Pythonic Code

Pythonic Code

Python에는 재미있는 몇가지 기능이 있다.

1. list comprehension
2. Walrus Operator
3. Packing and Unpacking
4. Generator

모든 언어는 각 언어별로 특징이 있다. 각 언어의 특징에 맞추어 코드를 작성해주어야만 코드도 그 언어에 맞게 깔끔해지며, 완성된 어플리케이션의 성능에도 영향을 준다.

위 네가지 항목은 Python의 특징이며 추후 조금 더 “파이썬 스러운”코드를 작성하는 데에 도움을 줄것이다.

List Comprehension

기본적으로 Python에서 루프(loop)를 생성하는 문법 중 하나인 for 문을 활용하여 간단하게 리스트를 생성하는 구문이다. 코드가 간결해지고 동일한 코드임에도 for문보다 실행속도가 현저히 빠르다는 장점이 있다. 단점으로는 for문보다 유연하지 못하여 복잡한 루프는 `list comprehension`으로 구현하기 어렵다. 또한 루프를 통하여 list를 생성하는 것이 목적이 아닐때는 일반적인 for문이 더 좋은 선택지가 될 수 있다.

Example 1

기본적인 list comprehension의 문법을 알아보기 위해 아래 for문의 코드를 list comprehension코드로 변경해본다.

```
# for statement

l = []
for x in range(1000):
    l.append(x)

m = []
for x in range(1000):
    m.append(str(x))

n = []
for x in range(1000):
    n.append(x + 1)

b = []
for x in range(1000):
    if x % 2 == 0:
```

```
        b.append(x)

k = []
for x in ["1", "hello", "2", "world"]:
    if x.isdigit():
        k.append(int(x))
    else:
        k.append(x)

j = []
for x in [0, 1, 2, 3]:
    if x == 0:
        j.append("success")
    elif x == 1:
        j.append("fail")
    else:
        j.append("else")
```

위 for 문을 list comprehension으로 바꿔보면 아래와 같다.

`리스트괄호 (대괄호, [ ])` 안에 for문을 작성하며, loop를 통하여 반환되는 결과가 가장 처음으로 오는것 외에는 일반적인 for문과 큰 차이는 없다.

만약 list comprehension 안에서 if 문을 함께 사용하는 경우라면, if 문만 사용한다면 for문 뒤에 그리고 if 와 else가 함께 사용된다면 for문 앞에 온다.

list comprehension에는 일반적인 for문과 다르게 `elif`, `continue`, `break`, `pass` 등의 키워드는 사용할 수 없다. 위 for문으로 작성된 코드와 아래 list comprehension으로 작성된 코드를 비교해가며 어떤식으로 각 키워드를 대체하였는지 확인해보자.

```
# list comprehension

l = [x for x in range(1000)]    # l = [0, 1, 2, 3, 4, 5, ..., 998, 999]
m = [str(x) for x in range(1000)]    # m = ["0", "1", "2", ..., "998", "999"]
n = [x + 1 for x in range(1000)]    # n = [1, 2, 3, 4, 5, ..., 999, 1000]
b = [x for x in range(1000) if x % 2 == 0]   # b = [0, 2, 4, ..., 996, 998]
k = [int(x) if x.isdigit() else x for x in ["1", "hello", "2", "world"]]
# [1, "hello", 2, "world"]
j = ["success" if x == 0 else "fail" if x == 1 else "else" for x in [0, 1, 2, 3]]
# ["success", "fail", "else", "else", "else"]
```

Example 2

일반적인 for문을 활용할때와 list comprehension을 활용하여 루프(loop)를 통해 리스트를 만드는데 걸리는 시간을 측정해보자.

```
import time

# Script with for statement

start_time = time.time()

l = []
for i in range(int_range):
    l.append(i)

end_time - time.time()
print(f"RUN TIME: {end_time - start_time}")
```

```
import time

# Script with list comprehension
```

```
start_time = time.time()

l = [i for i in range(int_range)]

end_time = time.time()
print(f"RUN TIME: {end_time - start_time}")
```

위 코드의 `int_range` 값을 조정하며 테스트 한 결과는 아래와 같다.

for loop

int_range	RUN TIME
10_000	0.00874757766
100_000	0.00936269760
10_000_000	0.85900902748
100_000_000	8.63819503784

list comprehension

int_range	RUN TIME
10_000	0.00444412231
100_000	0.00495719909
10_000_000	0.47207117080
100_000_000	5.0039086341

위 결과에서 볼 수 있듯 loop의 크기가 커지면 커질수록 소요 시간의 차이또한 점점 커지는것을 확인할 수 있다.

Walrus Operator

`Walrus operator(바다코끼리 연산자)`라고 불리는 이 연산자의 정식 명칭은 `대입표현식(Assignment Expression)`이다. 정식명칭과 다르게 `바다코끼리 연산자` 라는 명칭이 붙은 이유는 해당 표현식을 위한 연산자의 생김새, `:=` , 가 바다코끼리를 닮아서이다.

`대입표현식`이라는 이름에서 유추할 수 있듯, 표현식에 값을 대입하고, 바로 대입된 값을 사용할 수 있게 해준다.

> 💡 해당 문법은 Python 3.8버전에서부터 도입되었으며, 그 이전버전에서는 사용이 불가능 하다.

Example 1

`대입표현식`이 등장하기 이전, Python ≤ 3.7,까지는 아래와 같이 코드를 작성해야하만 했다.

```
a = [1]
length = len(a)
target = 5

while length < target:
    print(f"list 'a' has {length} elements. Need {target - length} more elements.")
    a.append(a[-1] + 1)
    print(a)
    length = len(a)
    if length == target:
        print("Done")

# list 'a' has 1 elements. Need 4 more elements.
# [1, 2]
# list 'a' has 2 elements. Need 3 more elements.
# [1, 2, 3]
# list 'a' has 3 elements. Need 2 more elements.
# [1, 2, 3, 4]
# list 'a' has 4 elements. Need 1 more elements.
# [1, 2, 3, 4, 5]
# Done
```

위 코드와 같이 계속해서 `length` 변수를 while문 내부에서 업데이트 해주거나, 리스트의 길이를 확인할때 매번 `len()` 함수를 수행해줘야 한다.

위 코드를 `대입표현식`을 넣어 조금 더 간단하게 바꿔보자.

```
a = [1]
target = 5

while (length := len(a)) < target:
    print(f"list 'a' has {length} elements. Need {target - length} more elements.")
    a.append(a[-1] + 1)
    print(a)
    if length == target:
        print("Done")

# list 'a' has 1 elements. Need 4 more elements.
# [1, 2]
# list 'a' has 2 elements. Need 3 more elements.
# [1, 2, 3]
# list 'a' has 3 elements. Need 2 more elements.
# [1, 2, 3, 4]
# list 'a' has 4 elements. Need 1 more elements.
# [1, 2, 3, 4, 5]
# Done
```

while 문에 `(length := len(a))` 구문을 추가하여 length라는 변수에 a 리스트의 길이를 계속해서 대입해주었다. 이렇게 생성된 변수 length는 while문 내부에서 계속해서 사용할 수 있게 된다.

`대입표현식` 을 감싸고있는 괄호, `()`, 는 필수적으로 작성해주어야 하는것은 아니다. 괄호 없이 `length := len(a)` 만 작성해주어도 문법상으로는 아무 오류가 없다. 다만 가독성을 해치고 코드가 난해해 질 수 있으므로 괄호를 사용하여 `대입표현식`을 사용했다는 것을 명시해준다.

`walrus operator` 또는 `assignment expression` 키워드로 검색하면 더 많은 예제를 찾아볼 수 있다.

Packing and Unpacking

이전 함수 토픽을 공부할 때 우리는 `*args`와 `**kwargs` arguments 를 공부했었다. 해당 두 키워드에서 사용했던 `*` 그리고 `**` 연산자는 `packing`과 `unpacking`을 모두 활용하는 좋은 예시이다.

Packing

패킹은 말그대로 여러개의 데이터를 하나의 데이터로 합치는 것이다.

아래 간단한 예제를 보자.

```
# normal way to create a tuple

tup_1 = (1, 2, 3, 4)
tup_2 = (1, )
print(tup_1, tup_2)
# (1, 2, 3, 4, ) (1, )

# using packing

tup_3 = 1, 2, 3, 4
tup_4 = 1,
print(tup_3, tup_4)
# (1, 2, 3, 4, ) (1, )
```

일반적으로 tuple을 생성할 때 괄호를 사용하여 생성을 하게 되지만, 괄호를 생략하고 하나의 변수 명과 등호 문자 뒤에 여러개의 객체를 입력해주면 자동으로 하나의 튜플이 생성된다. 여러개의 객체를 하나의 객체로 묶어 합쳐주는 `packing`의 예시이다.

이제 `*args` 와 `**kwargs`를 통하여 인자값을 받아올때 packing이 자동적으로 이루어 진다. 아래 예제를 통해 확인해보자.

```
def pack_arguments(*args):
    print(args)

pacK_arguments(1, 2, 3, 4, 5)

# (1, 2, 3, 4, 5)
```

위 예제에서 `pack_arguments` 함수에 총 5개의 인자값을 넣어주었다. 그렇게 전달된 5개의 인자들은 `*args`의 `*` 연산자에 의하여 하나의 tuple 데이터로 합쳐진다.

`**` 연산자를 사용하게되면 tuple대신 dictionary 자료형으로 데이터가 합쳐지게된다.

```
def pack_arguments(**kwargs):
    print(kwargs)

pack_arguments('a'=1, 'b'=2, 'c'=3, 'd'=4)

# {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

이렇게 여러 데이터(객체)를 하나의 변수(객체)로 합쳐주는것을 packing이라고 한다.

Unpacking

그러면 그 반대의 경우도 살펴보자.

`Packing` 과 반대로 하나의 객체를 여러개의 객체로 분리할 수 있다.

```
my_values = [1, 2, 3]

# normal way to assign values to separate variables
a = my_values[0]
b = my_values[1]
c = my_values[2]

# using unpacking
a, b, c = my_values
```

위 예제와 `my_values`리스트 내의 각 원소를 각각 `a, b, c` 변수에 한번에 대입하여 새로운 객체를 생성해줄 수 있다.

하지만 생성하고자 하는 변수의 개수와 리스트 내 원소의 개수가 다르다면 아래와 같은 `ValueError`가 발생할 수 있다.

```
my_values = [1,2,3,4,5]

# this occures ValueError
a, b, c = my_values

# ValueError: too many values to unpack (expected 3)
```

이 경우 다시 위 `packing` 개념을 활용하여 `a` 와 `b` 에 각 하나씩 원소를 할당하고, 그 외 나머지 원소는 모두 c 변수에 `packing`하여 할당해줄 수 있다.

```
my_values = [1,2,3,4,5]

# Assign 1 to a, 2 to b and pack the rest to c
a, b, *c = my_values

print(a)
print(b)
print(c)
# 1
```

```
# 2
# [3, 4, 5]
```

만약 각 `a, b, c` 변수에 `1, 2, 3` 을 할당하고, 그 외 4, 5는 별도의 객체로 할당하고 싶지 않다면 아래와 같이 `_`를 사용하여 처리할 수 있다. Python에서 변수 명을 `_` 로 하면 `throwaway` 변수라 하여 Linux 의 /dev/null 과 같이 값을 버리는 용도로 사용할 수 있다.

```
my_values = [1, 2, 3, 4, 5]

# Assign 1 to a, 2 to b, 3 to c and throwaway the rest
a, b, c *_ = my_values

print(a)
print(b)
print(c)
# 1
# 2
# 3
```

`Packing`과 동일하게 `*`와 `**`연산자를 통하여 인자값을 전달할 때에도 `unpacking`을 할 수 있다.

```
def unpack_arguments(*args, **kwargs):
    print(args, kwargs)

lst = [1,2,3,4,5]

unpack_arguments(lst)
unpack_arguments(*lst)

# ([1, 2, 3, 4, 5], ) {}
# (1, 2, 3, 4, 5) {}
```

`unpack_arguments` 함수를 호출할 때 인자값에 `*` 연산자를 사용하고 안하고의 차이는 위 예제에서 볼 수 있듯, `lst` 리스트 자체를 하나의 객체로 받아오는지, 아니면 해당 리스트의 각각의 원소를 각각의 객체로 받아오는지 차이가 있다.

`**`연산자 또한 동일하게 dictionary객체를 unpacking 할수 있다.

```
def unpack_arguments(*args, **kwargs):
    print(args)
    print(kwargs)

dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

unpack_arguments(dic)
unpack_arguments(**dic)

# ({'a': 1, 'b': 2, 'c': 3, 'd': 4}, ) {}
# () {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

`list comprehension`과 `unpacking`을 사용하여 아래와 같은 응용도 가능하다.

```
a, b, c, d, e, *_ = [x for x in range(10)]

print(a, b, c, d, e)
# 0, 1, 2, 3, 4
```

Generator

마지막으로 Python의 `Generator(제너레이터)`와 관련된 내용을 살펴보자.

Generator는 `lazy evalution(지연 평가방식)` 방식을 사용하는 `iterator`로 for문과 비슷하지만, 루프를 한번에 시작부터 끝까지 실행하는 것이 아닌 호출될 때 마다 하나의 루프를 수행한다는 점에 차이점이 있다.

이러한 `지연 평가방식` 은 큰 데이터를 다룰 때 메모리 활용에 큰 도움을 주게 된다. 예를들어 1억개의 데이터가 들어있는 하나의 리스트 객체를 생성한다고 가정해보자. 이 때 일반적인 for 또는 list comprehension을 사용하게 되면 실행과 동시에 모든 데이터는 메모리에 올라가게된다. 따라서 해당 데이터가 지금 당장 필요하지 않더라도 프로그램 runtime 동안 (또는 garbage collector에 의하여 정리되기 전까지)은 메모리의 공간을 계속 차지하고 있게된다.

하지만 `지연 평가방식`의 Generator를 활용한다면 오로지 다음 순서만 기억한 상태로 generator 객체가 생성되며, 호출이 되었을 때 다음 값을 산출할 만큼의 메모리를 사용하며 그 이후 다시 실행 대기 상태가 된다. 따라서 현재 필요한 데이터 만큼만 꺼내서 사용할 수 있다는 장점이 있다.

`Generator`를 생성하는 방법은 매우 간단하다. 함수에서 결과값을 반환할 때 `return` 대신 `yield`키워드를 사용하면 된다. 생성된 `generator`를 호출하여 다음 값을 받아올 때는 `next` 함수를 사용할 수 있다.

```
# Normal for statement
# get each argument squared

def get_squared(numbers: list):
    squared_result = []
    for number in numbers:
        squared_result.append(number ** 2)
    return squared_result

result = get_squared([1, 2, 3, 4, 5])
print(result)
# [1, 4, 9, 16, 25]
```

```
# Generator

def gen_squared(numbers: list):
    for number in numbers:
        yield number ** 2


gen = gen_squared([1, 2, 3, 4, 5])
print(gen)    # <generator object gen_squared at 0x7fd25156ac81>
next(gen)
# 1
next(gen)
# 4
next(gen)
# 9
next(gen)
# 16
next(gen)
# 25
next(gen)
# StopIterator
```

`generator`함수를 일반 함수와 같이 호출하게되면 하나의 `generator 객체`가 생성된다.

`next`함수에 해당 `generator 객체`를 전달해주면 그 안에 정의되어있는 `__next__()` 함수가 호출되면서 그 다음 값을 반환해주게 된다.

해당 `generator`객체로 부터 모든 값이 반환되고, 더 이상 반환할 값이 없을 땐 `StopIterator` 오류가 발생하게된다.

간단하게 generator를 생성하는 방법, Generator Expression

```
my_gen = (x for x in range(10))
print(my_gen)   # <generator object <genexpr> at 0x7f2ec5asdfa90>
```

List comprehension을 생성할때 괄호를 대괄호가 아닌 중괄호를 활용한다면 손쉽게 Generator 객체를 생성할 수 있다.

초기 임시 비밀번호를 무한히 생성하는 Generator 예제

```
from typing import Iterator
import string
import random


def gen_code() -> Iterator[str]:
    """ generates and yields random code """
    while letters := string.ascii_letters
            strset_1 = "".join([random.choice(letters) for _ in range(5))])
            strset_2 = "".join([random.choice(letters) for _ in range(8))])
            intset_1 = str(random.randrange(100000, 999999))
            yield strset_1 + intset_1 + strset_2

code = gen_code()
while True:
    try:
        name = input("Enter username: ")
        print(f"Hello {name}, your initial password is {next(code)}\n")
    except KeyboardInterrupt:   # when user press ctrl + c, end program
        break
print("Good Bye")


# Enter name: user1
# Hello user1, your initial password is UOHNS222710rxNDjxYc

# Enter name: user2
# Hello user2, your initial password is orIBn911768kZSunjtl

# Enter name: user3
# Hello user3, your initial password is avQbR314720EFeaDMtH
```

Conclusion

당연한 얘기지만 코드를 눈으로 한번만 읽는것 만으로는 그 기능을 제대로 이해하기 어려울 수 있다. 예제를 토대로 하여 새로운 예제를 직접 작성해보거나, for문과 list comprehension을 비교하는 코드를 작성해보거나 직접 generator 함수를 작성하여 활용해보는 등 실습을 하다보면 어느순간 이해가 되는 순간이 있을것이다. 그 때 기존에 본인의 코드에 작성되어있는 코드를 새로운 방식으로 수정하면서 리펙토링을 진행해보자. 조금 더 나은 코드를 작성할 수 있게 될것이다.

이해하는