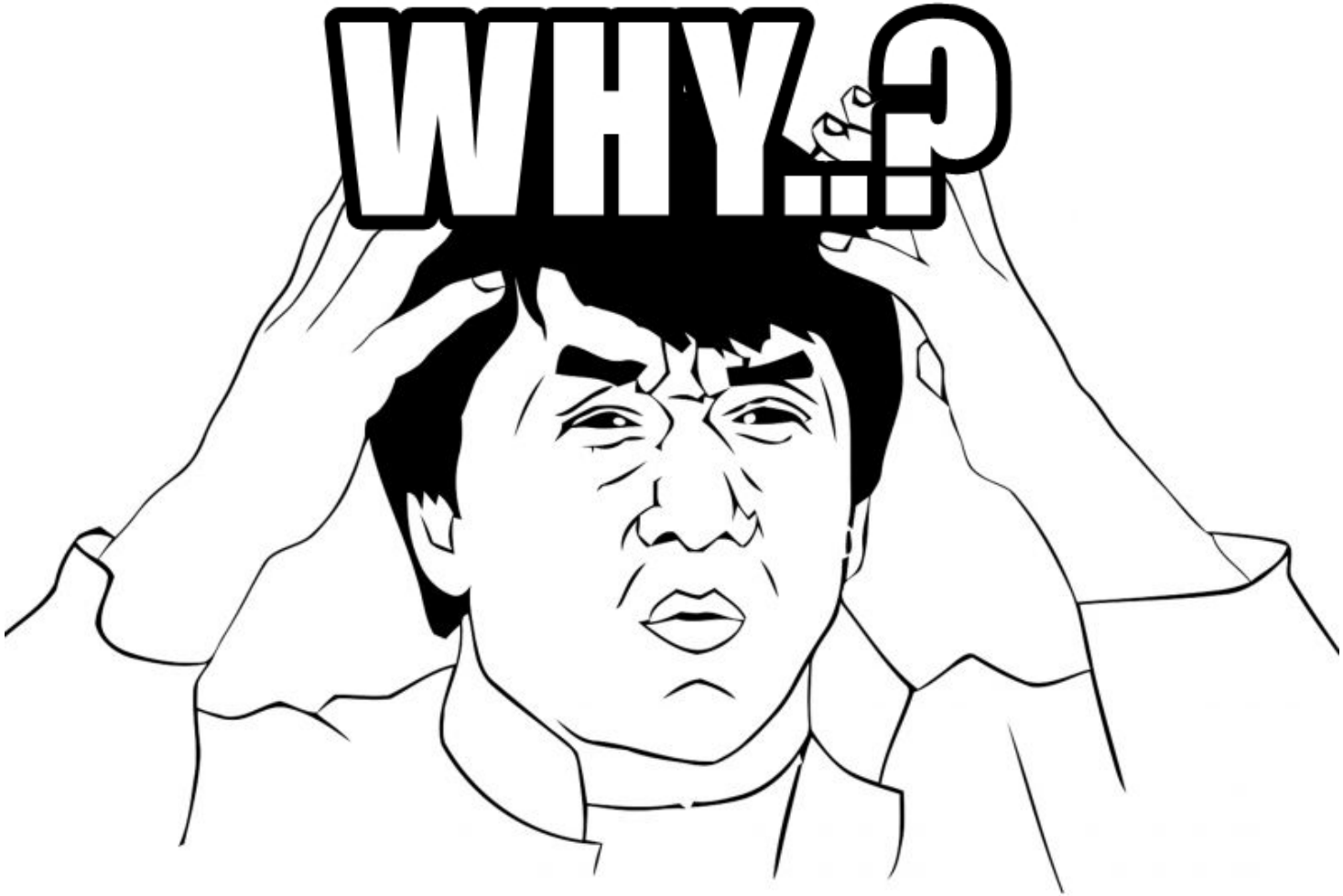
STRONGLY TYPED PYTHON

GINTARE URBONE
SOFTWARE BACKEND ENGINEER @ THOUGHT MACHINE

Thought Machine

IN THIS TALK

- Why care about strong typing?
- Typing in python
- Static VS runtime type validation in python
- Do it yourself
  - Validate function arguments using decorators
  - Lock class attribute types with meta-classes
  - Validate class method arguments using decorators
  - Automatically generated doc-strings from typing attributes
  - Create typed data-structures (the tuple example)
- To sum-up


WHY..?

WHY WE CARE ABOUT STRONG TYPING?

Goal: python based ‘smart contracts’ for core-banking system

Challenge: manual testing takes too long

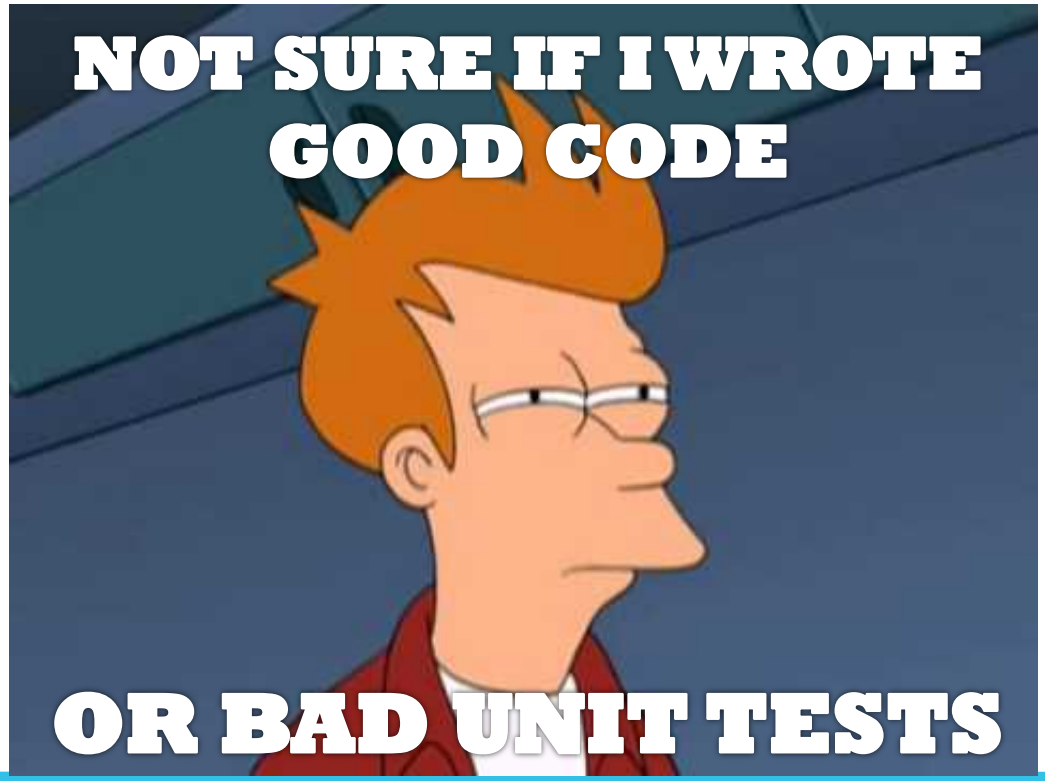
Solution: strong typing in automated ‘smart contract’ testing framework → avoid incompatible data-type errors at runtime and false positive assertions in tests

BUT REALLY WHY?

WHY EVERYONE SHOULD CARE ABOUT STRONG TYPING?

- Handling bad input errors when they happen is not always an option.
- For python libraries that act as a ‘middleware’ between a main system and user interface, majority of code errors can be caught if full ‘middleware’ code data-types validation is done.
- Even having full unit test coverage not necessarily prevents incompatible data-type errors as dependencies are usually mocked in unit test.

HAVING FULL UNIT TEST COVERAGE DOES NOT GUARANTEE NO ISSUES IN PRODUCTION BECAUSE OF BAD DATA TYPES



DISCLAIMER

- Today I will present my home project which is inspired by the work I have done at Thought Machine to build ‘smart contract’ code type validation.
- With my project I took a new approach to solve this problem and re-implemented validation in a new way.
- All code examples in this presentation are from my personal type validation library.

TYPING IN PYTHON

```
def submit_payment(amount, account_id, reference=None):
    """
    Submit a payment instruction to payment processor queue
    """
    print(
        'Parameter types: account_id - {}, reference - {}, amount- {}'.format(
            type(account_id), type(reference), type(amount)
        )
    )
    # Submit payment to payment processor queue: no feedback if payment is
invalid
    payment_processor.submit(account_id, amount, reference)
    return

submit_payment(12.3, 'INTERNAL_123', 'Paying interest')
submit_payment('12.3', 123, 42)
```

python3 example_1.py

TYPING IN PYTHON

```
$ python3 example_1.py
Parameter types: account_id - <class 'str'>, reference -
<class 'str'>, amount- <class 'float'>

Parameter types: account_id - <class 'int'>, reference -
<class 'int'>, amount- <class 'str'>
```

TYPING IN PYTHON

https://docs.python.org/3/library/typing.html
https://www.python.org/dev/peps/pep-0483/
https://www.python.org/dev/peps/pep-0484/

```
import typing

def submit_payment(
        amount: float, account_id: str, reference: typing.Optional[str] = None
) -> None:
    """
    Submit a payment instruction to payment processor queue
    """
    print(
        'Parameter types: account_id - {}, reference - {}, amount- {}'.format(
            type(account_id), type(reference), type(amount)
        )
    )
    # Submit payment to payment processor queue: no feedback if payment is invalid
    payment_processor.submit(account_id, amount, reference)
    return

print(typing.get_type_hints(submit_payment))
submit_payment('12.3', 123, 42)
```

python3 example_2.py

TYPING IN PYTHON

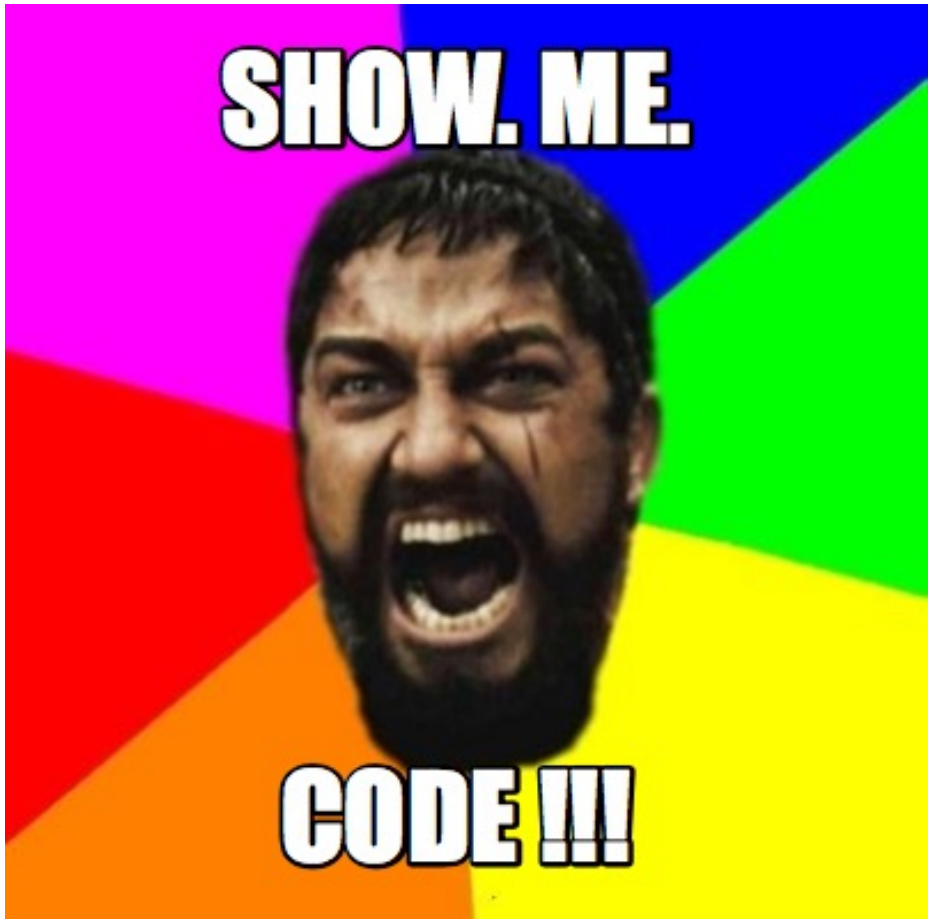
```
$ python3 example_2.py
{'amount': <class 'float'>, 'account_id': <class 'str'>,
'reference': typing.Union[str, NoneType], 'return':
<class 'NoneType'>}

Parameter types: account_id - <class 'int'>, reference -
<class 'int'>, amount- <class 'str'>
```

NO TYPE ERRORS RETURNED EVEN THOUGH TYPES ARE DECLARED

STATIC VS RUNTIME TYPE VALIDATION IN PYTHON

- The python typing module introduces type annotations, however, it does not enforce correct types at runtime.
- There are multiple open-source libraries available that do static type checking or runtime type validations (i.e. Mypy, Enforce).
- Today I will demonstrate how to ‘do-it-yourself’ the runtime type validation in python.
- Run time validation works better if we want to use objects from the same library in unit tests and ensure correct data-types are asserted.



Follow code examples @https://github.com/urbonegi/strongly_typed_python

DIY: VALIDATE FUNCTION ARGUMENTS USING DECORATORS

```
@validate_function_decorator
def submit_payment(
        amount: float, account_id: str, reference: typing.Optional[str] = None
) -> None:
    """
    Submit a payment instruction to payment processor queue
    """
    print(
        'Parameter types: account_id - {}, reference - {}, amount- {}'.format(
            type(account_id), type(reference), type(amount)
        )
    )
    # Submit payment to payment processor queue: no feedback if payment is
invalid
    payment_processor.submit(account_id, amount, reference)
    return

submit_payment('test', 123, 42)
```

python3 example_3.py

DIY: VALIDATE FUNCTION ARGUMENTS USING DECORATORS

```
$ python3 example_3.py
Traceback (most recent call last):
  File "example_3.py", line 60, in <module>
    submit_payment('test', 123, 42)
  File "example_3.py", line 12, in wrapper
    validate_method_arguments(func, args, kwargs)
  File "example_3.py", line 30, in
validate_method_arguments
    validate_type(attr_type, args[index], attr)
  File "/apps/strongly_typed_python/validation.py", line
44, in validate_type
    expected_type
TypeError: Attribute `amount` value - `test` does not
match type declared in annotations - <class 'float'>.
```

DIY: VALIDATE FUNCTION RETURN TYPE USING DECORATORS

```
@validate_function_decorator
def submit_payment(
        amount: float, account_id: str, reference: typing.Optional[str] = None
) -> None:
    """
    Submit a payment instruction to payment processor queue
    """
    print(
        'Parameter types: account_id - {}, reference - {}, amount- {}'.format(
            type(account_id), type(reference), type(amount)
        )
    )
    # Submit payment to payment processor queue: no feedback if payment is invalid
    payment_processor.submit(account_id, amount, reference)
    return 1

submit_payment(12.3, 'INTERNAL_123', 'Paying interest')
```

python3 example_4.py

DIY: VALIDATE FUNCTION RETURN TYPE USING DECORATORS

```
$ python3 example_4.py
Parameter types: account_id - <class 'str'>, reference - <class
'str'>, amount- <class 'float'>
Traceback (most recent call last):
  File "example_4.py", line 60, in <module>
    submit_payment(12.3, 'INTERNAL_123', 'Paying interest')
  File "example_4.py", line 14, in wrapper
    validate_return(func, return_value)
  File "example_4.py", line 40, in validate_return
    validate_type(return_type, return_value, 'return_value')
  File "/apps/strongly_typed_python/validation.py", line 44, in
validate_type
    expected_type
TypeError: Attribute `return_value` value - `1` does not match
type declared in annotations - <class 'NoneType'>.
```

DIY: VALIDATE FUNCTIONS USING DECORATORS

```
from functools import wraps

def validate_function_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        validate_method_arguments(func, args, kwargs)
        return_value = func(*args, **kwargs)
        validate_return(func, return_value)
        return return_value
    return wrapper
```

DIY: VALIDATE FUNCTIONS USING DECORATORS

```
def validate_method_arguments(func, args, kwargs):
    # typing.get_type_hints return example from before:
    # {'amount': <class 'float'>, 'account_id': <class 'str'>,
    # 'reference': typing.Union[str, NoneType], 'return': <class 'NoneType'>}
    types_dict = typing.get_type_hints(func)
    types_dict.pop('return', None)
    # Validate argument types
    for index, (attr, attr_type) in enumerate(types_dict.items()):
        if attr in kwargs:
            validate_type(attr_type, kwargs[attr], attr)
        elif len(args) - 1 >= index:
            validate_type(attr_type, args[index], attr)
        else:
            validate_type(attr_type, None, attr)
```

DIY: VALIDATE FUNCTIONS USING DECORATORS

```
def validate_return(func, return_value):
    types_dict = typing.get_type_hints(func)
    return_type = types_dict['return']
    # Validate return type
    if return_value:
        validate_type(return_type, return_value, 'return_value')
```

VALIDATE TYPES

```
def validate_type(expected_type, given_value, property_name):
    if not _validate_type(expected_type, given_value):
        raise TypeError(
            'Attribute `{}` value - `{}` does not match '
            'type declared in annotations - {}.'.format(
                property_name,
                given_value,
                expected_type
            )
        )
```


MY EYESSS!

CLASS ATTRIBUTES VALIDATION WITH META-CLASSES

```
class Payment(metaclass=StrongTyping):
    """
    Payment object defines funds transfer event.
    """
    amount: float
    account_id: str

    def __init__(self, amount: float, account_number: int, internal: bool) -> None:
        self.account_id = '{}_{}'.format('INTERNAL' if internal else 'EXTERNAL',
account_number)
        self.amount = amount

    def __repr__(self) -> str:
        return 'Account ID: {}, type ({}); amount: {}, type ({}).'.format(
            self.account_id, type(self.account_id),
            self.amount, type(self.amount)
        )

payment = Payment(amount=1.2, account_number=123, internal=True)
print(payment.account_id, type(payment.account_id))
payment.account_id = 123
```

python3 example_5.py

CLASS ATTRIBUTES VALIDATION WITH META-CLASSES

```
$ python3 example_5.py
INTERNAL_123 <class 'str'>
Traceback (most recent call last):
  File "example_5.py", line 48, in <module>
    payment.account_id = 123
  File "example_5.py", line 12, in set_attribute
    validate_type(property_type, property_value,
property_name)
  File "/apps/strongly_typed_python/validation.py", line
44, in validate_type
    expected_type
TypeError: Attribute `account_id` value - `123` does not
match type declared in annotations - <class 'str'>.
```

CLASS ATTRIBUTES VALIDATION WITH META-CLASSES

```
class StrongTyping(type):
    @staticmethod
    def _add_attribute(cls, property_name, property_type):
        def set_attribute(self, property_value):
            validate_type(property_type, property_value, property_name)
            setattr(self.__class__, '_' + property_name, property_value)

        def get_attribute(self):
            return getattr(self, '_' + property_name, None)
        setattr(cls, property_name, property(get_attribute, set_attribute))

    def __init__(cls, name, bases, dct):
        # class attribute validation - create settter/getter prop
        for key, val in typing.get_type_hints(cls).items():
            StrongTyping._add_attribute(cls, key, val)
        super(StrongTyping, cls).__init__(name, bases, dct)
```

VALIDATE CLASS METHODS USING DECORATORS

```
class Payment(metaclass=StrongTyping):
    """
    Payment object defines funds transfer event.
    """
    def __init__(
            self, amount: float, account_number: int, internal: typing.Optional[bool] = False
    ) -> None:
        self.account_id = '{}_{}'.format('INTERNAL' if internal else 'EXTERNAL',
account_number)
        self.amount = amount
        self.timestamp = datetime.now().timestamp()

    def get_status(self, timestamp: datetime) -> str:
        """
        Get payment event status
        """
        return 'PENDING' if self.timestamp > timestamp.timestamp() else 'COMMITTED'

payment = Payment(amount=1.2, account_number=42)
payment.get_status(timestamp=datetime.now())
payment.get_status(timestamp=1)
```

python3 example_6.py

VALIDATE CLASS METHODS USING DECORATORS

```
$ python3 example_6.py
Traceback (most recent call last):
  File "example_6.py", line 88, in <module>
    payment.get_status(timestamp=1)
  File "example_6.py", line 36, in wrapper
    validate_method_arguments(func, args, kwargs)
  File "example_6.py", line 18, in
validate_method_arguments
    validate_type(attr_type, kwargs[attr], attr)
  File "/apps/strongly_typed_python/validation.py", line
44, in validate_type
    expected_type
TypeError: Attribute `timestamp` value - `1` does not
match type declared in annotations - <class
'datetime.datetime'>.
```

VALIDATE CLASS METHODS USING DECORATORS

```
class StrongTyping(type):
    @staticmethod
    def _add_attribute(cls, property_name, property_type):
        def set_attribute(self, property_value):
            validate_type(property_type, property_value, property_name)
            setattr(self.__class__, '_' + property_name, property_value)

        def get_attribute(self):
            return getattr(self, '_' + property_name, None)
        setattr(cls, property_name, property(get_attribute, set_attribute))

    def __init__(cls, name, bases, dct):
        # Add decorators to each class method to validate args and return value
        for name, obj in dct.items():
            # Only decorate class methods
            if hasattr(obj, '__call__'):
                setattr(cls, name, validate_class_method_decorator(obj))
        # Validate class attributes - create settter/getter properties
        for key, val in typing.get_type_hints(cls).items():
            StrongTyping._add_attribute(cls, key, val)
        super(StrongTyping, cls).__init__(name, bases, dct)
```


PLEASE
NO MORE

AUTOMATICALLY GENERATED DOC-STRINGS

- Doc-string of Payment class and its methods do not have arguments and their types information. Let's see what we can do about that?

```
print(Payment.__doc__)
```

```
    Payment object defines funds transfer event.
```

```
print(Payment.get_status.__doc__)
```

```
        Get payment event status
```

- Let's add all class attributes, method arguments and return types to the doc-strings of our classes and their methods. We will use this doc-string pattern here:

  :param arg1: description
  :param arg2: description
  :type arg1: type description
  :type arg2: type description
  :return: return description
  :rtype: the return type description

AUTOMATICALLY GENERATED DOC-STRINGS

```
class Payment(metaclass=StrongTyping):
    """
    Payment object defines funds transfer event.
    """
    amount: Type(float, "Payment amount")
    account_id: Type(str, "Target account ID")
    timestamp: Type(float, "Payment timestamp")

    def __init__(self, amount: Type(float, "Payment amount"),
                 account_number: Type(int, "Target account number"),
                 internal: Type(bool, "Is account internal or external?")) -> Type(None,
""):
        """
        Payment object constructor method
        """
        self.account_id = '{}_{}'.format('INTERNAL' if internal else 'EXTERNAL',
account_number)
        self.amount = amount
        self.timestamp = datetime.now().timestamp()
```

python3 example_7.py

AUTOMATICALLY GENERATED DOC-STRINGS

```
print(Payment.__doc__)
print(Payment.__init__.__doc__)
print(Payment.get_status.__doc__)
```

```
    Payment object defines funds transfer event.
    :param amount: Payment amount
    :param account_id: Target account ID
    :param timestamp: Payment timestamp
    :type amount: <class 'float'>
    :type account_id: <class 'str'>
    :type timestamp: <class 'float'>

        Payment object constructor method
        :param amount: Payment amount
        :param account_number: Target account number
        :param internal: Is account internal or external?
        :type amount: <class 'float'>
        :type account_number: <class 'int'>
        :type internal: <class 'bool'>
        :return:
        :rtype: None

        Get payment event status
        :param timestamp: Payment validation timestamp
        :type timestamp: <class 'datetime.datetime'>
        :return: Payment status at given timestamp
        :rtype: <class 'str'>
```

AUTOMATICALLY GENERATED DOC-STRINGS

```
class Type:
    def __init__(self, type, description=None):
        self.description = description
        self.type = type

def add_function_docs(wrapper, func):
    type_annot = func.__annotations__
    func_doc = func.__doc__ or '\n        '
    wrapper.__doc__ = func_doc + extend_docstring(type_annot, spaces=8)
    return wrapper

def validate_class_method_decorator(func):
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        validate_method_arguments(func, args, kwargs)
        return_value = func(self, *args, **kwargs)
        validate_return(func, return_value)
        return return_value
    return add_function_docs(wrapper, func)
```

AUTOMATICALLY GENERATED DOC-STRINGS

```
def extend_docstring(type_annot, spaces=4):
    docs = []
    docs.extend(
        (':param {}: {}'.format(attr, attr_type.description)
         for (attr, attr_type) in type_annot.items() if attr != 'return')
    )
    docs.extend(
        (':type {}: {}'.format(attr, attr_type.type)
         for (attr, attr_type) in type_annot.items() if attr != 'return')
    )
    # Add return type if exist
    if 'return' in type_annot:
        docs.extend([':return: {}'.format(type_annot['return'].description),
                     ':rtype: {}'.format(type_annot['return'].type)])
    return ('\n' + ' ' * spaces).join(docs)
```

TYPED DATA-STRUCTURES: TUPLE

```
class TypedTuple(tuple, metaclass=StrongTyping):
    """
    Fixed type tuple
    """

    def __new__(
            cls, iterable: Type(typing.Optional[typing.Iterable[str]], "Iterable of type
str") = ()
    ) -> Type(typing.Tuple[str, ...], "Tuple of any length with items of str type"):
        return super().__new__(cls, iterable)


print(TypedTuple())
print(TypedTuple(['1', '2']))
print(TypedTuple.__new__.__doc__)
TypedTuple([1, 2])
```

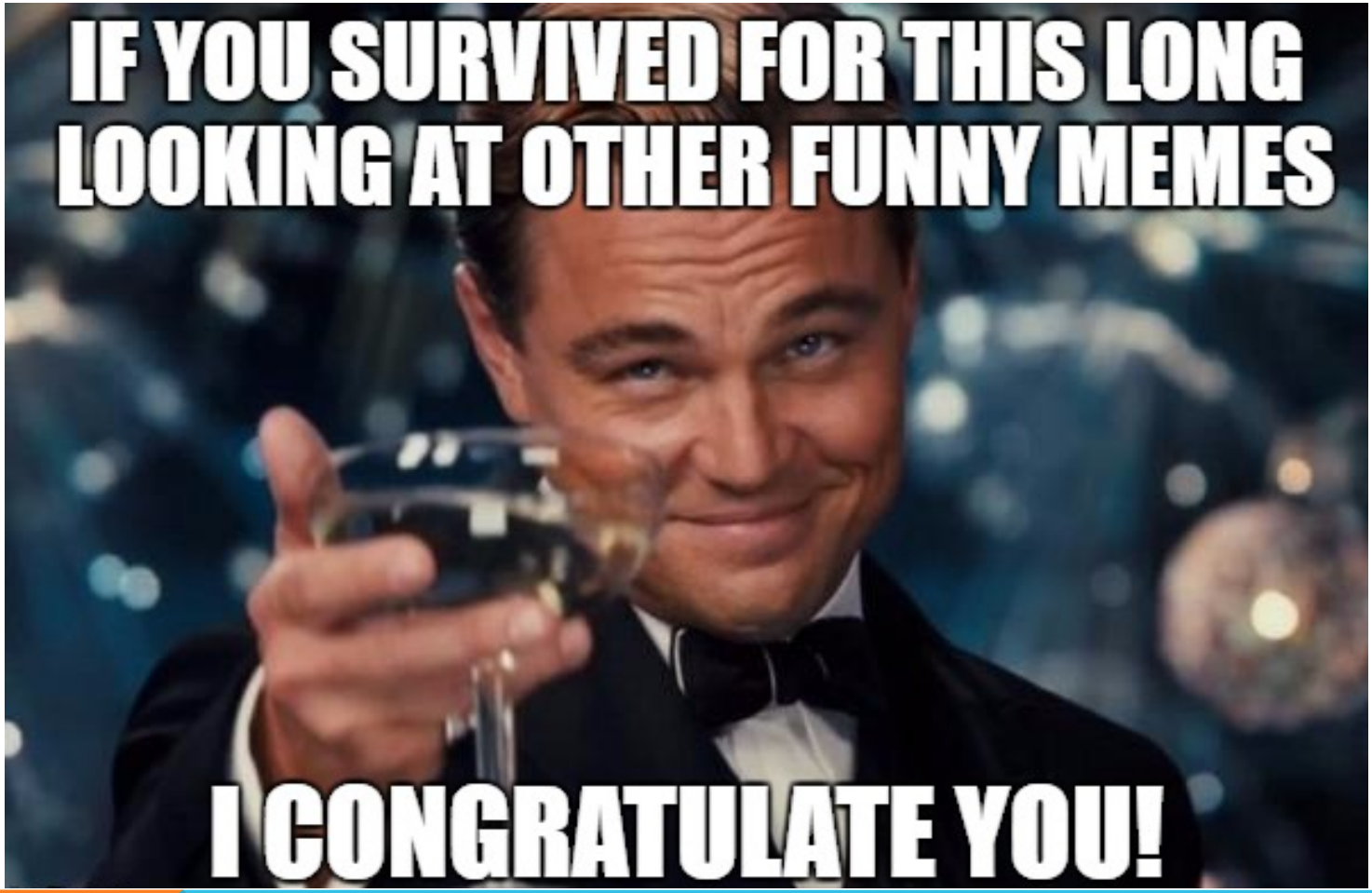
python3 example_8.py

TYPED DATA-STRUCTURES: TUPLE

```
$ python3 example_8.py
()
(1,2)

    :param iterable: Iterable of type str
    :type iterable: typing.Union[typing.Iterable[str], NoneType]
    :return: Tuple of any length with items of str type
    :rtype: typing.Tuple[str, ...]

Traceback (most recent call last):
  File "example_8.py", line 18, in <module>
 TypedTuple([1,2])
  File "/apps/strongly_typed_python/example_7.py", line 63, in
wrapper
    validate_method_arguments(func, args, kwargs)
  File "/apps/strongly_typed_python/example_7.py", line 17, in
validate_method_arguments
    validate_type(attr_type.type, args[index], attr)
  File "/apps/strongly_typed_python/validation.py", line 44, in
validate_type
    expected_type
TypeError: Attribute `iterable` value - `[1, 2]` does not match type
declared in annotations - typing.Union[typing.Iterable[str],
NoneType].
```


IF YOU SURVIVED FOR THIS LONG
LOOKING AT OTHER FUNNY MEMES
I CONGRATULATE YOU!

TO SUM-UP

- The typing module is a great pythonic way to declare type annotations. However, it does not do runtime type validations.
- There has been a growing need to support strong typing in python for some use cases.
- I think that the typing library will implement such methods as *'_validate_type()'* eventually.
- By looking at quite a few code examples today, we realized that it is not so difficult to DIY type validations at runtime using python.
- Having strong type validation at runtime could prevent many errors and bugs, especially for middleware applications.

THANK YOU!

QUESTIONS?

Contact me

@Twitter:
GintareUrbone
@Linkedin:
https://www.linkedin.com/in/gintare-urbone-1203206/
@Github:
https://github.com/urbonegi/strongly_typed_python

Checkout Thought Machine:
https://www.thoughtmachine.net

Thought Machine

APPENDIX

APPENDIX: STATIC TYPES VALIDATION: MYPY

```
$ mypy example_2.py
example_2.py:23: error: Argument 1 to "submit_payment" has
incompatible type "str"; expected "float"
example_2.py:23: error: Argument 2 to "submit_payment" has
incompatible type "int"; expected "str"
example_2.py:23: error: Argument 3 to "submit_payment" has
incompatible type "int"; expected "Optional[str]"
```

APPENDIX: DYNAMIC TYPE CHECKING: ENFORCE

```
from enforce import runtime_validation

@runtime_validation
def hello(greet: str) -> str:
    return 1

@runtime_validation
class Test:
    name: str

    def __init__(self, name: str) -> None:
        self.name = name

test = Test('name')
test.name = 1
```

APPENDIX: _VALIDATE_TYPE

```
def _validate_type(expected_type, given_value):
    if expected_type == type(given_value):
        return True
    elif expected_type.__class__ == Union.__class__:
        if any(
            _validate_type(tp, given_value) for tp in expected_type.__args__
        ):
            return True
 elif issubclass(expected_type, Iterable) and isinstance(given_value, Iterable):
        if all(
            _validate_type(expected_type.__args__[0], i) for i in given_value
        ):
            return True
    return False
```

APPENDIX: AUTOMATICALLY GENERATED DOC-STRINGS

```
$ python3 example_7.py

Payment object defines funds transfer event.
:param amount: Payment amount
:param account_id: Target account ID
:param timestamp: Payment timestamp
:type amount: <class 'float'>
:type account_id: <class 'str'>
:type timestamp: <class 'float'>

Payment object constructor method
:param amount: Payment amount
:param account_number: Target account number
:param internal: Is account internal or external?
:type amount: <class 'float'>
:type account_number: <class 'int'>
:type internal: <class 'bool'>
:return:
:rtype: None

Get payment event status
:param timestamp: Payment validation timestamp
:type timestamp: <class 'datetime.datetime'>
:return: Payment status at given timestamp
:rtype: <class 'str'>
```

APPENDIX: _VALIDATE_TYPE

```
    elif issubclass(expected_type, Tuple) and isinstance(given_value, tuple):
        if len(expected_type.__args__) > 1 and expected_type.__args__[1] ==
Ellipsis:
            # Tuple of any length of elements with same type
            if all(
                _validate_type(expected_type.__args__[0], value) for value in
given_value
            ):
                return True
        else:
            # Fixed length tuple
            if len(given_value) == len(expected_type.__args__) and \
                all(
                    _validate_type(
                        val, given_value[index]
                    ) for index, val in enumerate(expected_type.__args__)
                ):
                return True
```

APPENDIX: VALIDATE CLASS METHODS USING DECORATORS

```
def validate_class_method_decorator(func):
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        validate_method_arguments(func, args, kwargs)
        return_value = func(self, *args, **kwargs)
        validate_return(func, return_value)
        return return_value
    return wrapper
```

APPENDIX: AUTOMATICALLY GENERATED DOC-STRINGS

```
def validate_method_arguments(func, args, kwargs):
    types_dict = func.__annotations__.copy()
    types_dict.pop('return', None)
    # Validate argument types
    for index, (attr, attr_type) in enumerate(types_dict.items()):
        if attr in kwargs:
            validate_type(attr_type.type, kwargs[attr], attr)
        elif len(args) - 1 >= index:
            validate_type(attr_type.type, args[index], attr)
        else:
            validate_type(attr_type.type, None, attr)

def validate_return(func, return_value):
    types_dict = func.__annotations__
    return_type = types_dict['return']
    # Validate return type
    if return_value:
        validate_type(return_type.type, return_value, 'return_value')
```

APPENDIX: UNDERSTANDING SMART CONTRACTS