호출 가능한 객체

Python에서 함수(`function`)나 메서드(`method`)는 모두 괄호 등을 사용하여 **호출**을 해줄 수 있습니다.

```
print()        # print 함수 호출
1 + 3          # __add__ 메서드 호출
```

한 편, 전 시간에 우리는 함수와 메서드도 모두 *인스턴스*임을 배웠습니다. Python은 이렇게 인스턴스에 호출 기능을 넣어줄 방법으로 `__call__` 메직 메서드를 제공합니다. 이번 시간에는 호출 가능한 인스턴스와 다양한 특성들에 대해 배워보겠습니다.

호출 메서드

Python에서 생성된 인스턴스에 호출 기능을 넣기 위해서는 `__call__` 함수를 사용합니다. 인스턴스 호출 `instance()`는 `instance.__call__()`에 대응됩니다.

```
class CallObject:
    def __call__(self):
        print("Called!")

myCall = CallObject()
# Called!
```

사실, 객체에 호출 역할을 넣어주는 것은 이게 전부입니다. 그러므로 오늘은 호출에 관한 다양한 유틸리티 함수들과 여러 호출 가능한 객체들을 살펴봅시다.

호출 가능성

어떤 인스턴스가 호출 가능한 지 알고 싶을 때는 `callable` 함수를 사용할 수 있습니다.

```
print(callable(myCall))
# True
```

+) 그렇다면, 클래스는 호출 가능할까요?

> 네, 그렇습니다.

클래스도 인스턴스라는 점을 주목해주세요! 모든 클래스는 `type` 클래스의 인스턴스이며, `type` 클래스는 `__call__`을 가지고 있습니다! 그리고 이 덕분에, `Class()`가 인스턴스를 생성할 수 있는 겁니다!

```
print(callable(CallObject))
# True
```

(당연히 `__call__`이 없는 클래스를 넣어도 다 됩니다)

> `type` 클래스의 특성에 대해서는 다음 시간, *메타 클래스*라는 주제로 살펴보겠습니다.

함수 뜯어보기

다음으로 호출 가능한 객체들의 특성을 알아볼 겁니다. 가장 먼저 함수에 대해 살펴봅시다. 함수는 `function` 클래스의 인스턴스일 뿐이라는 점을 잘 생각해주세요. 일반적인 인스턴스에서

```
x = 1
```

처럼 사용하듯이,

```
def x():
    print(1)
```

는 사실 `function` 클래스로 선언한 인스턴스를 `x`라는 변수에 덮어씌웠을 뿐입니다. 따라서 다음과 같이 변수에 덧씌울 수도 있습니다.

```
y = x
y()
# 1
```

그런데 신기한 건, `y`를 출력하려고 시도하면 다음과 같이 나타납니다.

```
print(y)
# <function x at 0x770616b35440>
```

우리가 배우기로는 `x`는 변수일 뿐인데, 어떻게 `function x`라는 이름을 사용하고 있는 걸까요?

함수의 구성

사실 함수를 선언할 때에는 정해진 형식에 따라:

- `__name__`: 함수의 이름
- `__doc__`: 함수의 설명
- `__annotations__`: 인자와 반환의 타입 힌트
- `__defaults__`: 인자의 기본값들

같은 것들이 자동으로 해석됩니다. 예를 들어서,

```
def add(a: int, b: int = 1) -> int:
    """두 인자를 더하는 함수"""
    return a+b
```

처럼 정의되었다고 합시다. 그럼 우선 함수를 선언할 때 사용한 변수의 이름인 `add`를 `__name__`으로, 함수 바로 밑에 작성한 문자열을 `__doc__`로 취합니다.

```
print(add.__name__, add.__doc__, sep=", ")
# add, 두 인자를 더하는 함수
```

타입 힌트는 `__annotations__`에서 찾을 수 있습니다.

```
print(add.__annotations__)
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

인자의 기본값은 기본적으로 튜플로 저장합니다.

```
print(add.__defaults__)
# (1,)
```

그러나 이 때 `a`는 기본값을 설정해주지 않았으나 `a`에 대한 정보는 포함되어 있지 않습니다. 이 경우, 맨 뒤부터 차례대로 값을 채워넣어보면 `b=1`임을 알 수 있습니다.

익명 함수

함수의 작동 방식은 이름과 무관하기 때문에, 사실 이름이 없는 함수도 생각해볼 수 있겠습니다. 간단한 함수는 이름 없이 정의하는 것도 가능하며, 이를 **익명 함수**라고 합니다. 익명 함수 이론은 모두 함수형 프로그래밍 패러다임의 람다 대수를 기반으로 하기 때문에 Python에서도 `lambda` 키워드로 정의합니다.

```
square = lambda x: x*x
square(2)
# 4
```

직관적으로 정의되어 있습니다. `lambda` 뒤에 사용할 인자들을 적고, `:` 뒤에 반환할 값을 적는 겁니다! 다만, 익명 함수는 타입 힌트라던가 기본값 설정 같은 고급 기능은 사용할 수 없습니다.

익명 함수는 보통 `map`과 함께 많이 사용됩니다. `map`은 함수와 반복 가능한 객체를 인자로 받아, 반복 가능한 객체에서 차례로 얻은 값에 함수를 적용시켜 반환하는 반복 가능한 객체입니다.

```
for x in map(square, range(5)):
    print(x, end=", ")
# 0, 1, 4, 9, 16,
```

> **주의**: 익명 함수는 굳이 이름을 할당하지 않는 것에 가치가 있는 것이므로 위에서처럼 변수에 할당하는 것은 무가치합니다. `map` 과 같이 작용을 정해줘야 할 때 직접 넣어 사용해주세요.

```
  for x in map(lambda x: x*x, range(5)): ...
```

로 변경하는 게 권장됩니다.

익명 함수도 함수입니다. 다만 이름이 없어서 그냥 `<lambda>`로 표기됩니다.

```
print(square)
# <function <lambda> at 0x718528215440>
```

클로저

함수 밖에 선언되었지만, 함수 내에서 참조해야 하는 변수가 존재하는 경우가 있습니다. 다음의 함수 안의 함수를 다루는 코드를 살펴봅시다.

```
def outer(x):
    def inner():
        print(x)
    return inner
inner = outer(3)
inner()
# 3
```

이렇게:

1. 외부 함수 안에 내부함수가 정의되어 있고
2. 외부 함수가 내부 함수를 반환하고
3. 내부 함수는 외부 함수의 변수를 참조하는

경우 내부 함수를 **클로저**라고 합니다. 이 예제에서 `outer` 함수는 `inner` 함수를 반환하므로, `outer(3)`은 함수입니다. 그리고 이 함수는 호출하면 `3`을 출력하는 함수입니다.

코드상에서 `inner`는 당연히 `x`를 찾을 수 있습니다. 그러나 밖에서 실제로 실행할 때에는, `inner` 함수 안에는 그 어디에도 `x`를 정의하는 부분이 없기 때문에 위 코드를 실행할 수가 없습니다! 그래서 이러한 경우에, *자신을 정의하는 함수에 있으나 꼭 참조해야 하는 변수*를 반드시 따로 저장해야 합니다(이러한 `x`를 **자유 변수**라고 함.)

클로저 뜯어보기

사실 대부분의 자습서에서는 *자유 변수는 어딘가에 저장되어 있다~* 하고 넘어갑니다. 하지만 후에 `inspect` 모듈을 다루면서 클로저의 작동 방식에 대한 이해가 필수로 필요합니다. 그래서, 뜯어보겠습니다. Python에서는 자유 변수를 `__closure__`에 저장합니다.

```
print(inner)
# <function outer.<locals>.inner at 0x74d2c6632480>
print(inner.__closure__)
# (<cell at 0x76298e8686a0: int object at 0xb37468>,)
```

재밌는 점은, 여기에 출력된 위치 `0xb37468`가 실제 `x`의 저장 위치와 똑같다는 것입니다.

```
def outer(x):
    print(hex(id(x)))

    def inner():
        print(x)

    return inner


inner = outer(3)
# 0xb37468
print(inner.__closure__)
# (<cell at 0x79d6681706a0: int object at 0xb37468>,)
```

`__closure__`은 `tuple[CellType]` 형식으로 저장되어 있는데, 실제로 내부 값을 확인하고 싶으면 `cell_contents`를 사용할 수 있습니다.

```
print(inner.__closure__[0].cell_contents)
# 3
```

한 편, `__closure__`는 딕셔너리가 아니므로 `__closure__[x]` 같은 방법으로 `x`를 참조할 수 없습니다. 그래서, 자유 변수들을 따로 모아두는 공간이 필요합니다. `__code__`는 함수에서 실행을 위해 중요한 정보들을 따로 모아두는 변수입니다. 그리고 `__code__.co_freevars`에 자유 변수가 나열되어 있습니다.

```
print(inner.__code__.co_freevars)
# ('x',)
```

그리고 이 인덱스는 `__closure__`에 저장된 값의 인덱스와 일치합니다. 그러므로 다음과 같이 `x`의 값을 참조할 수 있겠습니다.

```
print(inner.__closure__[inner.__code__.co_freevars.index("x")].cell_contents)
# 3
```

마지막으로, 각종 함수 유틸리티에 대해 알아보고 정리합시다.

고차 함수에 사용하기

고차 함수는 함수를 인자로 받는 함수들을 말합니다. `map` 외에도 다양한 고차 함수들을 배워봅시다.

`partial` 에서

함수를 유용하게 사용하기 위해서 Python은 `functools` 모듈을 제공합니다. 한 편 여기서 제공하는 많은 기능은 아직 공부해보지 않은 *데코레이터*에 의존하기 때문에, 오늘은 `partial` 클래스만 사용해보겠습니다. `partial` 클래스는 함수에 인자를 *미리 넣어* 새로운 함수를 만듭니다.

```
from functools import partial

plus_one = partial(add, 1)
print(plus_one(3))
# 4
```

위의 `plus_one` 함수는 `add` 함수에서 인자 `a`에 미리 `1`을 넣어둡니다. 그래서 나머지 `b`만 넣으면 완성이 되는 것이죠.

`filter` 에서

Python의 `filter` 함수는 함수와 반복 가능한 객체를 받아, 함숫값이 `True`인 것들만을 순서대로 반환합니다. 예를 들어, 아래와 같이 짝수만 반환하도록 할 수 있습니다.

```
for i in filter(lambda x: x%2 == 0, range(10)):
    print(i, end=", ")
# 0, 2, 4, 6, 8,
```

`sorted` 에서

Python에서 `sorted` 함수는 기본적으로 내부 값들을 순서대로 정렬하여 리스트로 반환해주는 함수입니다. 예를 들어, `random.shuffle` 로 섞었다고 해도 복구합니다.

```
from random import shuffle
x = [1, 2, 3, 4, 5]
shuffle(x)
print(sorted(x))
# [1, 2, 3, 4, 5]
```

한 편, `sorted` 에 `key` 로 함수를 넣어주면 정렬 기준을 정해줄 수 있습니다. `key` 인자에 들어간 함수는 반드시 `__le__` 가 정의된 값을 반환해야 합니다.

(`reverse=True` 는 내림차순 정렬을 유도합니다.)

```
print(sorted(x, key=lambda v: len(bin(v)), reverse=True))
# [4, 5, 2, 3, 1]
```

> 수고하셨습니다.
> 과제 바로 열기