CS206A: Data Structures

2017-09-13

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯
2022년 7월 2일

자료형과 자료 구조

자료형(資料型, data type). (추상 자료형(abstract data type) 혹은 ADT라고도 부른다.) 객체가 지원하는 연산과 행태(operation and behavior)를 정의한다. 개념이다. "함수", "집합", "수열" 같은 수학 개념과 비슷하다.

자료형과 자료 구조(data structure)를 혼동하지 마라. 중요하다. 자료형을 구현(implement)하면 자료 구조다. 자료 구조는 객체를 하나씩 구현한다. 바로 자료형에서 정의한 모든 연산을 정확한 형태로 제공하는 객체다.

동일한 자료형을 다르게 구현할 수 있다. 꽤 자주 일어나는 일이다. 예를 들어 스택은 배열이나 링크드 리스트(linked list)로 구현할 수 있다. 집합은 탐색 트리(search tree)나 해시 테이블(hash table)로 구현할 수 있다.

CS206A 강의명은 자료 구조다. 하지만 자료형과 자료 구조가 더 나은 이름이겠다. 여러 중요한 자료형과 만날 예정이다. 특히 스택, 큐, 집합, 우선순위 큐, 맵과 만나겠다. 또 자료형 구현에 쓸 수 있는 다양한 자료구조 기법을 논할 예정이다. 특히 링크드 리스트, 트리, 균형 트리, 해시 테이블과 만나겠다.

문제로 고민할 때는 해결에 사용할 자료형을 고찰하는 편이 좋다. 추상 자료형을 정의하려면 해야 하는 일이 있다.

- 자료형이 지닐 수 있는 여러 값을 특정해야 한다
- 자료형으로 수행할 수 있는 여러 연산을 정의해야 한다

할 일을 했다면 새로운 자료형으로 손수 문제 해결에 임할 수 있다. 구현 세부 사항은 무시한다.

허용 연산 정의에 주의를 기울이면 자료형 구현으로 접근하려는 클라이언트는 멈추어 선다. 논리적인 오류도 방지한다.

자료형은 나중에 구현을 바꿀 수 있다. 원래 구현이 너무 느리다는 사실이나 다른 단점을 알아채는 순간 빛을 발하는 중요한 능력이다. 신규 구현이 동일한 연산과 행태를 제공하기만 하면 클라이언트 코드는 변하지 않는다.

마지막으로 추상 자료형. 큰 프로그램에서 작은 모듈로 나누는 깔끔하고 체계적인 방법을 제공한다. 작은 모듈은 관리할 만하다. 그러므로 서로 다른 프로그래머나 팀이 구현할 수도 있다.

요일 계산기 예제

예제를 살핀다. 요일 계산기를 구현하려고 한다. 요일 계산기는 여러 물음에 답한다.

- 주어진 날짜에 대해 무슨 요일인가?
- 주어진 두 날짜에 대해 얼마나 많은 나날이 있는가?
- 주어진 날짜와 주어진 나날에 대해 나날 이전 날짜와 나날 이후 날짜는 무엇인가?

프로그램을 작성하려면 날짜(dates)를 표현하는 객체를 지니는 편이 좋겠다. 분명하다. 추상 자료형 `Date`를 아래처럼 정의한다.

- `Date(yr, m, d)`는 신규 날짜 객체를 생성한다.
- `day()`는 일(日)을 반환한다.
- `month()`는 월(月)을 반환한다.
- `year()`는 연(年)을 반환한다.
- `dayOfWeek()`는 $0 \ldots 6$ (0이 월요일) 가운데 수로 요일을 반환한다.
- `numDays(otherDate)`는 두 날짜 사이 나날 수를 반환한다.
- `advanceBy(n)`은 n 나날 이후 날짜를 (혹은 n이 음수인 경우 n 나날 이전 날짜를) 반환한다.

또 날짜 비교는 `==`와 `<`로 할 수 있기를 바란다.

ADT를 정의했다. 고로 실제 클라이언트 코드를 작성할 수 있다. (바로 `Date` 자료형을 이용하는 코드를 뜻한다. 기억해 둬라.) 예를 들면 이렇게.

```
>>> a = Date(1996, 9, 3)
>>> b = Date(2015, 9, 8)
>>> a.numDays(b)
6944
>>> print(a.advanceBy(7000))
2015//11/03
```

첫 구현

이제 `Date` 자료 구조를 구현한다. 객체 필드에 데이터가 지니는 연월일을 저장한다. 알기 쉬운 구현이다. 생성자와 첫 세 메소드는 이렇게 생겼다.

```
class Date():
    def __init__(self, year, month, day):
        self._year = year
        self._month = month
        self._day = day

    def year(self):
        return self._year

    def month(self):
        return self._month

    def day(self):
        return self._day
```

dayOfWeek와 numDays를 구현하고자 한다. 고로 날짜를 나날 수 혹은 날수로 변환할 수 있어야 한다. 날수는 단순히 고정해 둔 과거 날짜에서 나날을 센 수다. _toJulianDay는 날짜를 율리우스력 날수(Julian day number)로 변환하는 비공개(private) 메소드다. _toJulianDay를 사용해 아래처럼 구현할 수 있다.

```
    def dayOfWeek(self):
        jday = self._toJulianDay()
        return jday % 7
    def numDays(self, otherDate):
        return otherDate._toJulianDay() - self._toJulianDay()
```

advanceBy를 구현하려면 또 다른 비공개 함수가 필요하다. 함수는 율리우스력 날수를 Date 객체로 되돌린다.

```
    def advanceBy(self, days):
        jday = self._toJulianDay() + days
        y, m, d = _jdayToYMD(jday)
        return Date(y, m, d)
```

클라이언트 프로그램

(예제 코드 date1.py에 있는) 첫 구현 덕분에 요일 계산기를 작성하고 검증할 수 있다. 예제 코드 days1.py에서 구현을 확인할 수 있다.

구현 대안

date1.py에서 구현은 멀쩡하다. 하지만 항상 최선은 아니다. 수백만 Date 객체를 저장해야 하는 상황을 고려해 보라. Date 객체를 가능한 만큼 작게 만드는 편이 최선일 테다. 연월일을 위해 정수 세 개를 저장하기보다는 그냥 정수 하나를 저장하는 편이 낫겠다. 율리우스력 날수 하나만 저장하겠다.

date2.py는 대안을 구현한다.

```
class Date():
    def __init__(self, year, month, day):
        self.jday = _toJulianDay(year, month, day)

    def _toYMD(self):
        return _jdayToYMD(self._jday)

    def year(self):
        return self._toYMD()[0]

    def month(self):
        return self._toYMD()[1]

    def day(self):
        return self._toYMD()[2]
```

생성자는 연월일을 율리우스력 날수로 변환해야 한다. 그리고 각 접근자(accessor) 메소드는 다시 원래대로 변환해야 한다. 한편 dayOfWeek와 numDays는 이제 아주 쉽다.

```
    def dayOfWeek(self):
        return self._jday % 7

    def numDays(self, otherDate):
        return otherDate._jday - self._jday
```

Date 자료형 구현이 둘이나 있다. 둘 다 요일 계산기에 동등하게 잘 쓸 수 있다. 그리고 요일 계산기 프로그램에서 import 라인으로 둘을 간단히 바꿀 수 있다.

추상 자료형이 지니는 힘이다. 클라이언트 코드는 오직 자료형 정의에 관해 알 뿐이다. 그래서 어느 자료형 구현으로도 동등하게 잘 돌아간다.

비교

아직 구현에 비교 연산을 추가하지 않았다. 두 번째 구현에서 추가하는 편이 쉽다. 그냥 율리우스력 날수를 비교하면 끝나기 때문이다. 요술 메소드를 몇 가지 정의하면 충분하다.

```
    def __eq__(self, rhs):
        return self.jday == rhs._jday

    def __lt__(self, rhs):
        return self._jday < rhs._jday

    def __le__(self, rhs):
        return self._jday <= rhs._jday
```

__eq__ 메소드는 상등(equality)을 구현한다. 그리고 ==, != 연산자가 제대로 돌아가도록 한다. __lt__ 메소드는 "미만" 연산 <을 구현한다. __le__ 메소드는 "이하" 연산 <=을 구현한다. 파이썬은 충분히 똑똑하다. 그래서 같은 메소드를 >와 >= 연산에도 이용한다.

예외

두 Date 구현 모두 생성자에서 부당한 날짜를 처리하지 않았다.

```
>>> from date2 import Date
>>> d = Date(2017, 8, 32)
```

사용자가 부당한 날짜를 입력하면 오류 메시지를 출력하고자 한다. 하지만 Date 자료형은 묵묵히 날짜를 받아들이고 만다.

부당한 날짜를 감지하는 작업은 사실 쉽다. 연월일을 율리우스력 날짜로 변환한다. 그런 다음 다시 원래대로 변환한다. 이때 동일한 값을 얻는지 확인하면 끝난다.

하지만 생성자는 항상 Date 객체를 반환한다. 문제다. 전달인자가 부당하다고 보고할 방법은 무엇일까? 예외를 일으켜라(raise an exception).

```
    def __init__(self, year, month, day):
        jday = _toJulianDay(year, month, day)
        y, m, d = _jdayToYMD(jday)
        if y != year or m != month or d != day:
            raise ValueError("Invalid Gregorian date")
        self._jday = jday
```

부당한 날짜가 더는 먹히지 않으리다.

```
>>> from date4 import Date
>>> d = Date(2017, 8, 32)
ValueError: Invalid Gregorian date
```

그러나 이제 클라이언트 코드에서도 예외를 처리해야 한다. 아니면 사용자가 부당한 날짜를 입력했을 때 프로그램은 그냥 충돌하리다. try와 except 키워드를 이용한다. (days2.py를 보라)

```
while True:
    s = input("> ")
    f = s.split()
    try:
        if len(f) == 0:
            return
        elif len(f) == 1:
            show_weekday(f[0])
        elif len(f) == 2:
            show_difference(f[0], f[1])
        elif len(f) == 3:
            show_advance(f[0], f[1], f[2])
        else:
            print("Incorrect command")
    except ValueError as e:
        print(e)
```

try 블록 어디든 예외가 일어난다면 실행을 중지한다. 이어서 예외에 부합하는 except 절로 넘어간다. 어디든이란 try 블록에서 호출한 임의의 (가령 show_weekday와 show_weekday에 따른 Date 생성자) 함수를 포함하는 말이다. 그리고 Date 생성자가 ValueError 예외를 일으킨다. 그래서 실행은 오류 메시지 출력으로 넘어간다. 그런 다음 루프는 사용자에게 다음 요일 계산을 묻는다.

예외, 더

지난 절에서는 요일 계산기에 예외를 사용하는 방식을 살폈다. 더 자세히 살핀다.

당신이 나와 같다면 프로그램이 오류나 예외 메시지로 종결하는 광경을 여러 번 목도했을 테다. 아래는 예외 메시지 예시다.

```
>>> a = 3
>>> a // 0
ZeroDivisionError: integer division or modulo by zero
>>> s = "abc"
>>> int(s)
ValueError: invalid literal for int() with base 10: 'abc'
>>> f = open("test.text", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
>>> data = [ None ] * 10000000000
MemoryError
```

MemoryError 같은 오류는 심각한 실패를 암시한다. 프로그램이 말도 안 된다는 뜻이다.

다만 다른 예외는 프로그램에 기대하지 않았거나 비정상적인 조건이 나타났다고 암시할 따름이다. 가령 프로그램 입력 데이터에 실수를 범하면 예외가 일어날 수 있다. 다룰 수 있는 실수다. 이렇게 말한다. 예외가 잡혔다(caught). 혹은 다뤄졌다(handled).

가령 ValueError는 사용자가 수를 잘못 입력했다고 암시한다. 오류 메시지 출력 이후 새로운 입력을 요청하면 옳은 반응이리라.

`FileNotFoundError`는 열려고 한 파일이 존재하지 않는다는 뜻이다. 상황에 따라 옳은 반응이 다르다. 사용자에게 다른 파일 이름을 물어 다른 파일을 시도하는 편이 옳을 수도 있다. 아니면 그냥 다음 파일로 읽기를 건너뛰는 편이 옳을 수도 있다.

예외 잡기

아래 코드는 사용자에게 수를 요청한다. 표준 함수 `input`은 문자열을 반환한다. 그리고 `int` 함수를 이용해 정수로 변환할 필요가 있다. "abc""나 "123ab"처럼 문자열은 수가 아닐 수 있다. 그렇다면 `int` 함수는 예외를 일으킬(raise) 테다. 예외는 `try` 블록에서 핵심부를 둘러싸 잡을(catch) 수 있다. 그런 다음 예외 처리 목적으로 `except` 절을 추가할 수 있다.

```
s = input("Enter an integer> ")

try:
    x = int(s)
    print("You said %g" % x)
except ValueError:
    print("'%s' is not a number" %s)
```

`try` 블록은 멀쩡하게 실행할 수 있다. 그렇다면 `except` 절을 건너뛴다. 하지만 (직접 혹은 간접적으로 호출 받은 임의의 메소드를 포함해) `try` 블록 어딘가 예외가 발생(thrown)할 수 있다. 그렇다면 `try` 블록을 즉각 정지한다. 그런 다음 예외에 부합하는(matches) 첫 `except` 절로 넘어간다. "부합"이란 예외가 절에서 나열하는 예외 타입과 동일한 타입을 지닌다는 뜻이다.

`except` 절을 포함하는 코드를 예외 처리기(exception handler)라고 부른다.

위 예제에서 문자열이 (가령 "abc"라서) 정수를 나타내지 않는다면 `int(s)`로 인해 `ValueError` 예외가 발생한다. `try` 블록은 종결한다. (그리고 `x`에는 아무 값도 대입하지 않는다.) 그런 다음 실행은 `ValueError`에 대한 `except` 절로 넘어간다. 예제가 아래 있다.

```
$ python3 catch1.py
Enter an integer> 123a
'123a' is not a number
$ python3 catch1.py
Enter an integer> -234
You said: -234
```

예외 대(對) 오류 코드

C처럼 낡은 프로그래밍 언어에는 예외가 없다. 그래서 예외나 비 일반적인 조건은 오류 코드(error code)로 다룬다. C++는 여전히 오류 코드를 폭넓게 쓴다. C++가 C와 나누는 호환성 가운데 하나다.

`int(s)`처럼 간단하고 우아한 함수 호출은 예외가 없다면 불가능하다. `int(s)`에서는 두 가지 결과로 돌아가야 한다. 하나는 변환 성공 여부를 나타내기 위한 부울 값이다. 다른 하나는 정수 값 자체다.

고로 예외 덕분에 `int(s)`가 지니는 본질에 주목할 수 있다. 본질이란 이렇다. 문자열을 취한다. 그런 다음 수를 반환한다. 다만 예외가 지니는 진정한 위력은 다음 절에서 본모습을 드러낸다...

심원(深遠)한 예외

try 블록 내부 호출 함수 안에서 발생한 예외 또한 잡을 수 있다. 예외 관련 이점이다.

변환 예제로 돌아간다. 문자열을 별개 함수로 변환하는 판본이다.

함수 test(s)는 문자열을 더블(double)로 변환한다. 다만 그런 다음 소수 둘째 자리에서 반올림한다. 그리고 정수를 반환한다.

변환 오류가 나타날 수 있다. test(s) 내부에서 일어날 테다. 그래도 show(s) 함수 내부에서 예외로 잡을 수 있다.

```
def test(s):
    return int(100 * float(s))

def show(s):
    try:
        print(test(s))
    except ValueError:
        print("'%s' is not a number" % s)
```

아래는 두 가지 show 호출이다.

```
$ python3 -i catch2.py
>>> show("123.456")
12345
>>> show("123a456")
'123a456' is not a number
```

(float 함수 내에서) 예외가 나타나면 float 함수는 반환하지 않는다. test 함수는 반환하지 않는다. print 함수는 실행하지 않는다. 대신 except 절로 실행이 넘어간다.

보통 예외가 나타나면 (예외가 일어난다(raised)고 하는데) 실행이 지니는 평범한 흐름이 방해받는다. 그리고 가장 가까운 (최심부(最深部, innermost) 혹은 최신) try 블록으로 넘어간다. 다시 말해 부합하는 예외 유형이 잡힌(caught) (즉 올바른 타입을 지니는 예외 처리기가 존재하는) try 블록으로 넘어간다.

아래 프로그램을 고찰하며 더 세세하게 살핀다.

```
def f(n):
    print("Starting f(%d) ... " % n)
    g(n)
    print("Ending f(%d) ... " % n)

def g(n):
    print("Starting g(%d) ... " % n)
    m = 100 // n
    print("The result is %d" % m)
    print("Ending g(%d) ... " % n)

def main():
    while True:
        s = input("Enter a number> ")
        if s.strip() == "":
            return
        try:
            print("Beginning of try block")
            n = int(s)
            f(n)
```

```
            print("End of try block")
        except ValueError:
            print("Please enter a number!")
        except ZeroDivisionError:
            print("I can't handle this value!")

main()
```

실행본이다.

```
$ python3 except1.py
Enter a number> 12
Beginning of try block
Starting f(12) ...
Starting g(12) ...
The result is 8
Ending g(12) ...
Ending f(12) ...
End of try block
Enter a number> abc
Beginning of try block
Please enter a number!
Enter a number> 0
Beginning of try block
Starting f(0) ...
Starting g(0) ...
I can't handle this value!
```

"12" 입력값에 대해 `try` 블록이 지니는 시작과 끝을 관찰했다. 또 `f`, `g` 함수를 관찰했다. "abc" 입력값에는 `int(s)` 함수 때문에 예외가 발생했다. 그래서 `f` 함수가 호출하지 않았다. `0` 입력값에는 `g` 함수 안으로 이루어진 나눗셈 때문에 `ZeroDivisionError`가 발생했다. 보시다시피 실행은 예외 처리기에서 즉각(immediately) 넘어간다. `g`, `f` 함수나 `try` 블록은 완수하지 않는다.

예외 일으키기

지금껏 라이브러리 함수에서 일어난 예외만 잡았을 따름이다. 하지만 직접 예외를 일으킬 수도 있다. 가령 함수 `g(n)`이 비(非)음수만 처리해야 한다고 가정하자. 음수 전달인자에는 `ValueError`가 발생하도록 해서 보장할 수 있는 처리다. 전체 스크립트는 이렇게 생겼다.

```
def f(n):
    print("Starting f(%d) ... " % n)
    g(n)
    print("Ending f(%d) ... " % n)

def g(n):
    print("Starting g(%d) ... " % n)
    if n < 0:
        raise ValueError("Cannot handle negative numbers")
    print("The result if %d" % n)
    print("Ending g(%d) ... " % n)

def main():
    while True:
        s = input("Enter a number> ")
        if s. strip() == "":
            return
        try:
            print("Beginning of try block")
            n = int(s)
```

```
            f(n)
            print("End of try block)
        except ValueError as e:
            print("Cannot handle this input: %s" % e)

main()
```

예외 처리기 내부 예외에 속하는 메시지를 이용하는 방법을 보이는 예제다.

예외란 객체다. 그리고 여타 객체와 마찬가지로 생성자 호출을 통해 생성한다. 유의하라. 예외 메시지는 예외 객체 필드로 저장한다.

같은 스크립트를 다른 입력으로 다시 실행해 본다.

```
$ python3 except2.py
Enter a number> 123
Beginning of try block
Starting f(123) ...
Starting g(123) ...
The result is 123
Ending g(123) ...
Ending f(123) ...
End of try block
Enter a number> abc
Beginning of try block
Cannot handle this input: invalid literal for int() with base 10: 'abc'
Enter a number> -123
Beginning of try block
Starting f(-123) ...
Starting g(-123) ...
Cannot handle this input: Cannot handle negative numbers
```

예외는 입력 데이터에서 오류를 감지할 때 자주 쓴다.

프로그램 내부 적합한 장소에서 예외를 잡을 수 있다. 그런 다음 오류 메시지를 출력한다. 아니면 문제를 다른 방식으로 처리한다.