

# PyCon China 2024

For Good . For fun.  
2024/11/23 中国 上海





# NoneType

会是类型系统的奇点吗？

王宏府

星商创新技术有限公司

Python 文档简中翻译组成员

PSF Managing/Contributing Member



# None

```
In [1]: n1, n2 = None, None
```

```
In [2]: id(n1) == id(n2)
```

```
Out[2]: True
```

```
In [3]: def f():  
...:     ...  
...:
```

```
In [4]: print(f())
```

```
None
```

```
In [5]: bool(None)
```

```
Out[5]: False
```


## Null Pointer References: The Billion Dollar Mistake

*I call it my billion-dollar mistake...At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler.*

*But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

*– Tony Hoare, inventor of ALGOL W.*

```
from types import NoneType
```

```
usr > lib > python3.12 >  types.py > ...
```

```
326 GenericAlias = type(list[int])
```

```
327 UnionType = type(int | str)
```

```
328
```

```
329 EllipsisType = type(Ellipsis)
```

```
330 NoneType = type(None)
```

```
331 NotImplementedType = type(NotImplemented)
```



有意隱藏 `NoneType`

OR

受限於歷史原因



## Contents

- Abstract
- Rationale and Goals
  - Non-goals
- The meaning of annotations
- Type Definition Syntax
  - Acceptable type hints
  - Using None
  - Type aliases
  - Callable
  - Generics
  - User-defined generic types
  - Scoping rules for type variables

## PEP 484 – Type Hints

**Author:** Guido van Rossum <guido at python.org>, Jukka Lehtosalo <jukka.lehtosalo at iki.fi>, Łukasz Langa <lukasz at python.org>

**BDFL-Delegate:** Mark Shannon

**Discussions-To:** [Python-Dev list](#)

**Status:** Final

**Type:** [Standards Track](#)

**Topic:** [Typing](#)

**Created:** 29-Sep-2014

**Python-Version:** 3.5

**Post-History:** 16-Jan-2015, 20-Mar-2015, 17-Apr-2015, 20-May-2015, 22-May-2015

**Resolution:** [Python-Dev message](#)

## Using None

When used in a type hint, the expression `None` is considered equivalent to `type(None)`.



# NoneType

## 到底有什么问题？



# PEP 483 – The Theory of Type Hints

**Author:** Guido van Rossum <guido at python.org>, Ivan Levkivskyi <levkivskyi at gmail.com>

**Discussions-To:** [Python-Ideas list](#)

**Status:** Final

**Type:** Informational

**Topic:** [Typing](#)

**Created:** 19-Dec-2014

**Post-History:**

## Subtype relationships

A crucial notion for static type checker is the subtype relationship. It arises from the question: If `first_var` has type `first_type`, and `second_var` has type `second_type`, is it safe to assign `first_var = second_var`?

A strong criterion for when it *should* be safe is:

- every value from `second_type` is also in the set of values of `first_type`; and
- every function from `first_type` is also in the set of functions of `second_type`.

The relation defined thus is called a subtype relation.

By this definition:

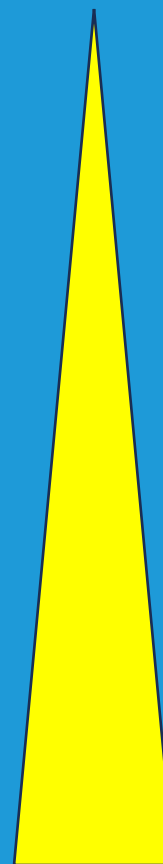
- Every type is a subtype of itself.
- The set of values becomes smaller in the process of subtyping, while the set of functions becomes larger.

Class Animal  
父类型

值集

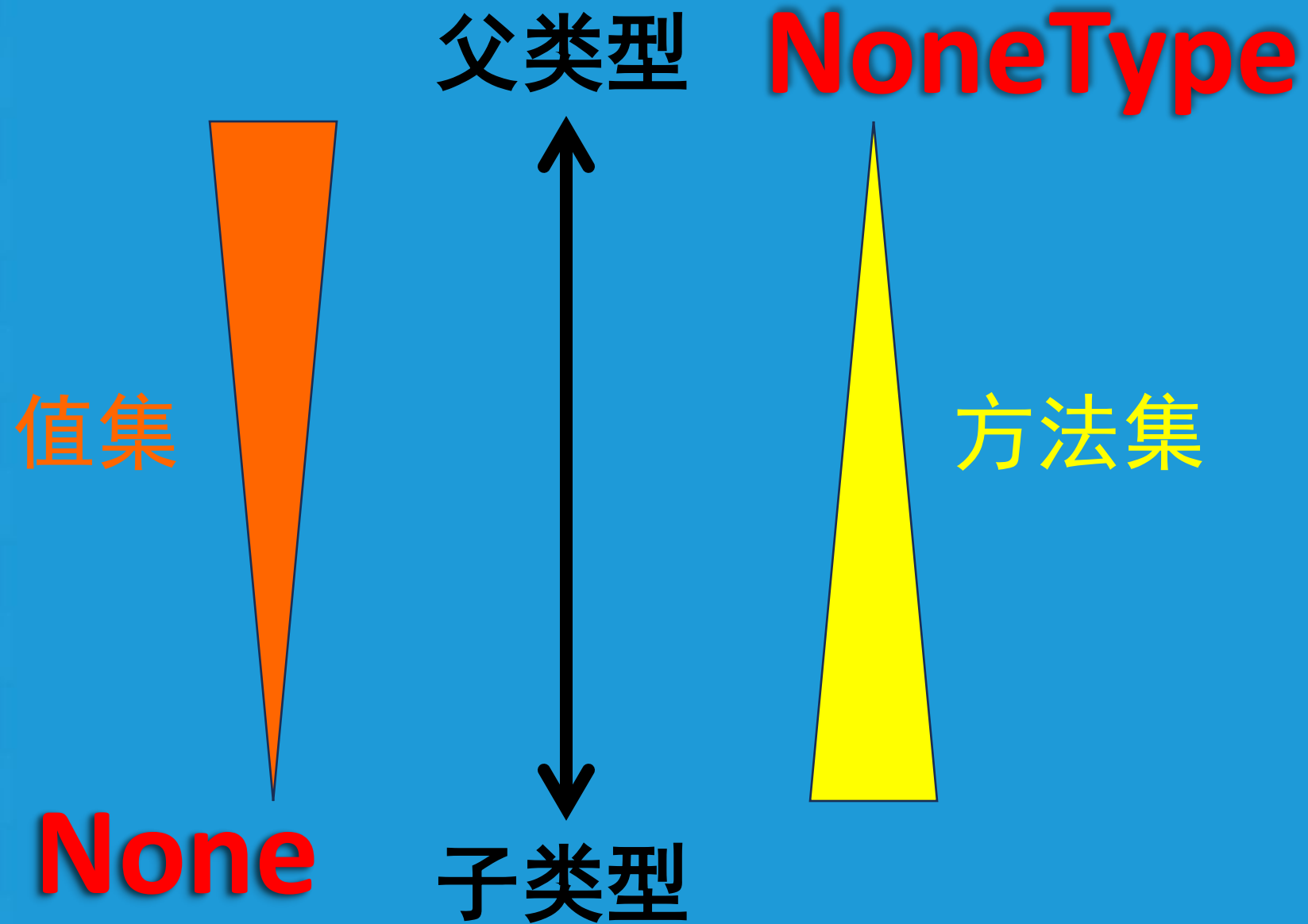


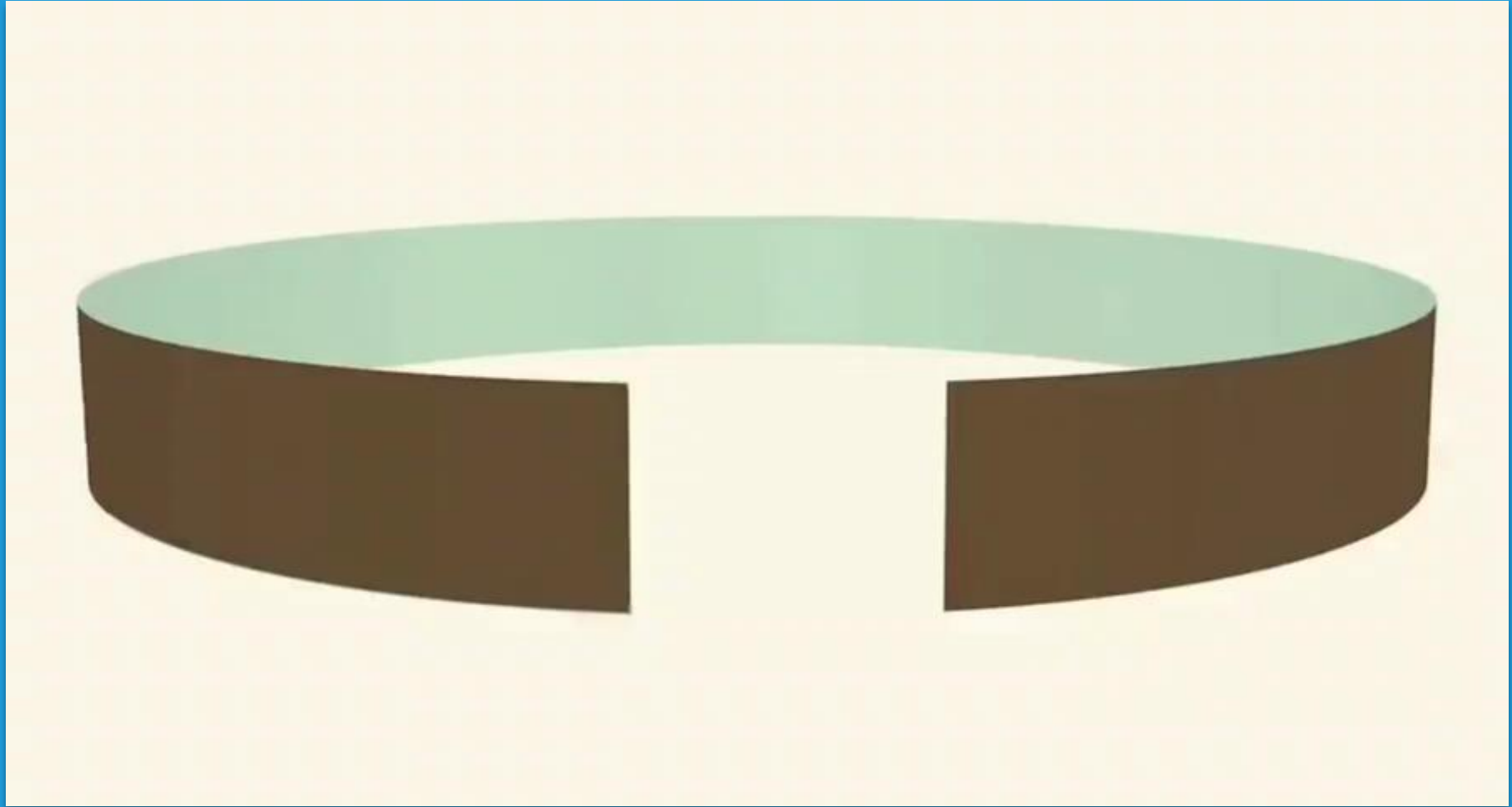
方法集



子类型

class Cat(Animal)







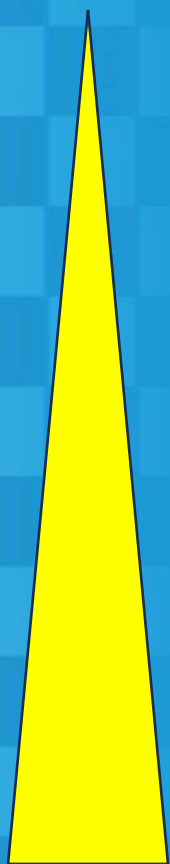
**其他语言就没问题？**

各编程语言中 NULL 特性的比较表

语言	NULL 的形式	关键字/常量	底层具体值	类型	流程控制中的使用	支持的方法示例	便利性	伴生的问题	与其他语言的区别
Python	<code>None</code>	内置常量	单例对象	<code>NoneType</code>	使用 <code>is</code> 或 <code>is not</code> 判断是否为 <code>None</code>	无专属方法, 可使用 <code>is</code> , <code>is None</code> 等	统一表示空值, 避免可变默认参数陷阱	易与假值混淆, 需要注意布尔上下文中的使用	<code>None</code> 是单例, 使用 <code>is</code> 比较
C	<code>NULL</code>	预处理器宏	通常为 <code>((void *)0)</code> 或 <code>0</code>	指针类型 ( <code>void *</code> ) 或 <code>int</code>	用于指针与 <code>NULL</code> 比较, 检查指针是否为空	无方法, 常用于指针初始化和检查	统一表示空指针, 避免野指针	类型不安全, 可能导致空指针解引用错误	<code>NULL</code> 是宏, 可能与整数 <code>0</code> 混淆
C++	<code>NULL</code> / <code>nullptr</code>	<code>NULL</code> 是宏, <code>nullptr</code> 是关键字	<code>NULL</code> 为 <code>0</code> , <code>nullptr</code> 为空指针常量	<code>NULL</code> 为 <code>int</code> , <code>nullptr</code> 为 <code>std::nullptr_t</code>	使用 <code>nullptr</code> 进行指针比较, 避免与整数混淆	无方法, <code>nullptr</code> 可安全转换为任意指针类型	<code>nullptr</code> 提供类型安全的空指针, 解决函数重载歧义	旧代码混用 <code>NULL</code> 和 <code>nullptr</code> 可能导致混淆	<code>nullptr</code> 是类型安全的空指针, 消除重载歧义
Java	<code>null</code>	字面量, 保留字	对象引用的默认值, 不指向任何对象	无类型, 可赋值给任何引用类型	检查引用是否为 <code>null</code> , 避免 <code>NullPointerException</code>	无方法, 常与 <code>Objects</code> 工具类一起使用	统一表示空引用, 避免垃圾值	<code>NullPointerException</code> , 需频繁进行空值检查	无基本类型的 <code>null</code> , 缺乏语言层面的空安全机制
C#	<code>null</code>	关键字	引用类型的默认值, 不指向任何对象	无类型, 可赋值给引用类型和可空值类型 ( <code>Nullable&lt;T&gt;</code> )	检查引用是否为 <code>null</code> , 利用 <code>??</code> 、 <code>?.</code> 等空值处理运算符	<code>??</code> 、 <code>?.</code> 、 <code>Nullable&lt;T&gt;</code> 等	空值处理运算符简化代码, 可空值类型使值类型可为 <code>null</code>	空引用异常, 需注意可空值类型的拆箱问题	丰富的空值处理运算符, 值类型可为 <code>null</code>
JavaScript	<code>null</code> , <code>undefined</code>	字面量	<code>null</code> 表示空对象引用, <code>undefined</code> 表示未定义	<code>null</code> 类型为 <code>object</code> (历史问题), <code>undefined</code> 类型为 <code>undefined</code>	使用严格比较 <code>===</code> 检查是否为 <code>null</code> 或 <code>undefined</code>	无方法, 可使用可选链 <code>?.</code> 进行安全访问	表示空值和未定义的值, 可选链和空值合并运算符简化空值处理	<code>typeof null</code> 返回 "object", 与 <code>undefined</code> 混淆	同时存在 <code>null</code> 和 <code>undefined</code> , 需特别注意区别
Kotlin	<code>null</code>	关键字	与 Java 的 <code>null</code> 对应	可空类型 ( <code>Type?</code> ) 或非空类型 ( <code>Type</code> )	编译期强制空值检查, 使用 <code>?.</code> 、 <code>?:</code> 、 <code>!!</code> 等操作符处理可空值	<code>?.</code> 、 <code>?:</code> 、 <code>!!</code> 、 <code>let</code> 等	编译期空安全, 消除空指针异常, 简洁的空值处理语法	滥用 <code>!!</code> 可能导致异常, 与 Java 互操作可能存在空安全问题	类型系统内置空安全, 可空类型与非空类型区分严格
Go	<code>nil</code>	预定义标识符	类型的零值, 不同类型的 <code>nil</code> 不同	无类型, 可赋值给指针、切片、映射、通道等	判断变量是否为 <code>nil</code> , 控制程序流程	无方法, 但方法接收者可为 <code>nil</code> , 需处理 <code>nil</code> 情况	统一零值表示, 简化初始化和检查, 节省内存	空指针解引用导致 panic, 接口的 <code>nil</code> 判断陷阱	<code>nil</code> 可用于多种类型, 方法接收者可为 <code>nil</code>
Rust	<code>None</code> (无 <code>null</code> )	<code>Option</code> 枚举的变体	<code>Option&lt;T&gt;</code> 的 <code>None</code> 变体	<code>Option&lt;T&gt;</code> 枚举类型	使用 <code>match</code> 、 <code>if let</code> 等模式匹配 <code>Option</code> 类型	<code>is_some</code> 、 <code>unwrap_or</code> 、 <code>map</code> 等方法	编译期空值检查, 消除空指针异常, 类型安全	滥用 <code>unwrap</code> 可能导致 panic, 需要显式处理空值	无传统的 <code>null</code> , 通过类型系统避免空指针, <code>Option</code> 类型显式表示可空值
Ruby	<code>nil</code>	对象	<code>nil</code> 是 <code>NilClass</code> 的唯一实例	<code>NilClass</code>	<code>nil</code> 在条件判断中视为假, 判断变量是否为 <code>nil</code>	基础方法, 如 <code>nil?</code> , 可与其他对象方法一起使用	统一表示空值, <code>nil</code> 也是对象, 可直接调用方法	<code>nil</code> 与 <code>false</code> 都在条件判断中为假, 需注意逻辑区分	<code>nil</code> 是对象, 且在条件判断中为假, 仅有 <code>false</code> 和 <code>nil</code> 为假
Swift	<code>nil</code>	关键字	可选类型 <code>Optional</code> 的无值	<code>Optional&lt;T&gt;</code> 或 <code>T?</code>	使用可选绑定、强制解包等处理可选值	<code>map</code> 、 <code>flatMap</code> 、 <code>unwrap</code> 等方法	强类型空值检查, 避免空指针异常, 提高安全性	强制解包可能导致崩溃, 需要谨慎处理	可选类型内置于语言, 非可选类型不能为 <code>nil</code>



方法集



NULL



`((void *)0)`

`void *`

允许隐式转换到  
任何（所有）  
其他对象指针类型

->

`(*ptr).`

可以通过指针函数调用



# NullPointerException



?.

```
val nullPerson: Person? = null  
val cityName3 = nullPerson?.company?.address?.city  
println("City: $cityName3") // 输出: City: null
```





Tony Hoare's presentation in 2009 at QCon:

- Null references have historically been a bad idea

# Common Sentinel

```
fn divide(numerator: f64, denominator: f64) -> Option<f64> {  
    if denominator == 0.0 {  
        None  
    } else {  
        Some(numerator / denominator)  
    }  
}  
  
// The return value of the function is an option  
let result = divide(2.0, 3.0);  
  
// Pattern match to retrieve the value  
match result {  
    // The division was valid  
    Some(x) => println!("Result: {x}"),  
    // The division was invalid  
    None    => println!("Cannot divide by 0"),  
}
```

method	self	function input	function result	output
<code>and_then</code>	<code>None</code>	(not provided)	(not evaluated)	<code>None</code>
<code>and_then</code>	<code>Some(x)</code>	<code>x</code>	<code>None</code>	<code>None</code>
<code>and_then</code>	<code>Some(x)</code>	<code>x</code>	<code>Some(y)</code>	<code>Some(y)</code>



**None**  
**不会打断**  
**链式调用**

You, 上周 | 1 author (You)

```
class ValidWarehouse(BaseModel):  
    warehouseDisable: bool | None = None  
    warehouseId: str | None = None  
    warehouseName: str | None = None
```

You, 上周 | 1 author (You)

```
class WarehouseDTO(BaseModel):  
    validWarehouseList: list[ValidWarehouse] | None = None  
    siteId: int | None = None  
    siteName: str | None = None
```

You, 上周 | 1 author (You)

```
class WarehouseDTOList(BaseModel):  
    warehouseDTOList: list[WarehouseDTO] | None = None
```

You, 上周 | 1 author (You)

```
class PullLocationResponse(BaseModel):  
    success: bool  
    requestId: str | None = None  
    errorCode: int | None = None  
    errorMsg: str | None = None  
    result: WarehouseDTOList | None = None
```





```
if resp.success:
    assert resp.result is
    if warehouse_dto_list
        for warehouse_dto
            assert wareho

    for valid_war
        assert va
```



**理论：**

**NoneType**

**是当前渐进式类型系统中的奇点**

**实践：**

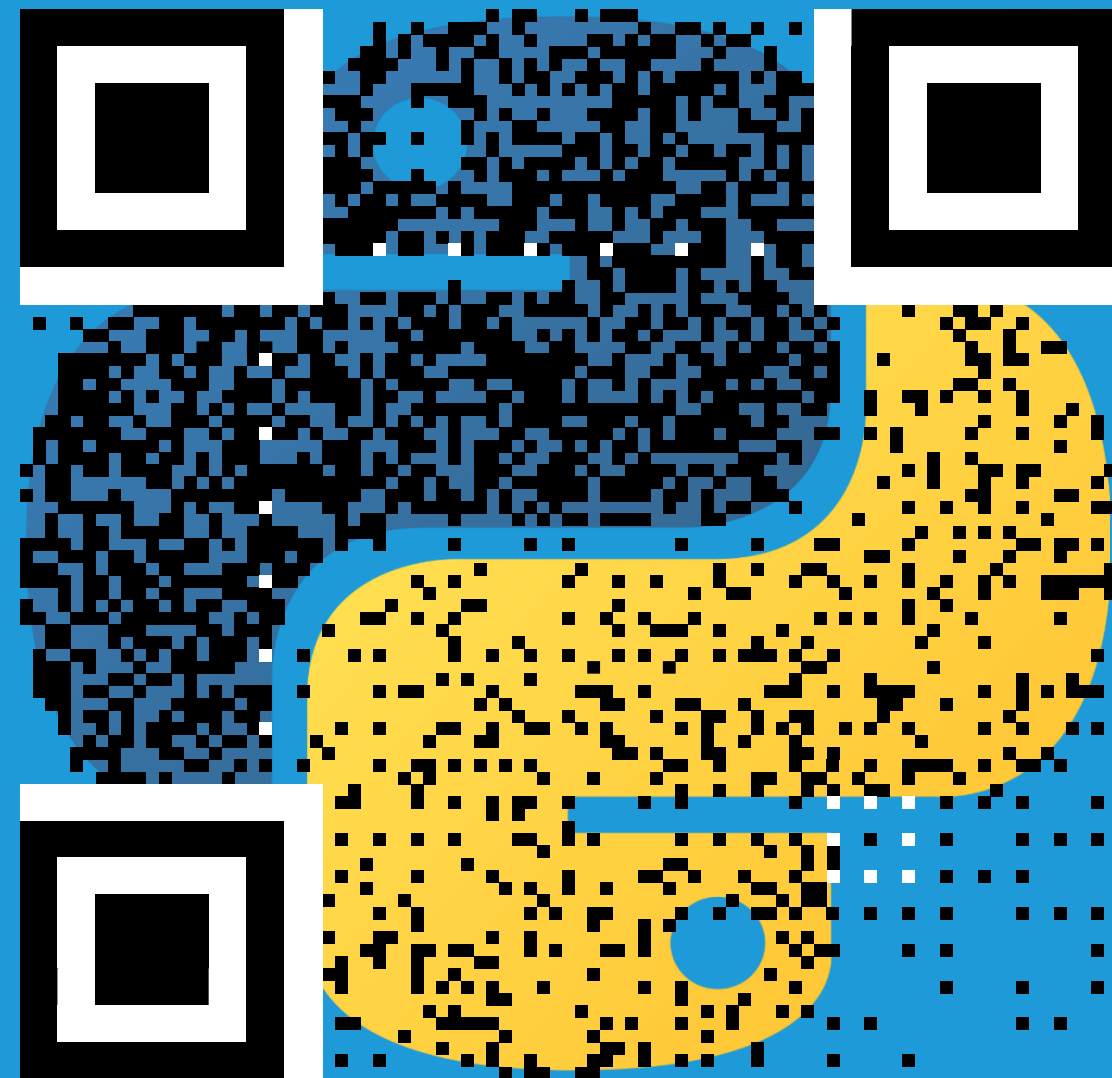
**我们想要一个可链式调用的 None  
以简化嵌套空值的判断**

# Thanks

&

See you at

Future



*[blog.wh2099.com](http://blog.wh2099.com)*

