김범수

Python 시작하기2

Basic of python

By POSCAT

Contents

리스트 심화

- 전 강의에서는 리스트를 만들고, sort 함수를 정렬해보는 것에 대해서만 만들어보았습니다. 이번에는 전 강의에서 리스트에 대해 다루지 못했던 내용들을 조금 더 다뤄보겠습니다.
- 리스트는 list[인덱스]와 같은 방법으로 리스트의 원소에 접근할 수 있습니다. 하지만, 인덱스 가 list의 길이 범위를 넘어간다면, 인덱스가 음수라면 어떤 일이 일어날까요?
- 아래 코드처럼 리스트의 인덱스가 범위를 넘어가면 인덱스 에러가 뜨면서 오류가 발생하는 것을 볼 수 있습니다.

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

print(List[3])
print(List[4])
print(List[5])
```

```
4
5
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print(List[5])
IndexError: list index out of range
```

리스트 심화

- 전 강의의 연습문제2에서 음수 인덱스에 대해서 살짝 언급하였었습니다. List[-x]는 뒤에서 x번째 원소에 접근할 때 사용합니다. 맨 끝 원소, 맨 끝에서 2번째 원소, 이런 식으로 접근하고 싶을 때 사용합니다. 마찬가지로, List의 범위를 벗어나면 index 에러가 발생합니다.

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

print(List[-1])
print(List[-2])
print(List[-3])
print(List[-4])
print(List[-5])
print(List[-6])
```

```
5
4
3
2
1
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print(List[-6])
IndexError: list index out of range
```

리스트 심화

- 인덱스를 사용하면 하나의 원소에 접근할 수 있습니다. 여러 개의 원소에 한 번에 접근하고 싶다면 어떻게 하면 될까요?
  list[시작 인덱스:끝 인덱스] 형식으로 여러 개의 원소를 한 번에 접근할 수 있습니다. 이를 리스트 슬라이싱이라고 합니다.
  3~5번째 줄을 보면, [시작 인덱스 ~ 끝 인덱스)에 해당하는 리스트가 만들어지는 것을 볼 수 있습니다.
- 6번째 줄: 인덱스에 음수가 있어도 잘 처리하는 것을 볼 수 있습니다.
- 7번째 줄: 인덱스 범위가 시작 인덱스가 끝 인덱스보다 커도 에러가 나지 않고 빈 리스트가 됩니다.
- 8~9번째 줄: 하나의 원소에 대입하는 것처럼 여러 개의 원소에 대입하는 것도 잘 되는 것을 볼 수 있습니다.

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

```
[2]
[2, 3]
[1, 2]
[2, 3, 4]
[]
[1, 6, 7, 4, 5]
```

리스트 심화

- 리스트 슬라이싱에서 시작 인덱스와 끝 인덱스를 생략하면 각각 0, 리스트의 길이가 됩니다.
- 만약 2, 4, 6… 번째 원소, 3,6,9… 번째 원소 이렇게 띄엄띄엄 간격을 가지고 접근을 하고 싶다면, list[시작:끝:간격]의 형식으로 간격을 가지고 접근할 수 있습니다. 또한, 간격에 음수를 넣으면 역순으로 접근하게 됩니다.

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

print(List[:3])
print(List[1:])
print(List[:])

print(List[::2])
print(List[::-1])
```

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

리스트 심화

- 더하기, 곱하기를 사용하여 두 리스트를 합치거나 한 리스트를 반복할 수 있습니다.

```
List1 = [1, 2, 3, 4]
List2 = [5, 6, 7, 8]
print(List1 + List2)
print(List1 * 3)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

- 지금까지는 리스트의 값을 바꾸는 법만 배웠습니다. 하지만, 리스트에 새로운 원소를 넣고 싶다면, 아니면 원소를 삭제하고 싶다면 어떻게 할까요?
- 리스트의 함수를 이용하여 리스트 원소를 추가, 삭제할 수 있습니다. 저번 강의에서는 리스트의 함수, sort()를 사용하여 리스트를 정렬하였죠. 사실 리스트에는 더 많은 함수가 존재합니다. 리스트에 존재하는 함수 중, 우리가 많이 사용할 함수에 대해서 알아봅시다.

리스트 심화

- 먼저 리스트에 원소를 추가할 때는 append, insert를 사용합니다. append는 리스트의 가장 마지막에 새로운 원소를 추가하는 함수이고, insert는 내가 원하는 자리에 원소를 넣을 수 있습니다.

```
List1 = [1, 2, 3, 4]
List2 = [1, 2, 3, 4]

List1.append(5)
List2.insert(1, 5)

print(List1)
print(List2)
```

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

- append는 끝에만 원소를 추가하고, insert는 모든 위치에 원소를 추가할 수 있는데 왜 append하는 함수가 존재할까요? 실행속도를 비교하면, append가 insert보다 속도가 빠르다는 것을 알 수 있습니다. 따라서 가능하다면 insert보다는 append를 사용하는 것이 효율적입니다.

```
1  List1 = []
2  for i in range(10000000):
3      List1.append(i)
4
[Finished in 2.3s]
```

```
1  List1 = []
2  for i in range(10000000):
3      List1.insert(len(List1), i)
4
[Finished in 3.7s]
```

리스트 심화

- 리스트 원소의 삭제는 del 함수를 사용해서 할 수 있습니다. del은 다른 함수와는 다르게 “del 객채” 꼴로 사용됩니다. 따라서 del list[index] 꼴로 index번 원소를 삭제할 수 있습니다.

```
List1 = []
List2 = []
for i in range(10):
    List1.append(i)
    List2.append(i)
del List1[5]
del List2[3:6]
print(List1)
print(List2)
```

```
[0, 1, 2, 3, 4, 6, 7, 8, 9]
[0, 1, 2, 6, 7, 8, 9]
```

- 이외에도 count, index 등의 함수가 존재하지만, 시간상 모든 내용을 다루기는 힘드므로 나중에 사용하게 될 때 설명하겠습니다.

리스트 심화

- 지금까지는 리스트 안에 원소를 넣어보았습니다. 그렇다면 리스트 안에 리스트를 넣을 수도 있을까요? 물론 가능합니다.

```
List1 = [1, 2, 3, 4]
List2 = [5, 6, 7, 8, 9]
List_in_List = [List1, List2]
print(List_in_List)
```

```
[[1, 2, 3, 4], [5, 6, 7, 8, 9]]
```

- 아래 코드만 보면 List는 [[1, 2], [1, 2, 3]]이 되어야 할 것 같은데 왜 아래처럼 출력이 될까요? 그 이유는 python은 리스트를 복사할 때, list 전체를 복사하지 않고, list의 id만 복사하기 때문입니다. 따라서 List에는 List1의 id가 2개 들어있는 것이죠. 따라서 List1을 바꾸면, List 내부의 원소도 바뀌는 것처럼 보이는 것입니다. 따라서 우리가 원하는 대로 List를 만들고 싶다면, 리스트를 복사하는 copy라는 모듈을 사용해야 합니다. copy 모듈 사용법은 뒤에서 소개하겠습니다.

```
List1 = [1, 2]
List = []
List.append(List1)
List1.append(3)
List.append(List1)
print(List)
```

```
[[1, 2, 3], [1, 2, 3]]
```

튜플, 딕셔너리

- 튜플은 리스트와는 달리 대괄호 []가 아니라 소괄호 ()를 이용합니다. 튜플은 리스트와 모두 동일하지만, 튜플은 리스트와는 달리 원소의 추가, 삭제, 수정이 불가능합니다. 그 이외는 리스트와 거의 동일하다고 보면 됩니다.
- 리스트는 list[index] = 원소 꼴일 때, 딕셔너리는 dic[key] = value 꼴입니다. 즉, 리스트는 원소만 저장할 때, 딕셔너리는 key와 value를 한 세트로 저장할 수 있습니다. 딕셔너리는 중괄호를 사용하여 정의합니다.
- 클래스를 배운 후, 딕셔너리를 리스트로 구현해보는 연습문제를 진행하면서 딕셔너리에 대한 자세한 내용을 뒤에서 소개하겠습니다.

```
Tuple = (1, 2, 3)
dic = {"apple" : "red", "banana" : "yellow"}
print(Tuple[1])
print(dic["banana"])
```

```
2
yellow
```

클래스

- 저번 주에는 보다 편하게 프로그램을 짜기 위해 함수를 사용하였습니다. 하지만 함수만으로는 어딘가 부족합니다. 한 번 계산기를 만드는 예제를 풀어보면서 함수의 한계점을 알아봅시다.
- 먼저 계산기의 가장 기본적인 사칙연산은 함수를 통해 구현할 수 있습니다. 하지만, 계산기에는 가장 마지막 계산의 결과를 저장하는 기능이 있습니다. 마지막 결과를 result라는 변수에 저장해서 간단한 계산기를 한 번 구현해보겠습니다.
- 그러면 오른쪽 코드처럼 result에 사칙연산 결과를 저장하는 기능을 함수에 넣어서 구현할 수 있습니다.
- 여기까지는 큰 문제가 없습니다. 하지만 계산기를 하나 더 만들려면 어떻게 해야 할까요? 두 번째 계산기의 계산 결과를 저장할 result2라는 변수를 하나 더 만들고, 계산 결과를 result2에 저장하는 함수를 또 만들어야 할 것입니다. 계산기를 5개 만든다면, 10개 만든다면? 그러면 코드는 쓸데없이 엄청 길어질 것입니다.

```
result = 0

def add(x, y):
    result = x + y
    return result

def sub(x, y):
    result = x - y
    return result

def mul(x, y):
    result = x * y
    return result

def div(x, y):
    result = x / y
    return result
```

클래스

```
class 클래스 이름:
    def 매서드(self, 매개변수):
        코드
```

- 클래스는 오른쪽의 형태로 정의할 수 있습니다.
- Self는 말 그대로 자기 자신을 가리킵니다. 즉 매개변수 뿐만 아니라 자기자신도 전달한다는 뜻입니다. 지금 당장 이해하기는 어려우니 계산기를 클래스로 만들어보면서 이해해봅시다.
- 아까 말했듯이 클래스를 만든다는 것은 설계도를 만든다는 것과 비슷합니다. 계산기 설계도 안에는 최근 계산 결과를 저장할 메모리와 사칙연산을 진행하는 함수(매서드) 4개가 필요합니다.
- 먼저 메모리를 정의합니다. 자신의 메모리라는 의미로, self.memory로 정의할 수 있습니다.
- 그 후, 위 형식대로, 사칙연산 함수를 만듭니다. 각 매서드는 매개변수로 x, y를 받고, self.memory에 계산 결과를 저장한 후, 계산 결과를 반환합니다.

클래스

- 완성된 계산기 클래스는 오른쪽과 같습니다.
- 앞으로 만들 클래스도 이것과 비슷한 틀로 만들게 됩니다. 클래스의 형식에 익숙해지도록 합시다.

```
class mini_calculator:

    def add(self, x, y):
        self.memory = x + y
        return self.memory

    def sub(self, x, y):
        self.memory = x - y
        return self.memory

    def mul(self, x, y):
        self.memory = x * y
        return self.memory

    def div(self, x, y):
        self.memory = x - y
        return self.memory
```

클래스

- 지금까지는 계산기의 설계도를 만들었습니다. 이제는 설계도로 계산기를 만들고 사용해보겠습니다.

```
calc = mini_calculator()
print(calc.add(1, 2))
print(calc.mul(3, 4))
print(calc.memory)
```

```
3
12
12
```

- 이런 방식으로 우리가 만든 설계도(클래스)로 계산기를 만들고 사용할 수 있습니다. 위 코드에서 클래스로 만든 calc라는 계산기를 인스턴스라고 합니다. 후에 강화학습에서 모델을 만들 때 class로 모델을 만들게 되니 알아둡시다.
- 처음 계산기를 만들고, 사칙연산을 진행하지 않은 채로 memory를 출력하려고 하면 에러가 발생합니다. 이는 사칙연산을 진행할 때, memory 변수를 만드는 데, 변수를 만들기 전에 memory를 출력하면 아직 memory 변수가 정의되지 않았으므로 에러가 발생합니다.
- 이를 고치려면 계산기를 만드는 순간에 memory라는 변수를 만들어야 합니다. 어떻게 할 수 있을까요?

클래스

- 파이썬에는 특별한 메서드들이 존재합니다. 그 중 가장 많이 쓰이는 __init__ 라는 메서드에 대해 알아보겠습니다.
- __init__ 메서드는 클래스로 인스턴스를 만들 때 한 번 실행되는 메서드입니다. 전의 계산기에서 __init__ 메서드를 만들어 memory를 0으로 초기화해보죠.
- 아까와는 달리 에러가 나지 않고, memory가 잘 출력됩니다.
- 만약 memory를 0이 아니라 자신이 원하는 값으로 초기화 하고 싶다면 어떻게 해야 할까요? __init__함수에서 초기화할 값을 매개변수로 받아 인스턴스를 만들 때, 매개변수를 통해 원하는 값으로 초기화할 수 있습니다.
- 이외에도 다양한 특별한 메서드가 존재합니다.

```
class mini_calculator:

    def __init__(self):
        self.memory = 0
calc = mini_calculator()
print(calc.memory)
```

```
0
```

```
class mini_calculator:

    def __init__(self, x):
        self.memory = x
calc = mini_calculator(10)
print(calc.memory)
```

```
10
```

클래스

- 지금까지 우리는 클래스로 계산기를 만드는 법을 배웠습니다. 하지만, 계산기에 사칙연산 말고 제곱, 나머지 같은 연산을 추가하거나, 나누기 함수가 나눗셈 결과가 아닌 몫을 반환하도록 바꾸고 싶다면 어떻게 해야 할까요? 클래스 자체를 직접 바꾸는 방법도 있지만, 클래스가 복잡하다면 일일이 바꾸는 것도 매우 힘들겠죠. 계산기 클래스의 메서드를 전부 가져오고, 필요한 메서드만 수정, 추가하는 방식으로 하면 편리할 것입니다. 이는 클래스의 상속을 통해 사용 가능합니다.
- 클래스의 상속은 오른쪽의 형식으로 이루어지게 됩니다.

```
class 클래스 이름(상속받을 클래스):
    클래스 내용
```

- 보통 상속받을 클래스는 부모 클래스, 상속받은 클래스는 자식 클래스라고 표현합니다.
- 우리가 만든 mini_calculator를 상속받아 새로운 계산기 클래스 calculator를 만들고, 제곱, 나머지 연산을 추가하고, 나누기 연산을 몫 계산으로 변경하는 예제를 보고 상속에 대해 이해해보겠습니다.

클래스

- 오른쪽의 코드를 보면, calculator class를 mini_calculator를 상속받아 만들어졌음을 알 수 있습니다.
- pow, mod라는 새로운 메서드를 만들고, div 메서드를 몫 연산으로 바꾸었습니다. 이때 div 메서드처럼 부모 클래스의 메서드를 자식 클래스에서 재정의하는 것을 오버라이딩이라 합니다.
- 계산기의 memory가 0으로 초기화되는 것을 통해 부모 클래스에서 정의한 init 함수가 자식 클래스에서도 작동함을 알 수 있습니다.

```python
class calculator(mini_calculator):

    def pow(self, x, y):
        self.memory = x ** y
        return self.memory

    def mod(self, x, y):
        self.memory = x % y
        return self.memory

    def div(self, x, y):
        self.memory = x // y;
        return self.memory

calc = calculator(0)
print(calc.memory)
print(calc.pow(3, 3))
print(calc.mul(3, 3))
print(calc.div(5, 3))
```

```
0
27
9
1

***Repl Closed***
```

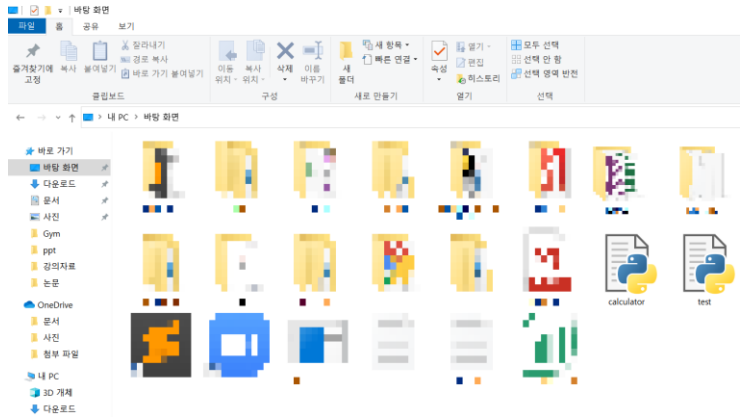
모듈

- 지금까지는 클래스가 무엇인지 배우고, 클래스를 이용하여 간단한 계산기를 만들어보았습니다. 이번에는 하나의 파일이 아니라 여러 개의 파일로 코딩하는 법을 배워보겠습니다.
- 모듈이란 함수나 변수, 클래스 등을 모아둔 파일입니다. 지금까지 우리가 한 코딩들도 하나의 모듈이라고 할 수 있습니다. 지금까지는 하나의 파일이 코딩했지만, 코딩 내용이 너무 복잡하거나, 다른 사람이 만들어놓은 함수를 사용할 때에는 하나의 파일로 하기에는 불편함이 많습니다. 따라서 여러 모듈로 코딩하는 기법에 대하여 알아봅시다.
- 아까 만들었던 계산기를 모듈화 해봅시다.

모듈

- 먼저 ctrl+n을 눌러 새 파일을 만들고, 지금까지 만든 계산기 class 2개를 calculator.py라는 이름으로 저장해봅니다. 이제 우리는 calculator라는 모듈을 만든 것입니다.
- 모듈을 사용하기 위해서는 “import 모듈명” 형태로 모듈을 불러와야 합니다. 우리가 만든 계산기를 import calculator로 불러온 후 사용해봅시다.
- 아래 코드는 calculator.py라는 파일에서 calculator class를 가져와 add 연산을 사용하는 예제입니다.

```
1   import calculator
2
3   calc = calculator.calculator(0)
4   print(calc.add(1, 2))
```



자신이 만든 모듈을 사용하기 위해서는 자신이 코딩하는 파일과 같은 경로에 모듈이 존재해야 합니다.

모듈

- 지금은 class를 불러오는 것이라 별로 불편하지 않지만, 만약 calculator 모듈에 add라는 함수를 정의했고, add라는 함수를 사용하고자 한다면 어떻게 쓸 수 있을까요? [Code1]처럼 쓸 수 있겠지만, calculator모듈 이름을 계속 적는 것은 코드 가독성도 떨어지고 상당히 귀찮은 일입니다. 따라서 [Code2] 처럼 “import A as B”의 형태로, A 모듈을 불러오지만, B라는 이름으로 사용하겠다고 할 수 있고, [Code3]처럼 “from A import B”의 형태로, A의 모듈에서 B라는 함수만 가져와서 사용할 수도 있습니다.
- 주로 한 모듈에서 여러 종류의 함수를 사용할 때는 [Code2]와 같은 형태로 사용되고, 모듈의 특정한 한 함수만 사용할 때는 [Code3]과 같은 형태로 사용됩니다.

```
import calculator

c = calculator.add(1, 2)
```

[Code1]

```
import calculator as calc

c = calc.add(1, 2)
```

[Code2]

```
from calculator import add

c = add(1, 2)
```

[Code3]

파이썬 라이브러리

- 앞서 말했던 모듈은 자신과 같은 경로인 파일 이외에도, 파이썬에서 지정한 특별한 경로에 있는 경로에 있는 파일도 불러올 수 있습니다. 이 경로에는 파이썬에서 기본적으로 제공해주는 모듈들이 저장되어 있고, pip 명령어를 사용하여 추가적으로 모듈을 설치할 수 있습니다. (pip에 관한 내용은 후에 pytorch를 다룰 때 설명드리겠습니다.)
- 이처럼 파이썬에서 기본적으로 제공해주는 모듈들을 파이썬 라이브러리라고 합니다. 강화학습에서 사용될 몇몇 파이썬 라이브러리를 익혀봅시다.

파이썬 라이브러리

- 모듈 time에는 시간에 대한 다양한 모듈이 들어있습니다. 현재 시각을 알아낼 수도 있고, 다양한 형식으로 시각을 출력할 수도 있습니다. 이 중에서 우리는 이 모듈의 sleep함수를 활용합니다. sleep(x)함수는 x초 동안 기다리는 함수로, 일정한 시간 간격으로 반복문 내용을 출력해보고 싶을 때 사용합니다.

```python
from time import sleep

print("Start")
sleep(2)
print("End")
```

```
Start
End
[Finished in 2.5s]
```

컴파일 시간 때문에 0.5s정도 더 걸립니다.

- 모듈 random에는 다양한 난수(랜덤한 수)관련 함수가 들어있습니다.
  random 모듈의 random()함수는 0과 1사이의 난수를 반환합니다.
  randint(a, b)는 a와 b사이의 난수(정수)를 반환합니다.
  shuffle(list)는 list의 항목을 무작위로 섞습니다.
  choice(list)는 list에서 무작위로 하나를 선택하여 돌려줍니다.

```python
import random

List = ["apple", "orange", "banana", "tomato"]

print(random.random())
print(random.randint(1, 10))
print(random.choice(List))
random.shuffle(List)
print(List)
```

```
0.7940682856546896
4
orange
['orange', 'tomato', 'apple', 'banana']

***Repl Closed***
```

파이썬 라이브러리

- 이전에, list의 원소로 list를 추가할 때는 list값 모두가 아니라 list의 id가 추가되므로 주의해야 한다고 배웠습니다. 그렇다면, list 전체를 list의 원소로 추가하려면 어떻게 해야할까요? 모듈 copy를 사용하면 list값 전체를 복사하고, 복사한 list의 id를 반환합니다. copy 모듈의 copy함수를 사용하여 이전의 예제를 다시 짜보면 아래처럼 잘 작동하는 것을 볼 수 있습니다.
- copy를 사용하여 새로운 list를 만들고, 그 리스트를 LList에 추가하므로 기존의 List값이 바뀌어도 LList내부의 원소의 값이 바뀌지 않음을 알 수 있습니다.

```
1   from copy import copy
2
3   List = [1, 2, 3, 4]
4   LList = []
5   LList.append(copy(List))
6   List.append(5)
7   LList.append(copy(List))
8   List.append(6)
9
10  print(LList)
```

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

***Repl Closed***
```

연습문제1

- calculator 모듈을 조금 더 발전시켜 봅시다. 지금까지는 최근의 계산 결과만 저장하였습니다. 하지만 계산 결과 말고 계산 식 전체를 저장해보는 건 어떨까요? 그리고 최근 계산 식이 아니라 최근 10회의 계산 식을 저장해볼까요?
- 연산이 너무 많으면 귀찮아지니 사칙연산(덧셈, 뺄셈, 곱셈, 나눗셈)만 지원하는 계산기를 만들어봅니다.
- 이번 연습문제에서는 __getitem__, __str__이라는 특수한 메서드를 이용합니다.
- __getitem__ 메서드를 이용해 x번째 계산 식을 가져옵니다.
- __str__ 메서드를 이용해 최근 10번의 계산식을 모두 출력할 수 있도록 합니다.

연습문제1

- 지금까지 배운 내용을 바탕으로 오른쪽 코드처럼 계산기를 구현합니다.
- 이전에는 메모리에 값만 저장했지만, 이번엔 여러 개의 계산식을 저장하므로, 리스트로 정의합니다.
- update_memory 함수를 추가하여 memory의 크기가 10보다 커지면 가장 마지막 메모리를 제거해줍니다.
- format 함수를 이용해 형식에 맞춰 계산 식을 string 형으로 저장합니다.
- 자세한 설명은 github 코드에 주석으로 설명하였으니 참고하면 될 것 같습니다.

calculator.py

```python
class calculator:

    def __init__(self):
        self.memory = []

    def update_memory(self, string):
        self.memory.append(string)
        if len(self.memory) == 11:
            del self.memory[0]

    def add(self, x, y):
        result = x + y
        self.update_memory("{} + {} = {}".format(x, y, result))
        return result

    def sub(self, x, y):
        result = x - y
        self.update_memory("{} - {} = {}".format(x, y, result))
        return result

    def mul(self, x, y):
        result = x * y
        self.update_memory("{} * {} = {}".format(x, y, result))
        return result

    def add(self, x, y):
        result = x / y
        self.update_memory("{} / {} = {}".format(x, y, result))
        return result
```

연습문제1

- 이번 계산기는 여러 개의 log를 가지고 있습니다. 물론 calculator.memory[x]와 같이 x번째 log에 접근할 수는 있겠지만, calculator[x]와 같이 바로 log에 접근할 수는 없을까요? 물론 가능합니다. __getitem__라는 특수한 메서드로 접근할 수 있습니다.
- calculator.__getitem__(x)는 calculator[x]와 동등합니다. 따라서 __getitem__(self, x)를 구현하여 계산기 class에 index를 매길 수 있습니다.
- 만약 x가 메모리 범위를 초과하면 계산 식 대신 “Out of Range”를 반환합니다. (x는 0 또는 양수만 허용합니다.)

calculator.py

```
31      def __getitem__(self, x):
32          if x < 0 or x >= len(self.memory):
33              return "Out of range"
34          return self.memory[x]
```

연습문제1

- print(calculator)를 하면 어떻게 될까요? python에서는 인스턴스가 저장된 id가 출력됩니다. 하지만 이는 우리한텐 의미가 없습니다.
- 우리는 print(calculator)를 하면, calculator의 모든 메모리가 출력되도록 하고 싶습니다. 물론 calculator의 모든 메모리를 출력하는 함수를 따로 만들 수 있지만, 이번에는 __str__라는 메서드를 이용해서 출력해봅시다.
- 0. 0번째 계산 식
  1. 1번째 계산 식 …의 형식으로 출력하도록 합니다.

calculator.py

```
36      def __str__(self):
37          ret = "".
38          for x in range(len(self.memory)):
39              ret += "{}. {}\n".format(x, self.memory[x])
40          return ret
```

연습문제1

- 대부분의 공학용 계산기에는 편의를 위해 값을 저장하는 공간이 존재합니다. 우리도 값을 저장하는 공간(캐시)을 만들어봅시다.
- calc[index] = value 꼴로 캐시를 저장할 것입니다. getitem과 유사하게 __setitem__(self, index, value) 메서드를 이용해서 정의할 수 있습니다. (보통은 index를 key라 합니다.)
- 만약 int형이나 float형이 아니면 값을 저장하지 않습니다. (type함수를 이용해 확인할 수 있습니다.)

```
3       def __init__(self):
4
5           self.memory = []
6           self.cache = [0, 0, 0, 0, 0]
```

```
45      def __setitem__(self, key, value):
46      if str(type(key)) != "<class 'int'>" or key < 0 or key >= 5:
47          return
48      if str(type(value)) != "<class 'int'>" and str(type(value)) != "<class 'float'>":
49          return
50      self.cache[key] = value
```

연습문제 1

- 새 파일을 만들어 calculator 모듈을 불러오고 잘 작동되는 지 테스트해봅시다. 모듈이 잘 만들어졌다면 오른쪽 그림처럼 작동할 것입니다.

practice1.py

```python
import calculator

calc = calculator.calculator()
print(calc.add(5, 2))
print(calc.sub(5, 2))
print(calc.mul(5, 2))
print(calc.div(5, 2))
print(calc[1])
print(calc[4])
print(calc)
calc[1] = 2.345
calc[2] = "1234"
print(calc.cache)
```

sublimeREPL

```
7
3
10
2.5
5 * 2 = 10
Out of Range
0. 5 / 2 = 2.5
1. 5 * 2 = 10
2. 5 - 2 = 3
3. 5 + 2 = 7

[0, 2.345, 0, 0, 0]

***Repl Closed***
```

연습문제2

- 연습문제 1에서는 __getitem__, __setitem__, __str__ 메서드에 대해 배워보았습니다. 강화학습 자체에서는 잘 사용할 일이 없지만 python의 class에서는 중요한 메서드라 한 번 소개해보았습니다.
- 연습문제 2에서는 딕셔너리를 직접 구현해봅니다. dictionary.py 파일에 Dictionary class를 만들어서 구현합니다.
- 딕셔너리는 리스트를 이용해 구현합니다. (당연하지만, 딕셔너리를 사용하지 않고 구현해봅니다.)
- 여기서는 key를 저장하는 list와 value를 저장하는 list, 2개의 list로 구현해보지만, [key, value] 리스트를 원소로 가지는 하나의 list도 구현할 수도 있습니다.

연습문제2

- __init__ 메서드에서 key와 value를 저장할 빈 list를 만듭니다.
- 테스트를 하기 쉽게 __str__ 메서드를 먼저 구현합니다.
- {(0번째 key : 0번 value), (1번째 key : 1번째 value), ...}의 형식으로 출력합니다.
- 만약 정상적으로 구현하였다면 처음에는 빈 딕셔너리이므로 {} 가 출력됩니다.

dictionary.py

```python
class Dictionary:

    def __init__(self):
        self.key_list = []
        self.value_list = []

    def __str__(self):
        ret = "{"
        for x in range(len(self.key_list)):
            ret += "({} : {})".format(self.key_list[x], self.value_list[x])
            if x != len(self.key_list) - 1:
                ret += ", "
        ret += "}"
        return ret
```

practice2.py

```python
import dictionary

dic = dictionary.Dictionary()
print(dic)
```

sublimeREPL

```
{}

***Repl Closed***
```

연습문제2

- 딕셔너리는 dic[“one”] = 1과 같은 형식으로 값을 추가합니다. (key는 “one”, value는 “1”이 됩니다.)
- 만약 “one” : 1 이 존재하는 데, dic[“one”] = 2를 한다면, 새로운 (key : value)가 추가되는 것이 아니라, (“one” : 2) 로 해당 key의 value를 수정해야 합니다.
- __setitem__ 메서드를 이용해 구현합니다.
- list.index(x)함수는 x가 저장되어 있는 index를 반환합니다. 만약 list에 x가 존재하지 않는다면 에러가 발생합니다.
- list.index(x)의 에러를 막기 위해 x in list 를 이용하여 list에 x가 있는 지 확인할 수 있습니다.

```
27 ▼    def __setitem__(self, key, value):
28 ▼        if key in self.key_list:
29              idx = self.key_list.index(key)
30              self.value_list[idx] = value
31 ▼        else:
32              self.key_list.append(key)
33              self.value_list.append(value)
```

연습문제2

- dic[key]로 value에 접근할 수 있습니다.
- 딕셔너리는 pop 함수를 통해 값을 삭제할 수 있습니다. dic.pop(key) 꼴로 (key, value)를 삭제합니다. 이 때 pop함수는 value를 반환합니다.
- dic.keys(), dics.values() 함수는 key리스트, value 리스트를 반환합니다. (이 때, 단순히 인스턴스의 key_list와 value_list를 반환하면 원래 값이 훼손될 위험이 있으니 copy로 복사하여 반환합니다.)
- dic.items()는 (key, value)로 이루어진 튜플 리스트를 반환합니다.
- dic.clear()는 딕셔너리를 비웁니다.
- 위의 내용을 모두 구현해봅시다.

연습문제2

- 만약 잘 구현되었다면, 아래와 같은 테스트 결과가 나올 것입니다.

practice2.py

```python
import dictionary

dic = dictionary.Dictionary()
dic["one"] = 0
dic["one"] = 1
dic["two"] = 2
dic["three"] = 3
print(dic)
print(dic["two"])
print(dic.keys())
print(dic.values())
print(dic.items())
dic.pop("two")
print(dic)
dic.clear()
dic["four"] = 4
dic["five"] = 5
print(dic)
```

sublimeREPL

```
{(one : 1), (two : 2), (three : 3)}
2
['one', 'two', 'three']
[1, 2, 3]
[('one', 1), ('two', 2), ('three', 3)]
{(one : 1), (three : 3)}
{(four : 4), (five : 5)}

***Repl Closed***
```