PyCon China 2020

PyConChina 2020
PYTHON 中国开发者大会 2020

Python For Good

Python强类型编程最佳实践

丁来强

阿里云SLS（日志服务）上海团队负责人

关于我

- 工作10+年，熟悉大数据分析、ITOps、SecOps等领域
- 阿里云SLS（日志服务）上海负责人，之前在Splunk。

- 2015年起，连续在多届PyCon活动上，分享超过9个不同Python相关议题
- 云栖大会或社区累计分享15+个大数据系统或Python相关议题

日志服务钉钉群

往届视频与PPT

议题

- Python强类型检查的历史、背景
- Python强类型检查的策略和常用场景实践（20+个）

Pyton语言与类型限定

- Python是一门强类型的动态类型语言
  - 可以动态构造脚本执行、修改函数、对象类型结构、变量类型
  - 但不允许类型不匹配的操作

```
cmd = """
def say_hello(name):
    print(f"你好 {name}")
say_hello("小明")
"""

exec(cmd)

say_hello.__name__ = "greeting"

v1 = [1, ("a", "b"), {1, "x", "y"}]
```

```
a = 100
b = "200"
a + b

Traceback (most recent call last):
  File "pybasic.py", line 16, in <
      module>
    a + b
TypeError: unsupported operand type(s)
    for +: 'int' and 'str'
```

动态语言 vs 类型提示

- 有了类型标注提示后，可以在编码时即发现错误。

```
a = 100
b = "abc"

def add(v1, v2):
    """两个数字相加"""
    return v1+v2

add(1, 2)
add("abc", "xyz")
# > 代码不会报错，工具不会提示错误
add("abc", 1)
# > 代码错误，部分工具会提示错误
```

```
a: int = 100
b: str = "abc"

def add(v1: int, v2: int) -> int:
    return v1+v2

add(1, 2)
add("abc", "xyz")
# > 可以执行，工具/IDE扫描会提示错误
add("abc", 1)
# > 执行错误，工具/IDE扫描会提示错误
```

函数标注（PEP 3107）

- 对函数参数、返回值标注，放在__annotations__里面。
- IDE、工具可以在代码层面静态检查并提示错误。

```
def add(v1: int, v2: int) -> int:
    return v1 + v2


add(100, "abc")
```

```
add("xyz", "zzz")
```

Expected type 'int', got 'str' instead

Expected type 'int', got 'str' instead

```
pep3107
def add(v1: int,
        v2: int) -> int
```

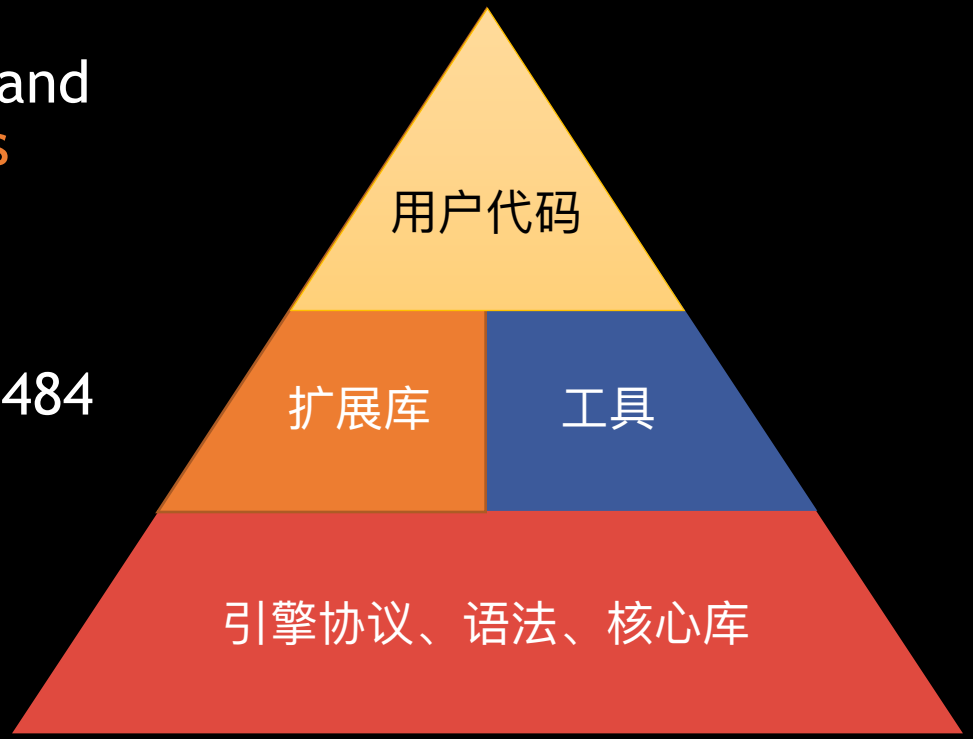
强类型检查的优势与现状

- 几大优势：
  - 易读：比docstring更易理解接口协议，也更易使用三方库（IDE工具支持）
  - 排错：编码、编译期间即可发现错误（IDE工具支持）。
  - 重构：接口范围明确，更易于理解和放心重构。
  - 性能：（可能）可以做到静态编译优化。
- 现实：
  - 生态-IDE：三方类型检查工具比较全面（PyCharm、VSCode、VIM等）。
  - 生态-库工具：三方库有官方或大厂支持且很流行。
    - 官方/mypy-9.5K star、Facebook/pyre-5K Star，Google/pytype-3K star等。
  - 成熟度：静态检查已被广泛验证有效性，尤其大型工程（dropbox迁移400万行到静态标注）。
  - 行业：主流编程语言以静态强类型检查为主（C++、Java、Go等）；其他动态语言的静态类型扩展迅猛发展（如TypeScript）。

Python强类型检查的策略

- Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.

——Guido in PEP 484



Python类型系统到Py3.7（2018.6）才基本成型

- PEP 3107(2006.12 #Py3.0)：Py3从一开始就引入函数标注语法（Function Annotations）
- PEP 483（2014.12 纯文档）：详细介绍了PEP 484（类型提示）中涉及到的的理论
- PEP 484（2014.09 #Py3.5）：引入类型提示（Type Hints）
- PEP 526（2016.08 #Py3.6）：引入变量标注语法（Variable Annotations）
- PEP 544（2017.03 #Py3.8）：引入结构化子类型协议（static duck typing）
- PEP 557（2017.06 #Py3.7）：引入Data Classes数据结构
- PEP 560（2017.09 #Py3.7）：在CPython解释器中新增类型提示与泛型支持
- PEP 563（2017.09 #Py3.7）：支持延迟（禁用）类型提示延迟评估
- PEP 585（2019.03 #Py3.7/9）：标准集合支持泛型类型提示
- PEP 586（2019.03 #Py3.8）：引入字面量类型（Literal Types）
- PEP 589（2019.03 #Py3.8）：引入类型字典（TypedDict）
- PEP 591（2019.03 #Py3.8）：引入final限定符（typing.final and Final）
- PEP 593（2019.04 #Py3.9）：引入类型标注类型（Annotated）
- PEP 613（2020.01 #Py3.10）：引入显式类型别名（TypeAlias）

2018.6 Py3.7

- 扩展库方面
- 引擎协议语法方面

议题

- Python强类型检查的历史、背景
- Python强类型检查的策略和常用场景实践（20+个）

什么时候需要或不需要类型检查?

- 以下建议采用:
  - 提供SDK、库/接口给其他人时。
    - 比docstring更清晰、主流IDE支持提示和校验。
  - 代码行数越多，价值越大。
    - 规范化编码，通过工具可以辅助发现潜在BUG。
  - 需要写UT（单元测试）的地方，就需要类型检查 (by Bernat Gabor)
- 以下需要一定策略:
  - 原型（Prototype）或验证性质项目（POC）的代码，可以先不引入。
  - 大量旧有代码，需要逐步阶段性引入（参考Dropbox经验）。
  - 不熟悉Python和类型提供功能用法时，可以先不引入。

复杂数据结构的标注（使用别名）（PEP 484）

- 类型别名用于简化类型提示标注。（这里直接使用了内置类型[]做标注（PEP 585））

```python
from collections.abc import Sequence

ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]

def fn1(message: str,
        servers: Sequence[Server]) -> None:
    ...

# 等价于
def fn1(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str, str]]])
        -> None:
    ...
```

变参类型标注（PEP 484）

- 对变参、命名变参直接标注其值的类型。

```python
def foo(*args: str, **kwds: int): pass

foo('a', 'b', 'c')
foo('a', 'b', 1)  # 错误
foo(x=1, y=2)
foo(x=1, y='x')   # 错误
foo('', z=0)
```

类别选择 Union vs. TypeVar（PEP 484）

- 使用Union定义可能的类型选择： x、y、返回值的类型同一时刻可以不同
- 使用TypeVar定义一组类型：x、y、返回值的类型同一时刻必须相同

```
from typing import Union, TypeVar, Text

AnyStr1 = Union[Text, bytes]

def concat1(x: AnyStr1, y: AnyStr1) -> AnyStr1:
    pass

AnyStr2 = TypeVar('AnyStr2', Text, bytes)

def concat2(x: AnyStr2, y: AnyStr2) -> AnyStr2:
    pass
```

```
concat1('a', 'b')    # 正确
concat1(b'a', b'b') # 正确
concat1('a', b'b')  # 正确

concat2('a', 'b')    # 正确
concat2(b'a', b'b') # 正确
concat2('a', b'b')  # 错误
```

```
Union[T1, None] == Option[T]
```

函数类型与泛型（PEP 484）

- 函数类型：Callable[[参数1类型, 参数2类型, ...], 返回类型]

```
from collections.abc import Callable

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    pass
```

- 泛型：TypeVar

```
from collections.abc import Sequence
from typing import TypeVar

T = TypeVar('T')   # 一个泛型类型

# 接受一个元素类型都是T的序列,返回值的类型也是T
def first(l: Sequence[T]) -> T:    # 泛型函数
    return l[0]

```

```
first([1, 2]) + 100          # 正确
first(['a', 'c']).upper() # 正确
first(['a', 'c']) + 100    # 错误
```

泛型类型（PEP 484）

- 容器下的泛型，使用TypeVar

```
from typing import TypeVar, Iterable

T = TypeVar('T', int, float, complex)
Vector = Iterable[tuple[T, T]]

def inproduct(v: Vector[T]) -> T:
    return sum(x*y for x, y in v)
```

- 类下的泛型，继承Generic

```
from typing import TypeVar, Generic

T = TypeVar('T')
class MyClass(Generic[T]):
    def meth_1(self, x: T) -> T: pass
    def meth_2(self, x: T) -> T: pass

a: MyClass[int] = MyClass()
a.meth_1(1)     # OK
a.meth_2('a')   # 错误
```

使用注释（Comments）进行标注（Annotation）（PEP 484）

- 使用注释 # type: xxxx 进行标注
- 注意函数的标注方式：
  - (参数类型列表) -> 返回值类型

```
from typing import List

class A(object):
    def __init__():
        # type: () -> None
        self.elements = []  # type: List[int]

    def add(element):
        # type: (List[int]) -> None
        self.elements.append(element)
```

将标注放在单独文件（PEP 484)

- 独立的Stub文件（.pyi）与源文件并行即可。
- 优点：
  - 不需要修改源代码（减少引入BUG可能）
  - Pyi可以使用最新的语法（源文件可以是低版本）
  - 测试友好
  - 不拥有的三方库，也可以补充标注信息。
- 缺点：
  - 代码重复写了一遍头（工作量）
  - 打包变得复杂

```python
# a.py
class A(object):
    def __init__():
        self.elements = []

    def add(element):
        self.elements.append(element)


# a.pyi（与a.py同目录）
from typing import List

class A(object):
    elements = ... # type: List[int]
    def __init__() -> None: ...
    def add(element: int) -> None: ...
```

前置引用（PEP 484）

- 例如二叉树节点，需要引用自己。可以用字符串代替类型标注。

```
class Tree:
    def __init__(self, left: 'Tree', right: 'Tree'):
        self.left = left
        self.right = right

    def leaves(self) -> List['Tree']:
        pass
```

函数标注扩展overload（PEP 484）

- 使用overload进行参数返回值描述。
- 必须有一个无修饰版本做真正实现。
- 仅用于静态类型检查，运行时重载可以使用functools.singleddispatch等

```
@overload
def utf8(value: None) -> None: pass

@overload
def utf8(value: bytes) -> bytes: pass

@overload
def utf8(value: unicode) -> bytes: pass

def utf8(value):
    ...实际的实现，必须有....
```

- 有时可以用TypeVar代替（更简洁）

```
from typing import TypeVar

AnyStr = TypeVar('AnyStr', None, unicode, bytes)
AnyStrRet = TypeVar('AnyStr', None, unicode, bytes)

def utf8(value: AnyStr) -> AnyStrRet:
    ...实际实现...
```

协变(covariant)与逆变(contravariant) PEP 484

- 协变：
  - 让一个粗粒度接口（或委托）可以接收一个更加具体的接口（或委托）作为参数（或返回值）；
  - 例如：老鹰列表赋值给鸟列表

- 逆变：
  - 让一个接口（或委托）的参数类型（或返回值）类型更加具体化，也就是参数类型更强，更明确。
  - 例如：鸟列表赋值给老鹰列表

```python
from typing import TypeVar, Generic

T_co = TypeVar('T_co', covariant=True) #协变

class MyList(Generic[T_co]):
    pass

class Bird: pass

class Eagle(Bird): pass


brds: MyList[Bird]
egls: MyList[Eagle] = [Eagle()]

brds = egls
```

dataclasses（PEP 557）

- 带默认值的可变的命名元祖

```
@dataclass
class InventoryItem:
    name: str
    unit_price: float
    quantity_on_hand: int = 0
```

- 复杂域默认值

```
@dataclass
class C:
    mylist: List[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

- 延迟初始化

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

- 与tuple、dict互转

```
@dataclass
class Point:
    x: int
    y: int

@dataclass
class C:
    mylist: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

静态duck typing（PEP 544）

- 定义了类似于Java的接口
  - 可以有实现，可以被继承
  - 可以继承多个接口，构建一个新接口
- 实际类型检查时，不要求继承关系
  - 只需要对象的所有成员方法signature匹配Protocol即可（duck-typing）

```python
from typing import Sized, Protocol

class SizedAndClosable(Sized, Protocol):
    def close(self) -> None:
        pass
```

```python
from typing import Protocol, Iterable

class IResource(Protocol):
    def close(self) -> None:
        pass

class Resource: # 也可以继承IResource（非必须）
    def close(self) -> None:
        pass

def close_all(things: Iterable[IResource]):
    for t in things:
        t.close()

f = open('foo.txt')
r = Resource()
close_all([f, r])  # 通过
close_all([1])     # 不通过
```

泛型静态duck typing（PEP 544）

- 支持泛型Protocol、支持协变或逆变

```python
T = TypeVar('T')

class Iterable(Protocol[T]):
    @abstractmethod
    def __iter__(self) -> Iterator[T]:
        pass
```

```python
T_co = TypeVar('T_co', covariant=True)
                                # 支持协变或逆变
class Box(Protocol[T_co]):
    def content(self) -> T_co:
        pass

box: Box[float]
second_box: Box[int]
box = second_box  # 协变
```

运行时化duck typing（PEP 544）

- 让Protocol可以运行时访问，例如isinstance检查
- 使用runtime_checkable装饰即可

```
from typing import runtime_checkable, Protocol

@runtime_checkable
class SupportsClose(Protocol):
    def close(self):
        ...

assert isinstance(open('some/file'), SupportsClose)
```

字面量类型（PEP 586）

- 不修改类型为enum的情况下，限定传递参数：Literal[字面量1, 字面量2, ...]

```python
# 总是返回True
def validate_simple(data: Any) -> Literal[True]:
    pass

# 只能是这几个值
MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
    pass


open_helper('/some/path', 'r')      # 通过
open_helper('/other/path', 'typo') # 不通过
```

延迟类型提示执行（PEP 563）

- 标注依然会占用资源（空间和时间），可以延迟执行（转化为字符串）。

```
def foo(v1: [1, 2, 3],
        v2: max(1, 2, 3)) \
        -> 1+2+3:
    pass


foo.__annotations__

# 值为:
{'v1': [1, 2, 3], # 列表
 'v2': 3,         # 函数调用结果
 'return': 6}     # 计算结果
```

```
from __future__ import annotations

def foo(v1: [1, 2, 3],
        v2: max(1, 2, 3)) \
        -> 1+2+3:
    pass

foo.__annotations__

# 值为:
{'v1': '[1, 2, 3]',      # 字符串
 'v2': 'max(1, 2, 3)',   # 字符串
 'return': '1 + 2 + 3'} # 字符串
```

静态检查时与运行时区分（PEP 484）

- 可以在静态检查时导入特定的库，运行时不做。
- 这种情况下，相关标注只能用注释或字符串（前置）方式标注。

```
import typing

if typing.TYPE_CHECKING:
    import expensive_mod

def a_func(arg: 'expensive_mod.SomeClass') -> None:
    a_var = arg  # type: expensive_mod.SomeClass
```

关闭静态类型检查（PEP 484）

- 使用no_type_check修饰函数或类来关闭。
- 使用no_type_check_decorator修饰装饰器来关闭。

```
@no_type_check
def add(v1: int, v2: int)->int: pass

@no_type_check_decorator
def log_enter_exit(fn):
    def __wrapped(*args: int, **kwargs: int):
        pass
    return __wrapped
```

typing.Final（PEP 591）

- 指定变量被初始化后无法再被修改、类变量无法被子类修改。
- 声明为Final的类成员的变量，未初始化的，必须在__init__里面初始化。

```python
from typing import Final

MAX_SIZE: Final = 9000
MAX_SIZE += 1  # 错误

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # 错误
```

```python
class ImmutablePoint:
    x: Final[int]
    y: Final[int] # 错误

    def __init__(self) -> None:
        self.x = 1  # 未初始化 y
```

typing.final（PEP 591）

- 运行时版本
- Python官方版本的final
- 可以修饰类（不能被继承）和类的方法（不能被重写）

```python
from typing import final

@final
class Base: pass
class Derived(Base): pass # 错误


class Base:
    @final
    def foo(self): pass

class Derived(Base):
    def foo(self): pass # 错误
```

新的Hook方法（PEP 560）

- 在Python-Core总支持新的Hook方法__class_getitem__，在传入类型时会自动调用。
- 简化typing的实现方式。

```
class MyList:
    def __getitem__(self, index):
        return index + 1
    def __class_getitem__(cls, item):
        return f"{cls.__name__}[{item.__name__}]"

class MyOtherList(MyList):
    pass

assert MyList()[0] == 1
assert MyList[int] == "MyList[int]"

assert MyOtherList()[0] == 1
assert MyOtherList[int] == "MyOtherList[int]"
```

类型标注（PEP 593）

- 对标注类型进一步增加额外负数信息，方便静态检查或运行时做更进一步的检查。
- 主要针对运行时扩展机制。

- 例如：分数（值从1-100的整数类型）

```
Score = Annotated[int, ValueRange(1, 100)]
value: Score = 30
```

- 例如：包含最多10个元素的元组列表

```
Typevar T = TypeVar('T')
Vec = Annotated[List[Tuple[T, T]], MaxLen(10)]
V = Vec[int]
# 等价于
V == Annotated[List[Tuple[int, int]], MaxLen(10)]
```

工具参考

- typeshed: Python内置标准和三方库的pyi集合repo（PyCharm、mypy、pytype已包含）
- mypy: 官方标准静态类型检查工具。
- pyre：Facebook开源的静态类型检查工具。
- pytype：google开源的Python静态代码扫描工具（不依赖标注）。
- pyannotate：dropbox开源的自动给Python添加类型标注的工具。
- pydantic：一个基于标注的Python数据校验与配置管理库。

什么时候需要或不需要类型检查?

- 以下建议采用:
  - 提供SDK、库/接口给其他人时。
    - 比docstring更清晰、主流IDE支持提示和校验。
  - 代码行数越多，价值越大。
    - 规范化编码，通过工具可以辅助发现潜在BUG。
  - 需要写UT（单元测试）的地方，就需要类型检查 (by Bernat Gabor)
- 以下需要一定策略:
  - 原型（Prototype）或验证性质项目（POC）的代码，可以先不引入。
  - 大量旧有代码，需要逐步阶段性引入（参考Dropbox经验）。
  - 不熟悉Python和类型提供功能用法时，可以先不引入。

PyCon China 2020

Python For Good

PyConChina 2020 | PYTHON 中国开发者大会 2020

THANK YOU

本人微信

Github下载PPT

日志服务钉钉群