01. Packing, Unpacking

https://python.bakyeono.net/chapter-5-5.html

*변수 로 함수의 매개변수에 전달하면 그 변수 시퀀스의 요소를 하나씩 꺼내어 전달이 된다. 이 때가 'Unpacking'

반면에 함수를 정의할 때 매개변수를 *변수로 선언하면 입력되는 여러 변수들을 리스트로 만들어 전달한다. 이 때가 'Packing'이다.

*변수와 **변수 차이는 전자는 리스트, 후자는 딕셔너리를 받거나 전달한다고 생각하면 된다.

02. filter 함수

https://wikidocs.net/32#filter

걸러내는 함수로 첫 번째 인수로 함수 이름, 두 번째 인수로 그 함수에 차례로 들어갈 반복 가능한 자료형을 받는다. 첫 번째 인수의 함수에 두 번째 인수의 요소가 하나씩 대입이 되고 그 함수가 `True`로 값이 리턴이 되면 대입한 값을 묶어주는 역할을 하고 있다. 그 결과를 리스트로 사용하고 싶으면 `filter`의 결과물을 리스트로 형변환해서 사용해야한다.

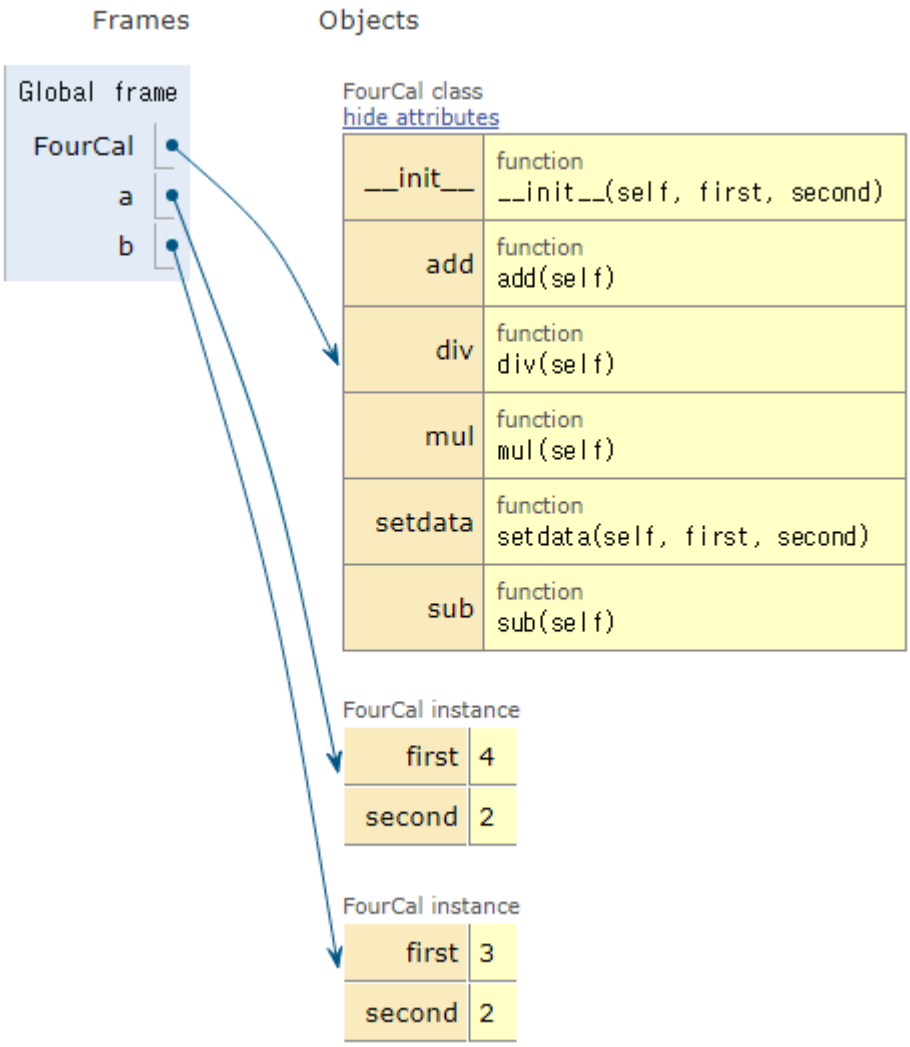
03. 재귀함수

재귀함수의 궁극적인 목적은 문제를 작게 만들어 해결한다는 점이다.

04. class

```
class FourCal:
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def setdata(self, first, second):
        self.first = first
        self.second = second
    def add(self):
        result =  self.first + self.second
        return result
    def mul(self):
        result = self.first * self.second
        return result
    def sub(self):
        result = self.first - self.second
        return result
    def div(self):
        result = self.first / self.second
        return result
```

```
a  = FourCal(4,2)
b = FourCal(3,2)
```



결과

`self` 매개변수는 별도로 받지않는다. `self`는 해당 객체를 가리키며 `self.first`와 같은 경우 해당 객체에 객체변수 `first`를 생성하라는 의미이며 객체 `a`와 객체 `b`와 서로 별개로 생성된 것을 위와 같이 확인할 수 있다. `add`이하로 선언된 함수들에 매개변수가 `self`로 받아드리는데 `self`의 객체변수를 활용하기위해 위와 같이 사용이된다.

`__init__` 메소드는 생성자 (Constructor)를 선언할 때 사용한다. 이것을 선언하면 `FourCal`에 매개변수를 대입하면 `__init__` 메서드로 선언한 대로 작동이 된다. 만약 생성자를 선언했는데 매개변수를 대입하지 않으면 오류가 난다.

```
class MoreFourCal(FourCal):
        pass
```

상속은 위와 같이 매개변수 자리에 상속할 클래스 이름을 넣으면 된다. 위와 같은 경우 `FourCal`에 있는 메소드 등을 사용할 수 있다.

05. 모듈

```
from 모듈이름 import 모듈함수
import 모듈이름 # from 모듈이름 import *와 같다.
```

위와 같이 두 가지 방법으로 모듈을 불러올 수 있다 . 직접 사용할 모듈의 함수만 불러오거나 모듈 통째로 불러올 수 있다.

```
# mod1.py
def add(a, b):
    return a+b

def sub(a, b):
    return a-b

if __name__ == "__main__":
    print(add(1, 4))
    print(sub(4, 2))
```

만약 `if __name__ == "__main__"`가 없는 상태에서 `mod1`을 다른 파일에서 모듈로 불러오면 오류가 발생한다. 이것은 `mod1`이 직접 실행되었을 때

※ 명령 프롬프트 창에서는 /, \든 상관없지만, 소스코드 안에서는 반드시 / 또는 \ 기호를 사용해야 한다.

1. sys.path.append(모듈을 저장한 디렉터리) 사용하기 `sys.path`라는 리스트에 모듈이 저장되어있는 디렉터리를 `append`로 추가시켜 모듈을 불러올 수 있게 한다.

```
>>> sys.path.append("C:/doit/mymod")
>>> sys.path
['', 'C:\\Windows\\SYSTEM32\\python36.zip', 'c:\\Python36\\DLLs',
'c:\\Python36\\lib', 'c:\\Python36', 'c:\\Python36\\lib\\site-packages',
'C:/doit/mymod']
>>> import mod2
>>> print(mod2.add(3,4))
7
```

2. PYTHONPATH 환경 변수 사용하기 모듈을 불러와서 사용하는 또 다른 방법으로는 PYTHONPATH 환경 변수를 사용하는 방법이 있다.

```
C:\doit>set PYTHONPATH=C:\doit\mymod
C:\doit>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type "help", "copyright", "credits" or "license" for more information.
>>> import mod2
>>> print(mod2.add(3,4))
7
```

06. package

init.py

```
__all__ = ['echo']
```

패캐지로 사용할 폴더마다 `__init__.py`를 만들어 저장한다. 위의 코드를 넣지 않고 빈 내용으로 저장해도 상관은 없지만 위 코드는 사용할 모듈이 직접 포함된 디렉터리 아닌 그 상위 디렉터리 이상에서 모두를 불러낼때 사용할 수있다. `__init__.py`가 있어야 패키지의 일부로 인식이 된다. 그 후 패캐지가 될 디렉터리 주소를 `sys.path`에 추가시키거나 `PYTHONPATH`에 추가시키면 된다.

```
C:/doit/game/__init__.py
C:/doit/game/sound/__init__.py
C:/doit/game/sound/echo.py
C:/doit/game/graphic/__init__.py
C:/doit/game/graphic/render.py
```

위와 같이 파일이 저장되어 있고 `echo.py`에 'echo'라는 문자열을 출력하는 `echo_test` 메소드가 선언되어 있다면 아래와 같이 선언하여 불러낼 수 있다.

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo

>>> from game.sound import echo
>>> echo.echo_test()
echo

>>> from game.sound import *
#만약 sound폴더의 __init__.py 내용에 '__all__ = ['echo']'를 추가시키지 않으면 오류가
발생한다.
>>> echo.echo_test()

>>> from game.sound.echo import echo_test
>>> echo_test()
echo

>>> from game.sound.echo import *
# 최하위 디렉토리에서 불러올 경우 __init__.py이 빈내용이여도 상관 없다.
>>> echo_test()
echo
```

python 파일 안에서 `from ..sound.echo import echo_test` 처럼 상대 경로를 사용할 수 있다. `..`은 부모 디렉터리인 `game`을 의미한다. `.`은 현재 디렉터리를 의미한다.

07. 예외처리

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

위와 같이 오류가 발생하면 `발생 오류: 오류 메시지`로 출력이 된다. 오류가 나면 다른 작업을 수행할 수 있는 방법이 있는데 이것이 예외처리이다.

```
try:
    4 / 0
except ZeroDivisionError as e:
    print(e)
```

형식은 `try`부분이 실행되어 `except` 에서 지정된 '발생 오류'가 발생하면 미리 선언된 코드를 실행시키는 식으로 되어있다. 오류메시지를 출력하고 싶다면 `as`를 통해 지정한 변수에 오류메시지를 저장한다. `except`만 작성하면 발생한 모든 오류에 대해 실행된다. 만일 오류가 발생했는데 무시하고 싶다면 내용에 `pass`를 입력하면 된다.

```
class Bird:
    def fly(self):
        raise NotImplementedError

class Eagle(Bird):
    def fly(self): # 오버라이드를 하지 않으면 오류가 발생한다.
        print("very fast")

eagle = Eagle()
eagle.fly()
```

의도적으로 오류를 내고 싶다면 위와 같이 `raise`를 사용하면 된다. `NotImplementedError`는 python 내장 오류로, 꼭 작성해야 하는 부분이 구현되지 않았을 경우 일부러 오류를 발생시키고자 사용한다. 그렇기에 Bird 클래스를 상속받는 자식 클래스는 반드시 `fly`라는 함수를 구현해야한다. 즉 메소드 오버라이드를 해야한다.

```
class MyError(Exception):
    pass

def say_nick(nick):
    if nick == '바보':
        raise MyError()
    print(nick)

try:
```

```
    say_nick("천사")
    say_nick("바보")
except MyError:
    print("허용되지 않는 별명입니다.")
```

위와 같이 오류를 만들어 낼 수 있다. 만약 오류메시지를 이용하면 다음과 같이 작성하면 된다.

```
class MyError(Exception):
    def __str__(self):
        return "허용되지 않는 별명입니다."

def say_nick(nick):
    if nick == '바보':
        raise MyError()
    print(nick)

try:
    say_nick("천사")
    say_nick("바보")
except MyError as e:
    print(e)
```

오류를 출력시키는 클래스에 `__str__` 메소드를 구현시켜야 한다.

python에서 재귀함수의 최대 깊이는 1000번이라고 한다.

08. Comprehensions

```
def reverse_letter(n):
    result = []
    for i in n :
        if i.isalpha() :
            result.append(i)
    return "".join(result[::-1])
```

위의 코드와 같이 `for문-if문-수식 한 줄`로 되어 있는데 이렇게 있으면 다음과 같이 한 줄로 줄일 수 있다.

```
def reverse_letter(n):
    a = [c for c in n if c.isalpha()]
    return "".join(a[::-1])
```

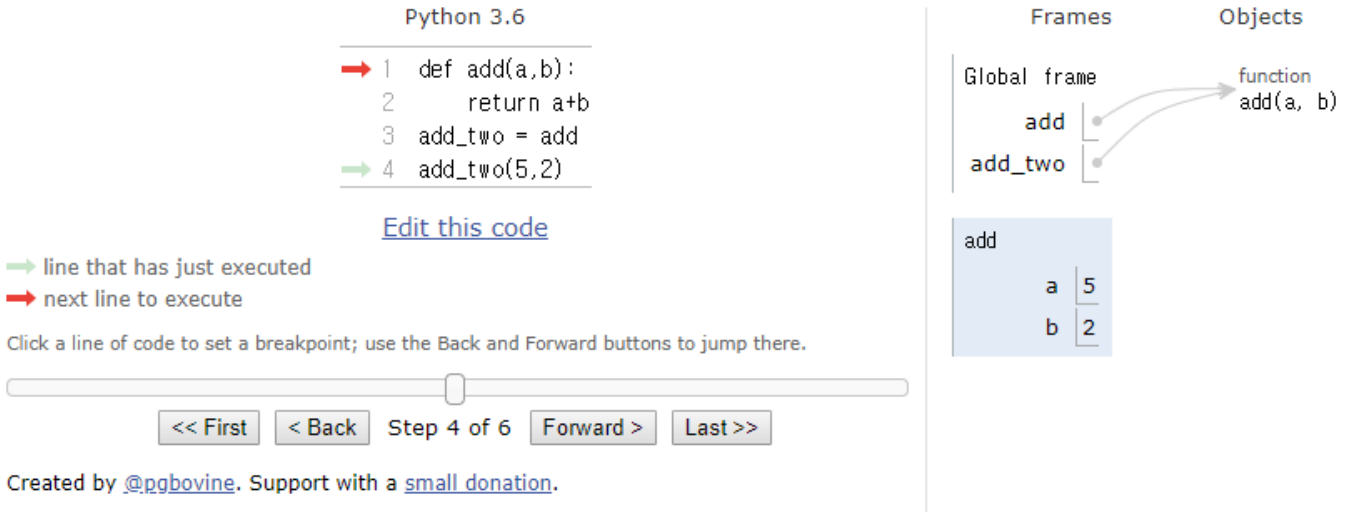
한 줄로 줄일 수 있는 전제조건은 for문과 if문 사이에 어떠한 식이 오면 안되고 수식은 한 줄로 구성되어 있어야 가능하다.

```
dusts = {'서울': 72, '경기': 82, '대전': 29, '중국': 200}
dics = {key : '매우나쁨' if value>150 else '나쁨' if value>80 else '보통' for key,
value in dusts.items()}
print(dics)
```

if문만 구성하거나 for문만 구성할 수 있다.

09. 일급 객체 함수

C와 Java와 다르게 python은 함수를 매개변수로 넘기거나 변수에 대입하는 것이 가능하다.



위와 같이 `add_two` 이름으로 함수가 실행 될 줄 알았으나 사실 `add`를 호출하여 구동되는 것이다.

```
map(int, input().split())
```

일급 객체가 가능하기 때문에 우리가 자주 쓰는 `map`이 사용가능하다.

위 상황을 정리하면 다음과 같다.

```
1. 선언한 함수의 기능을 add 이름에 할당한 것
2. 함수를 호출(call)한다는 것은 add() 형식으로 호출
3. print(add)로 출력하면 add 함수가 저장된 주소값이 출력됨
4. 그렇기에 add_two는 add 함수를 가리키는 것이므로 실제 호출되는 것은 add 함수임
```

```
def ohiyou(n):
    print("ohiyou",n)
def greeter(fnc):
    fnc('hi')
# 함수를 인자로 사용할 수 있다.
greeter(ohiyou)
```

인자로 사용될 함수가 매개변수를 사용해야하는 함수이면 `greeter(ohiyou('hi'))`처럼 인자로 전달하면서 매개변수를 동시에 전달 시킬 수 없다. 그러나 `greeter` 내부에서 매개변수를 전달시키거나 아예 별도의 매개변수를 전달시키면 실행이 가능하다.

10. class

```
class 클래스명 [(상속받을 클래스명)]:
    클래스변수명 = 값
    #...
    def __init__ (self[, 매개변수[=default값],..]): #생성자
        self.매개변수 = 매개변수 #인스턴스 변수 선언
        #...
    def __del__ (self[, 매개변수[=default값],..]): #소멸자
        #...
    def 메서드명 (self[, 매개변수[=default값],..]): #인스턴스 메서드
        #...
    @staticmethod #정적 메서드
    def 메서드명 ([매개변수[=default값],..]):
        #...
    @classmethod #클래스 메서드
    def 메서드명 (cls[, 매개변수[=default값],..]):
        #...
    def __add__(self,obj): # (연산자) 오버라이드
        #...
```

1) self

인스턴스 객체 자기 자신을 의미하며 특별한 상황을 제외하고 무조건 메서드에서 첫번째 인자인 `self`에 인스턴스 객체를 받는다. 굳이 'self'로 이름을 정의할 필요는 없지만 통일시켜야한다.

2) 생성자 & 소멸자

생성자는 인스턴스 객체가 생성될 때 호출되는 함수이며, 소멸자는 `del`이나 가비지 컬렉터로 인해 인스턴스 객체가 소멸될 때 호출되는 함수이다.

3) 인스턴스 메서드

`self`로 인스턴스 객체를 받는다. 클래스에서 접근하는 방법으로는 직접 인스턴스 객체를 인수로 받아 사용할 수 있다.

4) 정적 메서드

`@staticmethod`를 적고 나서 함수 선언하면 된다. `self`를 받지 않으며 인스턴스 객체에서도 메서드 사용이 가능하다. 정적 메서드는 주로 클래스 변수 등 데이터를 수정하지 않는 메서드를 만들 때 사용한다.

5) 클래스 메서드

`@classmethod`를 적고 나서 함수 선언하면 된다. 클래스를 받으므로 클래스에서도 메서드 사용이 가능하다. 클래스 메서드는 클래스 변수를 조작하는 메서드를 만들 때 주로 사용된다.

6) 상속

상속은 부모 클래스의 모든 속성이 복사되는 것이 아니라 자식 클래스에서 없으면 부모 클래스로 넘어가 찾아서 사용하므로 마치 복사된 것처럼 보이는 것이다. 상속 관계에서의 이름공간은 `인스턴스 -> 자식 클래스 -> 부모 클래스 -> 전역` 순으로 확장된다.

7) 오버라이드

연산자 오버라이드, 메서드 오버라이드가 있으며 상속 받거나 기존에 있는 메서드를 다시 사용자가 재정의하는 것을 의미한다. 자식클래스에서 부모 클래스의 메서드 내용을 재사용하는 `super()`를 사용할 수 있다.

8) 클래스 이름공간(namespace)

- 클래스를 정의하면, 클래스 객체가 생성되고 해당되는 이름 공간이 생성된다.
- 인스턴스를 만들게 되면, 인스턴스 객체가 생성되고 해당되는 이름 공간이 생성된다.
- 인스턴스의 어트리뷰트가 변경되면, 변경된 데이터를 인스턴스 객체 이름 공간에 저장한다.
- 즉, 인스턴스에서 특정한 어트리뷰트에 접근하게 되면 인스턴스 -> 클래스 순으로 탐색을 한다.