Day 6 - 함수

함수 기본

함수는 코드에 대한 이름표입니다.

```
def hello_world():
    print("안녕하세요")
```

파이썬 인터프린터는 위 함수 정의 (function defionition)을 읽고 다음 동작을 진행합니다.

- 메모리에 함수 **객체**를 할당한다
- 함수 이름을 함수 객체에 바인딩 한다

함수 구조는 아래와 같습니다

```
def 함수_이름(매개변수): #매개 변수는 있을수도, 없을 수도 있습니다
    수행할_문장1
    수행할_문장2
    ....
    return 무언가 # 리턴은 있을수도, 없을 수도 있습니다
```

아래와 같이 다른 코드에서 호출할 수 있습니다

```
def say_hello_world(target):
    return str(f"hello {target}")

if __name__ == "__main__":
    print(say_hello_world("dongwoo")) # hello dongwoo
    print(say_hello_world("jaeun")) # hello jaeun
```

왜 작성하나요?

- 반복적인 코드를 재사용 하기 위해서
    - 두번 이상 사용될 코드는 함수로
- 목적과 구현을 분리하는 경우

함수 잘 만들기

함수를 작성할때 꼭 생각해봐야 할 것

- 함수의 길이가 너무 커지면, 분리가 가능하지 않은지 생각해보세요
    - 읽기도 어렵고, 유지보수 하기도 어렵습니다
        - 버그를 찾기 어려울수도 있습니다
    - 참고로, 구글에서는 40줄을 기준으로 삼고 있습니다
- 가급적 하나의 기능만 하는 함수를 만드세요
- 함수 구현 사이사이에 실행 코드 **절대** 넣지 마세요

예를 들어, 아래 코드는

```
def print_owing(invoice):
    outstanding = 0

        # 배너 출력
    print("*****************")
    print("*** 고객 채무 ***")
```

```
    print("*****************")

    # 미해결 채무(outstanding)을 계산
    for o in invoice.orders:
        outstanding += o.amount

    # 마감일(dueDate) 기록
    today = datetime.date.today()
    invoice.due_date = today + datetime.timedelta(days=30)

    # 세부 사항 출력
    print(f"고객명: {invoice.customer}")
    print(f"채무액: {outstanding}")
    print(f"마감일: {invoice.due_date}")
```

이렇게 바꿀 수 있습니다

```
def print_owing_refactored(invoice):
    print_banner("고객 채무")                  # 배너 출력
    outstanding = calculate_outstanding(invoice) # outstanding 계산
    record_due_date(invoice)                   # 마감일 기록
    print_details(invoice, outstanding)        # 세부 사항 출력

def print_banner(contents: str):
    print("*****************")
    print(f"*** {contents} ***")
    print("*****************")

def calculate_outstanding(invoice):
    result = 0
    for o in invoice.orders:
        result += o.amount
    return result

def record_due_date(invoice):
    today = datetime.date.today()
    invoice.due_date = today + datetime.timedelta(days=30)
```

```
def print_details(invoice, outstanding):
    print(f"고객명: {invoice.customer}")
    print(f"채무액: {outstanding}")
    print(f"마감일: {invoice.due_date}")
```

이렇게 기존 구현 로직은 바꾸지 않으면서, 유지보수성을 올리는 작업을 **리팩터링 (refactoring)**이라고 합니다

실행 가능한 코드는 **절대로** 함수 구현 사이에 넣지 마세요

버그를 찾을 수 없습니다

```
def 함수1(.....):
    어려운 구현내용

print("나 여기 숨어있지롱")

def 함수2(...):
    어려운 구현내용2
```

파라미터에 대한 힌트 주기

- 파이썬 3.5 버전부터 type annotation이 추가되었습니다
- 최근에는 코드 작성시 사실상 표준이 되어가고 있습니다
    - 파라미터에 들어가는 타입을 체크해, 빌드 타임에 버그 유무를 파악할 수 있습니다

파이썬은 약타입 언어로, 타입에 대한 유연함을 가질 수 있습니다

```
apple = 1 # 이런 코드 이후에
apple = str("사과") # 이런 코드가 올 수 있습니다

#=======================

def my_print(content):
    print(content)
```

```
my_print(3) # 같은 함수에 int도 들어가고
my_print("안녕") # str도 들어갑니다
```

이런 유연함은 애플리케이션의 구조나 복잡성이 커질때, 치명적인 버그로 작용할 수 있습니다

따라서 일반적으로 협업이 요구되는 어플리케이션에선 type hinting이라는 별도 개발 프로세스를 도입합니다

```
num: int = 1 # 이렇게까진 안하는 경우도 많지만

# 이렇게 prameter와 return 값의 타입 힌트는 일반적으로 사용합니다
def repeat(message: str, times: int) -> list[str]:
    return [message] * times
```

다만 런타임에서 이런 타입 힌트를 어겨도 에러를 만드는 것은 아닙니다

```
print(repeat("hello", 3)) # ['hello', 'hello', 'hello']
print(repeat(1, 3))       # [1, 1, 1]
```

아래와 같은 형식으로 타입 힌트를 추가할 수 있습니다

```
name: str = "John Doe"

age: int = 25

emails: list[str] = ["john1@doe.com", "john2@doe.com"]

address: dict[str, str] = {
  "street": "54560 Daugherty Brooks Suite 581",
  "city": "Stokesmouth",
  "state": "NM",
  "zip": "80556"
```

```
}

def 함수명(<필수 인자>: <인자 타입>, <선택 인자>: <인자 타입> = <기본값>) -> <반
    return something
```

좀 더 복잡한 타입 어노테이션을 추가할때는, typing 내장 모듈을 사용할 수 있습니다

```
from typing import Dict, List, Optional, Set, Tuple, Union

nums: List[Optional[int]] = [1, None, 2]

unique_nums: Set[Union[int, float]] = {6, 7.5}

vision: Dict[str, float] = {"left": 1.0, "right": 0.9}

john: Tuple[int, str, List[float]] = (25, "John Doe", [1.0, 0.9])
```

일급 객체로서의 함수

함수는 객체다

처음 함수의 동작 과정을 설명할 때, 이렇게 설명했습니다

- 메모리에 함수 **객체**를 할당한다

왜 굳이 객체에 강조 표시를 했을까요? 그것은 파이썬에서는 함수도 객체기 때문입니다 (C 언어와는 다릅니다)

아래 코드를 실행해보면, 함수가 객체란 것을 알 수 있습니다

```
import sys

def my_print():
    print("hello world")
```

```
print(hex(id(my_print))) # 함수객체 주소
print(sys.getrefcount(my_print)) # 함수 객체의 레퍼런스 카운트
print(sys.getsizeof(my_print)) # 함수 객체의 사이즈
print(my_print.__annotations__) # 함수 객체의 어노테이션
print(type(my_print)) # 함수 객체의 타입 == class 'function'
```

함수를 관리하는 PyFunctionObject는 대략 이렇게 생겼습니다

https://img1.daumcdn.net/thumb/R1280x0/?scode=mtistory2&fname=https%3A%2F%2Fblog.kakaocdn.net%2Fdn%2FOGCK1%2Fbtq9Egzug5x%2FYw9LbAjzS3w1KjwojlcH2K%2Fimg.png

함수에 내장된 __doc__ 을 실행하는 예제입니다.

```
def sample():
    """This is a sample function"""
    return 1

print(sample.__doc__) # This is a sample function
```

이렇게 객체 내부에 미리 구현되어 있는 함수들을 매직 매서드(magic method)라고 합니다

.

일급 객체로서의 함수

파이썬의 함수는 일급함수입니다. (사실 요즘 언어 대부분이 이렇습니다)

일급 함수란, 일급객체로서의 함수를 줄여 표현하는 말인데, 모든 함수는 일급함수이므로 그렇게 좋은 표현은 아닐 수도 있겠습니다.

일급 객체(first class object) 는 다음 성질을 만족하는 프로그램 객체를 뜻합니다

- 런타임에 생성할 수 있다
- 데이터 구조체의 변수나 요소에 할당할 수 있다

```
def factorial(n):
    '''Factorial Function → n : int'''
    if n == 1:
        return 1
    return n * factorial(n-1)

var_func = factorial
```

- 함수 인수로 전달할 수 있다
- 함수 결과로 반환할 수 있다

```
# ...위 코드에 이어서

print(var_func)
print(var_func(10))
print(list(map(var_func, range(1,11))))
# 출력
<function factorial at 0x000001CF5FB4DFC0>
3628800
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]

# 요즘은 list_comprehention을 사용하지만, 이전에는 이렇게 썼습니다
# map은 iterator의 각 요소를  파라미터로 입력받은 함수의 파라미터로 넘깁니다
# 이후에는 반환받은 값들을 한대 묶어 반환합니다
```

이런 일급함수의 특성을 생각하면, 이런 방식의 코드 구현도 가능하게 됩니다

아래 예제는 전략 패턴 (Strategy Pattern)의 구현 예제입니다.

> **전략 패턴이란?**
>
> 알고리즘을 여러 개 사전에 정의하고, 이를 런타임에 적절한 알고리즘으로 선택할 수 있게 하는 디자인 패턴

```
def taxi_algorithm(start_point: str,end_point: str) → str:
    return f"{start_point}에서 {end_point}로 택시를 타고 가는데 걸리는 시간은 300분
```

```
def ktx_algorithm(start_point: str,end_point: str) → str:
    return f"{start_point}에서 {end_point}로 KTX를 타고 가는데 걸리는 시간은 120분

class TimeCalculator:
    def __init__(self,algorithm):
        self.algorithm = algorithm

    def calculate(self,start_point: str,end_point: str) → str:
        return self.algorithm(start_point,end_point)

if __name__ == "__main__"():
    taxi = TimeCalculator(taxi_algorithm)
    ktx = TimeCalculator(ktx_algorithm)

    print(taxi.calculate("서울","부산"))
    print(ktx.calculate("서울","부산"))
```