

# Object-Oriented Programming



PYTHON 프로그래밍 실습

***POSTECH***

# 들어가기 앞서

- 단일 은행 계좌 모델링
  - balance: 잔액을 나타내는 변수
  - deposit(amount): amount를 받아 잔액에 입금하는 함수
  - withdraw(amount): amount를 받아 잔액에 출금하는 함수
- 주의: 전역변수 수정시 ? keyword 사용해야 수정 가능

```
1 balance = 0
2
3 def deposit(amount):
4     pass
5
6 def withdraw(amount):
7     pass
```

```
1 print(balance)
2 print(deposit(500))
3 print(balance)
4 print(withdraw(300))
5 print(balance)
6
0
500
500
200
200
```

# 들어가기 앞서 (cont.)

- 여러 은행 계좌 모델링
  - balance0, balance1, balance2 ?
    - deposit0, ...
    - withdraw0, ...
- 사전을 활용하여 스코프를 로컬로 만들어 해결 가능

```
1 def make_account():
2     return {'balance': 0}
3
4 def deposit(account, amount):
5     pass
6
7 def withdraw(account, amount):
8     pass
```

```
1 a = make_account()
2 b = make_account()
3 print(a['balance'])
4 print(b['balance'])
5 print(deposit(a, 1000))
6 print(deposit(b, 2000))
7
```

```
0
0
1000
2000
```

# 객체 지향 프로그래밍

- 프로그래밍 패러다임(paradigm) 중 하나
- 객체란? 모종의 변수와 메서드를 담는 추상적 개념
- 프로그램을 객체 간의 상호작용으로 서술

# Python은 모든 것이 객체

- Python은 객체 지향 프로그래밍
- 객체란? 모종의 변수와 메서드를 담는 추상적 개념
- int, string, list, function ... 모든 것이 객체

```
list_ex = list([1,2,3])
```

```
list_ex.
```

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse

```
dict_ex = dict({"dog":1})
```

```
dict_ex.
```

- clear
- copy
- fromkeys
- get
- items
- keys
- pop
- popitem
- setdefault
- update

```
int_ex = int(5)
```

```
int_ex.
```

- bit\_length
- conjugate
- denominator
- from\_bytes
- imag
- numerator
- real
- to\_bytes

# 클래스 (Class)

- 대부분의 객체 지향 프로그래밍 언어에 사용되는 개념
- 클래스는 객체의 기반이 되는 틀을 정의함
- 클래스 정의 문법
  - 클래스의 정의가 끝날 때 생성됨

```
class ClassName:  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

# 클래스 객체 (Class Object)

- 클래스 객체는 두 종류의 연산을 지원
  - 어트리뷰트 참조 (표준 문법 사용: obj.name)
    - MyClass.i
    - MyClass.f

```
class MyClass:  
    """A simple example class"""  
    i = 12345  
  
    def f(self):  
        return 'hello world'
```

데이터 어트리뷰트 →

메서드 어트리뷰트 →

- 인스턴스 만들기 (class(들)를 이용하여 사례(case) 생성)
  - 호출(call) 사용

```
x = MyClass()
```

# 클래스 객체 (Class Object) (cont.)

- 특정 초기 데이터를 가지는 객체 생성
  - 특수 메서드 정의 (생성자) : `__init__()`
  - 인스턴스 생성시 자동으로 호출

```
def __init__(self):  
    self.data = []
```

- 인자들을 가지는 `__init__()`

```
>>> class Complex:  
...     def __init__(self, realpart, imagpart):  
...         self.r = realpart  
...         self.i = imagpart  
...  
>>> x = Complex(3.0, -4.5)  
>>> x.r, x.i  
(3.0, -4.5)
```



# 인스턴스 객체 (Instance Object)

- 인스턴스 객체는 오직 한가지 연산만 가능
  - 어트리뷰트 참조

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

- 데이터 어트리뷰트는 처음 대입될 때 생성, 선언 불필요

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

# 메서드 객체 (Method Object)

- 일반 함수 객체와 달리, 클래스에 종속되어 있는 함수를 지칭
- 대상이 되는 객체가 있어야 사용 가능

```
class MyClass:  
    """A simple example class"""  
    i = 12345  
  
    def f(self):  
        return 'hello world'
```

- 메서드의 특별함 (첫 번째 인자에 객체 자동 전달)
  - 아래 두 호출은 동등

`x.f()`      `MyClass.f(x)`

# 클래스와 인스턴스 데이터

- 인스턴스 데이터 = 인스턴스가 가지는 데이터
- 클래스 데이터 = 클래스의 모든 인스턴스가 공유하는 어트리뷰트와 메서드를 지칭

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

# 클래스와 인스턴스 변수 (cont.)

## ■ 클래스 변수의 특징

- 아래와 같이 kind의 값의 수정은 불가능 (immutable 불가능)
- 변경시 해당 인스턴스에 새로운 변수가 복사, 생성됨
- 단 mutable 객체의 경우 직접 수정 가능 (주의)

```
class Dog:
    kind = 'canine'
```

```
d = Dog()
e = Dog()
print(d.kind) # canine
print(e.kind) # canine
d.kind = 'puppy'
print(d.kind) # puppy
print(e.kind) # canine
```

```
canine
canine
puppy
canine
```

# 클래스와 인스턴스 변수 (cont.)

- 클래스 변수의 특징
  - 단 mutable 객체의 경우 직접 수정 가능 (주의)

```
class Dog:

    tricks = []           # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks           # unexpectedly shared by all dogs
['roll over', 'play dead']
```

# 클래스와 인스턴스 변수 (cont.)

- 클래스 변수의 특징
  - 선호되는 설계는 인스턴스 변수로 사용하는 것

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

# 실습 문제 1

- 은행 계좌를 클래스를 사용하여 모델링하세요
  - 계좌 클래스의 초기화 값은 balance와 name을 지님
    - 계좌 생성시 초기값으로 balance와 name을 지정 가능
    - 지정 안할 시 각 초기값은 0과 “none”을 가질 것
  - 계좌의 기능은 입금, 출금, 계좌정보출력
  - 출금은 출금하고자하는 금액이 잔고에 있을 때 출금할 것
  - 계좌에 1000원 입금 900원 출금하여 잔고를 100으로 만들기
  - 계좌에 400원 입금 후 600원 출금하여 “잔액 부족” 출력하기

```
1 class BankAccount:
2     def __init__(self, ?):
3         pass
4
5     def deposit(self, amount):
6         pass
7
8     def withdraw(self, amount):
9         pass
10
11     def get_info(self):
12         pass
```

```
1 a = BankAccount(100, "Gildong Hong")
2 ... 입금, 출금
3 a.get_info() # 계좌 정보 출력
```

이름: Gildong Hong

잔고: 100

# 상속 (Inheritance)

- 기존 클래스의 속성을 물려 받아 새로운 클래스를 만드는 것
  - 새 클래스는 기존 클래스의 모든 변수 및 메서드를 가짐
  - 주로 기존 클래스를 확장하는 용도로 사용
  - 다중 클래스도 상속 가능 ", "로 구분

```
class DerivedClassName(BaseClassName):  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```



# 상속 (Inheritance)

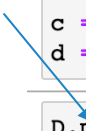
```
class A:  
    a = 1  
    def print_a():  
        print("A class")
```

```
class B:  
    b = 1  
    def print_b():  
        print("B class")
```

```
class C(A):  
    c = 1  
    def print_c():  
        print("C class")
```

```
class D(C,B):  
    d = 1  
    def print_d():  
        print("D class")
```

Method  
Resolution  
Order



```
c = C()  
d = D()
```

```
D.mro()
```

```
[__main__.D, __main__.C, __main__.A, __main__.B, object]
```

# 메서드 오버라이딩 (Overriding)

- 기존 클래스 메서드를 상속 클래스에서 재정의 하는 것
  - 재정의하지 않은 메서드는 기존 클래스의 것을 그대로 사용
  - 오버라이딩한 경우 기존 클래스의 메서드를 사용하고자 하는 경우 `super()` 사용

```
1 class A():
2     def __init__(self):
3         print("this is [A] class and [__init__] function")
4         self.var1 = 0
5         self.var2 = 0
6
7 class B(A):
8     def __init__(self):
9         super().__init__()
10        print("this is [B] class and [__init__] function")
11        self.var3 = 0
12        self.var4 = 0
```

```
1 a = A()
2 b = B()
```

```
this is [A] class and [__init__] function
this is [A] class and [__init__] function
this is [B] class and [__init__] function
```

## 실습 문제 2

- 최소 잔액을 유지해야 하는 계좌 클래스를 만드세요
  - 기본적으로 기존 실습 문제1의 계좌와 기능 동일함
  - 하지만 출금시 잔액이 최소 잔액 미만이면 출금 못함 그리고 “최소 잔액을 유지해야 합니다” 메시지 출력
- 힌트: 상속 및 오버라이딩 활용

```
1 class MinimumBalanceAccount(?):  
2     def __init__(self, minimum_balance):  
3         pass  
4     ...
```

```
1 a = MinimumBalanceAccount(500, "Kim")
```

```
1 a.deposit(1000)  
2 a.withdraw(1500)
```

최소 잔액을 유지해야 합니다

# 연산자 오버로딩 (Operator Overloading)

- Python의 연산자는 모두 내장된 특수 메소드를 호출함
  - 해당 메소드를 override 하는 것으로 기능 변경 가능
  - e.g. `a + b` 는 자동으로 `a.__add__(b)` 를 호출

```
class Account:
    ...

    def __iadd__(self, amount):
        self.deposit(amount)
        return self
    def __isub__(self, amount):
        self.withdraw(amount)
        return self

acc = Account(1000)
acc += 1000
print(acc.balance)
```

# 연산자 오버로딩 (Operator Overloading) (cont.)

연산	호출되는 함수
<code>a + b</code>	<code>a.__add__(b)</code>
<code>a - b</code>	<code>a.__sub__(b)</code>
<code>a * b</code>	<code>a.__mul__(b)</code>
<code>a / b</code>	<code>a.__truediv__(b)</code>
<code>a // b</code>	<code>a.__floordiv__(b)</code>
<code>a % b</code>	<code>a.__mod__(b)</code>
<code>a ** b</code>	<code>a.__pow__(b)</code>
<code>a &amp; b</code>	<code>a.__and__(b)</code>
<code>a   b</code>	<code>a.__or__(b)</code>
<code>a ^ b</code>	<code>a.__xor__(b)</code>
<code>-a</code>	<code>a.__neg__()</code>
<code>~a</code>	<code>a.__invert__()</code>

연산	호출되는 함수
<code>a += b</code>	<code>a.__iadd__(b)</code>
<code>a -= b</code>	<code>a.__isub__(b)</code>
<code>a *= b</code>	<code>a.__imul__(b)</code>
<code>a /= b</code>	<code>a.__itruediv__(b)</code>
<code>a //= b</code>	<code>a.__ifloordiv__(b)</code>
<code>a %= b</code>	<code>a.__mod__(b)</code>
<code>a **= b</code>	<code>a.__ipow__(b)</code>
<code>a &amp;= b</code>	<code>a.__iand__(b)</code>
<code>a  = b</code>	<code>a.__ior__(b)</code>
<code>a ^= b</code>	<code>a.__ixor__(b)</code>

연산	호출되는 함수
<code>a &lt; b</code>	<code>a.__lt__(b)</code>
<code>a &lt;= b</code>	<code>a.__le__(b)</code>
<code>a == b</code>	<code>a.__eq__(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>
<code>a &gt; b</code>	<code>a.__gt__(b)</code>
<code>a &gt;= b</code>	<code>a.__ge__(b)</code>

연산	호출되는 함수
<code>a[key]</code>	<code>a.__getitem__(key)</code>
<code>a[key]</code>	<code>a.__setitem__(key)</code>
<code>len(a)</code>	<code>a.__len__()</code>
<code>print(a)</code>	<code>a.__str__()</code>

# 과제 (1 page)

- Python의 기본 자료구조 set 를 모방한 클래스 Set을 구현하세요
  - 임의의 원소를 중복 없이, 순서 없이 담는 집합형 자료구조
  - 아래의 메서드/연산자를 명시된 기능대로 구현
    - 생성자 (`__init__`)
      - list를 받아 중복 제거; 매개 변수 없이 생성 시, 빈 집합 상태로 생성
    - `add(elem)`
      - Set에 elem이 존재하지 않으면 추가
    - `discard(elem)`
      - Set에 elem이 존재하면 삭제
    - `clear()`
      - Set에 존재하는 모든 원소 삭제
    - `__len__()`
      - Set에 존재하는 원소 개수 반환

# 과제 (2page)

- Python의 기본 자료구조 set 를 모방한 클래스 Set을 구현하세요
  - 아래의 메서드/연산자를 명시된 기능대로 구현
    - `__str__()`
      - Set에 존재하는 원소를 '{1, 2, 3}'의 형태로 반환
    - `__contains__(elem)` ##### in 멤버체크
      - Set에 elem이 존재하면 참 반환, 아니면 거짓 반환
    - `self <= other`
      - self가 other의 부분집합이면 참 반환, 아니면 거짓 반환
    - `self >= other`
      - other가 self의 부분집합이면 참 반환, 아니면 거짓 반환

# 과제 (3page)

- Python의 기본 자료구조 set 를 모방한 클래스 Set을 구현하세요
  - 아래의 함수/연산자를 명시된 기능대로 구현
    - self | other
      - self와 other의 원소를 모두 포함하는 합집합 Set 반환
    - self & other
      - self와 other가 공통으로 포함하는 원소를 포함하는 교집합 Set 반환
    - self - other
      - self의 원소 중 other에 없는 원소만을 포함하는 차집합 Set 반환
    - |=, &=, -=
      - 위의 기능에 맞추어 구현, In-place operation
      - 자신을 return 하는 것을 잊지 말 것



# 과제 (4page)

1

- list 관련 함수 사용 무방  
set 함수는 사용 안됨

- 복합 대입 연산자 경우  
대입되는 결과 return  
하는 것 잊지 말것

- 제출 목록

1

2

- ipybn 파일
- 오른쪽 두 그림 캡처

```
a = Set([1, 2, 3, 4])  
b = Set([2, 3, 4])
```

```
print(a)  
print(b)  
print()
```

```
a.discard(4)  
b.discard(2)  
print(a)  
print(b)  
print()
```

```
print(len(a))  
print(1 in a)  
print(1 in b)  
print()
```

```
print(a | b)  
print(a & b)  
print(a - b)  
print()
```

```
print(a <= b)  
print(a <= a | b)  
print(a >= b)  
print(a >= a & b)  
print()
```

```
b.clear()  
print(b)
```

2

```
{1, 2, 3, 4}  
{2, 3, 4}
```

```
{1, 2, 3}  
{3, 4}
```

```
3  
True  
False
```

```
{1, 2, 3, 4}  
{3}  
{1, 2}
```

```
False  
True  
False  
True
```

```
{}
```

# 이터레이터 (Iterator)

```
for i in 5:  
    print(i)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-83-2ce37977c71e> in <module>  
----> 1 for i in 5:  
      2     print(i)  
  
TypeError: 'int' object is not iterable
```

range    open    list   tuple   dict   set   str   ...

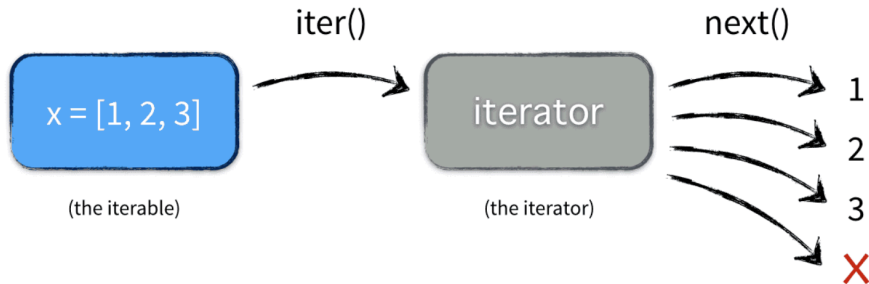
# 이터레이터 (Iterator)

- 지금까지 배운 컨테이너 객체들의 루핑: for
  - for 문은 기본적으로 iter() 내장 함수를 호출
  - iter() 함수는 \_\_next\_\_()를 정의하는 이터레이터 객체 반환
  - \_\_next\_\_()는 컨테이너의 요소를 한 번에 하나씩 접근
  - 남은 요소가 없다면 StopIteration 예외 발생 for 루프 종료

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

# 이터레이터 (Iterator)

- iterable 가능한 객체
  - `dir(object) > __iter__()` 메서드를 가지고 있다.
- iterable 객체 iterator 객체로 만들기
  - `iter(object) > iterator` 객체
- for문은 in 뒤에 iterable한 객체만 수용
  - `iter(object) → next(iter_obj) → iterable 객체 원소 반환`



# 이터레이터 (Iterator) (cont.)

## ■ 이터레이터 예제

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

# 이터레이터 (Iterator) (cont.)

- 클래스 객체에 이터레이터 동작 추가
  - `__iter__()` 정의
    - `__next__()` 매서드를 가진 객체를 돌려줌
  - `__next__()` 정의
    - 원하는 형태의 동작 정의

```
class xrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
>>> y = xrange(3)
>>> next(y)
0
>>> next(y)
1
>>> next(y)
2
>>> next(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __next__
StopIteration
```

# 실습 문제 3

- 주어진 문자열을 반대로 출력하는 이터레이터 클래스를 구현하세요.
  - Reverse('spam')

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    ...
```

```
rev = Reverse('spam')
print(next(rev))
m
print(next(rev))
a
print(next(rev))
p
print(next(rev))
s
```

# 제너레이터 (Generator)

- “yield”를 사용하여 데이터를 하나씩 반환하는 함수
  - 이터레이터와 같은 동작 수행, 하지만 보다 간단
    - `__iter__()`와 `__next__()` 메서드 자동 생성
  - 일반적인 함수처럼 작성되지만 값을 반환하고 싶을 때마다 “yield”문을 사용
  - 마지막 반환 결과를 기억, 재호출시 그 위치부터 다시 시작
  - 데이터가 대량일 경우 일부씩 처리시 유용
  - On demand 계산을 하나씩 처리하고 싶은 경우

# Generator 함수

```
def gen():  
    yield 1  
    yield 2  
    yield 3
```

# Generator 객체

```
g = gen()  
print(type(g)) # <class 'generator'>
```

# next() 함수 사용

```
n = next(g); print(n) # 1  
n = next(g); print(n) # 2  
n = next(g); print(n) # 3
```

# for 루프 사용 가능

```
for x in gen():  
    print(x)
```



# 제너레이터 (Generator) 예제

```
1 def square_numbers(nums):
2     for x in nums:
3         yield x * x
4
5 result = square_numbers([1, 2, 3, 4, 5])
6
7
8 print(next(result))
9 print(next(result))
10 print(next(result))
11 print(next(result))
12 print(next(result))
13 print(next(result))
```

```
1
4
9
16
25
```

```
-----
StopIteration                                Traceback (most recent call last):
  <ipython-input-4-2718dd0a2a49> in <module>
      11 print(next(result))
      12 print(next(result))
--> 13 print(next(result))
```

```
StopIteration:
```

# 실습 문제 4

- 주어진 문자열을 반대로 출력하는 제너레이터 함수를 구현하세요.
  - `def reverse(input_string):`
    - tip: use `range(start, end, step)` function

```
def reverse(input_string):  
    pass
```

```
# For loop to reverse the string  
for char in reverse("hello"):  
    print(char)
```

```
o  
l  
l  
e  
h  
>>>
```

# 제너레이터 표현식 (Generator Expression)

- (...)를 사용하여 표현되는 제너레이터 표현식
  - \* comprehension과 비슷하지만 ( ) 사용
  - 제너레이터 함수의 간결한 표현 형태 하지만 융통성 부족
  - 표현식만을 갖는 제너레이터 객체만 반환

```
1 def square_numbers(nums):  
2     for x in nums:  
3         yield x * x  
4  
5 result = square_numbers([1, 2, 3, 4, 5])  
6
```

```
>>> numbers = [1, 2, 3, 4, 5, 6]  
>>> [x * x for x in numbers]  
[1, 4, 9, 16, 25, 36]
```

```
>>> {x * x for x in numbers}  
{1, 4, 9, 16, 25}
```

```
>>> {x: x * x for x in numbers}  
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

```
>>> lazy_squares = (x * x for x in numbers)  
>>> lazy_squares  
<generator object <genexpr> at 0x10d1f5510>  
>>> next(lazy_squares)  
1  
>>> list(lazy_squares)  
[4, 9, 16, 25, 36]
```

# First Class Object

- First-class란? 아래의 조건을 만족하는 객체
  - 변수나 데이터 구조안에 담을 수 있다
  - 함수 인수로 전달할 수 있다
  - 함수 결과로 리턴할 수 있다

```
def yell(text):  
    return text.upper() + '!'  
  
>>> yell('hello')  
'HELLO!'
```

# Func. is First Class Object (cont.)

- 함수 = First-class object
  - 변수나 데이터 구조안에 담을 수 있다

```
>>> bark = yell
```

```
>>> bark('woof')  
'WOOF!'
```

```
>>> funcs = [bark, str.lower, str.capitalize]  
>>> funcs  
[<function yell at 0x10ff96510>,  
 <method 'lower' of 'str' objects>,  
 <method 'capitalize' of 'str' objects>]
```

# Func. is First Class Object (cont.)

- 함수 = First-class object
  - 변수나 데이터 구조안에 담을 수 있다

```
>>> for f in funcs:  
...     print(f, f('hey there'))  
<function yell at 0x10ff96510> 'HEY THERE!'  
<method 'lower' of 'str' objects> 'hey there'  
<method 'capitalize' of 'str' objects> 'Hey there'
```

```
>>> funcs[0]('heyho')  
'HEYHO!'
```

# Func. is First Class Object (cont.)

- 함수 = First-class object
  - 함수 인수로 전달할 수 있다

```
def greet(func):  
    greeting = func('Hi, I am a Python program')  
    print(greeting)
```

```
>>> greet(yell)  
'HI, I AM A PYTHON PROGRAM!'
```

- 고계함수 (higher-order function)
  - 함수의 인자로 다른 함수를 받거나 반환할 수 있는 것

```
>>> list(map(yell, ['hello', 'hey', 'hi']))  
['HELLO!', 'HEY!', 'HI!']
```

# Nested functions

- 함수 = First-class object
  - 함수 결과로 리턴할 수 있다

```
def get_speak_func(volume):  
    def whisper(text):  
        return text.lower() + '...'  
    def yell(text):  
        return text.upper() + '!!'  
    if volume > 0.5:  
        return yell  
    else:  
        return whisper
```

```
>>> get_speak_func(0.3)  
<function get_speak_func.<locals>.whisper at 0x10ae18>  
  
>>> get_speak_func(0.7)  
<function get_speak_func.<locals>.yell at 0x1008c8>  
  
>>> speak_func = get_speak_func(0.7)  
>>> speak_func('Hello')  
'HELLO!'
```



# Lexical Closures

- Lexical Closures (Closures, for short)
  - 내부 함수가 상위 함수의 상태를 가져오는 행동

```
def make_adder(n):  
    def add(x):  
        return x + n  
    return add  
  
>>> plus_3 = make_adder(3)  
>>> plus_5 = make_adder(5)  
  
>>> plus_3(4)  
7  
>>> plus_5(4)  
9
```

# Callable

```
1 list = 5
2 x = list(5)
```

```
-----
TypeError                                T1
<ipython-input-160-24977d33b5c4> in <module>
      1 list = 5
----> 2 x = list(5)

TypeError: 'int' object is not callable
```

- 함수처럼 행동하는 객체
  - 모든 함수=객체 but 모든 객체!=함수
  - callable(?) 함수로 판단가능 True/False
- Callable = 함수처럼 호출할 수 있는 객체 "()"
  - `__call__` 메소드 정의

```
class Adder:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return self.n + x
```

```
>>> plus_3 = Adder(3)
```

```
>>> plus_3(4)
```

```
7
```

객체 생성

객체 호출

# Decorator

- 어떤 함수/클래스에 기능을 추가한 뒤 이를 다시 함수의 형태로 반환 (즉, 어떤 함수/클래스를 꾸며주는 함수)
- 어떤 함수의 내부 수정 없이 기능 추가시 사용
- 데코레이터 구조

```
def 데코레이터이름(func): # 기능을 추가할 함수를 인자로 받아온다.  
    def 내부함수이름(*args, **kwargs):  
        기존 함수에 추가할 명령  
        return func(*args, **kwargs)  
    return 내부함수이름
```

# Decorator 예제

- introduce 함수

```
def introduce(name):  
    print(f'My name is {name}!')  
introduce('Chaewon')
```

My name is Chaewon!

- introduce 함수를 확장하는 decorator 함수

```
def decorator(func):  
    def wrapper(*args, **kwargs):  
        print('Hello')  
        return func(*args, **kwargs)  
    return wrapper
```

```
decorated_introduce = decorator(introduce)
```

```
decorated_introduce('Chaewon')
```

Hello

My name is Chaewon!

# Decorator 예제 (cont.)

## ■ 데코레이터 기호 '@'의 사용

```
def decorator(func):  
    def wrapper(*args, **kwargs):  
        print('Hello')  
        return func(*args, **kwargs)  
    return wrapper
```

• `@decorator` # 데코레이터 함수를 적용할 함수 바로 위에 '@데코레이터이름'을 붙여준다.

```
def introduce(name):  
    print(f'My name is {name}!')
```

```
introduce = decorator(introduce)
```

```
introduce('Chaewon')
```

Hello

My name is Chaewon!

# Decorator 예제 (cont.)

- 그 밖의 decorator..
  - decorator with argument
  - class decorator

# Reference

- python tutorial
  - <https://docs.python.org/ko/3/tutorial/>
- first class object
  - <https://dbader.org/blog/python-first-class-functions>
- iterator, generator
  - <https://nvie.com/posts/iterators-vs-generators/>
- decorator
  - <https://nachwon.github.io/decorator/>

Thanks!

**Any questions?**