파이썬의 데이터 타입 이해하기

효과적인 데이터 기반 과학 및 계산을 위해서는 데이터가 어떻게 저장되고 가공되는지 이해해야 한다.

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
   result += i;
}
```

그러나 파이썬에서는 같은 연산을 다음과 같이 작성할 수 있다.

```
# Python code
result = 0
for i in range(100):
   result += i
```

주요차이점을 알아보면 c 에서는 각 변수의 데이터 타입을 명시적으로 선언했지만, 파이썬은 타입을 동적으로 추론한다. 이것은 곧 모든 변수에 어떤 종류의 데이터 든 할당할 수 있다는 뜻이다.

```
# Python code
x = 4
x = "four"
```

위 코드는 x 의 내용을 정수에서 문자열로 바꾼 것이다. C 에서 똑같은 작업을 하면 컴파일러 에러나 다른 의도하지 않은 결과가 발생할 수도 있다. (컴파일러 설정에 따라 다름)

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

이러한 유연성은 파이썬과 다른 동적 타입 지정 언어를 사용하기 편하고 쉽게 만드는 특징 중 하나다. 이 것이 어떻게 가능한지를 이해하는 것이 파이썬으로 효과적이면서 효율적으로 데이터를 분석하는 방법을 배우는 데 중요한 부분이다. 그러나 이러한 유연성은 또한 파이썬 변수가 그 값 이상의 무언가를 나타낸다는 뜻이기도 하다. 즉, 변수는 그 값의 유형에 대한 부가 정보도 함께 담고 있다.

파이썬 정수는 장수 이상이다.

표준 파이썬은 c 로 구현돼 있다. 이 말은 곧 모든 파이썬 객채가 그 값뿐만 아니라 다른 정보까지 포함하는 똑똑하게 위장한 C 구조체라는 뜻이다. 예를 들어 x = 10000 과 같이 파이썬에서 정수를 정의할 때 x 는 단순히 있 는 그대로의 정수를 의미하지 않는다. 실제로는

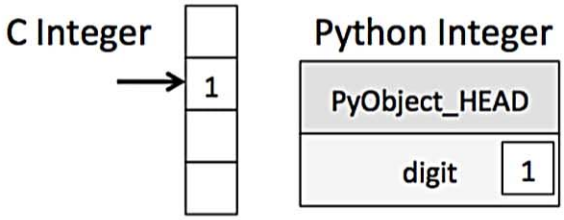
여러 값이 들어 있는 복합적인 C 구조체다. C 매트로를 확장하여 파이썬 3.6 소스코드를 보면 정수(long)타입 정의가 실제로 다음과 같이 되어 있다.

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

파이썬 3.6 의 단일 정수는 실제로 다음 네 가지 구성요소를 갖는다.

- ob_refcnt : 파이썬이 조용히 메모리 할당과 해제를 처리할 수 있게 돕는 참조 횟수
- ob_type : 변수 타입을 인코딩
- ob_size : 다음 데이터 멤버의 크기를 지정
- ob_digit : 파이썬 변수가 나타내는 실제 정수값을 포함

이것은 아래 그림처럼 C 와 같은 컴파일 언어에서 정수를 저장하는 것에 비해 파이썬에서 정수를 저장할 때 어느 정도 오버헤드가 있다는 의미다.



여기서 PyObject_HEAD 는 참조 횟수, 타입 코드, 그리고 전에 언급하 다른 정보를 포함한 구조체의 일부다.
차이점이라면 C 정수는 근본적으로 정수값을 나타내는 바이트를 포함하는 메모리 위치를 가리키는 레이블이고 파이썬 정수는 정수값을 답고 있는 바이트를 포함한 모든 파이썬 객체 정보를 포함하는 메모리의 위치를 가리키는 포인터라는 사실이다. 파이썬 정수 구조체의 이 추가 정보 덕분에 파이썬에서 그토록 자유롭게 동적으로 코드를 작성할 수 있는 것이다. 하지만 파이썬 타입에 있는 모든 추가 정보에는 비용이 따르며, 특히 이 객체들을 여러 개 결합하는 구조체에서 그 비용이 분명하게 드러난다.

파이썬 리스트는 리스트이상이다.

이번에는 여러 개의 파이썬 객체를 담은 파이썬 자료구조를 사용할 때 어떤 일이 벌어지는지 생각해 보자. 파이썬에서 여러 개의 요소를 담는 가변적인 표준 컨테이너는 리스트이다. 다음과 같이 정수 리스트를 만들 수 있다.

```
In [1]: L = list(range(10))
    L
```

```
Out[1]:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: type(L[0])
Out[2]: int
```

또는 이와 비슷하게 문자열 리스트를 만들 수 있다.

```
In [3]: L2 = [str(c) for c in L]
        L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
In [4]: type(L2[0])
Out[4]:str
```

파이썬의 동적 타이핑(Dynamic typing) 덕분에 서로 다른 데이터 타입의 요소를 담는 리스트를 만들수도 있다.

```
In [5]: L3 = [True, "2", 3.0, 4]
     [type(item) for item in L3]
Out[5]: [bool, str, float, int]
```

그러나 이 유연성에는 비용이 따른다. 이렇게 유연한 타입을 허용하려면 리스트의 각 항목에 타입 정보와 참조 횟수, 기타 정보가 들어가야 한다. 즉, 각 항목이 완전한 파이썬 객체인 셈이다. 모든 변수가 같은 타입인 경우에는 이 정보가 대부분 불필요하게 중복되므로 고정 타입 배열에 데이터를 저장하는 것이 더 효율적일 수 있다. 동적 타입 리스트와 고정 타입(Numpy 스타일) 배열의 차이는 아래 그림에 나타냈다.

구현 레벨에서는 배열이 근본적으로 인접한 데이터 블록을 가르키는 단일 포인터를 담고 있다. 반면 파이썬 리스트는 앞에서 본 파이썬 정수와 같이 완전한 파이썬 객체를 가르치는 포인터의 블록을 가르키는 포인터를 답고 있다. 다시 말해 리스트의 장점은 유연성이다. 각 리스트 요소가 데이터와 타입 정보를 포함하는 완전한 구조이기 때문에 리스트를 원하는 어떤 타입으로도 채울 수 있다. 고종 타입의 Numpy 스타일 배열은 이러한 유연성은 부족하지만 데이터를 저장하고 가공하기에는 훨씬 효율적이다.

파이썬의 고정 타입 배열

파이썬은 데이터를 효율적인 고정 타입 데이터 버퍼에 저장하는 다양한 방식을 제공한다. 내장 array 모듈(파이썬 3.3 부터 제공됨)은 단일 타입의 조밀한 배열(dense array)을 만드는 데 사용할 수 있다.

```
In [6]: import array
     L = list(range(10))
     A = array.array('i', L)
     A
```

```
        # 값은 해당 메모리 위치에 이미 존재하고 있는 값으로 채움
        np.empty(3)
Out[21]:array([ 1.,  1.,  1.])
```

NumPy 표준 데이터 타입

Numpy 배열은 한 가지 타입의 값을 담고 있으므로 해당 타입과 그 타입의 제약 사항을 자세히 아는 것이 중요하다. Numpy 는 C 로 구현됐기 때문에 Numpy 데이터 타입은 C 와 포트란, 그 밖의 다른 관련언어의 사용자에게 익숙할 것이다.

표준 Numpy 데이터 타입을 아래 표의 정리했다. 배열을 구성할 때 데이터 타입은 문자열을 이용해 지정할 수 있다.

```
np.zeros(10, dtype='int16')
```

또는 해당 데이터 타입과 관련된 Numpy 객체를 사용해 지정한다.

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)

float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

빅 엔디언이나 리틀 엔디언 숫자처럼 고급 타입을 지정하는 것도 가능하다.