4장 변수와 자료형

4.1 변수

- 자료를 넣을 수 있는 상자인 변수와 파이썬에서 지원하는 자료형(데이터 타입)

In [ ]:

```
12340 * 1/2
```

In [ ]:

```
12340 * 1/4
```

In [ ]:

```
12340 * 1/5
```

- 매번 연산할 때마다 12340을 반복적으로 썼음
- 간단한 연산에서도 숫자를 반복해 서 입력해야 할 경우 매번 같은 숫자를 입력하는 것
- 숫자 12340 을 어떤 상자에 넣고 이름을 붙인 후 이 이름을 부를 때마다 상자에 넣은 숫자가 나온다면 편리

 # 숫자와 같은 자료(data)를 넣을 수 있는 상자를 변수(variable)라 하고 상자에 붙인 이름을 변수명(혹은 변수 이름)

- 실제로 변수는 컴퓨터의 임시 저장 공간(메모리)에 저장
- 파이썬에서는 등호(=)를 이용해 변수에 자료를 넣음(할당)
- '변수명 = data' 같은 형태로 사용
- 자료가 숫자라면 data에 숫자를 쓰면 되고 문자열이라면 문자열을 쓰면 됨
- 변수명과 등호, 등호와 data 사이의 공백은 무시(즉, 공백은 있든 없든 상관없음)

In [ ]:

```
abc = 12340
print(abc)
```

- 변수에 자료를 할당한 경우 print(변수명)으로 변수명에 할당한 값을 출력할 수 있음
- print(변수명)을 이용하지 않더라도 변수명을 실행해 변수에 할당된 값을 출력할 수도 있음

In [ ]:

```
abc
```

- 숫자 12340을 쓰는 대신 변수 abc를 이용해 연산하고 출력

In [ ]:

```
print(abc * 1/2)
print(abc * 1/4)
print(abc * 1/5)
```

 # 규칙을 지키면서 변수명 작성

- 변수명을 문자, 숫자, 밑줄 기호(_)를 이용해 만듬
- 숫자로 시작하는 변수명을 만들수 없음
- 대소문자를 구분
- 공백을 포함할 수 없음
- 밑줄 이외의 기호는 변수로 시용할 수 없음
- 예약어(Reserved word)는 변수명으로 이용할 수 없음

 # 상수

- 프로그래밍 언어에서 어떤 숫자를 변수에 할당한 후에 프로그램이 끝날 때까지 그 변수의 값을 변경하지 않고 사용하는 경우
- 한 번 지정한 후 그 값이 변하지 않는 변수를 상수(constant variable)라고 함

4.2 문자열

문자열 만들기

- 문자열은 문자의 나열을 의미하는데, 파이씬에서는 따옴표로 둘러싸인 문자의 집합
- 문자열을 표시하기 위해 문자열 시작과 끝에 큰따옴표(")나 작은따옴표(')를 지정
- 둘 중 어떤 것을 사용해도 되지만 양쪽에는 같은 기호를 이용

In [ ]:

```
print("String Test")
```

In [ ]:

```
print('String Test')
```

- 문자열을 변수에 저장한 후 print() 함수로 출력

In [ ]:

```
string1 = "String Test 1"
string2 = 'String Test 2'
print(string1)
print(string2)
```

- 변수에 할당한 문자열의 타입(type)은?

In [ ]:

```
type(string1)
```

In [ ]:

```
type(string2)
```

- 출력 결과는 str
- 변수 string과 string2에 할당한 데이터의 타입이 문자열(string)인 것을 의미

- 문자열 안에 큰따옴표나 작은따옴표도 포함하려면 어떻게 해야 할까요?

In [ ]:

```
string3 = 'This is a "double" quotation test'
string4 = "This is a 'single' quotation test"
print(string3)
print(string4)
```

- 문자열에 큰따옴표를 포함하려면 문자열을 작은따옴표로 감싸고 작은따옴표를 포함하려면 큰따옴표로 감싸면 됨

- 문자열에 큰따옴표와 작은따음표를 모두 포함하고 싶거나 문장을 여러 행 넣고 싶거나 입력한 그 대로 출력하고 싶을 때는?

In [ ]:

```
long_string1 = '''[삼중 작은따옴표를 사용한 예]
파이썬에는 삼중 따옴표로 여러 행의 문자열을 입력할 수 있습니다.
큰따옴표(")와 작은따옴표(')도 입력할 수 있습니다.'''

long_string2 = """[삼중 큰따옴표를 사용한 예]
파이썬에는 삼중 따옴표로 여러 행의 문자열을 입력할 수 있습니다.
큰따옴표(")와 작은따옴표(')도 입력할 수 있습니다."""

print(long_string1)
print(long_string2)
```

- 문자열 전체를 삼중 큰따옴표 (""") 나 삼중 작은따옴표 (""")로 감싸면 됨

- 파이씬 3.x에서는 한글도 문자열에 이용할 수 있음
- 파이썬 3.x이 유니코드(Unicode)를 지원해 기본적으로 문자열이 유니코드로 처리되기 때문

문자열 다루기

문자열에는 더하기 연산자(+)와 곱하기 연산자(*)를 이용할 수 있음

- 더하기 연산자(+)는 문자열 끼리 연결(concatenation)해 문자열을 하나로 만듬
- 곱하기 연산자(*)는 곱한 만큼 문자열을 반복함

In [ ]:

```
a = 'Enjoy '
b = 'python!'
c = a + b
print(c)
```

In [ ]:

```
print(a * 3)
```

4.3 리스트

- 숫자, 문자열, 불 데이터 타입 (이런 데이터 타입은 데이터를 하나씩만 처리할 수 있음)

- 데이터를 묶어 놓으면 처리하기가 필요할때가 있음
- 예를 들면, 학교에서 학생의 과목별(국어 , 영어, 수학, 과학 등) 시험 점수를 처리한다거나 학급별로 학생의 이름 을 지정할 때 데이터를 묶어서 관리하면 편리함

- 이렇게 할 수 있는 것이 바로 리스트(List)

리스트 만들기

- 리스트는 대괄호([ ])를 이용해 만듬
- 대괄호 안에 올 수 있는 항목{혹은 요소)의 데이터 타입은 다양함
- 숫자, 문자열, 불, 리스트 등을 넣을 수 있음, 또한 튜플, 세트, 딕셔너리 도 넣을 수 있음

- 리스트를 만들 때 각 항목의 데이터 타입은 같지 않아도 됨
- 데이터는 입력한 순서대로 지정되며 항목은 콤마(,)로 구분
- 대괄호 안에 아무것도 쓰지 않으면 빈 리스트가 만들어짐
- 빈 리스트에는 데이터는 없지만 데이터 형태는 리스트임

In [ ]:

```
# 1번 학생의 국어, 영어, 수학, 과학 점수가 각각 90,95,85,80
student1 = [90,95,85,80]
student1
```

- type() 함수를 이용 해 변수 student1 의 데이터 타입을 확인

In [ ]:

```
type(student1)
```

리스트 타입의 데이터가 할당된 변수(리스트 변수)의 구조

- 리스트에서 각 항목은 , 변수명[i]로 지정 할 수 있음
- 여기서 i를 리스트 변수의 인덱스(index)라고 함
- 만약 N개의 항목이 있는 리스트 타입의 데이터가 있다면 인덱스 i의 범위는 0부터 'N - 1'까지 임
- 첫 번째 요소는 '변수명[0]',두 번째 요소는 '변수명[1],세 번째 요소는 변수명[2],
- 인덱스의 숫자가 증가하다가 마지막 요소는 '변수명[N-1]'이 됨
- 마지막 항목은 '변수명 [-1]'로도 지정할 수도 있음

In [ ]:

```
student1[0]
```

In [ ]:

```
student1[1]
```

In [ ]:

```
student1[-1]
```

리스트의 특정 항목을 변경하려면 '변수명[i] = new_data'를 이용해 리스트 변수에서 인덱스 i 항목을 new_data로 변경할 수 있음

- 리스트 변수 student1 에서 두 번째 항목을 새로운 데이터로 할당하고 리스트 변수를 출력

In [ ]:

```
student1[1] = 100 # 두 번째 항목에 새로운 데이터 할당
student1
```

리스트에 문자열을 입력

In [ ]:

```
myFriends = ['James', 'Robert', 'Lisa', 'Mary']
myFriends
```

리스트 변수 myFriends에서 세 번째, 네 번째, 마지막 항목을 지정

In [ ]:

```
myFriends[2]
```

In [ ]:

```
myFriends[3]
```

In [ ]:

```
myFriends[-1]
```

- 리스트 변수 myFriends의 네 번째 항목이 마지막 항목이므로 myFriends[3]과 myFriends[-1]은 출력 결과가 같음

숫자, 문자열 , 불 , 리스트를 혼합한 형태의 리스트

In [ ]:

```
mixedList = [0, 2, 3.14, 'python', 'program', True, myFriends]
mixedList
```

리스트 다루기

리스트 더하기와 곱하기

- 문자열과 마찬가지로 리스트도 더하기와 곱하기를 할 수 있음
- 더하기는 두 리스트를 연결하고 곱하기는 리스트를 곱한 수만큼 반복함

In [ ]:

```
list_con1= [1,2,3,4]
list_con2 = [5,6,7,8]
list_con = list_con1 + list_con2 # 리스트 연결

print(list_con)
```

곱하기

In [ ]:

```
list_con1= [1,2,3,4]
list_con = list_con1 * 3 # 리스트 반복

print(list_con)
```

리스트 중 일부 항목 가져오기

- 인덱스의 범위를 지정 해 리스트 중 일부 항목을 가져올수 있음

- 리스트 [i_start : i_end]
- 리스트 [i_start : i_end : i_step]

- i_start, i_end, i_step은 각각 인덱스의 시작, 끝 , 스텝(증가 단계)
- 인텍스의 범위를 지정하면 'i_start'에서 'i_end - 1'까지의 리스트를 반환
- i_start를 생략하면 인덱스는 0 으로 간주하고 i_end를 생략하면 인덱스는 마지막이라고 간주

In [ ]:

```
list_data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list_data)
print(list_data[0:3])
print(list_data[4:8])
print(list_data[:3])
print(list_data[7:])
print(list_data[::2])
```

리스트에서 항목 삭제하기

- 리스트에서 인덱스가 i인 항목을 삭제하려면 'del 리스트[i]'를 이용

In [ ]:

```
print(list_data)
del list_data[6]
print(list_data)
```

- 'del list_data[6]'을 실행해 인덱스 6의 위치에 있는 항목(여기서는 6)이 삭제

리스트에서 항목의 존재 여부 확인하기

- 리스트에 어떤 항목이 있는지 확인하려면 '항목 in 리스트'를 이용
- 리스트에 항목이 있으면 True를, 없으면 False를 반환

In [ ]:

```
list_data1 = [1, 2, 3, 4, 5]
print(5 in list_data1)
print(6 in list_data1)
```

- 결과에서 5는 리스트 list_data1 의 요소이므로 True를 반환했고 6은 리스트의 요소가 아니므로 False 를 반환

리스트 메서드 활용하기

- 파이썬에서는 데이터 타입(자료형)별로 이용할 수 있는 다양한 함수를 제공하는데 이를 메서드라고 함

- 메서드는 다음과 같은 형식으로 사용 할 수 있음
- 자료형.매서드이름()

- 데이터를 변수에 할당했다면 다음과 같이 변수명을 이용할 수도 있음
- 변수명.매서드이름()

- 리스트에서는 데이터의 추가 , 삽입, 삭제 등의 작업을 메서드로 수행

append()는 리스트의 맨 끝에 새로운 항목을 추가

In [ ]:

```
myFriends = ['James', 'Robert', 'Lisa', 'Mary']
print(myFriends)
myFriends.append('Thomas')
print(myFriends)
```

insert()

- insert() 메서드는 리스트의 원하는 위치에 데이터를 삽입 하는데 이용
- 자료형.insert(i,data)
- 항목의 위치를 나타내는 인덱스 i에 data가 삽입
- 이때 기존의 인덱스가 i 이상인 항목은 인덱스가 1씩 증가하면서 이동

In [ ]:

```
myFriends = ['James', 'Robert', 'Lisa', 'Mary']
print(myFriends)
myFriends.insert(1,'Paul')
print(myFriends)
```

extend()

- extend()메서드는 리스트의 맨 끝에 여러 개의 항목을 추가하는데 이용

In [ ]:

```
myFriends = ['James', 'Robert', 'Lisa', 'Mary']
print(myFriends)
myFriends.extend(['Laura', 'Betty'])
print(myFriends)
```

4.4 튜플

- 튜플(Tuple)은 리스트와 유사하게 데이터 여러 개를 하나로 묶는 데 이용
- 튜플의 항목은 숫자, 문자열, 불 , 리스트, 튜플, 세트, 딕셔너리 등으로 만들 수 있음
- 튜플의 속성은 리스트와 유사
- 튜플 데이터는 한 번 입력(혹은 생성)하면 그 이후에는 항목을 변경할 수가 없음

튜플 만들기

- 튜플은 대괄호 대신 소괄호(( ))를 사용하거나 괄호를 사용하지 않고 데이터를 입력
- 항목은 리스트와 마찬가지로 콤마(,)로 구분

In [ ]:

```
tuple1 = (1,2,3,4)
tuple1
```

- type()을 이용해 앞에서 생성한 데이터의 타입을 확인

In [ ]:

```
type(tuple1)
```

- 리스트처럼 튜플도 '변수명[i]'로 튜플의 각 요소를 지정

In [ ]:

```
tuple1[1]
```

- 소괄호를 사용하지 않고 튜플을 생성

In [ ]:

```
tuple2 = 5,6,7,8
print(tuple2)
```

In [ ]:

```
type(tuple2)
```

- 항목을 하나만 갖는 튜플을 생성할 때는 항목을 입력한 후에 반드시 콤마(,)를 입력
- 소괄호가 있거나 없거나 모두 콤마를 반드시 입력

In [ ]:

```
tuple3 = (9,) # 반드시 쉼표(',') 필요
tuple4 = 10,  # 반드시 쉼표(',') 필요
print(tuple3)
print(tuple4)
```

튜플 다루기

- 한번 생성된 튜플은 요소를 변경하거나 삭제할 수 없음
- 튜플의 이런 특성으로 튜플은 한 번 생성한 후에 요소를 변경할 필요가 없거나 변경할 수 없도록 하고 싶을 때 주로 이용

In [ ]:

```
tuple5 = (1,2,3,4)
tuple5[1] = 5 # 한번 생성된 튜플의 요소는 변경되지 않음
```

In [ ]:

```
del tuple5[1] # 한번 생성된 튜플 요소는 삭제되지 않음
```

- 튜플의 요소를 변경하거나 삭제하는 메서드는 튜플에서 이용할 수 없지만
- index() 메서드나 count() 메서드처럼 요소를 변경하지 않는 메서드는 튜플에서도 사용할 수 있음

- index() 메서드는 인자와 일치하는 첫 번째 항목의 위치를 반환하고
- count() 메서드는 인자와 일치하는 항목의 개수를 반환

In [ ]:

```
tuple6 = ('a', 'b', 'c', 'd', 'e', 'f')
tuple6.index('c')
```

- index() 메서드의 인자 c 와 일치하는 튜플의 위치는 세번째이므로 2를 반환했음

- 특정 인자와 일치하는 항목의 개수를 알고 싶을 때는 count() 메서드를 이용

In [ ]:

```
tuple7 = ('a', 'a', 'a', 'b', 'b', 'c', 'd')
tuple7.count('a')
```

- count() 메서드의 인자 'a'와 일치하는 튜플 항목의 개수는 세개이므로 3을 반환

4.5 세트

- 세트는 수학의 집합 개념을 파이썬에서 이용할 수 있도록 만든 데이터 타입
- 세트가 리스트와 튜플과 다른 점은 데이터의 순서가 없고 데이터를 중복해서 쓸 수 없다는 것
- 세트는 리스트에서 사용했던 메서드 외에 집합의 기본이 되는 교집합 , 합집합, 차집합을 구하는 메서드를 사용할 수 있음

세트 만들기

- 세트를 생성할 때는 중괄호({})로 데이터를 감싸면 됨
- 항목과 항목 사이에는 리스트나 튜플과 마찬가지로 콤마(,)가 들어감

In [ ]:

```
set1 = {1, 2, 3}
set1a = {1, 2, 3, 3}
print(set1)
print(set1a)
```

- 두개의 변수(set1, set1a)에 각각 세트를 생성해 대입

- 변수 set1a에는 3을 중복해서 대입했지만 출력된 결과를 보면 set1a에는 중복된 데이터는 하나만 입력된 것을 볼 수 있음

- 세트의 데이터 타입을 확인하기 위해서는 type() 함수를 이용

In [ ]:

```
type(set1)
```

세트의 교집합, 합집합, 차집합 구하기

- 세트에서 사용할 수 있는 교집합, 합집합, 차집합 메서드를 이용

- 교집합(intersection): 두 집합 A와 B가 았을 때 집합 A에도 속하고 집합 B에도 속하는 원소로 이뤄진 집합.
- 합집합(union): 두 집합 A와 때 있을 때 집합 A에 속하거나 집합 답에 속하는 원소로 이뤄진 집합.
- 차집합(difference): 두 집합 A와 B가 있을 때 집합 A에는 속하고 집합 B에는 속하지 않는 원소로 이뤄진 집합

- 교집합 , 합집합 , 차집합을 위한 메서드는 각각 intersection(), union(), difference()

In [ ]:

```
A = {1, 2, 3, 4, 5}         # Set A
B = {4, 5, 6, 7, 8, 9, 10}  # Set B
A.intersection(B)  # 집합 A에 대한 집합 B의 교집합(A∩B)
```

In [ ]:

```
A.union(B) # 집합 A에 대한 집합 B의 합집합(A∪B)
```

In [ ]:

```
A.difference(B) # 집합 A에 대한 집합 B의 차집합(A-B)
```

- 집합의 교집합, 합집합, 차집합을 구하기 위해 메서드를 이용했지만 연산자를 이용할 수도 있음
- 세트가 두 개 있다고 할 때 두 집합의 교집합 , 합집합, 차집합을 위한 세트 연산자는 각각 '&', '|', '-'임

In [ ]:

```
A = {1, 2, 3, 4, 5}         # Set A
B = {4, 5, 6, 7, 8, 9, 10}  # Set B
A & B  # 집합 A에 대한 집합 B의 교집합(A∩B)
```

In [ ]:

```
A | B  # 집합 A에 대한 집합 B의 합집합(A∪B)
```

In [ ]:

```
A - B  # 집합 A에 대한 집합 B의 차집합(A-B)
```

리스트, 튜플, 세트 간 타입 변환

- 여러 데이터를 다루다 보면 연산이나 처리를 할 때 데이터의 타입을 변환해야 할 필요
- 데이터 타입은 list(), tuple(), set()을 이용해 서로 변환

In [ ]:

```
a = [1,2,3,4,5]
```

In [ ]:

```
type(a)
```

- 리스트를 튜플로 변환

In [ ]:

```
b = tuple(a)
b
```

In [ ]:

```
type(b)
```

- 리스트 a를 세트로 변환

In [ ]:

```
c = set(a)
c
```

In [ ]:

```
type(c)
```

- 튜플과 세트로 변환된 데이터를 다시 리스트로 변환

In [ ]:

```
list(b)
```

In [ ]:

```
list(c)
```

4.6 딕셔너리

- 파이썬의 딕셔너리도 사전과 유사하게 구성
- 사전의 표제어와 설명은 파이썬에서 각각 키(key)와 값(value)
- 딕셔너리는 키와 값이 항상 쌍으로 구성
- 리스트나 튜플은 인덱스를 이용해 항목을 다뤘지만 딕셔너리는 인덱스 대신 키를 이용해 값을 다룸

- 리스트나 튜플에서 인덱스는 0부터 시작하는 숫자였지만 딕셔너리의 키는 임의로 지정한 숫자나 문자열이 될 수 있음
- 값으로는 어떤 데이터 타입도 사용할 수 있음

딕셔너리 만들기

- 딕셔너리를 만들려면 딕셔너리 데이터 전체를 중괄호({})로 감싸면 됨
- 키와 값의 구분은 콜론(:)으로 함
- 키와 값으로 이뤄진 각 쌍은 콤마(,)로 구분

In [ ]:

```
country_capital = {"영국":"런던", "프랑스":"파리", "스위스": "베른", "호주":"멜버른", "덴마크": "
country_capital
```

- 코드와 출력된 결과를 비교해 보면 딕셔너리의 경우 입력한 순서대로 출력되지 않는 것을 알 수 있음
- 딕셔너리는 순서 보다 키와 값의 쌍으로 데이터를 입력해야할 때 주로 이용

- 생성한 변수 country_capital의 타입

In [ ]:

```
type(country_capital)
```

- 지정한 키의 값만 얻고 싶으면 딕셔너리 변수에 원하는 키를 넣으면 됨

In [ ]:

```
country_capital["영국"]
```

- 딕셔너리의 키는 숫자와 문자열이 될 수 있고 값은 어떤 데이터 형태도 올 수 있음
- 우선 키는 숫자로, 값은 문자열로 입력

In [ ]:

```
dict_data1 = {1:"버스", 3: "비행기", 4:"택시", 5: "자전거"}
dict_data1
```

- 딕셔너리에서 키를 3으로 지정해 출력

In [ ]:

```
dict_data1[3]
```

- dict_data1[3]에서 입력된 3은 딕셔너리 키이지, 리스트의 인덱스처럼 위치를 뜻하는 3이 아님

- 키와 값이 모두 숫자인 딕셔너리

In [ ]:

```
dict_data2 = {1:10, 2: 20, 3:30, 4: 40, 5:50}
print(dict_data2)
print(dict_data2[4])
```

- 키가 문자열이고 값은 리스트인 딕셔너리

In [ ]:

```
dict_data3 = {"list_data1":[11,12,13], "list_data2": [21,22,23]}
print(dict_data3)
print(dict_data3["list_data2"])
```

- 다양한 형태의 키와 값을 갖는 딕셔너리

In [ ]:

```
mixed_dict = {1:10, 'dict_num': {1:10, 2:20}, "dict_list_tuple": {"A":[11,12,13], "B":(21,22,23)}
mixed_dict
```

- 키에 사용할 수 있는 데이터 타입은 숫자나 문자열이지만 값에는 숫자, 문자열 , 튜플, 리스트, 딕셔너리 등 다양한 데이터를 사용할 수 있음

딕셔너리 다루기

딕셔너리에 데이터 추가하고 변경하기

- 생성된 딕셔너리에 새로운 키와 값을 추가하거나 기존의 값을수정하려면 'dict_variable[key] = value' 형태로 입력

In [ ]:

```
country_capital["독일"]= "베를린"
country_capital
```

- 기존 키의 값을 다른 값으로 변경하려면 기존 키를 지정하고 새로운 값을 할당

In [ ]:

```
country_capital["호주"]= "캔버라"
country_capital
```

딕셔너리에서 데이터 삭제하기

- 딕셔너리의 특정 키와 값을 삭제하려면 'del 딕셔너리데이터[key]'를 입력해 딕셔너리에서 키와 이와 쌍을 이루는 값을 삭제할 수 있음

In [ ]:

```
del country_capital["덴마크"]
country_capital
```

딕셔너리 메서드 활용하기

- 딕셔너리의 경우도 딕셔너리 데이터를 처리하기 위한딕셔너리 메서드
- 딕셔너리 메서드는 '딕셔너리데이터.메서드이름()'과 같은 형태로 사용할 수 있음

- 과일 이름을 키로 하고 숫자를 값으로 하는 딕셔너리를 생성

In [ ]:

```
fruit_code = {"사과":101, "배":102, "딸기":103, "포도":104, "바나나":105}
```

- 딕셔너리 메서드 keys()를 이용해 딕셔너리에서 키만출력

In [ ]:

```
print(fruit_code.keys())
```

- values()로 딕셔너리의 값을 출력

In [ ]:

```
print(fruit_code.values())
```

- items()로 키와 값의 쌍을 출력

In [ ]:

```
print(fruit_code.items())
```

- 딕셔너리 메서드인 keys(), values(), items()는 각각 dict_keys, dict_values, dict_iterns 라는 데이터 형태로 값을 반환
- 딕셔너리는 리스트로 변환해서 데이터를 처리하면 편리한 경우가 있음

- list() 함수를 이용해 리스트로 변환

In [ ]:

```
list(fruit_code.keys())
```

In [ ]:

```
list(fruit_code.values())
```

In [ ]:

```
list(fruit_code.items())
```

- update()를 이용해 기존의 딕셔너리 데이터에 새로운 딕셔너리 데이터를 추가

In [ ]:

```
fruit_code2 = {"오렌지":106, "수박":107}
```

- update() 메서드를 이용해 딕셔너리 fruit_code에 fruit_code2를 추가

In [ ]:

```
fruit_code.update(fruit_code2)
fruit_code
```

- clear() 메서드는 딕셔너리의 모든 항목을 삭제

In [ ]:

```
fruit_code2.clear()
print(fruit_code2)
type(fruit_code2)
```

- 딕셔너리 fruit_code2의 모든 항목을 clear()메서드로 삭제해 빈 딕셔너리({})가 됐지만 데이터 타입은 여전히 딕셔너리(dict)임을 알수 있음

4.7 정리

- 데이터 타입은 문자열, 리스트, 튜플, 세트, 딕셔너리