03. 수치

3.1 수치자료형

수치 자료형에는 정수형과 실수형(부동 소수점형, Floating Point), 복소수형이 있다. 각각에 대해서 자세히 알아보도록 하자.

- 정수형 상수

정수형 상수는 10진수와 2진수, 8진수, 16진수 상수가 있다. 같은 정수형이지만 다양하게 표현하는 방법이 있다. 컴퓨터는 내부에서 2진수로 수치를 표현하므로 2진수와 관련된 8진수나 16진수도 종종 사용한다. 정수형 상수의 표현 범위에 제한은 없다. 하지만, CPU 레지스터로 표현할수 있는 크기보다 큰 정수를 다루어야 한다면 연산 속도는 상당히 느려질수 있다.

```
>>> a = 23              # 10진수 상수이다.
>>> type(a)             # 자료형을 확인한다.
<class 'int'>
>>> isinstance(a, int)          # a가 정수형인지 확인한다.
True
>>> b = 0o23            # 8진수 상수이다. 0o로 시작하면 8진수이다.
>>> c = 0x23            # 16진수 상수이다. 0x이나 0X로 시작하면 16진수이다.
>>> d = 0b1101          # 2진수 상수이다. 0b로 시작하면 2진수이다.
>>> print(a, b, c, d)           # 10진수로 출력한다.
23 19 35 13
>>> 2 ** 1024           # 정수형의 표현 범위에 제한은 없다.
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021
120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952
085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137
216
>>> n = 2 ** 1024       # 필요한 비트 수를 얻을수 있다.
```

만일 다른 진법의 수(문자열)를 10진수로 변환하려면 int() 함수를 사용한다. 5진수 123을 10진수로 변환하는 예이다.

```
>>> int('123', 5) )
38
```

10진수 데이터를 2진수와 8진수, 16진수로 변환하는 함수는 다음과 같다.

```
>>> bin(23)     # 2진수로 변환한다.
'0b10111'
>>> oct(23)     # 8진수로 변환한다.
'0o27'
>>> hex(23)     # 16진수로 변환한다.
'0x17'
```

다른 자료형으로부터 정수를 얻으려면 int() 함수를 사용한다.

```
>>> int(2.9)    # 소수점 이하는 버려진다.
2
>>> int(-2.9)
-2
>>> int('123')  # 문자열을 정수로 변환한다.
123
>>> int('123.45')       # 정수로 변환이 안된다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.45'
>>> int(float('123.45'))        # 문자열 -> 실수형 -> 정수형
123
>>> int(2 + 3j)         # 복소수도 직접 정수로 변환이 안된다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to int
```

다음은 정수형 데이터를 다른 수치 자료형으로 변환하는 예이다.

```
>>> a = 123
>>> float(a)    # 실수형으로 변환한다.
123.0
>>> str(a)      # 문자열로 변환한다.
'123'
>>> complex(a)  # 복소수형으로 변환한다.
(123+0j)
```

정수형 데이터를 직접 바이트 열로 변환할수 있다.

```
>>> (1024).to_bytes(2, byteorder = 'big')       # 2바이트 빅 엔디안으로 변환한다.
b'₩x04₩x00'
>>> (1024).to_bytes(2, byteorder = 'little')    # 2바이트 리틀 엔디안으로 변환한다.
b'₩x00₩x04'
>>> (-1024).to_bytes(4, byteorder = 'big', signed = True) )
b'₩xff₩xff₩xfc₩x00'
```

바이트 열에서 정수형으로 변환하는 예는 다음과 같다.

```
>>> int.from_bytes(b'₩x04₩x00',byteorder = 'big')
1024
>>> int.from_bytes(b'₩x00₩x04',byteorder = 'little')
1024
>>> int.from_bytes(b'₩xff₩xff₩xfc₩x00',byteorder = 'big')
4294966272
```

```
>>> int.from_bytes(b'₩xff₩xff₩xfc₩x00', byteorder = 'big', signed = True)
-1024
```

리스트와 같이 시퀀스형을 이용하는 것도 가능하다.

```
>>> int.from_bytes([4, 0], byteorder = 'big') )
1024
```

- 실수형 상수

실수형 상수는 소수점을 포함하거나 e나 E로 지수를 포함한다. 컴퓨터에서는 실수를 부동소수점 방식으로 표현하기 때문에 부동 소수점형이라고 부른다. 실수형이라고 해도 큰 문제는 없다.

```
>>> a = 1.2     # 소수점을 포함하면 실수이다.
>>> type(a)     # 자료형을 확인한다.
<class 'float'>
>>> isinstance(a, float)        # a가 실수형인지 확인한다.
True
>>> b = 3e3     # 지수(e)를 포함해도 실수이다. 3곱하기 10의 3승과 같다.
>>> c = -0.2e-4         # 지수부는 정수일수 있으나 실수일수는 없다.
>>> print(a, b, c)
1.2 3000.0 -2e-05
```

실수형 상수는 C나 Java 언어에서의 double형과 동일하며 8바이트(64비트)로 표현한다. 수치 표현 범위는 유효자리 15이며, 2.2250738585072014e-308 ~ 1.7976931348623157e+308 범위에서 표현한다. 자세한 정보는 sys 모듈의 float_info를 통해서 알수 있다.

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
>>> sys.float_info.max
1.7976931348623157e+308
>>> sys.float_info.min
2.2250738585072014e-308
```

무한대의 수는 다음과 같이 표현한다.

```
>>> float('inf')
inf
>>> float('-inf')
-inf
>>> infinity = float('inf')
>>> infinity / 1000
inf
```

실수형인 경우네 정수로 오차없이 표현할수 있는 값인지를 메서드 is_integer()를 사용하여 알수 있다.

```
>>> a = 1.2
>>> a.is_integer()
False
>>> a = 2.0
>>> a.is_integer()
True
```

실수를 정수로 변환하는 또 다른 방법은 반올림과 올림, 내림을 사용하는 것이다.

```
>>> round(1.2)          # 반올림
1
>>> round(1.8)          # 반올림
2
>>> import math
>>> math.ceil(1.2)      # 1.2보다 같거나 큰 정수
2
>>> math.floor(1.0)     # 1.9보다 같거나 작은 정수
1
```

실수 연산에서 발생하는 반올림 에러를 살펴보자. 다음의 코드는 0.1을 1000번 더하여 100을 만드는 코드이다. 출력 결과가 얼마가 나오는지 보자. 결과가 100이 아닌 이유는 진법 변환 에러 때문이다. 컴퓨터에서 실수 연산은 특수한 경우를 제외하고는 항상 어느 정도의 오차를 포함하고 있다고 보아야 한다.

```
>>> e = 0.0
>>> for k in range(1000):
...     e += 0.1
...
>>> e
99.9999999999986
```

- 복소수형 상수

복소수형 상수는 실수부와 허수부로 표현한다. 허수부에는 j나 J를 숫자 뒤에 붙여야 한다. 실수부와 허수부 각각은 실수형 상수로 표현한다.

```
>>> c = 4 + 5j
>>> o = d = 7 - 2J
>>> print(c * d)
(38+27j)
>>> c.real      # 복소수의 실수부만 취한다.
4.0
```

```
>>> c.imag      # 복소수의 허수부만 취한다.
5.0
>>> a = 3       # 일반 실수나 정수로부터 complex() 함수를 사용하여 복소수를 만들 수 있다.
>>> b = 4
>>> complex(a, b)       # a가 실수부, b가 허수부가 된다.
(3+4j)
>>> c.conjugate()       # 켤레 복소수
(4-5j)
```

cmath 모듈을 사용하면 다양한 함수에서 복소수 연산을 할수 있다.

```
>>> import cmath
>>> cmath.sin(0.1 + 0.2j)
(0.10183674942129743+0.20033016114881572j)
>>> cmath.sqrt(-2)
1.4142135623730951j
>>> cmath.log(2j)
(0.6931471805599453+1.5707963267948966j)
```

다음은 오일러 공식을 적용한 예이다.

```
>>> from math import e, pi, cos, sin
>>> e ** (pi / 4 * 1J)
(0.7071067811865476+0.7071067811865476j)
>>> complex(cos(pi / 4), sin(pi / 4))
(0.7071067811865476+0.7071067811865476j)
```

- fractions 모듈을 사용해서 분수를 표현하기
fractions 모듈을 사용하면 분수 연산을 할수 있다.

```
>>> from fractions import Fraction
>>> Fraction('5/7')            # 5/7
Fraction(5, 7)
>>> Fraction(5, 7)
Fraction(5, 7)
>>> Fraction(123)              # 123/1
Fraction(123, 1)
>>> Fraction('1.414213')
Fraction(1414213, 1000000)
>>> Fraction(2476979795053773, 225179981368248)
Fraction(825659931684591, 75059993789416)
>>> Fraction(5, 7) + Fraction(2, 5)   # 5/7 + 2/5
```

```
Fraction(39, 35)
>>> float(Fraction(39, 35)
1.1142857142857143
```

- decimal 모듈을 사용해서 고정 소수점을 표현하기
앞서 진법 변환 에러에서 컴퓨터의 부동 소수점에 의한 실수 표현은 어쩔수 없이 오차를 동반한다고 했다.

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
```

오차가 없는 정확한 결과가 필요하면 decimal 모듈을 사용한다. 이 모듈은 두 개의 클래스 Decimal과 Context를 제공한다. Decimal 클래스는 숫자를 표현하고, Context 클래스는 정확도나 반올림 방법과 같은 환경을 지정한다.

I. Decimal 객체
숫자를 표현하는 Decimal 객체를 생성하는 예는 다음과 같다.

```
>>> from decimal import *
>>> Decimal(1234)       # 정수로부터 Decimal 객체를 생성할수 있다
Decimal('1234')
>>> Decimal('1234')     # 문자열로부터도 Decimal 객체를 생성할수 있다.
Decimal('1234')
>>> Decimal(1.1)        # 부동 소수점으로부터 가능하다. 오차를 고려해야 한다.
Decimal('1.100000000000000088817841970012523233890533447265625')
>>> Decimal('1.1')      # 문자열로부터 생성하면 정확하게 표현할수 있다.
Decimal('1.1')
>>> Decimal('Infinity') # +무한대
Decimal('Infinity')
>>> Decimal('-Infinity')        # -무한대
Decimal('-Infinity')
>>> Decimal("Nan")      # Not a Number
Decimal('NaN')
```

다음은 Decimal 객체를 사용하여 0.1을 1000번 더해서 100을 구하는 예이다.

```
>>> e = Decimal('0.0') )
>>> delta = Decimal('0.1') )
>>> for k in range(1000):
...     e += delta
...
>>> e
Decimal('100.0')
```

```
>>> print(e)
100.0
```

계산 결과가 정확하게 나왔다. 꼭 필요할때만 사용해야 한다. 계산이 상대적으로 오래 걸린다. Decimal 객체 간에는 일반 수치형처럼 연산할수 있다.

```
>>> a = Decimal('35.72') )
>>> b = Decimal('1.73') )
>>> a + b
Decimal('37.45')
>>> a - b
Decimal('33.99')
>>> a * b
Decimal('61.7956')
>>> a // b
Decimal('20')
>>> a % b
Decimal('1.12')
>>> a ** b
Decimal('485.88871096498864516866004 98')
```

또한, Decimal 객체와 정수형과의 연산은 가능하지만 부동 소수점(실수)형과의 연산은 안된다.

```
>>> a = Decimal('35.72') )
>>> a + 2       # 연산이 가능하다
Decimal('37.72')
>>> a + 3.2     # 연산이 안된다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'decimal.Decimal' and 'float'
```

모듈 math와 cmath와 함께 사용할수도 있다. 계산결과는 부동 소수점(실수)형이다.

```
>>> import math, cmath
>>> d= Decimal('123.456789012345') )
>>> math.sqrt(d)
11.11111106111108
>>> cmath.sqrt(-d)
11.11111106111108j
```

Decimal 객체가 직접 지원하는 연산도 있다. 유효 자릿수 조정이 자유롭다.

```
>>> getcontext().prec = 38      # 유효 자리를 38자리로 조정한다.
>>> Decimal('2').sqrt()         # 루트 2
```

```
Decimal('1.41421356237309504880168872420969807856')
>>> getcontext().prec = 28      # 유효 자리를 28자리로 조정한다.
>>> Decimal('2').sqrt() )
Decimal('1.4142135623730950488016887242')
>>> Decimal(2).exp()            # e² 연산
Decimal('7.389056098930650227230427461')
>>> Decimal('10').ln()  # log(10)       )
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()       # log10 10
Decimal('1')
```

quantize() 메서드로 원하는 소수점 자리의 수치를 얻을수 있다. 다음예에서 첫인수 Decimal('.01')은 자릿수를 결정한다. 값은 별 의미가 없고 자릿수만 중요하다. 즉, Decimal('1.12')로도 동일한 결과를 얻는다.

```
>>> # 소수점 이하 두 자리까지만 취한다. 나머지는 버린다.
...
>>> Decimal('7.329').quantize(Decimal('.01'), rounding = ROUND_DOWN) )))
Decimal('7.32')
>>> # 소수점 이하 두자리까지 무조건 올림하여 취한다.
...
>>> Decimal('7.321').quantize(Decimal('1.12'), rounding = ROUND_UP) )))
Decimal('7.33')
>>> # 소수점 이하 두 자리까지 반올림하여 취한다.
...
>>> Decimal('7.325').quantize(Decimal('.01'), rounding  = ROUND_HALF_UP) )))
Decimal('7.33')
```

Decimal 객체를 생성하는 또다른 방법은 튜플을 사용해서 보호와 지수부, 가수부를 별도로 지정하는 것이다.

```
>>> Decimal((1, (1, 4, 7, 5), -2))
Decimal('-14.75')
```

첫숫자 1은 부호(0은 양수, 1은 음수)를 지정하고 -2는 소수점의 자리를 지정한다.

2 Context 객체

Context 객체는 Decimal 객체의 연산에 필요한 계산 정확도와 반올림 방법을 지정한다.

```
>>> getcontext()
Context(prec=28,  rounding=ROUND_HALF_EVEN,  Emin=-999999,  Emax=999999,  capitals=1,  clamp=0,
flags=[InvalidOperation, Inexact, FloatOperation, Rounded], traps=[InvalidOperation, DivisionByZero,
Overflow])
>>> Decimal(1) / Decimal(7)     # 유효 자리는 28자리이다.
```

```
Decimal('0.142857142857142857142857142857')
>>> getcontext().prec = 9       # 유효자리를 소수점 이하 9자리로 지정한다.
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
```

decimal 모듈은 미리 생성된 Context 객체 세가지 ExtendedContext와 BasicContext DefaultContext를 제공한다.

```
>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[])
>>> BasicContext
Context(prec=9, rounding=ROUND_HALF_UP, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[Clamped, InvalidOperation, DivisionByZero, Overflow, Underflow])
>>> DefaultContext
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
```

setcontext() 메서드를 사용해서 설정을 변경할수 있다.

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> setcontext(DefaultContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
```

자세한 내용은 라이브러리 레퍼런스에서 decimal 모듈을 찾아보기 바란다.

3.2 수치 연산자
이 절에서는 추리 자료형에 사용하는 연산자를 알아보기로 하자.

- 산술 연산자
① 산술연산자의 종류

앞서 계산기로 파이썬을 사용할 때 잠시 설명했지만 정리를 해보자. 파이썬의 기본적인 산술 연산자는 +(덧셈)과 -(뺄셈), *(곱셈), /(나눗셈), /(몫), **(지수), %(나머지) 등이다. 이들중에서 나눗셈과 몫, 지수, 나머지에 대한 예만 보이겠다.

```
>>> 5 / 2       # 나누기이다. 항상 부동 소수점의 결과를 얻는다.
2.5
>>> 5 // 2      # 몫만 취한다.
2
>>> 5 % 2       # 5를 2로 나눈 나머지를 얻는다.
1
>>> divmod(5, 2)        # 몫과 나머지를 한번에 계산한다.
(2, 1)
>>> 2 ** 3      # 2³
8
>>> 2 ** 3 ** 2 # 2³ ²  으로 2 ** (3 **2)와 같다.
512
>>> (2 ** 3) ** 2
64
```

음수의 지수형 연산은 다음과 같이 실수로 지수부를 표현하여 연산할수 있다.
```
>>> 5 ** -2.0
0.04
```

파이썬의 나머지는 제수의 부호와 같다.

```
>>> 5 % 3
2
>>> 5 % -3
-1
>>> -5 % 3
1
>>> -5 % -3
-2
```

따라서 몫은 다음과 같아진다.
```
>>> 5 // 3      # 1(몫) * 3 + 2 (나머지) == 5
```

```
1
>>> 5 // -3     # -2 * -3 -1 == 5
-2
>>> -5 // 3     # -2 * 3 + 1  == -5
-2
>>> -5 // 3     # -2 * 3 + 1  == -5
-2
```

2 산술 연산자의 우선순위

여러 연산자가 식 하나에 함께 있을 때 어떤 연산자는 다른 연산자보다 먼저 계산에 참여한다. 예를 들어, a + b * c 라는 식에서 b * c를 우선 계산한다. * 연사자가 + 연산자보다 우선순위가 높기 때문이다. 만일 a + b를 먼저 계산하고 싶으면 (a + b) * c와 같이 괄호를 사용하면 된다. 산술연산에 있어서 연산자의 우선순위는 표 3-1에서 확인할수 있다.

산술 연산자의 우선순위

산술 연산자	설명	결합 순서
+ -	단항 연산자	오른쪽에서 왼쪽으로(<-)
**	지수	오른쪽에서 왼쪽으로(<-)
* / % //	곱하기, 나누기, 나머지, 몫	왼쪽에서 오른쪽으로(->)
+ -	더하기, 빼기	왼쪽에서 오른쪽으로(->)

단항 연산자 -는 *연산자보다 우선순위가 높으므로 다음과 같은 연산을 할수 있다.

```
>>> 4 * -5
-20
```

3 산술 연산자의 결합 순서

동일한 우선순위를 갖는 연산자가 연속해서 나올 경우 결합순서는 왼쪽에서 오른쪽이다. +와 -, *, /, % 연산자는 왼쪽에서 오른쪽으로의 결합순서를 가진다. 결합 순서를 바꾸려면 (a + b)*c와 같이 괄호를 사용하면 된다.

```
>>> 2 + 3 * 4
14
>>> (2+3)*4
20
```

다항 연산자+와 -는 오른쪽에서 왼쪽으로의 결합순서를 가진다.

```
>>> ++ 3        # +(+3)
3
>>> -- 3        # -(-3)
3
>>> -+ 3        # -(+3)
```

```
-3
>>> +- 3         # +(-3)
-3
```

다음과 같은 수식에서 결합순서를 혼동하지 말아야 한다.

```
>>> 4 / 2 * 2
4.0
```

독자는 4/2*2는 4/(2*2)와는 다르다는 것을 알 것이다. 계산은 왼쪽에서 오른쪽으로 이루어지므로 (4/2)*2이 된다. 결과는 4.0이다. 하지만, ** 연산자는 결합순서가 다른 연산자와는 다르다. 이 연산자는 오른쪽에서 왼쪽으로 결합한다.

```
>>> 2 ** 3 ** 4
2417851639229258349412352
```

앞의 결과는 다음과 같은 결합 순서로 계산한 결과와는 다르다.

```
>>> (2 ** 3) ** 4
4096
```

- 관계 연산자

객체의 대소를 비교하는 연산을 관계 연산이라고 한다. 값을 서로 비교하는 연산자로는 ==을 사용한다. 치환 연산자 =와 혼동하지 말기 바란다.

```
>>> 6 == 9
False
>>> 6 != 9
True
>>> 1 > 3
False
>>> 4 <= 5
True
>>> a = 5
>>> b = 10
>>> a < b
True
```

파이썬은 다음과 같이 복합적인 관계식도 지원한다.

```
>>> 0 < a < b
True
>>> 0 < a and a < b     # 앞의 식과 같은 의미이다.
```

```
True
```

관계 연산자를 정리하면 표와 같다.

관계 연산자	설명
>	큰지 비교한다.
<	작은지 비교한다.
>=	크거나 같은지 비교한다.
<=	작거나 같은지 비교한다.
==	같은지 비교한다.
!=	같지 않은지 비교한다.

관계 연산자는 숫자뿐 아니라 객체 간에도 크기를 비교한다.

```
>>> 'abcd' > 'abd'
False
>>> (1, 2, 4) < (2, 1, 0)
True
>>> [1, 3, 2] == [1, 2, 3]
False
```

문자열의 비교는 사전순이다. 즉, 사전에서 앞에 나오는 단어가 순서가 빠르므로 값은 작게 취급된다. 튜플이나 리스트인 경우네 순서는 앞에서부터 하나씩 비교한다. (1, 2, 4) < (2, 3, 5)인 경우 우선 1과 2를 비교하고, 만일 두값이 같으면 두 번째 값인 2와 3을 비교한다. 언제든지 결과가 결정되면 비교를 중단한다.

하지만, 다른 자료형 간의 값은 비교할수 없다.

```
>>> 123 < 'abc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

관계 연산자 중 ==는 객체가 같은 값을 가지고 있는지를 검사한다. 만일 두 개의 참조가 같은 객체를 참조하고 있는지 알고 싶으면 is 연산자를 사용한다. 예를들어, 다음과 같은 세 개의 이름을 만들어 보자.

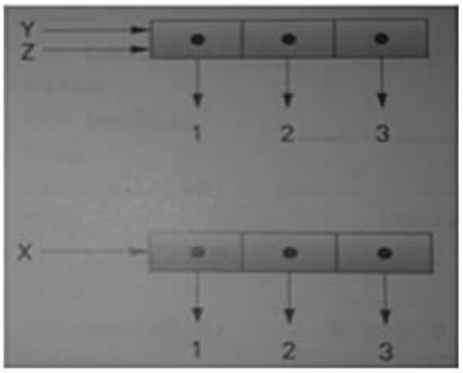
```
>>> X = [1, 2, 3]
>>> Y = [1, 2, 3]
>>> Z = Y
```

다음은 X와 Y, Z가 같은 값인지, 같은 객체인지 확인하는 예이다.

```
>>> X == Y
True
>>> X == Z
True
```

```
>>> X is Y
False
>>> X is Y
False
>>> Y is Z
True
```

X와 Y는 값이 같지만 다른 객체이고 Y와 Z는 값도 같고 객체임을 알수 있다. 그 이유는 Z = Y에 의해서 새로운 객체가 생성되는 것이 아니라 동일 객체에 대하여 참조가 복사되기 때문이다.



값을 변경하여 앞의 결과를 다시 한번 확인해 보자.

```
>>> Y[1] = 100
>>> X
[1, 2, 3]
>>> Y
[1, 100, 3]
>>> Z
[1, 100, 3]
```

즉, Y와 Z는 동일한 객체임을 알수 있다.

- 논리 연산자

□ 논리 연산자의 종류

논리 연산자는 진릿값을 피연산자로 취해서 논리값을 계산한다. 논리 연산자는 산술 연산자보다 우선순위가 낮다.

논리 연산자 표

논리 연산자(우선순위 순)	설명
not x	x가 거짓이면 True이고, 아니면 False이다.
x and y	x가 거짓이면 x이고, 아니면 y이다.

x or y	x가 참이면 x이고, 아니면 y이다.

다음은 간단한 예이다.

```
>>> a = 20
>>> b = 30
>>> a > 10 and b <50
True
```

2 객체의 진릿값

파이썬은 진릿값의 결과로 참이면 True, 거짓이면 False를 돌려준다. 내부에서 True는 1, Fasle는 0의 값을 가진다.

```
>>> True + 1
2
>>> False + 1
1
```

피연산자로 사용할 경우 정수는 0, 실수는 0.0 다른 자료형은 빈객체 (()，{}，[])가 거짓이며, 나머지는 모두 참으로 간주한다. None이라는 특별한 객체는 '아무것도 없다' 혹은 '아무것도 아니다.' 를 표현하기 위해서 사용하는 파이썬 내장 객체이다. 이 객체의 진릿값은 언제나 거짓이다.

```
>>> bool(0)     # 정수 0은 거짓이다.
False
>>> bool(1)
True
>>> bool(100)
True
>>> bool(-100)
True
>>> bool(0.0)   # 실수 0.0은 거짓이다.
False
>>> bool(0.1)
True
>>> bool('abc') )
True
>>> bool('') )
False
>>> bool([])    # 빈 리스트는 거짓이다.
False
>>> bool([1, 2, 3])
True
>>> bool(())    # 빈 튜플은 거짓이다.
```

```
False
>>> bool((1, 2, 3))
True
>>> bool({})    # 빈 사전은 거짓이다.
False
>>> bool({1:2})
True
>>> bool(None)  # None 객체는 거짓이다.
False
```

정리하면 다음과 같은 값들이 파이썬에서 거짓으로 간주하고 나머지 값은 참으로 간주한다.

```
None
0, 0.0, 0L, 0.0 +0.0j
“” , [], (), {}
```

3 any()와 all() 내장함수를 사용해서 여러 요소의 진릿값을 판정하기

진릿값 연산 결과가 리스트나 튜플과 같은 저장형에 있을 경우, 이 값들이 모두 True인지 아니면 하나라도 True가 있는지를 검사할수 있다.

```
>>> bool_list = [True, True, False]
>>> all(bool_list)      # 모두가 참인 경우 True이다.
False
>>> any(bool_list)      # 하나라도 참인 경우 True 이다.
True
```

다음은 실제로 활용한 예로 튜플 내장(나중에 배움)을 사용하여 여러 값을 한번에 검사한다.

```
>>> L = [1, 3, 2, 5, 6, 7]
>>> all(e < 10 for e in L)      # 모든 요소가 10 미만인가?
True
>>> any(e < 5 for e in L)       # 한 요소라도 5 미만인가?
True
```

4 논리식의 계산순서

and와 or연산자가 포함된 논리식은 결과를 판정하는데 최종적으로 기여한 객체를 식의 값으로 반환한다. 다시 말하면, and와 or연산자를 왼쪽부터 식을 계산하다가, 어떤 시점에서 결과가 알려지면 계산을 중단하고 해당 시점의 객체를 반환한다. 반환 값은 참이나 거짓이 아니다. 주의하기 바란다.

```
>>> 1 and 1
1
>>> 1 and 0
0
```

```
>>> 0 or 0
0
>>> 1 or 0
1
>>> [] or 1     # []는 거짓이므로 1을 참조한다.
1
>>> [] or ()    # []와 ()이 거짓이다.
()
>>> [] and 1    # []이 거짓이므로 1은 참조할 필요가 없다.
[]
>>> 1 and 2
2
>>> 1 or 2
1
>>> [[]] or 1   # [[]]는 참으로 간주한다. [] 요소를 가지는 리스트이다.
[[]]
>>> [{}] or 1
[{}]
>>> '' or 1     #  ' '은 거짓이다.
1
```

- 비트 연산자

파이썬에서도 C언어처럼 비트 단위로 조작하는 연산자를 제공한다. 이들 연산자는 정수형에만 적용된다. 비트 연산자는 다음 표와 같다.

비트 연산자

비트 연산자(우선순위 순)	설명
~	비트를 반전(1의 보수)시킨다.
<< >>	비트를 왼쪽으로 이동시키거나, 오른쪽으로 이동시킨다.
&	비트 단위 AND 연산을 수행한다.
^	비트 단위 XOR 연산을 수행한다.
|	비트 단위 OR 연산을 수행한다.

단항 연산자 ~는 비트 패턴을 반대로 한다. 0은 1로 1은 0으로 변환시킨다. 비트가 반전되었을때에 10진수로 얼마가 되는지는 2의 보수 개념을 이해해야 한다. 이 책의 범위를 벗어나서 자세한 설명은 하지 않았다.

```
>>> ~5          # 0000...0101
-6           <-  1111...1010
>>> ~-1         # 1111...1111
0            <-  0000...0000
```

시프트 연산자 <<는 비트를 왼쪽으로 지정한 숫자만큼 이동시킨다. 가장 왼쪽 비트는 버려지고 가장 오른쪽은

비트 0으로 채워진다.

```
>>> a = 3        # 0000 0011 편의상 8비트만 표시한다.
>>> a << 2       # 0000 1100 왼쪽으로 두 비트 이동한다. 오른쪽은 0으로 채워진다.
12
>>> 1 << 128
340282366920938463463374607431768211456
```

반면에 시트로 연산자 >>는 비트를 오른족으로 지정한 숫자만큼 이동시킨다. 가장 오른쪽 비트는 버려지고 가장 왼쪽은 최상위 비트로 채워진다.

```
>>> a = 4        # 0000 011
>>> a >> 1       # 0000 0010 왼쪽이 0으로 채워진다.
2
>>> a = -4       # 1111 1100                 워진다.
>>> a >> 1       # 1111 1110 왼쪽이 1로 채워진다.
-2
```

다음은 &와 |, ^ 세 개의 연산자에 대해 연산을 수행한 결과를 정리한 표이다. 각 값은 참과 거짓의 진릿값이 아니라 비트 값이다.

비트 연산자 &와 |, ^의 연산 결과

x 비트	y 비트	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

실제로 실행한 예를 보자

```
>>> a & 2        # 0000 0011 비트와 0000 0010 비트를 AND 연산한다.
0
```

```
>>> a | 8        # 0000 0011 비트와 0000 1000 비트를 OR 연산한다.
-4
>>> 0x0f ^ 0x06          # 0000 1111 비트와 0000 0110 비트를 XOR 연산한다.
9
```

비트 연산자 &는 특정한 비트를 0으로 만들기 위해서, 비트 연산자 |는 특정한 비트를 1로 만들기 위해서 비트 연산자 ^는 특정한 비트를 반전시키기 위해서 많이 사용한다.

- a & 0x0f a의 마지막 4비트만 그대로 하고 나머지는 모두 0으로 만든다. 마스킹이라고도 한다.
- a | 0x0f a의 마지막 4비트를 모두 1로 한다.
- a ^ -1 a의 모든 비트를 반전시킨다.

- 연산자 우선순위

파이썬의 전체 연산자에 대해 우선순위를 다음 표로 정리하였다. 우선순위가 높은 것부터 낮은 것으로 연산자가 나열되어 있다.

연산자의 우선순위(높은순)

연산자	설명
{key:value...}	사전 표시
[expressions...]	리스트 표시
(expressions...)	튜플 표시
f(arguments...) x[index:index] x[index] x.attribute	함수호출 슬라이싱 인덱싱 속성참조
**	지수
~	비트 단위 NOT(1의보수)
+ -	양수, 음수(단항 연산자)
* / // %	곱하기, 나누기, 몫, 나머지
+ -	더하기, 빼기(이항 연산자)
<< >>	시프트 연산
&	비트 단위 AND
^	비트 단위 XOR
|	비트 단위 OR
< <= > >= <> != ==	크기 비교
is, not is in, not in	신원 확인 멤버검사
not	논리 연산 not
and	논리 연산 and
or	논리 연산 or
Lambda	람다 표현식(함수 참조)

3.3 수치 연산 함수

- 수치 연산 함수

연산자 이외에도 내장된 수치 연산 함수들이 있다.

내장 수치 연산 함수

함수	설명
abs(x)	x의 절댓값을 구한다.
int(x)	x를 정수형으로 변환한다.
float(x)	x를 실수형으로 변환한다.
complex(re, im)	실수부 re와 허수부 im를 가지는 복소수를 구한다.
c.conjugate()	복소수 c의 켤레 복소수를 구한다.
divmod(x, y)	(x // y, x % y) 쌍을 구한다.
pow(x, y)	x의 y승을 구한다.
max(iterable),min(iterable)	최댓값과 최솟값을 구한다.
sum(iterable)	합을 계산한다.

내장 함수 목록 전체를 보려면 라이브러리 레퍼런스 2장을 참고하기 바란다. 다음은 수치 내장 함수를 사용하는 예이다.

```
>>> abs(-3)
3
>>> int(3.141592)
3
>>> int(-3.1415)
-3
>>> float(5)
5.0
>>> complex(3.4, 5)
(3.4+5j)
>>> complex(6)
(6+0j)
>>> c = complex(2, 3)
>>> c.conjugate()
(2-3j)
>>> divmod(5, 2)
(2, 1)
>>> pow(2, 3)
8
>>> pow(2.3, 3.5)
18.45216910555504
>>> max([1, 3, 5, 2, 7])
```

```
7
>>> min([1, 3, 5, 2, 7])
1
>>> sum([1, 3, 5, 2, 7])
18
```

- math 모듈의 수치 연산 함수

수치 연산을 수행하는 내장 함수 이외의 함수로는 별도로 만들어진 표준 모듈을 사용한다. 실수연산을 위해서 math, 복소수 연산을 위해서 cmath 모듈이 준비되어 있다. 모듈이란 파이썬이나 C로 만들어진 프로그램으로 내부에는 변수와 함수, 클래스 등이 정의되어 있다. 모듈을 사용하려면 우선 모듈을 가져와야 한다.

```
>>> import math
```

그러고서는 모듈(math)에 포함된 변수(pi)나 함수(sin) 등을 사용하기 위해서 다음과 같이 코드를 작성해야 한다.

```
모듈이름.변수
모듈이름.함수
```

다음은 math 모듈을 사용하는 예이다.

```
>>> print(math.pi)
3.141592653589793
>>> math.e
2.718281828459045
>>> print(math.sin(0.1))        # 0.1 라디안에 대한 사인 값이다.
0.09983341664682815
```

계속해서 math 모듈을 사용한 예이다.

```
>>> import math          # 모듈을 우선 가져온다.
>>> math.sqrt(2)         # 제곱근이다.
1.4142135623730951
>>>
>>> r = 5.0     # 반지름을 설정한다.
>>>            # 모듈 안에 정의된 상수나 함수를 사용하려면
...
>>>            # modulename.variable 또는 modulename.function과 같이 사용한다.
...
>>> a = math.pi * r * r         # 면적을 구한다. 모듈에 PI 상수 값이 정의되어 있다.
>>> degree = 60.0
>>> rad = math.radians(degree)  # 각도를 라디안으로 변환한다.
```

```
>>> # sin과 cos, tan 값을 계산한다.
...
>>> print(math.sin(rad), math.cos(rad), math.tan(rad))
0.8660254037844386 0.5000000000000001 1.7320508075688767
```