CS206A: Data Structures

2017-09-06

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯

2022년 6월 26일

객체

파이썬 프로그램으로 생성되고 사용되는 모든 데이터를 객체(object)라고 부른다. (정수 및 부동 소수점 수를 비롯한) 수와 부울 값(Boolean, `True` 및 `False`) 같은 단순한 객체도 있다. 문자열, 리스트, 튜플(tuple) 같은 표준 객체도 있다. 또 특수 라이브러리나 사용자가 정의한 객체도 있다. 이미지, `cs206draw` 모듈, CS201 수업 로봇 객체를 예로 들 수 있다.

객체는 정보를 저장한다. 이 정보를 객체의 상태(state)라고 부른다. 요컨대 한 수 객체의 상태란 그 수의 값이다. 한 문자열의 상태란 그 문자열을 이루는 문자들이다. 한 리스트의 상태란 그 리스트의 길이와 내용이다.

이와 더불어 객체는 객체의 상태에 접근하거나 상태를 조작하기 위한 (즉, 객체에 의해 저장된 정보를 바꾸기 위한) 메소드(method, 방법)를 제공한다. 파이썬에서는 특수한 구문(syntax)을 사용해 다양한 메소드를 호출한다. 가령 `len(s)`는 실제로 `s.__length__` 메소드를 호출한다. `a + b`는 `a.__add__(b)` 메소드를 호출한다. `a in b`는 `b.__contains__(a)` 메소드를 호출한다.

모든 객체는 특정 타입(type, 型, 형)을 지닌다. 타입은 객체로 할 수 있는 일을 결정한다. 정확히 말해 객체가 지원하는 메소드를 결정한다. 이를테면 `b`가 리스트일 때는 `a in b`를 작성할 수 있다. 하지만 `b`가 수일 때는 작성할 수 없다. 이와 유사하게 `a`와 `b`가 수일 때는 `a + b`라는 표현식이 덧셈을 뜻한다. 하지만 `a`와 `b`가 문자열일 때는 결합(concatenation)을 뜻한다.

지금껏 파이썬 내장 또는 라이브러리 제공 객체 타입을 이미 조금은 사용했다. 장차 각자만의 타입을 만들고 싶을 테다. `class` 키워드를 이용하면 끝날 일이다. 클래스(class, 類, 류)란 객체를 위한 청사진이다. 클래스 이름 작성만으로도 청사진에서 객체를 생성해 낼 수 있다.

변수와 힙

모든 객체는 힙(heap, 더미)이라고 부르는 파이썬 인터프리터 메모리 영역에 저장된다.

변수(variable)는 정말 단순히 객체에 대한 이름(name)에 다름 아니다. 따라서 변수는 값을 "포함"하지 않는다. 변수는 단지 객체에 대한 레퍼런스(reference, 참조, 지시체)다. 이에 변수가 객체를 연결한다(links)고 말하거나 참조한다(references)고 말할 것이다.

동일한 객체는 다중의 이름을 지닐 수 있다. (즉 여러 변수는 동일한 객체를 동시에 "참조"할 수 있다) 한 이름의 의미는 프로그램 진행 중에 변화할 수 있다. 이것이야말로 대입 명령(assignment instruction)의 업무다. 바로 변수로 하여금 다른 객체를 참조하도록 만드는 것이다.

이름은 쓰이지 않을 수도 있다. 이때는 변수가 `None` 값을 지닌다고 말한다. `a is None`이라고 작성하여 `a` 변수가 쓰이지 않았는지 검증할 수 있다. 아니면 부정형인 `a is not None`을 작성하여 검증할 수도 있다. (이것이 일반적인 부정형 `not (a is None)`보다 짧으며 가독성도 좋다는 점에 주목하라)

가변 및 불변 객체

어떤 객체는 한 번 생성된 이상 자신의 상태를 바꿀 수 없다. 그런 객체를 불변(immutable)이라고 한다.

가령 수 타입은 불변이다. 예를 들어 3이라는 수를 위해 하나의 객체를 한 번 생성한 이상 이 객체는 항상 3이라는 수를 저장하리다. 이 객체는 절대 그 값을 바꿀 수 없다. 학생들은 종종 여기서 헷갈린다. 아래처럼 작성할 수 있기 때문이다.

```
1    a = 3
2    a = 5
```

하지만 주의하라. 두 번째 라인은 첫 번째 라인에서 생성된 3 값 객체의 값을 바꾸지 않는다. 대신 두 번째 라인은 새로운 객체를 5 값으로 생성한다. 그리고 변수 a로 하여금 이 새로운 객체를 연결하도록 바꾼다.

파이썬에서는 문자열도 불변이다. 문자열 객체를 한 번 생성한 이상 문자를 대체하는 등 그 값을 바꿀 수 없다. 문자열을 바꾸는 것처럼 보이는 임의의 문자열 메소드는 사실 새로운 문자열 객체를 반환하는 것이다. 아래 예에서

```
>>> s = "Hello CS109"
>>> t = s.upper()
>>> s
'Hello CS109'
```

`upper()` 메소드 호출에도 `s`가 변하지 않는다는 점을 관찰하라. `upper()` 메소드는 새로운 문자열 객체를 반환할 따름이다.

튜플은 또 다른 불변 객체다. 하나의 튜플은 한 번 생성한 이상 구성 요소를 갱신할 수 없다.

마지막으로 `frozenset`이라고 부르는 특수한 집합 객체가 있다. 어디에 쓰이는지는 에 살핀다.

```
>>> s1 = set([1, 2, 3])
>>> s2 = frozenset([1, 2, 3])
>>> s1.add(9)
>>> s1
{1, 2, 3, 9}
>>> s2.add(9)
AttributeError: 'frozenset' object has no attribute 'add'
```

파이썬 객체 대다수는 가변(mutable)이다. 즉 객체를 생성하고 나서도 상태를 바꿀 수 있다는 뜻이다. 가변 객체는 다음처럼 "흥미로운" 행동을 유도할 수 있다.

```
>>> A = [1, 2, 3, 4]
>>> B = A
>>> A
[1, 2, 3, 4]
>>> B
[1, 2, 3, 4]
```

```
>>> A[2] = 99
>>> A
[1, 2, 99, 4]
>>> B
[1, 2, 99, 4]
```

분명 A 리스트의 내용을 조정했다. 하지만 그러자 B 리스트의 내용도 변했다. 무슨 일이 일어났는가?

답은 물론 이렇다. 바로 리스트 객체란 오직 하나뿐이라는 것이다. A와 B 모두 동일한 객체에 대한 이름이다. 혹은 달리 말하면 A와 B라는 두 변수는 동일한 리스트 객체를 참조한다.

종국에 다중 이름을 지닐 수 있는 가변 객체를 다룰 적에는 주의를 기울여야 한다. 가변 객체는 여러 이유 덕분에 일반적으로 선호된다. 그런데 효율성이 쟁점인 경우는 예외다. 오늘날에는 여러 CPU가 동일한 데이터에 동시에 접근하는 멀티코어 컴퓨터 때문에 불변 객체 설계가 더욱더 중요해지고 있다.

지역 변수

앞서 논했듯 모든 객체는 힙에 저장된다. 어느 객체도 다른 곳에서는 살아갈 수 없다. 또 변수는 단지 힙 내부의 객체에 대한 레퍼런스를 저장할 뿐이다. 실제로 값을 포함하지는 않는다.

그렇다면 이들 레퍼런스는 어디에 저장되는가? 어떤 레퍼런스는 객체 내부에 저장된다. 아래 예를 고찰해 보라.

```
a = [ "Hello CS206", 13 ]
```

여기서 a는 리스트 객체에 대한 이름이다. 리스트 객체 내부는 두 가지 또 다른 객체인 문자열과 정수에 대한 레퍼런스들이다.

하지만 프로그램에서 쓰인 변수 이름은 어떤가? 아래 자그마한 함수를 고찰해 보라.

```
1  def test(m):
2      k = m + 27
3      s = "Hello World"
4      A = [ len(s), k, m ]
```

이 함수는 네 개의 지역 변수를 지닌다. m, k, s, A. 맞다. 제대로 읽었다. 매개변수(parameter) m 또한 지역 변수다. 매개변수와 지역 변수 간의 유일한 차이는 매개변수의 경우 함수 호출 시 전달인자(arguments)에서 값이 주어진다는 사실뿐이다.

test 함수가 호출될 때마다 파이썬 인터프리터는 (함수가 활성화되고 있다는 이유로 인해) 활성화 레코드(activation record)라고 부르거나 (이 이름에 관해서는 나중에 다룰 텐데) 스택 프레임(stack frame)이라고 부르는 특수한 기억 영역을 생성한다. 함수가 반환될 때 스택 프레임은 파괴된다. 그리고 지역 변수가 더는 존재하지 않게 된다.

지역 변수를 다뤘으니 전역 변수(global variables)가 어디 저장되는지 궁금할 테다. 프로그램의 모든 파이썬 모듈에 대해 객체가 존재한다. 그리고 이 객체가 전역 변수를 저장한다. (그러니 어떤 의미에서는 전역 변수란 객체 안의 필드(fields, 場, 장)와 같다)

가비지 수집

전형적인 프로그램은 짧은 기간 쓰이고 마는 다수의 작은 객체를 생성한다. 결과는 쓰이지 않는 객체로 채워지는 힙이다. 어느 지점에 도달하면 힙은 단순히 말해 꽉 찬다. 더는 새로운 객체를 저장할 수 없다.

이때 파이썬 인터프리터는 가비지 수집(garbage collection)을 수행해야 한다. 힙을 채운 모든 객체를 각각 조사하고 객체가 가비지(garbage, 쓰레기)인지 결정하는 것이다. 가비지라면 말 그대로 객체가 더는 쓰일 수 없다. 이들 가비지 객체는 폐기되고 프로그램은 계속될 수 있다.

인터프리터는 하나의 객체가 여전히 유용한지 어떻게 아는가? 자, 하나의 객체가 유용하려면 객체는 이름을 지녀야 한다. 다시 말해 그 객체를 참조하는 변수가 (또는 다른 객체 내부의 필드가) 있어야 한다. 따라서 파이썬 인터프리터는 모든 전역 변수와 활동 중인 모든 지역 변수에서 시작해 여전히 도달할 수 있는 모든 객체를 발견하고자 이들 변수의 레퍼런스를 따라간다. 이 절차에서 발견하지 못한 모든 객체는 가비지다.

C나 C++로 프로그래밍해 봤다면 (`malloc` 또는 `new`를 통해) 모든 객체에 대해 할당(allocate)했을 때 나중에는 (`free` 또는 `delete`를 통해) 메모리를 해제(free)해야 한다고 배웠을 테다. C와 C++는 가비지 수집을 지니지 않는다. 그래서 프로그래머가 힙 관리의 책무를 맡는다. 그 결과는 바로 C와 C++ 프로그램이 종종 메모리 누수(memory leaks)를 지닌다는 것이다. 어떤 객체는 생성되지만 절대 파괴되지 않는다. 그렇게 천천히 힙은 쓰이지 않는 객체로 채워진다. 그리고 프로그램은 점점 더 많이 컴퓨터의 메모리를 사용한다. 파이썬, 자바(Java), 코틀린(Kotlin)을 비롯하여 가비지 수집을 지닌 여타 언어에서는 메모리 누수를 걱정할 필요가 없다.

현대 가비지 수거기는 상술(上述)한 도식보다 정교하다. 현대 가비지 수거기는 가비지가 되는 순간 재빠르게 알아채기 위해 (그런 다음 즉각 폐기하기 위해) 각 객체의 부기(簿記, bookkeeping) 정보를 저장할 수도 있다. 또 어떤 가비지 수집은 계산(computation)과 병렬적으로 증가하여 수행할 수도 있다. 효율적인 가비지 수집은 활동적인 연구 분야다.

클래스 정의하기

앞선 수업의 계산기에서 `Token` 클래스를 사용해 클래스 정의 방법을 학습하겠다. 아래에 클래스 정의가 있다.

```
1  class Token(object):
2      def __init__(self, text, pos, type):
3          self.pos = pos
4          self.type = type
5          if type == "number"
6              self.value = float(text)
7          else:
8              self.value = text
```

`class` 키워드는 대문자로 시작하는 관행이 있는 클래스의 이름으로 이어진다. 괄호 안에는 "슈퍼 클래스"(superclass)의 이름이 이어진다. 이 수업에서는 특정한 슈퍼 클래스가 없다는 점을 나타내기 위해 거의 항상 `object`라고 작성할 것이다. 객체 지향 프로그래밍을 학습할 때 슈퍼 클래스, 서브클래스, 클래스, 계층 구조에 관해 배울 것이다.

클래스 내부의 def 키워드는 메소드 정의(method definition)를 개괄한다. 여기에는 특수한 요술 이름인 __init__을 지니는 메소드가 하나 있다. 이 메소드를 Token 클래스의 생성자(constructor)라고 부른다. 생성자는 Token 타입의 객체가 생성될 때마다 실행된다. 생성자의 책무는 객체의 상태를 정확하게 초기화하는 것이다.

우리의 토큰 객체는 아주 단순하다. 토큰 객체는 토큰의 타입을 ("number", "identifier", "symbol", "stop" 가운데 하나의 문자열로) 저장하고, 토큰의 값을 (수 토큰에 대해서는 수치로 혹은 식별자나 기호 토큰에 대해서는 문자열로) 저장하고, 토큰이 시작하는 입력 문자열 상의 위치를 (오류 보고를 위해) 저장한다.

생성자는 이들 세 가지 정보를 위해 세 매개변수를 취하고, 이들을 Token 객체의 필드 type, value, pos에 저장한다. 다만 생성자 메소드가 실제로는 네 가지 매개변수를 지닌다는 점에 유의하라. 첫 번째 매개변수는 항상 self라고 부르는 것이며 생성되고 있는 Token 객체를 지시한다. 따라서 self.type에 대한 대입은 객체 내부에 type 필드를 생성한다. self.value, self.pos에 대해서도 비슷하다.

이 코드 덕분에 Token 객체를 생성할 수 있다. 바로 이렇게.

```
>>> t = Token("3.5", 7, "number")
>>> t.type
'number'
>>> t.value
3.5
>>> h = Token("*", 12, "symbol")
>>> h.type
'symbol'
>>> h.value
'*'
```

생성자에 의해 객체가 한 번 마련된 이상 객체의 상태를 포함하는 필드에 접근할 수 있다는 점에 주목하라.

메소드 정의하기

Token이 단순한 클래스더라도 파싱 코드의 가독성을 증진하려면 메소드를 추가하는 편이 낫겠다. 가령 t.type == "number" 대신 t.isNumber()라고 적고자 한다. 그리고 아래처럼 적는 대신

```
t.type == "symbol" and t.value == "*"
```

t.isSymbol("*")라고 적고자 한다. 밑에 메소드 정의가 있다. 다시 말하지만 이들은 클래스 정의 내부에 들어간다.

```
1     def isNumber(self):
2         return self.type == "number"
3     def isSymbol(self, s):
4         return self.type == "symbol" and self.value == s
5     def isIdentifier(self):
6         return self.type == "identifier"
7     def isStop(self):
8         return self.type == "stop"
```

각 메소드의 첫 매개변수를 self라고 부른다는 점에 다시 주목하라. 메소드를 호출할 때 이 매개변수는 Token 객체 자체를 참조할 것이다. 가령 t.isSymbol("*")을 호출할 때는 self가 t 객체가 될 것이며 s가 문자열 "*"가 될 것이다.

문자열로 변환하기

객체를 근사한 포맷(format)에 출력할 메소드는 클래스에 추가하는 편이 종종 용이하다. 이번에도 요술 메소드 이름이 존재한다. 바로 __str__이다. 이 메소드는 객체가 str 함수를 통해 문자열로 변환될 때마다 호출된다. 가령 print 함수로 객체를 출력할 때 이런 일이 일어난다. print 함수는 전달인자를 자동으로 문자열로 변환한다.

아래는 Token 클래스의 __str__ 메소드를 위한 정의다.

```
    def __str__(self):
        if self.isNumber():
            return "Number: %g" % self.value
        if self.isIdentifier():
            return "Identifier: %s" % self.value
        if self.isStop():
            return "Stop"
        return "Symbol: %s" % self.value
```

다시 말하지만 self는 Token 객체를 지시한다. 아래는 조금의 테스트다.

```
>>> str(h)
'Symbol: *'
>>> str(t)
'Number: 3.5'
>>> print(h)
Symbol: *
>>> print(t)
Number: 3.5
```

print가 어떻게 Token 객체를 문자열로 변환하는지 유의하라.

파이썬은 사실 문자열 변환의 두 번째 형식을 제공한다. __repr__이라는 요술 메소드 이름을 사용한다. 이 메소드는 repr 함수 사용으로 호출된다. 또 각 입력 라인의 결과를 보이기 위해 인터랙티브 모드에서도 쓰인다.

__repr__ 메소드는 디버깅을 위한 것이다. 검증을 위한 프로그래머에 의해 인터랙티브 모드에서 주로 쓰인다. "말단 사용자"를 위한 코드에서는 잘 쓰이지 않는다. 따라서 그 결과가 사뭇 상이하다.

```
>>> print(7)
7
>>> print(repr(7))
7
>>> print("hello")
hello
>>> print(repr("hello"))
'hello'
```

수에 대해서는 str과 repr의 결과 간에 차이가 없다. 다만 문자열에 대해서는 repr이 한 쌍의 따옴표를 더 지닌다는 점을 확인할 수 있다. 문자열에 특수한 문자들을 포함하면 차이점은 더 두드러진다. (\t는 탭(tab) 문자를 뜻하며, \n은 개행(newline) 문자를 뜻한다)

```
>>> s = "Hello\tCS206!\n"
>>> print(s)
Hello    CS206!

>>> print(repr(s))
'Hello\tCS206!\n'
```

문자열을 출력할 적에는 특수 문자들이 해석된다. 다만 repr의 출력은 특수 문자를 포함한다. 이로 인해 우리는 repr의 출력을 그대로 파이썬 인터프리터에 되먹여서 동일한 문자열의 사본을 생성할 수 있다.

이것이 repr 변환의 일반적인 설계 원리다. repr의 출력은 종종 객체의 사본을 생성하게끔 인터프리터로 다시 복사될 수 있는 포맷을 지닌다. 이 설계를 Token 클래스에 이용해 보자.

```
    def __repr__(self):
        return "Token(%s, %d, %s)" % (repr(self.value), self.pos, repr(self.type))
```

이것이 수와 문자열 값 모두에 대해 정확하게 작동한다는 점에 주목하라.

```
>>> print(repr(t))
Token(3.5, 7, 'number')
>>> print(repr(h))
Token('*', 12, 'symbol')
```

이 출력을 다시 인터프리터로 복사해 Token 객체를 재생성할 수도 있다.

repr 변환을 정의하기 위한 좋은 이유 하나는 바로 repr이 객체의 리스트를 보도록 허락한다는 점이다.

```
>>> toks = tokenize("a23 * (3 - 4 * x)")
>>> toks
    [Token('a23', 0, 'identifier'), Token('*', 4, 'symbol'),
     Token('(', 6, 'symbol'), Token(3.0, 7, 'number'), Token('-', 9, 'symbol'),
     Token(4.0, 11, 'number'), Token('*', 13, 'symbol'),
     Token('x', 15, 'identifier'), Token(')', 16, 'symbol'),
     Token('', 17, 'stop')]
```

이것이 작동하는 이유는 리스트 객체에 대한 문자열 변환이 각 리스트 원소에 대해 repr 변환을 사용하기 때문이다.

객체 구현과 클라이언트 코드

이상적으로 말하면 특정 타입의 객체를 이용하는 코드는 그 객체가 어떻게 구현되었는지 알 필요가 없다. 요컨대 파이썬 인터프리터가 파이썬의 리스트 객체를 어떻게 그다지도 효율적으로 구현하는지 알 필요 없이 파이썬의 리스트 객체를 사용할 수도 있다. (하지만 알게 되리라, 내가 약속한다!)

따라서 종종 클래스를 구현하는 코드와 클래스를 이용하는 코드를 구분하는 편이 말이 될 때가 있다. 일반적으로는 하나의 클래스에 대한 (혹은 소수의 밀접하게 연관된 클래스들에 대한) 정의와 아마 이 타입의 객체로 작동하는 함수를 포함하는 하나의 파일을 지닌다. 이런 파일을 파이썬 모듈(module)이라고 부른다.

한편 객체를 이용하는 코드를 클라이언트 코드(client code)라고 부른다. 클래스가 어떤 유형의 서비스를 제공한다고 생각해 보라. 클라이언트 코드는 이 서비스의 "소비자"다. 그리하여 "클라이언트 코드"라는 이름인 셈이다.

클래스를 이용하려면 클라이언트 코드는 클래스를 정의하는 모듈을 임포트(import, 들여오기)해야 한다. 이 경우에는 아래 라인이 필요하다.

```
import tokens
```

그런 다음 Token 클래스를 이용하고 모듈 이름으로 접두사를 붙여서 tokenize 함수를 이용할 수 있다.

```
>>> import tokens
>>> h = tokens.Token("3.5", 7, "number")
>>> toks = tokens.tokenize("3 * 5 - 7 / x")
```

어떤 때는 모듈 이름을 매번 적는 일이 거슬리기도 한다. 그런 경우 모듈에서 모든 클래스와 함수를 클라이언트 코드의 이름 공간(name space)에 임포트할 수 있다. 바로 이렇게.

```
>>> from tokens import *
>>> h = Token("3.5", 7, "number")
>>> toks = tokenize("3 * 5 - 7 / x")
```

객체 상등

== 연산자는 임의 종류의 객체의 상등(相等, equality) 검증에 쓰일 수 있다.

```
>>> h
Token(3.5, 7, 'number')
>>> h1 = h
>>> h2 = Token("3.5", 7, "number")
>>> h2
Token(3.5, 7, 'number')
>>> h == h1
True
>>> h == h2
False
```

h == h1이 참이라는 점에 주목하라. 왜냐하면 h와 h1이 실제로 동일한 객체이기 때문이다. 하지만 h == h2는 두 객체가 정확히 동일한 상태를 지니는데도 거짓을 반환한다.

Token 클래스에 또 다른 요술 이름과 함께 하나의 메소드를 추가함으로써 이를 고칠 수 있다.

```
1     def __eq__(self, rhs):
2         return (self.type == rhs.type and
3                 self.value == rhs.value)
```

이 예제에서 pos 필드를 확인하지 않았고 그러므로 타입과 값이 모두 일치할 때 두 Token 객체를 상등으로 간주한다는 점에 주목하라.