PEP 8 스타일 가이드를 따르자

파이썬 개선 제안서(Python Enhancement Proposal) #8, 다른 말로 PEP 8은 파이썬 코드를 어떻게 구성할지 알려주는 스타일 가이드다. 문법만 잘 지킨다면 파이썬 코드를 맘대로 작성해도 괜찮다. 하지만 일관성 있는 스타일을 사용하면 유지보수가 더욱 쉬워지고 가독성도 높아진다. 더 큰 커뮤니티에 속한 다른 파이썬 프로그래머들과 공통된 스타일을 공유하면 다양한 프로젝트에서 협업도 가능하다. 하지만 작성한 코드를 읽는 사람이 자신뿐이라고 해도 스타일 가이드를 따르면 나중에 수정하기가 더 쉬울 것이다.

PEP 8에는 파이썬 코드를 명확하게 작성하는 방법이 자세히 나와 있다. 파이썬 언어가 진화하는 데 맞춰 PEP 8도 지속적으로 업데이트될 것이다. 전체 가이드는 온라인(https://www.python.org/dev/peps/pep-0008/)에서 볼 수 있다. 반드시 따라야 하는 몇 가지 규칙은 다음과 같다.

화이트스페이스(whitespace): 파이썬에서 화이트스페이스(공백)는 문법적으로 의미가 있다. 파이썬 프로그래머는 특히 코드의 명료성 때문에 화이트스페이스의 영향에 민감한 편이다.
- 탭이 아닌 스페이스로 들여쓴다.
- 문법적으로 의미 있는 들여쓰기는 각 수준마다 스페이스 네 개를 사용한다.
- 한 줄의 문자 길이가 79자 이하여야 한다.
- 표현식이 길어서 다음 줄로 이어지면 일반적인 들여쓰기 수준에 추가로 스페이스 네 개를 사용한다.
- 파일에서 함수와 클래스는 빈 줄 두 개로 구분해야 한다.
- 클래스에서 메서드는 빈 줄 하나로 구분해야 한다.
- 리스트 인덱스, 함수 호출, 키워드 인수 할당에는 스페이스를 사용하지 않는다.
- 변수 할당 앞뒤에 스페이스를 하나만 사용한다.

명명(naming): PEP 8은 언어의 부분별로 독자적인 명명 스타일을 제안한다. 이 스타일을 따르면 코드를 읽을 때 각 이름에 대응하는 타입을 구별하기 쉽다.
- 함수, 변수, 속성은 lowercase_underscore 형식을 따른다.
- 보호(protected) 인스턴스 속성은 _leading_underscore 형식을 따른다.
- 비공개(private) 인스턴스 속성은 __double_leading_underscore 형식을 따른다.
- 클래스와 예외는 CapitalizedWord 형식을 따른다.
- 모듈 수준 상수는 ALL_CAPS 형식을 따른다.
- 클래스의 인스턴스 메서드에서는 첫 번째 파라미터(해당 객체를 참조)의 이름을 self로 지정한다.
- 클래스 메서드에서는 첫 번째 파라미터(해당 클래스를 참조)의 이름을 cls로 지정한다.

표현식과 문장: 파이썬의 계명(The Zen of Python)에는 “어떤 일을 하는 확실한 방법이 (될 수 있으면 하나만) 있어야 한다.”는 표현이 있다. PEP 8은 표현식과 문장의 본보기로 이 스타일을 정리하고 있다.
- 긍정 표현식의 부정(if not a is b) 대신에 인라인 부정(if a is not b)을 사용한다.
- 길이를 확인(if len(somelist) = = 0)하여 빈 값([] 또는 '')을 확인하지 않는다. if not somelist를 사용하고, 빈 값은 암시적으로 False가 된다고 가정한다.
- ‘비어 있지 않은 값([1] 또는 'hi')에도 위와 같은 방식이 적용된다. 값이 비어 있지 않으면 if somelist 문이 암시적으로 True가 된다.
- 한 줄로 된 if 문, for와 while 루프, except 복합문을 쓰지 않는다. 이런 문장은 여러 줄로 나눠서 명료하게 작성한다.
- 항상 파일의 맨 위에 import 문을 놓는다.

- 모듈을 임포트할 때는 항상 모듈의 절대 이름을 사용하며 현재 모듈의 경로를 기준으로 상대 경로로 된 이름을 사용하지 않는다. 예를 들어 bar 패키지의 foo 모듈을 임포트하려면 그냥 import foo가 아닌 from bar import foo라고 해야 한다.
- 상대적인 임포트를 해야 한다면 명시적인 구문을 써서 from . import foo라고 한다.
- 임포트는 '표준 라이브러리 모듈, 서드파티 모듈, 자신이 만든 모듈' 섹션 순으로 구분해야 한다. 각각의 하위 섹션에서는 알파벳 순서로 임포트한다.

Note

Pylint 도구(http://www.pylint.org/)는 파이썬 소스 코드를 정적 분석하는 도구로 유명하다. Pylint는 자동으로 PEP 8 스타일 가이드를 강요하고 파이썬 프로그램에서 일어나는 많은 유형의 일반적인 오류를 검출한다.

핵심 정리

- 파이썬 코드를 작성할 때 항상 PEP 8 스타일 가이드를 따르자.
- 큰 파이썬 커뮤니티에서 다른 사람과 원활하게 협업하려면 공통된 스타일을 공유해야 한다.
- 일관성 있는 스타일로 작성하면 나중에 자신의 코드를 더 쉽게 수정할 수 있다.

발췌: '파이썬 코딩의 기술

for와 while 루프 뒤에는 else 블록을 쓰지 말자 (저자의 의견임!)

파이썬의 루프에는 대부분의 다른 프로그래밍 언어에는 없는 추가적인 기능이 있다. 루프에서 반복되는 내부 블록 바로 다음에 else 블록을 둘 수 있는 기능이다.

```
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')

>>>
Loop 0
Loop 1
Loop 2
Else block!
```

놀랍게도 else 블록은 루프가 종료되자마자 실행된다. 이걸 왜 else라고 부르는 걸까? and라고 해야 하지 않을까? if/else 문에서 else는 '이전 블록이 실행되지 않으면 이 블록이 실행된다'는 의미다. try/except 문에서 except도 마찬가지로 '이전 블록에서 실패하면 이 블록이 실행된다'고 정의할 수 있다.

비슷하게 try/except/else의 else도 '이전 블록이 실패하지 않으면 실행하라'는 뜻이므로 이 패턴을 따른다 (**"try/except/else/finally에서 각 블록의 장점을 이용하자"** 참고). try/finally도 '이전 블록을 실행하고 항상 마지막에 실행하라'는 의미이므로 이해하기 쉽다.

파이썬에서 else, except, finally 개념을 처음 접하는 프로그래머들은 for/else의 else 부분이 '루프가 완료되지 않는다면 이 블록을 실행한다'는 의미라고 짐작할 것이다. 실제로는 정확히 반대다. 루프에서 break 문을 사용해야 else 블록을 건너뛸 수 있다.

```
for i in range(3):
    print('Loop %d' % i)
    if i == 1:
        break
    else:
        print('Else block!')

>>>
Loop 0
Loop 1

Loop 0
Loop 1
```

다른 놀랄 만한 점은 빈 시퀀스를 처리하는 루프문에서도 else 블록이 즉시 실행된다는 것이다.

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')

>>>
For Else block!
```

else 블록은 while 루프가 처음부터 거짓인 경우에도 실행된다.

```
while False:
    print('Never runs')
else:
    print('While Else block!')

>>>
While Else block!
```

이렇게 동작하는 이유는 루프 다음에 오는 else 블록은 루프로 뭔가를 검색할 때 유용하기 때문이다. 예를 들어 두 숫자가 서로소(coprime: 공약수가 1밖에 없는 둘 이상의 수)인지를 판별한다고 하자. 이제 가능한 모든 공약수를 구하고 숫자를 테스트해보겠다. 모든 옵션을 시도한 후에 루프가 끝난다. else 블록은 루프가 break를 만나지 않아서 숫자가 서로소일 때 실행된다.

```
a = 4
b = 9
for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a%i == 0 and b%i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')

>>>
Testing 2
Testing 3
Testing 4
Coprime
```

실제로 이런 방식으로 코드를 작성하면 안 된다. 대신에 이런 계산을 하는 헬퍼 함수를 작성하는 게 좋다. 이런 헬퍼 함수는 두 가지 일반적인 스타일로 작성한다.

첫 번째 방법은 찾으려는 조건을 찾았을 때 바로 반환하는 것이다. 루프가 실패로 끝나면 기본 결과(True)를 반환한다.

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

두 번째 방법은 루프에서 찾으려는 대상을 찾았는지 알려주는 결과 변수를 사용하는 것이다. 뭔가를 찾았으면 즉시 break로 루프를 중단한다.

```
def coprime2(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

이 두 가지 방법을 적용하면 낯선 코드를 접하는 개발자들이 코드를 훨씬 쉽게 이해할 수 있다. else 블록을 사용한 표현의 장점이 나중에 여러분 자신을 비롯해 코드를 이해하려는 사람들이 받을 부담감보다는 크지 않다. 루프처럼 간단한 구조는 파이썬에서 따로 설명할 필요가 없어야 한다. 그러므로 루프 다음에 오는 else 블록은 절대로 사용하지 말아야 한다.

핵심 정리

- 파이썬에는 for와 while 루프의 내부 블록 바로 뒤에 else 블록을 사용할 수 있게 하는 특별한 문법이 있다.
- 루프 본문이 break 문을 만나지 않은 경우에만 루프 다음에 오는 else 블록이 실행된다.
- 루프 뒤에 else 블록을 사용하면 직관적이지 않고 혼동하기 쉬우니 사용하지 말아야 한다.

발췌: '파이썬 코딩의 기술

try/except/else/finally에서 각 블록의 장점을 이용하자

파이썬에는 예외 처리 과정에서 동작을 넣을 수 있는 네 번의 구분되는 시점이 있다. try, except, else, finally 블록 기능으로 각 시점을 처리한다. 각 블록은 복합문에서 독자적인 목적이 있으며, 이 블록들을 다양하게 조합하면 유용하다

finally 블록

예외를 전달하고 싶지만, 예외가 발생해도 정리 코드를 실행하고 싶을 때 try/finally를 사용하면 된다. try/finally의 일반적인 사용 예 중 하나는 파일 핸들러를 제대로 종료하는 작업이다.

```
handle = open('/tmp/random_data.txt') # IOError가 일어날 수 있음
try:
    data = handle.read() # UnicodeDecodeError가 일어날 수 있음
finally:
    handle.close()       # try: 이후에 항상 실행됨
```

read 메서드에서 발생한 예외는 항상 호출 코드까지 전달되며, handle의 close 메서드 또한 finally 블록에서 실행되는 것이 보장된다. 파일이 없을 때 일어나는 IOError처럼, 파일을 열 때 일어나는 예외는 finally 블록에서 처리하지 않아야 하므로 try 블록 앞에서 open을 호출해야 한다.

else 블록

코드에서 어떤 예외를 처리하고 어떤 예외를 전달할지를 명확하게 하려면 try/except/else를 사용해야 한다. try 블록이 예외를 일으키지 않으면 else 블록이 실행된다. 'else 블록을 사용하면 try 블록의 코드를 최소로 줄이고 가독성을 높일 수 있다. 예를 들어 문자열에서 JSON 딕셔너리 데이터를 로드하여 그 안에 든 키의 값을 반환한다고 하자.

```
def load_json_key(data, key):
    try:
        result_dict = json.loads(data)    # ValueError가 일어날 수 있음
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]           # KeyError가 일어날 수 있음
```

데이터가 올바른 JSON이 아니라면 json.loads로 디코드할 때 ValueError가 일어난다. 이 예외(exception)는 except 블록에서 발견되어 처리된다. 디코딩이 성공하면 else 블록에서 키를 찾는다. 키를 찾을 때 어떤 예외가 일어나면 그 예외는 try 블록 밖에 있으므로 호출 코드까지 전달된다. else 절은 try/except 다음에 나오는 처리를 시각적으로 except 블록과 구분해준다. 그래서 예외 전달 행위를 명확하게 한다.

모두 함께 사용하기

복합문 하나로 모든 것을 처리하고 싶다면 try/except/else/finally를 사용하면 된다. 예를 들어 파일에서 수행할 작업 설명을 읽고 처리한 후 즉석에서 파일을 업데이트한다고 하자. 여기서 try 블록은 파일을 읽고 처리하는 데 사용한다. except 블록은 try 블록에서 일어난 예외를 처리하는 데 사용한다. else 블록은 파일을 즉석에서 업데이트하고 이와 관련한 예외가 전달되게 하는 데 사용한다. finally 블록은 파일 핸들을 정리하는 데 사용한다.

```
UNDEFINED = object()

def divide_json(path):
    handle = open(path, 'r+')       # IOError가 일어날 수 있음
    try:
        data = handle.read()  # UnicodeDecodeError가 일어날 수 있음
        op = json.loads(data)  # ValueError가 일어날 수 있음
        value = (
            op['numerator'] /
            op['denominator']) # ZeroDivisionError가 일어날 수 있음
    except ZeroDivisionError as e:
        return UNDEFINED
    else:
        op['result'] = value
        result = json.dumps(op)
        handle.seek(0)
        handle.write(result)    # IOError가 일어날 수 있음
        return value
    finally:
        handle.close()          # 항상 실행함
```

이 레이아웃은 모든 블록이 직관적인 방식으로 엮여서 동작하므로 특히 유용하다. 예를 들어 결과 데이터를 재작성하는 동안에 else 블록에서 예외가 일어나도 finally 블록은 여전히 실'행되어 파일 핸들을 닫는다.

핵심 정리

- try/finally 복합문을 이용하면 try 블록에서 예외 발생 여부와 상관없이 정리 코드를 실행할 수 있다.
- else 블록은 try 블록에 있는 코드의 양을 최소로 줄이는 데 도움을 주며 try/except 블록과 성공한 경우에 실행할 코드를 시각적으로 구분해준다.
- else 블록은 try 블록의 코드가 성공적으로 실행된 후 finally 블록에서 공통 정리 코드를 실행하기 전에 추가 작업을 하는 데 사용할 수 있다.

왜 프로그래밍 기반 데이터 분석이 중요한가?

데이터 분석 작업을 하는 사람들이 프로그래밍을 배워야하는 이유는 여럿 있다. 우선 수작업이 불가능한 방대한 규모의 데이터를 처리하고 분석하는 것이 가능해진다. 두 가지 상황을 생각해보자.

첫 번째는 엑셀에서 다루기 어려운 (혹은 열리지 않는) 대용량 파일을 처리하는 상황이다. 설사 파일이 열린다 해도 수작업으로는 정상적인 데이터 처리를 기대하기 어렵다. 데이터양이 많으면 수정하는데 너무 많은 시간이 걸릴 뿐 아니라 작업량이 많아지면서 실재로 수정이 필요한 부분을 놓치게 된다. 즉 데이터 자체에 오류가 발생할 확률이 그만큼 높아지는 것이다.

두 번째는 다량의 파일에 포함된 데이터를 처리하는 상황이다. 때에 따라서는 수 십, 수백 심지어 수천 개의 파일 속에 포함된 데이터를 일일이 처리해야한다. 파일 개수가 늘어날수록 수작업으로 처리하는 것은 불가능해진다. 이 두 가지 상황 모두 프로그래밍으로 해결 할 수 있다. 특히 파이썬 스크립트는 대용량 파일과 다량의 파일을 빠르고 효율적으로 처리 할 수 있다.

데이터 분석 작업을 하는 사람들이 프로그래밍을 배워야하는 또 다른 이유는 작업의 지동화이다. 데이터처리 및 분석은 대부분 반복적이고 많은 시간이 걸리는 작업이다. 흔히 데이터 매니지먼트는 고객이나 공급자로부터 데이터를 수집하고, 보관이 필요한 데이터를 추출하고, 분석을 위해 데이터 형태나 자료형을 변환한 뒤, 데이터베이스 등 저장소에 저장하는 순서로 진행된다(데이터 과학자들은 이러한 과정을 ETL(Extract, Transform, Load이라 부른다). 일반적인 데이터 분석 과정 역시 유사하다. 데이터 수집과 전처리를 진행하고 분석을 실행한 뒤 그 결과를 보고서로 작성한다. 분석의 전 과정을 단계별로 명확하게 정의하면. 필요한 연산을 자동으로 수행하는 파이썬 코드를 작성할 수 있다. 자동화를 위한 스크립트가 완성되면 그동안 반복 작업에 소모된 시간을 다른 더욱 중요한 일에 쓸 수 있다.

또한 데이터 분석 작업이 자동화되면 에러가 발생할 가능성이 작아진다. 수작업으로 데이터를 처리할 때에는 복사/붙여넣기를 잘못하거나 오타가 발생할 확률이 높다. 작업을 급하게 진행하다가 혹은 작업자의 컨디션에 따라 실수가 발생할 수 있기 때문이다. 대용량 파일이나 다량의 파일을 동시에 처리해야하는 상황이라면 반복 작업으로 인한 에러발생 가능성은 더욱 커진다. 반면에 자동화가 되면 정선 산만이나 육체 피로 등의 컨디션 난조에서 자유롭다. 작업자는 스크립트를 디버그하고 실행만 하면 된다. 파이썬 스크립트는 지치지 않고 일관되게 작업을 수행할 것이다.

데이터 분석 작업을 하는 사람들이 프로그래밍을 배워야하는 마지막 이유는 재미와 자신감을 얻을 수 있다는 것이다. 기본적인 프로그래밍 문법을 익히고 데이터 분석이라는 목표를 달성하는 과정에서 재미를 얻을 수 있다. 온라인의 수많은 예시 코드를 자선의 작업에 맞도록 수정하려면 개인의 창의력과 문재 해결 능력이 필요하다. 결국 자신의 작업에 적합한 코드나 구문을 찾아내고 그것이 작동하도록 수정하는 창의적 과정에서 재미가 발생한다. 뿐만 아니라 프로그래밍은 작업자에게 자신감을 심어준다. 위에서 살펴본 것처럼 대용량이나 다량의 파일을 처리하는 상황에서 프로그래밍을 모른다면 많은 시간이 걸리거나 처리가 불가능할 것이다. 하지만 프로그래밍을 할 수 있다면 이러한 문재를 파이썬 스크립트로 쉽고 빠르게 해결할 수 있다. 프로그래밍을 배우기 이전에는 힘들거나 불가능했던 작업이 가능해지면서 자신감을 갖게 될 것이다.