

Python Safe & Sound



M. Boerger
2025.02.27

In Python anything can be anything

- Let's talk about this - and stick to Python 3.10+
- How to prevent the mess
- Tools



Let's do some type checking

Do we have data?

- Works for almost anything

```
if foo:  
    bar()
```

Let's do some type checking

Do we have data?

- Works for almost anything
- But are "", 0, False, [], {}, None all the same?

```
if foo:
```

```
    bar()
```

```
print(foo == None)
```

```
print(len(foo))
```

```
# TypeError: object of type 'int' has no len()
```

Let's do some type checking

Do we have data?

- Works for almost anything
- But are "", 0, False, [], {}, None all the same?
 - None likely means nothing
 - All other values above could mean something else!
- The **is** operator

```
if foo:
```

```
    bar()
```

```
print(foo == None)
```

```
print(len(foo))
```

```
# TypeError: object of type 'int' has no len()
```

```
print(foo is None)
```

```
# SyntaxWarning: "is" with a literal...
```

Let's do some type checking

Do we have data?

- Works for almost anything
- But are "", 0, False, [], {}, None all the same?
 - None likely means nothing
 - All other values above could mean something else!
- The **is** operator
- The **type** function

```
if foo:
```

```
    bar()
```

```
print(foo == None)
```

```
print(len(foo))
```

```
# TypeError: object of type 'int' has no len()
```

```
print(foo is None)
```

```
# SyntaxWarning: "is" with a literal...
```

```
print(type(V) is list and len(V) or 42) # BAD!
```

Let's do some type checking

Do we have data?

- Works for almost anything
- But are `""`, `0`, `False`, `[]`, `{}`, `None` all the same?
 - `None` likely means nothing
 - All other values above could mean something else!
- The `is` operator
- The `type` function
- The `isinstance` function

```
if foo:
```

```
    bar()
```

```
print(foo == None)
```

```
print(len(foo))
```

```
# TypeError: object of type 'int' has no len()
```

```
print(foo is None)
```

```
# SyntaxWarning: "is" with a literal...
```

```
print(len(V) if (type(V) is list) or 42)
```

```
print(len(V) if isinstance(V, list) else 42)
```

Some hours later and we have functions

Functions have parameters and return values.

People, including our code, do not often RTFM/D/C.

People like to change what their functions do.

Python likes to do its best

- which can easily be **wrong** after type changes
- which **at best** leads to crashes!

```
def Foo(input):  
    """Do Foo with input."""  
    pass
```

```
def Count(input):  
    """Count values in input."""  
    return len(input)
```

```
def CountUnique(i):  
    """Count unique values in i."""  
    return {v: len([v==w for w in i]) for v in i}
```

Some basic types

bool

bytes

float

int

object

str

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

<code>bool</code>	<code>bytes</code>	<code>list</code>	<code>list[T]</code>
<code>float</code>	<code>int</code>	<code>set</code>	<code>set[T]</code>
<code>object</code>	<code>str</code>	<code>dict</code>	<code>dict[K, V]</code>
		<code>tuple</code>	<code>tuple[T1, ...TN]</code>
		<code>tuple[T, ...]</code>	

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]
float	int	set	set[T]
object	str	dict	dict[K, V]
		tuple	tuple[T1, ...TN]
		tuple[T, ...]	

Type | Other

Type | None

Every defined class is a typehint

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]
float	int	set	set[T]
object	str	dict	dict[K, V]
		tuple	tuple[T1, ...TN]
		tuple[T, ...]	

Type | Other

Type | None

Every defined class is a typehint

type[C] *the type of C*

... *means any number*

type: ignore

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]	from typing
float	int	set	set[T]	
object	str	dict	dict[K, V]	Any
		tuple	tuple[T1, ...TN]	
		tuple[T, ...]		

Type | Other

Type | None

Every defined class is a typehint

type[C] *the type of C*

... *means any number*

type: ignore

Avoid ANY
whenever possible.

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]	from typing	
float	int	set	set[T]		
object	str	dict	dict[K, V]	Any	
		tuple	tuple[T1, ...TN]		
		tuple[T, ...]			
Type Other				Union[U, V]	== U V
Type None				Optional[U]	== U None

Every defined class is a typehint

type[C] *the type of C*

... *means any number*

type: ignore

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]
float	int	set	set[T]
object	str	dict	dict[K, V]
		tuple	tuple[T1, ...TN]
		tuple[T, ...]	

Type | Other

Type | None

Every defined class is a typehint

type[C] *the type of C*

... *means any number*

type: ignore

from typing OR collections.abc

Apparently we are supposed to use `collections.abc`.

Callable[[Arg, ...ArgN],Result]

Callable[..., Result]

Iterable[T]

Sequence[T] == tuple[T, ...]

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Some basic types

bool	bytes	list	list[T]	from collections.abc
float	int	set	set[T]	
object	str	dict	dict[K, V]	
		tuple	tuple[T1, ...TN]	
		tuple[T, ...]		

Type | Other

Type | None

Every defined class is a typehint

type[C] *the type of C*

... *means any number*

type: ignore

Callable

Callable[...]

Iterable[T]

Sequence[T] == tuple[T, ...]

Iterable has `__iter__`
and is **uncountable**.

<https://docs.python.org/3/library/collections.abc.html>

Some basic types

bool	bytes	list	list[T]	from collections.abc
float	int	set	set[T]	
object	str	dict	dict[K, V]	
		tuple	tuple[T1, ...TN]	
		tuple[T, ...]		

Type | Other

Type | None

Every defined class is a typehint

`type[C]` *the type of C*

Sequence is a
Collection, Reversible,
has `__getitem__`
and is countable.

s any number

`Callable[[Arg, ...ArgN],Result]`

`Callable[..., Result]`

`Iterable[T]`

`Sequence[T] == tuple[T, ...]`

<https://docs.python.org/3/library/collections.abc.html>

Some basic types

Reversible has
`__reversed__`.

Collection is a
Container, Iterable,
Sizeable
and is countable.

Sequence is a
Collection, Reversible,
has `__getitem__`
and is countable.

from collections.abc

Type | Other
Type | None

Every defined class is a typehint

`type[C]` the type of C

s any number

`Callable[[Arg, ...ArgN],Result]`

`Callable[..., Result]`

`Iterable[T]`

`Sequence[T] == tuple[T, ...]`

<https://docs.python.org/3/library/collections.abc.html>

Some basic types

Reversible has
`__reversed__`.

Collection is a
Container, Iterable,
Sizeable
and is **countable**.

Container has
`__contains__`.

Iterable has `__iter__`
and is **uncountable**.

Sequence is a
Collection, Reversible,
has `__getitem__`
and is **countable**.

Sizeable has `__len__`
and is **countable**.

<https://docs.python.org/3/library/collections.abc.html>

Some basic types

bool	bytes	list	list[T]	from collections.abc
float	int	set	set[T]	
object	str	dict	dict[K, V]	
		tuple	tuple[T1, ...TN]	
		tuple[T, ...]		

Type | Other

Type | None

Every defined class is a typehint

`type[C]` *the type of C*

Sequence is a Collection, Reversible, has `__getitem__` and **is countable**.

... any number

Callable
Callable[...]
Iterable[T]
Sequence[T] == tuple[T, ...]

Iterable has `__iter__` and **is uncountable**.

<https://docs.python.org/3/library/collections.abc.html>

Types for functions

Functions have parameters and return values.

People, including our code, do not often RTFM/D/C.

People like to change what their functions do.

Python likes to do its best

- which can easily be **wrong** after type changes
- which **at best** leads to crashes!

Return values can be type annotated.

```
def Foo(input) -> None:  
    """Do Foo with input."""  
    pass
```

```
def Count(i) -> int:  
    """Count values in `i`."""  
    return len(i)
```

```
def CountUnique(i) -> dict:  
    """Count unique values in `i`."""  
    return {v: len([v==w for w in i]) for v in i}
```

Types for functions

Functions have parameters and return values.

People, including our code, do not often RTFM/D/C.

People like to change what their functions do.

Python likes to do its best

- which can easily be **wrong** after type changes
- which **at best** leads to crashes!

Return values can be type annotated.

Some types allow to do so more or less detailed.

```
def Foo(input) -> None:
    """Do Foo with input."""
    pass
```

```
def Count(i) -> int:
    """Count values in `i`."""
    return len(i)
```

```
def CountUnique(i) -> dict[int, int]:
    """Count unique values in `i`."""
    return {v: len([v==w for w in i]) for v in i}
```

Types for functions

Functions have parameters and return values.

People, including our code, do not often RTFM/D/C.

People like to change what their functions do.

Python likes to do its best

- which can easily be **wrong** after type changes
- which **at best** leads to crashes!

Return values can be type annotated.

Some types allow to do so more or less detailed.

You can still easily state one of some types.

```
def Foo(input) -> None:
    """Do Foo with input."""
    pass
```

```
def CountList(i) -> int | None:
    """Count values in `i`."""
    return len(i) if isinstance(i, list) else None
```

```
def CountUnique(i) -> dict[int, int]:
    """Count unique values in `i`."""
    return {v: len([v==w for w in i]) for v in i}
```

Types for functions

Functions have parameters and return values.

People, including our code, do not often RTFM/D/C.

People like to change what their functions do.

Python likes to do its best

- which can easily be **wrong** after type changes
- which **at best** leads to crashes!

Return values can be type annotated.

Some types allow to do so more or less detailed.

You can still easily state one of some types.

More importantly parameters can have typehints.

```
def Foo(input: Any) -> None:  
    """Do Foo with input."""  
    pass
```

```
def CountList(i: list) -> int | None:  
    """Count values in input."""  
    return len(i) if isinstance(i, list) else None
```

```
def CountUnique(i: Sequence) -> dict[int, int]:  
    """Count unique values in `i`."""  
    return {v: len([v==w for w in i]) for v in i}
```

But this does not work!

The following **should fail** if called with anything but an int!

```
def Count(input: int):  
    return len(input)
```

But this does not work!

The following **should fail** if called with anything but an int!

But it works just fine, since **Python ignores the type hints**.

```
def Count(input: int):  
    return len(input)
```

```
>>> print(Count({}))
```

```
0
```

```
>>> print(Count([]))
```

```
0
```

```
>>> print(Count("foo"))
```

```
3
```

```
>>> print(Count(["foo"]))
```

```
1
```

But this does not work!

The following should fail if called with anything but an int!

But it works just fine, since Python ignores the type hints.

For starters we would need to annotate the return type.

Even `-> None` will do just fine - if that is correct.

```
def Count(input: int) -> int | None:  
    return len(input)
```

```
>>> print(Count({}))
```

```
0
```

```
>>> print(Count([]))
```

```
0
```

```
>>> print(Count("foo"))
```

```
3
```

```
>>> print(Count(["foo"]))
```

```
1
```

But this does not work!

The following should fail if called with anything but an int!

But it works just fine, since Python ignores the type hints.

For starters we would need to annotate the return type.

Even `-> None` will do just fine - if that is correct.

```
def Count(input: int): -> int | None:  
    return len(input)
```

```
>>> print(Count({}))
```

```
0
```

```
>>> print(Count([]))
```

```
0
```

```
>>> print(Count("foo"))
```

```
3
```

```
>>> print(Count(["foo"]))
```

```
1
```

But this does not work!

The following should fail if called with anything but an int!

But it works just fine, since Python ignores the type hints.

For starters we would need to annotate the return type.

Even `-> None` will do just fine - if that is correct.

But more importantly we need tools! We need MyPy.

<https://www.mypy-lang.org/>

Apparently a new language :-)

```
def CountList(i) -> int | None:  
    return len(i) if isinstance(i, list) else None
```

```
>>> print(CountList({}))
```

```
None
```

```
>>> print(CountList([]))
```

```
0
```

```
>>> print(CountList("foo"))
```

```
None
```

```
>>> print(CountList(["foo"]))
```

```
1
```

But this does not work!

The following should fail if called with anything but an int!

But it works just fine, since Python ignores the type hints.

For starters we would need to annotate the return type.

Even `-> None` will do just fine - if that is correct.

But more importantly we need tools! We need MyPy.

<https://www.mypy-lang.org/>

Apparently a new language :-)

```
def CountList(i: Sequence) -> int | None:  
    return len(i) if isinstance(i, list) else None
```

```
>>> print(CountList({}))  
error: Argument 1 to "CountList" has  
incompatible type "dict[Never, Never]";  
expected "Sequence[Any]" [arg-type]  
>>> print(CountList([]))  
0  
>>> print(CountList("foo"))  
None  
>>> print(CountList(["foo"]))  
1
```

MyPy

Install

Use

```
python3 -m pip install mypy
```

```
mypy program.py
```

MyPy

Install

Use

```
python3 -m pip install mypy
```

```
mypy program.py
```

MyPy is a useless product. Python is a dynamic language. You are wasting yours, and, something I will never forgive you, my time. Please remove this [...]. Get a life.

MyPy

Install

```
python3 -m pip install mypy
```

Use

```
mypy program.py
```

For us it flagged a lot of dynamic typing bugs before stuff went to production so it was cost effective.
(usecase: enterprise software).

Nothing is ever easy

Install

```
python3 -m pip install mypy
```

Use

```
mypy program.py
```

Nothing is checked unless there are type annotations.

Functions **must** have a return type annotation.

Flags:

- Disallow untyped
- Strict
- Things get complex -> use a config file.

https://mypy.readthedocs.io/en/stable/config_file.html

```
mypy --disallow-untyped-defs
```

```
mypy --strict
```

```
mypy --config-file <ini-style-file>
```

All code should be annotated

An example...

```
def Read(
    files,
    root = None):
    text = None
    for f in files:
        path = root + f
        with open(path, 'r') as f:
            text = text + f.read()
    return text
```

All code should be annotated

An example... let's add some types.

```
def Read(  
    files: list[str],  
    root: str | None = None) -> str:  
    text: str | None = None  
    for f in files:  
        path: str = root + f  
        with open(path, 'r') as f:  
            text = text + f.read()  
    return text
```

All code should be annotated

```
$> mypy test.py
```

```
test.py:6: error: Unsupported left operand type for +  
("None") [operator]
```

```
test.py:6: note: Left operand is of type "str | None"
```

```
test.py:7: error: Incompatible types in assignment  
(expression has type "TextIOWrapper", variable has type  
"str") [assignment]
```

```
test.py:8: error: "str" has no attribute "read" [attr-defined]
```

```
test.py:9: error: Incompatible return value type (got "str |  
None", expected "str") [return-value]
```

```
Found 4 errors in 1 file (checked 1 source file)
```

```
def Read(  
    files: list[str],  
    root: str | None = None) -> str:  
    text: str | None = None  
    for f in files:  
        path: str = root + f  
        with open(path, 'r') as f:  
            text = text + f.read()  
    return text
```

All code should be annotated

An example... let's add some types.

Run mypy

Address errors!

- 1) If we concat root dir and filename, then we likely want a '/' separator.
- 2) The variable `f` was already defined as a string. It cannot suddenly become a TextIOWrapper. So we use a different name to clearly differentiate.
- 3) The function may return None, so we annotate that.

```
def Read(
    files: list[str],
    root: str | None = None) -> str | None:
    text: str | None = None
    for f in files:
        path: str = (root + '/' if root else '') + f
        with open(path, 'r') as file:
            text = text + file.read()
    return text
```

Nothing is really ever easy

Typehints work nicely for everything that is already defined.

```
class Foo:  
    pass
```

```
class Bar:  
    def Foos(self, foo: Foo) -> None:  
        pass
```

Nothing is really ever easy

Typehints work nicely for everything that is already defined.

Sometimes we need forward declares.

```
class Foo:  
    def Bars(self, bar: Any) -> None:  
        pass
```

```
class Bar:  
    def Foos(self, foo: Foo) -> None:  
        pass
```

Nothing is really ever easy

Typehints work nicely for everything that is already defined.

Sometimes we need forward declares.

- Do not use ANY
- Two options, futures and string-literals

```
class Foo:  
    def Bars(self, bar: "Bar") -> None:  
        pass
```

```
class Bar:  
    def Foos(self, foo: Foo) -> None:  
        pass
```

Nothing is really ever easy

Typehints work nicely for everything that is already defined.

Sometimes we need forward declares.

- Do not use ANY
- Two options, futures and string-literals

```
from __future__ import annotations
```

```
class Foo:  
    def Bars(self, bar: Bar) -> None:  
        pass
```

```
class Bar:  
    def Foos(self, foo: Foo) -> None:  
        pass
```

Nothing is really ever easy

Init methods should be annotated.

- They have a shortcut:
The None return type can be omitted, if at least one param is annotated.
- But that is inconsistent, so avoid.

```
from __future__ import annotations
```

```
class Foo:  
    def __init__(self) -> None:  
        pass  
  
    def Bars(self, bar: Bar):  
        pass  
  
class Bar:  
    def __init__(self, foo: Foo):  
        pass  
  
    def Foos(self, foo: Foo):  
        pass
```

Finally clear class variable

Typehints allow to clearly state a class var is a class var.

The type for the Class var can be omitted resulting in Any.

```
from typing import ClassVar
```

```
class Foo:  
    member: ClassVar[int] = 42
```

Finally clear class variable

Typehints allow to clearly state a class var is a class var.

The type for the Class var can be omitted resulting in Any.

A callable member can be a ClassVar.

```
from typing import Callable, ClassVar
```

```
class Foo:
```

```
    member: ClassVar[int] = 42
```

```
    call: ClassVar[Callable[[Foo, int], None]]
```

Circularity

Sometimes code is separated into multiple files.

And sometimes that creates circular dependencies.

Using `typing.TYPE_CHECKING` allows to break those.

bar.py

```
from foo import BarList
```

```
class Bar:  
    def BarList(self) -> 'list[Bar]':  
        return BarList(self)
```

util.py

```
from typing import TYPE_CHECKING
```

```
if TYPE_CHECKING:  
    import bar
```

```
def BarList(arg: 'bar.Bar') -> 'list[bar.Bar]':  
    return [arg]
```

Type Stubs

Some libraries and generated code have type stubs.

Their annotations may not be available at runtime.

TYPE_CHECKING + future annotations to the rescue.

```
from __future__ import annotations
from typing import TYPE_CHECKING
```

```
if TYPE_CHECKING:
    from _typeshed import SupportsRead
```

```
def Read(data: SupportsRead, count) -> Any:
    return data.read(count)
```

Type Stubs

TYPE_CHECKING sometimes helps you pick right.

```
from __future__ import annotations
from typing import TYPE_CHECKING
```

```
if TYPE_CHECKING:
```

```
    from grpc_health.v1 import health_pb
```

```
else:
```

```
    from health_grpc_py_proto_pb.src.proto.grpc.health.v1 import health_pb
```

```
def CheckHealth(health: health_pb) -> bool
```

```
    return True
```

Type Stubs

Type Stubs allow to explain complex type rules.

The can handle generics, methods and much much more.

They are very useful for SWIG and other forms of binding.

first.py

```
def _marker = object()

def first(iterable, default=_marker):
    try:
        return next(iter(iterable))
    except StopIteration as e:
        if default is _marker:
            raise ValueError('empty value or no default'
                               ) from e
        return default
```

first.pyi

```
from typing import Iterable
from typing_extensions import TypeVar, overload

_T = TypeVar('_T')
_U = TypeVar('_U')

@overload
def first(iterable: Iterable[_T]) -> _T: ...

@overload
def first(iterable: Iterable[_T], default: _U) -> _T|_U: ...
```

Type Stubs

One of the most useful things is to keep runtime simple.

While at the same time being very precise when checking!

The example on the right demonstrates `OpenTextMode`.

open.py

```
def open(file, mode = 'r'):
    ...
```

open.pyi

```
from typing import Iterable
from typing_extensions import TypeVar, overload

OpenTextMode = Literal["r", "r+"] # AND MANY MORE

def open(file: str, mode: OpenTextMode = 'r')
```

Beyond the ducks

Type stubs can explain complex class type requirements.

The example defines a `Protocol` typehint `OpenRead`:

- which has two methods: `open` and `read`.
- the signature of `open` follows builtin `open`.
- the signature of `read` has no parameters.

```
from typing import Protocol
```

```
class OpenRead(Protocol):  
    def open(self, file: str) -> None: ...  
  
    def read(self) -> str: ...
```

Beyond the ducks

Type stubs can explain complex class type requirements.

The example defines a `Protocol` typehint `OpenRead`:

- which has two methods: `open` and `read`.
- the signature of `open` follows builtin `open`.
- the signature of `read` has no parameters.

All variables/parameters of type `OpenRead` must adhere!

```
from typing import Protocol
```

```
class OpenRead(Protocol):  
    def open(self, file: str) -> None  
  
    def read(self) -> str: ...
```

```
def Read(thing: OpenRead) -> str:  
    return thing.read()
```

Beyond the ducks

Type stubs can explain complex class type requirements.

The example defines a `Protocol` typehint `OpenRead`:

- which has two methods: `open` and `read`.
- the signature of `open` follows builtin `open`.
- the signature of `read` has no parameters.

All variables/parameters of type `OpenRead` must adhere!

Their types (e.g. `Data`) must also adhere but not inherit!

```
from typing import Protocol
```

```
class OpenRead(Protocol):  
    def open(self, file: str) -> None  
  
    def read(self) -> str: ...
```

```
class Data:  
    def open(self, file: str) -> None  
        self.data = open(file)
```

```
    def read(self) -> str:  
        return data.read()
```

```
def Read(thing: OpenRead) -> str:  
    return thing.read()
```

**Before we invest,
can we automate?**

Before we invest, can we automate?

Github

.pre-commit-config.yaml

```
name: "mypy check"
on: [pull_request]
jobs:
  static-type-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v3
        with:
          python-version: '3.11'
      - run: pip install mypy
      - name: Get Python changed files
        id: changed-py-files
        uses: tj-actions/changed-files@v23
        with:
          files: **.py
      - name: Run if any of the listed files above is changed
        if: steps.changed-py-files.outputs.any_changed == 'true'
        run: mypy ${{ steps.changed-py-files.outputs.all_changed_files }} --ignore-missing-imports
```

Do not be disruptive

Checks have to be fast.

Checks have to be helpful.

The team has to accept (not agree).

Things must be documented.

Sanity for everyone

GitHub

.pre-commit-config.yaml

repos:

- repo: <https://github.com/pre-commit/pre-commit-hooks>
rev: v3.4.0
hooks:
 - id: trailing-whitespace
 - id: end-of-file-fixer
 - id: check-yaml
- repo: <https://github.com/MichaelAquilina/pre-commit-hooks>
rev: 316de29ff011015cf49b2d64d9fba41abf8e4281
hooks:
 - id: requirements-txt-fixer
- repo: <https://github.com/psf/black>
rev: 24.4.2
hooks:
 - id: black
- repo: <https://github.com/pycqa/isort>
rev: 5.12.0
hooks:
 - id: isort

Custom stuff

Github

.pre-commit-config.yaml

repos:

- repo: local

hooks:

- id: no-do-not-merge

name: No 'DO NOT MERGE'

description: |

* No: 'DONOTMERGE', 'DONOTSUBMIT'

* No: 'DO NOT MERGE', 'DO NOT SUBMIT'

* No: 'DON'T MERGE', 'DON'T SUBMIT'

* Or the same with underscores instead of spaces to prevent merging.

* To run `pre-commit` without this hook, run `SKIP="no-do-not-merge" kpre-commit`.

* Use `export SKIP="no-do-not-merge"` to disable the hook locally.

language: pygrep

args: [-i]

entry: DO([_]?NOTIN'T)[_]?(SUBMITMERGE)

exclude: ^.pre-commit-config.yaml\$

types: [text]

Custom stuff

Github

.pre-commit-config.yaml

repos:

- repo: local

hooks:

- id: no-todos-without-context

name: No TODOs without context

description: |

* Use descriptive, referenceable TODOs and FIXMEs. Like:

* `// TODO(nero.windstriker)`

* `# FIXME(https://my.company.com/bugs/XYZ-919)`

* `/* TODO(XYZ-919) */`

* Read google style guide for details.

* A bug or other URL reference is better than a person, as the person may leave the team/company.

* A developer can be identified by their email addresses (or just the handle@).

* If a developer is referenced then it does not need to be the one writing the TODO.

* It is more important to note the most knowledgeable person.

* If you add someone else, first clarify with them.

language: pygrep

args: [-i]

entry: (#[//|/[*]).*(FIXME|TODO)\b(?:!|[(((\w\w[-]\w)+@(\w+([\.]w+)*?)|(https?://[^\s]+)\w+-[1-9][0-9]*)))(,[^\s]+)?[)])

types: [text]



Thanks

—

Questions?

Links

<https://mypy.readthedocs.io/en/stable/index.html>

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

<https://docs.python.org/3/library/collections.abc.html>