Typing made easy

How to make mypy happy

Hana Burtin
Backend dev @ octopus energy

HanaPoulpe

Sometime you have something that works:

```
def main() -> None:
    numbers = first pi decimal()
    print(dict(count_individual_values(numbers)))
```

Sometime you have something that works:

```
def main() -> None:
    numbers = first pi decimal()
    print(dict(count_individual_values(numbers)))
```

But then this happens

demo.py:20: error: Argument 1 to "count_individual_values" has incompatible type "Generator[int, None, None]"; expected "List[int]"

Sometime you have something that works:

```
def main() -> None:
    numbers = first pi decimal()
    print(dict(count_individual_values(numbers)))
```

But then this happens

demo.py:20: error: Argument 1 to "count_individual_values" has incompatible type "Generator[int, None, None]"; expected "List[int]"

You’re in a hurry and use this solution:

```
def main() -> None:
    numbers = first pi decimal()
    print(dict(count_individual_values(list(numbers))))
```

But what happened?

```
def count individual values(values: list[int]) -> dict[int, int]:
    individual_values: dict[int, int] = defaultdict(int)

    for v in values:
        individual_values[v] += 1

    return individual_values
```

But what happened?

```
def count individual values(values: list[int]) -> dict[int, int]:
    individual_values: dict[int, int] = defaultdict(int)

    for v in values:
        individual_values[v] += 1

    return individual_values
```

How to fix it?

```
def count_individual_values(values: Iterable[int]) -> dict[int, int]:
```

My rule of thumb:

1. **Function parameter should request the minimal implementation**
2. **Function returns should be as detailed as possible**

ABC	Inherits from	Abstract Methods	Mixin Methods
Container [1]		`__contains__`	
Hashable [1]		`__hash__`	
Iterable [1] [2]		`__iter__`	
Iterator [1]	Iterable	`__next__`	`__iter__`
Reversible [1]	Iterable	`__reversed__`	
Generator [1]	Iterator	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized [1]		`__len__`	
Callable [1]		`__call__`	
Collection [1]	Sized, Iterable, Container	`__contains__`, `__iter__`, `__len__`	
Sequence	Reversible, Collection	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	Sequence	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited Sequence methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
ByteString	Sequence	`__getitem__`, `__len__`	Inherited Sequence methods

Other cool tools:

```
class Parser(Protocol):
    def parser(self, input: str) -> Any:
        ...
```

Python 3.12

PEP 695: Type Parameter Syntax

Generic classes and functions under **PEP 484** were declared using a verbose syntax that left the scope of type parameters unclear and required explicit declarations of variance.

PEP 695 introduces a new, more compact and explicit way to create generic classes and functions:

```
def max[T](args: Iterable[T]) -> T:
    ...

class list[T]:
    def __getitem__(self, index: int, /) -> T:
        ...

    def append(self, element: T) -> None:
        ...
```

In addition, the PEP introduces a new way to declare type aliases using the `type` statement, which creates an instance of `TypeAliasType`: