정규식에 **re.MULTILINE** 옵션을 부여하면 모든 라인에 대해 각각 검사합니다.

```
>>> re.findall('^Oh.+', source) # 전체 문자열에서 Oh로 시작하고 이후에 공백 문자를 포함한 문자열이 1개 이상 나오는지 검사하기
['Oh her eyes her eyes']

>>> re.findall('^.+t.+', source, re.MULTILINE) # 각 라인별 검사하기. 이 때 중간에 문자열 t가 껴있는지 검사한다.
["Make the stars look like they're not shining", 'Falls perfectly without her trying']

>>> re.findall('\w+ing$', source, re.MULTILINE) # ing로 끝나는 문자 패턴 검사하기
['shining', 'trying']
```

실습 : 끝이 ly로 끝나는 문자열을 추출 하려면 어떻게 해야 할까요?

특수 문자 패턴 살펴보기

다음 표는 정규식에서 사용 할 수 있는 특수 문자들 입니다. 이 특수 문자를 이용해서 원하는 패턴을 손쉽게 지정 가능 합니다.

패턴	일치
\d	숫자만 일치 됩니다.
\D	숫자가 아닌 것만 일치 됩니다.
\w	문자, 숫자, 언더바(_) 만 일치됩니다.
\W	문자, 숫자, 언더바 (_) 를 제외한 문자만 일치됩니다.
\s	공백문자만 일치 됩니다. (‘\r’,’\t’,’ ‘, ’\n’)
\S	공백문자가 아닌 것만 매칭 됩니다.

string 모듈에서 정규식 테스트 하기

string 모듈에서는 우리가 간단히 테스트 해볼 수 있는 문자열 상수가 미리 정의 되어 있습니다. 각각 특수기호 사용을 유의깊게 보세요

```
>>> import re
>>> import string
>>> printable = string.printable
>>> printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
>>> re.findall('\d', printable) # printable에서 숫자만 찾기
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> re.findall('\w', printable) # printable에서 숫자, 문자, 언더바 ( _ )만 찾기
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
      'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '_']
>>> re.findall('\s', printable) # printable에서 공백문자만 찾아내기
[' ', '\t', '\n', '\r', '\x0b', '\x0c']
>>> re.findall('\W', printable) # printable에서 숫자, 문자, 언더 바 가 아닌 특수문자만 찾기
['!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/', ':', ';', '<', '=', '>', '?', '@', '[', '\\', ']', '^', '`', '{', '|', '}', '~', ' ', '\t', '\n', '\r', '\x0b', '\x0c']
```

실습 ! : 원하는 노래 가사에서 각각 위의 특수문자로 테스트 해보세요