[**1. 파이썬 기본 핵심 이해하기** 7](#_Toc487713414)

[1.1 값(value)처리 7](#_Toc487713415)

[1.1.1 리터럴(literal) 7](#_Toc487713416)

[1.1.2 표현식(expression ) 8](#_Toc487713417)

[1.1.3. 조건식(condition expression) 9](#_Toc487713418)

[1.2. 이름(name) 처리 11](#_Toc487713419)

[1.2.1. 예약어(keyword) 11](#_Toc487713420)

[1.2.2. 명명규칙(naming convention) 15](#_Toc487713421)

[1.2.3. 변수(Variable) 18](#_Toc487713422)

[1.2.4. 함수(function), 클래스(class) 이름 관리 방법 20](#_Toc487713423)

[1.3. 데이터 모델(data model) 23](#_Toc487713424)

[1.3.1 클래스(class), 객체(object), 인스턴스(instance) 23](#_Toc487713425)

[1.3.2 파이썬 기본 내장 함수 이해 29](#_Toc487713426)

[1.3.3. 메타클래스(meta class) : type class 32](#_Toc487713427)

[1.3.4. 내장 자료형(built-in data type) 34](#_Toc487713428)

[1.3.5. 객체 비교 방식(is, ==), 레퍼런스 확인함수 39](#_Toc487713429)

[1.3.6. 자료형의 변환(Mutation) 41](#_Toc487713430)

[1.4. 바인딩(binding) 및 평가방식(evaluation) 45](#_Toc487713431)

[1.4.1. 바인딩에 따른 자료형 결정 45](#_Toc487713432)

[1.4.2. 표현식(expression)의 평가 방법 47](#_Toc487713433)

[1.4.3. interface 처리 규칙 50](#_Toc487713434)

[1.5. 요약 52](#_Toc487713435)

[**2. 파이썬 문장 이해하기** 53](#_Toc487713436)

[2.1. 문장 꾸미기 53](#_Toc487713437)

[2.1.1. 줄 들여쓰기 (Lines and Indentation) 54](#_Toc487713438)

[2.1.2. 멀티 라인 문 (Multi-Line Statements) 55](#_Toc487713439)

[2.1.3. 인용 (Quotation) 57](#_Toc487713440)

[2.1.4. 주석 (Comments) 58](#_Toc487713441)

[2.2. 프로그램 문장 59](#_Toc487713442)

[2.2.1. 할당(assignment) 문장 59](#_Toc487713443)

[2.2.2. 단순 제어문 61](#_Toc487713444)

[2.2.3. 복합 제어문 66](#_Toc487713445)

[2.2.4. 순환문 처리 : for 68](#_Toc487713446)

[2.2.5. 순환문 처리 : while 70](#_Toc487713447)

[2.2.6. continue/break 71](#_Toc487713448)

[2.2.7. else 문 추가하기 72](#_Toc487713449)

[2.2.8. global 문 과 nonlocal 문 처리 74](#_Toc487713450)

[2.3 함수 및 클래스 정의문 처리 80](#_Toc487713451)

[2.3.1. 함수 정의문 80](#_Toc487713452)

[2.3.2. 클래스 정의문 82](#_Toc487713453)

[2.3.3. pass 문장 84](#_Toc487713454)

[2.4 모듈(module)/패키지(package) 처리 85](#_Toc487713455)

[2.4.1. import 구문 85](#_Toc487713456)

[2.4.2 모듈 생성 및 실행하기 90](#_Toc487713457)

[2.4.3. 모듈 네임스페이스 확인하기 93](#_Toc487713458)

[2.4.4 패키지(package) 처리 95](#_Toc487713459)

[2.5. 네임스페이스(Namespace) 관리 규칙 103](#_Toc487713460)

[2.5.1. 모듈 네임스페이스에 할당하기 103](#_Toc487713461)

[2.5.2. 전역 Namespace 처리 규칙 105](#_Toc487713462)

[2.5.3. dir 함수를 통해 네임스페이스 출력하기 107](#_Toc487713463)

[2.5.4. 단순한 namespace 처리용 클래스 109](#_Toc487713464)

[2.6. 요약 112](#_Toc487713465)

[**3. 자료형 : 숫자 원자형** 113](#_Toc487713466)

[3.1. 숫자 자료형의 특징 113](#_Toc487713467)

[3.1.1. 숫자 타입의 특징 114](#_Toc487713468)

[3.1.2 숫자 타입의 형변환 115](#_Toc487713469)

[3.1.3 문자열 중에 숫자 형태의 숫자 형변환 118](#_Toc487713470)

[3.1.4. 2, 8, 16진법을 정수에서 변환 하기 119](#_Toc487713471)

[3.1.5. 숫자 자료형 최대 처리 수 확인 120](#_Toc487713472)

[3.2. 내장 숫자 자료형 이해하기 124](#_Toc487713473)

[3.2.1 정수 int class 124](#_Toc487713474)

[3.2.2. 불리언 타입 : bool 125](#_Toc487713475)

[3.2.3 실수 float 128](#_Toc487713476)

[3.2.4 복소수 complex class 130](#_Toc487713477)

[3.2.5. 유리수 Fraction class 131](#_Toc487713478)

[3.2.6. 큰 수 decimal class 133](#_Toc487713479)

[3.3. 연산자(Operator) 137](#_Toc487713480)

[3.3.1. 연산자 표기법 137](#_Toc487713481)

[3.3.2. 연산자 우선순위 138](#_Toc487713482)

[3.3.3. 산술연산(arithmetic operation) 139](#_Toc487713483)

[3.3.4. 비교 연산(comparison operation) 141](#_Toc487713484)

[3.3.5. 비트 연산(bits wise operation) 142](#_Toc487713485)

[3.3.6. 논리 연산(logica operation) 및 단축연산(short cut operation) 145](#_Toc487713486)

[3.4. 요약 148](#_Toc487713487)

[**4. 자료형 : Sequence 형** 149](#_Toc487713488)

[4.1. 파이썬 Sequence자료형의 특징 149](#_Toc487713489)

[4.1.1. 런타임에 속성 추가 여부 149](#_Toc487713490)

[4.1.1. 변경가능 여부 : Mutable & Immutable 151](#_Toc487713491)

[4.1.3. 컬렉션(collections) 여부 153](#_Toc487713492)

[4.1.4. sequence 타입 내의 메소드 처리 기준 156](#_Toc487713493)

[4.1.5. interning 처리 157](#_Toc487713494)

[4.2. 문자열 자료형(string data type) 159](#_Toc487713495)

[4.2.1. 문자열 생성 방법 159](#_Toc487713496)

[4.2.2. 문자열 주요 메소드 160](#_Toc487713497)

[4.3 바이트 자료형(bytes data type) 171](#_Toc487713498)

[4.3.1. 바이트 생성 171](#_Toc487713499)

[4.3.2 바이트 주요 메소드 172](#_Toc487713500)

[4.4. bytearray 자료형(data type) 175](#_Toc487713501)

[4.4.1. bytearray 생성 175](#_Toc487713502)

[4.4.2. bytearray 내의 속성/메소드 176](#_Toc487713503)

[4.5. 튜플 자료형(tuple data type) 180](#_Toc487713504)

[4.5.1. 튜플 생성 180](#_Toc487713505)

[4.5.2. 튜플 메소드 182](#_Toc487713506)

[4.5.3. 튜플 원소로 Mutable 자료형 처리 방법 183](#_Toc487713507)

[4.6. 리스트 자료형(list data type) 185](#_Toc487713508)

[4.6.1. 리스트 생성 185](#_Toc487713509)

[4.6.2 리스트 복사 처리 186](#_Toc487713510)

[4.6.3 리스트 내의 메소드 190](#_Toc487713511)

[4.6.4 리스트 내의 리스트 초기화 하는 법 194](#_Toc487713512)

[4.7. Sequence 자료형 형변환 198](#_Toc487713513)

[4.7.1. Sequence 생성자를 이용한 형변환 198](#_Toc487713514)

[4.7.2. Sequence 메소드를 이용한 형변환 199](#_Toc487713515)

[4.8. 요약 202](#_Toc487713516)

[**5. Mapping/Set 자료형(data type)** 203](#_Toc487713517)

[5.1. 파이썬 Mapping 자료형의 특징 203](#_Toc487713518)

[5.1.1 딕셔너리(dict) 의 키 구성 및 생성 기준 203](#_Toc487713519)

[5.1.2. 딕셔너리(dict) 생성하기 205](#_Toc487713520)

[5.2. 딕셔너리(dict)타입 메소드 이해하기 212](#_Toc487713521)

[5.2.1. 딕셔너리(dict) 자료형 추가 /삭제 메소드 212](#_Toc487713522)

[5.2.2. 딕셔너리(dict) 타입에 keys, values, items 이해하기 214](#_Toc487713523)

[5.2.3. 딕셔너리(dict) 자료형 조회 및 갱신 메소드 217](#_Toc487713524)

[5.3. set 자료형 처리 하기 219](#_Toc487713525)

[5.2.1 set 219](#_Toc487713526)

[5.2.2. frozenset 이해하기 227](#_Toc487713527)

[5.4. 요약 229](#_Toc487713528)

[**6. 정수형 검색, 지능형, 문자열 포맷팅하기** 230](#_Toc487713529)

[6.1. 정수/키 검색(indexing) 이해하기 230](#_Toc487713530)

[6.1.1. Sequence 타입 index 순서 이해하기 230](#_Toc487713531)

[6.1.2. 딕셔너리(dict) Key indexing 이해하기 233](#_Toc487713532)

[6.1.3. 딕셔너리(dict) Key 검색 메소드 237](#_Toc487713533)

[6.2. sequence 자료형의 슬라이스(slicing) 239](#_Toc487713534)

[6.2.1. 리터럴로 직접 슬라이스 처리하기 239](#_Toc487713535)

[6.2.2. 슬라이스 클래스 이용하기 245](#_Toc487713536)

[6.3. 지능형(comprehension) 처리 249](#_Toc487713537)

[6.3.1 지능형 list 249](#_Toc487713538)

[6.3.2 지능형 dict/set 254](#_Toc487713539)

[6.3.3. 지능형 set 255](#_Toc487713540)

[6.3.4. 지능형 Namespace 이해하기 257](#_Toc487713541)

[6.4. 문자열 포맷팅(string formmating) 처리 259](#_Toc487713542)

[6.4.1 포맷팅 기본 이해하기 259](#_Toc487713543)

[6.4.2. 포맷코드 이해하기 262](#_Toc487713544)

[6.4.3. 포맷함수 및 메소드 인자 매핑 266](#_Toc487713545)

[6.4.4. 포맷팅 꾸미기 269](#_Toc487713546)

[6.4.5. 날짜 포매팅 처리 274](#_Toc487713547)

[6.5. 요약 277](#_Toc487713548)

**1. 파이썬 기본 핵심 이해하기**

파이썬은 다양한 개념들이 버전 별로 추가되면서 진화한 프로그램 언어이다. 객체지향, 함수형 언어의 특징도 파이썬 내부에 추가되었고, 데이터 과학 등을 위한 다양한 모듈들도 기능으로 제공되면서 발전하고 있다.

우리는 프로그램 언어를 배울 때 주로 문법과 이 문법에 대한 작동원리만 주로 공부하지만 문법에 반영된 소프트웨어에 대한 기본 개념들은 알아보려 하지 않는다. 이번 장에서 일단 기본 용어에 대한 문법을 알아보면서 하나하나씩 알아가 보면 왜 모든 프로그램언어들이 동일한 개념 용어, 연산자, 자료형 등을 사용하는지 이해를 할 것이다.

## 값(value)처리

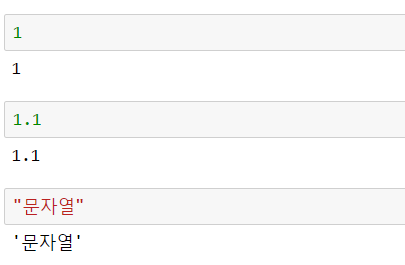
파이썬은 모든 것을 값(value)로 처리하고 이를 재사용하려면 값을 저장하는 변수(이름)이 필요하다. 먼저 값이 가리키는 것, 파이썬 내부의 객체들을 말하며 이 값들이 어떻게 관리되고 처리되어야 하는 지를 간단히 알아보겠다.

### 1.1.1 리터럴(literal)

리터럴이란 프로그램 언어로 작성된 코드에서 값을 대표하는 용어이다. 파이썬 언어에서는 기본으로 사용되는 정수, 부동 소수점 숫자, 문자열, 불린 등의 객체로 평가되는 모든 것이 값이고 이를 평가할 수 있다는 것을 말한다.

예제 1-1 : 리터럴 값을 넣고 즉시 실행

리터럴은 값이므로 셀에 입력하고 실행하면 그 결과를 처리한다. 정수, 실수, 문자열 등의 자료형은 입력하고 실행하면 자기 자신을 그대로 출력한다.



위 예제 1-1에서 주피터 노트북을 실행한 후에 빈 셀(cell)에 1, 1.1, “문자열”등을 입력하고 실행(shift+enter)을 하면 결과가 1이라는 정수, 1.1이라는 실수, “문자열”이라는 문자열 자료형의 리터럴 값을 바로 출력한다.

### 1.1.2 표현식(expression )

리터럴 값을 표현식에서는 피연산자(operand)로 보고 이들을 계산하는 연산자(operator)와 묶어 표현식을 만들 수 있다. 이 표현식도 리터럴처럼 평가가 되며 이 평가의 결과는 항상 값으로 만들어진다.

* 표현식(expression)

표현식(expression)은 프로그래밍 언어가 해석하는 하나 이상의 명시적 값, 상수, 변수, 연산자조합이고 함수도 실행되면 하나의 값이 되어 이를 조합해도 표현식으로 인식한다.

표현식을 평가해서 실행될 때에는 우선 순위 및 연관 규칙에 따라 해석해 실행되며 평가된 결과는 하나의 값인 리터럴로 표현된다. 표현식 처리 순서는 좌측부터 우측으로 가며 평가하고 ( ) 연산자를 만나면 최우선으로 처리된다.

예제 1-2 : 표현식 평가

정수나 문자열을 사이에 연산자를 표현하면 하나의 표현식이 된다. 이를 실행하면 결과값이 나온다.



위 예제 1-2는 3+4, “문자”+”열”이라는 표현식을 주피터 노트북에 입력하고 실행(shift+enter)하면 표현식을 평가해서 최종 값을 결과로 처리한다.

### 1.1.3. 조건식(condition expression)

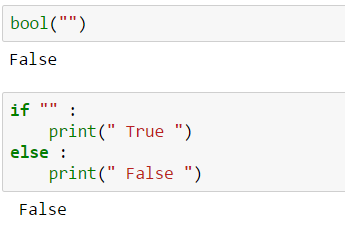
표현식에서 특정 조건문 등에 제한적으로 사용되는 것을 조건식이라 말한다. 주로 특정 문장인 if 문이나 while 문에서 처리된다. 조건을 판단이 필요한 경우에는 표현식 중에 조건식을 사용해서 항상 참과 거짓으로 평가해서 처리하는 것이 좋다.

* 조건식(condition)

조건식은 표현식이지만 특정 조건을 판단할 수 있는 구문인 If, while 문 등에 사용되며 조건이 평가되면 True나 False로 결과가 처리된다.

예제 1-3 : 조건식 평가

조건식을 평가하면 bool 자료형의 결과로 처리가 된다.



위 예제 1-3은 주피터 노트북에 if 문을 넣고 문자열에 아무 값도 없다는 조건을 입력하면 빈 문자열(“”)은 False로 인식해서 else 문장을 처리한다.

## 1.2. 이름(name) 처리

파이썬에서 프로그램을 작성한다는 것은 값들을 재사용하는 로직들로 구성되어 기능을 처리한다는 것이다. 모든 값들에 대한 저장을 관리하는 네임스페이스(Namespace)에 대한 처리를 이해하기 위해서는 반드시 어떻게 이름을 정의할 지를 명확히 이해해야 한다.

이름에 대한 처리 방식을 식별자라고 하고 이 식별자(identifier)는 프로그램언어로 프로그램을 작성할 때 변수, 함수, 클래스, 모듈, 패키지 등을 명확히 구별이 필요한 부분에 사용되는 개념이다. 파이썬에서 이름을 관리하는 방식이 중요하므로 이를 구별하는 식별자에 대한 처리도 동시에 중요하다.

패키지, 모듈, 프로그램, 클래스, 함수, 변수 등에 대한 이름을 명명하고 실제 실행환경에서 이를 사용할 때에 이름들이 명확하게 식별이 되지 않으면 충돌이 생겨 처리상의 이슈가 발생할 수 있다.

내부적으로 이름으로 처리할 수 있는 것을 하나씩 알아보자.

### 1.2.1. 예약어(keyword)

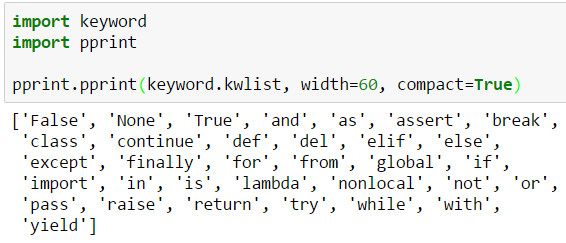
모든 것을 이름으로 관리하지만 파이썬 내부에서 먼저 문법적으로 사용해야 할 주요 단어를 사용하지 못하도록 정의했다. 이를 키워드 즉 예약어(keyword)로 지정해서 관리하고 있다. 파이썬에서는 keyword라는 모듈을 관리하면서 이를 사용자에게 제공하고 있다.

* 키워드 모듈 확인하기

파이썬에서는 keyword 모듈에 자신들이 사용하는 예약어 목록을 관리한다.

예제 1-4 : Keyword 모듈 알아보기

키워드 모들에 kwlist라는 속성에 파이썬 버전에 맞는 키워드를 관리하고 있다.



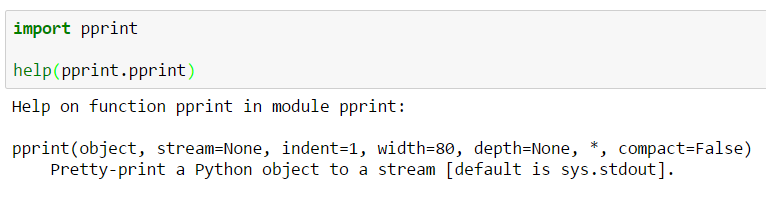
위 예제 1-4는 keyword와 pprint 모듈을 import하고 이를 사용해서 실제 keyword를출력한다. 파이썬은 버전마다 키워드 변경이 조금씩 있어와서 keyword 버전을 확인하면 현재 사용되는 것을 확인할 수 있다. pprint내의 width=60과 compact=True를 지정하면 60라인 내에 붙여서 출력한다.

* Pretty print 처리

모듈 pprint 내의 pprint 함수를 이용하면 print문을 사용할 때보다 리스트(list), 딕셔너리(dict) 등의 원소들을 보기 좋게 출력한다.

예제 1-5 : pprint 모듈 알아보기

모듈 pprint를 import하고 이 모듈에 있는 함수 pprint에 대한 도움말을 출력한다.

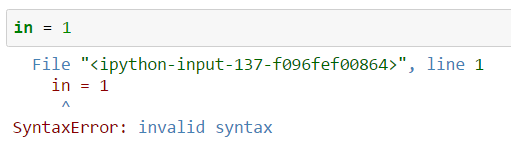


위 예제 1-5는 pprint 모듈에 pprint 함수에 대한 help 기능을 확인해 보면 object에 출력 값을 넣고 출력하면 들여쓰기(indent), 출력 라인 폭(width) 등에 대해 조정이 가능하다는 것을 보여준다.

* 키워드로 변수를 정의할 때 SyntaxError를 처리

키워드를 변수로 정의하면 SyntaxError를 발생시킨다.

예제 1-6 : 키워드를 변수로 지정시 예외 발생

키워드인 in을 변수인 이름으로 지정해서 값인 1을 할당하면 SyntaxError를 발생시킨다. 키원드는 변수로 사용할 수 없다는 표현이다.  


* 모듈 내에 식별자 이름이 중복될 경우

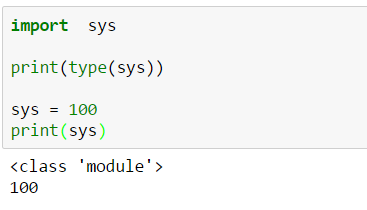
모듈명, 변수명 등도 전부 식별자로 사용되는 이름이다. 동일 모듈 내에서는 하나의 Namespace가 만들어지며 이를 Global Namespace로 사용된다.

같은 모듈에서는 동일한 이름이 사용되고 있다면 이 이름에 다른 내용도 할당이되므로 내부적으로 명확한 명명규칙을 가지고 처리해야 한다.

파이썬은 이름과 값에 대한 바인딩이 동적으로 되므로 실제 처리할 때 바인딩이가능하면 실행이 되므로 별자를 정의할 때 항상 주의해서 처리되어야 한다.

예제 1-7 : sys 모듈 이름을 변수로 사용

모듈 sys를 import하고 sys라는 이름을 변수로 사용해도 에러가 없이 처리된다.



위 예제 1-7처럼 외부의 모듈명 sys를 import해서 내 모듈에서 사용하고 있다. 이를 모르고 sys를 변수로 사용해서 100인라는 값을 할당해도 에러가 발생하지 않는다.

실제 sys라는 이름으로 모듈을 사용하다 내부적으로 정수로 바꾼 것이기에 이름은 동일하지만 실제 바인딩 시점에 바인딩되는 값만 달라진 것을 알 수가 있다.

### 1.2.2. 명명규칙(naming convention)

파이썬은 변수, 함수, 클래스, 모듈, 패키지 등 다양한 이름에 대한 명명규칙을 정의가 필요하며 이런 모든 것은 동일한 식별자로 생각하므로 이름을 가지고 모든 것을 체크해서 다른 프로그램 언어보다 명명규칙이 더 중요하다.

각 변수, 함수, 메소드, 클래스, 모듈, 패키지 등이 이름을 명확히 부여해서 누가 보더라도 이를 구분 할 수 있어야 하지만 많은 제공하는 모듈 등도 버전이 추가되면서 이런 규약이 잘 반영되어 있지는 않다.

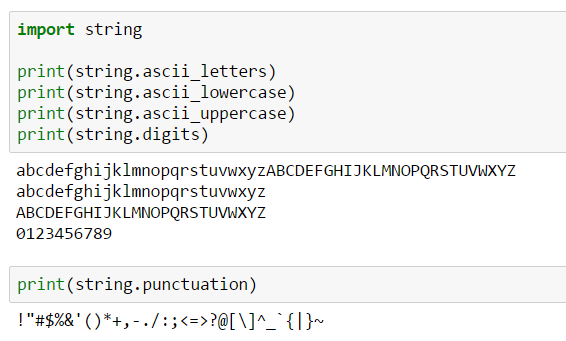
하지만 명명규칙을 이해해야 식별자로 쓰이는 방식을 이해하는 첫걸음이고 추후에 설명할 Namespace 와 scope 규칙을 이해하는데 도움이 될 것이다.

* 식별자(identifier) 가능 문자

파이썬이 문자열을 아스키(ASCII) 코드에서 유니코드(Unicode)로 변경했으므로 다양한 유니코드 문자도 수용하지만 관행 상 아스키코드인 영어로 표현한다.

예제 1-8 : 문자열 모듈 알아보기

문자열(string) 모듈을 import 해서 기본적으로 처리되는 문자들을 확인할 수 있다.



위 예제 1-8은 아스키 문자를 관리하는 문자열(string) 모듈에 있는 정보를 확인해 본 것이다. 관행 상 ascii\_letters, digits, punctuation 중에 \_만으로 식별자를 표현한다.

파이썬 3버전부터는 유니코드 문자열을 지원하므로 한글도 변수로 사용해도 된다.

* 파이썬 권장 명명 규칙 : 모듈, 패키지
* 모듈 이름은 짧아야 하고, 전부 소문자로 작성하는 것을 권장한다. 가독성을 위해서라면 밑줄(\_)을 쓸 수 있다.
* 패키지 이름 또한 짧아야 하고, 전부 소문자로 작성하지만 밑줄은 권장하지 않다.
* 파이썬 권장 명명 규칙 : 클래스
* 클래스 이름은 Capitalized words 형식(단어를 대문자로 시작)을 따른다. 카멜표기도 사용이 가능하다.
* exception은 클래스로, class와 동일하게 Capitalized words 형식 적용되지만 다만, 맨 뒤는 "Error"로 끝내는 것을 권장한다.
* 파이썬 권장 명명 규칙 : 상수, 변수, 함수, 메소드
* 변수, 함수와 메소드의 이름은 원칙적으로 소문자여야 하고, 가독성을 위해서 밑줄(\_)로 단어로 사용하고 상수 이름은 전부 대문자와 밑줄로 쓰는 것을 권장한다.
  + - 보호 속성일 때는 맨 앞에 \_ (leading underscore)를 추가적으로 붙인다.
    - 키워드와 동일 변수일 때는 맨 뒤에 \_ (leading underscore)를 추가적으로 붙인다.
    - 비공개 속성일 때는 맨 앞에 \_\_(double underscore) 하나를 붙인다.
    - 스페셜 속성일 때는 앞과 뒤에 \_\_(double underscore) 하나를 붙인다.
* 모듈에서 정의되는 상수는 all capitalized words 형식을 따른다.
* 식별자(identifier)를 명확히 작성해야 하는 이유

파이썬은 변수, 함수명, 클래스명 등을 정의하지만 내부적으로 이름으로 식별한다.

식별자의 이름으로 제일 먼저 구별하고 그 이름의 변수, 함수, 클래스, 인스턴스 들이 들어있는지를 확인한다. 그래서 명명규칙을 명확히 준수해야 명명규칙으로 정의되어 있는 것이 변수, 함수, 클래스, 인스턴스인지 식별이 가능하다.

### 1.2.3. 변수(Variable)

파이썬에서 변수는 다른 언어의 변수와 차이가 많다. 변수를 단지 이름으로 사용되면 재사용 가능한 것을 단순히 보관하는 역할만 한다. 이 말은 파이썬 내부에서 만들어지는 값을 식별해서 재사용해서 사용만 한다는 것을 의미한다.

파이썬에서 만들어진 값은 전부 객체이므로 이 객체가 만들어진 후에 객체를 명확히 재사용할 것인가를 결정하면 다음에 이 객체를 구별할 수 있도록 식별이 가능한 이름으로 변수를 정의해서 사용해야 한다.

프로그램을 작성할 때 변수는 단순한 이름이고 변수 내에 저장되는 것도 단순히 객체를 가르키는 레퍼런스(reference) 즉 객체가 어디에 위치한 지를 나태내는 주소이다. 실행 중에 변수 내의 값은 항상 변경이 가능하고 변경될 때마다 주소인 레퍼런스(reference)가 변경되어 실제 값인 객체가 어디에 있는 지를 가르키는 값만 보관하는 용도이다.

* 변수를 처리하는 예시



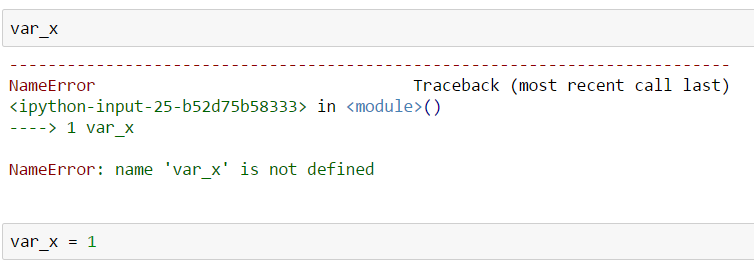
* 변수 정의

문자(알파벳, 숫자)와 언더스코어로 구성된 변수를 정의하면 반드시 값을 할당해서 초기화해야 변수가 정의가 되는 것이다.

예제 1-9 : 변수 정의

변수 var\_x를 정의하고 값인 1을 할당한 것이다. 파이썬은 변수만을 별도로 정의할 수 없기에 변수가 정의되면 할당문을 사용해서 값을 할당해야 한다.

변수만 정의하고 실행을 시키면 이 모듈이 Namespace에 이 변수가 들어가 있지 않으므로 이 변수를 찾을 수 없어 에외가 발생한다. 그래서 반드시 변수를 정의하면 할당을 해야 하는 것이다.

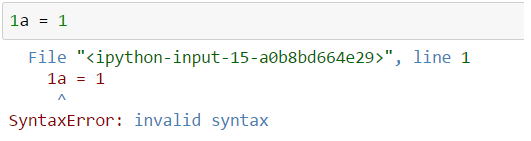


* 숫자를 첫번째 위치로 정했을 경우

명명규칙에 첫 자리는 반드시 문자나 \_(underscore)로 변수를 구성해야 한다. 숫자가 맨 처음에 오면 파이썬에서는 문장 구성의 오류인 Syntax 에러를 처리한다.

예제 1-10 : 변수 이름 오류

1a로 변수를 지정하고 값인 1을 할당해서 실행하면 문법적인 오류를 발생한다.



### 1.2.4. 함수(function), 클래스(class) 이름 관리 방법

함수나 클래스가 정의되면 함수와 클래스에 대한 이름을 내부 속성(\_\_name\_\_)으로 관리한다. 식별을 하기 위한 식별자와 이름이 같지만 실제 내부적인 처리규칙은 다르다.

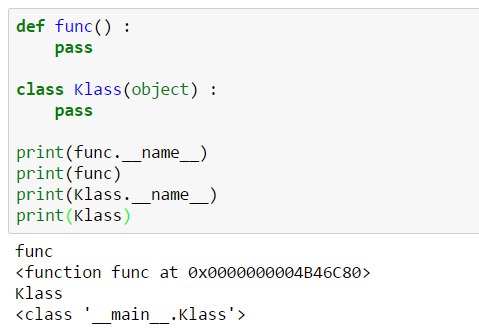
* 함수(function)명/클래스(class) 명

파이썬에서는 함수명, 클래스명을 정의하면 관리하는 기준도 두 개로 분리된다. 하나는 변수명으로 Namespace에 등록하고 실제 함수, 클래스, 모듈 이름은 객체 내의 \_\_name\_\_속성에 문자열로 저장한다.

예외는 익명함수인 lambda 인 경우만 \_\_name\_\_속성에 이름이 없다는 표시로 lambda를 등록해서 처리한다.

예제 1-11 : 함수와 클래스 정의 및 함수명과 클래스명 확인

변수와 실제 함수명과 클래스명을 구분해서 이해해야 한다. 정의된 함수와 클래스의 변수를 출력하면 함수와 클래스가 만들어진 객체가 출력되는 것을 알 수 있다.



위 예제 1-11은 함수 func와 클래스 Klass를 정의했지만 실제 기능을 정의하지 않아 pass문으로 정의했다. 실제 func과 Klass는 변수이고 이에 대한 실제 함수명과 클래스명은 별도의 \_\_name\_\_ 영역에 존재하는 것을 알 수 있다.

## 1.3. 데이터 모델(data model)

파이썬의 데이터 모델(data mode)은 실제 파이썬 내의 자료형(data type)을 어떻게 만드는 지에 대한 규칙을 관리하는 것이다. 파이썬은 모든 것을 값으로 관리한다.

모든 값은 리터럴이고 이는 자료형의 인스턴스(instance)인 값 객체를 생성하는 것이다. 자료형은 즉, 클래스(class)로 만들어지고 이 클래스에 대한 정보들이 어떻게 관리되는 지가 데이터 모델이 관리하는 규칙을 설명하는 것이다.이런 규칙은 자료형과 자료형에서 파생되는 객체들간의 관계를 명확히 관리하는 것이 데이터 모델이다.

지금부터 데이터 모델에 필요한 용어로 이해해 보겠다. .

### 1.3.1 클래스(class), 객체(object), 인스턴스(instance)

객체 지향 개념(object-oriented paradigm)에는 클래스는 객체를 만드는 틀로 사용되고 클래스에서 객체를 만드는 것을 인스턴스 만든다라고 정의한다.

파이썬에서는 클래스라는 명확한 개념이 있지만 이 클래스도 하나의 객체로사용될 수도 있다. 상세히 들여다보면 객체가 객체를 생성하는 구조로 작성되어 클래스 객체는 인스턴스 객체를 만드는 툴이고 일반적인 객체를 인스턴스 객체로 정의해서 사용되고 있다.

일반적으로 파이썬에서 객체라고 하면 클래스나 인스턴스가 다 해당이 되므로 클래스인지 인스턴스인지 여부를 확인이 필요한 경우도 존재한다.

클래스 객체와 인스턴스 객체 용어에 대해 알아보자.

* 객체 내부 검사(object introspection)

객체 내부에 대한 정보를 조회할 수 있는 함수나 클래스를 제공해서 객체 내부정보를 검사할 수 있도록 지원한다.

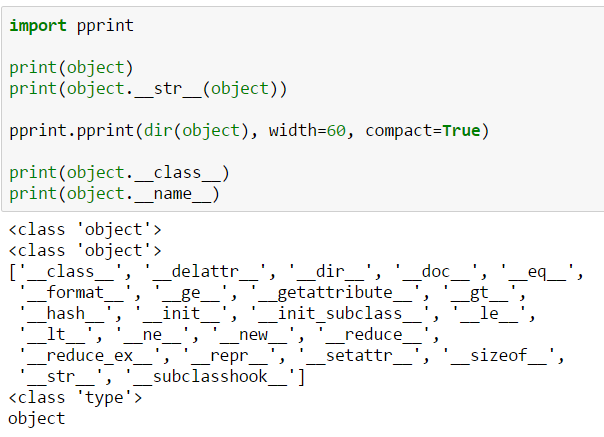
* dir함수 : 객체 내부의 속성과 메소드 확인
* type클래스 : 객체를 누가 만들었는지에 대한 정보를 확인
* id 함수 : 객체들이 레퍼런스를 정수로 확인
* 클래스 객체(class object)

파이썬에서 클래스 객체는 인스턴스 객체를 생성하고 인스턴스 객체의 속성에 접근할 수 있는 모든 메소드를 가지는 집합으로 볼 수 있다.

인스턴스 생성되면 인스턴스에 속한 속성만 관리하지만 인스턴스 객체에서 메소드를 호출하면 실제 클래스 내에 있는 메소드와 바인딩되어 처리되도록 구성되어 있다.

예제 1-12 : object클래스 내의 속성과 메소드 관리

파이썬에서 제공하는 객체 내부조사 함수인 dir을 이용해서 기본 클래스인 object 내의 속성과 메소드를 확인한다. 그리고 내부 속성인 \_\_class\_\_와 내부 메소드인 \_\_str\_\_을 실행해서 결과를 확인한다.



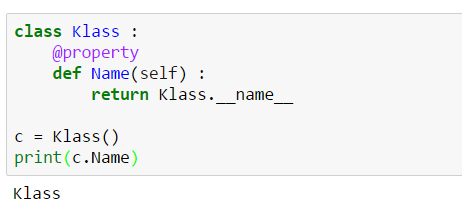
위 예제 1-12는 object를 print로 출력하면 <class ‘object’>로 출력한다. 클래스를 print 하라는 뜻은 \_\_str\_\_메소드를 출력하라는 뜻과 동일한다.

클래스 내의 있는 속성과 메소드를 확인하기 위해서는 dir 함수를 사용해 클래스 내부의 즉, object 클래스 내부의 스페셜 속성과 메소드들을 확인한다. 내부속성 \_\_class\_\_는 이 클래스가 어떤 클래스에 위해 만들어졌는지에 대한 정보를 출력하고 \_\_name\_\_은 실제 이 클래스의 이름을 출력한다.

예제 1-13 : 프로퍼티(property)

파이썬에서는 속성(attribute)와 프로퍼티(property)를 구분해서 사용한다. 프러퍼트는 객체의 행위에 연관되어 속성처럼 접근해서 사용되지만 내부적으로는 메소드가 작동되는 하나의 패턴이다.

주로 @property로 메소드를 감싸서 프로퍼티를 만들게 해준다. 프로퍼티를 설명하는 장에서 상세하게 설명하겠다.



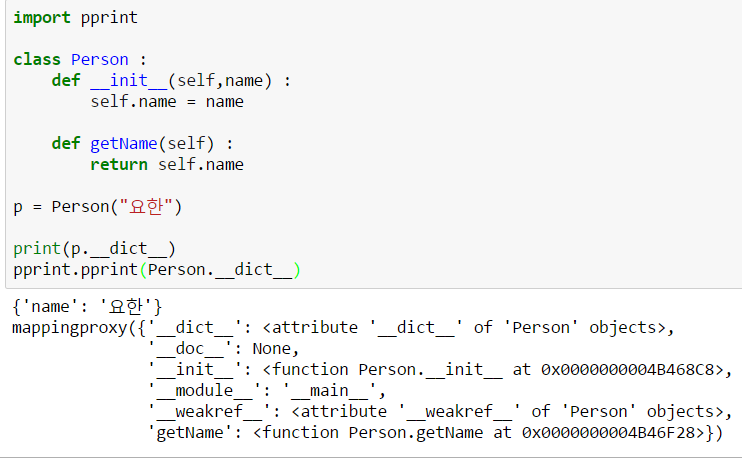
위 예제 1-13는 사용자 정의 클래스 Klass를 정의하고 Name이라는 메소드를 프로퍼티로 정의했다. 프로퍼티는 실제 이름으로 접근하므로 메소드 매개변수에 아무것도 부여하지 않았지만 실제 Klass.\_\_name\_\_를 출력하는 것을 볼 수 있다.

* 인스턴스 객체(instance object)

사용자 클래스(user defined class)를 정의하고 이 클래스를 가지고 인스턴스 객체를 생성해서 클래스와 인스턴스의 각 속성과 메소드를 접근해서 사용해 본다. 인스턴스 객체는 자기 속성과 클래스 객체의 모든 것을 접근할 수 있다.

예제 1-14 : 사용자 클래스에 대한 인스턴스 객체 확인하기

하나의 사용자 클래스 Person을 정의하고 이를 기준으로 인스턴스를 만들어서 내부를 조회하겠다. 일단 자세한 사용법은 클래스를 설명할 때 자세히 하기로 하겠다.



위 예제 1-14는 사용자 정의 클래스 Person을 정의하고 하나의 p 인스턴스를 출력했다. 실제 \_\_init\_\_ 메소드에 인스턴스에 들어갈 속성들을 정의했다.

클래스와 인스턴스 내의 Namespace는 \_\_dict\_\_ 속성을 가지고 있으며 이 속성을 조회하면 인스턴스에는 \_\_init\_\_ 내에 지정된 속성만 표시되는 것을 알 수 가 있다. Person 클래스에는 getName 메소드도 표시되어 있는 것을 확인할 수 있다.

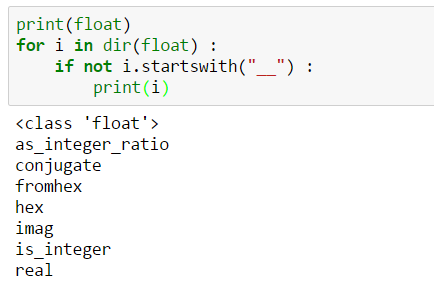
* 내장 자료형(built-in data type) 의 클래스 정보 확인

파이썬은 인스턴스 객체에는 인스턴스 객체가 가지는 기본 속성들만 관리하므로 실제 클래스 객체의 속성을 접근해서 메소드 등을 사용한다.

클래스의 정보를 확인하는 방법을 알아보자.

예제 1-15 : float 클래스 내부의 일반 속성과 메소드 조회

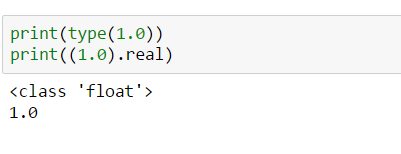
클래스 float 내의 속성이나 메소드를 확인해서 인스턴스 객체로 속성을 접근해서 실행한다.



위 예제 1-15는 float클래스의 내부 속성과 메소드 정보를 dir함수로 가져오고 실제 내부의 문자열의 앞 두글자가 “\_\_”를 제외한 정보만 출력했다.

예제 1-16: float 자료형의 값에서 real 속성으로 값보기

내장 자료형은 인스턴스 객체에 특별한 속성이 없다. 인스턴스 객체에서 속성을 접근하면 클래스 내의 속성이 연결되어 클래스 속성이 알려주는 값을 바로 처리할 수 있다.

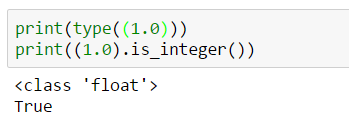


위 예제 1-16는 1.0이라는 float 객체의 인스턴스를 괄호로 표시하고 객체의 속성을 접근하는 객체접근연산자( . )를 사용하고 real클래스의 real이라는 속성을 조회하면 실제 값을 표시한다.

예제 1-17 : float 클래스의 인스턴스에서 메소드 호출하기

객체의 행위(behavior) 즉 상태의 변화를 발생시키는 함수이고 파이썬에는 3가지 메소드 (인스턴스 메소드(instance metho, 클래스 메소드(class method), 스태틱 메소드(static method))를 지원한다.

인스턴스에서는 이 모든 메소드를 전부 접근해서 사용이 가능하다. 상세한 사항은 클래스를 설명하는 곳에서 알아보기로 하겠다.



위 예제 1-17은 1.0 인스턴스 객체를 이용해서 float 클래스에 있는 is\_integer 메소드를 호출해서 실행한다. is\_integer 메소드가 하는 일은 이 실수 값이 바로 소수점을 제외하면 정수와 값이 같은지를 확인해 준다. 1.0은 정수 1과 같으므로 True로 결과값이 출력된다.

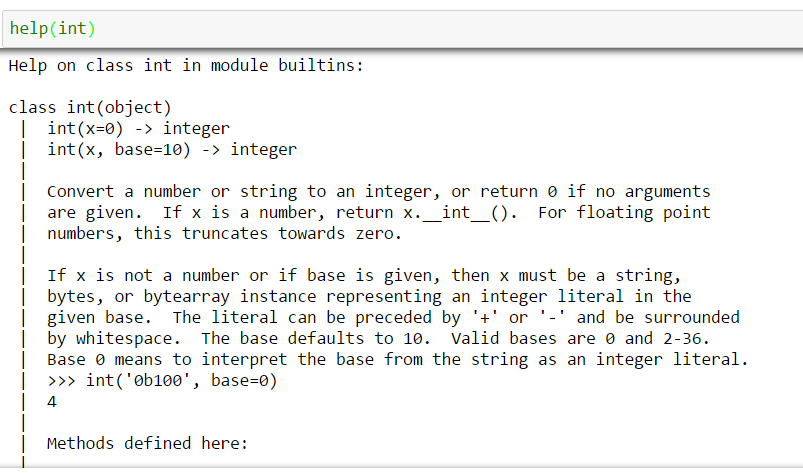
### 1.3.2 파이썬 기본 내장 함수 이해

파이썬 언어의 특징은 내장함수를 이용해서 프로그램 내부의 다양한 정보를 확인할 수 있다. 세부적인 문법을 알아보기 기본으로 이해해야 할 함수와 객체가 비교되는 부분을 먼저 알아보기로 하겠다.

* 내장 자료형 int에 대한 help 조회

정수 값들을 관리하는 int는 함수가 아닌 클래스이며 내부의 속성과 메소드들에 대한 정보가 출력된다.

예제 1-18 : int 클래스에 대한 세부 help 보기



위 예제 1-18은 int 클래스에 대한 기본 도움말을 출력한 것이다. 클래스 int로 인스턴스 생성할 때 하나의 값과 기준되는 단위 표시를 넣으면 해당되는 수를 10진법의 정수의 인스턴스로 만든다는 것을 알 수 있다.

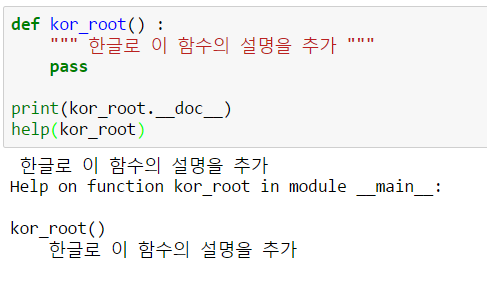
* 사용자 정의 함수에 help 함수가 실행을 위한 작성 기준

사용자 함수를 def 키워드를 이용해서 함수를 정의할 때 블록이 첫 번째 라인의 주석을 함수의 설명으로 인식한다.

함수의 설명은 “”” “”” (여러 줄을 작성하는 문자열)로 작성된 문자열로 주석을 달면 내부적으로 docstring으로 인지해서 함수 내부속성인 \_\_doc\_\_에 저장해둔다. Docstring은 help 함수로 조회하면 화면에 보여준다.

예제 1-19 : 함수 내부의 도움말 추가

함수를 정의하고 바로 아래에 함수에 대한 설명을 추가할 수 있다.



위 예제1-19는 함수 정의 헤드(def kor\_root() : )를 작성한 후에 body부의 첫뻔째 문장에 “”” “”” 문자열로 이 함수의 기능을 설명한다. 그럼 파이썬 내부에서 이 문자열을 이 함수의 내부 속성인 \_\_doc\_\_에 할당하고 나중에 함수 help로 호출하거나 \_\_doc\_\_ 속성을 직접 접근하면 저장된 것을 출력해서 보여준다.

* 함수 print 처리 예시

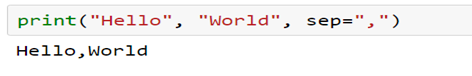
문자열은 두개 넣어서 출력하면 분리자(sep)가 기본으로 빈공칸(space)을 가지므로 겹쳐서 나오지는 않는다.

예제 1-20 : print 함수 출력

함수 print에 출력하려면 여러 개의 매개변수를 주면 쉼표로 분리해서 출력한다.



분리자(sep)에 쉼표를 “,” 넣고 출력하면 문자열 중간에 쉼표가 표시된다.



출력된 마지막에 개행 문자(줄바꿈 문자) 대신 \*\*\*를 넣으면 마지막에 출력된다.



### 1.3.3. 메타클래스(meta class) : type class

파이썬 언어 내에 모든 것을 객체는 클래스(class) 객체와 인스턴스(instance) 객체로 관리한다. 모든 클래스 객체에는 누가 만들었는지를 확인을 메타클래스에 의해 하고 또한 모든 인스턴스 객체는 누가 만들었는지를 클래스 객체에 의해 할 수 있다.

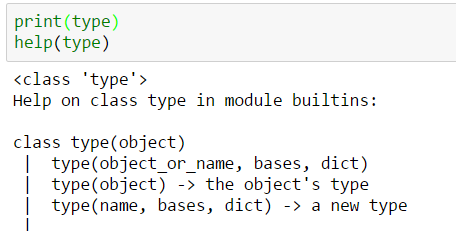
다른 언어에서는 별도의 자료형을 확인하는 함수가 있지만 파이썬에서는 이런 별도의 함수를 제공하지 않고 type 클래스로 이를 확인할 수 있다.

이번에는 간단히 메타클래스에 대한 기본을 알고 별도의 창에서 메타 클래스 설명을 하므로 상세한 내용은 나중에 설명한다.

* help함수를 이용해서 type class 조회

type은 함수가 아닌 class 객체이며 인자로 객체를 넣으면 어떤 클래스에 위해 만들어졌는지를 보여준다

예제 1-21 : 메타 클래스인 type 조회

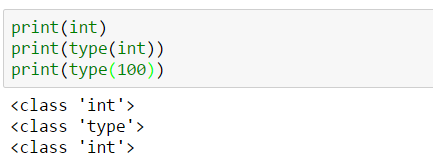


위 예제 1-21은 메타 클래스 type을 print로 출력하면 <class type>이라고 출력한다. Help로 type을 출력하면 type이 매개변수가 하나 일 경우 이를 누가 만들었는지를 보여준다. 3개의 매개변수를 만들면 새로운 클래스가 만들어진다는 것을 확인할 수 있다.

* 타입 체크하기

파이썬에서 타입 체크는 type 클래스를 이용해서 정보를 확인하는 것이다. 100 인스턴스를 넣고 조회하면 명확히 int가 출력되지만 int를 넣고 조회하면 type이 나온다. 이 부분은 뒤에 메타클래스에서 상세한 설명을 하겠다.

예제 1-22 : int 클래스와 100 정수 인스턴스 타입 체크



위 예제 1-22는 type 클래스의 매개변수로 클래스와 인스턴스를 주면 실제 누가 만들었는 지를 확인할 수 있다. type(int)가 <class type>이 출력되고 type(100)을 넣으면 <class int>로 출력하는 것을 확인할 수 있다.

### 1.3.4. 내장 자료형(built-in data type)

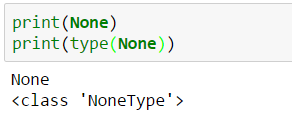
파이썬에서 기본으로 제공하는 자료형중에 기본으로 이해할 내장 클래스 객체(class object)에 대해 일부를 설명을 하고 가겠다. 기본적인 클래스이므로 가장 사용을 많이 하지만 관행대로만 사용하는 경우가 많아 명확히 설명해서 이해해 보기로 하겠다.

* 아무것도 없는 자료형(None data type)

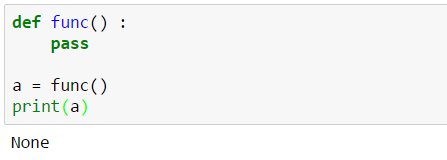
다른 프로그램언어에서는 Null로 많이 표현되지만 파이썬에서는 아무 일도 하지않거나 아무 것도 없다는 것으로 사용된다. 특히, 함수처리 후에 반환값이 없다면 None으로 처리하므로 많이 사용되는 자료형이다.

예제 1-23 : None 타입의 None 인스턴스 확인

None은 키워드 이면서 클래스 NoneType이 유일한 인스턴스이다. 실제 하나의 인스턴스로 사용되는 것을 잘 이해해야 한다.



함수 func을 아무것도 하지않도록 정의한다. 이를 실행한 것을 변수 a에 할당해서 출력해보면 None을 표시한다. 파이썬은 실제 함수가 아무일을 하지 않아도 결과로 None를 반환하는 것을 확인할 수 있다.

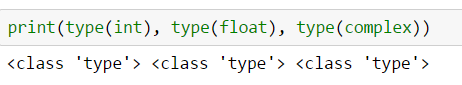


* 숫자 자료형(number data type)

원자(atom) 타입인 숫자 형식에는 정수(int), 실수(float), 복소수(complex) 클래스를 제공한다. 숫자 자료형의 int, float, complex가 클래스라는 것을 확인하기 위해 type을 이용해서 각 타입인 누구에 의해 만들어져 있는지를 확인한다. 모든 클래스는 메타 클래스에 의해 만들어져 있으므로 type 클래스라고 출력된다.

예제 1-24 : 숫자 클래스를 만든 메타클래스 확인

정수, 실수, 복소수를 처리하는 내장 클래스의 type을 알아보면 전부 type에 의해 만들어져 있다는 것을 확인할 수 있다.



* 시퀀스(Sequence) 자료형

문자열(string), tuple, list, 바이트(bytes), 바이트어레이(bytearray) 등의 자료형을 제공한다.

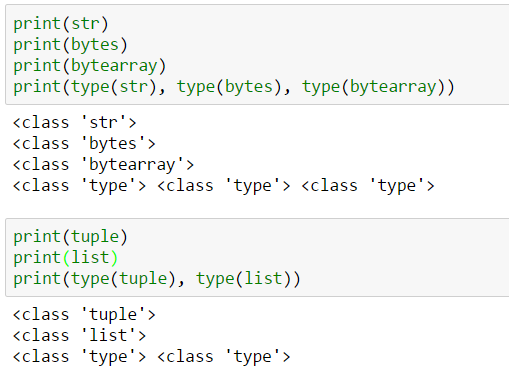
* Sequence의 특징

1. 동질성을 가지고 각 원소가 순서를 가지는 모임이다. 단, 리스트(list)만 동질성을 보다 상위 수준인 객체이지만 순서를 가지는 모임은 동일하다.
2. 순서와 원소가 정해진 형태도 있지만 순서와 원소의 수가 고정적이지 않을 수도 있다.
3. Sequence 의 형태를 유지하면 기본 연산은 동일하게 처리된다. 단, 각 자료형의 특징상 추가된 연산들도 존재한다.

Sequence 자료형의 str, bytes, bytearray, tuple, list가 클래스라는 것을 확인하기 위해 type에 인자로 넣어 실행해보면 메타클래스인 type이 나온다.

예제 1-25 : Sequence 자료형에 대한 메타클래스 확인

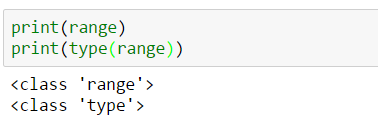
문자열, 바이트문자열, 바이트문자열의 배열, 튜플, 리스트는 전부 type에 의해 만들어진 것을 확인할 수 있다.



* 스트림(Stream) 형태 자료형

파이썬에서는 정적인 sequence도 제공하지만 동적인 Sequence 타입인 스트림(stream)제공한다. 여기에는 제너레이터(generator), 반복자(iterator)를 이용해서 만들고 한번에 하나씩만 처리되는 구조이다. 스트림(stream) 형태는 반복형(iterable)과 반복자(iterator) 설명하는 곳에서 자세히 알아보기로 하겠다.

예제 1-26 : range 클래스



위의 예제 1-26은 가장 간단한 Stream 자료형인 range를 확인해서 클래스라는 것을 알아보기로 하겠다. 메타 클래스인 type으로 누가 만들었는지를 확인해 보면 type이라고 출력되므로 range도 클래스라는 것을 알 수 있다.

* Mapping형태 자료형(data type)

사전형 타입(dictionary data type)인 딕셔너리(dict) 내장 자료형(data type) 제공한다. Mapping 자료형의 딕셔너리(dict)이 클래스라는 것을 확인하기 위해 type에 인자로 넣어 실행해보면 메타클래스인 type이 나온다.

예제 1-27 : 딕셔너리(dict) 클래스를 만든 메타 클래스

딕셔너리도 클래스이고 이를 print하면 클래스 정보가 나오지만 type으로 출력하면 type에 의해 만들어졌다는 것을 표시한다.

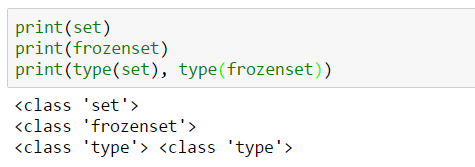


* Set형태 자료형

집합형(Set data type)인 set, frozenset 자료형을 제공한다. set 자료형에는 원소에 대한 변경이 가능한 set과 변경이 불가한 frozenset이 있다. 클래스라는 것을 확인하기 위해 type에 인자로 넣어 실행해 보면 메타클래스인 type이 나온다.

예제 1-28 : Set 클래스를 만든 메타 클래스

클래스를 출력하면 자기 자신을 보여주지만 type으로 클래스를 확인하면 메타클래스로 만들었다는 것을 출력한다.



### 1.3.5. 객체 비교 방식(is, ==), 레퍼런스 확인함수

파이썬은 모든 것을 객체로 인식하기에 객체들을 비교할 때 is 연산을 이용해서내부의 레퍼런스(reference)를 비교한다. 내장 클래스들의 인스턴스가 동일해야 할 때도 가끔 다른 레퍼런스를 제공할 수 있으므로 사용자 정의 클래스로 만든 인스턴스를 제외하고는 권고하지 않는 것을 알 수가 있다.

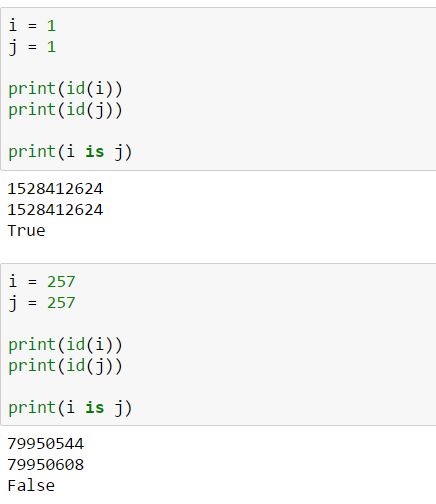
내장 클래스들은 주로 값을 가지고 비교하는 것을 권고하므로 == 연산자를 이용해서 처리한다. 그리고 객체들에 대한 레퍼런스 확인하는 함수인 id도 알아본다.

* 키워드 is

두 객체 동일한 객체인지를 비교할 때 is를 사용한다. 객체들이 레퍼런스를 비교해서 동일한지를 체크해서 True나 False로 제공한다.

특히 None 처럼 인스턴스를 비교할 경우에만 관행으로 is 연산을 사용한다.

예제 1-29 : 인스턴스의 주소 확인 및 동일한 객체 여부 점검하기



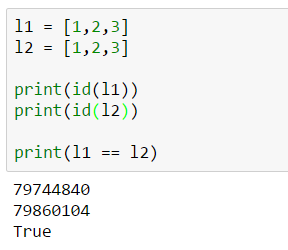
위 예제 1-29는 두 개의 변수를 정의하고 동일한 정수를 할당했다. 각 변수의 내부 객체의 레퍼런스(reference)를 id 함수로 조회해서 출력한다. 정수 1일 경우는 동일해서 True가 나오지만 257를 넣으면 False로 나온다.

내부적으로 동일해야 하지만 파이썬에서 처리할 때 256까지는 동일한 레퍼런스를 가지지만 이 수보다 큰 경우 다른 레퍼런스를 가지므로 is 연산으로 사용할 경우 원하지 않는 처리가 발생할 수 있다.

* 원소의 값이 동일 체크 : == 연산자, id 함수

두 인스턴스 객체의 원소들이 동일하다는 것을 비교해서 같으면 True를 표시한다. 내장 자료형들 중에 정수, 실수 등은 동일한 값이 동일한 레퍼런스를 유지해야 하지만 파이썬 내부적으로 두 개의 다른 인스턴스 객체를 만들 경우도 있으므로 객체의 비교를 ==연산자를 파이썬에서는 권고한다. 실제 객체들의 레퍼런스는 id 함수로 조회한다.

예제 1-30 : 인스턴스 내의 원소의 동등성 점검하기



위 예제 1-30은 두 개의 리스트를 정의해서 두 개의 리스트 인스턴스 객체의 레퍼런스(reference)를 출력한다. 두 개의 리스트의 원소들이 동일한 지를 == 연산자로 비교한다. 동일한 원소들로 구성되었으므로 True라고 표시한다.

### 1.3.6. 자료형의 변환(Mutation)

파이썬은 자료형의 변환은 다른 자료형으로 생성해서 점검하는 것을 말한다.

클래스 내에서 매개변수로 다른 자료형의 인스턴스를 받아서 처리하지만 변환이 불가능할 경우 처리가 되지 않고 예외를 발생한다.

자료형의 변환은 인스턴스 생성이므로 인스턴스를 생성할 때 어떤 값들이 들어가는 지를 명확히 이해하고 있어야 한다. 자료형이 어떤게 변환을 처리하는 지를 이해해 보겠다.

* 자료형 변경가능 여부

자료형에 대한 변경가능은 만들어진 인스턴스 객체가 바뀌는 것이 아니라 내부의 원소를 변경하는 것을 말한다. 자료형을 구성할 때 원자형(atom)인 숫자 자료형은 변경이 불가능하지만 분자형(molecule)은 원소를 추가나 삭제가 가능한지를 구분하는 원칙이다.

* 변경 가능 : Mutable

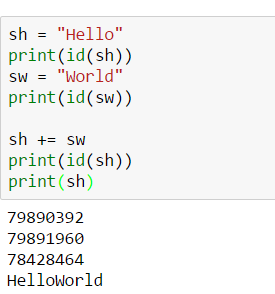
분자형일 경우 원소를 추가나 삭제 등이 가능한 리스트, 딕셔너리 등이 속한다. 사용자 정의 클래스는 여기에 속한다.

* 변경 불가능 : Immutable

원자형일 경우는 원소가 하나 밖에 없으므로 원소를 변경하는 것 자체가 객체를 대체하는 것이므로 변경이 불가능하고 분자형일 경우 문자열, 튜플 등변경이 되지않을 경우에 대한 효용이 있는 경우에 변경이 불가능하다.

예제 1-31 : 변경 불가능한 문자열 대체하기

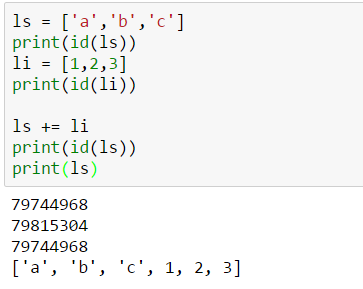
문자열은 변경이 불가능하지만 변수에 할당되는 것은 우측 표현식이 처리된 결과가 할당되는 것이므로 실제 인스턴스 객체가 아닌 다른 객체가 할당된 것을 알 수 있다.



위 예제 1-31은 sh변수에 할당된 문자열에 할당연산자(+=)를 이용해서 다른 문자열을 추가했다. 문자열은 변경이 불가능하므로 실제 새로운 문자열 인스턴스를 만들어서 변수 sh에 할당하는 것이다.

예제 1-32 : 변경 가능한 리스트에 문자열을 원소로 처리

리스트는 변경이 가능하므로 실제 변환이 발생하는 것이 아니라 원소가 추가된 것을 알 수 있다.



위 예제 1-32는 문자열을 리스트의 원소를 갖는 ls와 정수를 원소로 갖는 li을 정의하고 이를 ls에 할당연산자(+=)로 추가할 경우 기존 ls는 그대로 있고 내부의 원소가 추가되는 것을 확인할 수 있다. 변경이 가능하다는 것은 실제 내부의 원소가 변경되는 것을 알 수 있다.

## 1.4. 바인딩(binding) 및 평가방식(evaluation)

이름과 값이 언제 바이딩이 되는 지가 파이썬에서 중요한 개념이다. 실제 객체가전부 값이므로 이 값을 언제 변수에 바인딩되어 처리가 되는 지가 아주 중요하다. 바인딩하기 전에 표현식은 항상 평가를 수행하고 수행된 결과만 변수에 바인딩이 되므로 실제 계산된 결과인 값만 변수에 할당되는 것을 알 수가 있다.

파이썬에서는 모든 것을 엄격하게 평가하는 것이 아니라 특정한 문장에서는 실제평가를 호출하는 시점에 처리되는 경우도 존재하고 평가를 전체를 엄격히 처리하는 경우와 일부만 처리해서 결과를 내는 지연 평가도 존재한다.

실제 값을 처리하기 위해 어느 시점에 엄격한 평가가 필요한지 아니면 호출할 때 마다 지연 평가가 필요하는 지를 알아보겠다.

### 1.4.1. 바인딩에 따른 자료형 결정

파이썬의 핵심은 값(value) 즉 객체(object)와 이름(name) 즉 변수(variable)이고 실제 바인딩도 이름이 값에 바인딩되는 구조이다.

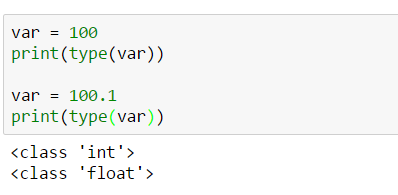
실제 변수에 보관하는 것은 값(value)의 레퍼런스(reference)이기에 바인딩 시점에 값이 연결되면 그 값이 자료형에 의해 변수의 자료형이 동적으로 변경된다.

* 정적 타입(static typing) 과 동적 타입(dynamic typing) 처리 방식

많은 프로그램 언어들은 변수를 정의할 때 먼저 자료형을 결정해서 처리하는 정적 타입과 바인딩할 때 들어오는 자료형을 보고 결정하는 동적 타입처리가 있다. 파이썬에서는 바인딩할 때 들어오는 자료형을 보고 결정하므로 동적 타입을 지원한다.

예제 1-33 : 동적 타입에 대한 변수 할당

변수가 할당될 때마다 자료형이 변경이 동적으로 이루어진다.



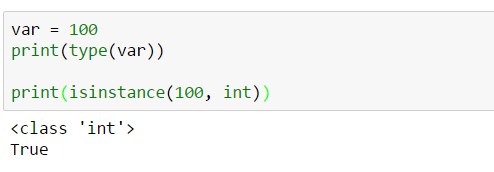
위 예제 1-33은 var 변수에 100 이라는 정수를 할당했고 이때 변수는 int라는 자료형을 가지고 있었다. 이를 100.1이라는 float으로 바꾸면 변수 var의 자료형도 자동으로 float으로 바뀌는 것을 볼 수 있다. 바인딩 시점에 변수의 자료형이 결정되는 동적 타입임을 알 수 있다.

* 강한 타입(strong typing) 과 약한(weak typing) 타입

데이터 모델이 명확히 정의되어 타입을 더 엄격하게 적용할수록 강한 타입(strong typing) 언어와 데이터 모델이 명확히 정의되지 않아 덜 엄격한 약한 타입(weak typing) 언어가 있다.

파이썬 언어는 데이터 모델을 확정해서 값으로 들어갈 수 있는 자료형을 강제한다. 그래서 강한 타입을 처리하도록 구성되어 있다. 단지 바인딩할 때마다 자료형이 변경되어 처리하므로 약한 타입을 사용하는 것처럼 느껴질 수는 있다.

예제 1-34 : 인스턴스는 모두 클래스인 자료형을 가지고 있음



위 예제 1-34는 변수 var에 100이라는 정수의 인스턴스가 할당되었다. 100이라는 정수는 int 클래스에 의해 만들어진 정수이므로 is instance 관계를 유지할 것이다. 그래서 isinstance 함수를 이용해서 100과 int 클래스간의 관계를 확인해 보면 True라고 나오는 것을 확인할 수 있다.

모든 값은 항상 만들어진 자료형을 가지고 있으므로 항상 강한 타입으로 처리되는 것을 알 수 있다.

### 1.4.2. 표현식(expression)의 평가 방법

파이썬에서는 표현식을 평가할 때 엄격한 평가와 지연 평가 두 가지를 전부 사용한다. 변수의 할당 등은 명확히 처리를 해야하므로 엄격한 평가를 하고 축약형 논리연산 등에 평가할 때는 덜 엄격한 평가를 사용한다. 또한 반복자(iterator), 제너레이터(generator) 등도 실제 호출할 때 평가해서 처리하는 지연 평가를 처리한다.

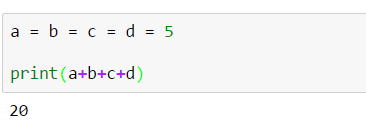
* 엄격 평가(eagar evaluation)

다른 표현으로 탐욕스러운 평가(greedy evaluation)라고 부르기도 한다. 대부분의 전통적인 프로그래밍 언어에서 사용되는 평가 전략이다.

엄격평가에서 표현식 등이 문장이 처리하는 것을 즉시 모두 처리한다.

예제 1-35 : 연속 변수 할당하기

한 문장으로 연속적인 변수 할당이 일어날 경우 이 문장은 한번에 모두 처리된다.



위 예제 1-35는 변수 a, b, c, d에 값 5를 할당한 것이다. 우측부터 좌측으로 평가되어 변수에 전부 5가 들어간다.

* 지연 평가(Lazy evaluation)

계산의 결과값이 필요할 때까지 계산을 늦추는 기법이다. 제너레이터 처리 방식은 호출할 때마다 로직을 작동하고 다시 호출될 대까지 처리가 되지 않아 지연평가를 잘 유지한다.

예제 1-36 : 파이썬 반복자 지연평가

내장 클래스 range는 반복형 자료형이다. 이를 반복자로 바꿔서 호출할 때마다 평가되어 결과를 처리해 보겠다.



반복형 range를 iter 내장함수를 통해 반복자로 만들고 next 함수를 통해 호출할 때마다 실제 값을 계산해서 하나씩 결과를 처리한다.

처리결과를 전부 처리하면 이 반복자는 StopIteration 예외를 처리하고 다시 평가가 되지 않는다.



### 1.4.3. interface 처리 규칙

파이썬은 내부는 모두 공개되어 퍼블릭(public)으로 관리한다. 실제 인터페이스를 위한 추상 클래스(abstract class)도 도입이 늦었다. 대신 상속이 없이도 인터페이스를 처리할 수 있는 덕 타이핑(duck typing)을 도입해서 protocol 규칙이 맞으면 처리가 가능하도록 사용해왔다.

덕 타이핑은 정적인 타입을 체크하지 않지만 규약만 준수하면 인터페이스라는 것을 인지해서 호출이 가능하도록 처리를 하는 특징을 가지고 있다.

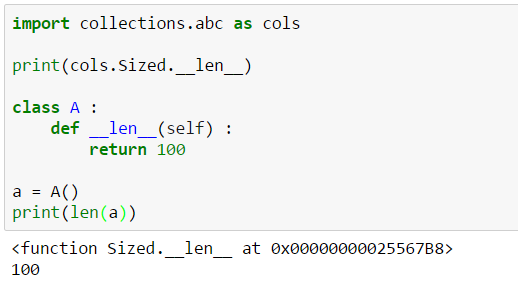
대부분 객체의 메소드들이 실행 처리는 타입에 상관없이 동일한 인터페이스인 메소드가 존재하면 실행하는 기법인 덕타이핑 (duck-typing) 규칙에 의해 처리되었다. 덕 타이핑은 비공식적으로 지정된 프로토콜인 인터페이스를 처리하는 기준이다.

세부적인 사항은 클래스를 설명하는 장에서 예제와 같이 설명하기로 하겠다.

예제 1-37 : 인터페이스 규약 이해하기

추상 클래스 Sized에 실제 \_\_len\_\_ 스페셜 메소드가 있다. 이는 내장 함수 len을 호출해서 처리하면 실제 인스턴스의 길이를 출력한다. 하나의 사용자 클래스를 정의하고 내부 인스턴스를 위한 \_\_len\_\_ 메소드를 정의하고 반환 값으로 100을 보낸다. 내장함수 len의 인자로 A클래스의 인스턴스 a를 넣어서 호출하면 100이라는 것을 출력하는 것을 확인할 수 있다.

실제 Sized 클래스를 상속하지 않아도 \_\_len\_\_을 처리하는 것을 알 수 있다.



## 1.5. 요약

파이썬 프로그램 언어는 인터프리터 언어이지만 이 프로그램을 실행하기 위해 기본적인 규칙을 가지고 있다. 이 규칙에 대한 기본 용어와 어떻게 사용하는 지에 대해 알아봤다.

이번 장에서 알아본 것은 앞으로 계속 사용되므로 이 기본 용어를 잘 이해해야 한다.

* 값 처리 기준 : 리터럴, 표현식, 조건식
* 이름 처리 기준 : 예약어, 변수 명명규칙, 변수 할당, 함수와 클래스 명명
* 데이터 모델 : 클래스, 객체, 인스턴스 등 자료형 처리 규칙, 자료형 즉 클래스를 만드는 메타클래스
* 바인딩하기
* 표현식 평가
* 인터페이스 규칙(덕 타입핑)

**2. 파이썬 문장 이해하기**

파이썬을 실제 실행하고 제어하는 기능들이 추가되려면 이를 문장(statement)으로 구성해야 한다. 여러 문장들로 구성이 필요할 경우 이를 블록(block) 단위의 문장인 제어문 이나 순환문 등로 구성한다.

하나의 기능으로 문장을 만들어야 하는 경우는 함수로 정의해서 사용할 수 있고 데이터와 기능을 하나의 단위로 정의할 경우에는 클래스로 정의해서 사용할 수 있다.

함수나 클래스 등을 하나로 묶어 모듈로 만들어서 재사용할 수 있도록 작성할 수 도 있다.

이 장에서는 문장들이 사용에 대해 알아보겠다.

## 2.1. 문장 꾸미기

문장을 하나만 작성할 수도 있고 문장의 그룹인 블록을 구성할 수 있다. 블록으블 구성된 문장은 블록( : )을 표시해야 한다. 블록구문은 이후의 문장은 줄 들여쓰기를 사용해야한다.

그리고 한 라인이 넘는 경우 멀티라인 작성, 인용(quotation) 과 주석(commemt)에 대한 기본 사항부터 알아보기로 하겠다.

### 2.1.1. 줄 들여쓰기 (Lines and Indentation)

파이썬 프로그램 시 첫번째 주의 사항은 코드 블록에 대한 세부적인 기호가 시작할 때가 있다. .

클래스와 함수, 제어문과 순환문 등의 블록처리는 콜론( : )이고 이 다음의 블록을 인지하려면 줄 들여쓰기를 사용해서 구분한다.

{ }, ( ) ,[ ] 기호로 시작할 경우 여러 줄로 작성되어도 하나의 문장으로 인식하기에 들여쓰기가 인식되지 않으며 줄 연결 기호( 역슬래쉬 \)를 사용해도 아직 한 문장이 끝나지 않았으므로 들여쓰기가 인지되지 않는다.

들여쓰기는 공백(Space), 탭(Tab) 모두 가능하지만 단, 들여쓰기 공백 수는 블럭내에 동일 수로 해야 한다.

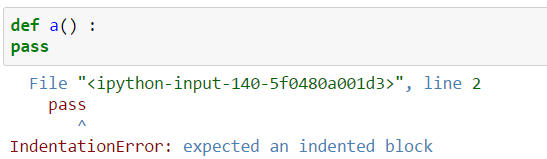
* 문장의 블록 만들기 : 4칸 들여쓰기

: 는 블럭구문을 하는 표시이므로 반드시 이후에 작성된 문장은 4칸을 들여쓰기를 권고한다. 그리고 대부분 개발 툴이 4칸 들여쓰기를 지원한다.

예제 2-1 : 함수 블록 내의 들여쓰기 오류

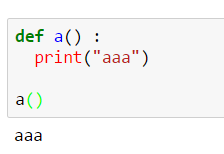
함수 정의시 내부 문장은 함수 블록 다음에 들여쓰기를 해서 작성해야 한다.

블록 구문이 표시가 된 이후에 들여쓰기를 하지 않으면 IndentationError가 발생한다.



예제 2-2 : 함수 블록 들여쓰기를 일부만할 경우

4칸 들여쓰기는 권고사항이지만 블록 구문에서 들여쓰기를 일부분만 할 경우에도 실행이된다. 하지만 명확히 구분을 위해 들여쓰기를 명확히 해야 한다.



### 2.1.2. 멀티 라인 문 (Multi-Line Statements)

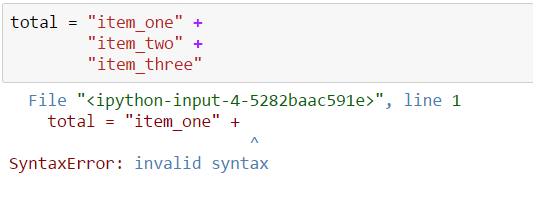
파이썬은 일반적으로 새로운 줄이 되면 끝난다.그러나 줄 연속 문자(\)의 사용으로 줄을 계속 사용할 수 있다.

* 역 슬래쉬 사용해서 문장 연결하기

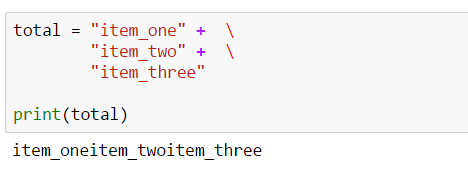
표현식을 사용할 경우 여러 문장을 하나의 문장으로 표시할 때 역슬래쉬를 이용해서 하나의 문장으로 만든다.

예제 2-3 : 문장을 하나로 연결하기

멀티 라인으로 작성이 필요할 경우 멀티라인을 표시하지 않으면 SyntaxError를 발생한다.



멀티라인이라는 표시를 문장의 마지막에 붙이면 하나의 라인으로 인식해서 처리된다.

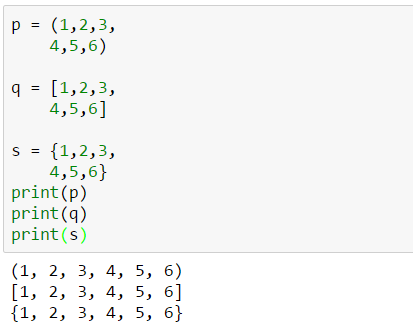


* ( ), [ ], { } 일 경우 역 슬래쉬 없이 문장이 연결

이 기호가 사용되면 닫힌 표시가 있기전까지는 하나의 문장이 끝난 것으로 보지 않기에 닫혀야 문장으로 인식한다.

예제 2-4 : 하나의 문장으로 자동 연결하기

괄호, 대괄호, 중괄호는 쉼표에 의해 멀티라인도 자동으로 하나의 라인으로 인식해서 처리된다. 별도의 멀티라인이라는 표시가 필요하지 않는다.



### 2.1.3. 인용 (Quotation)

single('), double("), triple(''' or """) 를 사용하여 문자열을 감싸서 사용한다.

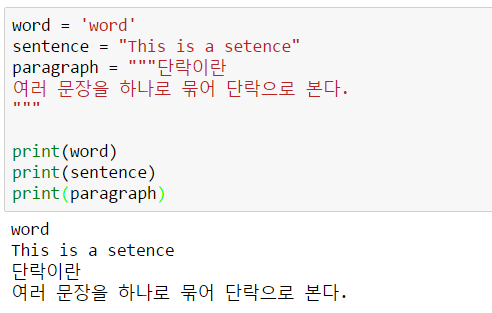
특히 triple 따음표는 여러 줄에 걸쳐 문자열을 작성할 때 사용되면 함수 등의 doctsring 작성 할 때 사용된다.

* 인용 : 문자열 처리

인용은 코딩시 문자열을 표시하는 방법이다. 3가지 방법을 이용해서 문자열을 출력해 보겠다.

예제 2-5 : 문자열을 정의하기

문자열을 표시할 수 있는 인용표시도 하나의 문장과 여러 문장을 표시할 수 있다.

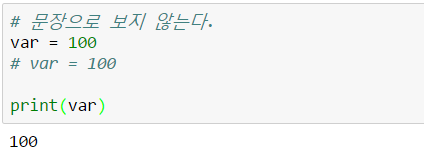


### 2.1.4. 주석 (Comments)

기호 (#)를 사용해 주석을 시작하며 기호 (#) 후 물리적 라인 끝까지 모든 문자를 주석으로 간주한다. 주석은 프로그램 실행시 아무런 행위도 하지 않고 단지 프로그램에 대한 설명으로만 인식한다.

예제 2-6 : 주석문 달기

주석은 문장으로 보지 않기 때문에 주석을 표시하는 #을 사용하면 이 다음에 있는 문장을 주석으로 처리한다.



## 2.2. 프로그램 문장

프로그래밍에서는 다시 사용하기 위한 방법으로 문장(Statement)을 제공한다. 단순히 값만 처리하면 한번 처리된 결과를 다시 사용할 수 없다. 문장은 은 리터럴과 표현식 등이 조합으로 완전한 처리결과를 할 수 있는 구조이다. 표현식은 평가가 되어 하나의 값만을 나타내고 재사용 될 수 있는 처리가 없지만 문장은 재사용이 가능하도록 상태를 보관하는 기능도 포함된 프로그램언어의 로직을 처리할 수 있는 구조를 말한다.

### 2.2.1. 할당(assignment) 문장

파이썬에서 할당문이 중요한 이유는 리터럴과 표현식이 결과를 변수에 할당할 경우 그 변수가 이전에 정의가 없으면 처음으로 변수가 초기화 된 것으로 본다.

* 변수에 리터럴 할당(변수 정의)

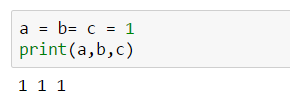
직접 값을 변수에 할당하게 표현이 되었지만 실제는 변수에 인스턴스 레퍼런스만 할당해서 이 인스턴스 객체를 찾을 수 있는 참조만 할당한다.

예제 2-7 : 변수에 정수 할당

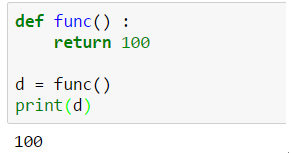
변수 하나에 대한 할당은 하나의 변수에 대한 초기화 처리이다.



멀티 할당은 변수에 동일한 값을 정의할 때 사용할 수 있다.



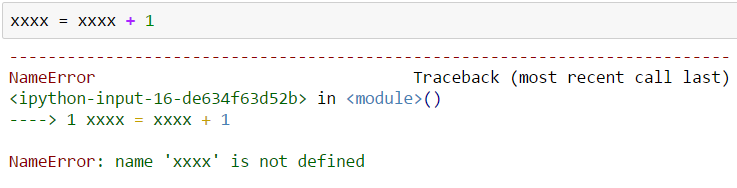
함수를 처리하고 결과를 반환해서 변수에 할당할 수 있다.



예제 2-8 : 변수 초기화 할당이 중요한 이유

파이썬에서는 변수에 대한 초기화 문법이 별도로 없다. 반드시 할당해야 초기값이 할당되어 변수가 정의가 되는 것이다.

변수의 할당없이 처리되면 그 변수를 찾을 수 없다고 에러를 먼저 발생하므로 반드시 변수가 정의되면 할당을 해야 한다.



### 2.2.2. 단순 제어문

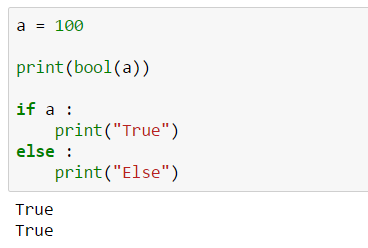
제어문은 표현식인 조건식을 평가해서 참과 거짓을 평가해서 그 결과에 따라제어문을 처리하는 것을 말한다.

* 조건식의 평가 처리 : True/False로 처리

if문은 True만 처리하므로 True 값이 전달되면 if 블록만 처리되고 아니면 else 구문을 처리한다.

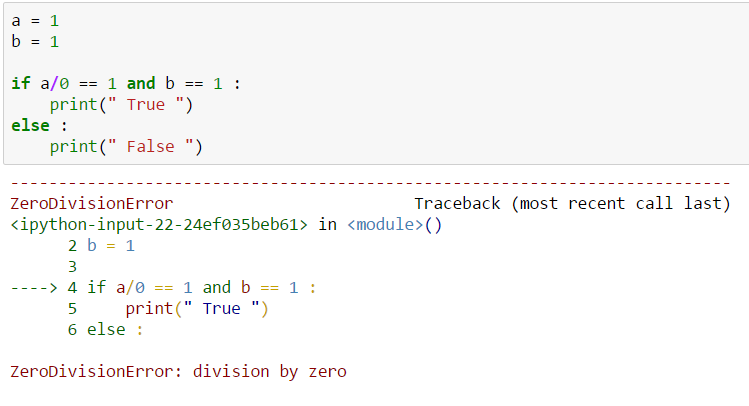
예제 2-9 : 조건식의 평가 결과는 True/False

조건식으로 변수 a를 할당했지만 변수 a의 값을 bool 자료형으로 인스턴스를 생성해 보면 True 값이 처리되는 것을 볼 수 있다. 제어문에서 조건식을 평가하면 True이기에 출력도 True로 표시한다.



예제 2-10 : 조건식 평가 시에 에외발생

조건식 평가(condition expression evaluation)할 때 예외가 발생하면 조건식 평가가 제대로 안되고 예외를 처리가 우선순위가 먼저로 처리된다.

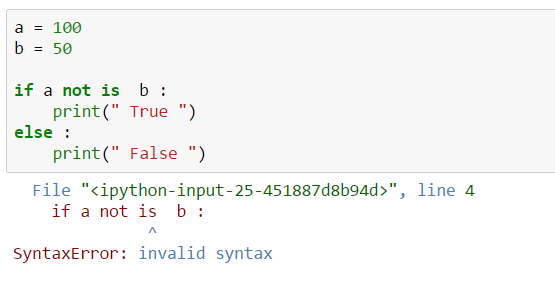


* 조건식을 사용할 때 주의할 점

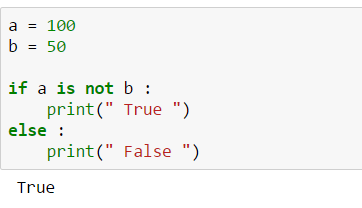
긍정표현식의 부정보다는 인라인 부정( 변수 is not 변수)를 사용한다. 길이를 확인하는 것보다 빈값은 false 의미이므로 not 변수명으로 사용한다.

예제 2-11 : 인라인 부정처리

인라인 부정시 주의할 점은 is 앞에 not을 사용하면 SyntaxError가 발생한다.

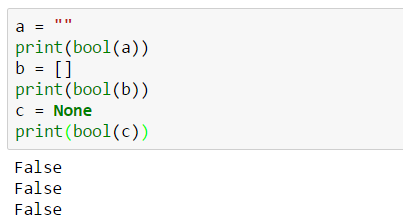


키워드 is 다음에 not을 넣어서 처리하는 것이 좋다.

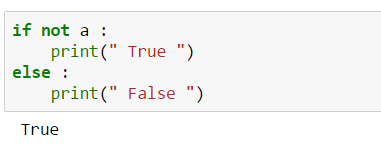


예제 2-12 : 빈 값을 False로 처리

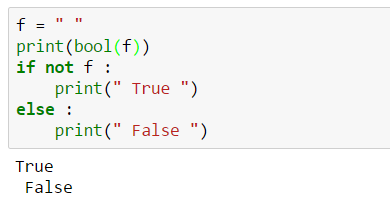
False로 처리하는 경우에 대한 빈값은 모두 False로 평가되어 처리된다.



제어문에서 빈칸을 넣으면 bool 평가결과가 전부 False로 처리된다.



공백 문자가 들어간 문자열은 실제 문자가 있기 때문에 True로 표시하므로 not으로 평가하면 False가 나온다.

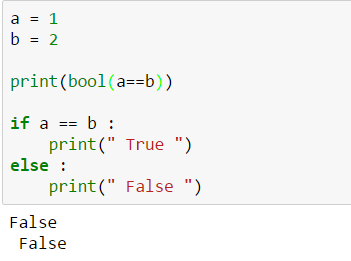


* 값을 비교연산으로 처리

제어문을 이용해서 조건식에서 값에 대한 비교를 연산할 경우 실제 비교된 값에 대한 참과 거짓을 평가한 후에 결과에 따라 if문이나 else 문에 정해진 로직을 처리한다.

예제 2-13 : 값에 대한 동등성 비교

두개의 변수에 들어간 값들이 동일한지 여부는 ==을 사용해서 동등성을 비교해서 처리한다.

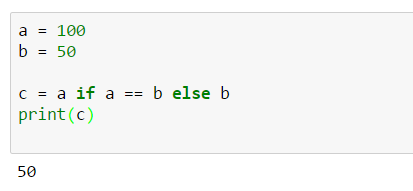


* 삼항 연산 처리(Ternary operators)

파이썬에서는 문법적으로 삼항연산자가 별도로 없고 if문을 이용해서 조건식이 참일 경우와 거짓일 경우를 처리하는 삼항연산를 만들 수 있다. 대신 블럭구문이 없어야 하므로 콜론( : )이 없는 것을 볼 수 있다.

예제 2-14 : 삼항연산자 : 인라인으로 if문 사용

두 변수가 있고 이 변수가 동일한 값을 가지면 a를 아니면 b를 처리한다. if문을 조건식이 참인 경우 뒤에 인라인으로 작성해서 else일 경우에 아닌 경우를 넣어서 처리한다.



### 2.2.3. 복합 제어문

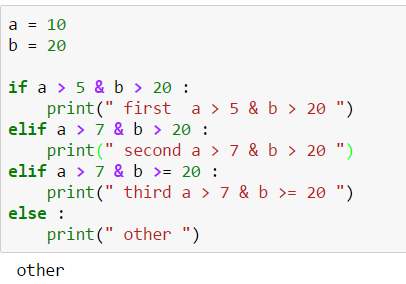
복합 제어문은 여러 개의 조건식이 발생할 경우 이 조건식 중에 하나의 만족하는 결과를 처리할 때 사용한다. 문법적은 elif 구문이 추가되며 이 문장 다음에 조건식을 명기해서 평가를 처리해서 해당되는 결과를 처리한다.

* 복합 제어문 : elif

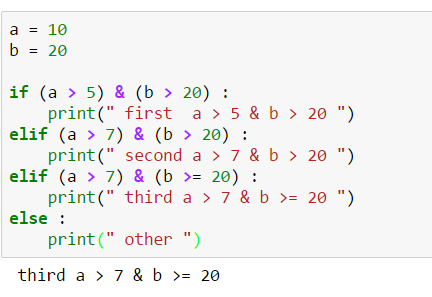
다양한 조건식을 사용하고 싶을 경우 elif문에 조건을 추가해서 로직을 넣는다. 해당 if 조건이 만족하지 않으면 그 다음에 정의 된 elif 조건식을 평가하고만족하면 내부 블록의 로직을 처리한다.

예제 2-15 : 조건식 세분화하기

비교연산과 논리 연산 처리시 주의해야 할 점은 실제 연산자 우선순위에 따른 결과값이 다르게 나올 수 있다.

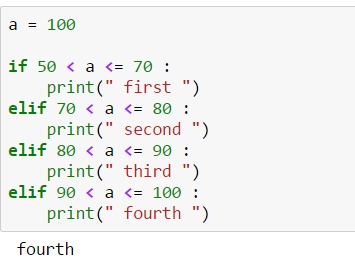


괄호로 우선 순위를 맞추고 처리하면 조건식 중에 해당되는 것을 만족해서 출력을 한다.



예제 2-16 : else문 대신 elif문으로 명확히 하기

조건식을 작성할 때 실제 논리 연산을 제외하고 비교연산을 연속해서 사용이 가능하다. 마지막 조건을 else문을 사용하지 않고 elif문으로 처리를 확정해서 사용하는 경우가 더 명화해서 좋다.

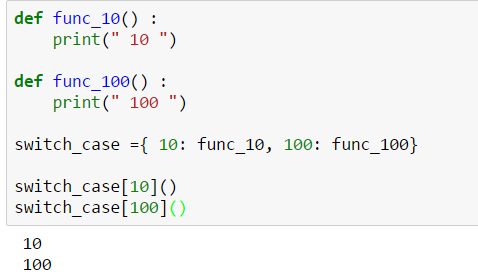


* 스위치 케이스(switch case) 문 처리 방법

파이선에는 스위치 케이스(switch case)문을 제공하지 않으므로 복합제어문을사용해야 한다. switch case문의 특징인 조건식이 평가되면 특정 case만 선택되어 블록을 실행하 하나의 case만 실행하므로 이 처리절차를 이해해서 특정 조건을 딕셔너리(dict) 타입의 키로 실행로직을 함수로 정의해서 값으로 넣는다. 실행 호출할 때는 키값이 만족하는 것을 찾아서 실행하도록 로직화하면 된다.

예제 2-17 : 딕셔너리의 값으로 함수를 넣어 케이스 처리

함수 2개를 정의해서 딕셔너리에 넣어서 특정 경우가 오면 딕셔너리에서 꺼내서 처리할 수 있다. 이렇게 처리하면 특정 조건이 만족하는 경우 하나의 기능을 처리하므로 여러 조건 중에 하나만을 처리하는 switch-case문 처리와 동일한 기능이 가능하다.



위 예제 2-17은 a\_10과 a\_1 두 함수를 정의해서 switch\_case에 딕셔너리할당 할 때 값으로 넣었다. Switch\_case를 정수 검색으로 값을 가져오고 실행연산자인 괄호로 호출하면 실제 함수가 실행되는 것을 알 수 있다.

### 2.2.4. 순환문 처리 : for

파이썬 for문은 다른 언어와 차이점은 반복형(iterable)/반복자(iterator) 자료형의 객체를 처리할 수 있다는 것이다.

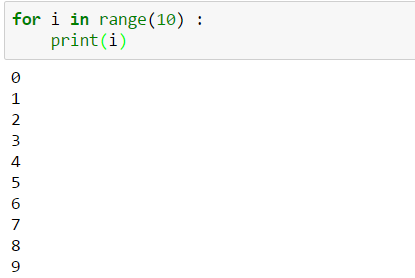
for문에 반복형이 들어오면 반복자를 생성시켜서 원소 하나씩 순환해서 처리하도록 해준다.

반복형이나 반복자가 원소가 다 처리되면 StopIteration 예외 처리를 하지만 for문 내에서는 이를 마지막으로 인식하기 때문에 예외처리 없이 처리가 가능하다.

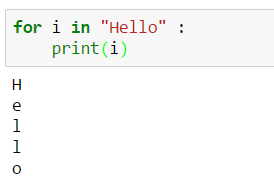
문자열, 리스트, 튜플, 딕셔너리, 파일 등 반복자나 반복형 등의 타입이 온다.

예제 2-18 : 반복형 range로 처리

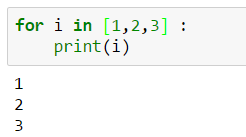
for문에 반복자인 range를 주면 자동으로 10개의 원소를 처리한다.



문자열을 넣어도 for문이 순환되면서 하나의 문자씩 출력한다.



리스트가 주어질 경우도 원소 하나씩 출력하는 것을 볼 수 있다.



### 2.2.5. 순환문 처리 : while

while 문은 for문과 달리 특정 조건식이 만족할 때까지 처리되는 순환문이다.

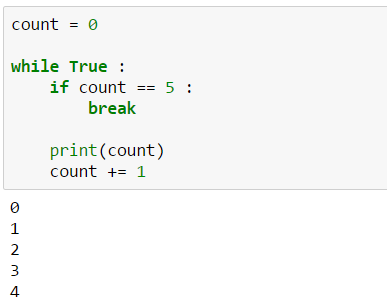
* True로 무한순환

while문은 조건이 참일 경우에만 내부 블록이 실행되므로 while의 조건을 True 지정하면 무한 순환이 되어 처리하므로 내부에 종료할 수 있는 로직을 처리한다.

무한 순환은 프로그램 상에 문제가 발생하므로 반드시 무한순환이 종료될 수 있는 로직을 추가해야 한다.

예제 2-19 : 무한순환과 특정 값일 경우 순환을 중단하기

While문 조건식에 True로 배정하면 무한순환이 처리된다. 내부에 특정 제어문을 만들어서 그 조건이 만족할 경우 break문을 사용해서 순환문에서 벗어난다.



### 2.2.6. continue/break

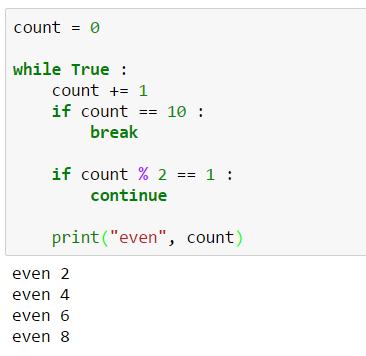
순환문 등에서 특정 기능을 처리하지 않거나 중단하려면 continue문과 break 문을 사용한다.

* continue : 순환 처리는 해야하지만 특정 로직을 처리하지 않을 경우 순환문으로 돌아감
* break : 순환처리시 순환을 종료하는 경우 사용
* 순환문에서 continue와 break 처리 예시

While 문으로 무한 순환을 시키고 내부에서 특정 조건을 부여해서 로직을 스킵할지 계속할지를 처리하도록한다.

예제 2-20 : 무한순환시 특정값 배제 및 순환을 강제 종료하기

짝수에 대해서만 출력하기를 while 순환문으로 작성할 때 홀수는 continue로 제외하고 출력하면 실제 처리가 된다.



### 2.2.7. else 문 추가하기

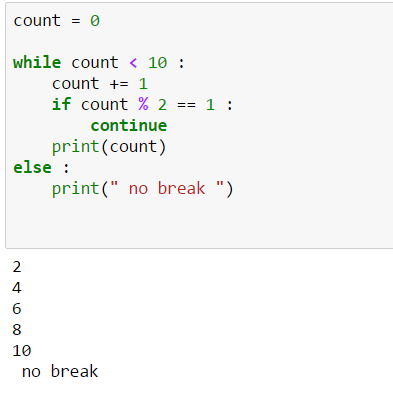
파이썬에서는 다른 언어와 달리 순환문에 else 문을 사용할 수 있다. 순환문에서 특정하게 빠져나오는 break 처리가 아닐 경우에 대해 else문 내의 기능을 출력할 수 있도록 처리하면 순환문 등이 기능이 정상적으로 처리가 끝났다는 것을 확인하고 다른 기능들도 추가할 수 있다.

* 순환문에서 break 없을 경우 예시

순환문에 else 문을 첨가하면 순환 작업이 break 없이 처리되면 그 다음 로직을 처리할 수 있는 부문을 이 블록 내부에 정의해서 실행하도록 처리한다.

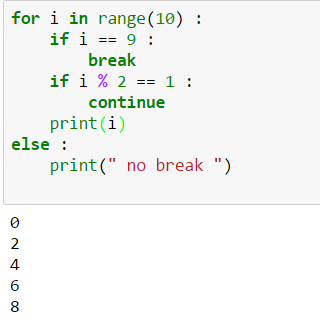
예제 2-21 : 순환 처리 시에 강제 중단 없을 경우 추가 처리

break문이 발생하지 않으면 else 문 내의 로직도 처리되는 것을 볼 수가 있다.



예제 2-22 : 순환문에서 break 발생할 경우

순환문 for문에서 break와 continue 문을 전부 사용할 수 있다. 특정 조건이 만족할 경우 break문을 발생해서 순환문이 종료되어 else문이 처리되지 않을 것을 확인할 수 있다.



### 2.2.8. global 문 과 nonlocal 문 처리

Global과 nonlocal 키워드는 특정 변수 앞에 작성하면 그 변수가 위치한 Namespace의 위치를 지정하므로 실제 로직에서 변수를 참조할 때 특정 영역(global은 모듈, nonlocal은 상위 함수)을 확인한다.

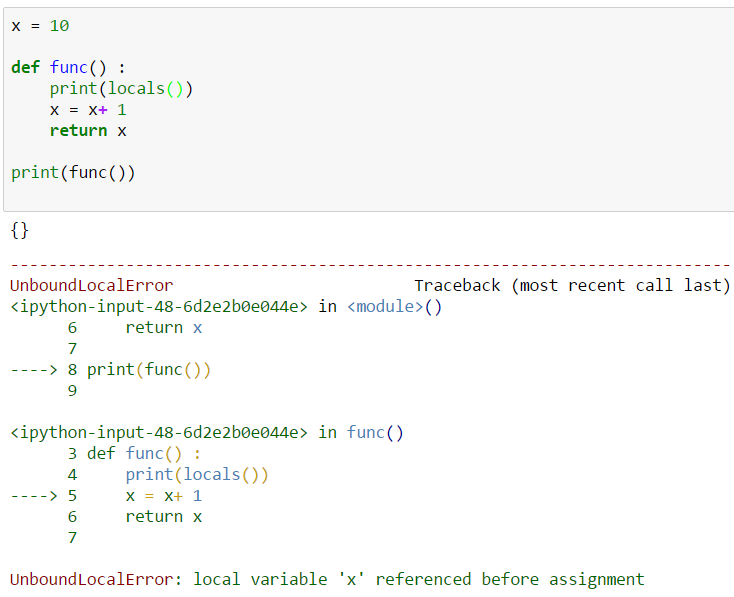
global 문은 주로 함수 영역에서 모듈의 전역 영역의 변수를 참조해서 갱신이 필요할 경우 정의해서 사용한다. nonlocal문은 주로 함수 내에 함수를 정의할 경우 내부 함수에서 외부 함수에 있는 변수를 갱신할 경우 정의해서 사용한다.

* 함수 내에서 지정되지 않는 변수변경

파이썬에서는 변수 정의는 변수에 값을 할당하는 것이다. 표현식에 사용되는변수가 처음으로 사용되어지면서 미리 할당을 하지 않으면 그 변수를 전역 영역에서 검색하고 없으면 에러가 나온다.

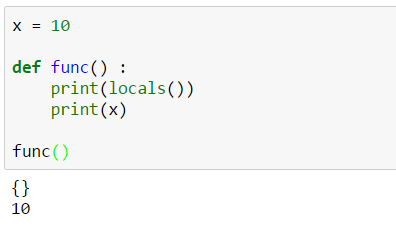
예제 2-23 : 함수 내에 표현식에 할당되지 않은 변수가 존재

함수 내에서 x = x+1에서 x+1이 처리될 때 x 내의 값이 갱신이 되어야 하므로 일단 함수 내의 지역영역의 x 변수를 검색했지만 실제 없어서 지역지역에 없는 변수를 갱신한다고 에러를 발생했다.



예제 2-24 : 함수 내에서 전역 변수만 조회할 경우

함수 내에서 단순히 전역 변수를 조회하면 지역영역에 없는 것은 전역영역에서 검색해서 처리하는 것을 볼 수 있다.

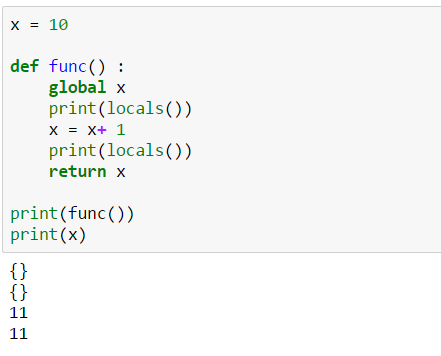


* 키워드 global를 사용

함수 내에 변수를 할당하면 지역 변수로 인식되므로 전역 변수를 처리하지않는다. 갱신 불가한 전역 변수를 함수 내부에서 갱신하고 싶을 경우는 명시적으로 global 키워드로 변수를 정의하고 갱신해야 한다.

예제 2-25 : 함수에 없는 전역 변수를 지정해 사용하기

전역 변수를 가지고 갱신이 필요한 경우는 명확히 global 키워드를 사용하기 전에 지정해야 한다. 전역변수를 가져와서 사용할 경우 지역변수는 없지만 전역변수를 가지고 처리한 결과는 전역에 전부 상태가 보관되는 것을 볼 수 있다.



* 내부함수에서도 변수 할당 없이 갱신하는 경우

함수 내부에 함수를 정의할 경우에도 지역 변수가 별도로 생성되므로 변수 할당 없이 사용하면 변수가 정의가 되지않고 사용하는 것이므로 예외가 발생한다. 예외 발생을 해결하려면 변수를 할당해서 정의한 후에 사용해야 한다.

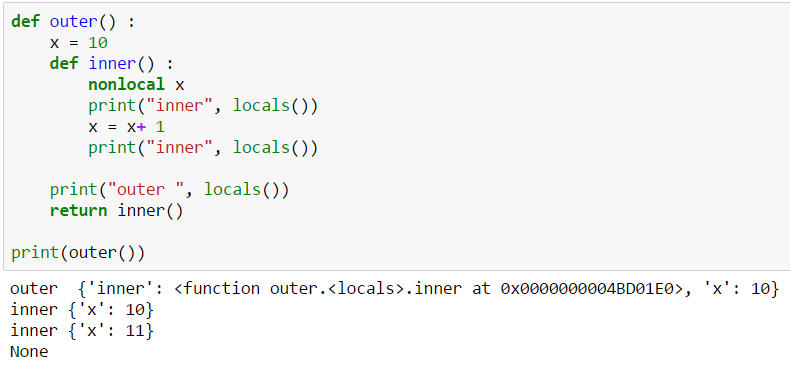
예제 2-26 : 내부 함수 내에 미지정된 변수 사용

외부함수 outer를 정의하고 내부 변수로 x를 지정했다. 이 외부 함수의 지역에 내부함수 inner를 정의하고 변수 x를 갱신했다. 함수 내부에서 갱신을 할 때 일단 함수 지역변수에 있는 지를 확인하는 데 실제 지역에는 x라는 변수가 없어서 예외가 발생했다.



예제 2-27 : 내부 함수 내에 미지정된 변수를 nonloacl 지정

inner 함수 내에 미정의된 변수를 nonlocal 키워드를 이용해서 inner 함수에 정의된 변수가 아니라는 것을 명확히 표시했다. outer 함수에 x가 정의되어 있으므로 x = x+1에서 x라는 변수를 처리시 outer 함수의 x를 찾아서 처리한다. 실제 처리된 결과는 inner 함수 내의 지역에 저장되어 변경된 것을 확인할 수 있다.



## 2.3 함수 및 클래스 정의문 처리

파이썬에서는 함수나 사용자 클래스를 사용하기 위해서는 정의를 먼저 해야 한다. 함수와 클래스 정의하는 법을 간단하게 알아본다.

### 2.3.1. 함수 정의문

함수를 정의하는 이유는 파이썬 내부적으로 이 정의를 보고 함수를 function 클래스의 인스턴스로 만들어 함수가 호출될 때마다 하나의 인스턴스로서 역할을 할 수 있도록 만들어 준다.

* 함수(function) 정의문

함수가 호출되면 파이썬 엔진에 호출된 함수가 올라가 실행되는 것을 지원한다.

* 함수 정의 키워드 : def 함수명

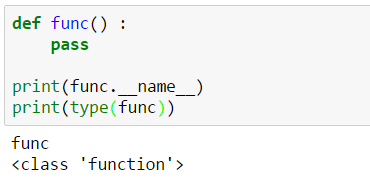
함수명은 함수의 이름으로도 사용되지만 함수가 작성된 모듈 내의 Namespace 관리되는 변수명으로 사용된다. 실제 함수명은 함수의 \_\_name\_\_속성에서 함수명을 문자열로 관리한다.

* 함수 매개변수(parameter) :

함수의 매개변수는 함수가 호출될 때 전달되는 인자(argment)들과 매칭되는 지역 변수(local variable)이다. 매개변수와 인자는 항상 동일한 개수가 매칭되어야 하지만 매개변수에 초기값을 부여할 경우 인자가 다 들어오지 않아도 처리된다.

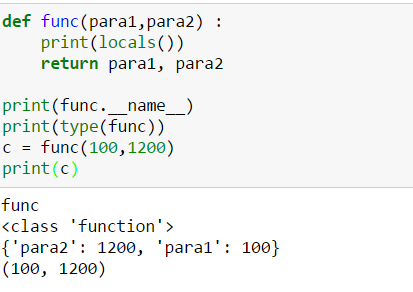
예제 2-28 : 함수 정의 할 때 아무 기능이 없이 정의도 가능

함수 func를 정의하고 매개변수도 하나도 없고 본문에 로직이 pass로 정의해서 하나도 처리를 하지 않지만 실제 함수를 정의하면 함수의 \_\_name\_\_가 함수의 자료형에 대한 값은 만들어진다.



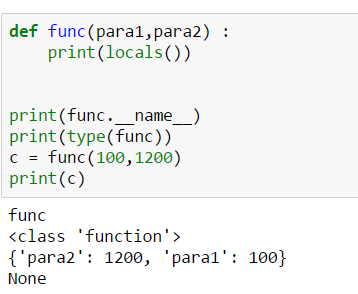
예제 2-29 : 반환값과 return문

함수에 매개변수를 2개를 넣고 locals 함수로 출력하면 함수의 호출할 때 인자가 매개변수에 들어가서 지역변수로 표시되는 것을 알 수 있다. 함수의 결과를 매개변수로 반환했으므로 결과는 튜플로 변수에 할당되는 것을 볼 수 있다.



예제 2-30 : return문이 없을 경우 처리

함수에 return문은 옵션이라서 필히 지정하는 것은 아니다. 반환되는 결과를 명기하지 않았지만 함수 처리 결과를 None으로 반환한다.



### 2.3.2. 클래스 정의문

클래스 정의문은 하나의 클래스 객체를 만드는 것으로써 이 정의된 클래스를 기준으로 인스턴스 객체를 만들 수 있다.

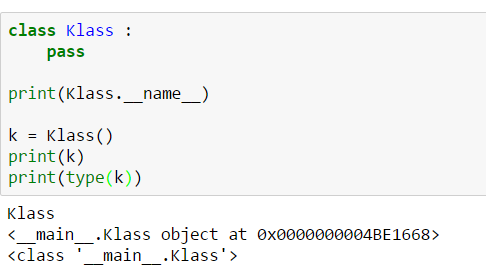
* 클래스(class) 정의문

사용자 정의 자료형인 클래스를 만들어서 다양한 기능을 처리할 수 있도록 정의하면 파이썬 내부에서는 클래스 객체를 만들어준다.

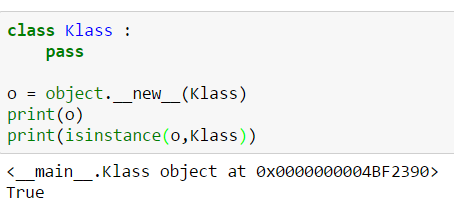
예제 2-31 : 아무것도 하지 않는 클래스 정의

클래스 정의문만 정의하고 내부 로직이 아무것도 없지만 클래스에 대한 이름인 \_\_name\_\_이 존재하고 이를 기반으로 인스턴스도 만들 수 있다.

이런 이유는 object를 상속을 받았기에 object 내의 \_\_init\_\_이 자동으로 호출되어 실제 인스턴스가 만들어지는 것이다.



아무것도 없는 Klass이지만 실제 인스턴스가 생성되는 이유는 실제 object.\_\_new\_\_(Klass)를 넣고 인스턴스를 만들면 이 인스턴스와 Klass간의 isinstance 함수를 통해 관계를 비교하면 True로 출력되는 것을 확인할 수 있다.



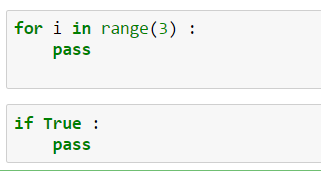
### 2.3.3. pass 문장

파이썬 구현할 때 실제 로직이 필요하지 않으면 pass문을 넣을 수 있다. 함수나 클래스 외에도 다양한 곳에 사용이 가능하다.

실제 문장의 완성을 위해 아무런 일을 하지 않는 표현이 필요하다. 파이썬에서는 pass를 하나의 문장처럼 사용하도록 정의했다.

예제 2-32 : pass 문 사용

제어문이나 순환문에서도 특별한 처리가 필요하지 않을 경우에 pass문을 사용할 수 있다. 실제 프로그램 작성할 때 임시로 처리하는 곳에도 pass문을 정의해서 사용이 가능하다.



## 2.4 모듈(module)/패키지(package) 처리

프로그램이 작성되면 여러 개의 모듈로 관리할 수 있고 이를 다양한 패키지로 묶어서 관리할 수 있다.

이번 장에서는 간단한 모듈 및 패키지 관리하는 방식을 이해해 보기로 하겠다.

### 2.4.1. import 구문

패키지 내에 있는 모듈이나 타 패키지에 있는 모듈을 직접 접근해서 사용하려면 그 모듈이 있는 위치를 명확히 알고 현재 작성하는 곳에 import 처리를 해야 한다.

* Import 구문 사용법
* 모듈을 올리기 : import 패키지.모듈명, import 모듈명
* 패키지나 모듈 내에 일부 요소만 올리기 : from 패키지명 import 모듈명, from 패지명.모듈명 import 함수/클래스명
* 이름이 충돌을 막거나 축약형으로 명칭을 사용하기 위해 as 다음에 별칭(alias)를 부여
* 모듈을 작성할 때 import 문은 항상 처음에 위치하도록 작성한다.
* 모듈을 import 할때는 항상 절대 경로를 사용하도록 한다.
* 상위와 하위의 모듈이 존재할 때 이를 import 할 경우에는 알파벳 순으로 작성한다.



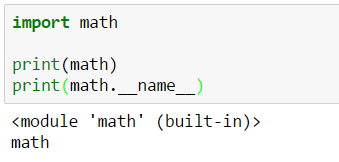
* 특정 모듈 전체 import

모듈 전체를 import 할 경우에는 import 구문만 사용한다.

내부 모듈이 변수, 함수, 속성 등은 객체접근연산자(.)을 표시하고 그 다음에 내부에 포함된 요소를 붙여서 사용한다.

예제 2-33 : 파이썬 제공 모듈을 import 하기

파이썬 내에 수학을 처리하는 함수 들을 제공하는 모듈인 math를 import하면 이 math라는 이름을 출력하면 모듈이라는 정보를 출력한다. 이 math에 대한 \_\_name\_\_을 확인하면 math라고 출력하는 것을 볼 수 있다.

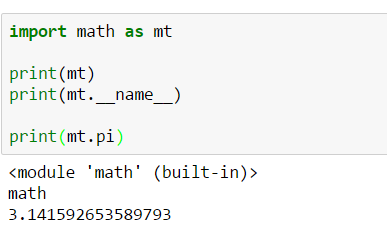


* 특정 모듈 전체 import하고 별칭 사용하기

모듈 이름이 길어지면 이를 줄여서 별칭으로 사용할 수 있다.

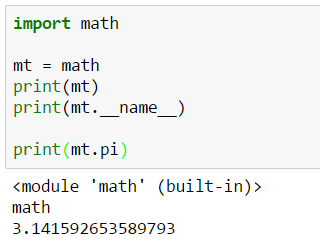
예제 2-34 : 내부 제공 모듈 import 할 때 별칭을 부여하기

수학을 함수를 제공하는 모듈에 별칭을 부여해서 pi 상수를 점연산으로 접근해서 출력할 수 있다. 별칭은 실제 math 모듈을 별도의 mt로 할당해서 처리하는 것과 동일한 결과를 나타낸다.



키워드 as를 사용하지 않으면 별도의 mt 변수를 정의하고 이곳에 import한 모듈 math를 직접 할당해서 처리해도 동일한 결과가 나오는 것을 볼 수 있다.

별칭이란 실제 별도의 변수를 지정해서 그 변수에 모듈의 변수를 재할당하는 것이라는 것을 확인할 수 있다.



* from/import 로 모듈 내에 특정 속성만 사용하기

특정 모듈이나 패키지에 전부를 import 하는 것이 아니라 특정 요소만 사용이 필요한 경우에 필요한 요소만 지정해서 import 처리한다.

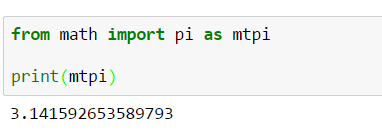
예제 2-35 : 모듈 내의 특정 속성만 원래 이름으로 사용하기

모듈을 import해서 처리하는 것이 아니라 모듈 내의 변수와 함수 이름을 직접 import해서 사용도 가능하다. 이렇게 해서 처리하는 단점은 기존에 동일한 이름이 있다면 이 이름과 충돌이 나서 다르게 처리할 수 있으므로 권장하지는 않는다.

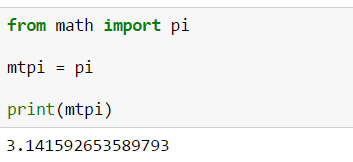


예제 2-36 : 모듈 내의 특정 속성에 별칭 부여하기

속성에 대한 이름이 충돌을 방지하기 위해 pi 변수에 대해 실제 별칭으로 mtpi로 이름을 할당해서 처리하도록 할 수도 있다.



별칭은 실제 별도의 변수를 지정해서 import한 것을 할당해서 사용하는 것과 동일한 처리라는 것을 알 수 가 있다.



### 2.4.2 모듈 생성 및 실행하기

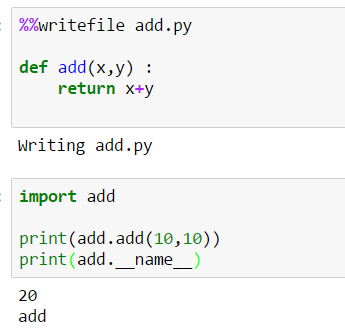
사용자 정의 모듈에 함수를 정의하고 실제 이를 import해서 처리하는 예시를 확인해 보겠다.

* 모듈을 생성하고 import 후에 메인 모듈에서 처리하기

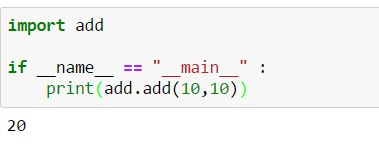
파이썬에는 main 함수가 제공하지 않는다 대신 module이 \_\_name\_\_ 관리 규칙이 현재 실행되는 모듈일 경우는 “\_\_main\_\_” 으로 관리하므로 이 기준으로 실행을 확인할 수 있다.

예제 2-37 : 함수를 정의해서 파이썬 파일로 저장

함수 add를 만들고 add 라는 이름으로 정의한 py 파일을 하나 만들었다. Import 해서 add 함수를 호출해서 사용하면 결과를 출력한다.



지금 작성하는 모듈과 먼저 작성한 모듈의 차이가 있다. 현재 작성하는 모듈은 내부에서 \_\_name\_\_이 \_\_main\_\_으로 바뀌어서 실제 제어문으로 체크해서 \_\_main\_\_ 이름에서 처리하면 직접 호출된 모듈만 처리할 수 있는 것을 알 수 있다.



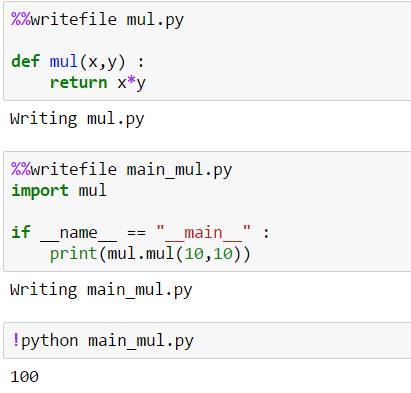
* 모듈을 생성하고 shell에서 실행하기

파이썬에서는 모듈 내에 함수나 클래스를 정의해서 관리한다. 실제 모듈들이 shell에서 실행되는 것을 확인해 보겠다.

예제 2-38 : 두개의 모듈을 생성하고 shell에서 실행하기

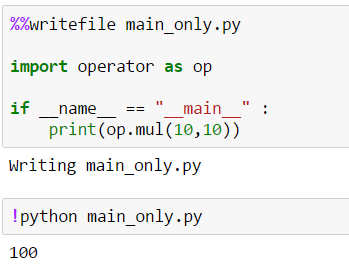
호출되는 관계를 확인하기 위해서 2개의 모듈을 작성한다. Main\_mul은 shell에서 호출해서 처리되는 main 모듈이고 mul모듈은 실제 main에서 호출되는 mul 함수를 관리하는 모듈이다.

실행 창에서 python과 메인모듈명을 가지고 호출하면 메인 모듈 내의 if \_\_name\_\_ == “\_\_main\_\_” : 내부의 로직이 처리되는 것을 확인할 수 있다.



예제 2-39 : 하나의 모듈을 만들고 shell에서 실행하기

하나의 모듈을 만들어서 내부에 함수를 가진 operator 모듈을 import하고 그 안의 mul함수를 모듈 내의 if \_\_name\_\_ == “\_\_main\_\_” : 내에서 호출되어 처리하도록 정의했다. 하나의 모듈이라도 이 모듈부터 호출되므로 실제 main 처리가 되므로 하나의 모듈 내에 모든 것이 다 있는 것을 확인할 수 있다.



### 2.4.3. 모듈 네임스페이스 확인하기

전역과 지역에 해당되는 Namespace는 모듈과 함수에 만들어지는 네임스페이스영역으로 모듈과 함수 내에 정의된 모든 이름을 관리한다.

파이썬에서 모든 Namespace는 딕셔너리(dict) 자료형으로 만들어지므로 항상 키와 값을 쌍으로 관리한다.

네임스페이스를 조회하는 함수는 dir이지만 모듈의 전역은 내장함수 globals로 조회가 가능하고 함수 내의 지역은 내장함수 locals 를 사용해서 조회가 가능한다. 조회할 결과는 딕셔너리로 제공하므로 검색을 할 때도 딕셔너리 조회하는 방식을 이용해서 처리가 가능하다.

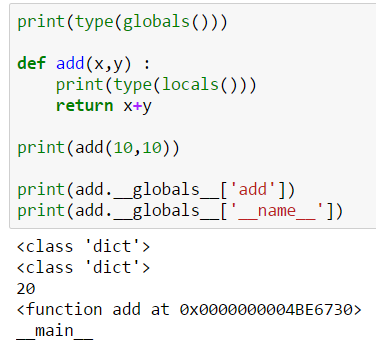
* 모듈과 함수 내의 네임스페이스 확인

함수와 모듈 내의 네임스페이스를 처리하기 위해서 어떻게 네임스페이스를 관리하는 지를 확인해 본다. 실제 함수에도 \_\_globals\_\_로 속성이 있어 모듈이 가지는 전역 네임스페이스와 연결되어 있는 것을 확인할 수 있다.

예제 2-40 : globals와 locals함수 처리

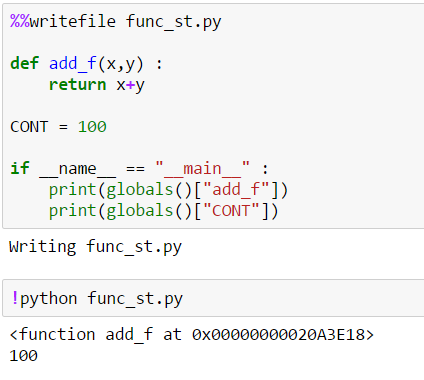
내장함수 globals를 모듈에서 실행하면 모듈 내의 모든 이름을 관리하는 딕셔너리 타입을 처리하고 내장함수 locals는 함수 내의 모든 이름을 관리하는 딕셔너리 타입을 처리한다.

함수 add 내에서 \_\_globals\_\_라는 속성을 가지고 있고 이 속성은 이 함수가 속한 모듈이 전역 네임스페이스와 동일한 정보를 가지고 있다.



예제 2-41 : 모듈에 변수와 함수를 정의하고 전역에서 조회

하나의 모듈 func\_st 내에 함수 하나와 변수 하나를 정의한다. 전역 네임스페이스를 관리하는 globals 함수를 실행하고 나온 딕셔너리 내에 add\_f 함수와 CONT 변수를 조회해서 실제 관리하는 지를 확인해 보면 값을 출력하는 것을 확인할 수 있다.



### 2.4.4 패키지(package) 처리

많은 모듈들이 존재하면 이것도 여러 개의 패키지로 묶어 관리할 수 있다. 모델들을 패키지로 묶고 다시 이 패키지를 패키지로 묶어 그룹으로 관리도 가능하다.

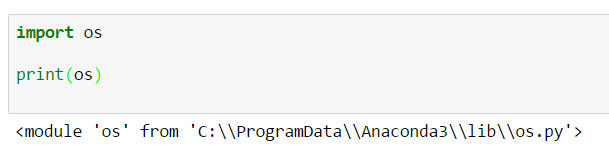
이런 패키지를 가지고 모듈에서 import 해서 사용할 수 있고 이를 이용해서 어떻게 패키지가 import해서 사용하는 지를 알아보겠다.

* testpk 패키지 생성하기

모듈을 관리하는 상위 testpk 패키지를 만든다. 파이썬 모듈이 관리하는 패키지에는 반드시 \_\_init\_\_.py 모듈을 만들어야 한다.

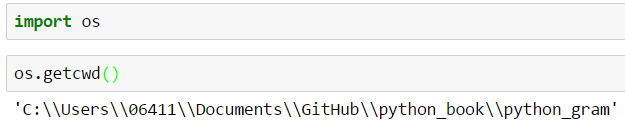
예제 2-42 : os 모듈 확인

파이썬에서 os 모듈은 다양한 operating system을 위한 wrapper 처리 용 모듈로써 리눅스와 윈도우 등의 os 명령어를 함수로 처리할 수 있도록 지원을 한다.



예제 2-43 : 현재 디렉토리를 확인

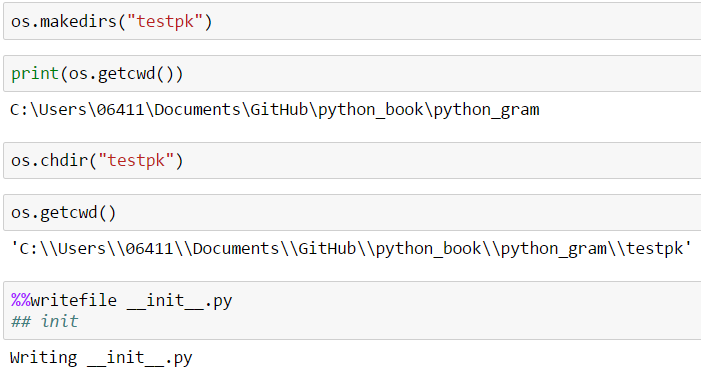
파있너에서 제공하는 모듈os를 import한 후에 현재 디렉토리를 확인한다



예제 2-44 : 패키지 만들기 : testpk

모듈 os 내의 함수 makedirs를 이용해서 현재 디렉토리에 testpk 를 만든다. 모듈 os 내의 chdir 함수를 이용해서 방금 만든 testpk 디렉토리(패키지)로 이동한다. 모듈 os내의 getcwd를 이용해서 현재 디렉토리를 확인하면 testpk 디렉토리에있는 것을 확인할 수 있다.

파이썬은 패키지(디렉토리)도 하나의 객체로 관리한다. 객체가 되려면 \_\_init\_\_.py를 반드시 지정해야한다. \_\_init\_\_.py 내부에는 아무 것도 없어도 패키지로 인식하는데는 이상이 없다.



* testpk 패키지 내의 하위 패키지 생성하기

모듈을 관리하는 상위 testpk 패키지를 만든다. 파이썬 모듈이 관리하는 패키지에는 반드시 \_\_init\_\_.py 모듈을 만들어야 한다.

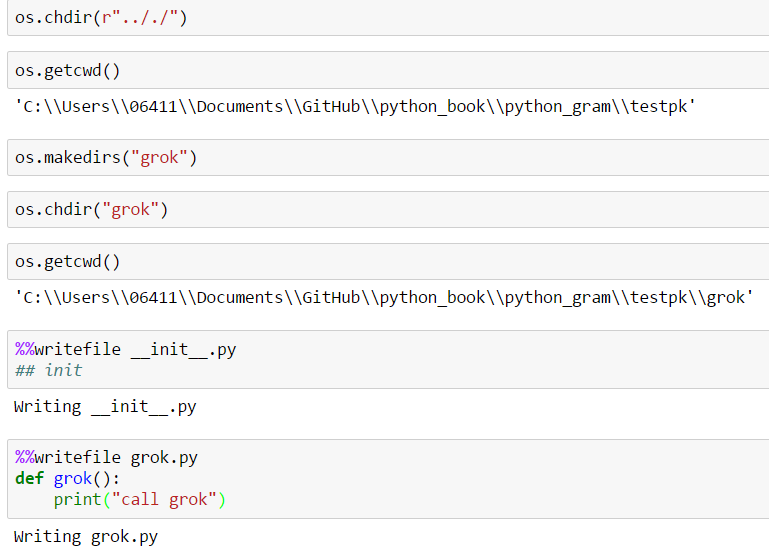
예제 2-45 : 하위 패키지 만들기 : spam

상위 패키지 testpk 내부에 spam 패키,spam 모듈, spam 함수로 정의한다.



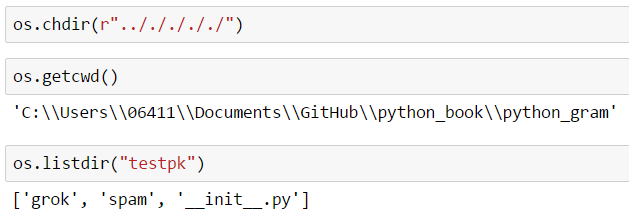
예제 2-46 : 하위 패키지 만들기 : grok

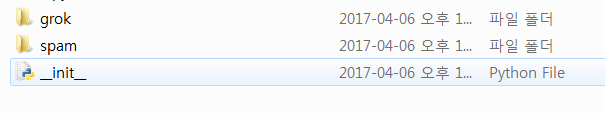
다시 상위 패키지인 testpk로 이동해서 grok 패키지와 grok 모듈과 grok 함수를 정의한다.



예제 2-47 : 상위 패키지 내부 정보 확인하기

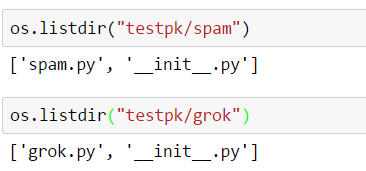
상위 패키지의 정보를 확인하면 두개의 패키지와 \_\_init\_\_ 모듈이 만들어져 있는 것을 확인할 수 있다.





예제 2-48 : 하위 패키지 확인하기

상위 패키지(testpk) 내부의 spam과 grok 패키지의 정보를 확인해서 내부에있는 모듈들을 확인한다.

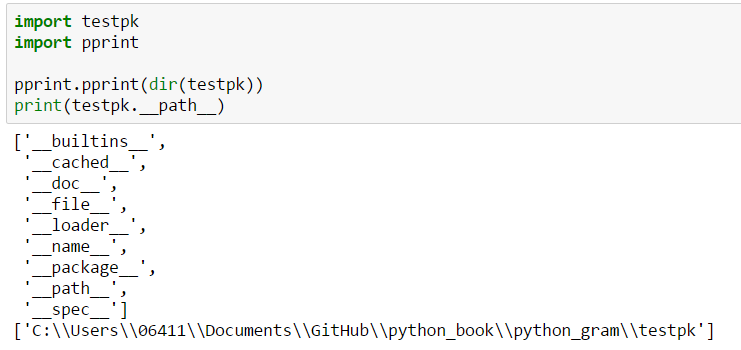


* testpk 패키지 내부 속성 확인

파이썬은 패키지도 객체로 인식하므로 패키지도 dir 함수로 조회하면 스페셜 메소드가 자동으로 만들어져 있다. 내부 모듈들에 대한 재로딩을 import로만 처리하는 것을 기준으로 설명한다.

예제 2-49 : package를 import 해서 내부 속성을 확인

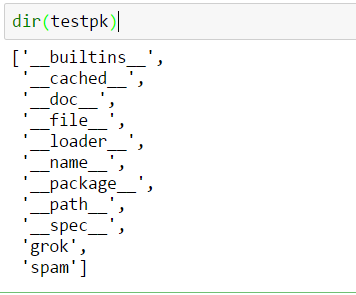
파이썬은 두 패키지를 전부 import 하고 실제 testpk 패키지의 내부 정보를 dir함수로 조회하면 실제 만들지 않은 속성들이 자동으로 생성되어 있는 것을 볼 수 있다.



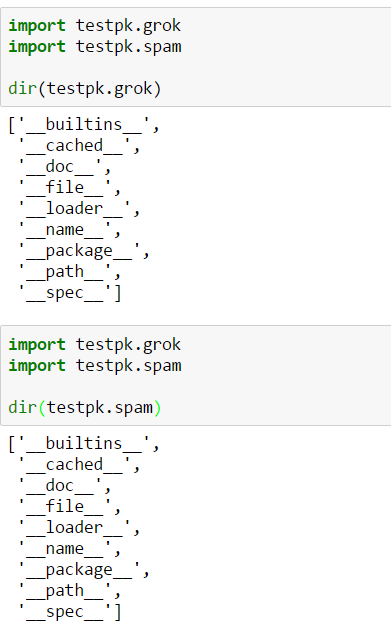
예제 2-50 : 내부 타입 체크

모듈 types는 파이썬 인터프리터에서 사용하는 자료형을 가지고 있다.이 모듈을 이용해서 패키지를 import 한 대상들과 동일한 타입으로 생겼는지 확인해 본다.

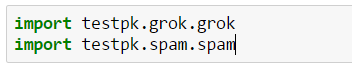
최상위 패키지에 대한 정보를 확인한다. 자동으로 컴파일이 안되면 import로 내부 패키지를 로딩해서 처리하면 된다.



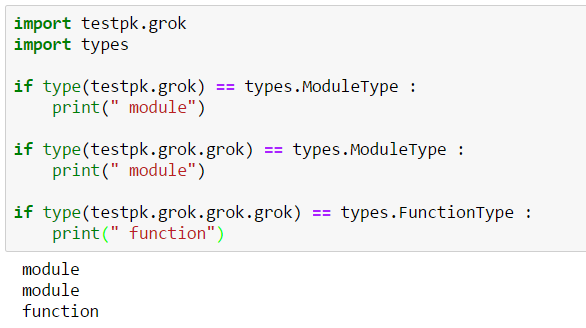
하위 패키지를 로딩해서 확인한다. 로딩이 제대로 안될 경우 로딩해서 처리하면 패키지까지 재로딩이 된다.



모듈들을 로딩해서 확인하기.



실제 grok 패키지를 다시 로딩하고 파이썬 내부의 타입을 점검하는 types 모듈을 import해서 세부적인 타입을 체크해서 확인해 본다. 패키지나 모듈은 실제 ModuleType으로 점검하고 실제 함수는 FunctionType으로 체크해서 확인해야 한다.

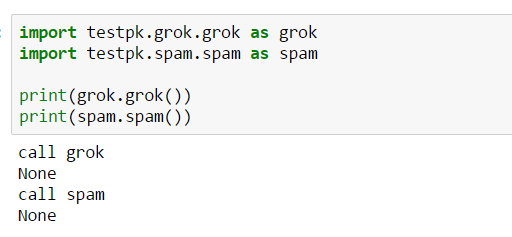


* 내부 모듈의 함수 실행

Testpk 내부 패키지 내의 패키지와 모듈을 import해서 내부 함수를 호출해서 실행을 시킨다.

예제 2-51 : import로 패키지 내부의 모듈 내의 함수를 호출하기

패키지 testpk내부에 서브 패키지 spam 밑에 spam 모듈 내에 spam 함수를 호출해서 처리한다.



## 2.5. 네임스페이스(Namespace) 관리 규칙

파이썬 개념에서 가장 중요한 네임스페이스 부분을 다시 한번 재설명을 하는 이유는 정확히 이해하기 때문이다.

네임스페이스(Namespace)가 만들어지면 실제 함수 등의 로직에서 어느 위치에 있는 네임스페이스를 검색해서 처리할 지에 대한 명확한 규칙인 Scope 규칙을 추가해서 알아본다.

이런 세부적인 사항은 함수 등의 세부 설명하는 장에서 자세히 설명하겠다.

### 2.5.1. 모듈 네임스페이스에 할당하기

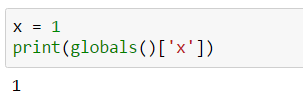
파이썬 모듈(.py)로 작성되면 내부에는 Namespace가 존재하며 이 내용을 조회할 수 있는 내장함수인 globals를 제공한다.

* Namespace에 변수, 함수, 클래스 할당하는 방법

모듈에서 변수, 함수, 클래스를 정의하면 파이썬 인터프리터는 자동으로 모듈 내의 Namespace에 변수로 정의된 것을 문자열로 전환해서 등록한다.

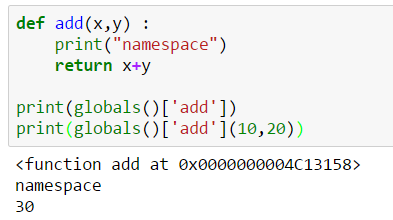
예제 2-52 : Namespace는 변수 할당하고 조회

변수 x 를 정의하고 1을 할당하였다. 하나의 모듈 내에 선언된 것을 확인하기 위해서는 함수 globals()를 실행해 보면 나오지만 다른 것들도 나오기에 goloabls 함수의 실행결과가 딕셔너리(dict) 타입이므로 [ ] 연산자 내에 변수 x를 문자열로 넣어 조회하면 변수 x에 할당된 값을 조회할 수 있다.



예제 2-53 : Namespace는 함수 할당하고 조회

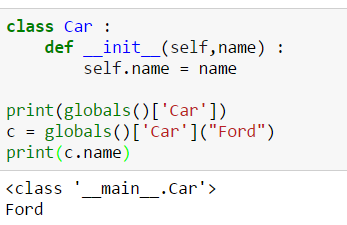
add함수를 정의했지만 모듈 내의 Namespace에는 변수명과 함수객체가 등록된다. 이를 globals 함수로 조회할 때는 변수와 동일한 방법으로 처리되고 함수가 저장된 객체의 레퍼런스를 출력하고 실제 이 레퍼런스를 실행하보면 함수가 실행되는 것을 알 수 있다.



예제 2-54 : Namespace는 클래스 할당하고 조회

Car 클래스를 정의를 해서 모듈 내의 Namespace에는 변수명과 클래스 객체가 등록된다. 이를 globals 함수로 조회할 때는 변수와 동일한 방법으로 처리되고 클래스에 대한 정보를 출력해 준다.

실제 globals에서 조회해서 인스턴스를 만들어서 이 인스턴스가 제대로 만들어져 있는지 확인을 해본다.



### 2.5.2. 전역 Namespace 처리 규칙

파이썬은 모듈별로 전역 네임스페이스를 관리한다. 모듈을 import도 해서 사용해도 전역 네임스페이스 관리 규칙은 변하지 않는다.

함수가 속한 모듈이 전역영역이 그 함수의 전역이 되는 것을 명확히 이해한다.

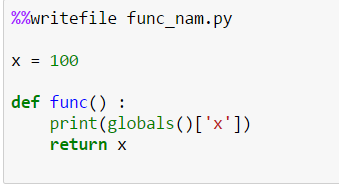
* 함수가 보는 전역 네임스페이스 확인

함수는 모듈 안에 작성될 때부터 실제 전역영역은 결정된다. 어떻게 자기의 전역을 검색해서 결과를 주는 지를 확인해 본다.

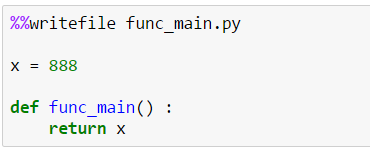
예제 2-55 : 함수 내의 지역 Namespace 확인

세 개의 모듈을 작성한다. 각 모듈별로 왜 네임스페이스가 만들어져 있는 지를 알아본다. 함수에 어떤 전달도 없을 경우 전역변수가 출력하는 것을 비교해 본다.

함수 func인 속한 모듈인 func\_nam에서 x라는 변수를 정의하고 결과로 x를 리턴한다.



함수 func\_main이 속한 모듈인 func\_main에서 x라는 변수를 정의하고 결과로 x를 리턴한다.

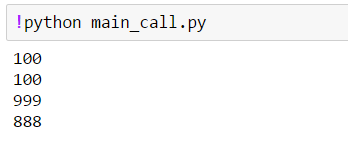


위의 2개의 모듈을 import해서 함수들을 호출해서 처리하를 한다.

두 개의 모듈을 import하고 호출한 함수에서 x 전역변수의 처리와 자기 모듈 내의 x 전역변수 출력값이 어떻게 되는지를 print 함수로 출력한다.



실제 실행하면 결과는 각 모듈의 전역 네임스페이스가 다르다는 것을 확인할 수 있다.



### 2.5.3. dir 함수를 통해 네임스페이스 출력하기

파이썬은 모든 것은 객체이다. 모든 객체에 대한 네임스페이스를 출력이 가능하다. 모듈일 경우는 모듈에 대한 네임스페이스를 출력한다.

예제 2-56 : 최상위 패키지에 대한 조회

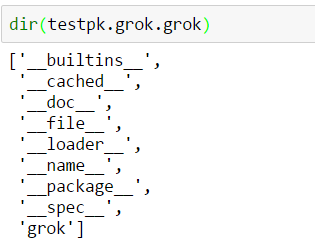
패키지에 대한 내부 정보를 조회할 수 있다. 언더스코어 표시가 없는 것은 내부에 만들어진 패키지를 의미한다.



하위의 패키지 grok 에 대한 조회 결과이다



하위의 패키지 grok 내의 모듈 grok에 대한 조회 결과이다



최하위에 있는 함수에 대한 내부 속성들을 조회한 결과이다.



### 2.5.4. 단순한 namespace 처리용 클래스

하나의 단순한 클래스를 이용해서 별도의 네임스페이스 처럼 사용할 수 있다. Types 모듈 내에 SimpleNamespace 클래스를 제공해서 필요할 경우 간단히 네임스페이스 처리용 객체를 만들어서 사용할 수 있다.

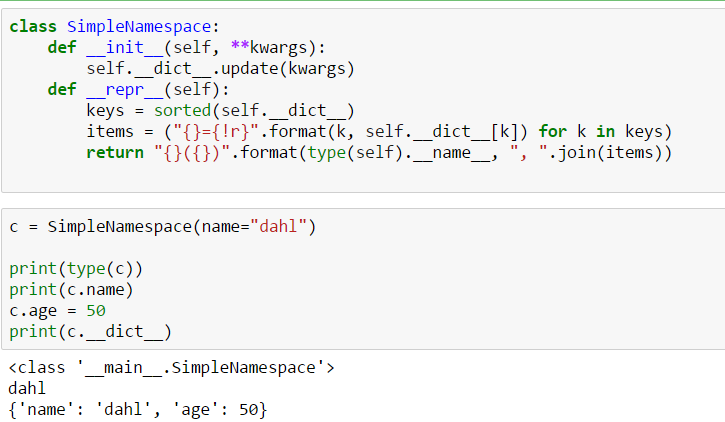
예제 2-57 : 간단한 네임스페이스 처리

SimpleNamespace 클래스는 단순하게 초기화 처리와 \_\_repr\_\_ 처리만을 제공하고 실제 \_\_dict\_\_ 내부에 속성들을 추가해서 다른 네임스페이스와 구별해서 사용할 수 있도록 만들었다.

일단 인스턴스를 만들어서 다양한 속성들을 추가하면서 사용할 수 있어 좋다.



SimpleNamespace 내부는 두개의 메소드만 구현되어 있는 것을 알 수 있다. 이런 클래스를 정의해서 네임스페이스 등으로 사용하면 다른 이름과이 충돌을 예방해서 사용할 수 있다.



## 2.6. 요약

파이썬 프로그램을 작성하기 위한 기본 문장들을 이해했다. 1장에서 정의된 값과 이름 등이 문장에서 어떻게 표현되는지를 추가적으로 이해하게되었다. 프로그램을 작성하면 패키지, 모듈, 함수나 클래스 등의 단위로 묶어서 표현할 수 있고 이 단위로 실제 프로그램 내의 이름 등을 관리하는 네임스페이스 등의 개념을 알아보았다

* 기본 문장 이해하기 : 할당, 인용, 주석 등의 기본 문장
* 프로그램 내의 문장 이해하기 : 제어문, 순환문, 컨텍스트, 예외처리 등의 문장
* 함수와 클래스 정의 : 함수와 클래스를 정의하는 문장
* 모듈과 패키지 처리 : 모듈과 패키지들의 관리 방법 및 import 해서 사용하는 법
* 함수와 모듈 등의 네임스페이스 : 지역 네임스페이스와 전역 네임스페이스 관리 기준
* 함수는 항상 자신의 속한 모듈의 전역네임스페이를 사용

**3. 자료형 : 숫자 원자형**

파이썬에 다양한 자료형이 있지만 제일 먼저 숫자에 대한 것부터 알아보는 것이좋다.

숫자형 자료형에는 정수(int), 실수(float), 복소수(complex)가 있고, 추가적으로 유리수(Rational), 큰 수 처리용(decimal) 등은 별도의 모듈로 제공한다.

숫자형 자료형의 특징은 원자형인 값만 가지고 있다. 절대로 변경할 수 없는 불변(Immutable)객체이고 인스턴스 객체를 생성할 때도 내장 타입은 리터럴과 생성자 2가지 방법을 유리수와 큰 수 처리용은 생성자만 제공한다.

또한 이런 자료형은 런타임에 사용자에 의한 속성이나 메소드를 추가할 수 없고 별도의 속성이나 메소드가 필요할 경우 사용자 클래스를 만들어서 사용해야 한다.

이번 장에서는 숫자 타입과 이를 지원하는 연산자를 이해해본다.

## 숫자 자료형의 특징

파이썬 내부 인터프리터 엔진에서 하나의 원소가 즉 하나의 인스턴스 객체로 처리된다. 동일한 숫자는 동일한 객체여야 하지만 특정한 경우에 한해서만 지원하므로 동일성을 점검할 때도 인스턴스의 레퍼런스로 체크하면 안되고 권고사항은 원자형도 값이 동등성만을 == 연산자로 사용해서 처리하는 것을 권고한다.

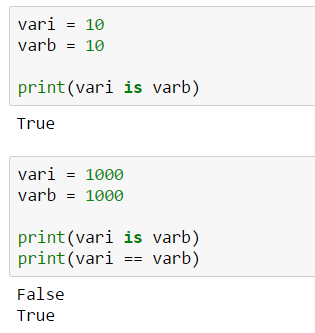
### 숫자 타입의 특징

파이썬 숫자 자료형도 레퍼런스에 처리 이슈가 존재해서 두 숫자의 레퍼런스 비교보다는 값으로 비교하도록 권고하고 있고, 숫자 자료형도 객체이므로 런타임시 속성을 추가가 가능해야하지만 규약상 이를 제거했다.

진법 처리가 필요한 경우 정수일 때만 처리가 가능하다.

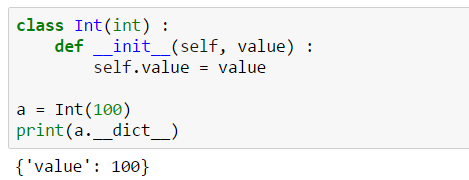
예제 3-1 : 원자형 정수의 레퍼런스 처리

파이썬 원칙상 두 정수의 레퍼런스는 동일해야 할 것 같지만 파이썬 내부 처리 상 인스턴스가 2개 만들어진 것을 볼 수 있다. 내장 클래스 객체로 생성한 인스턴스 객체를 비교할 때 주로 값으로 처리하는 것을 권고한다. 세부적으로 확인해 보면 255보다 작은 수일 경우는 동일한 레퍼런스로 처리되는 것도 알 수 있다..특정 숫자도 동일한 객체여야 하지만 파이썬에서는 별도의 객체로 만든다. 이런 이유로 동일한 처리를 하려면 == 연산을 이용해서 처리하는 것이 좋다.

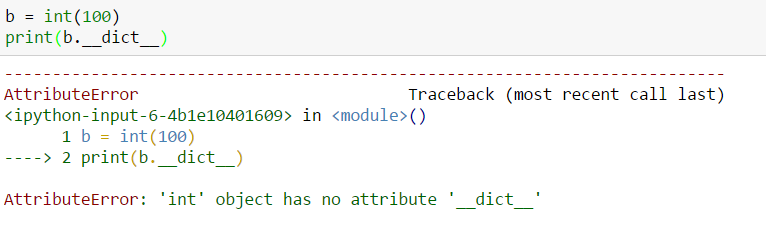


예제 3-2 : 런타임에 속성 추가 금지

사용자 클래스 Int를 int를 상속을 받아 정의한다. 하나의 인스턴스를 만들어서 \_\_dict\_\_ 내부를 조회하면 객체의 네임스페이스가 인스턴스에도 별도로 만들어져 있는 것을 확인할 수 있다.



내장 숫자 자료형 int 일 경우는 \_\_dict\_\_로 조회하면 이런 속성이 존재하지 않는다고 나온다. 이런 이유로 내부에 런타임 시에 속성이나 메소드를 추가할 수 없다.



### 3.1.2 숫자 타입의 형변환

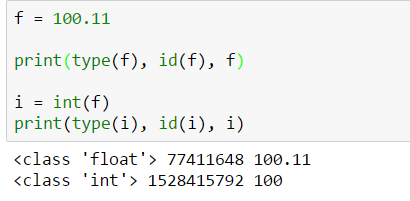
파이썬에서 형변환을 하는 별도의 함수는 존재하지 않는다. 형변환은 클래스 생성자로 새로운 인스턴스를 만드는 것이다.

생성자를 가지고 숫자 자료형에서 형변환하는 것을 알아보겠다.

예제 3-3 : 정수와 실수 형변환

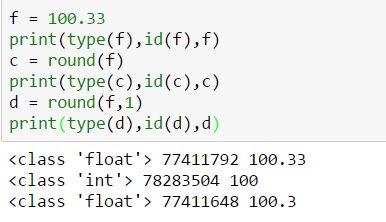
정수와 실수는 바로 변환이 가능하면 소수점 이하는 정수 변환이 절사가 발생하여 삭제된다.

실수를 정수로 형변환을 하면 전혀 다른 인스턴스가 생기는 것을 확인할 수 있다.



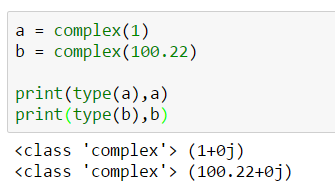
실수의 값을 정수로 버려도 실제 새로운 자료형이 만들어지므로 기존과 다른 인스턴스가 만들어져 있는 것을 확인할 수 있다. 실수의 소수점을 바꿔도 실제 값이 변경되므로 다른 인스턴스가 나오는 것을 확인할 수 있다.

숫자 자료형은 기본적으로 내부의 값이 변경이 된다는 것은 실제 새로운 인스턴스를 만들어서 제공하는 것을 확인할 수 있다.

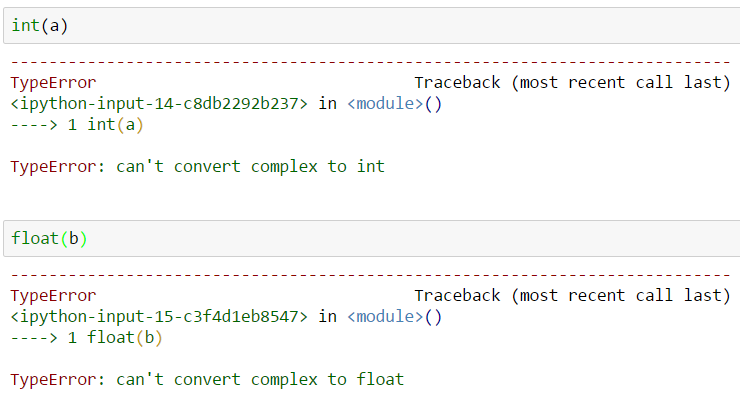


예제 3-4 : 복소수, 정수, 실수 형변환

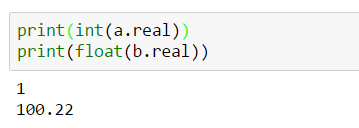
정수나 실수를 복소수로 전환하면 허수가 0으로 처리되는 것을 알 수가 있다. 숫자 체계에서 정수와 실수보다 복소수가 크기 때문에 이런 변환도 가능하다.



복소수를 정수나 실수로 형을 바꾸면 실제 복소수 부분이 처리가 되지 않는다.



위의 복소수를 만든 예제에서 실제 실수부만 가지고 형변환을 하면 허수부가 없으므로 형변환이 가능하다.



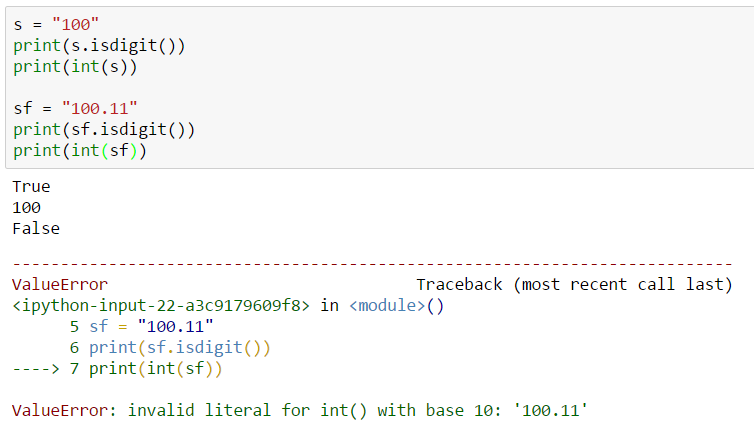
### 3.1.3 문자열 중에 숫자 형태의 숫자 형변환

문자열로 들어온 숫자에 대한 형변환이 필요할 때가 있다. 대부분의 파일이나 IO로 넘어온 것을 보면 문자열처리이므로 이를 가지고 숫자로 변환해서 처리하는 규칙을 이해할 필요가 있다.

예제 3-5 : 문자열을 정수로 형변환

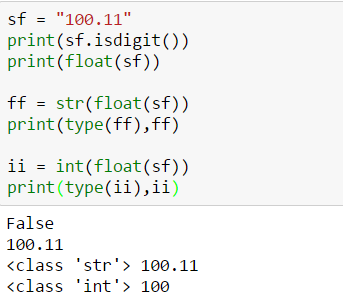
문자열 내의 isdigit 메소드를 이용해서 문자열 내의 모든 것이 숫자일 경우만 정수에서 처리가 된다.

문자열에 숫자가 아닌 점이 들어와서 실제 정수로 변환이 안된다. 정수는 실수로 들어온 경우를 제외하고 문자열 내의 실수의 점은 문자로 인식해서 처리할 수 없다고 표시한다.



예제 3-5 : 문자열(str)과 실수(float) 변환

문자열로 실수를 처리할 때는 float로 바로 전환을 하고 이를 다시 int로 전환하면 정수로도 변환이 가능하다. 실수로 전환한 것을 다시 문자열로 처리해도 소수점은 그대로 표시되는 것을 확인할 수 있다.

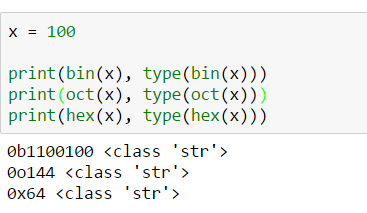


### 3.1.4. 2, 8, 16진법을 정수에서 변환 하기

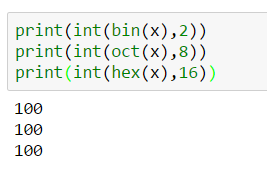
파이썬에서는 정수일 때만 2진수은 bin 함수를 이용해서 0b+ 숫자로 표시하고 8진수는 oct 함수를 이용해서 0o+ 숫자로 표시하며, 16진수는 hex 함수를 사용해서 0x+숫자로 표현한다.

예제 3-6 : 2/8/16진법으로 변환

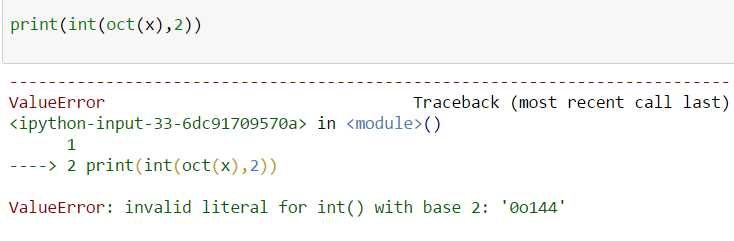
정수를 변수에 할당하고 bin, oct, hex 함수를 이용해서 값을 변환하면 문자열로 출력한다.



문자열로 변환된 2, 8, 16 진수를 int 생성자에 각 진법에 해당된 것을 넣고 처리하면 그에 맞는 정수로 변환을 한다.



8진수에 대해 2진수로 변환을 하라고하면 내부적으로 base가 상이해서 처리가 되지 않는다는 예외를 처리한다.



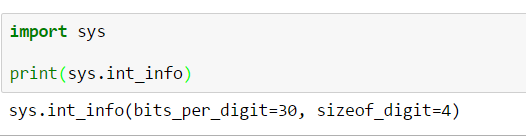
### 3.1.5. 숫자 자료형 최대 처리 수 확인

파이썬에서 제공되는 sys 모듈을 이용해서 숫자 자료형 정보를 확인해 본다. 각 os마다 조금씩 범위가 다를 수 있다.

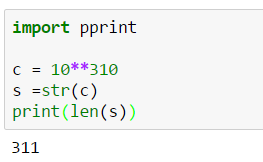
예제 3-7 : 숫자에 대한 범위 확인하기

파이썬에서 숫자에 대한 기본 처리 범위에 대해서는 sys 모듈을 import 해서 int\_info, float\_info 함수를 통해 확인할 수 있다.

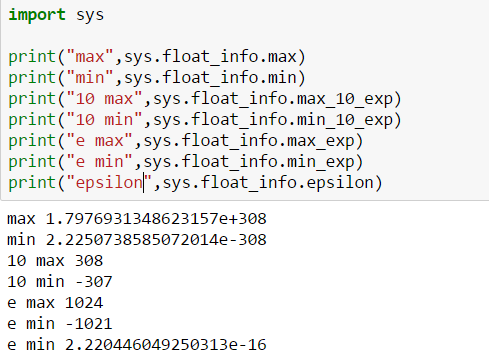
2\*\*bits\_per\_digit 까자의 숫를 생성할 수 있다는 것을 알 수 있다.



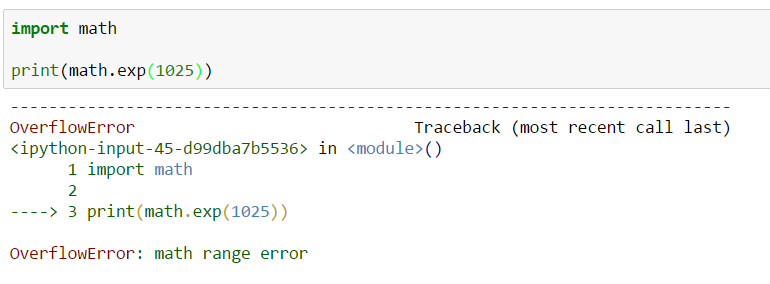
정수일 경우는 지수 범위가 310 제곱을 처리해도 실제 311자리의 숫자를 출력한다. 자리수가 너무 커서 이를 문자열로 바꿔서 총 숫자의 길이를 확인해보면 311자리를 표시하는 것을 확인할 수 있다.



실수 범위에 대한 범위도 확인할 수 있다.



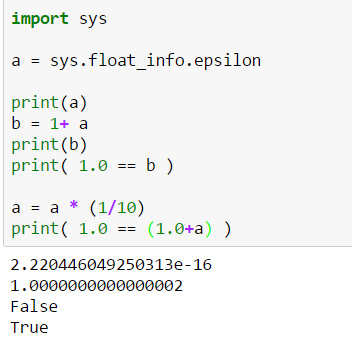
실수에 대한 e에 대한 지수 제곱에 대한 최대값 범위를 넘어서 처리하면 실제 오버플로우가 에러를 발생시킨다.



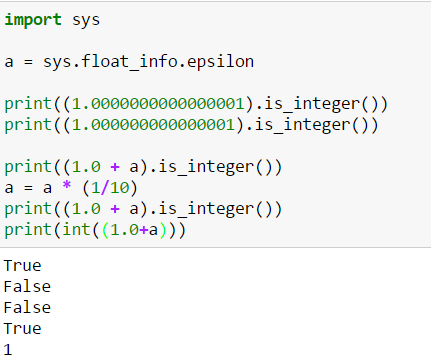
예제 3-8 : 엡실론(epsilon) 이해하기

엡실론은 함수의 극한에서 사용되는 작은 수를 의미하며 파이썬 실수 계산에서도 정밀도 처리를 위해 아주 작은 값으로 사용이 되어진다. float로 표현할 수 있는 1과 최소값이 1보다 큰 값 사이의 차이므로 실수 연산 후에 정밀도의 기본을 표현하는 값이 된다.

이 엡실론이 범위가 벗어난 숫자는 동등성 비교에서 True라는 것을 표시한다.



실수에서 정수로 변환이 가능한 경우는 소수점 이하 범위는 16자리까지이다. 이보다 적은 소수점 숫자는 정수로 변환이 불가하다. 엡실론 범위까지 처리가 된 실수는 정수로 변환이 불가능하는 것을 확인할 수 있다.



## 3.2. 내장 숫자 자료형 이해하기

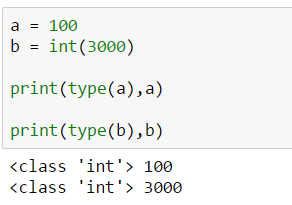
정수, 유리수(모듈), 실수, 복소수, decimal(모듈) 등의 자료형 즉 클래스와 내부 속성과 메소드에 대해 자세히 이해해보자.

### 3.2.1 정수 int class

파이썬에서 수학의 정수와 동일하며 파이썬 내의 부동소수점 숫자인 float를 제외한 모든 정수(integer)를 관리하는 자료형이다.

예제 3-9 : 리터럴과 생성자에 의한 생성

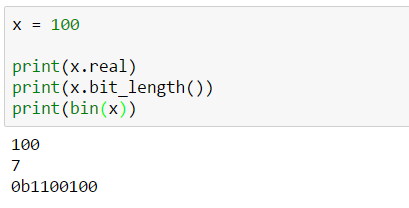
정수를 표기해서 리터럴로 인스턴스를 생성하거나 int 내의 매개변수로 숫자나 정수 문자열을 넣어서 정수를 생성한다.



예제 3-10 : 정수의 주요 속성 및 메소드

정수 내부의 속성 real로 실제 값을 조회할 수 있다. 정수에서만 이진 연산도 가능하므로 이진수로 변환한 경우 bit\_length 메소드로 이진수 크기를 알 수 있다.

정수를 bin 함수를 이용해서 변환하면 0b인 이진수 표현을 제외하면 총 7개 bit로 숫자가 만들이진 것을 볼 수 있다.



### 3.2.2. 불리언 타입 : bool

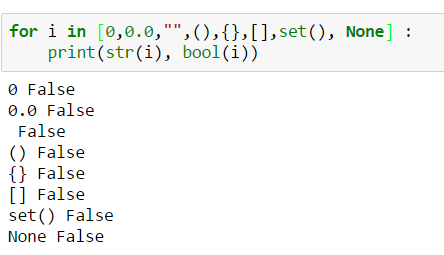
파이썬 bool 클래스는 int 클래스를 상속받아 처리되는 클래스이므로 int 클래스의 메소드가 다 실행되므로 사칙연산 등이 다 가능하다.클래스 bool 의 인스턴스는 True와 False 2개만 만들어져 있다.

* False로 추론되는 경우

파이썬에서는 bool 타입 체크는 다양한 조건을 수용해서 일단 데이터가 없거나 내부에 원소를 가지지 않았다는 것을 전부 False로 추론해서 처리한다.

예제 3-11 : False 추론

None, 숫자0 ,문자열에 없음(“”), 빈 리스트([ ]), 빈 튜플(()), 빈 dict({}, set( )) 등은 False이고 나머지 대부분은 True 본다.



* Bool 클래스 내부 정보 조회

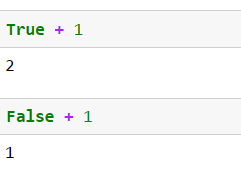
클래스 bool를 조회해 보면 int를 상속했고 특정 값을 bool 타입으로 변환한다는설명이 나온다.

예제 3-12 : bool 타입은 int를 상속을 받아 처리

실제 상속을 받았는 지를 \_\_bases\_\_로 확인하면 int 클래스가 있는 것을 볼수 가있고 issubclass로 확인해 보면 결과도 True로 표시한다.



내장 타입은 bool은 int 클래스를 상속을 받아 만들어진 클래스이므로 수치계산이 가능하지만 True는 1이고 False는 0으로만 처리되는 점도 알고 있어야 한다.

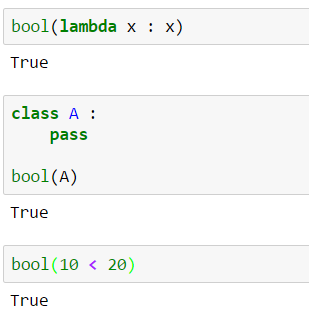


* 다른 인스턴스에 대한 변환처리

클래스 Bool로 생성할 때 리터럴, 표현식, 조건식이 다 가능하면 최종평가된 결과를 기준으로 True와 False로 결과값을 반환한다. .

예제 3-13 : 함수와 논리식을 변환

Bool 생성자에 함수, 클래스, 논리식 등을 넣어도 평가해서 논리값을 젝옹한다.



### 3.2.3 실수 float

파이썬에서 부동소수점 숫자인 float를 관리하는 자료형이다.

* help로 내부 정보 조회

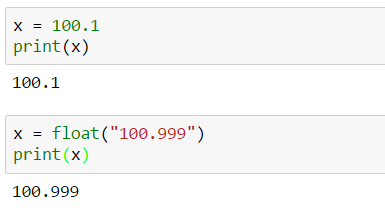
help함수, print 함수, type 클래스로 인스턴스 객체(1.1)에 대한 내부 정보를 조회하면 float 클래스 정보가 조회된다.

* 실수 생성방법

직접 리터럴로 생성하거나 클래스 float를 이용해서 인스턴스를 생성하면 된다.

예제 3-14 : 실수 리터럴과 생성자 호출

실수 값을 소수점 이하로 표시한 리터럴이나 생성자의 인수로 넣어 처리가 가능하다.

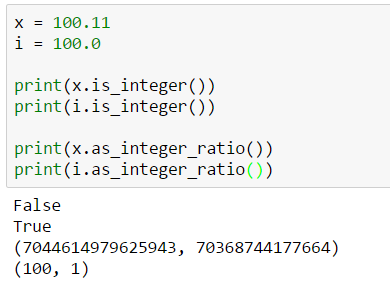


* 실수 속성과 메소드

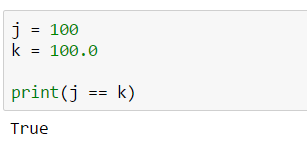
속성으로는 real, 메소드는 정수 값으로 실수에 대한 비율을 표시하는 as\_integer\_ratio와 정수(int) 전환 가능여부를 체크하는 is\_integer 가 있다.

예제 3-15 : 메소드 처리

As\_integer\_ratio 메소드는 가장 적절한 수를 표현하는 비율을 분자와 분모로 표시하고 is\_integer 메소드는 주어진 수들이 integer로 전환되는지 여부를 알려준다.



동일한 값인 1과 1.0에 대해 어떻게 판단하는지를 확인해 본다. 정수 int를 단순히 float화 한 경우를 비교하면 True로 인식한다.



### 3.2.4 복소수 complex class

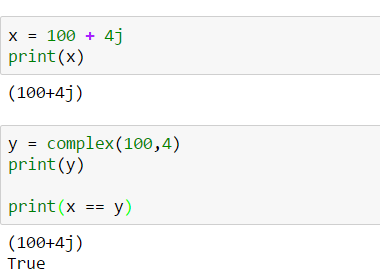
파이썬에서 복소수 숫자인 complex를 관리하는 자료형이다. 복소수의 허수 표기법이 i에서 j로 표시된다.

* 생성방법, 속성과 메소드

복소수도 다른 숫자 자료형과 동일한 방식으로 생성 및 메소드 처리를 한다. 차이점은 허수부가 추가로 존재한다.

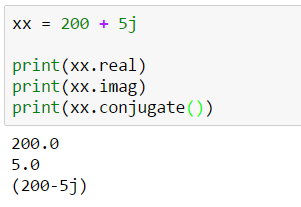
예제 3-16 : 복소수 리터럴과 생성자로 생성

리터럴과 클래스로 인스턴스를 생성하는 두 가지 방법이 다 가능하다.



예제 3-17 : 복소수 속성과 메소드

속성으로는 real(실수부), imag(허수부)가 있고 메소드 conjugate을 실행하면 현재 만들어진 복소수의 허수부의 부호가 반대인 켤레복소수를 표시한다.



### 3.2.5. 유리수 Fraction class

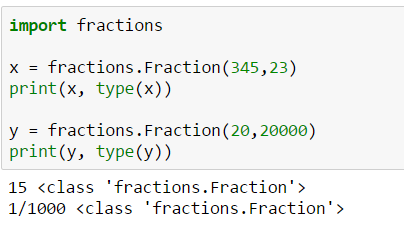
유리수(Rational)은 분수이고 즉, 분모 (denominator )와 분자(numerator)로 표시한다. 파이썬에서는 이 유리수를 분수인 fraction 모듈로 처리할 수 있도록 Fraction 클래스를 만들어서 지원한다. 유리수는 정수와 계산될 때에만 분수를 유지하므로 계산할 때 주의해야 한다.

* 유리수 생성방법

분모 (denominator )와 분자(numerator**)**에 대한 값을 정수로 넣어 유리수를 생성한다.

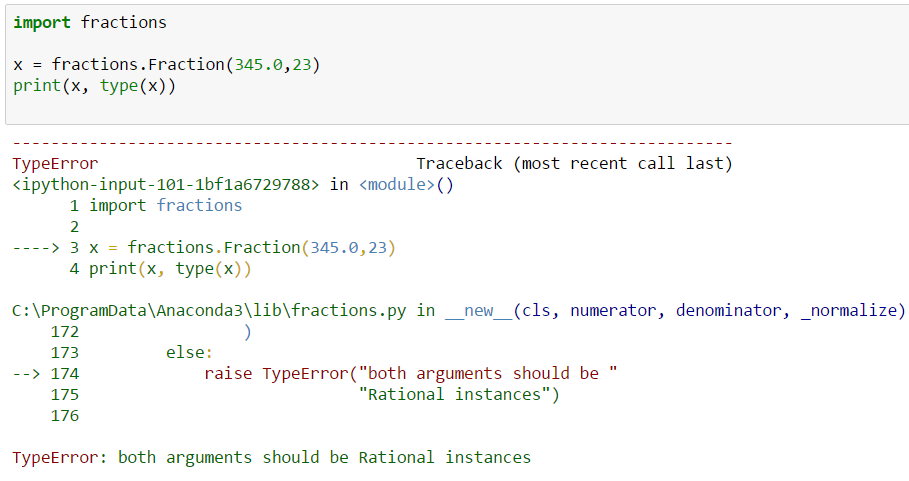
예제 3-18 : 유리수 생성자에 의한 생성

Fractions 모듈이 Fraction 클래스에 인자로 분모와 분자의 값을 주고 생성해야 한다.



예제 3-19 : 유리수 생성이 안 되는 경우

유리수는 분자와 분모가 정수일 경우만 처리가 되므로 실수가 오면 생성이 되지 않는다.



### 3.2.6. 큰 수 decimal class

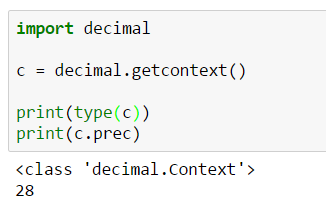
숫자를 계산시 정밀한 수치를 계산이 필요한 경우 decimal 모듈을 이용해서 처리해야 한다. Decimal로 생성은 정밀도에 대해 조정이 가능해서 실수로 계산하는 것보다 더 큰 정밀도도 계산이 가능하고 정밀도에 대한 조정도 가능하다.

* Decimal 모듈

실수보다 더 정밀한 계산을 위해서는 decimal 모듈을 이용해서 계산하는 것이 더 좋다.

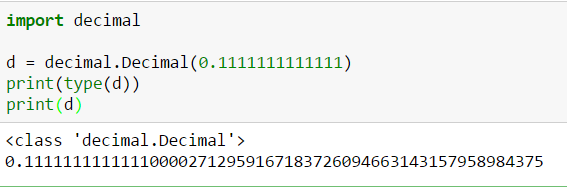
예제 3-20 :  Decimal 모듈의 환경 이해하기

Decimal 모듈을 사용하기 위해서는 context를 알아야 하고 현재 context는 getcontext 함수로 조회하면 Context 인스턴스를 가져온다. 그 내부의 속성인 prec는 소수점 이하의 정밀도이므로 정밀도에 대한 처리 기준을 알 수 있다.



예제 3-21 : 소수점 처리 확인

실제 입력한 것보다 더 정밀도가 높기 때문에 긴 소수점 숫자를 출력한다.

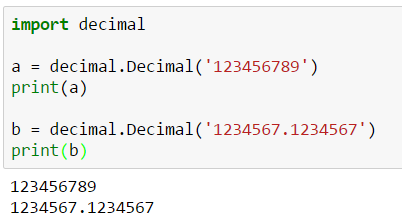


* 생성방법 및 계산 처리

Decimal 클래스의 인자로 정수와 실수 또는 문자열로 정수와 실수를 정의해서 생성하지만 다른 클래스와 계산 시 오류가 발생할 수 있으므로 자료형을 맞춰 계산을 수행한다.

예제 3-22 : Decimal 클래스 생성자

문자열로 정수나 실수를 배정하거나 정수나 실수를 배정해서 생성이 가능하다. 숫자문자열일 경우 표시되는 숫자까지만 생성한다.



예제 3-23 :  실수와 계산 오류 발생

파이썬 float 와 decimal 모듈의 인스턴스는 서로 다른 자료형이므로 계산 시 예외가 발생한다. 동일한 타입을 맞춰 계산해야 한다.



실수와 Decimal을 상호 자료형을 맞춰서 연산을 하면 예외없이 계산이 된다.



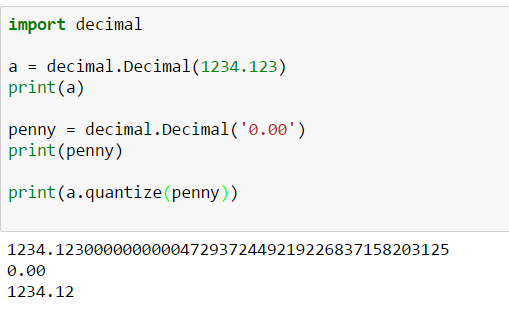
* 주요 메소드

특정 단위로 계산이 필요할 경우 명확한 단위를 별도의 인스턴스로 만들고 quantize 메소드로 단위를 정밀도를 명확히 할 수 있도록 조정한다.

예제 3-24 :  화폐 단위 등 특정 단위가 계산이 필요한 경우

Penny 등의 화폐단위를 추가적으로 계산이 필요한 경우는 별도의 단위를 만들어서 이를 quantize메소드에 넣어 단위를 조정할 수 있다.

단위의 값을 문자로 생성해야 소수점 이하의 값도 명확히 유지해서 처리되는 것을 알 수 있다.



## 3.3. 연산자(Operator)

수학을 기반으로 연산자를 만들었으므로 수학에서 처리되는 연산자의 원칙에 따라 파이썬 내에서도 연산방식이 동일하게 처리된다. 연산 시 연산자 우선순위에 따라 처리되므로 이 기준을 명확히 이해하고 처리해야 정확한 값이 계산된다.

연산자 우선순위를 정확히 맞추기 위해서는 제인 우선순위가 높은 괄호 연산을 사용해서 묶어주면 된다.

파이썬 내의 operator 모듈에는 이런 연산자를 함수로 만들어서 제공한다. 다른 함수에서 이 함수를 매개변수로 전달을 받아서 실행을 하면 함수형 프로그래밍 기법을 처리할 수 있다.

함수와 연산자가 동일하게 실행되는 지를 확인해 보기로 하겠다.

### 3.3.1. 연산자 표기법

단항 연산자(unary), 이항 연산자(binary) 등으로 구분하고 연산자 사이에는 피연산자(operand)가 들어간다.

* 피연산자와 연산자의 종류
* 단항 연산자(unary operation) : 단항 연산자는 피연산자 앞에 연산자를 나타나고 오른쪽에서 왼쪽으로 연결된다.
* 이항연산자(binary operation) : 이항 연산자는 피연산자와 피연산자 사이에 연산자를 나타나고 왼쪽에서 오른쪽으로 처리해서 두 대상으로부터 하나의 대상을 얻는 연산을 말한다
* 피연산자(operand) : 연산자의 연산의 대상이 되는 것을 말한다.

### 3.3.2. 연산자 우선순위

표현식 내의 연산자의 우선순위는 아래의 표처럼 처리 된다. 연산자 우선순위 처리시 오류가 발생할 수 있으므로 그룹 연산자인 괄호 ( )로 표시해서 처리해야 한다.

|  |  |  |  |
| --- | --- | --- | --- |
| 순위 | 구분 | **Operator** | **Description** |
| 0 | 그룹 | ( ) | Parentheses (grouping) |
| 1 | 함수 | *f*(args...) | 함수 호출 |
| 2 | 참조 | *x*[index:index] | 부분구조 읽기(슬라이싱) |
| 3 | 참조 | *x*[index] | 원소 참조 |
| 4 | 참조 | *x.attribute* | 속성 참조 |
| 5 | 산술 | \*\* | 제곱 구하기 |
| 6 | 비트 | ~*x* | Bitwise not |
| 7 | 부호 | +*x*, -*x* | 양수, 음수 |
| 8 | 산술 | \*, /, % | 곱하기, 나누기, 나머지구하기 |
| 9 | 산술 | +, - | Addition, subtraction |
| 10 | 비트 | <<, >> | 쉬프트 연산 |
| 11 | 비트 | & | 비트 and |
| 12 | 비트 | ^ | 비트XOR |
| 13 | 비트 | | | 비트 OR |
| 14 | 비트 | in, not in, is, is not, <, <=,  >,  >=,<>, !=, == | Comparisons, membership, identity |
| 15 | 논리 | not *x* | NOT 논리연산 |
| 16 | 논리 | and | AND 논리연산 |
| 17 | 논리 | or | OR 논리연산 |
| 18 | 함수 | lambda | 람다 표현식 |

### 3.3.3. 산술연산(arithmetic operation)

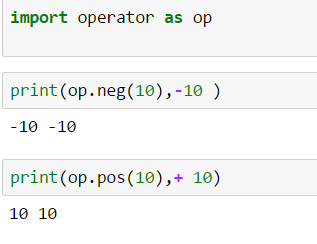
산술 연산자는 수학의 사칙연산 등 수치에 대한 계산을 처리하도록 한다.

* 산술연산 하기

단항연산, 이항연산 등 다양한 수학적인 단순한 산술연산자를 통해 처리하는 방법을 알아본다.

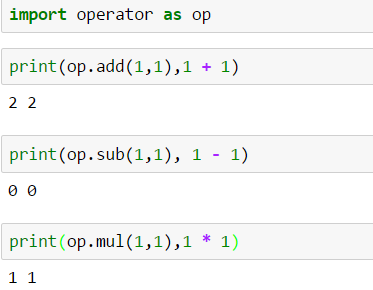
예제 3-25 : 부호 연산 : +, -

부호 표시를 하는 +,-는 단항 연산자(unary operation)로 숫자에 대한 부호를 표시한다.



예제 3-26 : 산술연산(arithmetic operation) : +, -, \*

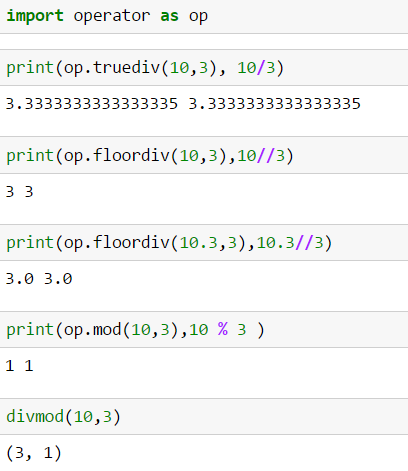
이항 연산자(binary operation)인 덧셈, 뺄셈, 곱셈은 숫자 타입의 값을 계산해서 연산자 특징의 맞춰 계산된 결과를 처리한다.



예제 3-27 : 산술연산(arithmetic operation) : /, //, %

이항 연산자(binary operation)인 truedevide 나누기(/), floordivide 나누기(//)와 나머지(%)가 있다. 나누기 연산에는 실수(float) 즉 소수점까지 계산하는 /, 정수(int)나 실수(float)일 경우는 소수점을 무시해서 나누기를 계산하는 //와 나머지 값을 계산하는 % 연산자들로 구성하면 계산된 값을 반환한다.

함수 divmod는 // 나누기와 % 나머지 연산을 한번에 처리되는 함수이다.



### 3.3.4. 비교 연산(comparison operation)

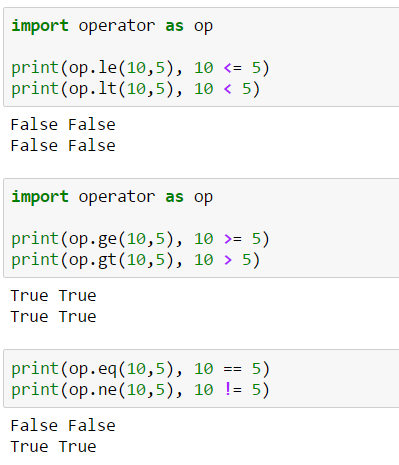
비교연산자도 이항 연산자(binary)이다. 비교 연산자는 두 수식을 비교하는 연산자로서 비교 연산한 결과를 True, False로 표현한다.

>(큼), <(작음), >=(크거나 같음), <= (작거나 같음), !=(같지 않음), ==(같음) 등 비교 연산은 조건식을 평가하는 if 문이나 while문에 사용되면 비교결과에 대한 값으로 제어문과 순환문에 대한 처리를 제한한다.

파이썬에서는 비교문은 문자 표현으로는 지원하지 않고 연산자만 지원한다.

예제 3-28 : 비교 연산(comparison operation)

Operator 모듈 내의 함수와 비교해서 비교연산이 제대로 실행되어 결과를 표시하는 지를 확인해 본다. 최종 결과는 True/False 값으로 반환한다.



### 3.3.5. 비트 연산(bits wise operation)

파이썬의 비트 연산자는 숫자를 2 진법으로 계산하는 연산자이다. 비트 연산자를 사용할 때는, 숫자를 0과 1로 구성된 문자열로 취급한다는 말이다.

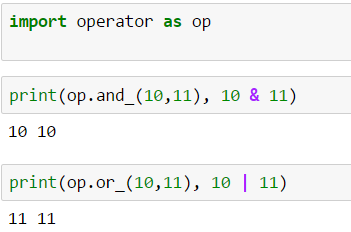
숫자 타입에서 비트연산자은 내장타입 int 타입일 경우에만 처리된다.

* 비트 처리 : and(&), or( | )

논리곱 연산자로(&/and )서 비트가 두 항에 모두 나타나는 경우 비트 결과를 표시, 논리합 연산자(|/or )로서 비트가 두 항 중 어느 곳에 나타나는 경우 비트 결과를 표시한다.

예제 3-29 : 논리곱, 논리합

정수에 비트연산을 처리하면 그 값이 이진수 처리와 동일하게 처리된다.

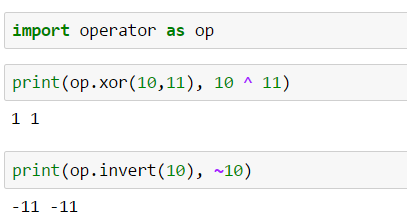


* 비트 처리 : ^, ~

배타적 논리합 연산자(^)로서 어느 한 쪽의 항에만 비트가 존재할 경우 비트 결과를 표시, 단항 연산자로써 invert(~) 로서 비트를 뒤집어서 각 비트에 대하여 정확히 반대를 반환하며 실제 처리결과는 숫자 1을 추가하고 부호를 바꾸면 된다.

예제 3-30 : 배타적 논리합, invert

정수에 비트 연산을 처리하면 이진수 처리하는 방식과 동일한 결과를 얻을 수 있다.

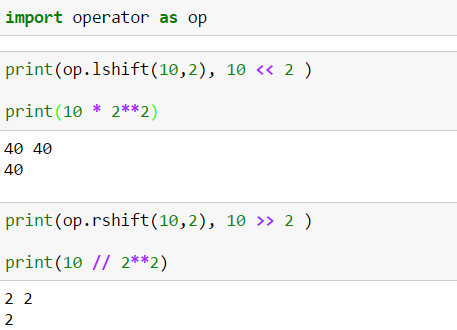


* 비트 이동 연산 : 시프트 연산

비트 연산 중에 비트의 자리를 이동해서 계산하는 방식으로 왼쪽으로 이동하면 곱하기 연산과 같고 오른쪽으로 이동하면 나누기 연산과 동일한 처리가 되어진다. .

예제 3-31 : shift 연산

정수에 대해 비트 연산을 사용하면 2를 가지고 곱하기와 나누기 한 값으로 처리된 결과를 확인할 수 있다.



### 3.3.6. 논리 연산(logica operation) 및 단축연산(short cut operation)

논리 연산자(Logical operation)는 Logical operator와 Boolean value를 함께 사용하여 Boolean value를 반환(return)한다.

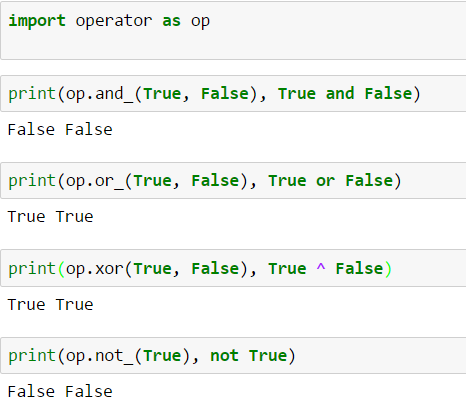
단축연산(short cut operation)은 표현식을 평가할 경우 둘다 논리적인 평가를 하지 않고 논리적인 판단에 해당되면 결과를 반환해서 처리한다.

* 논리 연산처리

논리합(and) x and y 평가에 있어서, x가 거짓으로 판명되면 x를 반환하고, 그렇지 않은 경우에는 y를 평가하여 결과값을 반환, 논리곱(or) x or y 평가에 있어서, x가 참으로 판명되면 x를 반환하고, 그렇지 않은 경우에는 y를 평가하여 결과값을 반환, 논리부정(not) not x 평가에 있어서, x가 거짓이라는 것은 x의 반대를 참값을 반환한다.

예제 3-32 : 논리 연산

bool 표현이 있을 경우 전부를 판단해서 처리한다.



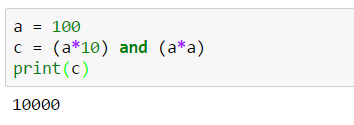
* 축약 연산 처리

Bool 표현이 아닌 산식을 계산하는 표현식에서는 축약 연산이 자동된다.

단, and와 or 연산인 경우만 축약연산이 처리된다.

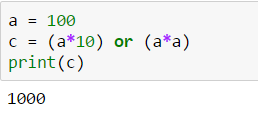
예제 3-33 : and 축약연산

and 연산은 첫번째가 True 일 경우는 and 우측 편을 반환하고 첫 번째가 False이면 and의 좌측 편을 반환한다.



예제 3-34 : Or 축약 연산

첫번째가 True 일 경우는 Or 좌측 편을 반환처리하고 첫번째가 False이면 or의 우측 편을 반환한다.



## 3.4. 요약

파이썬 데이터 모델 중에 숫자형 자료형과 유리수와 큰 수를 처리하는 모듈 등도 알아봤다.

숫자형 데이터 들의 특징은 실제 자료형으로 만들어진 인스턴스 객체에 연산을 실행하면 다른 인스턴스 객체가 만들어지는 원자형 구성임을 알 수 있다.

또한 주요 연산자를 구현한 operator 모듈과의 비교를 하면서 연산자가 어떻게 함수로 구현되었는 지를 이해해 보겠다.

* 산술연산자
* 비교연산자
* 비트연산자
* 논리연산자
* 축약형 논리연산자

**4. 자료형 : Sequence 형**

앞장에서 파이썬 숫자 자료형에 대해서 알아봤다. 하나의 값만을 가진 숫자 자료형은 주로 원자(atom)타입이지만 이제 여러 원자를 가지는 분자 타입(Molecule)들을 알아보기로 하겠다.

파이썬에서 여러 원자로 구성된 자료형을 시퀀스(Sequence) 라고 부르고 세부 자료형으로는 문자열, 리스트, 튜플, 바이트, 바이트어레이 등이 있다. Sequence 자료형의 특징은 동일한 타입의 원소를 가질 수도 있고 다양한 자료형을 원소로 가질 수 있는 하나의 컬렉션을 구성해서 각 원소들 위치(index)로 참조할 수 있다.

## 4.1. 파이썬 Sequence자료형의 특징

Sequence 자료형은 연속되는 데이터 구조를 처리하는 기본 방식을 제공하므로 프로그램 언어 처리시 가장 중요한 자료형이다. Sequence 타입에는 문자열(str), 바이트(bytes, bytearray), 리스트(list), tuple 등이 있다. 특히 파이썬은 동적이나 정적으로 Sequence를 처리하는 방식을 많이 제공하므로 이 자료형에 대한 특징을 잘 이해해야 할 것이다.

### 4.1.1. 런타임에 속성 추가 여부

내장 자료형은 실제 런타임에 내부 속성은 조회가 가능하나 추가는 할 수 없다. 스페셜 속성(\_\_dict\_\_ )을 제공하지 않는다.

스페셜 속성(\_\_dict\_\_)은 클래스 객체나 인스턴스 객체의 내부 속성을 관리하는 속성이지만 내장 자료형에는 제공하지 않으므로 실제 제공되어진 속성과 메소드만 사용해야 한다.

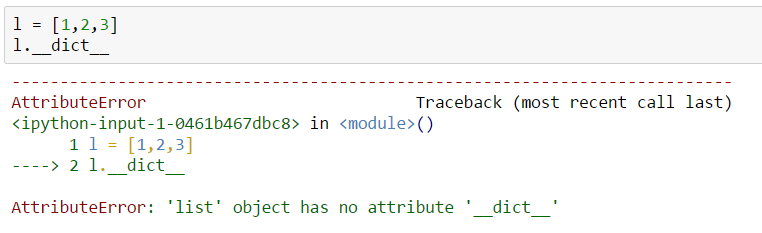
혹시 다른 기능과 속성을 추가하고 싶을 경우에는 사용자 클래스를 정의해서 정의해서 필요한 기능을 확장해야 한다.

* 파이썬 객체 Namespace(\_\_dict\_\_) 조회

내장 자료형들은 런타임에 인스턴스 객체에 속성을 추가할 수 없도록 내부에 \_\_dict\_\_ 속성이 없다. 인스턴스에서 \_\_dict\_\_속성을 조회하면 존재하지 않는 속성이므로 예외가 발생하므로 내장 자료형은 항상 dir 함수를 이용해서 접근을 해야 한다.

예제 4-1 : Sequence 내장 타입의 인스턴스 Namespce 미존재

리스트도 3장에서 설명한 숫자 내장 타입처럼 내부 속성인 \_\_dict\_\_을 지원하지 않는다. 이 말은 런타임에 실제 속성을 추가할 수 없다는 것을 알 수 있다.



### 4.1.1. 변경가능 여부 : Mutable & Immutable

변경 가능(Mutable)과 변경 불가능(immutable)에 대한 기본 개념은 변경 가능하다는 말은 객체 내부의 원소들을 추가, 삭제, 변경 등을 처리할 수 있다는 것을 말하는 것이지만 객체 자신을 변경할 수 있다는 말은 아니다.

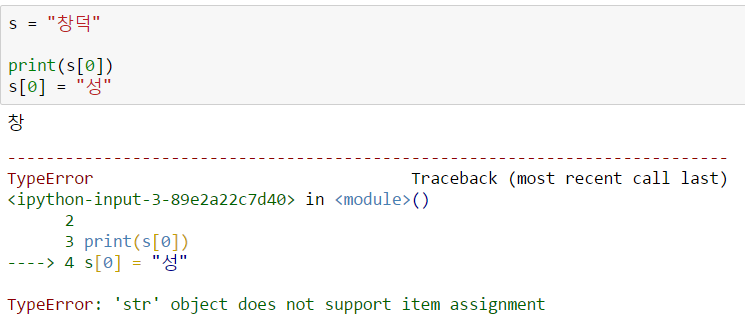
변경가능과 변경불가능에 대한 처리를 세부적으로 알아보기로 하자.

* 변경불가 자료형( Immutable )

변경이 불가하다는 뜻은 내부 원소의 값을 변경할 수 없다는 것이다. 문자열, 튜플, 바이트 자료형은 한번 생성되면 변경할 수 없는 구조가 되어지나 튜플일 경우 원소가 리스트 등의 자료형이 사용될 경우 변경이 가능해지는 단점도 가지고 있다.

예제 4-2 : 문자열은 변경불가

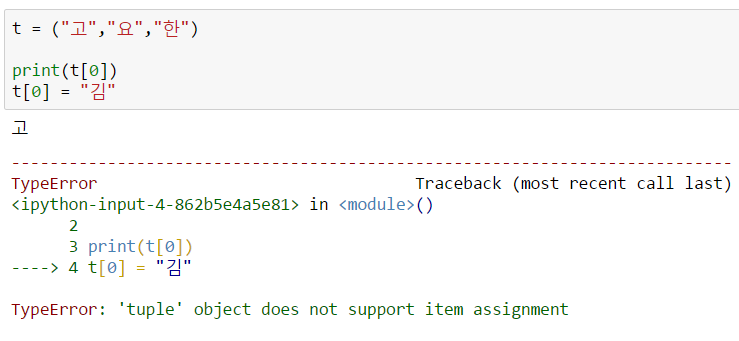
문자열을 리터럴로 정의하고 첫번재 원소의 값을 변경할 경우 변경이 불가능하다는 것을 알 수 있다.



.

예제 4-3 : 튜플(tuple) 변경불가

튜플을 리터럴로 정의하고 첫 번째 원소의 값을 변경하려고 하면 문자열과 동일하게 원소가 변경이 되지 않는다.



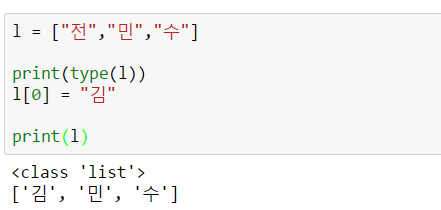
* 변경가능 자료형(Mutable )

Sequence 자료형에서 변경가능한 것은 리스트와 바이트어레이가 있다.

이 자료형은 기존에 만들어진 원소들을 변경, 삭제 및 추가 등을 자유롭게 할 수가 있다.

예제 4-4 : 리스트(list) 원소 변경

리스트를 생성해서 리스트 첫번째 위치를 변경해서 출력하면 첫번째 위치 원소가 변경된 것을 확인할 수 있다



### 4.1.3. 컬렉션(collections) 여부

컬렉션이란 다양한 원소를 가진 데이터 구조를 말한다. Sequence 자료형은 여러 원소들로 구성되므로 기본 컬렉션 이다.

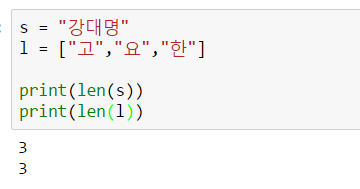
컬렉션 여부는 원소개수,포함관계, 반복가능형 여부를 확인할 수 있는지를 확인할 수 있으면 된다.

* 자료형(data type) 내의 원소 개수 확인 : len 함수

파이썬 Sequence 자료형 내부에 여러 원소를 가지므로 그 데이터에 대한 원소의 개수를 len함수로 확인할 수 있다.

예제 4-5 : 원소의 개수 확인

한글로 작성된 문자열의 길이를 확인하는 것이다. 파이썬 3버전 부터는 모든 문자열이 유니코드 문자로 처리되므로 문자열 길이는 바이트 단위가 아닌 문자 단어의 길이로 나타낸다. 리스트도 내부에 있는 원소의 개수를 길이로 나타낸다.



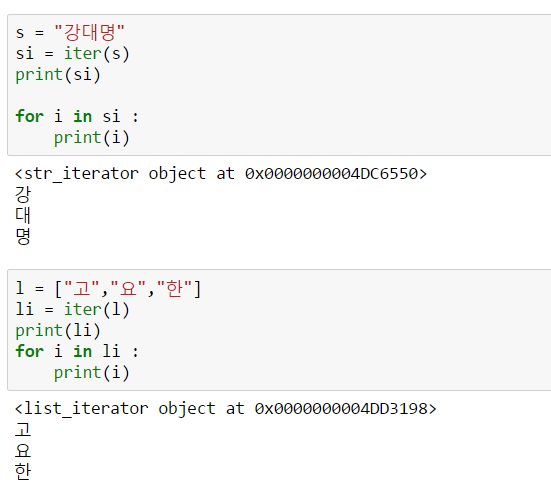
* 반복형을 반복자로 변형 처리: iter 함수

Sequence 자료형들은 내부에 원소들이 없거나 연속적으로 들어있어 반복해서 원소를 읽을 수 있어 반복형(Iterable)로 처리가 가능하다.

반복자 생성하는 iter함수로 호출하면 내부에 \_\_iter\_\_ 스페셜 메소드나 \_\_getitem\_\_이 존재해야 한다. 상세한 내용은 스페셜 메소드를 설명하는 장에서 하겠다.

예제 4-6 : 반복형을 반복자로 변환

문자열, 튜플, 리스트 등 반복형 자료형을 반복자로 변형하는 함수 iter를 이용해서 변환을 했다. 각 타입에 맞는 반복자(iterator )형이 만들어지는 것을 확인할 수 있다.

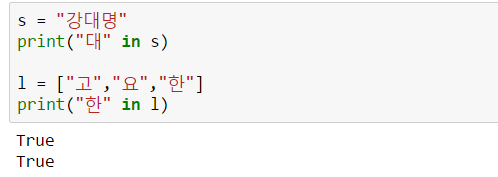


* Sequence자료형 내에 원소 포함 확인 : in 연산자

Sequence 자료형 내의 원소와 일치 여부를 확인이 가능한지를 in 연산자를 통해 처리한다

예제 4-7 : 원소들에 대한 포함여부 확인하기

Sequence 타입으로 생성된 모든 자료형 내의 원소의 포함여부를확인할 수 있는 키워드이다. 포함이 되면 True, 포함이 안되면 False로 표시한다.



### 4.1.4. sequence 타입 내의 메소드 처리 기준

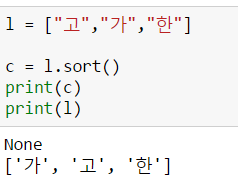
변경 가능(Mutable) 한 리스트, bytearray과 변경이 불가한(Immutable) 문자열, tuple, bytes 타입 내의 메소드 처리 결과가 다른 것은 원소 변경이 가능한 경우는 in-place 즉 내부 원소를 변경한다. 변경불가능한 경우는 항상 새로운 인스턴스 객체를 만들어서 결과값으로 반환처리한다.

* 내장 메소드 처리 기준

메소드 대부분은 자기 객체 내를 변경하므로 메소드 처리 결과값을 None으로 처리한다.

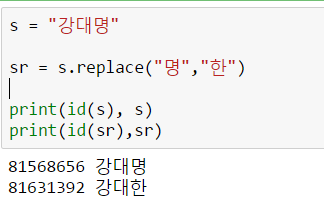
예제 4-8 : 변경 가능한 자료형의 메소드는 내부를 갱신

리스트를 생성하고 내부에 있는 sort 메소드를 이용해서 정렬한 경우이다. 실제 다른 인스턴스를 만들지 않으므로 처리결과가 None으로 표시된다.



예제 4-9 : 변경 불가능한 경우는 별도의 객체를 만들어서 반환 처리

문자열을 replace 메소드로 처리할 때 문자열은 원소의 값을 변경할 수 없으므로 실제 새로운 객체를 만들어서 대체한다. 실제 문자열 객체들을 확인해 보면 별도의 레퍼런스를 가지고 있다.



### 4.1.5. interning 처리

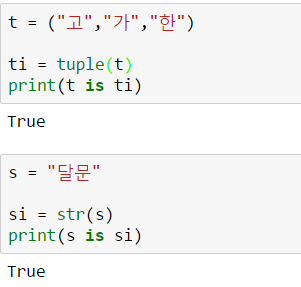
인터닝이랑 기존에 생성되어 있는 객체를 동일하게 사용하겠다는 뜻이며 Sequence 타입에서 어떻게 발생하는 지를 확인해 보겠다.

* 튜플이나 문자열처럼 변경 불가능할 경우

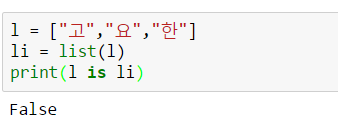
변경이 불가능한 자료형에서는 만들어진 객체를 다시 생성해도 기존 객체의 레퍼런스를 동일하게 반환해서 처리한다.

예제 4-10 : 변경불가능한 자료형의 interning 처리

튜플과 문자열은 기존에 만들어진 것을 다시 생성해도 내부적으로 동일한 것을 전달하는 interning 처리를 한다.



변경가능한 리스트는 기존 만들어진 것을 가지고 다시 생성하면 별도의 다른 리스트 인스턴스를 생성해서 준다. 즉 하나의 사본을 다시 만들어준다.



## 4.2. 문자열 자료형(string data type)

파이썬 3 버전은 문자열이 기본 유니코드(Unicode)로 처리 되며 문자열은 한번 생기면 그 안의 내용을 변경할 수 없다. 단, 새로운 문자열이 만들어져 다시 변수에 재할당되도록 처리할 수는 있다.

### 4.2.1. 문자열 생성 방법

문자열은 직접 리터럴을 이용하거나 str 클래스를 이용해서 생성이 가능하다. Bytes로 정의된 것을 decode 처리해서 문자열로 전환도 가능하다.

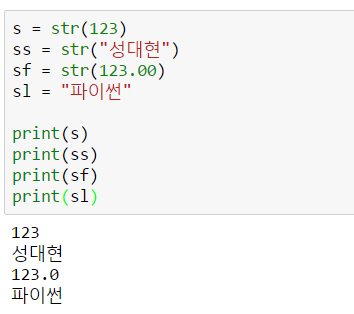
문자열은 기본 str 클래스의 인스턴스이다.

* 리터럴과 생성자로 문자열 생성

문자열 리터럴를 만들면 파이썬은 자동으로 str 클래스의 인스턴스를 만든 후에 변수에 레퍼런스를 할당한다.

예제 4-11 : 문자열을 리터럴과 생성자로 생성

문자열 str 클래스를 이용해서 직접 문자열을 생성할 수 있고, 문자열을 리터럴 값으로 즉시 생성도 가능하다.



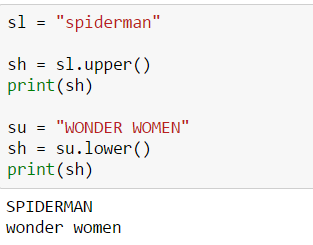
### 4.2.2. 문자열 주요 메소드

문자열은 생성되면 원소의 값을 변경할 수 없으므로 대부분 메소드는 새로운 문자열을 생성해서 결과를 반환한다.

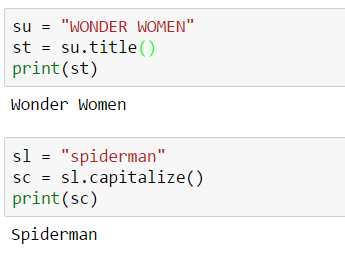
* 문자열 내의 문자 조정하기

문자열 내의 문자를 대소문자나 위치 등을 조정하는 메소드들을 확인해 본다.

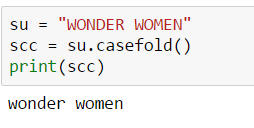
예제 4-12 : 대소문자 처리

문자열을 전체를 대문자는 upper와 소문자 lower로 전환해서 처리한다. 

문자열을 가지고 제목처럼 표시하기 위해서는 title 메소드를 사용하고 특정 단어의 첫 번째 문자만 대문자로 전환할 때는 capitalize를 사용한다.



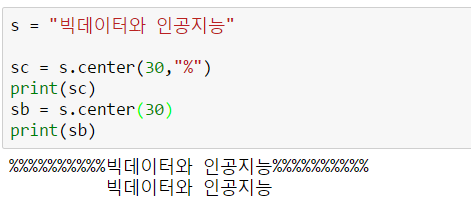
casefold 처리는 특수한 희랍문자 등의 경우를 제외하고는 소문자 처리하는 것을 볼 수 있다.



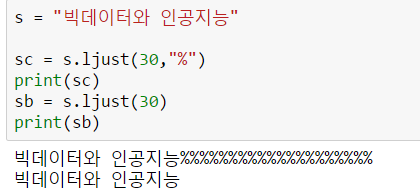
예제 4-13 : 문자열 위치 조정해서 꾸미기

문자열에 대한 크기를 정하고 가운데(center), 왼쪽(ljust), 오른쪽(rjust)로 정렬하고 나머지 빈 공간에 들어갈 값을 정해서 처리한다.

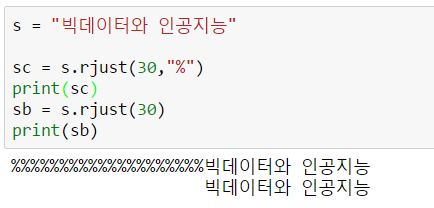
문자열을 중앙에 정리하고 빈공칸이나 %로 나머지를 채웠다..



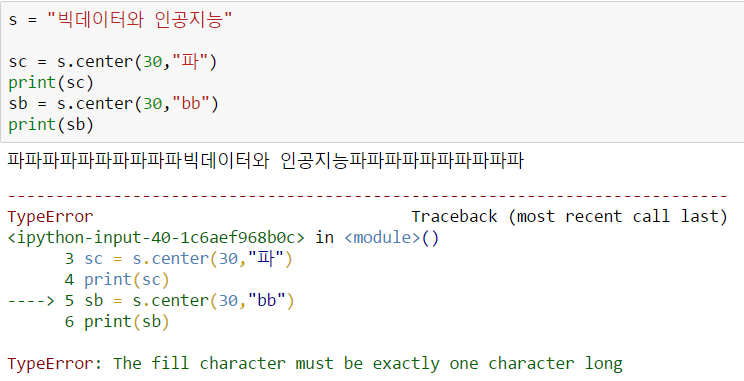
문자열을 좌측으로 정렬하고 빈공칸이나 %로 나머지를 채웠다.



문자열을 우측으로 정렬하고 빈공칸이나 %로 나머지를 채웠다.



문자열 위치 조정해서 처리할 때 주의할 점은 빈공칸을 채우는 것은 한 문자로 처리해야 한다. 한글로 한글자를 입력할 경우는 빈공칸을 채우지만 영어 두 글자를 넣을 경우 예외가 발생하는 것을 확인할 수 있다.

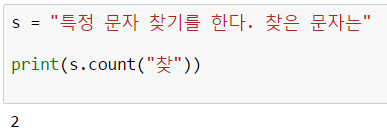


* 문자열 찾기 및 패턴 매칭하기

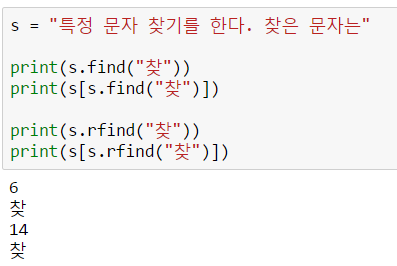
문자열에 대한 검색(finde, index) 및 개수(count)를 확인하는 메소드를 제공한다. 처음(startswith)과 마지막(endswith)로 부분문자열을 체크해서 있으면 True, 없으면 False를 제공하는 메소드들이다.

예제 4-14 : 특정 문자 찾기

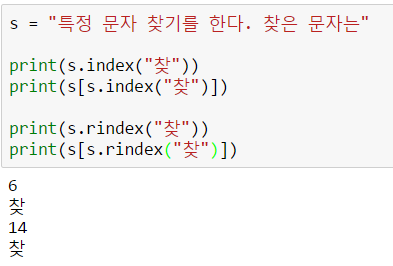
동일한 문자가 몇 개 있는지를 확인하기 위해 count 메소드를 이용해서 특정문자를 확인한다.



문자열의 fine와 rfind 메소드를 이용해서 특정문자 하나를 찾아서 위치 정보를 반환한다. 실제 검색을 통해서 문자도 출력해 본다.

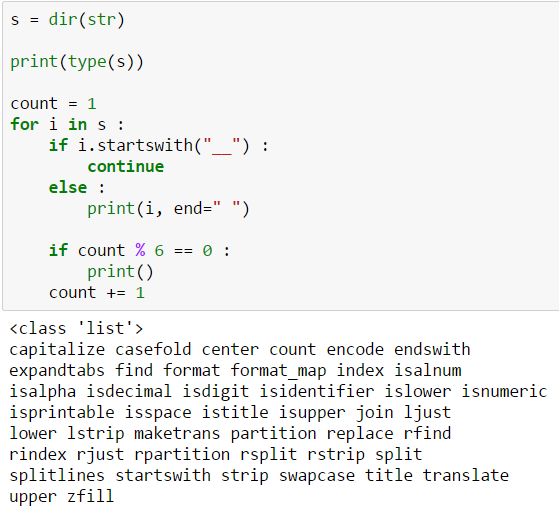


문자열 index와 rindex를 가지고 특정문자를 찾아서 위치정보를 가져왔다. Find 메소드와 동일한 처리를 하는 것을 확인할 수 있다.

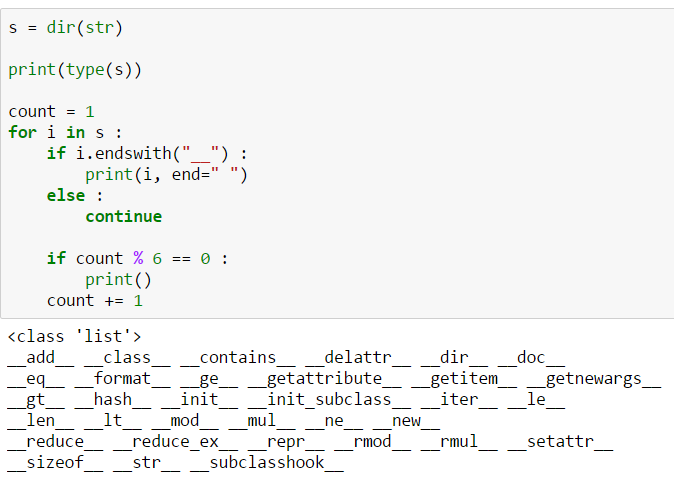


예제 4-15 : 문자열 패턴 매칭하기

문자열의 시작문자를 체크해서 패턴이 맞는 지를 확인하는 startswith 메소드를 가지고 문자열 내부에 가진 스페셜 속성과 메소드를 제외한 속성과 메소드를 출력해 본다.



문자열의 마지막에 있는 패턴을 확인해서 처리하는 endswith 메소드를 이용해서 문자열 클래스 내의 스페셜 속성과 메소드만을 출력해 본다.



* 문자열을 분리 및 문자열 결합

메소드 split은 문자열을 분리하려면 문자열 분리자를 인자로 넣어서 분리자별로 리스트의 원소로 구성되도록 한다. 메소드 join은 분리자를 문자열로 정하고 리스트를 인자로 받아 리스트의 원소를 분리자별로 나눠 문자열로 변경한다.

예제 4-16 : 빈 문자열로 분리하고 결합

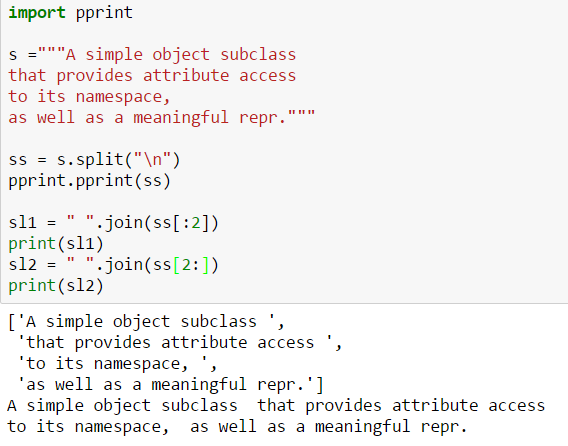
문자열 내의 공백문자를 기준으로 분리하면 리스트로 결과가 나오고 빈칸을 가진 문자열을 지정한 후에 이를 가지고 리스트 내의 문자를 결합하면 문자열로 결합한 것을 확인할 수 있다.



예제 4-17 : 개행문자가 있을 경우 문자열 분리

여러 문장을 처리하는 문자열(“”” “””)을 처리할 때는 일단 라인단위로 분리하고 이를 다시 붙여서 하나의 문장으로 만들어 표시할 수 있다.

일단 라인의 끝을 표시하는 “\n”을 기준으로 분리를 하고 이를 다시 연결하면 문장도 하나로 연결이 가능하다.



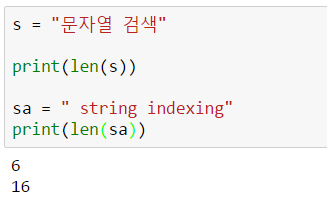
* 문자열 길이 및 검색

Len 함수로 문자열의 길이를 확인하고 그 내부에 있는 원소를 검색해서 표시한다파이썬은 정방향(양수)과 역방향(음수) index를 다 지원한다.

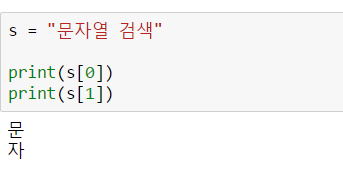
예제 4-28 : 검색 처리

Sequence 자료형은 정방향과 역방향이 다 가능하므로 검색 할 때 양수이면 좌측으로 검색해서 결과를 알려주고 음수일 경우 반대방향인 우측으로 이동하면서 검색결과를 알려준다.

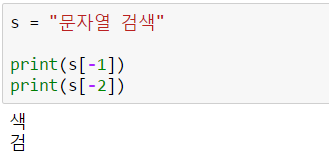
문자열의 길이는 문자열 내부에 있는 모든 문자들이 개수의 합이다. 유니코드로 처리하므로 한글도 하나의 문자로 처리되므로 문자열의 길이를 계산할 때도 한글 문자도 하나씩 증가하는 것을 확인할 수 있다.



정방향으로 검색하려면 검색에 대한 위치가 0부터 증가해서 총 문자열의 길이에서 -1을 뺀 인덱스를 사용한다.



역방향은 -1부터 시작해서 음수로 증가한다. 제일 마지막은 문자열 길이에 -1만큼 곱한 인덱스를 가지고 처리가 된다.



* unicode 에서 bytes로 전환하기

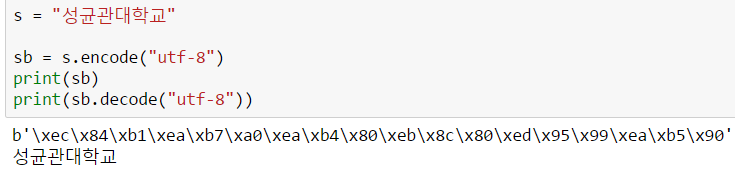
유니코드는 다양한 문자를 한 바이트에서 여러 바이트로 관리한다. 바이트로 전환하면 한글인 경우 대부분 문자당 2바이트로 전환된다.

유니코드 처리에 대한 조금 상세한 내용은 파일처리 부분에서 상세히 설명한다.

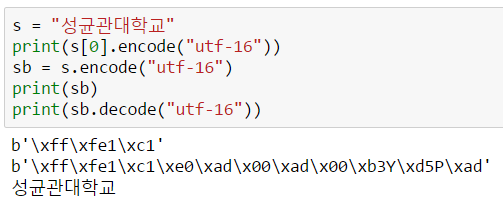
예제 4-19 : 암호화(encode)와 복호화(decode) 기준

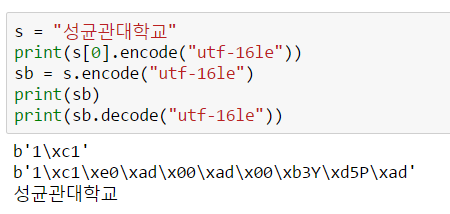
텍스트는 유니코드를 사용하고 이를 유니코드 타입의 bytes 단위로 encoding을 원하면 utf-8로 지정해서 변경을 한다.

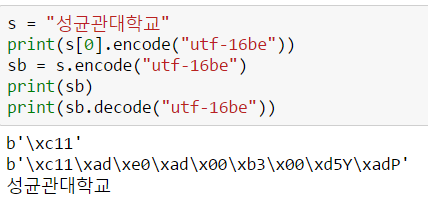
유니코드는 문자단위로 길이를 표시하지만 bytes로 변환이 된 경우는 바이트단위로 길이를 산정하므로 한글일 때는 유니코드와 바이트일 경우 총 원소의 개수가 달라진다.



유니코드를 utf-16, utf-16le로 변환하면 변환된 글자가 빅엔티언 처리와 반대로 표시되는 것을 확인할 수 있다.







## 4.3 바이트 자료형(bytes data type)

문자열은 유니코드를 처리하고 바이트 자료형은 bytes 단위의 문자열을 처리하도록 되어있다. 문자열처럼 변경이 불가능하므로 내부 처리되는 메소드도 동일하게 제공된다.

### 4.3.1. 바이트 생성

Bytes 타입도 리털렁은 b’문자열’과 bytes 클래스를 통해 인스턴스를 만든다.

* 리터럴과 bytes 생성자로 생성

ASCII 타입일 경우는 영문으로 표시되지만 그외 타입은 encoding 처리 후에 hex값으로 표시된다.

예제 4-20 : 바이트 생성

바이트로 생성할 때는 ASCII에 맞도록 처리되어야 한다. 리터럴로 생성할 때는 문자열 앞에 b를 붙여 바이트라고 명기를 해야한다.

또는 유니코드 문자열을 바이트로 변환해서 bytes생성자의 매개변수로 넣거나 유니코드 문자열을 넣고 encoding=”utf-8”를 넣어서 바이트를 생성하면 된다. 

### 4.3.2 바이트 주요 메소드

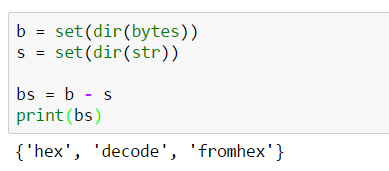
바이트(bytes) 클래스의 주요 메소드는 문자열과 거의 유사하지만 유니코드가 아닌 바이트 단위로 처리되므로 상이한 것이 몇 개 존재한다.

* 문자열과 다른 메소드들

문자열 자료형과 다른 메소드를 확인하기 위해 내부 속성의 이름으로 집합을 만들어서 2개의 집합의 차집합을 이용해서 바이트 타입에만 있는 3개의 메소드를 확인이 가능하다.

예제 4-21 : 바이트 자료형의 메소드 확인

파이썬에서 제공하는 Set 자료형을 이용해서 간단히 차집합으로 처리하면 바이트 자료형에 있는 메소드와 문자열의 메소드에 포함되지 않는 것만을 출력하면 3개의 다른 메소드가 있는 것을 확인할 수 있다.



바이트 자료형을 hex값으로 변환한다. 0x를 뺀 숫자만 표시하면 문자열로 hex값을 지정하면 이를 읽어와서 hex로 표시한다.



* 바이트 타입 에서 문자열로 전환

한글 등을 바이트 타입에서 사용하려면 문자열에서는 encode가 되어 바이트 타입으로 전환되어야 하고 바이트 타입을 문자열로 전환하려면 decode 메소드를 사용해서 전환해야 한다.

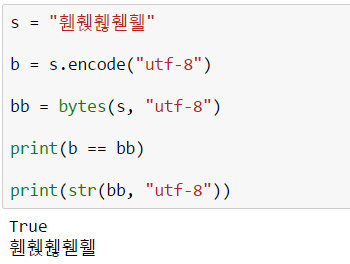
예제 4-22 : encode/decode 메소드 처리

문자열에서 encode 메소드로 bytes로 변환하고 bytes에서 decode 메소드로 문자열로 변환해서 확인하면 동일한 결과가 나오는 것을 알 수 있다.



예제 4-23 : bytes/str 생성자에서 직접 encode, decode 하기

클래스에서 직접 인스턴스를 만들 때에 encoding 처리까지 표시하면 문자열이 변경되어 저장하고 관리한다. encode/decode 메소드를 별도로 실행할 필요가 없다.



## 4.4. bytearray 자료형(data type)

Bytes 자료형은 str과 동일하게 변경이 불가하다. 바이트 타입에 대한 변경이 필요할 경우 bytearray로 생성해서 처리할 수 있다.

문자열과 거의 동일한 메소드들을 제공하지만 내부 원소에 대한 변경이 가능하므로 리스트에 필요한 메소드가 추가된다.

### 4.4.1. bytearray 생성

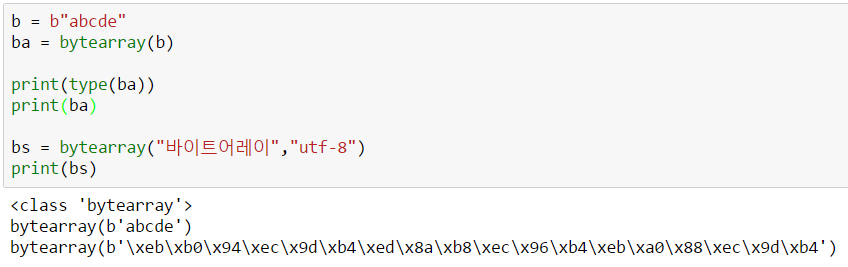
바이트 타입 중에 변경이 가능한 자료형을 생성하고 bytes 타입의 인스턴스를 변경하는 것을 이해할 수 있다.

* Bytearray 생성

클래스 byte는 변경이 불가능하므로 변경가능한 bytearray를 제공한다. 바이트 단위로 처리해야 하므로 문자열을 가지고 변경등을 사용하려면 encoding 처리가 되어야 한다.

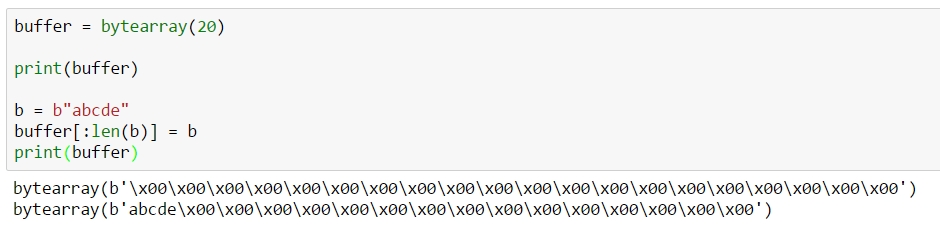
예제 4-24 : bytes로 생성한 것을 bytearray로 변환

바이트 문자열을 만들고 이를 바이트어레이로 생성이 가능하다. 직접 바이트어레이(bytearray) 생성자를 이용할 때 유니코드 문자열이 들어오면 직접 encoding을 처리해서 유니코드를 bytes 형태로 encode을 처리해서 바이트 기준의 bytearray 자료형을 생성한다.



예제 4-25 : 버퍼 처리하기 : bytearray

Bytearray 타입에 숫자를 주고 생성하면 숫자만큼의 길이를 \x00으로 바이트 단위로 버퍼를 만든다. 이 버퍼에 특정 값을 넣어서 처리가 가능하다. 일단 슬라이스를 이용해서 갱신을 실행했다.



### 4.4.2. bytearray 내의 속성/메소드

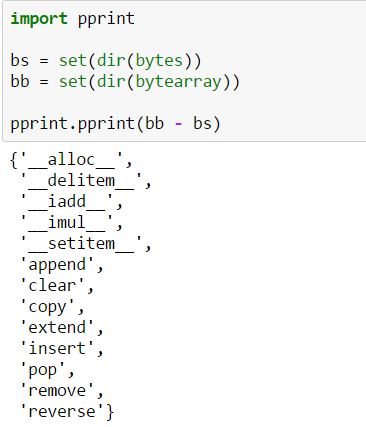
변경 가능한 bytearray는 리스트(list) 자료형처럼 내부 원소들을 갱신 및 삭제, 복사 할 수 있는 메소드들을 가지고 있다.

* Bytes 타입과 메소드 비교

변경가능한 bytearray 타입은 list에 가지고 있는 메소드들이 있어 삭제, 추가, 변경 등이 가능하다

예제 4-33 : bytes와 bytearray 메소드 차이 확인

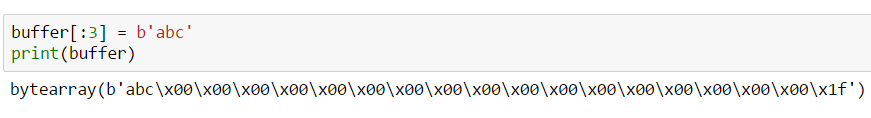
바이트어레이는 갱신이 가능하므로 원소 변경 및 삭제 등을 처리하는 메소드가 추가된 것을 볼 수 있다.



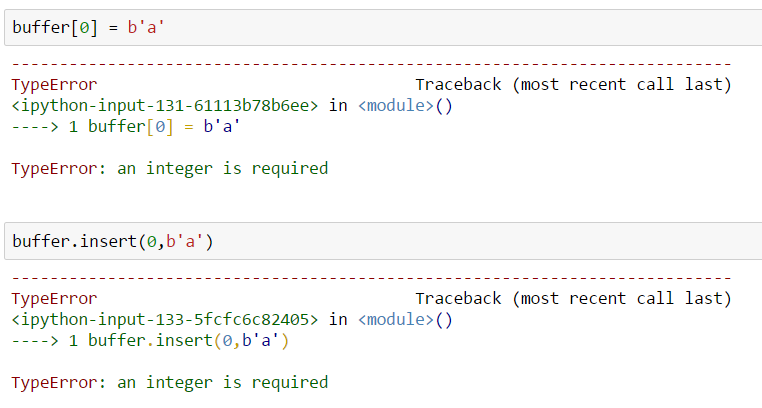
메소드 insert와 append 처리시에 값을 int로 주어야 처리된다.



슬라이스로 처리할 경우에는 대체가 되어 바이트 문자열도 처리가 된다.



바이트어레이 내의 메소드로 바이트를 갱신할 경우에는 바이트 문자가 아닌 정수값으로 넣어줘야한다. 바이트 문자열로 처리하면 예외가 발생하는 것을 볼 수 있다.



## 4.5. 튜플 자료형(tuple data type)

튜플 타입은 리스트 타입과 유사하지만 변경이 불가한 자료형이다. 파이썬 내부에서 변경 불가한 기능을 처리하기 위해 다양한 용도(가변위치인자, 결과값이 여러 개 존재 할 때 return문 등)로 사용되면 기본적인 개념은 데이터 들이 관계를 명확히 이용해서 처리하는 곳에 사용하기 위해 만들어진 것이다.

### 4.5.1. 튜플 생성

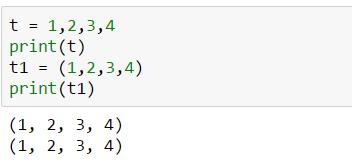
튜플도 리터럴과 tuple 클래스로 인스턴스를 생성할 수 있다.

* 튜플 리터럴 및 생성자

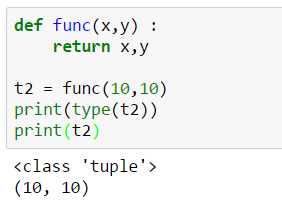
튜플 리터럴 생성은 괄호가 없이 쉼표를 기준으로 인식된다. 하지만 괄호를 사용해도 동일하게 인식되어 처리된다. Tupel 클래스를 이용해서 생성자로도 생성이 가능하다.

예제 4-27 : 튜플 리터럴 및 생성자로 생성

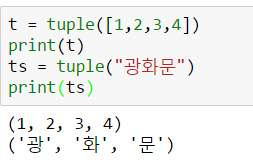
튜플은 괄호가 없어도 인지한다. 괄호를 사용할 때도 괄호의 의미는 단순히 그룹을 표시할 뿐이다. 둘다 튜플을 리터럴로 만드는 방식이다.



함수의 반환 결과가 여러 개일 경우 쉼표로 분리해서 사용한다. 이 방식은 튜플을 리터럴로 처리하는 방식과 동일하므로 실제 함수의 반환결과의 타입도 튜플이라는 것을 확인할 수 있다.



Tuple 클래스로 직접 생성도 가능하다. 다른 sequence 자료형을 인자로 받아 생성이 가능하다. 리스트로 들어오면 동일하지만 대괄호가 괄호로 바뀐 것을 알 수가 있다. 문자열로 들어오면 각 문자가 하나의 원소를 분리된 것을 할 수가 있다.

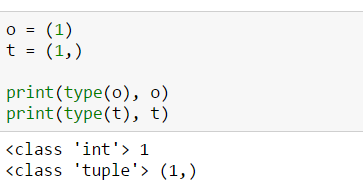


* 튜플 내의 원소가 하나 일 경우 주의 사항

튜플은 쉼표(,)로 인식되므로 원소가 하나라는 표시와 일반적인 괄호 연산과의 구별을 위해 괄호를 이용해서 튜플을 표시할 때 원소가 하나이면 반드시 쉼표를 표시해야 한다.

예제 4-28 : 단일 원소도 쉼표로 분리

괄호연산자와 튜플간의 구별을 보여주는 예시이다. 단순히 괄호연산으로 이해하면 숫자 1인 정수라는 것을 알 수 있다. 숫자와 쉼표가 같이 표현된 것은 튜플로 표현되는 것을 확인할 수 있다.



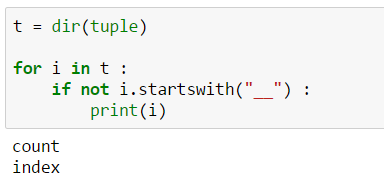
### 4.5.2. 튜플 메소드

변경이 불가하므로 count나 index 등 조회만을 위한 메소드를 제공한다.

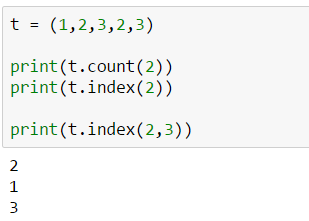
예제 4-39 : 튜플의 count, index 메소드

원소에 대한 개수(count)와 원소의 인덱스(index)를 찾아주는 메소드가 있다.

인덱스 메소드는 처음으로 동일한 것이 만나면 현재 위치의 결과를 반환해주므로 그 다음에 동일한 것을 찾지 않는다. 찾으려면 두 번째 인자에 위치를 주고 다시 실행해서 찾아야 한다.



동일한 원소의 개수를 확인하고 특정 원소의 위치를 확인해서 처리한다.

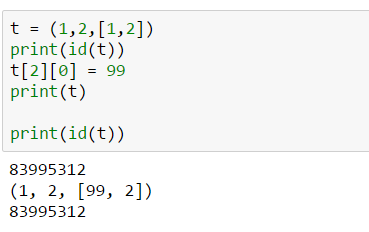


### 4.5.3. 튜플 원소로 Mutable 자료형 처리 방법

튜플은 완전하게 불변되는 자료형이 아니다. 튜플이 원소 중에 변경 가능한가능한 있을 때 튜플의 불변성이 깨진다. 튜플 내의 원소로 리스트가 들어오는 것을 예시로 왜 변경 가능한 원소가 들어오면 원소가 변경이 가능한 지를 이해해 보겠다.

예제 4-30 : 원소가 리스트일 경우

튜플 내부의 원소에 리스특 있을 경우 이 리스트의 원소는 변경이 가능하므로 튜플 내의 원소를 리스트 등 변경이 가능한 자료형을 넣을 때는 주의해야 한다.



## 4.6. 리스트 자료형(list data type)

변경 가능한 Sequence 자료형으로 다양한 자료형을 원소로 구성될 수 있다. int, float, tuple, dict, list 등의 다양한 자료형을 원소로 받을 수 있도록 구성이 가능하다. 각 원소는 순서를 가지므로 리스트를 처리할 때 index를 주어서 순서에 맞도록 검색이 가능하고 내부의 원소를 변경하거나 삭제 등을 할 수 있는 메소드들을 제공한다.

Sequence 처리를 위한 기본 자료형이므로 생성하는 법과 내부의 메소드를 이용해서 처리하는 기능을 잘 이해해야 한다.

### 4.6.1. 리스트 생성

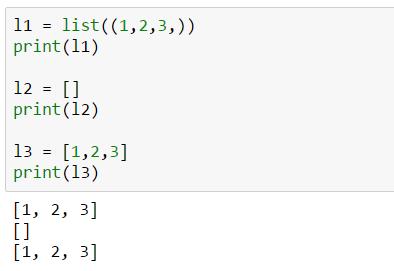
리스트도 리터럴과 list 클래스를 이용해서 인스턴스를 생성한다.

* 생성자와 리터럴 값으로 생성하기

리터럴과 생성자를 이용해서 생성하면 동일한 결과가 나온다.

예제 4-31 : 리스트 생성하기

첫째 튜플을 인자로 리스트로 전환하거나 리터럴로 리스트를 생성해도 동일한 원소를 같는 리스트가 생성되는 것을 볼 수 있다. 빈 리스트를 리터럴로 처리해도 결과는 동일한 빈 리스트를 표시한다.



### 4.6.2 리스트 복사 처리

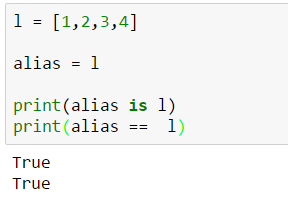
리스트를 복사하면 기본적으로 리스트 타입만 새로운 것으로 복사되고 리스트 내부의 원소들까지 새로운 것으로 만들지 않는다.

* 복사에는 2가지가 존재
* 얕은 복사(shallow copy) : 새로운 객체를 만들지만 내부 원소는 기존 원소를 참조
* 깊은 복사(deep copy) : 새로운 객체를 만들고 내부 원소들도 다른 원소로 만든다.
* 얕은 복사하기(shallow copy)

자기 자신을 새로운 사본으로 만들어서 새로운 객체를 만든다. 내부 원소까지 새로운 사본을 만들지 않기 때문에 원소로 리스트 등이 가변 객체가 있을 경우에는 공유가 되므로 원소 갱신할 때 원본과 복사본이 동시에 변경이 된다.

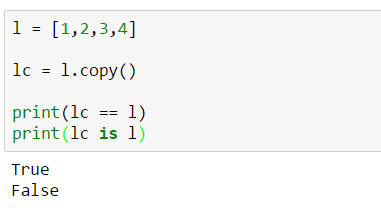
예제 4-32 : 리스트로 선언된 변수의 별칭 사용

리스트를 만들고 다른 변수에 할당하면 동일한 리스트를 가르키는 다른 변수가 생긴 것을 알 수 있다. 이를 별칭(alias)라고 부른다.



예제 4-33 : 리스트로 copy 메소드 사용

리스트의 copy 메소드를 사용하면 하나의 새로운 리스트를 만들어서 기존 리스트와 다른 리스트가 만들어진 것을 확인할 수 있다.

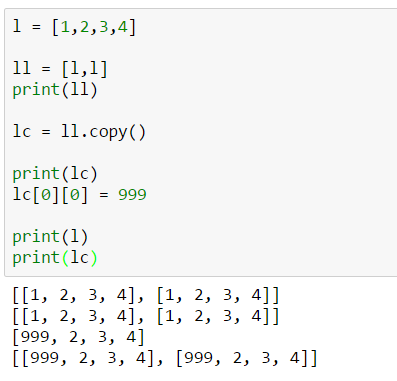


예제 4-34 : 리스트 내의 원소로 리스트를 가질 경우 copy 메소드 사용

리스트를 하나의 변수 l에 할당하고 이를 ll에 원소로 두번 지정을 했다.

이 ll를 copy해서 lc에 할당한다. 실제 새로운 사본이 만들어져서 내부의 원소인리스트를 수정해도 원본이랑 다를 것으로 생각하지만 변경이 가능한 리스트가 원소로 가질 경우 제일 밖에 있는 리스트만 복사가 된다.

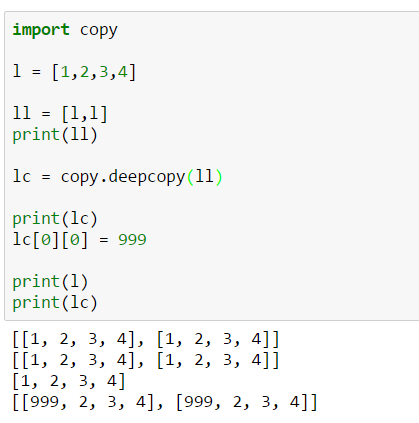
실제 원본인 l 변수 내부가 999로 변경되었으므로 이를 참조하는 모든 view는 항상 999가 첫 번째 원소의 값으로 보이는 것이다.



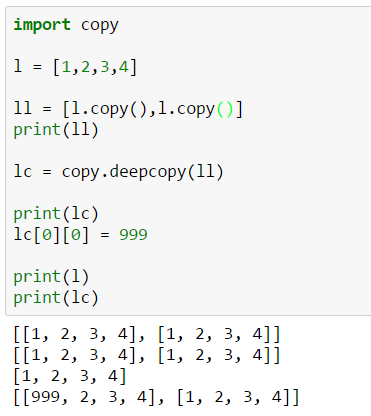
예제 4-35 : 깊은 복사(deepcopy)를 하는 이유

리스트 내에 내포된 리스트 등 변경이 가능한 자료형이 존재할 경우 원본이 변경되지 않도록 처리하기 위해서는 copy 모듈이 deepcopy 함수를 이용해서 새로운 리스트를 만들어야 한다.

원본은 갱신이 되지 않았지만 실제 내부의 있는 리스트 원소들이 동일해서 첫번째 원소가 갱신되는 것을 알 수 있다.



리스트를 원소로 가지려면 원본에서 복사한 사본을 전달을 받아야 원본가 관계가 없는 리스트가 만들어진다. 내부의 원소가 리스트일 경우 원본에 영향을 미치지 않으려면 항상 copy와 deepcopy를 이용해서 연결되는 고리를 끊어야 한다.



### 4.6.3 리스트 내의 메소드

리스트 내의 원소들을 처리하는 주요 메소드들에 대해 알아보자

예제 4-36 : 주요 메소드

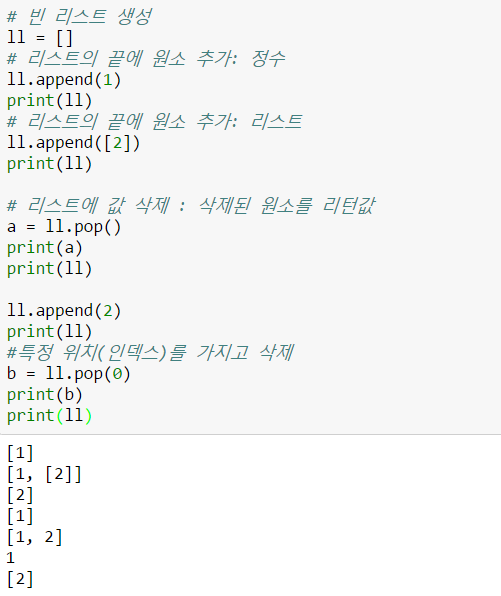
리스트 자료형에는 실제 리스트의 원소를 추가, 변경, 삭제할 수 있는 메소드들이 있다.



예제 4-37 : 리스트의 원소 추가 삭제 하기

리스트 내의 원소를 추가하는 append 메소드와 리스트 내의 맨 마지막에 있는 원소를 삭제하는 pop 메소드가 있다.

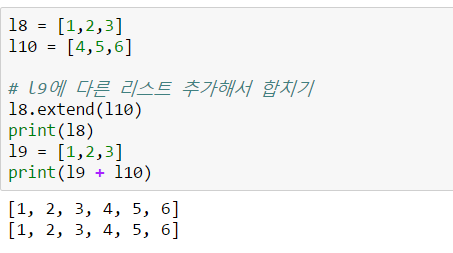
원소를 추가하는 append 메소드에 인자가 리스트로 들어와도 원소로 추가한다. Pop 메소드에 index를 인자로 넘기면 그 인덱스에 있는 원소를 삭제한다.



예제 4-38 : 리스트를 합치기 : extend

원본 리스트에서 extend 메소드를 이용해서 다른 리스트를 인자로 받아 원본 리스트에 하나로 합친다.

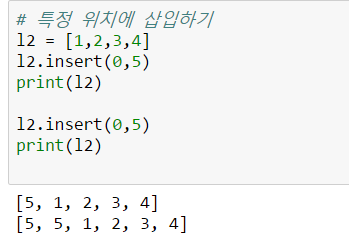
리스트와 리스트를 +연산자로 합치면 기존 원본은 그래도 있고 새로운 리스트가 생성된다. Extend 메소드를 이용하면 기존 원본 리스트에 새로운 리스트를 합쳐서 원본을 변경한다.



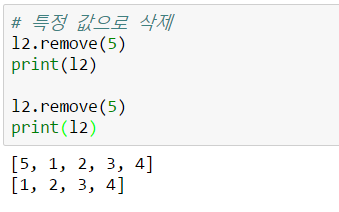
예제 4-39 : 리스트 특정 위치에 삽입 및 전체 삭제

특정 원소를 삽입하는 insert 메소드와 특정 값 하나를 삭제하는 remove 메소드가 있다. 전체 원소를 삭제하는 clear 메소드도 가지고 있다.

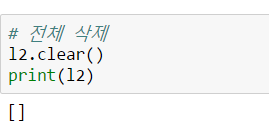
리스트를 정의하고 특정 원소의 값을 인덱스를 부여해서 하나씩 삽입도 가능하다.



특정 인덱스를 주고 원소를 추가하면 실제 원본 리스트에 원소가 삽입된다. 특정 원소의 값을 가지고 삭제하면 삭제된다. 삭제는 한 개씩만 된다.

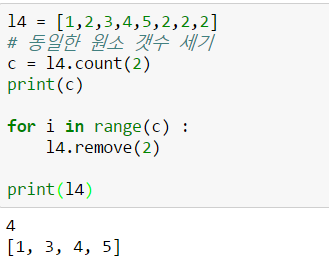


리스트를 정의하고 전체 원소를 삭제하려면 clear 메소드를 이용해서 처리한다.



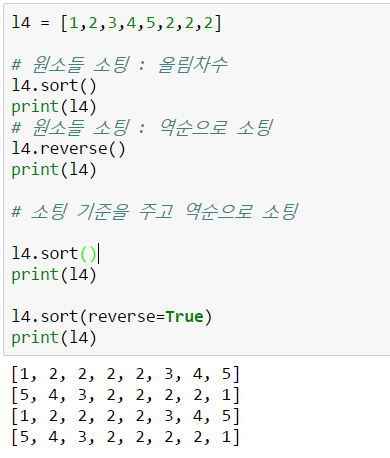
예제 4-40 : 리스트 내의 동일원소를 확인하고 삭제하기

동일한 원소를 count 메소드로 세고 이를 for문에 range에 넣어 동일한 값을 remove하면 전부 삭제된다.



예제 4-41 : 리스트 원소들 정렬하기 : sort, reverse

리스트 내부 원소들을 올림차순 별로 정렬도 가능하고 reverse=True을 인자로 주면 내림차순 별로 정렬이 가능하다. 역으로 정렬하기는 index 기준으로만 정렬한다. Sort에 reverse=True를 넣어서 정렬해도 동일하게 처리된다.

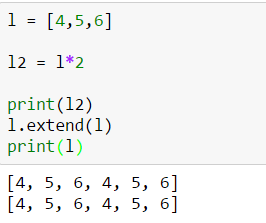


### 4.6.4 리스트 내의 리스트 초기화 하는 법

리스트 내에 리스트를 원소로 구성할 때 내부 원소의 리스트가 명확히 다른 것으로 인식되려면 내부 리스트를 넣을 때 매번 초기화 처리를 해줘야 한다.

예제 4-42 : 리스트로 \* 연산자 비교하기

리스트 내에 있는 원소를 반복해서 구성하라는 \*연산이므로 같은 원소를 반복해서 새로운 리스트를 만든다. 리스트 내의 extend 메소드와의 차이점은 새로운 리스트를 만드는 것이다.



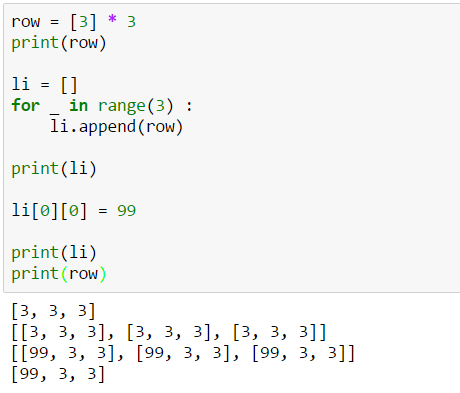
* 리스트를 초기화할 때 주의사항

리스트를 초기화 할 때도 리스트 초기화하는 위치에 따라 리스트의 객체를 공유할 수 있다. 공유되지 않도록 처리하기 위해서는 리스트도 매번 초기화처리가 필요하다.

예제 4-43 : 외부에 리스트 원소를 반복해서 처리하고 내부에 넣고 처리

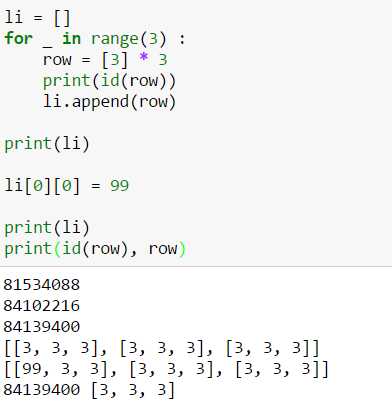
리스트에 \* 연산자로 외부에서 리스트를 만들었다. 이때 리스트에 대한 정보과 고정된다. 이를 가지고 for문으로 순환하면서 다른 리스트에 원소로 외부에서 만든 리스트를 넣을 경우 기존 리스트에 대한 동일한 레퍼런스가 계속적으로 들어간다.

리스트 내의 리스트 원소를 갱신할 경우 동일한 리스트를 참조하고 있으므로 동일한 값이 변경된 것처럼 보인다.



예제 4-44 : 리스트를 for 문 안에서 초기화

for 문 내에서 내부 리스트를 만들 경우 순환할 때마다 새로운 리스트가 만들어지는 것을 확인할 수 있다. 리스트의 첫번째 리스트의 원소를 갱신해도 리스트 내의 원소인 다른 리스트에는 아무런 영향이 없는 것을 확인할 수 있다.

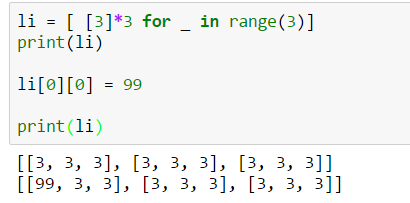


예제 4-45 : 지능형(comprehension) 리스트로 처리하기

지능형 리스트 내에서 리스트를 초기화 할 경우는 for 순환처럼 매번 리스트가 다시 만들어져서 리스트 내의 원소로 들어간다.

실제 지능형 리스트도 지역변수를 가지고 있어서 이 리스트의 레퍼런스 밖으로 공유가 되지 않으므로 실제 리스트 내의 리스트를 갱신할 때 원본과 연결고리가 전혀 발생하지 않아 다른 리스트에 영향을 미치지 않는다.

리스트에 리스트 원소를 넣어서 초기화할 경우 지능형 리스트를 사용해서 처리하는 것이 제일 좋다.

.

## 4.7. Sequence 자료형 형변환

파이썬의 형변환은 다른 자료형으로 다시 만드는 것으로 숫자 자료형과 동일한 방식을 사용한다.

### 4.7.1. Sequence 생성자를 이용한 형변환

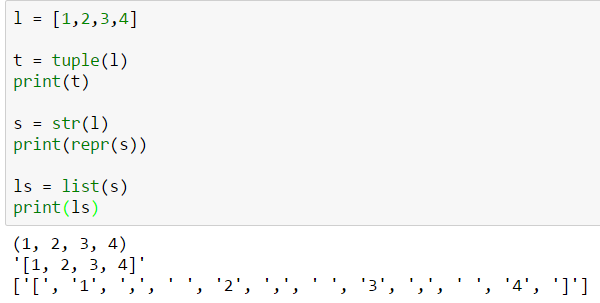
리스트를 문자열로 형변환을 하면 리스트 전체를 문자열로 만들어 버린다. 이를 다시 리스트로 전환하면 원하지 않는 결과가 나온다.

예제 4-46 : 클래스만 사용한 형변환

리스트와 튜플간의 형변환은 배열의 구조가 동일하므로 쉽게 변환이 된다. 차이점은 리스트는 변경가능한 것이고 튜플은 변경이 불가능한 구조이 차이만 있다.

리스트를 문자열로 형변환을 하면 리스트 전체를 문자열로 처리하므로 내장함수repr 로 출력했다.

문자열을 리스트로 다시 형변환을 하면 문자열 전체 구성 요소를 분리해서 리스트로 형변환이 생기는 것을 볼 수 있다.



### 4.7.2. Sequence 메소드를 이용한 형변환

문자열의 join 메소드를 이용해서 리스트나 튜플에 내의 원소들을 합쳐서 문자열로 형변환도 가능하다.

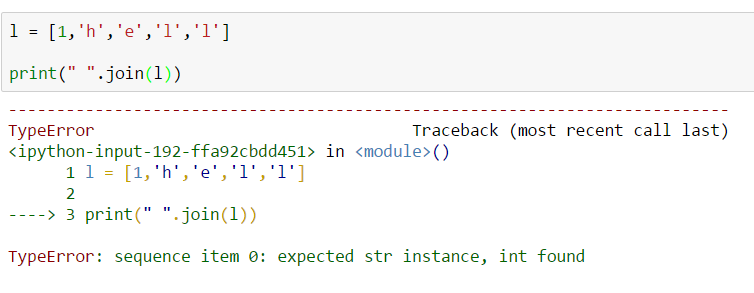
예제 4-47 : 문자열 원소로 구성된 리스트나 튜플을 문자열로 변환

문자열을 리스트나 튜플로 변환을 하면 문자별로 분리되어 리스트나 튜플의 원소로 저장된 것을 볼 수 있다. 이를 다시 문자열로 변환하려면 join 메소드를 이용해서 통합해야 한다.

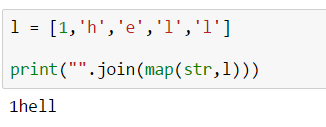


예제 4-48 : 문자열이나 숫자로 구성된 리스트나 튜플을 문자열로 변환

리스트를 문자열로 변환이 필요할 경우 원소 내부가 전부 문자열이어야 한다. 문자열이 아닌 경우는 문자열로 변환해야 한다. 문자열이 아닌 경우 예외를 발생시켰다.



리스트 원소를 전부 문자열로 변환하려면 내장함수인 map 을 이용해서 str을 매개변수로 전달을 받아서 리스트 원소들을 전부 문자열로 변환해서 처리한 후에 join 메소드가 실행되어야 문자열로 전환이 된다.



## 4.8. 요약

Sequence 자료형에는 변경가능(list, bytearray)하거나 변경불가(str, tuple,bytes)한 자료형이 존재한다. 변경가능 여부도 실제 내부 원소를 변경할 수 있는지를 확인하는 것이다.

* 문자열 처리 : str
* 바이트 처리 : bytes, bytearray
* 배열 처리 : list, tuple

Sequence 자료형 내의 다양한 메소드는 각 자료형의 특징에 따라 변경불가능할경우는 대체(새로운 객체 생성)이나 변경이 가능할 경우는 내부 원소를 추가, 삭제, 삽입 등을 처리한다.

리스트일 경우는 copy할 때 내포된 원소까지 전부 복사해서 사용할 지를 명확히 지정해서 copy와 deepcopy 사용여부도 결정해야 한다.

**5. Mapping/Set 자료형(data type)**

파이썬에서도 hash로 데이터의 key를 관리하고 이 key에 매핑된 값도 처리하는 Mapping 과 key로만 처리하는Set에 대한 자료형을 가지고 있다.

Mapping은 딕셔너리(dict), Set은 집합(set, frozenset )을 내장 자료형으로 제공한다.

이 3가지 중에 변경 불가능(immutable) 자료형은 frozenset이다.

## 5.1. 파이썬 Mapping 자료형의 특징

원소를 key와 값으로 구성하는 자료형이고 각 Key는 해쉬(hash)를 생성해서 저장한다. 해쉬(hash)의 특징은 순서가 없이도 저장되고 검색이 가능하다. 대신 indexing 처리보다 더 많은 메모리가 사용된다는 단점이 있다.

이번 장은 순서가 없이 저장되어 관리하는 mapping과 set 에 대해 알아보기로 하겠다.

### 5.1.1 딕셔너리(dict) 의 키 구성 및 생성 기준

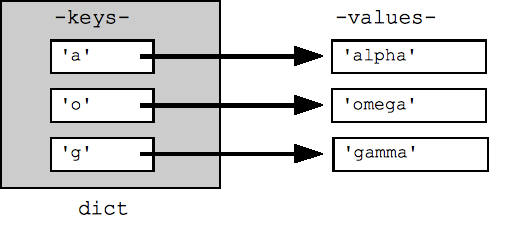
변경 불가능한 자료형(int, float, tuple, str, bytes, frozenset 등)이 키가 가능한 이유는 변경이 가능해야 해쉬를 이용해서 변경되지 않는 키를 만들 수 있다. 튜플의 원소가 리스트가 들어오면 키로 만들 수 없다.

* 딕셔너리(dict) 의 구조

해쉬 처리된 키와 값으로 구성되어 항상 키를 가지고 값을 처리될 수 있는 일대일 mapping 관계를 가지는 자료형 구조이다.

딕셔너리 자료형은 객체가 수명 동안 변경되지 않는 해시값(\_\_hash \_\_ )이 있고 다른 객체 (\_\_eq \_\_ )와 비교할 수 있는 해시 가능 객체입니다.

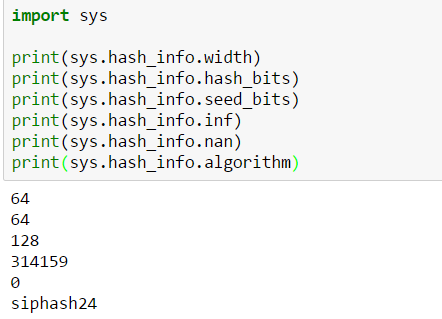
equal을 비교하는 해시 가능 객체는 동일한 해시 값을 가져야 하며 해시 가능성 (Hashability)은 객체가 사전 키와 집합 멤버로 사용할 수 있다. 이러한 데이터 구조는 해시 값을 내부적으로 사용하기 때문입니다.



예제 5-1 : 키를 구성하는 해쉬(hash)를 구성하는 기본 정보

딕셔너리 키에 사용되는 해쉬에 대한 기본 정보를 sys 모듈로 조회해보면 아래와 같은 속성들을 가지고 있다.

* width : 해시 값에 사용되는 비트의 너비
* hash\_bits : 해시 알고리즘의 내부 출력 크기
* seed\_bits : 해시 알고리즘의 seed key 크기
* inf : 양의 무한대에 해시 값이 반환
* nan : Nan에 대한 해쉬값
* algorithm : str, bytes, memoryview의 해시 알고리즘의 이름



### 5.1.2. 딕셔너리(dict) 생성하기

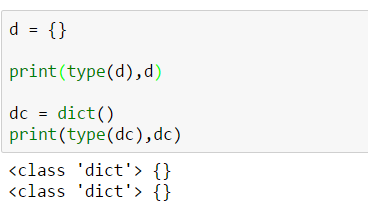
리터럴과 dict 클래스로 딕셔너리(dict)타입의 인스턴스를 생성할 수 있다.

* 딕셔너리(dict) 타입의 빈 인스턴스 생성하기 :

빈 딕셔너리는 중괄호를 이용해서 정의한다. Set 타입과 동일한 기호를 사용하지만 빈 딕셔너리에서 { }을 사용하는 것을 알 수 있다.

예제 5-2 : 빈 dict 생성

{ } 기호는 빈 딕셔너리(dict) 자료형의 텅빈 딕셔너리 인스턴스를 표시한다.

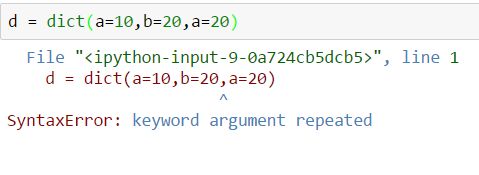


* 딕셔너리(dict) 의 키의 중복 처리

딕셔니리는 키가 생성되면 유일성을 유지한다. 동일한 키가 발생할 수 없는 구조를 가지고 있다. 키워드 인자로 처리 시에는 문법 상의 예외를 발생시키지만 리터럴 등으로 처리할 경우에는 생성된 키가 존재하면 그 키는 그대로 있고 나중에 있는 값으로 갱신이 된다.

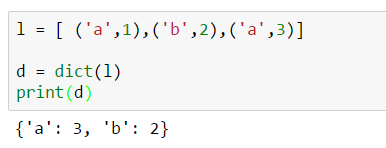
예제 5-3 : 키워드 인자에 동일한 키를 중복 시킬 경우

생성자에서 키워드 인자(keyword argument) 즉 키와 값으로 입력할 때에 키가 중복되면 오류가 발생한다.



예제 5-4 : 튜플 원소 리스트로 동일한 키를 중복 시킬 경우

2튜플의 원소를 가진 리스트로 딕셔너리 생성자를 통해 딕셔너리 인스턴스를 생성할 때에는 실제 내부에서 키(key) 중복에 대한 예외를 발생하지 않는다. 기존에 생성된 값을 최종 것으로 갱신한다.

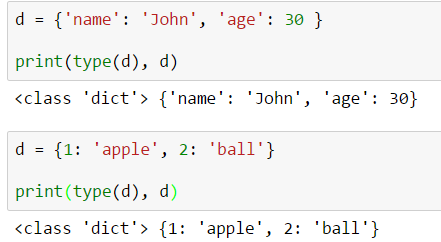


* 딕셔너리(dict) 생성 할 때 키값의 자료형

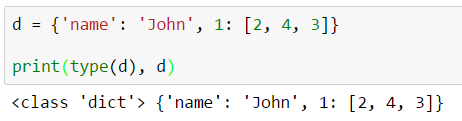
변경이 불가능한 자료형을 키로 사용할 수 있다. 한번 생성되고 변하지 않아야 hash로 만들 수 있다.

예제 5-5 : 딕셔너리 키에 대한 처리 방식

딕셔너리의 키를 변경이 불가능한 문자열과 숫자로 정의해서 생성도 가능하다.

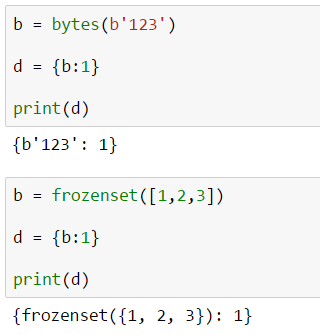


딕셔너리의 키를 정수와 문자열을 혼용해서 생성도 가능하다.



예제 5-6 : bytes와 frozenset도 키로 사용가능

변경 불가한 자료형으로 키를 생성이 가능하므로 bytes와 frozenset도 키가 가능하다. 특히 frozenset에 리스트나 set 자료형을 넣어서 키로 만들어서 처리가 가능하다.



* 리터럴 값을 가지고 딕셔너리 생성

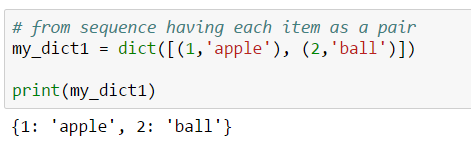
리터럴로 딕셔너리를 생성할 수 있다. 키와 값이 구조를 가진 리터럴이 들어오면 dict 생성자에서 키와 값을 매핑해서 생성한다.

예제 5-7 : 2 튜플 원소를 가진 리스트 인자

딕셔너리를 생성하기 위해 딕셔너리 인스턴스를 넣어서 생성해 본다.



위의 딕셔너리 인스턴스와 동일한 구조인 리스트 내의 원소가 2개의 원소를 갖는 튜플로 입력하면 키와 값을 자동으로 매핑해서 딕셔너리 인스턴스를 생성한다.

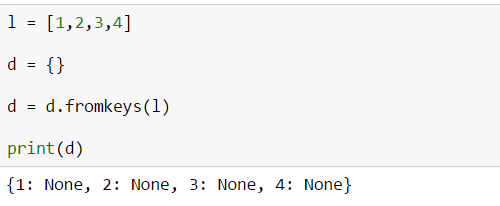


* 특정 key를 가져와서 딕셔너리(dict) 자료형 생성

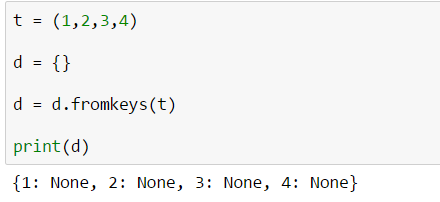
리스트 내의 원소가 dict 타입이 키가 가능할 경우 fromkeys 메소드를 이용해서 키값을 추가할 수 있다.

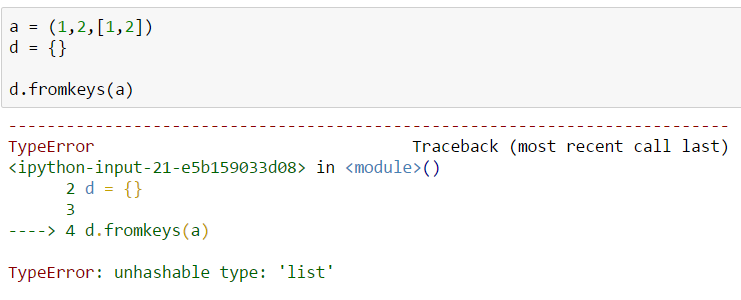
예제 5-8 : 리스트를 이용해서 키만 생성

리스트를 만들고 이 리스트를 fromkeys 메소드의 인자로 받아 키값만 생성하고 값은 None으로 처리한다.



튜플의 원소도 fromkeys 메소드를 가지고 새로운 딕셔너리 인스턴스를 생성할 수 있다.



다만 튜플 내에 리스트 원소가 있으면 이 리스트가 변경이 되므로 딕셔너리 타입의 키로 사용할 수 없어 예외가 발생한다. 

## 5.2. 딕셔너리(dict)타입 메소드 이해하기

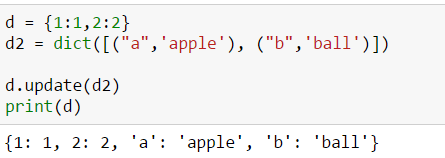
키와 값으로 구성된 딕셔너리(dict) 자료형은 키 순서대로 저장이 되는 것이 아니라 키가 해쉬(hash)로 처리되어 순서가 없어도 바로 검색할 수 있는 구조를 만들어서 저장된다. 내부 원소에 접근할 때도 반드시 키를 가지고 접근해야 빠른 검색이 가능하다.

### 5.2.1. 딕셔너리(dict) 자료형 추가 /삭제 메소드

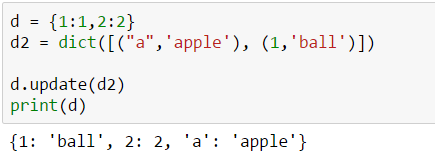
런타입 시점에 생성된 dict 클래스의 인스턴스에 대한 원소들을 추가 및 삭제를 해본다.

예제 5-9 : 딕셔너리에 딕셔너리 추가 : update

딕셔너리에 딕셔너리를 넣어서 기존 딕셔너리에 확장을 처리한다.



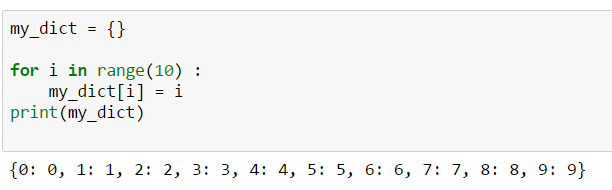
동일한 키가 존재할 경우 키는 그대로 있지만 저장되는 값이 인자로 전달된 딕셔너리 내의 값으로 갱신되어 있는 것을 확인할 수 있다.



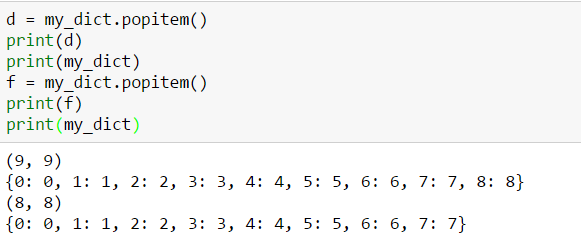
예제 5-10 : 딕셔너리 내부 item 삭제하기

딕셔너리(dict) 타입은 순서가 없으므로 임의의 항목(key/value)를 삭제하거나전체를 한번에 삭제할 수 있다.

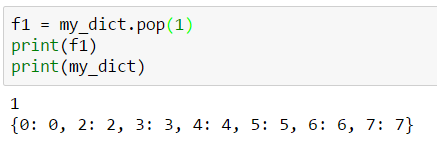
딕셔너리를 순환문으로 키와 값이 동일한 것을 생성한다.



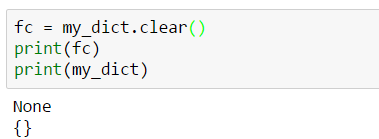
딕셔너리 내의 popitem 메소드로 삭제하면 제일 특정 딕셔너리의 원소를 삭제하고 삭제된 값을 반환한다.



특정 키를 인자로 넣어서 임의의 원소를 꺼내고 그 원소를 삭제하면 값을 반환한다.



메소드 clear를 실행하면 내부에 있는 원소가 전부 삭제하고 빈 딕셔너리가된다.



### 5.2.2. 딕셔너리(dict) 타입에 keys, values, items 이해하기

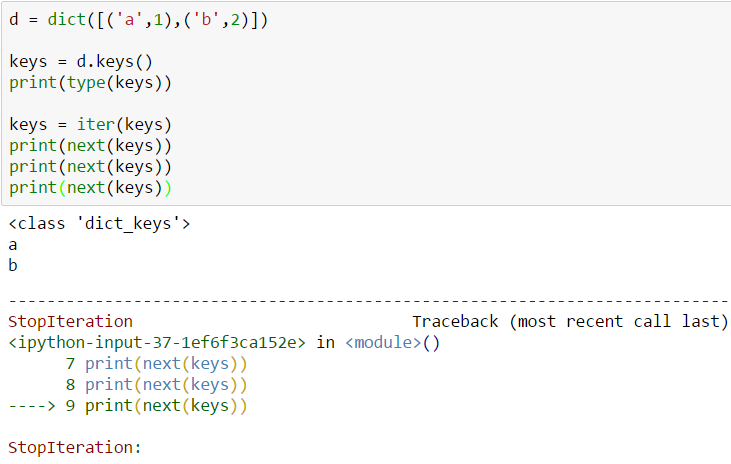
딕셔너리도 내부의 키와 값을 제공하는 view 메소드를 제공하고 그 메소드의 처리되는 결과를 별도의 자료형으로 제공한다. 뷰에 대한 별도의 자료형인 dict\_keys, dict\_values, dict\_items는 실제 반복자를 제공해서 값을 하나씩 호출할 때마다 처리하는 구조를 가지고 있다.

* 반복형 처리가 가능하도록 처리

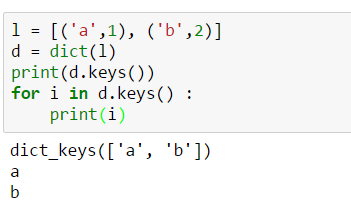
반복형이 가능하도록 클래스 정의 내부에 \_\_iter\_\_가 생성되어 있어 iter 함수를 이용해서 반복형을 만들고 next 함수로 호출이 가능하도록 구성된다.

예제 5-11 : keys 메소드로 key만 읽어오기

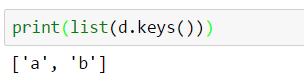
딕셔너리 내의 key만 읽어오는 새로운 view 객체(dict\_keys)를 만든다. 이를 반복자로 변환을 iter함수를 하고 next함수로 읽어서 처리가 가능하다.



내부에 구성된 전체의 키 값을 하나의 객체로 가져와서 이를 for순환문으로 처리하면 내부의 키만 출력되는 것을 확인할 수 있다.

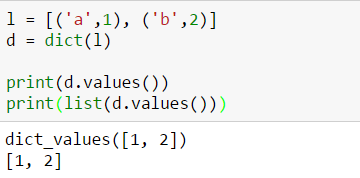


바로 list 생성자에 넣으면 키값이 리스트로 변환되어 처리된다.



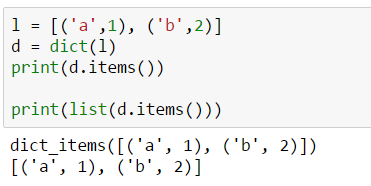
예제 5-12 : values로 value만 읽어오기

딕셔너리(dict) 타입 내에 value값을 가져오는 메소드로 결과값을 views 타입의 인스턴스로 반환한다.



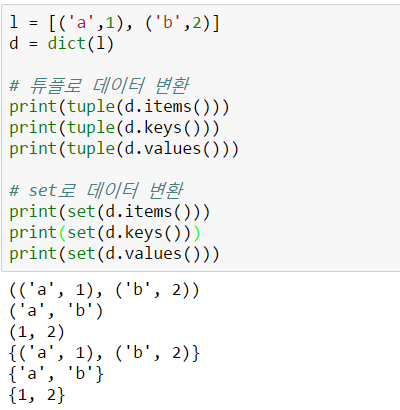
예제 5-13 : items로 key/value 읽어오기

key와 value를 전부 조회할 경우 사용하는 메소드이면 이를 리스트로 변환하면 내부는 튜플로 전환되어 보여준다.



예제 5-14 : 딕셔너리(dict) 내부 원소를 조회하고 tuple/set 변환

items, keys, values 메소드의 결과 tuple, set으로 변환도 가능도 가능한 이유는 반복자 타입으로 만들어져 있기에 생성자와 만나면 그 생성자에 맞춰 새로운 자료형의 인스턴스가 만들어지는 것이다.

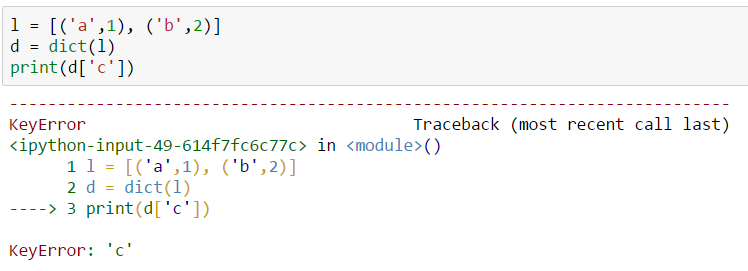


### 5.2.3. 딕셔너리(dict) 자료형 조회 및 갱신 메소드

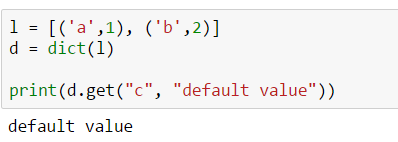
딕셔너리 인스턴스에서 없는 키를 가지고 조회하거나 갱신이 필요한 경우에 대한 예외를 방지하기 위해 메소를 제공한다.

예제 5-15 : 없는 키를 조회하거나 갱신

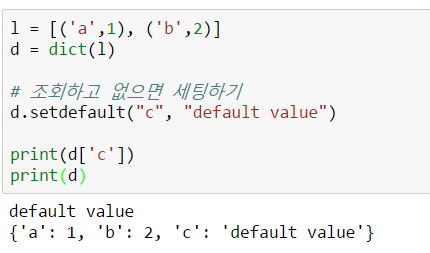
두 개의 키를 가지는 딕셔너리를 만들었다 조회할 때 없는 키로 검색을 하면 예외를 발생한다.



검색할 때 예외를 방지하기 위해 get 메소드에 키와 default 값을 넣어서 처리하면 실제 키가 없어도 예외를 발생시키지 않고 default 값을 반환한다.



딕셔너리에서 실제 키가 없을 경우 키와 값이 갱신할 필요가 있는 경우에는 get 메소드 대신에 setdefault 메소드로 처리해서 키와 값을 넣어서 처리하고 이 키를 조회하면 기본 값으로 세팅되어있으므로 이를 처리하는 로직을 추가해서 없었던 키라는 것을 알 수 있으면 된다.



## 5.3. set 자료형 처리 하기

파이썬에서 hash 처리가 되는 집합 자료형이다. 집합은 수학의 집합을 표현하도록 만들었다. 2가지의 set, frozenset 자료형을 제공한다.

집합 자료형에 주요 메소드는 수학의 기본 집합연산을 처리할 수 있다.

### 5.2.1 set

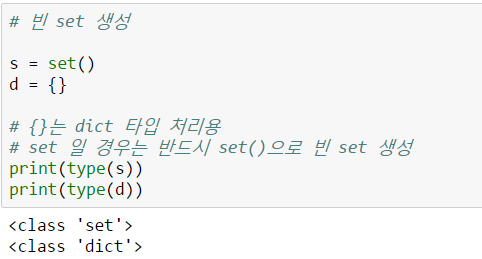
변경 가능한 집합인 set 클래스에 대해 알아보자

* set 생성하기

딕셔너리 자료형에서 아무것도 없는 딕셔너리 표현을 { }으로 정의되어 있어 빈 set 타입은 set() 으로 표현한다.

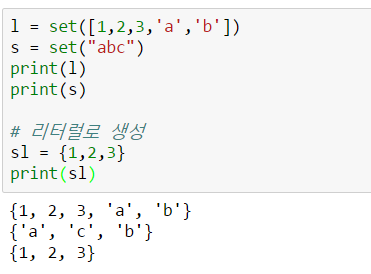
예제 5-16 : 빈 set 생성은 반드시 set()으로 처리

딕셔너리 자료형에서 빈 딕셔너리를 중괄호를 사용하기 때문에 set 자료형에서 빈 집합은 set()로 처리한다.



예제 5-17 : 리터럴이나 생성자로 집합 만들기

생성자를 이용해서 리스트, 문자열 등을 넣어 set을 생성할 수 있다. 중괄호를 이용해서 바로 리터럴로도 생성이 가능하다.



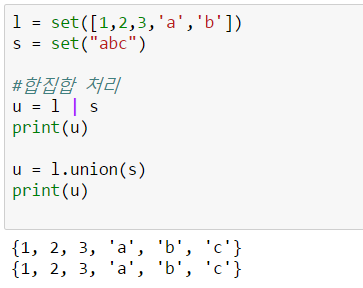
* set 기본 연산 처리하기

집합 자료형은 수학의 집합연산을 그대로 수용해서 처리하도록 메소드와 연산자를 다 사용해서 연산이 가능하다..

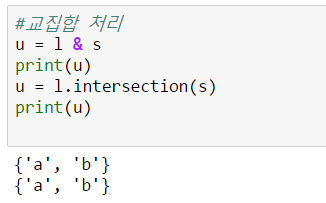
예제 5-18 : 집합 연산에 대한 연산자와 메소드 처리하기

합집합(|, union), 교집합( &, intersection), 차집합(-, difference) 등의 집합연산을 처리하는 메소드들을 제공한다. 연산자와 메소드의 처리결과는 자기 자신을 변경하는 것이 아니라 새로운 인스턴스를 만들어서 반환하는 것을 알 수 있다.

두 집합을 가지고 합집합을 연산자와 메소드로 처리를 한다. 동일한 결과를 출력하는 것을 확인할 수 있다.



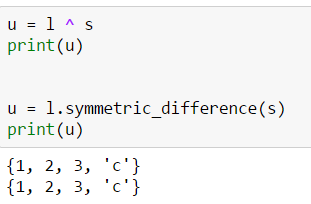
두 집합에 대한 교집합을 연산자와 메소드로 처리해도 동일한 결과를 출력한다.



두 집합에 대한 차집합을 연산자와 메소드로 처리해도 동일한 결과를 출력한다.



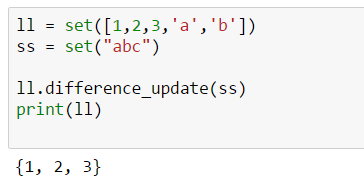
집합에 대한 대칭 차집합에 대해서도 연산자와 메소드를 가지고 처리하면 동일한 결과가 출력된다.



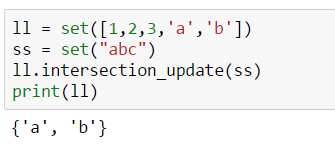
예제 5-19 : 집합 연산을 통해 자기 내부 변경하기

기본 메소드는 인스턴스를 갱신하지 않고 메소드가 처리된 결과를 다른 집합의 인스턴스를 만들고 반환하지만 difference\_update, intersection\_update, symmetric\_difference\_update 메소드는 자기 내부의 원소들을 변경한다.

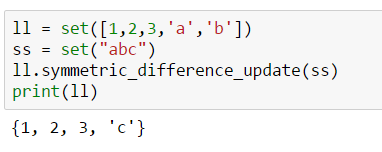
차집합을 처리하고 자기 내부를 차집합의 결과로 갱신한다.



교집합을 처리하고 자기 내부를 교집합의 결과로 갱신한다.



대칭 차집합을 처리하고 자기 내부를 대칭차집합의 결과로 갱신한다.



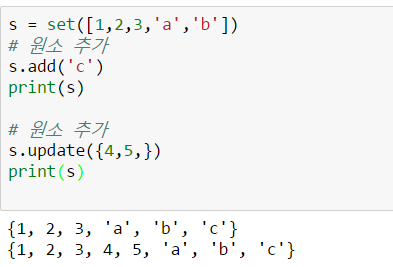
* set 내부 원소 값 조정하기

set 자료형은 변경이 가능하므로 내부 원소에 대한 원소 추가, 삭제 등을 처리하는 메소드를 제공한다.

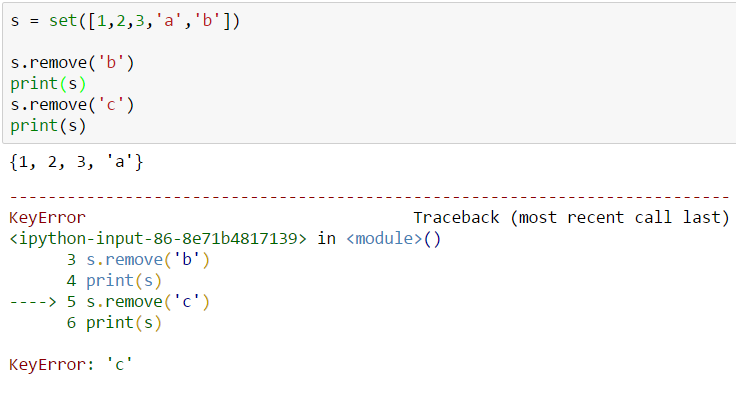
예제 5-20 : 집합 원소 처리

원소를 추가하는 add 메소드, 원소를 삭제하는 pop 메소드, 원소의 값을 삭제하는 remove 메소드나 discard메소드가 있고 discard는 실제 원소가 없으면 처리하지 않는다.

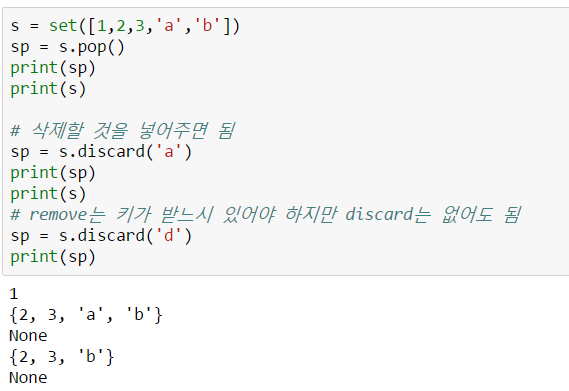
집합에 대한 원소 하나는 add로 추가하고 집합을 기준으로는 update 메소드로 추가한다.



집합을 삭제할 때 remove 메소드는 주의해서 사용해야 한다. 해당 값이 없으면 KeyError를 발생시킨다.



원소를 임의로 삭제하는 pop 메소드와 명확히 부여해서 discard로 삭제를 한다. Discard로 삭제할 때 실제 키가 없어도 처리된 결과를 반환하지 않는다.

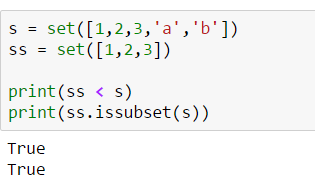


* 집합간의 관계를 확인하기

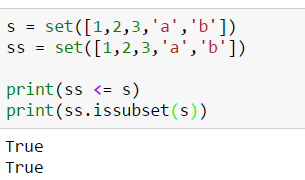
두 개의 집합를 만들어서 두 집합의 어느 집합의 부분 집합인지에 대한 메소드나 연산자도 있다 집합간의 계층관계를 확인할 수 있다.

예제 5-21 : 집합간의 관계 확인 연산자 및 메소드

두 집합을 비교해서 부분집합인지를 확인한다.



두 집합의 동등한 관계 여부도 확인할 수 있다.



### 5.2.2. frozenset 이해하기

Set 자료형과이 차이점은 생성되면 원소 내부를 변경할 수 없다는 것이다. Set 자료형은 변경이 가능해서 딕셔너리 키로 사용할 수 없다. Frozenset로 자료 변환을 하면 딕셔너리 키로 사용이 가능하다.

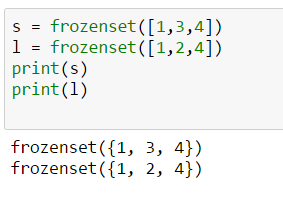
예제 5-22 : frozenset생성하기

집합을 만들 때 변경되는 않는 집합도 필요하다. 이 집합은 딕셔너리 타입의key로도 사용이 가능하지만 한번 생성되면 원소를 추가나 삭제할 수 없다. .

생성자의 인자를 넣지 않고 객체를 생성하면 빈 객체를 만들면 표시frozenset()로 표현된다.



변경이 불가능하지만 최초 생성시에는 실제 생성되는 원소를 넣고 만든다.

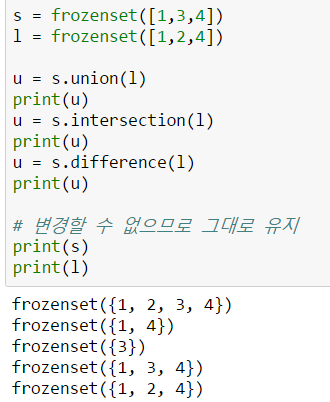


* 연산 처리

변경 불가한 자료형이므로 집합 연산 결과는 내부를 변경하지 않고 새로운 인스턴스 객체를 만든다.

예제 5-23 : set과 차이점은 새로운 연산의 결과를 만든다.

연산을 위한 메소드가 제공되지만 결과는 원본을 변경하지 않고 항상 새롭게 다른 객체를 만들어서 반환처리한다.



## 5.4. 요약

Mapping 타입과 Set 타입에 대한 생성하는 방법과 내부에 제공되는 메소드들을 간단히 실행해봤다.

특히 Mapping 타입은 다양한 방식을 통해 생성할 수 있고 딕셔너리(dict) 타입의 단점들을 보완한 데이터 구조는 뒤에 자료구조를 설명하는 장에서 세부적으로 알아보기로 하겠다.

* 딕셔너리 키구성 : 해쉬처리
* Mapping 자료형 : 딕셔너리
* Set 자료형 : set, frozenset

**6. 정수형 검색, 지능형, 문자열 포맷팅하기**

Sequence 자료형인 문자열, 리스트 등과 mapping 타입인 dict에 대한 검색 방식인 indexing, 부부집합 검색 방법인 slicing에 대해 알아본다

또한 리스트,dict, set 타입에 대한 지능형 처리(comprehension) 및 IO처리를 위한 문자열에 대한 formatting 처리를 이번 장에서 이해해 보기로 하겠다.

## 6.1. 정수/키 검색(indexing) 이해하기

Sequence (문자열, 리스트, 튜플 등) 자료형은 실제 index인 정수로 검색해서 조회, 수정, 삭제가 가능하고 딕셔너리(dict) 자료형은 key로 검색해서 조회, 수정, 삭제 처리를 한다.

파이썬에서 제공되는 기본 Indexing은 하나의 원소를 읽고 처리하는 것을 말하므로 index나 key 가 존재하지 않으면 예외가 발생하므로 예외를 발생하지 않는 다양한 기법을 제공한다. Indexing을 위한 연산자는 [ ] 이다.

### 6.1.1. Sequence 타입 index 순서 이해하기

파이썬에서 Sequence 자료형은 순방향은 0번부터 시작해서 읽을 수 있고 역뱡향은 -1부터 시작해서 반대로 처리할 수 있는 index를 제공한다.

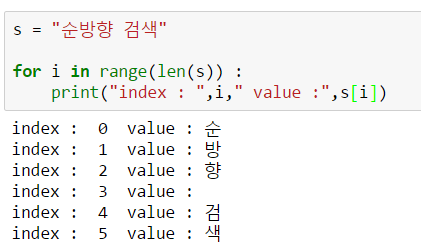
* 정수 검색(integer indexing)

Sequence 자료형은 index가 정수형으로만 구성되므로 항상 정수형 검색만 가능하다. 파이썬에서는 0 부터 양수값을 가지고 정방향으로 검색도 가능하고 이 반대인 역방향도 가능하다.

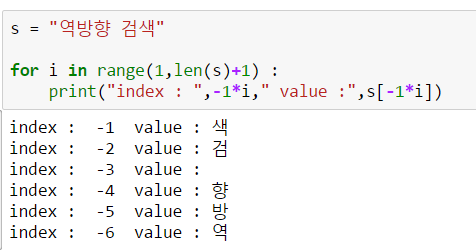
예제 6-1 : 순방향(forward)과 역방향(backward)으로 검색

정수형 index가 처리되는 범위는 0부터 시작해서 원소의 길이 -1까지이다.

이 범위에 맞는 정수를 입력해서 처리한다.

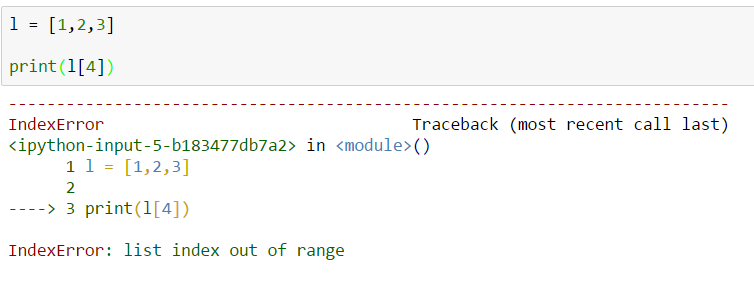


정수형 index가 처리되는 범위는 음수로 처리되면 -1부터 시작해서 원소의 개수만큼의 반대부호까지이다. 역방향도 -1 만큼씩 증가하면서 검색이 가능하다.



예제 6-2 : Index 범위가 넘어갈 경우

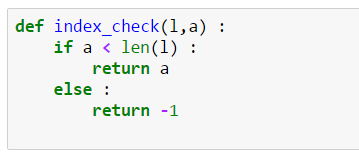
Sequence 자료형에서는 원소의 개수 즉 인덱스의 범위가 정해져 있다. 인덱스 범위가 벗어나서 검색이 발생하면 예외를 발생시킨다. 항상 범위 내에서 검색할 수 있어야 한다.



예제 6-3 : 길이보다 작은 경우까지만 인덱스가 처리

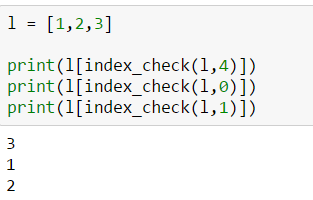
별도의 인덱스 제약을 하는 함수를 정의해서 인덱스 범위가 넘는 경우에 예외가발생하지 않도록 만들어서 사용할 수 있다.

하나의 함수를 정의해서 인덱스를 체크하고 범위가 넘을 경우 항상 -1로 맨 마지막 원소를 참조하게 정의한다



이 함수를 가지고 리스트 검색에 사용하면 인덱스 범위에 대한 에러는 발생하지 않고 범위가 넘는 것은 맨 마지막만 처리가 되는 이유를 알 수 있다.

검색할 때도 함수가 들어갈 수 있는 이유는 함수의 실행결과가 최종적으로 정수값이 만들어지므로 함수가 실행된 결과로 검색을 수행하기 때문이다.



### 6.1.2. 딕셔너리(dict) Key indexing 이해하기

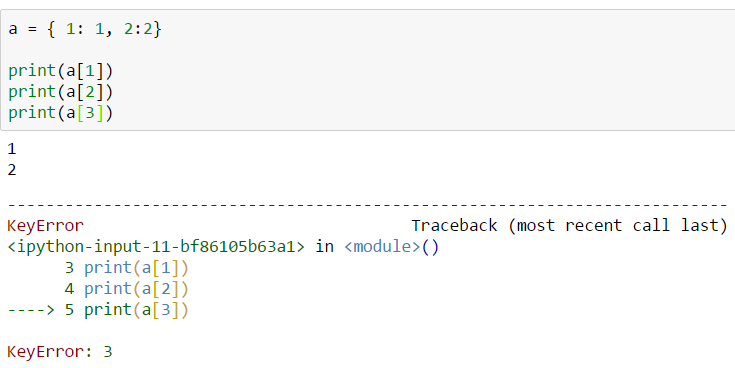
파이썬에서 Sequence와 동일한 대괄호([ ])로 검색하지만 실제 들어가는 값은 인덱스가 아닌 key 즉 hash 처리를 해서 내부 원소를 검색해서 읽을 수는 있는 구조이다. Hash는 순서가 없기에 방향성이 없다.

* 딕셔너리 원소 검색

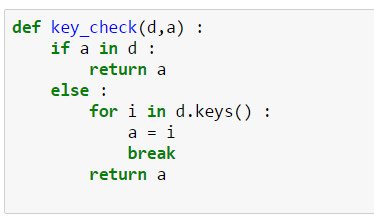
딕셔너리를 검색할 경우 해쉬가 가능한 자료형을 기준으로 키가 만들어진다. 키가 만들어진 범위 이내에서 검색할 수 있다.

예제 6-4 : 딕셔너리는 키가 정수일 경우

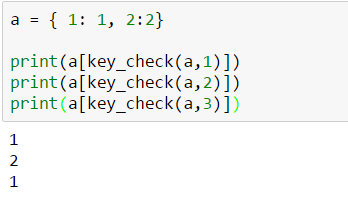
키 값이 정수로 되어 있으면 이를 이용해서 정수로 하나의 원소를 읽을 수 있다. 키 값이 없는 경우를 처리하면 index 오류가 아닌 key 오류가 발생한다.



Key 범위를 벗어난 것을 하결하기 위해 하나의 함수를 정의한다. 범위가 벗어날 경우 내부의 Key 중의 하나를 할당해서 조회가 되도록 정의한다.

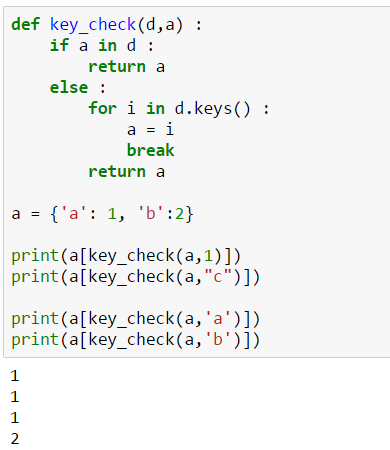


딕셔너리 검색에서 함수를 정의해서 함수가 실행된 후에 처리하면 실제 키 범위 내에서만 검색이 가능해진다.



예제 6-5 : 딕셔너리 키가 문자열일 경우

문자열로 구성된 키일 경우는 문자열을 넣어서 조회한다. 하지만 Key가 잘 못 들어오거나 범위가 벗어날 것을 대비해서 함수를 정의해서 Key 범위를 벗어나는 것을 제어하면 Key가 잘 못 들어와서 이를 처리하고 출력을 할 수 있다.



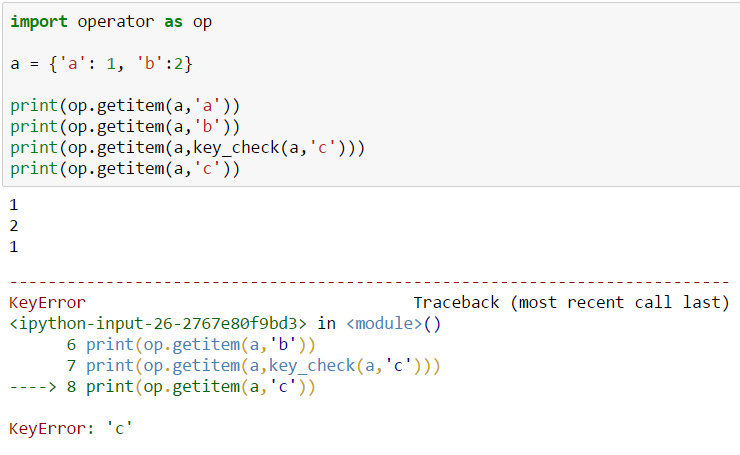
* operator 모듈의 함수 이용하기

operator 모듈 내의 getitem, itemgetter를 이용해서 key 검색과 동일하게 원소를 검색할 수 있다.

예제 6-6 : getitem 으로 인덱스 검색

[ ] 연산자는 기본 getitem을 조회하는 것과 동일하며 위의 dict 타입과 내부의 값을 인자로 지정하면 값을 직접 검색해준다.

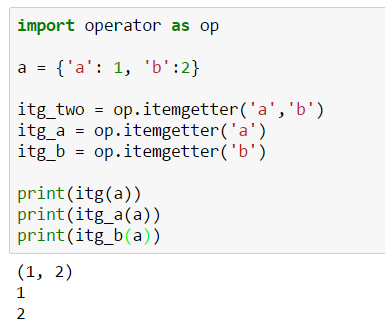
Key 범위가 벗어날 경우는 에러가 발생하므로 위의 예제에서 만들어진 key\_check 함수를 이용해서 처리하면 범위를 벗어나지 않고 처리가 된다.



예제 6-7 : itemgetter로 여러 개의 key를 동시에 검색

해당되는 키의 범위를 벗어나지 않도록 하기 위해 읽을 수 있는 키를 지정해서 처리할 수 있다.

여러 키를 넣고 동시에 읽고 싶을 경우에는 operator 모듈의itemgetter를 이용해서 여러 개의 키를 동시에 검색도 가능하고 각 키에 대해 각각 정의해서 이를 기반으로 검색을 처리하면 실제 Key의 범위를 벗어나는 것을 방지할 수 있다.

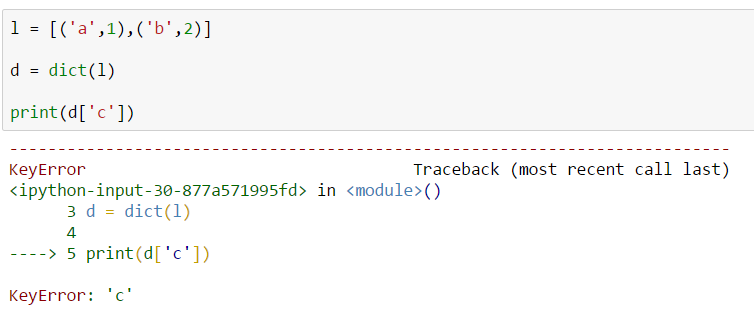


### 6.1.3. 딕셔너리(dict) Key 검색 메소드

대괄호( [ ])로 처리해서 범위가 벗어나면 예외가 발생한다. 이를 방지하기 위해 딕셔너리 자료형에 2개의 메소드를 get, setdefualt 제공한다.

예제 6-8 : 딕셔너리 내부 원소를 메소드로 검색하기

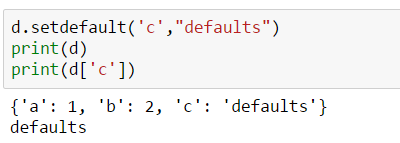
직접 접근해서 조회 시에 키 값이 존재하지 않으면 에러가 발생한다.



Key가 없는 경우 get 메소드를 이용해서 내부에 default 값을 지정할 수 있어 없으면 조회시 없으면 default값을 전달한다.



Setdefault 메소드를 사용하면 키가 없을 경우에는 추가하고 있을 경우에는 조회한다. 기존에 키가 없을 경우는 default 값을 세팅하고 조회를 처리한다.



## 6.2. sequence 자료형의 슬라이스(slicing)

Sequence 자료형의 슬라이스(slice) 처리는 Sequence 자료형들의 부분집합에 해당되는 원소의 묶음을 검색해서 처리한다는 뜻이다.

Sequence 자료형에서 indexing 처리( [ ]연산자)는 하나의 값만 들어가도록 되어있어서 슬라이싱(slicing)도 하나의 값만 들어간다. 슬라이싱으로 표현 [start: end: step] 으로 표현하지만 실제 들어가는 값은 slice 클래스의 인스턴스가 들어가므로 하나의 값만 표시되는 것이다.

### 6.2.1. 리터럴로 직접 슬라이스 처리하기

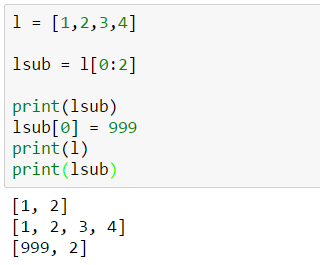
검색은 대괄호([ ])를 사용한다. 검색은 기본 하나의 값만 넣는다. 그래서 슬라이싱처리를 위해서는 파이썬 내부에서는 slice 클래스를 제공한다.

* 슬라이싱으로 부분 집합 만들기

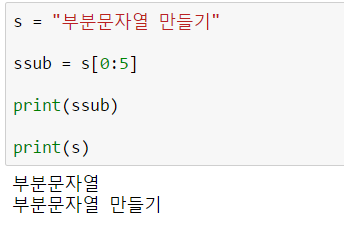
리스트, 튜플, 문자열 등 Sequence 자료형들은 부분집합을 슬라이스로 만들 수 있다.

예제 6-9 : 부분집합을 슬라이스로 생성

리스트를 만들고 슬라이싱을 처리해서 값을 할당하면 원본 리스트와 다른 부분집합의 리스트가 생성된다. 부분집합의 리스트의 원소를 갱신해도 원본 리스트는 내의 원소는 변경되지 않는다.



문자열도 Sequence 자료형이므로 슬라이스 처리로 부분 문자열을 만들 수 있다.

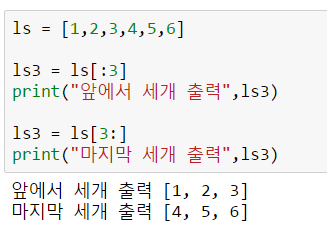


* 슬라이싱으로 순방향과 역방향 검색하기

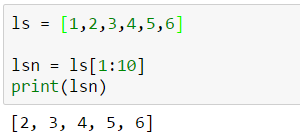
부분집합을 만들 수 있는 것은 순방향과 역방향으로 전부 처리할 수 있다. 순방향은 양수로 처리하고 역방향은 음수로 처리한다.

예제 6-10 : 순방향(forward) 인덱스

첫음부터 슬라이싱을 할 때에는 0 인덱스 없이 콜론 다음에 정수로 인덱스를 부여하면 부여된 인덱스-1까지만 읽어와서 부분집합을 만든다. 또한 중간에서부터 마지막까지 처리를 위해서는 처음에 인덱스를 부여하고 마지막에는 인덱스를 부여하지 않으면 중간 인덱스부터 마지막까지 조회해서 부분집합을 만든다.

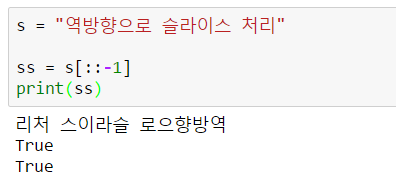


슬라이스 범위가 리스트의 인덱스 범위를 벗어나면 현재 리스트의 범위 내에서 부분 집합을 만들어 주는 것을 볼 수 있다.

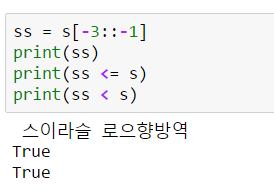


예제 6-11 : 역방향(backward) 인덱스로 리버스 처리

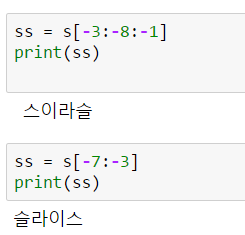
파이썬은 역방향으로 바로 접근할 수 있도록 -1 부터 역방향 index를 지원한다.



역방향으로 전체를 읽을 때는 시작과 끝은 표시를 하지 않고 step에 -1로 처리하고, 역으로 특정영역을 읽을 경우는 역으로 시작점과 종료점 그리고 step도 명기해야 한다.



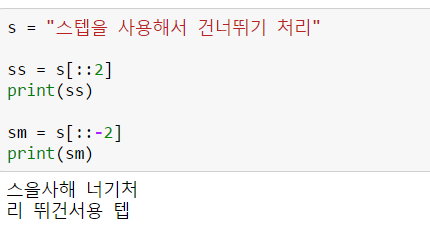
역방향이지만 반드시 step에 -1을 있어야 반대방향으로 이동할 수 있다. Step 이 없으면 음수이지만 역방향이 아니라 정방향 인덱스처럼 처리된다.



예제 6-12 : 슬라이싱(slicing)에서 step 사용하기

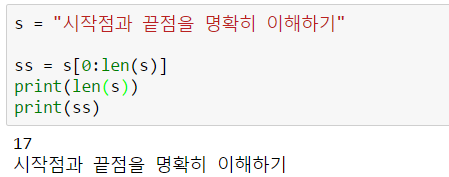
문자열 전체를 보고 스텝 즉, 원소들을 건너뛰어서 검색하는 것을 말한다. 양수를 넣으면 양의 방향으로 음수를 넣으면 음의 방향으로 원소들 사이를 건너뛰는 기능을 처리한다.

2칸씩 이동시키면서 부분 문자열을 만든다. 스텝에 양수와 음수로 2를 입력하면 양수는 정방향, 좌측으로 이동하면서 두 칸씩 가져와서 처리하고, -2를 입력하면 역방향, 우측으로 이동하면서 두 칸씩 건너뛰면서 가져와서 출력한다.

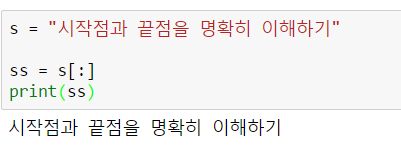


예제 6-13 : 시작과 끝을 명확히 표시해서 처리하기

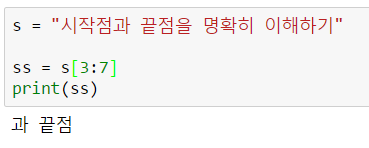
전체를 다 읽기 위해 시작점에 0을 주고 마지막 부분에는 문자열의 길이를 표시했다.



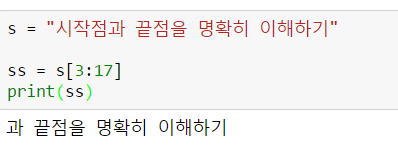
시작점과 끝점을 지정하지 않으면 전체를 읽으라는 것이므로 전체를 읽어서 표시한다.



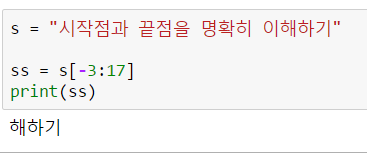
시작을 3과 끝점에 7을 주면 총 4개를 검색해서 표시하는 것을 볼 수 있다.



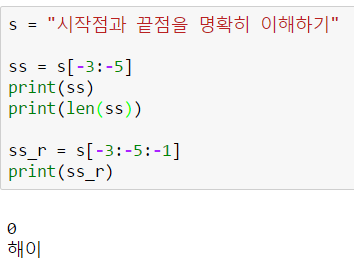
끝점이 범위가 벗어나도 실제 전체를 읽어와서 출력해 주는 것을 확인할 수 있다.



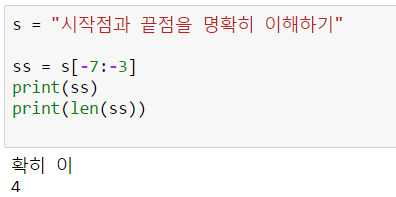
역방향 인덱스를 주고 마지막 범위를 넘는 것을 주면 실제 정방향으로 읽어서 마지막 3글자를 가져와서 처리한다.



시작점과 끝점이 역방향일 경우 시작점보다 더 작은 음수를 끝점에 넣으면 실제 검색한 것이 없어서 빈 문자열을 반환한다. 이럴 경우에는 항상 역방향으로 처리를 해야 검색해서 처리된다.



음수로 정방향을 처리하려면 더 작은 음수를 시작점에 넣고 끝점에는 더 큰 음수를 넣으면 양수의 인덱스와 동일한 처리를 한다.



### 6.2.2. 슬라이스 클래스 이용하기

파이썬에서 slice 클래스가 지원되면 이 클래스에서 인스턴스를 생성하려면 인자로 start, stop, step을 받아 생성해서 [ ] 연산자 내에 인스턴스를 넣고 처리하면 슬라이스로 검색이 되어 부분집합을 나타낸다.

* Slice도 하나의 클래스

Slice를 이용해서 하나의 인스턴스 객체를 만든다. 인자를 줄 때 하나만 표시하면 stop으로 이해한다. Start와 stop을 부여하면 시작점과 마지막점-1까지 처리하라는 표시이다.

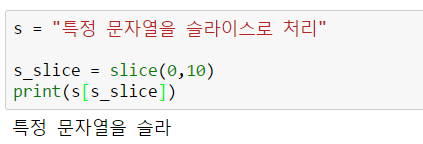
* 변수에 할당하기

Sequence 타입을 처리할 때 먼저 slice로 객체를 만들어서 위치를 고정해서 변수에 할당하고 이를 Sequence 검색 및 슬라이싱 연산자([ ]) 에 넣고 처리가 가능하다.

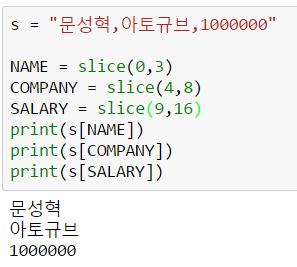
예제 6-14 : 슬라이스 클래스로 인스턴스 생성하기

슬라이스 클래스의 매개변수는 start, stop, step으로 구성되며 이것은 indexing 처리시 대괄호([start:stop:step])이 구조와 동일하게 표시된다.

시작점 0에서 종료점은 10-1까지 처리할 수 있는 슬라이스 인스턴스 생성한다.



슬라이스 인스턴스를 만들어서 변수에 할당하고(NAME, COMPANY, SALARY) 이를 검색의 값으로 넣어서 출력하면 문자열을 부분으로 짤라서 보여준다.



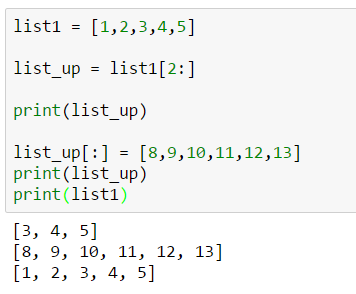
* Slice로 검색한 후에 갱신하기

slice(시작점, 종료점,스텝)을 객체로 만들면 [시작점, 종료점,스텝]가 동일한 결과를 나타내며 슬라이스 크기에 상관없이 주어진 원소를 처리하기에 슬라이스 크기보다 작은 경우에도 처리하므로 전체 원소의 개수가 줄여들 수도 있다.

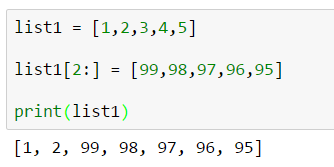
예제 6-15 : 슬라이스를 지정해서 내부 원소 갱신하기

슬라이스를 지정해서 내부 원소를 읽어온다. 이 부분집합에 슬라이스 전체에 더많은 리스트를 갱신하면 더 많은 리스트가 갱신된다.

슬라이스를 처리하면 원본과 별도의 사본을 만드는 것이므로 슬라이스 처리된 것을 갱신해도 원본에 영향을 미치지 않는다.



슬라이스 검색하고 다른 변수에 할당하지 않는 경우는 원본을 그대로 보고 있으므로 원본의 내용을 갱신한다. 실제 슬라이스의 범위보다 더 많은 원소를 가진 리스트도 원본 리스트의 원소에 다 들어가는 것을 확인할 수 있다.



## 6.3. 지능형(comprehension) 처리

Comprehesion을 한국어로 번역하면 내장, 내포 등을 사용하지만 자동으로 처리된다는 것을 보면 지능형이라는 단어가 더 마음에 들어 사용하기로 하겠다.

수학에서 comprehesion은 조건제시법을 말하므로 원소나열법을 축약해서 로직으로 처리한다는 방식을 채택해서 프로그램언어에 적용한 것이다. 지능형(comprehension)은 Sequence형(리스트)이나 maping형(dict), Set형(set)까지만 지원하고 변경불가능한 자료형은 지원하지 않는다.

파이썬 3버전부터 지능형 처리할 때 사용되는 내부 변수를 별도의 Namespace로 관리되므로 지능형 처리 이후에 지역변수는 다 사라지므로 지능형 내부로직과 동일한 변수를 사용하려면 모듈이나 함수에 별도로 변수를 지정해서 사용해야 한다.

### 6.3.1 지능형 list

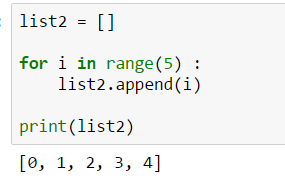
다양한 반복이 가능한 자료형을 이용해서 리스트 정의하는 [ ] 연산자 내부에 로직을 넣어서 리스트를 생성할 수 있도록 만든다.

For문 내에 반복형(iterable), 반복자(iterator)가 가능한 자료형을 넣어야하고 if 문은 순환문이 작동할 때 필요없는 원소들에 대한 필터링하는 기능을 수행한다.

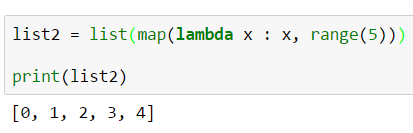
[ 표현식 for 표현식 in iterable/iterator if 표현식]

예제 6-16 : 지능형 리스트 처리

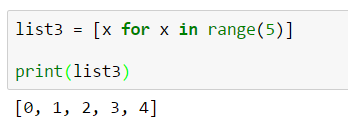
for문을 이용해서 원소를 빈 리스트에 추가해서 만들어 본다.



Map 함수를 이용해서 리스트로 생성해도 함수를 여러 개로 처리해서 사용해야 한다.

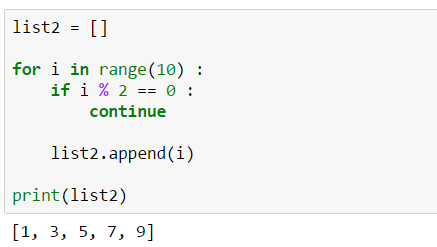


동일한 방법을 지능형 리스트를 이용해서 한 라인으로 작성하는 것이 더 편하다. 실제 이 문장이 로딩하면 리스트가 생성되어 있는 것을 알 수 있다.

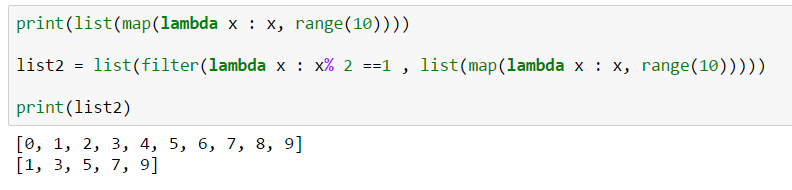


예제 6-17 : 지능형 리스트 필터 처리

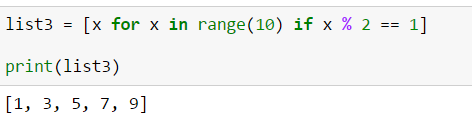
for문 이용해서 특정 부분을 제외해서 처리하려면 순환문 내에 if 제어문에 짝수를 제외하고 홀수만 리스트에 넣고 출력했다.



Map으로 리스트를 만들고 다시 filter 함수를 통해 홀수만 추출해서 리스트를 만들어서 이를 출력했다. Map과 filter 함수 내에 람다함수로 제어하는 것을 넣어서 처리했다.

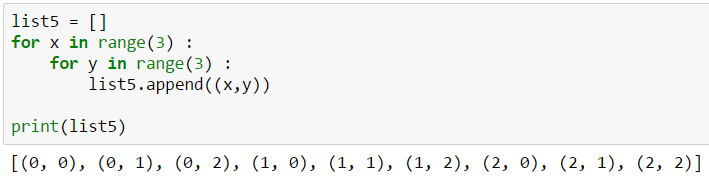


지능형 리스트는 for문 다음에 제어문을 한 라인에 작성해서 실제 처리되는 변수에서 홀수만 출력하도록 처리하면 간단히 리스트를 만든다.

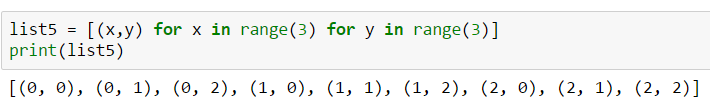


예제 6-18 : 지능형 내의 연속적인 순환문 처리

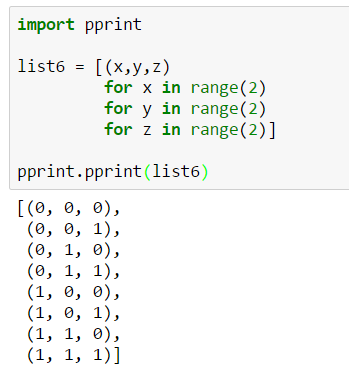
순환문을 2개를 사용해서 튜플의 원소를 들어가는 리스트를 만들어 본다.



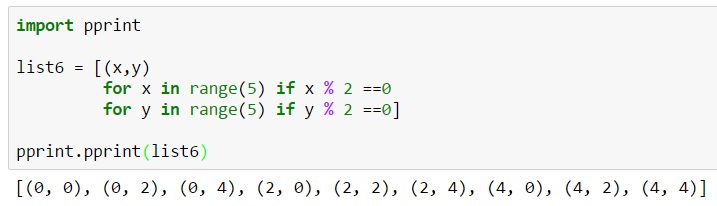
지능형 리스트를 사용할 경우는 더 단순하게 하나의 라인에 순환문을 2개 사용해서 처리가 가능하다.



세 개 이상의 for문을 하나의 라인으로 처리해서 3개의 원소를 갖는 튜플을 만들어서 처리도 가능하다.



지능형 리스트을 작성할 때 각 순환문 마다 필터링을 처리하려면 if문을 작성해서 변수의 값이 참일 경우만 실행할 수 있도록 제어를 하면 된다.



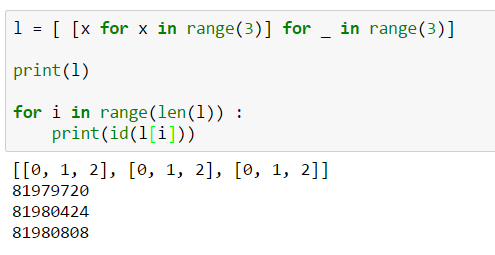
* 내포된 지능형 리스트 처리

지능형 처리도 지능형 내부에 재정의도 가능하다. 지능형으로 2차원 리스트를 만들어서 처리했다. 동일한 내부 리스트처럼 보이지만 실제 다른 리스트 객체이고 원소만 동일한 리스트들을 가지는 2차원 리스트라는 것을 확인할 수 있다.

예제 6-19 : 지능형 리스트 내의 내포된 지능형 리스트 처리

지능형 리스트 내에 지능형 리스트를 넣어서 리스트를 생성하면 2차원 리스트가만들어 진다. 내부 지능형 리스트에 대한 변수만 처리해서 리스트의 원소를 추가하기 위해 \_를 사용해서 두 번째 순환이 변수는 사용하지 않았다.

내부에 생성된 리스트의 레퍼런스를 확인하면 동일한 리스트가 생기지 않았다는 것을 알 수 있다.



### 6.3.2 지능형 dict/set

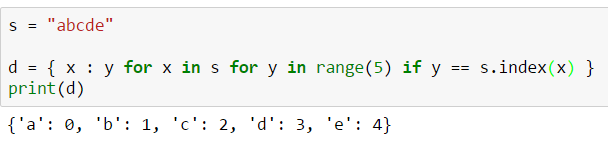
지능형 dict도 내부의 키:값 구분 하는 표현을 추가해서 반복형이나 반복자가 들어오면 처리하도록 만든다.

주의할 점은 반복형이 키와 값을 가진 구조여야 한다.

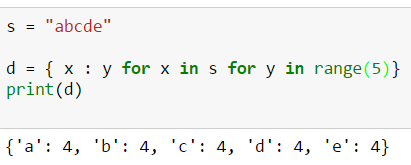
{key: value 표현식 for 표현식 in iterable/iterator if 표현식}

예제 6-20 : 지능형 dict 생성

특정 문자를 키로 구성하고 값은 인덱스와 동일하게 처리하도록 제어문에서 제한을 준다.



필터링 처리를 위해 제어문을 사용하지 않을 경우 동일한 Key에 값은 두 번째 for문이 값을 갱신하므로 최종 값인 4로 들어가있는 것을 볼 수 있다.



지능형 딕셔너리를 중복해서 만들려면 Key는 하나의 지능형이고 이 값 내에 지능형을 처리하는 것을 넣어서 처리하면 된다. 두 개의 지능형이 들어가서 처리되는 것을 확인할 수 있다.



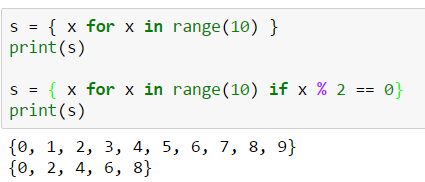
### 6.3.3. 지능형 set

지능형 set도 지능형 dict가 구조는 거의 유사하지만 차이점은 값이 없이 키만으로 구성된 구조이다.

{key 표현식 for 표현식 in iterable/iterator if 표현식}

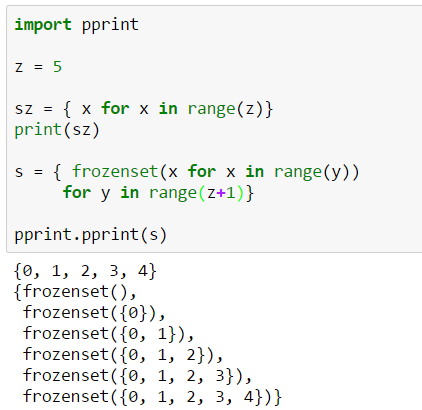
예제 6-21 : 지능형 set

표기법은 지능형 dict가 유사하지만 key만 존재하고 key를 기준으로 생성한다. 필터링은 조건이 참인 경우만 처리되므로 짝수일 경우만 만들어지도록 나머지가 0인 경우만 처리되므로 짝수의 원소만 출력된다.



지능형 set에서 지능형 set을 처리하면 hash 처리가 되지 않으므로 forzenset을 이용해서 지능형 set을 중복하게 만들 수 있다.

0부터 4까지 원소를 특정 집합에 대한 부분집합들을 만들기 위해서는 공집합부터 모든원소를 전부를 가진 집합까지를 전부 표현해야한다.

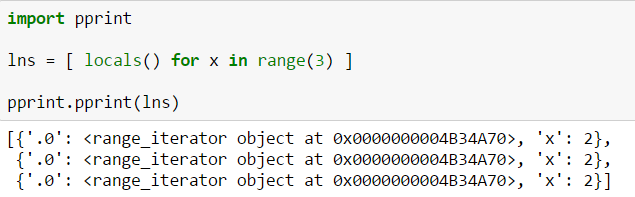


### 6.3.4. 지능형 Namespace 이해하기

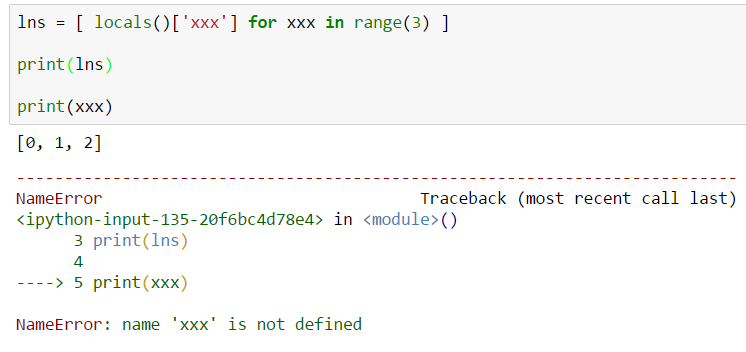
파이썬 3버전부터는 지능형 처리 내에 지정된 변수들도 지역영역으로만 처리된다. 이 말은 내부적으로 Namespace를 별도로 관리해서 처리한다는 것이다.

예제 6-22 : 지능형 리스트의 지역 네임스페이스 조회

지능형 리스트 내부에 locals 함수로 조회하면 지능형 내부의 지역영역의 Namespace가 조회된다. 지역변수 x 의 마지막 값이 처리되는 것을 알 수 있다.



지역변수에 대해 순환문이 작동할 때마다 어떤 값이 들어가 있는 지를 확인해 볼 수도 있다. 이 지역변수를 외부에서 참조하면 정의가 되어 있지 않다고 예외가 발생한다.



## 6.4. 문자열 포맷팅(string formmating) 처리

파이썬은 기본적으로 Text 기반으로 처리하므로 입력과 출력에 필요한 형식화(formmating) 체계가 존재한다.

파이썬에서 제공하는 format 함수와 문자열 자료형에 있는 format 메소드를 이용한다.

포맷팅 문자열 방식에 대해 Old % 표기를 기준으로 값과 연결하기와 New { } 표기를 기준으로 값과 연결하기 등으로 구분되어 사용된다.

### 6.4.1 포맷팅 기본 이해하기

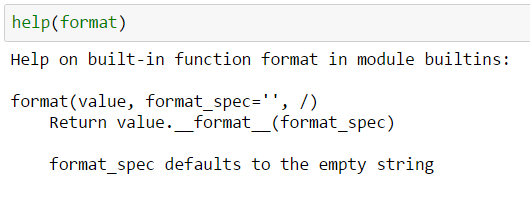
파이썬 내의 포맷을 사용해서 출력하는 문자열 처리, format 내장함수와 문자열 내의 format 메소드 등이 있다. 4가지 포매팅 처리에 대해 알아보겠다.

* 내장함수 format 구조

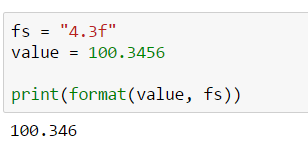
함수 help로 format 내장함수를 조회해보면 값과 format spec를 넣어서 처리하도록 되어있다. 실제 format 처리는 \_\_format\_\_ 스페셜 메소드가 실행되어 처리되는 구조이다.

예제 6-23 : 내장 format 함수 기본 알아보기

함수 format을 보면 실제 값을 넣는 value 매개변수와 실제 format이 들어가는 format\_spec을 넣어서 처리할 수 있다.



포맷 문자열을 지정하고 실수를 소수점 4자리까지 넣으면 자동으로 포매팅에 맞춰 반올림이 일어나는 것을 알 수가 있다.

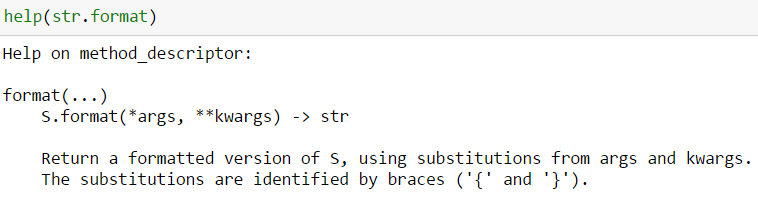


* 문자열 format 메소드

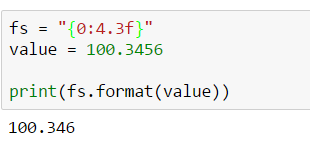
문자열 내에 포맷팅 폼을 지정하고 메소드 인자에 실제 값들이 들어와서 포맷팅 폼과 매칭해서 처리하는 메소드이다.

예제 6-24 : 문자열 format 메소드 알아보기

문자열 내의 format 메소드는 문자열에 포맷팅 정보를 처리하고 메소드 매개변수로 실제 데이터를 전달해서 처리한다.



포맷 문자열을 지정하면 이문자열을 기준으로 실제 값을 매칭해서 처리하는 방식을 사용한다. 인덱스 번호로 실제 메소드 인자와 매칭해서 값을 처리하는 것을 볼 수 있다.

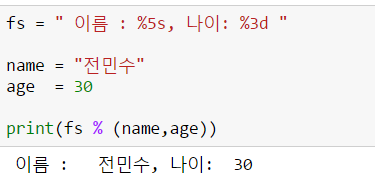


* 포맷 문자열을 사용해서 처리

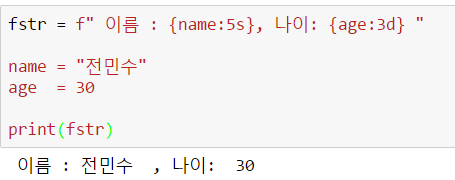
문자열에 %로 인자를 연결해서 처리하는 방식과 f 문자열은 포매팅 내부의 변수를 자동으로 매핑되어 처리하는 방식이 있다.

예제 6-25 : 문자열 포매팅 처리

문자열 내에 포매팅 코드를 차례로 넣고 인자를 % 다음에 넣으면 매핑되어 처리된다.



3.6 버전에서 추가된 f 문자열은 포맷만 정리하면 내부에 정해진 변수를 위에서 사용된 것과 매핑되어 결과를 출력한다.



### 6.4.2. 포맷코드 이해하기

문자열 포맷팅 폼을 정의할 때 인자로 받는 값들이 타입을 확인하고 적절하게 포맷팅하도록 지원하는 코드들이다.

* 주요한 포맷팅 코드

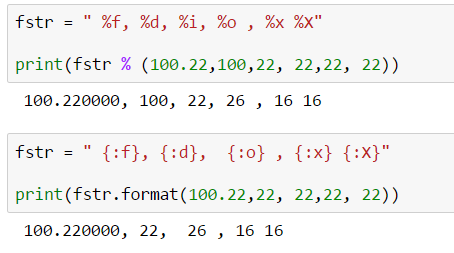
포맷팅 코드는 포맷을 정의하는 대표적인 코드로써 포맷팅 함수나 메소드에서 주어진 값들에 대한 포맷팅 기준을 표시하는 코드들이다.

|  |  |
| --- | --- |
| **코드** | **설명** |
| s | 문자열 (String) |
| d, i | 정수 (Integer), i는 old 포매팅만 지원 |
| f | 부동소수 (floating-point) |
| o | 8진수 |
| x | 16진수 |
| X | 16진수 |
| %, %% | % 표시 |
| !r, %r | \_\_repr\_\_ 처리 |
| !s. %r | \_\_str\_\_ |

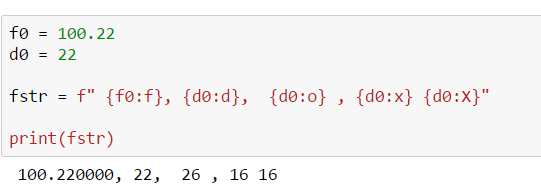
예제 6-26 : 숫자 포맷팅 코드 처리하기

위치별로 처리를 하기 위해서는 포매팅 코드를 실제 입력되는 인자에 맞춰 나열하면 된다. %로 처리하는 문자열에는 정수일 때 포맷코드가 i가 있지만 새로운 버전에서는 d로만 통합된 것을 알 수 있다.

인자 개수와 %, { }으로 표기된 개수가 동일해야 한다. 그리고 위치도 같아야 한다.

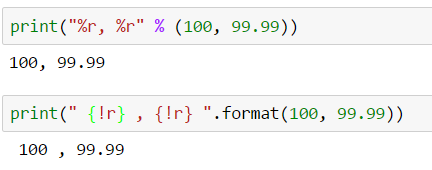


f문자열에도 숫자에 대한 코드가 작동되는 것을 확인할 수 있다.

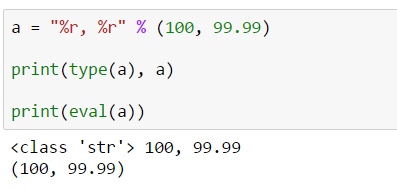


예제 6-27 : 포맷팅 코드에서 repr 함수 호출 처리

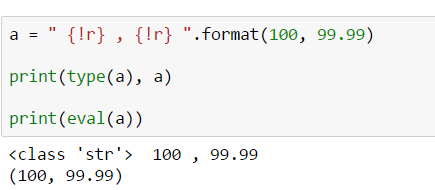
문자열 내에 포맷코드 r을 사용하면 실제 파이썬 내부 엔진에서는 \_\_repr\_\_ 메소드를 호출해서 처리된다. 실제 위치에 맞춰서만 연결이 되므로 인자의 이름을 사용해서 작성할 수 없다.



!r, %r 로 처리된 eval이나 exec등으로 실행이 가능한 구조를 만들어 준다. 이를 직접 처리해서 보면 두 개의 숫자가 쉼표로 분리된 문자열이므로 이를 eval로 평가하면 튜플이 만들어지는 것을 볼 수 있다.

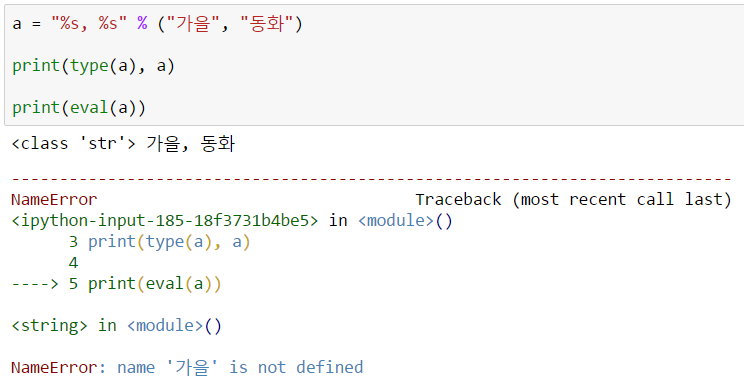


문자열 메소드로 사용해도 실제 위치인자만 지원하고 !를 사용해서 매핑하면 처처리가 된다.

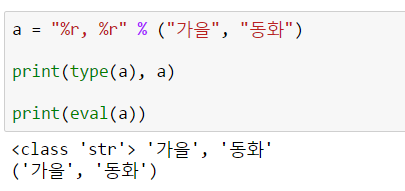


예제 6-28 : 포맷팅 코드에서 str 함수 호출 처리

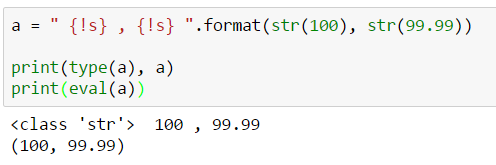
일반적인 print 문에서 사용되는 \_\_str\_\_ 메소드를 호출해서 처리한다. 문자열을 단순히 eval로 평가하면 내부적으로 변수정의로 이해해서 실제 변수가 없다는 예외가 발생한다. 문자열을 eval로 평가하면 변수로 인지되므로 이 변수가 없는 경우는 예외를 발생시킨다.



문자열 처리할 때 repr로 처리하면 실제 문자열도 eval로 평가되어 문자열이 출력되는 것을 알 수 있다. Repr로 처리된 결과는 문자열도 이중으로 인용부호가 있어 eval로 평가할 때 문자열로 출력이 되는 것을 확인할 수 있다.



숫자는 문자열로 표현된 것을 eval로 평가하면 숫자로 표시되는 것은 repr로 처리되는 것과 동일하다. 문자열일 경우만 eval로 평가하면 변수로 인식하므로 변수가 정의되지 않았다는 예외가 나는 것을 알 수 있다.

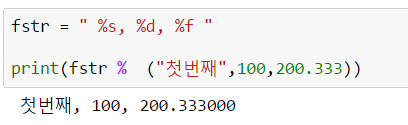


### 6.4.3. 포맷함수 및 메소드 인자 매핑

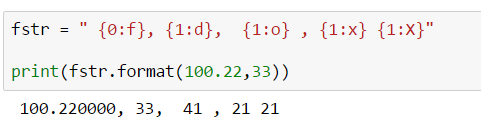
문자열에서 포맷팅 폼과 인자 사이의 연결을 하는 방법은 % , { } 두 가지 방법이 있다. 위치와 이름으로 매핑이 가능하므로 이를 상세하게 알아보겠다.

예제 6-29 : 인자와 포매팅 연결하기 : 위치인자

문자열 내에 %로 지정된 인자 개수를 지정하고 문자열 밖에 %로 표시된 다음에 인자의 개수를 넣으면 매칭되어 처리한다.

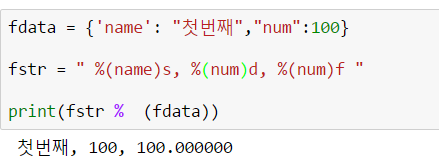


문자열 내에 중괄호{ } 를 지정해서 format 메소드 내의 인자와 매핑해 처리되도록 처리한다. 메소드 format의 인자의 위치로 포맷팅 코드 내에서 매핑하도록 강제화할 수 있다. 위치 값 변경도 가능하다.

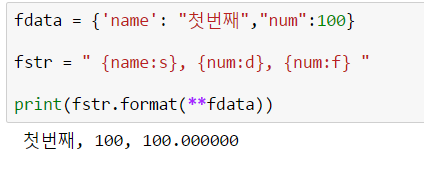


예제 6-30 : 이름으로 인자 매핑

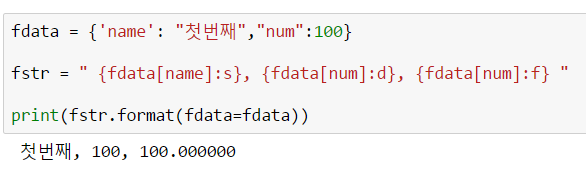
딕셔너리 자료형으로 전달 할 때 실제 포맷팅 코드 앞에 딕셔너리 내의 키를 괄호 ( )에 정의하면 이름으로 처리한다.



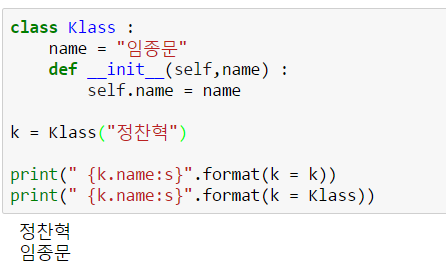
키워드 인자로 전달해서 이름을 직접접근하거나 키워드 인자의 key를 기준으로 [ ] 내에 dict 타입의 키를 넣어 검색해서 처리한다. Format 메소드에 dict 타입을 unpacking해서 키워드 인자로 넣어야 내부 키의 이름으로 매칭되어 처리된다.



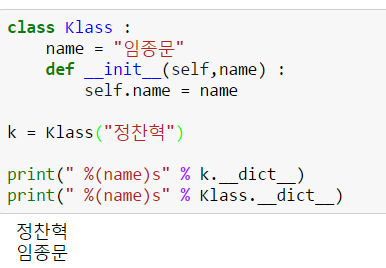
Format 메소드에 딕셔너리를 키워드인자로 제공하고 포맷팅에서 그 이름을 가지고 내부 검색을 위해 대괄호 연산자를 사용하면 실제 딕셔너리로 입력을 받은 데이터를 검색해서 출력이 가능하다.



Format 메소드 내에서 클래스와 인스턴스에 대한 정보를 전달해서 출력하고 싶을 경우에는 키워드 인자를 통해 전달하면 실제 인스턴스와 클래스의 정보를 점연산으로 검색해서 출력이 가능하다.



Old 방식일 경우 클래스와 인스턴스의 정보를 출력하고 싶을 경우에는 \_\_dict\_\_ 속성을 전달해서 그 내의 동일한 이름으로 출력을 하면 된다.

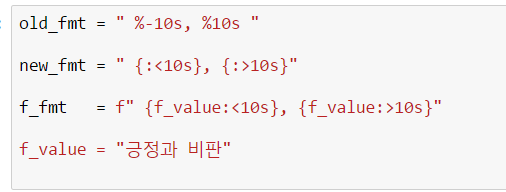


### 6.4.4. 포맷팅 꾸미기

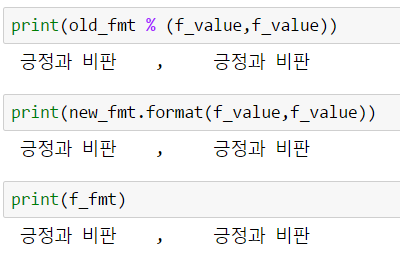
포맷팅은 실제 출력하는 것에 대해 다양하게 출력을 꾸미기 위해 사용되는 것이다. 문자와 숫자에 대해 어떻게 꾸밀 수 있는 지를 확인해 보겠다.

예제 6-31 : 출력 값들을 정렬하기

포맷팅코드 앞에 길이를 부여하고 정렬의 방향을 좌측(new 방식:>)이나 우측(old 방식: -, new 방식 : <)으로 부여하면 길이에 맞춰 포맷팅 처리가 된다. f문자열은 new 방식을 준수하지만 실제 변수명도 같이 작성해줘야 한다.



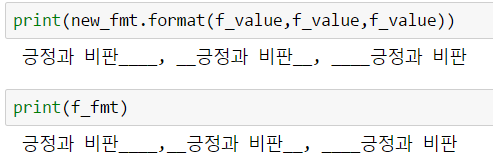
출력결과를 확인해보면 3가지의 특징 별로 문자열 꾸미기를 확인해 보면 f 문자열이 가장 변하게 작성할 수 있는 것을 알 수 있다.



정렬을 좌측, 우측, 중앙으로 처리할 수 있도록 하고 나머지 빈 공간에 임의의 문자( \_ )를 출력하는 것을 new 방식과 f문자열로 처리가 가능하다.



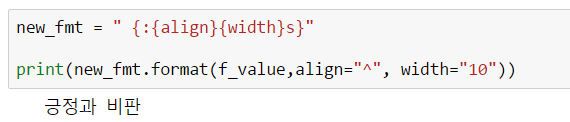
포맷팅을 한 결과를 출력해 보면 빈공칸에 언더스코어가 들어가 있는 것을 알 수 있다.



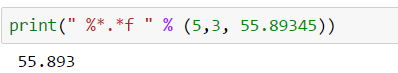
예제 6-32 : 포맷팅 형식을 매개변수로 정의하고 키워드 인자로 정의

포맷팅을 정리할 부분에 다시 중괄호를 지정해서 별도로 포맷팅을 조정해서 사용할 수 있다. 기존 포맷팅은 콜론( : ) 다음에 기호가 들어가는 부분을 정리해서 표시할 수 있다. 기호 다음에 포맷팅 기호를 변수로 받아서 처리가 가능하다. 총 10자리에 정렬을 가운데(^)하려고 한다.

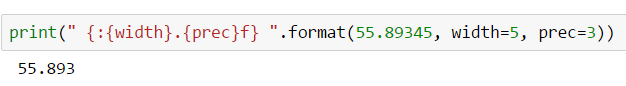
이처럼 포맷팅 기호도 매개변수와 인자로 정의해서 처리가 가능하다. .



Old 방식일 경우 특정 포매팅 코드에 세부적인 값을 할당해서 사용할 수 있다. 이때 포맷코드 앞에 별표(\*)를 부여해서 사용한다.



실수 포맷코드 f 앞에 {width}.{prec}를 부여하고 width=5, prec=2로 부여해서 처리한다.



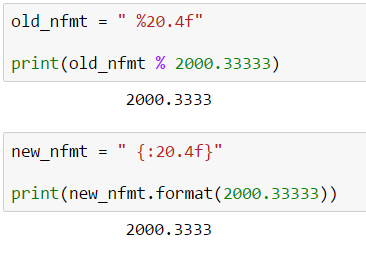
* 숫자 포맷팅 처리

숫자에 대한 정밀도 처리는 다양하게 변경할 수 있다. 정말도 부분도 별도의 인자로 주어서 처리한다면 포맷팅 폼을 유지한 상태에서 인자값만을 변경해서 다양한 포맷으로 처리가 가능하다.

예제 6-33 : 숫자 정밀도를 문자 포맷팅

문자열에 숫자 정밀도를 부여하면 소수점 이하로 숫자를 처리할 수 있다.

숫자의 정말도를 처리한다.

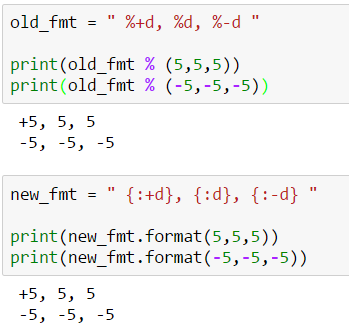


문자일 경우에도 소수점 표현을 사용할 수 있다. 두가지 처리 방식이 변경된 것을 알 수 있다. 예전 방식은 전체 중에 소수점 이하 글만 표시되고 새로운 방식은 소수점 이하 길이만 글자로 표시되는 것을 볼 수 있다.



예제 6-34 : 숫자 꾸미기

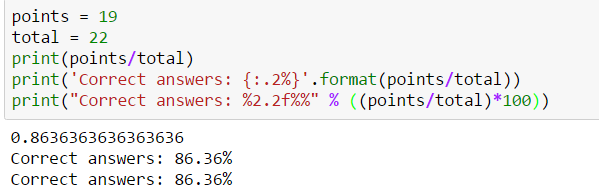
정수 포맷코드 앞에 +/- 기호로 값을 처리한다. +일 경우 양수일 때도 부호를 표시한다



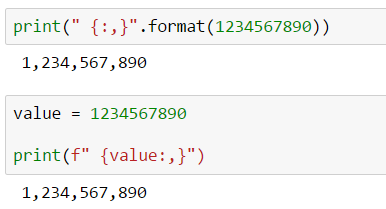
#을 넣으면 각 숫자의 추가구성항목인 16진법은 0x, 8진법은 0o, 2진법은 0b을 표시하는 기호가 추가적으로 더 붙는다.



포맷기호에 %까지 있다 나머지 문자는 문장열이므로 문자열 사용방식과 동일하고 표시하면 된다.



금액을 천 단위로 표시할 수 있도록 쉼표( , ) 기호를 제공한다.



### 6.4.5. 날짜 포매팅 처리

문자열 꾸미기에서 날짜가 들어오면 실제로 datetime에서 제공하는 날짜 포매팅 코드를 사용해서 처리가 가능하다.

* 날짜 포맷팅 코드 사용하기

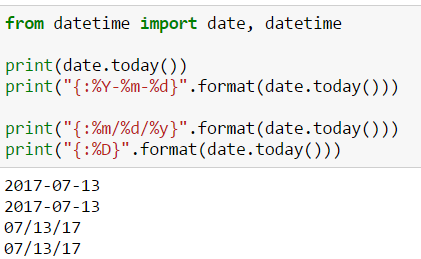
날짜 처리를 의한 포맷팅은 datatime 모듈을 참조해서 세부적으로 이해해야 한다. 간단히 날짜 처리에 대한 포맷팅 규칙도 가능하다는 것을 예시로 처리하는 것이다.

* 연 표시 : 대문자 Y는 4자리 년도 소문자 y는 두자리 년도
* 월 표시 : 소문자 m은 숫자, 소문자 b는 축약형 문자, 대문자 B는 전체 문자
* 일 표시 : d는 day 숫자
* 시간 표시 : 대문자 H는 숫자(00~23) ,대문자 I는 숫자(01~12)
* 분 표시 : 대문자 M은 숫자
* 초 표시 : 대문자 s는 숫자

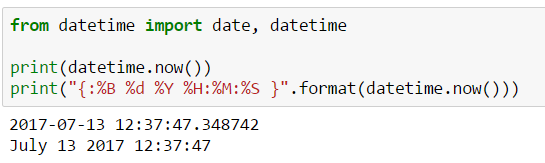
예제 6-35 : 날짜를 포맷팅 처리

년도, 월, 일, 시간과 분을 넣어서 format 메소드를 처리한다. 일단 포맷코드를 정리한 것을 문자열에 넣고 datatime으로 생성된 인스턴스를 매핑하면 날짜가 처리되어 출력된다.

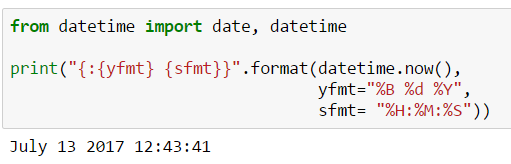
연도, 월, 일에 대한 다양한 표현도 가능하도록 표시를 해봤다.



월 표시를 영문으로 하고 시간, 분, 초를 표시하도록 표시한다.



날짜에 포맷도 인자로 전달해서 다양하게 변경이 가능하게 꾸밀 수 있다.



## 6.5. 요약

Collection 자료형인 Sequence 이나 Mapping 에 대한 원소 접근 , 원소에 대한 부분을 검출해서 처리하는 슬라이싱 등을 활용했다.

리스트, 딕셔너리, set 자료형에 대한 리터럴 생상을 쉽게 하기 위해 지능형 처리를 알아봤다.

또한 문자열로 출력을 위해 포매팅 처리를 위한 다양한 방법을 알아봤다.

.

* 정수 검색
* 슬라이싱, 슬라이스 클래스 이해
* 지능형 리스트,
* 지능형 딕셔너리
* 지능형 집합
* 문자열 포맷팅 처리
* 날짜에 대한 문자열 포맷팅 처리