Jaehong-Daniel-Kim adding python_typehints d36b4cc · 3 minutes ago History

Python Type Hints

Python Types

C, C++ 또는 Java의 특성과 Python 언어의 특성 중 한가지 큰 차이점이 있다.

먼저 아래 코드를 확인해보자

1. C++ 언어로 작성된 “Hello World” 출력 프로그램

```cpp
// C++ program to display "Hello World"

#include <iostream>
using namespace std;

int main()
{
        cout << "Hello World";

        return 0;
}
```

1. Java 언어로 작성된 “Hello World” 출력 프로그램

```java
// Java program to display "Hello World"

class HelloWorld {
        public static void main(String args[])
        {
                System.out.println("Hello World");
        }
}
```

1. Python 언어로 작성된 “Hello World” 출력 프로그램

```python
# Python program to display "Hello World"

def main():
        print("Hello World")


if __name__ == "__main__":
        main()
```

위 예제에서는 각 3개의 다른 언어(C++, Java, Python)를 활용하여 “Hello World” 를 화면에 출력하는 프로그램을 작성한 예제이다.

위 세개의 예제를 비교해보면 Python이 다른 언어에 비해서 작성하고, 읽기 쉽다는 특성 외에도 한가지 큰 차이점이 보인다. 그것은 C++와 Java는 데이터의 자료형을 직접적으로 작성해주었다는 점이다. 하지만 Python 코드에는 그 어디에도 자료형과 관련된 내용은 작성되어있지 않다.

Statically Typed Programming Language

이것은 프로그래밍 언어의 특성 차이인데, C++과 Java와 같은 언어의 경우 `Statically Typed Programming Language (정적인 언어)` 라고 하여 코드가 컴파일 되는 `Compile-time` 에서 데이터의 자료형에 대한 검사가 이루어 진다. `Compile-time` 이라고 하면 사람의 언어로 작성된 코드가 컴퓨터가 이해할 수 있는 Machine-language로 변환되는 과정을 뜻하며, 제대로된 변환이 이루어지기 위해서는 코드 자체에 데이터에 대한 정확한 자료형이 명시되어있어야 한다.

또한 데이터의 자료형은 데이터가 들어있는 변수에 묶이므로 해당 변수는 다른 데이터 값을 갖을 수는 있지만 다른 자료형의 데이터로 변환할 수 없다.

이와같은 특성을 갖고있는 언어는 위 두 언어와 함께 Kotlin, Go, Swift등의 언어가 있다.

Dynamically Typed Language

반면 Python의 경우 `Dynamically Typed Language(동적인 언어)` 라고 하며, 코드가 실행되는 `Runtime` 에서 데이터의 자료형에 대한 검사가 이루어 진다. 즉, 프로그램이 실행되는 동안에 데이터의 자료형에 대한 검사가 이루어 지는 것이다.

`Dynamically Typed Language` 의 경우 데이터의 자료형은 변수가 아닌 데이터의 값에 매핑되기 때문에 동일한 변수명에 새로운 자료형의 데이터를 다시 매핑할 수 있다는 특징이 있다.

이와같은 특성을 갖고있는 언어는 Python과 함께 JavaScript, PHP, Ruby, Perl 등의 언어가 있으며, 이러한 특성을 갖고있는 언어는 대부분 자체 interpreter를 통하여 코드를 바로 실행할 수 있는 `Interpreted-Language` 또는 `Scripting-Language` 이다.

Python Type Hint

그렇다면, Python언어를 활용하여 프로그램을 작성할 때 Runtime Error를 피하기 위하여 코드 실행 이전 단계에서 자료형에 대한 검사(Type Check)을 진행할 수 있는 방법이 아얘 없을까?

Python 3.5 버전부터 Python에서 정식으로 `Type Hint` (또는 `Type Annotation` 이라고 부르기도 한다) 기능을 지원하여 코드에 자료형에 대한 정보를 명시해줄 수 있다.

아래 코드를 살펴보자.

```
def count_in_stock(sizes, quantities):
        """Returns the quantities of each size of shirts as values in a dictionary

        :param size: List of sizes of shirts in stock in an order of largest to smallest
        :type size: list[int]
        :param quantities: List of quantities of each size of shirts in stock
        :type quantities: tuple[int, ...]

        :returns: The quantities of each size of shirts as values in a dictionary
        :rtype: dict[str, int]
        """
  result = {}
  for size, quantity in zip(sizes, quantities):
          result.update({size: quantity})
        return result


if __name__ == "__main__":
        shirt_sizes = ["xxl", "xl", "l", "m", "s"]
        quantities = (5, 3, 2, 1, 4)
        in_stock = count_in_stock(shirt_sizes, quantities)
```

위 코드는 큰 사이즈부터 작은 사이즈 순서대로 각 사이즈를 담고있는 리스트, `shirt_sizes` ,와 동일한 순서로 각 사이즈별 남아있는 재고 수량을 담고있는 리스트, `quantities` , 를 인자로 받아 각 사이즈별 남아있는 수량 데이터를 딕셔너리 형태로 반환해주는 함수를 작성한 코드이다.

함수에서는 지난 토픽에서 공부했던 docstring을 활용하여 각 인자 및 반환되는 데이터에 대한 타입을 정리하였으며, 그 외 코드에는 그 어떤 자료형에 대한 정보를 찾아볼 수 없다.

이제 위 코드를 `Type Hint` 를 적용하여 자료형에 대한 정보를 직관적으로 명시해보자.

```
def count_in_stock(
                sizes: list[str],
                quantities: tuple[int, ...]) -> dict[str, int]:
        """Returns the quantities of each size of shirts as values in a dictionary

        :param size: List of sizes of shirts in stock in an order of largest to smallest
        :param quantities: List of quantities of each size of shirts in stock

        :returns: The quantities of each size of shirts as values in a dictionary
        """

        result: dict = {}
        for size, quantity in zip(sizes, quantities):
                result.update()
        return result


if __name__ == "__main__":
shirt_sizes = ["xxl", "xl", "l", "m", "s"]
quantities = (5, 3, 2, 1, 4)
in_stock: dict[str, int] = count_in_stock(shirt_sizes, quantities)
```

위에 작성된 내용을 조금 더 자세히 살펴보자.

```python
def count_in_stock(
                sizes: list[str],
                quantities: tuple[int, ...]) -> dict[str, int]:
        # sizes: list[str]
        # quantities: tuple[int, ...]
        #   - 위 두 구문은 함수의 인수, 파라미터에 대한 자료형을 명시한다.
        #   - sizes는 문자열(str)을 원소로 포함한 리스트(list) 자료형만을 파라미터로 받는다.
        #   - quantities는 숫자형(int)을 원소로 포함한 튜플(tuple) 자료형만을 파라미터로 받는다.
        #   - 원소의 개수는 무관하지만 quantities의 경우 자료형이 섞이면 안된다.
        #      - shirt_sizes = []  (O)
        #      - shirt_sizes = ["a"]  (O)
        #      - shirt_sizes = ["a", "b", "c", "d", "e", "f"]  (O)
        #      - quantities = ("a", 1)  (X)

        # -> dict[str, int]
        #   - 해당 구문은 함수로 부터 반환(return)되는 데이터의 자료형을 명시한다.
        #   - 문자열 키(key)와 숫자형 값(value)을 갖는 사전형(dictionary) 자료형을 반환 한다.
        """Returns the quantities of each size of shirts as values in a dictionary

        :param size: List of sizes of shirts in stock in an order of largest to smallest
        :param quantities: List of quantities of each size of shirts in stock

        :returns: The quantities of each size of shirts as values in a dictionary
        """

        result = {}
        for size, quantity in zip(sizes, quantities):
                result.update()
        return result


if __name__ == "__main__":
shirt_sizes = ["xxl", "xl", "l", "m", "s"]
quantities = (5, 3, 2, 1, 4)
in_stock: dict[str, int] = count_in_stock(shirt_sizes, quantities)
```

- 모든 파라미터 또는 변수명뒤에 `:<Type>` 과 같은 형태로 `type hint` 를 작성할 수 있다.
- 함수의 반환 타입 (return type)은 함수의 가장 끝, 콜론(colon) 이전에 `→ <type>` 과 같은 형태로 type hint를 작성할 수 있다.
- … 은 `ellipsis` 라고 부르며 type hint에서는 `tuple[int, …]` 와 같은 사용법으로 사용된다.
  - 여기에서 … 의 의미는 `연속` 이라는 의미를 갖으며 `tuple 자료형의 원소로는 연속되는 int만 포함되어있다` 라는 의미가 된다.
  - 해당 튜플의 원소인 int의 개수가 0개이던, 1개이던 10개이던 100개이던 무관하지만 원소로 int 이외의 자료형은 존재하지 않는다(존재 해서는 안된다)는 의미를 갖는다.

💡 list[str, ...]과 같이 표기하지 않은건 프로그래밍 문화속에서 list는 단일 자료형의 원소만 포함하는 동종 자료형(homogeneous data type) 이기 때문이다. 반면, tuple의 경우 다종 자료형(heterogeneous data type)으로 여러 가지 자료형을 함께 저장할 수 있다.

물론 Python 언어에서 list와 tuple 모두 str, int, float 등 여러 자료형을 섞어서 저장할 수 있기때문에 위 문화를 무조건 적으로 따를 필요는 없다. 하지만 다른 언어에서는 list자료형에 다양한 자료형을 섞어 저장하는 것을 허용하지 않는 경우가 많다. 이러한 문화와 특성이 있다는 것은 기억하자.

💡 Python ≤ 3.8 버전에서는 list, tuple, dict와 같이 기본 자료형 타입 키워드로는 `list[<타입>]`과 같이 작성할 수 없으며 아래와 같이 별도의 타이핑 모듈을 import 하여 아래와 같이 작성해주어야 한다.

```python
#! python <= 3.8
from typing import List, Tuple, Dict

foo = List[str]   # O
foo = list[str]   # X

foo = Tuple[int, int, int]   # O
foo = tuple[int, int, int]   # X

foo = Dict[str, int]   # O
foo = dict[str, int]   # X
```

더 다양한 Type Hint 사용법

위에서 설명한 기초적인 type hint외로도 다양한 type hint 사용법이 존재한다.

1. 다양한 자료형

   만약 함수의 파라미터 또는 함수가 다양한 자료형을 받거나, 반환한다면 아래와같이 `Union` 활용할 수 있다.

   `Union[<Type1>, <Type2>, …]` 과 같이 사용한다.

   ```python
   from typing import Union

   def add_numbers(a: Union[str, int], b: Union[str, int) -> int:
           return int(a) + int(b)


   def only_print(var: Union[str, int, dict, list, tuple]) -> None:
           print(var)


   def converter(param: Union[str, int]) -> Union[str, int]:
           if isinstance(param, str):
                   return len(param)   # Here returns integer
           else:
             return str(param)   # Here returns string
   ```

   `Union` 은 명시한 자료형 중 아무거나 받을 수 있다는 것을 보여준다. 하지만 `Union` 내에 포함되어 있지 않은 자료형은 받을 수 없다.

   만약 함수에서 반환하는 값이 없다면 `None` 을 반환한다.

2. 어떠한 자료형이든 상관 없을 때

   `Any` 를 활용하면 명시적으로 어떠한 자료형이든 상관이 없다는 것을 보여준다.

   ```python
   from typing import Any

   def anything(param: Any) -> Any:
           pass
   ```

   어떠한 타입도 허용한다. 하지만 그만큼 허용하는 자료형의 범위가 넓기때문에 가능한 지양하는 것이 좋다.

3. 함수의 type hint

   callback 함수와 같이 함수에 다른 함수를 인자로 받아서 실행하고자 하는 경우가 있다. 이럴 때 함수를 받아오는 파라미터는 `Callable` 을 활용하여 type hint를 작성할 수 있다.

   `Callable[[<Arg1_Type>, <Arg2_Type>], <ReturnType>]` 과 같이 작성할 수 있다.

```
from typing import Callable

def add_all(values: list[int], *args: int) -> int:
        _sum = sum(values)
        additional_sum = sum(args)
        return _sum + additional_sum

def calculator(
                callback: Callable[[list[int], int], int],
                values: list[int],
                *args: int) -> int:
        return callback(values, *args)


if __name__ == "__main__":
        num_list = [1,2,3,4,5]
        summed_result = calculator(add_all, num_list)
```

Type Check

Python은 `Dynamically Typed Language` 이기 때문에 `Runtime` 에서 검사가 이루어 진다고 했다. 이러한 특성은 `scripting language` 의 이점으로 별도의 compile 과정 없이 일회성 스크립트 또는 소규모 어플리케이션을 빠르게 개발하고 실행할 수 있다.

다만, 개발한 프로그램의 규모가 커지거나 항상 백그라운드에서 실행중인 서버형 어플리케이션을 개발할 때에 발생하는 `Runtime Error` 는 치명적이며 어플리케이션의 안정성을 해치게된다. 따라서 규모가 큰 어플리케이션을 개발할 때에는 정적(static)으로 프로그램 실행 이전 단계에서 타입 검사를 먼저 진행하여 `Runtime` 에서 발생하는 `TypeError` 를 예방할 수 있다.

mypy 모듈

아쉽게도 지금까지 공부했던 type hint를 코드 곳곳에 꼼꼼히 적어놓아도 Python은 자체적으로는 타입 검사를 진행하지 않는다. PyCharm과 같은 IDE에서는 type hint의 자료형과 어긋나는 자료형을 입력하였을 때 warning으로 알려주는 기능이 있지만, “hint”를 제공해줄 뿐이며 프로그램 실행에 직접적인 영향을 끼치지는 않는다.

mypy모듈은 프로그램 실행전에 미리 작성해둔 type hint를 기반으로 코드를 읽고 검사하여 발생할 수 있는 `TypeError` 보여준다. 아래 예시를 통해 직접 확인해보자.

mypy 설치

```
# Install mypy
pip3 install mypy

# Check version
mypy --version

# if terminal displays "commands not found" error
# after successfully installing the module, try restarting your terminal.
```

mypy를 통한 소스코드 검사

테스트를 위하여 test.py 작성

```
# test.py

my_var: int = "Hello World"
print(my_var)
```

```
# execute test.py
python3 test.py

> Hello World
```

`my_var` 변수의 타입을 `int` 로 지정하였지만 실제 데이터는 `str` 이다.

하지만 앞서 설명하였듯 Python은 언어 자체적으로는 타입 검사를 하지 않는다. 또한 해당 변수의 데이터를 출력하는 `print` 함수는 자료형에 관계 없이 모든 자료형을 출력하므로 별도의 오류는 발생하지 않는다.

위 test.py 를 mypy를 활용하여 검사를 진행해보자.

```
# check test.py using mypy
mypy test.py

test.py:1: error: Incompatible types in assignment (expression has type "str", \
variable has type "int") [assignment] \
Found 1 error in 1 file (checked 1 source file)
```

mypy 모듈을 활용하여 test.py 스크립트를 검사하면 위와 같이 오류를 출력하며, 변수는 “int” 자료형을 갖고있지만, 실제 데이터는 “str” 자료형을 갖고있다는 오류 메시지를 보여준다.

Runtime TypeError

위에서 한번 봤던 코드를 아래와 같이 수정하여 calc.py 파일로 저장하였다.

```
# calc.py
from typing import Callable

def add_all(values: list[int], *args: int) -> int:
        _sum = sum(values)
        additional_sum = sum(args)
        return _sum + additional_sum

def calculator(
                callback: Callable[[list[int], int], int],
                zvalues: list[int],
                *args: int) -> int:
        return callback(values, *args)


if __name__ == "__main__":
        user_values = input("Enter Values to be Added: ")
        print(f"You Entered: {user_values}")
        summed_result = calculator(add_all, user_values)
```

코드를 실행해보자.

```
python3 calc.py

> Enter Values to be Added:
```

프로그램을 실행하면 위와같이 “Enter Values to be Added” 라는 구문이 출력되고 사용자의 입력을 기다린다. 즉, 프로그램은 정상적으로 실행되어 입력값을 받아오기를 기다리고 있는 상태이다.

이제 값을 넣어보자.

```
python3 calc.py

> Enter Values to be Added: 1 2 3 4 5

Traceback (most recent call last):
   File "calc.py", line 19, in <module>
     summed_result = calculator(add_all, user_values)
   File "calc.py", line 13 in calculator
     return callback(values, *args)
   File "calc.py", line 5, in add_all
     _sum = sum(values)
TypeError: unsupported operand type(s) for +: "int" and "str"
```

값을 입력하자 위와같이 `TypeError` 가 발생하게된다.

오류가 발생한 이유는 `input()` 함수로 받아온 값은 `str` 형태로 저장되지만, `str` 자료형은 `sum()` 함수로 더할 수 없기 때문이다.

이렇게 `Runtime` 도중에 발생하는 `TypeError` 는 어플리케이션의 안정성에 치명적일 수 있다.

하지만 어플리케이션을 실행하기 전 mypy를 통하여 type 검사를 미리 진행한다면 어플리케이션을 deploy 하기 이전에 미리 발견하여 수정을 할 수 있다.

```
mypy calc.py

calc.py:19: error: Argument 2 to "calculator" has incompatible type "str"; \
expected "list[int]" [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

ignore 활용하기

간혹 개발 혹은 테스트 단계에서 특정 구문에서 발생하는 TypeError를 의도적으로 무시하고 싶은 상황이 있을 수 있다. 이런 상황에서는 `ignore` 를 활용해보자. `ignore` 키워드를 사용하게되면 mypy 모듈에서 해당 구문에서 발생하는 오류는 출력하지 않고 그냥 넘어간다.

```
# calc.py
from typing import Callable

def add_all(values: list[int], *args: int) -> int:
        _sum = sum(values)
        additional_sum = sum(args)
        return _sum + additional_sum

def calculator(
                callback: Callable[[list[int], int], int],
                values: list[int],
                *args: int) -> int:
        return callback(values, *args)


if __name__ == "__main__":
        user_values = input("Enter Values to be Added: ")
        print(f"You Entered: {user_values}")
        summed_result = calculator(add_all, user_values)  # type: ignroe
```

```
mypy calc.py

Success: no issues found in 1 source file
```

`ignore` 키워드는 mypy 모듈을 통한 검사 과정에서만 적용되며, 해당 구문에서 발생하는 오류를 무시하는 기능을 한다. 실제 소스코드 실행에서는 영향을 주지 않으며, 위 코드를 실행하게 되면 `Runtime` 에서 동일한 `TypeError` 가 발생할것이므로 주의하자.

Conclusion

Python의 `Type Hint` (또는 `Type Annotation`)에 대한 활용법을 알아보았다.

일회성 스크립트 또는 작은 규모의 혼자서만 개발하는 프로그램의 경우 부수적인 일거리가 늘어나는 느낌일 수도 있고, 비교적 자료형(data type)에 대한 자유도가 높은 Python에서까지 이런걸 굳이 신경 써야하나 싶을 수도 있다. 또한 주석처리나 Type Annotation에 대한 내용을 깊이있게 잘 안다고 해도 실제 어플리케이션의 기능을 개발하거나 코딩 관련 자격증을 취득하는 데에는 아무런 도움이 되지 않을것이다.

하지만 Programming Language 라는 단어에서 알 수 있듯 개발 언어 또한 하나의 언어이며 소통 수단이다. 언제나 혼자 개발한다고 장담하기 어려우며, 언젠가 소스코드의 담당자가 바뀔수도 있고, 내가 작성한 어플리케이션의 기능이 너무 유용하고 좋아서 점점 새로운 기능들이 확장되어 어플리케이션의 규모가 커질 수도 있다. 어플리케이션의 규모가 커지면 당연히 혼자서 모든 기능을 개발할 수 없고, 많은 개발자들과 협업하게 될것이다. 그 때 잘 작성된 `Comment(주석)`와 `Type Annotation` 은 코드의 가독성과 확장성을 높여줄 뿐만 아니라 완성된 어플리케이션의 안정성또한 보완해준다.

따라서 작은 어플리케이션을 개발할 때에도 주석을 달고 type hint를 작성하며 조금씩 익숙해지자.