세상의 속도를
따라잡고 싶다면

점프 투 파이썬

박응용 지음(위키독스 운영자)

이지스퍼블리싱

07

파이썬 날아오르기

07-1 파이썬과 유니코드

최초의 문자 셋, 아스키코드

- 컴퓨터는 0과 1이라는 값만 인식할 수 있는 기계장치이므로 우리가 입력하는 문자를 컴퓨터가 인식할 수 있도록 바꿔야 함

- 과거부터 지금까지 사용하는 유일한 방법은 숫자마다 문자를 매핑(mapping)하는 문자 셋(character set)을 만드는 것

- 아스키(ASCII) 코드란 미국에서 최초로 만든 문자 셋 표준
  - 영어권 국가에서 사용하는 영문자, 숫자 등 127개의 문자 처리 가능

07-1 파이썬과 유니코드

유니코드의 등장

- 비영어권 국가에서도 문자를 컴퓨터로 표현하고자 하여 아스키를 사용할 수 없음
- 서유럽 문자셋인 ISO8859와 한국 문자 셋인 KSC5601 등이 등장하기 시작
- 모든 나라의 문자를 포함하도록 만든 유니코드(unicode)의 등장,
  세계 표준으로 자리 잡음

07-1 파이썬과 유니코드

유니코드로 문자열 다루기

인코딩하기

- 유니코드 문자열을 바이트 문자열로 바꾸는 방법

```
>>> a = "Life is too short"
>>> b = a.encode('utf-8')
>>> b
b'Life is too short'
>>> type(b)
<class 'bytes'>
```

  - utf-8을 인수로 인코딩

- 한글을 오류 없이 인코딩하는 방법

```
>>> a = "한글"
>>> a.encode('euc-kr')
b'\xc7\xd1\xb1\xdb'
>>> a.encode('utf-8')
b'\xed\x95\x9c\xea\xb8\x80'
```

  - euc-kr 또는 utf-8을 인수로 인코딩

07-1 파이썬과 유니코드

유니코드로 문자열 다루기

디코딩하기

- 인코딩한 바이트 문자열을 유니코드 문자열로 변환하려면 인코딩 방식을 똑같이 맞춰줘야 함

```
>>> a = '한글'
>>> b = a.encode('euc-kr')
>>> b.decode('euc-kr')
'한글'
```

- 잘못된 인코딩 방식으로 디코딩하려고 하면 오류 발생

```
>>> b.decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc7 in position 0: invalid
continuation byte
```

07-1 파이썬과 유니코드

유니코드로 문자열 다루기

입출력과 인코딩

파일을 읽거나 네트워크를 주고받을 때 추천하는 방법

① 입력으로 받은 바이트 문자열은 되도록 빨리 유니코드 문자열로 디코딩한다.
② 함수나 클래스 등에서는 유니코드 문자열만 사용한다.
③ 입력에 대한 결과를 전송하는 마지막 부분에서만 유니코드 문자열을 바이트 문자열로 인코딩해서 반환한다.

07-1 파이썬과 유니코드

▪ 유니코드로 문자열 다루기

▪ 입출력과 인코딩

▪ 예) euc-kr 방식으로 작성한 파일을 읽고 변경하여 저장하기

```
# 1. euc-kr로 작성된 파일 읽기
with open('euc_kr.txt', encoding='euc-kr') as f:
    data = f.read()  <-- 유니코드 문자열

# 2. unicode 문자열로 프로그램 수행하기
data = data + "\n" + "추가 문자열"

# 3. euc-kr로 수정된 문자열 저장하기
with open('euc_kr.txt', encoding='euc-kr', mode='w') as f:
    f.write(data)
```

- open( ) 함수가 encoding으로 읽은 문자열은 유니코드 문자열
- encoding 항목을 생략하면 기본값으로 utf-8이 지정됨

07-1 파이썬과 유니코드

유니코드로 문자열 다루기

소스 코드의 인코딩

- 소스 코드 파일이 현재 어떤 방식으로 인코딩되었는지를 뜻함
- 소스 코드도 파일이므로 인코딩 타입이 반드시 필요

- 파이썬 소스 코드 가장 위에 인코딩을 명시하는 문장 병기

```
# -*- coding: utf-8 -*-
```

```
# -*- coding: euc-kr -*-
```

07-2 클로저와 데코레이터

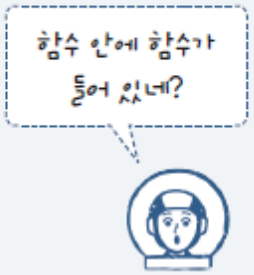
클로저(closure)란?

- 함수 안에 내부 함수(inner function)를 구현하고 그 내부 함수를 리턴하는 함수
- 이때 외부 함수는 자신이 가진 변숫값 등을 내부 함수에 전달할 수 있음

```
def mul(m):
    def wrapper(n):
        return m * n
    return wrapper

if __name__ == "__main__":
    mul3 = mul(3)
    mul5 = mul(5)

    print(mul3(10))
    print(mul5(10))
```



실행 결과

```
30
50
```

- 외부 함수 mul 안에 내부 함수 wrapper를 구현
- 외부 함수는 내부 함수 wrapper를 리턴

07-2 클로저와 데코레이터

▪ 데코레이터(decorator)란?

- 기존 함수를 바꾸지 않고 기능을 추가할 수 있게 만드는 클로저
- @ 문자를 이용해 함수 위에 적용하여 사용 가능
  - 파이썬은 함수 위에 **@+함수명**이 있으면 데코레이터 함수로 인식
- 예) 함수의 실행 시간 측정하기

```
import time

def elapsed(original_func):                          # 기존 함수를 인수로 받는다.
    def wrapper():
        start = time.time()
        result = original_func()                     # 기존 함수를 수행한다.
        end = time.time()
        print("함수 수행 시간: %f초" % (end - start))  # 기존 함수의 수행 시간을 출력한다.
        return result                                # 기존 함수의 수행 결과를 리턴한다.
    return wrapper

@elapsed
def myfunc():
    print("함수가 실행됩니다.")

# decorated_myfunc = elapsed(myfunc)
# decorated_myfunc()

myfunc()
```

실행 결과

```
함수가 실행됩니다.
함수 수행 시간: 0.000029초
```

07-2 클로저와 데코레이터

데코레이터(decorator)란?

- 데코레이터는 기존 함수가 어떤 입력 인수를 취할지 알 수 없기 때문에 기존 함수의 입력 인수에 상관없이 동작하도록 만들어야 함

- 기존 함수의 입력 인수를 알 수 없는 경우 *args와 **kwargs 매개변수 이용

```
import time

def elapsed(original_func):                          # 기존 함수를 인수로 받는다.
    def wrapper(*args, **kwargs):                    # *args, **kwargs 매개변수 추가
        start = time.time()
        result = original_func(*args, **kwargs)      # *args, **kwargs를 입력 인수로
기존 함수 수행
        end = time.time()
        print("함수 수행 시간: %f초" % (end - start))  # 수행 시간을 출력한다.
        return result                                # 함수의 결과를 리턴한다.
    return wrapper

@elapsed
def myfunc(msg):
    """ 데코레이터 확인 함수 """
    print("'%s'을 출력합니다." % msg)

myfunc("You need python")
```

실행 결과

```
'You need python'을 출력합니다.
함수 수행 시간: 0.000027초
```

07-3 이터레이터와 제너레이터

이터레이터(iterator)란?

- next 함수 호출 시 계속 그다음 값을 리턴하는 객체
- 리스트처럼 반복 가능(iterable)하다고 해서 이터레이터는 아니지만, iter 함수를 이용해 이터레이터로 만들 수 있음
- for 문을 이용해 이터레이터 값을 가져오는 것이 가장 일반적
  - for 문이나 next로 그 값을 한 번 읽으면 그 값을 다시는 읽을 수 없음

```
>>> a = [1, 2, 3]
>>> ia = iter(a)
>>> for i in ia:
...     print(i)
...
1
2
3
>>> for i in ia:
...     print(i)
...
>>>  <— 값이 출력되지 않는다.
```

07-3 이터레이터와 제너레이터

▪ 이터레이터 만들기

- 클래스에 __iter__와 __next__라는 2개의 메서드를 구현하여 만드는 방법
  - __iter__ 메서드는 해당 클래스로 생성한 객체를 반복 가능한 객체로 만드는 역할
  - __next__ 메서드는 반복 가능한 객체의 값을 차례대로 반환하는 역할

```
class MyIterator:
    def __init__(self, data):
        self.data = data
        self.position = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.position >= len(self.data):
            raise StopIteration
        result = self.data[self.position]
        self.position += 1
        return result

if __name__ == "__main__":
    i = MyIterator([1,2,3])
    for item in i:
        print(item)
```

실행 결과

```
1
2
3
```

07-3 이터레이터와 제너레이터

▪ 이터레이터 만들기

- 입력받은 데이터를 역순으로 출력하는 ReverseIterator 클래스를 만드는 방법

```
class ReverseItertor:
    def __init__(self, data):
        self.data = data
        self.position = len(self.data) -1

    def __iter__(self):
        return self

    def __next__(self):
        if self.position < 0:
            raise StopIteration
        result = self.data[self.position]
        self.position -= 1
        return result

if __name__ == "__main__":
    i = ReverseItertor([1,2,3])
    for item in i:
        print(item)
```

실행 결과

```
3
2
1
```

07-3 이터레이터와 제너레이터

▪ 제너레이터(generator)란?

- 이터레이터를 생성해 주는 함수

- 이터레이터와 마찬가지로 next 함수 호출 시 그 값을 차례대로 얻을 수 있음
  - 차례대로 결과를 반환하기 위해 return 대신 yield 키워드 사용

```
>>> def mygen():
...     yield 'a'
...     yield 'b'
...     yield 'c'
...
>>> g = mygen()
```

```
>>> next(g)
'a'
```

```
>>> next(g)
'b'
```

```
>>> next(g)
'c'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- 네 번째 next를 호출할 때는 더 리턴할 값이 없으므로 StopIteration 예외 발생

07-3 이터레이터와 제너레이터

▪ 제너레이터 표현식(generator expression)

- 제너레이터는 def를 이용한 함수로 만들 수 있지만, 튜플 표현식으로 좀 더 간단하게 만들 수도 있음
  - 리스트 컴프리헨션(list comprehension) 구문과 비슷하지만 리스트 대신 튜플을 이용함

```
def mygen():
    for i in range(1, 1000):
        result = i * i
        yield result

gen = mygen()

print(next(gen))
print(next(gen))
print(next(gen))
```

```
gen = (i * i for i in range(1, 1000))
```

실행 결과

```
1
4
9
```

07-3 이터레이터와 제너레이터

제너레이터와 이터레이터

- 클래스를 이용해 이터레이터를 작성하면 좀 더 복잡한 행동을 구현 가능
- 제너레이터를 이용하면 간단하게 이터레이터를 만들 수 있음
- 이터레이터의 성격에 따라 클래스로 만들 것인지, 제너레이터로 만들 것인지를 선택

- 예) (i * i for i in range(1, 1000)) 제너레이터를 이터레이터 클래스로 구현

```
class MyIterator:
    def __init__(self):
        self.data = 1

    def __iter__(self):
        return self

    def __next__(self):
        result = self.data * self.data
        self.data += 1
        if self.data >= 1000:
            raise StopIteration
        return result
```

07-3 이터레이터와 제너레이터

▪ 제너레이터 활용하기

- 시간이 오래 걸리는 작업을 한꺼번에 처리하기보다는 필요한 경우에만 호출하여 사용하는 **느긋한 계산법**(lazy evaluation)

- 예) 총 실행 시간이 1초인 longtime_job 함수를 5번 실행해 리스트에 그 결괏값을 담고 그 첫 번째 결과값을 호출

```
import time

def longtime_job():
    print("job start")
    time.sleep(1)
    return "done"

list_job = (longtime_job() for i in range(5))
print(next(list_job))
```

실행 결과

```
job start
done
```

07-4 파이썬 타입 어노테이션

동적 언어와 정적 언어

- 파이썬은 **동적 프로그래밍 언어**(dynamic programming language)
  - 프로그램 실행 중에 변수의 타입을 동적으로 바꿀 수 있음

```
>>> a = 1
>>> type(a)
<class 'int'>
```

```
>>> a = "1"
>>> type(a)
<class 'str'>
```

- 자바는 **정적 프로그래밍 언어**(static programming language)
  - 한 번 변수에 타입을 지정하면 지정한 타입 외에 다른 타입은 사용할 수 없음

```
int a = 1;   <- a 변수를 int형으로 지정
a = "1";   <- a 변수에 문자열을 대입할 수 없으므로 컴파일 오류 발생
```

07-4 파이썬 타입 어노테이션

동적 언어와 정적 언어

동적 언어의 장단점

- 장점: 유연한 코딩이 가능하므로 쉽고 빠르게 프로그램을 만들 수 있음
  타입 체크를 위한 코드가 없으므로 비교적 깔끔한 소스 코드 생성 가능

- 단점: 프로젝트의 규모가 커질수록 타입을 잘못 사용해 버그가 생길 확률도 높아짐

07-4 파이썬 타입 어노테이션

- **파이썬 타입 어노테이션(type annotation)**
  - 동적 언어의 단점을 극복하기 위해 파이썬 3.5 버전부터 지원하기 시작한 기능
  - 타입의 힌트를 알려 주는 정도의 기능만 지원
  - 동적 언어의 장점을 잃지 않고 기존에 작성된 코드와 호환 가능

```
num: int = 1
```

```
def add(a: int, b: int) -> int:
    return a + b
```

07-4 파이썬 타입 어노테이션

- **mypy**
  - 파이썬 타입 어노테이션은 체크가 아니 힌트 정도의 역할
  - 적극적으로 파이썬 어노테이션을 활용하려면 mypy를 사용하는 것이 좋음
  - mypy는 표준 라이브러리가 아니므로 설치한 후에 사용 가능

```
c:\doit>pip install mypy
```

  - 예) 타입 어노테이션으로 매개변수의 타입을 명시하더라도 다른 타입의 인수 입력 가능

```
def add(a: int, b: int) -> int:
    return a + b

result = add(3, 4)
print(result)
```

```
C:\projects\pylib\>mypy typing_sample.py
Success: no issues found in 1 source file
```

감사합니다.