AI VIET NAM
@aivietnam.edu.vn

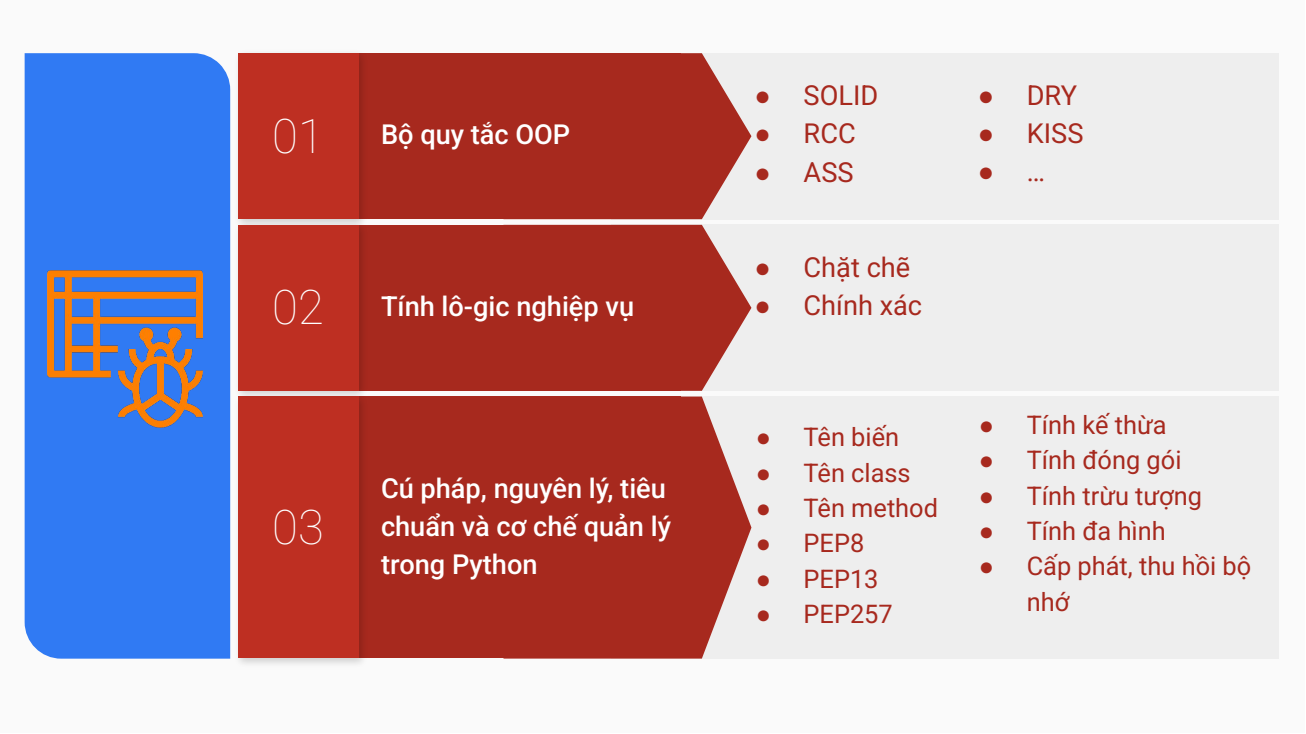
AI VIET NAM – AI COURSE 2025

Các lỗi thường gặp trong Lập trình hướng đối tượng dùng Python

Đinh Trần Minh Hiếu

I. Giới thiệu

Lập trình hướng đối tượng (OOP) là phương pháp lập trình trong đó các chương trình được xây dựng dựa trên các đối tượng thay vì các hàm và thủ tục. Mục tiêu của OOP là giúp cho mã nguồn được tổ chức một cách có quy tắc, dễ bảo trì và mở rộng.



Hình 1: Minh họa nguồn phát sinh lỗi khi thực hiện OOP với Python.

Để giúp lập trình viên xây dựng được bộ mã nguồn theo phương pháp lập trình này, OOP có một số tập các quy tắc nên được tuân thủ, chẳng hạn như: bộ quy tắc thiết kế Class (lớp) - SOLID, bộ quy tắc kết nối các thành phần trong mô-đun - RCC, bộ quy tắc về mức độ độc lập

giữa các mô-đun - ASS, ngoài ra còn có quy tắc tránh lặp lại mã nguồn - DRY, quy tắc giữ mã nguồn đơn giản - KISS, và nhiều hơn nữa.
Ngoài ra, khi thực hiện **OOP trong Python** còn cần tuân thủ theo các cú pháp, nguyên tắc lập trình trong Python, chẳng hạn như nguyên tắc về tính kế thừa, tính đóng gói, tính đa hình, và tính trừu tượng. Thêm vào đó là các cơ chế quản lý như cấp phát và thu hồi bộ nhớ, các tiêu chuẩn khi lập trình với Python như PEP8 hay PEP13. Một yếu tố quan trọng khác là tính logic nghiệp vụ khi thực hiện xây dựng bộ mã nguồn.

Với nhiều quy tắc và cú pháp, lập trình viên khó có thể viết mã suôn sẻ mà không gặp lỗi khi áp dụng OOP với Python. Tất cả cần trải qua quá trình luyện tập và học hỏi một cách nghiêm túc để đạt được sự nhuần nguyễn với phương pháp lập trình này. Tuy nhiên cần hạn chế những lỗi nghiêm trọng gây ảnh hưởng lớn đến mã nguồn.

Bài viết này sẽ đồng hành cùng bạn về các lỗi phổ biến khi lập trình hướng đối tượng với Python và các giải pháp để giảm thiểu các lỗi này.

Mục lục

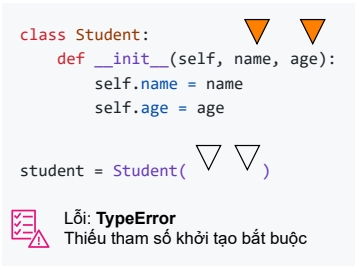
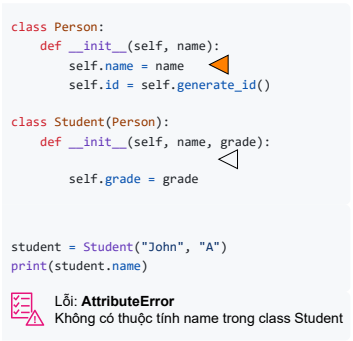
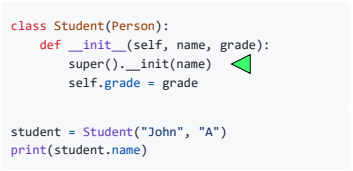
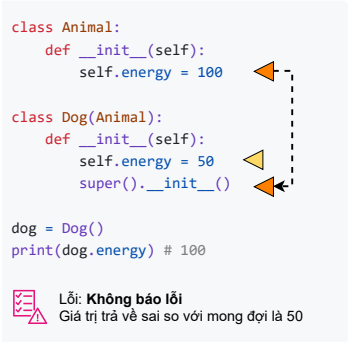
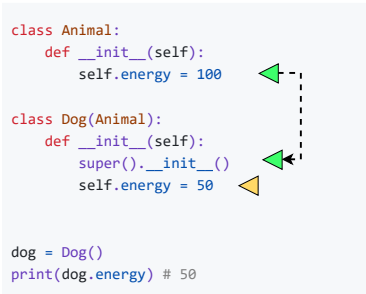
II. Các lỗi thường gặp trong OOP

Trong phần này, chúng ta sẽ xem xét các lỗi thường gặp theo các góc độ khác nhau khi lập trình hướng đối tượng với Python.

II.1. Nhóm lỗi nghiêm trọng

Các lỗi trong nhóm này làm cho chương trình không thể thực thi được. Bắt buộc phải thực hiện việc sửa lỗi thì chương trình mới chạy được.

II.1.1 Nhóm lỗi về khai báo và khởi tạo instance của lớp

Nhóm lỗi	X	✓
1. Thiếu tham số bắt buộc khi khởi tạo		
2. Quên gọi hàm `super().__init__()` khi kế thừa từ lớp khác		
3. Sai thứ tự gọi `super().__init__()` khi kế thừa từ lớp khác		

II.1.2. Nhóm lỗi về truy xuất thuộc tính của instance

Nhóm lỗi	X	✓

1. Lỗi truy xuất biến ở phạm vi private (*private variables*)	`class BankAccount:`
	`  def __init__(self):`
	`  self.__balance = 1000`
	`  def show_balance(self):`
	`  print(__balance)`
	`account = BankAccount()`
	`account.show_balance()`
	Lỗi: **NameError**
	Biến __balance không xác định
	`account = BankAccount()`
	`print(account.__balance)`
	Lỗi: **AttributeError**
	Biến (self.)__balance được khai báo ở phạm vi Private, không thể truy xuất ở ngoài class
		`class BankAccount:`
		`  def __init__(self):`
		`  self.__balance = 1000`
		`  def show_balance(self):`
		`  print(self.__balance)`
		`  @property`
		`  def balance(self):`
		`  return self.__balance`
		`account = BankAccount()`
		`account.show_balance() # 1000`
		`print(account.balance) # 1000`

2. Lỗi không sử dụng `self` khi truy xuất biến ở mức instance	`class Calculator:`
	`  def __init__(self):`
	`  self.result = 0`
	`  def add(self, value):`
	`  result = value + result`
	`  return result`
	`calculator = Calculator()`
	`calculator.add(5)`
	Lỗi: **NameError**
	Biến *result* không xác định, chưa được khai báo,
		`class Calculator:`
		`  def __init__(self):`
		`  self.result = 0`
		`  def add(self, value):`
		`  self.result += value`
		`  return self.result`
		`calculator = Calculator()`
		`print(calculator.add(5)) # 5`

3. Nhầm lẫn giữa biến ở mức class và biến ở mức instance	`class Counter:`
	`  count = 0` mức class
	`  def __init__(self):`
	`  count = 0` mức cục bộ
	`  def increment(self):`
	`  self.count += 1` mức instance
	`c1 = Counter()`
	`c2 = Counter()`
	`c1.increment()`
	`print(Counter.count) # 0`
	Lỗi: **Không báo lỗi**
	Sai kết quả mong đợi khi cần cập nhật thông tin về biến *count* ở mức class
		`class Counter:`
		`  count = 0` mức class
		`  def __init__(self):`
		`  count = 0` mức cục bộ
		`  def increment(self):`
		`  Counter.count += 1`
		`c1 = Counter()`
		`c2 = Counter()`
		`c1.increment()`
		`print(Counter.count) # 1`

II.1.3. Nhóm lỗi khai báo sai hàm (phương thức)

Nhóm lỗi	✗	✓

1. Lỗi quên đặt tham số `self` khi khai báo hàm	`class Circle:`
	`  def __init__(self, radius):`
	`  self.radius = radius`
	`  def area( ):`
	`  return 3.14 * self.radius ** 2`
	`circle = Circle(5)`
	`circle.area()`
	Lỗi: **TypeError**
	Circle.area() được định nghĩa không có tham biến, nhưng có 1 tham biến được truyền vào
		`class Circle:`
		`  def __init__(self, radius):`
		`  self.radius = radius`
		`  def area(self):`
		`  return 3.14 * self.radius ** 2`
		`circle = Circle(5)`
		`circle.area() # 78.5`

2. Dùng tên tham số khác thay vì `self` khi khai báo hàm	`class Student:`
	`  def __init__(obj, name):`
	`  obj.name = name`
	`  def study(this):`
	`  print(f"{this.name} is studying")`
	`student = Student("John Doe")`
	`student.study()`
	Lỗi: **Không báo lỗi**
	Tên tham biến đầu tiên của phương thức instance dễ gây nhầm lẫn khi đọc code
		`class Student:`
		`  def __init__(self, name):`
		`  self.name = name`
		`  def study(self):`
		`  print(f"{self.name} is studying")`
		`student = Student("John Doe")`
		`student.study()`

II.1.4. Nhóm lỗi về kế thừa đa lớp

Nhóm lỗi này thường liên quan đến cách hiểu về **"Thứ tự gọi phương thức (MRO - Method Resolution Order)"** khi thực hiện OOP với Python để vận dụng một cách chính xác. *(Xem thêm bài viết gốc về MRO ở phần tham khảo).*

Khi thực hiện đa kế thừa, Python sẽ quyết định phương thức hay thuộc tính nào được gọi bằng cách **"xây dựng đồ thị thứ tự"** gọi đến các lớp.

Trong phần này sẽ tiếp cận theo góc nhìn khác so với bài viết tham khảo gốc về `MRO` với mục đích đơn giản hoá về đa kế thừa mà vẫn đảm bảo độ chính xác trong cách vận dụng bằng cách sử phương pháp: cấp độ kế thừa của một lớp và phương thức có sẵn `<class>.mro()`.

Hãy cùng tìm hiểu về `MRO` qua các ví dụ sau:

Ví dụ 1: Cơ bản về thứ tự lớp kế thừa

```python
class A:
    def speak(self):
        print("speak A") # <<==

class B(A):
    def speak(self):
        print("speak B")
        super().speak() # <<==

class C:
    def speak(self):
        print("speak C") # <<--

class D(C):
    def speak(self):
        print("speak D")
        super().speak() # <<--


def cls_inheritance_order(cls):
    print(f"MRO of {cls.__name__}: ")
    for i, cls_name in enumerate(cls.mro()):
        print(f"Cấp: {len(cls.mro()) - i - 1}:\t{cls_name}")


cls_inheritance_order(B)
print("==========B Speak==========")
B().speak()
print("--------------------")
cls_inheritance_order(D)
print("==========D Speak==========")
D().speak()

### Kết quả:
MRO of B:
Cấp: 2: <class '__main__.B'>
Cấp: 1: <class '__main__.A'>
Cấp: 0: <class 'object'>
==========B Speak==========
speak B
speak A
--------------------
MRO of D:
Cấp: 2: <class '__main__.D'>
Cấp: 1: <class '__main__.C'>
Cấp: 0: <class 'object'>
==========D Speak==========
speak D
speak C
```

Trong Python, các đối tượng nói chung đều kế thừa từ **lớp object** nên thứ tự kế thừa ở cấp độ

gốc (cấp 0) luôn là *lớp object*

Ví dụ 2: Thứ tự gọi phương thức với super()
Trong ví dụ này, không phải lớp nào cũng dùng `super().speak()` để gọi phương thức ở lớp kế thừa. Hãy chú ý phương thức *speak* sẽ được gọi đến lớp kế thừa ở cấp nào?

```python
class A:
    def speak(self):
        print("speak A")

class B(A):
    def speak(self):
        print("speak B")
        super().speak()

class C(A):
    def speak(self):
        print("speak C")

class D(B, C):
    def speak(self):
        print("speak D")
        super().speak()

cls_inheritance_order(D)
print("========D Speak========")
D().speak()

### Kết quả:

MRO of D:
Cấp: 4: <class '__main__.D'>
Cấp: 3: <class '__main__.B'>
Cấp: 2: <class '__main__.C'>
Cấp: 1: <class '__main__.A'>
Cấp: 0: <class 'object'>
========D Speak========
speak D
speak B
speak C
```

Cần chú ý các điểm sau:

- Lớp D kế thừa **theo thứ tự** lớp B, đến lớp C.
- Phương thức *speak* ở lớp D, lớp B có gọi hàm kế thừa `super().speak()`, còn lớp C thì không.

Như vậy, khi gọi phương thức *speak* từ lớp D, theo thứ tự **MRO** của lớp D, phương thức *speak* sẽ trả về kết quả dừng lại ở lớp C (do không gọi phương thức kế thừa ở đây), và *không tiếp tục* đến lớp A, mặc dù cả lớp B và C đều kế thừa đến lớp A.

Ví dụ 3: Lỗi kế thừa đến cùng một lớp cha cao nhất mà không phải là lớp *object*
Hãy chú ý đến thứ tự và cấp độ của các lớp được kế thừa.

```
class A:
    pass

class B(A):
    pass

class C(A, B):
    pass

### Kết quả LỖI
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B

### Phần thông tin thêm
MRO of A:
Cấp: 1: <class '__main__.A'>
Cấp: 0: <class 'object'>
--------------------
MRO of B:
Cấp: 2: <class '__main__.B'>
Cấp: 1: <class '__main__.A'>
Cấp: 0: <class 'object'>
```

Cần chú ý các điểm sau:

- Các lớp đều kế thừa đến lớp A (lớp cha cao nhất).
- So với lớp *object*, lớp A ở cấp 1, lớp B ở cấp 2. lớp A ở cấp *"nông hơn"* lớp B.
- Lớp C kế thừa theo thứ tự lớp A (cấp 1), lớp B (cấp 2).

Nhận xét: Lỗi xảy ra khi các lớp đều kế thừa về lớp cha cao nhất là lớp A (không tính lớp object) và khi khai báo lớp C kế thừa nhiều lớp: lớp A có cấp độ *"nông hơn"* so với lớp kế thừa tiếp theo, lớp B.

Như vậy, thứ tự kế thừa đúng nên là lớp có **cấp độ "sâu hơn"hoặc bằng** lớp kết thừa tiếp theo mà **cùng kết thừa đến một lớp cha cao nhất** (không tính *lớp object*).

```
class A:
    pass

class B(A):
    pass

class C(B, A):
    pass
```

II.2. Nhóm lỗi về logic

Nhóm lỗi này thường vẫn làm cho chương trình chạy được. Tuy nhiên, có sai xót trong việc triển khai logic nghiệp vụ làm cho kết quả của đoạn code hoặc chương trình không được như mong đợi. Tuy nhiên lại tác động lớn về tính khả thi của chương trình và có thể gây thiệt hại về mặt kinh tế.

II.2.1 Nhóm lỗi về giá trị mặc định của tham biến trong phương thức là kiểu dữ liệu thay đổi được (Mutable)

Cùng tìm hiểu nhanh về kiểu dữ liệu **Mutable** và **Immutable** tích hợp sẵn trong Python:

- Kiểu dữ liệu **Mutable** là kiểu dữ liệu có thể thay đổi được *trạng thái nội tại* sau khi được tạo ra. Bao gồm các kiểu dữ liệu: **List, Dictionary, Set**.
- Kiểu dữ liệu **Immutable** là kiểu dữ liệu không thể thay đổi được *trạng thái nội tại* sau khi được tạo ra. Bao gồm các kiểu dữ liệu: **Numbers, Booleans, Strings, Bytes, Tuples**.

Nổi bật trong nhóm này là **lỗi sử dụng tham biến với giá trị mặc định là kiểu dữ liệu Mutable trong hàm khởi tạo của class.**

```python
# Ví dụ 1: Tham biến mặc định là List
class PersonList:
    def __init__(self, name, hobbies=[]):
        self.name = name
        self.hobbies = hobbies

person1 = PersonList("John")
person2 = PersonList("Jane")
person1.hobbies.append("music")
print(person2.hobbies)

### Kết quả của person2: ['music']

# Ví dụ 2: Tham biến mặc định là Dictionary
class PersonDict:
    def __init__(self, name, hobbies={}):
        self.name = name
        self.hobbies = hobbies

person1 = PersonDict("John")
person2 = PersonDict("Jane")
person1.hobbies["music"] = "Xin chào Việt Nam"
print(person2.hobbies)

### Kết quả của person2: ['music']
{'music': 'Xin chào Việt Nam'}

# Ví dụ 3: Tham biến mặc định là Set
class PersonSet:
    def __init__(self, name, hobbies=set()):
        self.name = name
        self.hobbies = hobbies
```

```
person1 = PersonSet("John")
person2 = PersonSet("Jane")
person1.hobbies.add("music")
print(person2.hobbies)

### Kết quả của person2: {'music'}
```

Nhận xét: Cả ba trường hợp ví dụ đều có chung đặc điểm sau:

- Tham biến *hobbies* được đặt giá trị mặc định là kiểu dữ liệu *Mutable*.
- Chỉ thực hiện thay đổi giá trị "*hobbies*"của *person1*.

Kết quả: Khi in ra giá trị *hobbies* của *person2*, mặc dù chưa thực hiện thay đổi, nhưng đã tham chiếu sang giá trị *hobbies* của *person1*. Lưu ý: Lỗi này **không báo lỗi khi chạy chương trình**.

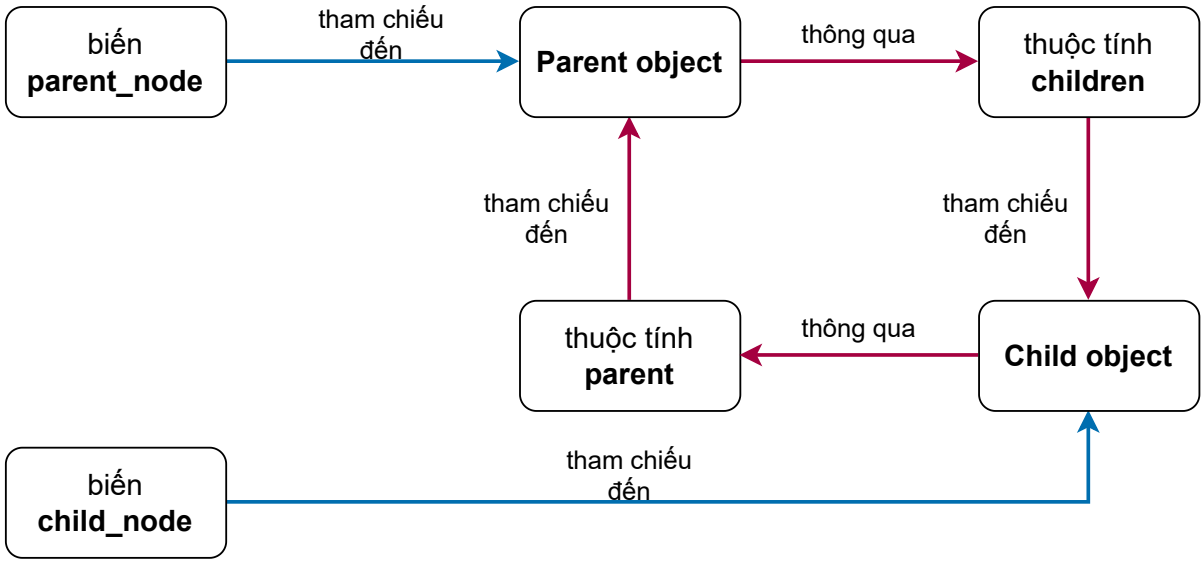
II.2.2 Vấn đề về Cấp phát bộ nhớ và Tham chiếu

II.2.2.1 Vấn đề Tham chiếu vòng tròn (Circular References)

```python
import ctypes
import gc

def get_ref_count(address):
    """Get the reference count of an object at a given memory address."""
    return ctypes.c_long.from_address(address).value


class Parent:
    def __init__(self):
        self.children = []

    def add_child(self, child):
        self.children.append(child)

        child.parent = self  # <<=== Điểm gây ra Circular reference
        print(f"Parent: {hex(id(self))}, Child: {hex(id(child))}")

    def __del__(self):
        print(f"Thu hồi Parent với id: {hex(id(self))}")

class Child:
    def __init__(self):
        self.parent = None
        print(f"Child: {hex(id(self))}")

    def get_parent(self):
        print(f"Parent from Child: {hex(id(self.parent))}")

    def __del__(self):
        print(f"Thu hồi Child với id: {hex(id(self))}")

```

```python
print("1. Tạo Circular Reference:")
parent_node = Parent()
child_node = Child()
parent_node.add_child(child_node)
child_node.get_parent()

id_parent_node = id(parent_node) # Lấy id vùng nhớ của biến parent_node
id_child_node = id(child_node) # Lấy id vùng nhớ của biến child_node

print(f"\n2. Đếm số lượng tham chiếu đến vùng nhớ:")
print(f"Số lượng tham chiếu đến vùng nhớ của parent_node: {get_ref_count(id_parent_node)}")

print(f"Số lượng tham chiếu đến vùng nhớ của child_node: {get_ref_count(id_child_node)}")

print(f"\n3. Thay đổi vùng nhớ của biến parent_node = None:")
parent_node = None

print(f"Số lượng tham chiếu đến vùng nhớ của parent_node: {get_ref_count(id_parent_node)}")

print(f"Số lượng tham chiếu đến vùng nhớ của child_node: {get_ref_count(id_child_node)}")

print("\n4. Thực hiện Thu hồi bộ nhớ")
gc.collect()

print(f"Số lượng tham chiếu đến vùng nhớ của parent_node: {get_ref_count(id_parent_node)}")

print(f"Số lượng tham chiếu đến vùng nhớ của child_node: {get_ref_count(id_child_node)}")

# Kết quả:
1. Tạo Circular Reference:
Child: 0x1029d3b90
Parent: 0x1029d3b60, Child: 0x1029d3b90
Parent from Child: 0x1029d3b60

2. Đếm số lượng tham chiếu đến vùng nhớ:
Số lượng tham chiếu đến vùng nhớ của parent_node: 2
Số lượng tham chiếu đến vùng nhớ của child_node: 2

3. Thay đổi vùng nhớ của biến parent_node = None:
Số lượng tham chiếu đến vùng nhớ của parent_node: 1
Số lượng tham chiếu đến vùng nhớ của child_node: 2

4. Thực hiện Thu hồi bộ nhớ
Số lượng tham chiếu đến vùng nhớ của parent_node: 1
Số lượng tham chiếu đến vùng nhớ của child_node: 2

5. Thay đổi vùng nhớ của biến child_node = None:
Số lượng tham chiếu đến vùng nhớ của parent_node: 1
```

```
Số lượng tham chiếu đến vùng nhớ của child_node: 1

6. Thực hiện Thu hồi bộ nhớ
Thu hồi Child với id: 0x1029d3b90
Thu hồi Parent với id: 0x1029d3b60
Số lượng tham chiếu đến vùng nhớ của parent_node: 0
Số lượng tham chiếu đến vùng nhớ của child_node: 0
```



Hình 2: Minh họa tham chiếu vòng tròn cho đoạn code trên

Đoạn code trên mô tả cho vấn đề tham chiếu vòng tròn giữa **Parent object** và **Child object** khiến cho việc thu hồi vùng nhớ không được diễn ra cho hai objects này do có hai biến bên ngoài là *parent_node* và *child_node* còn tham chiếu đến.
Ở bước 3 và 4, mặc dù thực hiện xoá tham chiếu của biến bên ngoài *parent_node* đến *Parent object*. Cơ chế thu hồi vùng nhớ vẫn không thể thực hiện được do vẫn còn biến bên ngoài *child_node* thông qua *Child object.parent* tham chiếu đến *Parent object* và đồng thời tham chiếu đến *Child object*.
Ở bước 5 và 6, thực hiện xoá tham chiếu của biến bên ngoài *child_node*. Lúc này, không còn biến bên ngoài nào tham chiếu đến *Parent object* và *Child object* nữa. Cơ chế thu hồi vùng nhớ thực hiện thành công.

Ngoài cách, xoá biến hoặc gán biến cho ví dụ trên có thể thử dụng thư viện sẵn có ‘weakref‘ để xử lý vấn đề tham chiếu vòng tròn một cách hiệu quả.

```python
# Dùng thư viện weakref
class Parent:
    def __init__(self):
        self.children = []
```

```
    def add_child(self, child):
        parent_weakref = weakref.ref(self)

        def remove_dead_ref(weak_ref):
            parent_obj = parent_weakref()
            # Xoá dead weakref from children list
            if parent_obj is not None and weak_ref in parent_obj.children:
                parent_obj.children.remove(weak_ref)
                print(f"Automatically removed dead child reference")

        child_weakref = weakref.ref(child, remove_dead_ref)

        self.children.append(child_weakref)

        child.parent = weakref.ref(self)  # <<=== Dùng weakref để không tham chiếu đến
                                            cùng vùng nhớ
        print(f"Parent: {hex(id(self))}, Child: {hex(id(child_weakref))}")

    def __del__(self):
        print(f"Thu hồi Parent với id: {hex(id(self))}")
```

II.2.2.2 Vấn đề vô ý chỉnh sửa biến ở mức class

```
class Student:
    all_grades = []  # Class variable

    def __init__(self, name):
        self.name = name

    def add_grade(self, grade):
        self.all_grades.append(grade)  # Chỉnh sửa trên class variable!

student1 = Student("John")
student2 = Student("Jane")
student1.add_grade(90)
student2.add_grade(85)
print(Student.all_grades)

# Kết quả:
[90, 85]
```

Biến ở mức class chia sẻ chung vùng nhớ giữa các instances của class đó. Do đó, việc thay đổi giá trị của biến ở mức class từ bất kỳ instance nào cũng sẽ ảnh hưởng đến các instances còn lại. Hãy thật sự lưu ý mỗi khi chỉnh sửa biến ở mức class.

II.2.3 Vấn đề về ghi đè phương thức trong kế thừa class (Method Overriding)

Một trong các vấn đề phổ biến ở nhóm này là sai cách khai báo phương thức ghi đè về số lượng tham số (Signature). Các điều kiện cơ bản để thực hiện việc ghi đè phương thức đảm bảo tính đa hình giữa các lớp như sau:

- Cần phải diễn ra sự kế thừa giữa các lớp.
- Phương thức cần ghi đè ở lớp kế thừa phải trùng tên với phương thức ở lớp được kế thừa (lớp cha).
- Số lượng tham số phải trùng nhau giữa phương ghi đè ở lớp kế thừa và lớp được kế thừa.
- Tên tham số tương ứng được khuyến nghị là trùng nhau ở phương thức ghi đè giữa lớp kế thừa và lớp được kế thừa

Cùng xem qua ví dụ dưới đây về lỗi ghi đè phương thức khác số lượng tham số

```
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def area(self, length, width):  # <<== Khác số lượng tham số
        return length * width

### Không thể dùng polymorphism
shapes = [Rectangle()]
for shape in shapes:
    print(shape.area())

### Báo lỗi:
TypeError: Rectangle.area() missing 2 required positional arguments: 'length' and '
                                            width'
```

Kiểm tra qua các điều kiện để thực hiện ghi đè phương thức dẫn đến lỗi như trên:

- Có diễn ra sự kế thừa: **lớp Rectangle kế thừa lớp Shape**.
- Tên phương thức ghi đè trùng nhau: **area**
- Số lượng tham số trùng nhau: **Không**. Ở lớp Rectangle có 2 tham số, còn ở lớp Shape không có tham số nào (không tính tham số *self*)

II.3. Nhóm lỗi về convention

Convention là một tập các hướng dẫn và khuyến nghị được chấp nhận rộng rãi nhằm mục đích hỗ trợ việc viết mã nguồn Python một cách gọn gàng, dễ đọc và dễ bảo trì. Convention luôn được cải thiện qua thời gian do những thay đổi ở các phiên bản Python khác và/hoặc sự chấp nhận những quy chuẩn mới từ cộng đồng.

Với đặc điểm trên Convention có thể không hoàn toàn thống nhất giữa các dự án khác nhau. Nó có thể được biên soạn lại cho phù hợp với dự án. Và cũng chính vì vậy có rất nhiều phiên bản convention khác nhau chẳng hạn như: PEP8, PEP13, PEP257, hay PEP20.

Cùng điểm qua một số lỗi phổ biến sau đây:

Nhóm lỗi	X	✓

1. Lỗi đặt tên không theo quy chuẩn	`class student_class:` ◀
	`  def __init__(self):`
	`  self.StudentName = ""` ◀
	`  self.STUDENT_AGE = 0` ◀
	`  def GetStudentInfo(self):` ◀
	`  pass`
	Lỗi: **Không báo lỗi**
	Không tuân theo quy chuẩn về đặt tên Lớp, biến, phương thức
		`class StudentClass:` ◀ theo kiểu PascalCase
		`  def __init__(self):`
		`  self.student_name = ""` ◀ theo kiểu snake_case
		`  self.student_age = 0`
		`  def get_student_info(self):` ◀ theo kiểu snake_case
		`  pass`

2. Hiểu sai cách sử dụng biến với phạm vi public của lớp	`class Person:`
	`  def __init__(self):`
	`  self.__nickname = "Turtle"` ◀ phạm vi private
	Lỗi: **Không báo lỗi**
	Mong muốn sử dụng biến *nickname* ở phạm vi public nhưng lại khai báo thành biến ở phạm vi private
		`class Person:`
		`  def __init__(self):`
		`  self.nickname = "Turtle"` ◀ phạm vi public
		Biến *nickname* mong muốn được sử dụng phạm vi public do không chứa thông tin nhạy cảm, không cần chỉ được sử dụng ở phạm vi nội bộ Lớp.
		Có thể được sử dụng:
		-- Trong hoặc ngoài Lớp và;
		-- Giữa các Lớp kế thừa

3. Hiểu sai cách sử dụng biến với phạm vi protected của lớp	`class Person:`
	`  def __init__(self):`
	`  self.address = "No.11 street ABC"` ◀ phạm vi public
	Lỗi: **Không báo lỗi**
	Mong muốn sử dụng biến *address* ở phạm vi protected nhưng lại khai báo thành biến ở phạm vi public
		`class Person:`
		`  def __init__(self):`
		`  self._address = "No.11 street ABC"` ◀ phạm vi protected
		`  @property`
		`  def address(self):`
		`  return self._address`
		`  @address.setter`
		`  def address(self, newvalue):`
		`  self._address = newvalue`
		Mong muốn sử dụng biến *address* ở phạm vi protected. Phạm vi này có thể được sử dụng:
		--Trong hoặc ngoài Lớp;
		--Không khuyến khích truy xuất biến loại này trực tiếp ở bên ngoài, mà cần không qua phương thức getter và setter;
		--Giữa các Lớp kế thừa

4. Hiểu sai cách sử dụng biến với phạm vi private của lớp	`class Person:`
	`  def __init__(self):`
	`  self._secret_number = "xxxx-yyy-zzz"` ◀ phạm vi public
	Lỗi: **Không báo lỗi**
	Mong muốn sử dụng biến *secret_number* ở phạm vi private nhưng lại khai báo thành biến ở phạm vi protected
		`class Person:`
		`  def __init__(self):`
		`  self.__secret_number = "xxxx-yyy-zzz"` ◀ phạm vi private
		Biến *secret_number* mong muốn được sử dụng phạm vi private do chứa thông tin nhạy cảm, chỉ được sử dụng ở phạm vi nội bộ Lớp đó.
		Có thể được sử dụng:
		-- Trong nội tại Lớp đó;
		-- Không sử dụng được giữa các Lớp kế thừa

II.4. Nhóm lỗi về quy chuẩn thiết kế, nguyên tắc thiết kế, hiệu năng và sử dụng bộ nhớ

II.4.1. God class - Thiết kế quá nhiều nhiệm vụ cho một lớp

```python
class Student:
    def __init__(self, name):
        self.name = name

    def study(self): pass
    def eat(self): pass
    def sleep(self): pass
    def calculate_gpa(self): pass
```

```
    def send_email(self): pass  # <--
    def manage_database(self): pass  # <--
    def generate_report(self): pass  # <--
```

Nhận xét: Với chủ thể là **Student** có thể nhận thấy phương thức hay hành động **send_email**, **manage_database**, **generate_report** dư thừa trong lớp này. Các phương thức này làm cho lớp **Student** quản lý quá nhiều tác vụ khác nhau thay vì tập trung vào những tác vụ chính của một Student. Các phương thức này nên được phân vào lớp khác như:

- **send_email** nên được đưa vào lớp **Email** hoặc mô-đun **EmailService** để quản lý.
- **manage_database** nên được đưa vào lớp **DatabaseConnection** để quản lý.
- **generate_report** nên được đưa vào lớp **Report** để quản lý.

Mỗi lớp nên đảm nhận một nhiệm vụ cụ thể để tuân thủ nguyên tắc đơn nhiệm vụ, *Nguyên tắc thiết kế đơn nhiệm vụ - Single Responsibility Principle* trong OOP.

II.4.2. Vấn đề liên kết quá chặt chẽ giữa các thành phần khác nhau trong mã nguồn - Tight Coupling

```
class EmailService:
    def send_email(self, message):
        smtp_server = SMTPServer("gmail.com") # <<===
        smtp_server.send(message)

class User:
    def __init__(self):
        self.email_service = EmailService()  # <<===
```

Nhận xét: Có thể lớp **EmailService** trực tiếp sử dụng lớp **SMTPServer** với tham số *gmail.com* bên trong. lớp **User** sử dụng trực tiếp lớp**EmailService** bên trong. Điều này có nghĩa là:

- lớp **EmailService** luôn phải sử dụng **SMTPServer** với tham số *gmail.com* đã được định sẵn, không thể chuyển sang một máy chủ khác như *outlook.com* khi cần thiết hoặc một máy chủ giả nào đó để phục vụ cho việc kiểm thử.
- lớp **User** chỉ có thể sử dụng dịch vụ mail từ lớp **EmailService**, không thể chuyển sang dịch vụ khác.

Điều này dẫn đến các lớp phụ thuộc vào nhau quá chặt chẽ, dẫn đến:

- **Khó kiểm thử**: Khó khăn hơn trong việc tạo dịch vụ giả giống như *EmailService* hay *SMTPServer* để thực hiện kiểm thử.
- **Khó bảo trì mã nguồn**: Bất kì thay đổi nào trong *EmailService* cũng có thể phải thực hiện thay đổi ở *User*.

- **Khó tái sử dụng**: Vì các lớp phụ thuộc bằng cố định dòng code nên việc thay đổi bằng lớp khác có thể gây ảnh hưởng không mong muốn trong việc tái sử dụng cho lớp khác.

Một trong các giải pháp cho vấn đề này là sử dụng một lớp Interface làm giao tiếp chung và kỹ thuật Depedency Injection để giảm sự phụ thuộc giữa các lớp:

```python
from abc import ABC, abstractmethod

class EmailServer(ABC):

    @abstractmethod
    def __init__(self, server_name) -> None:
        self._server_name = server_name

    @abstractmethod
    def send(self, message):
        """Send email"""
        print("Must override this method on your subclass")

class SMTPServer(EmailServer):
    def __init__(self, server_name) -> None:
        super().__init__(server_name)

    def send(self, message):
        print(self._server_name)
        print(message)

class EmailService:
    def __init__(self, email_server) -> None:
        self.email_server = email_server

    def send_email(self, message):
        self.email_server.send(message)


smtp_server = SMTPServer("gmail.com")
email_service = EmailService(email_server=smtp_server)
email_service.send_email("Hello John")
```

II.4.3. Vấn đề về tối ưu hiệu năng và bộ nhớ với ___slots___

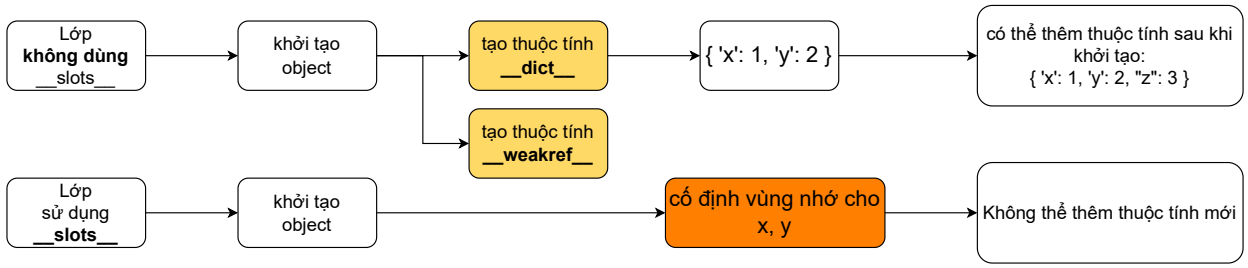
Cùng tìm hiểu nhanh về **___slots___** qua các nội dung sau:

- **___slots___** là gì?
  - Một biến ở mức lớp (class variable) và chỉ được sử dụng trong định nghĩa lớp, không thể sử dụng ở ngoài như một hàm tích hợp sẵn. Nó chứa một list hoặc tuple các biến ở mức instance (instance variables) được phép sử dụng.
  - Các biến được xác định trong nó sẽ được đưa vào vùng nhớ chỉ định trước cho việc truy xuất thuộc tính của lớp. Dẫn đến có thể thay đổi giá trị của biến **___slots___**, nhưng không thể thay đổi được địa chỉ vùng nhớ xác định sẵn sau khi lớp được khởi tạo.

  - Chỉ tạo sẵn vùng nhớ cho việc truy xuất thuộc tính, không tạo sẵn vùng nhớ cho giá trị của mỗi thuộc tính.
  - Khi được sử dụng, nó sẽ ngăn chặn việc tạo ra thuộc tính ___*dict*___ và ___*weakref*___ của lớp

- **Tại sao cần sử dụng ___slots___?**
  - *Tối ưu bộ nhớ*: Do được chỉ định sẵn vùng nhớ khi khởi tạo. Và không khởi thêm thuộc tính ___*dict*___ và ___*weakref*___ của lớp.
  - *Truy xuất thuộc tính được định nghĩa nhanh hơn*: Do đã định sẵn vùng nhớ cho các thuộc tính thay vì truy xuất từ một *dict* các thuộc tính của lớp.
  - *Ngăn chặn truy xuất thuộc tính chưa được định nghĩa trước*: Truy xuất thuộc tính không được định nghĩa trước sẽ báo lỗi *AttributeError*

- **Khi nào không nên sử dụng ___slots___?**
  - *Linh động trong sử dụng thuộc tính*: ___*slots*___ cố định các thuộc tính
  - *Cần sử dụng thuộc tính* ___*dict*___: Muốn truy xuất vào thuộc tính ___*dict*___ để lấy thông tin.
  - *Sử dụng tham chiếu nới lỏng*: Sử dụng thuộc tính ___*weakref*___.



Hình 3: Minh họa khởi tạo giữa lớp có và không có ___slots___

Cùng điểm qua một số ví dụ có thể cải thiện hiệu năng và bộ nhớ bằng cách sử dụng **___slots___**:

Ví dụ 1: Liên tục tạo mới object khi gọi đến phương thức của lớp

```python
# Code có vấn đề

class Calculator:
    def add(self, a, b):
        result = Result(a + b)  # <<== Tạo object mới mỗi lần gọi add
        return result.value

class Result:
    def __init__(self, value):
        self.value = value
        print("a result instance's memory id: ", id(self))

# Code tối ưu hơn trong trường hợp này:
```

```python

class Calculator:
    def __init__(self) -> None:
        self._result = Result()

    def add(self, a, b):
        self._result.add_value(a + b)

        return self._result.value

class Result:
    __slots__ = ("_value")

    def __init__(self):
        self._value = None
        print("a result instance's memory id: ", id(self))

    @property
    def value(self):
        return self._value

    def add_value(self, newvalue):
        self._value = newvalue
```

Ví dụ 2: Vấn đề hiệu năng giữa class có dùng và không dùng ___slots___

```python
import sys
import time
import gc

class ResultWithSlots:
    __slots__ = ("value",)

    def __init__(self, value):
        self.value = value

class ResultNoSlots:
    def __init__(self, value):
        self.value = value

def measure(cls, count=5_000_000):
    gc.collect()
    t0 = time.time()
    cls_instances = [cls(i) for i in range(count)]
    t1 = time.time()
    size_one = sys.getsizeof(cls_instances[0])
    total = sum(sys.getsizeof(o) for o in cls_instances)
    print(f"\nClass: {cls.__name__}")
    print(f"Sử dụng bộ nhớ của 1 instance: {size_one} bytes")
    print(f"Tổng sử dụng bộ nhớ (giá trị gần đúng): {total / (1024 * 1024):.2f} MB")
    print(f"Thời gian khởi tạo tất cả các instances: {t1 - t0:.3f} sec")

```

```
measure(ResultWithSlots)
measure(ResultNoSlots)

# Kết quả:
Class: ResultWithSlots
Sử dụng bộ nhớ của 1 instance: 40 bytes
Tổng sử dụng bộ nhớ (giá trị gần đúng): 190.73 MB
Thời gian khởi tạo tất cả các instances: 1.529 sec

Class: ResultNoSlots
Sử dụng bộ nhớ của 1 instance: 48 bytes
Tổng sử dụng bộ nhớ (giá trị gần đúng): 228.88 MB
Thời gian khởi tạo tất cả các instances: 1.598 sec
```

Nhận xét: Sau khi tạo 5 triệu instances cho mỗi lớp **ResultWithSlots**, **ResultNoSlots** thì tổng bộ nhớ dùng cho lớp không sử dụng ___**slots**___ tăng hơn 30Mb, thời gian tạo lâu hơn gần 0.07 giây so với với lớp có sử dụng.

Chứng tỏ việc sử dụng hợp lý ___**slots**___ mang lại sự tối ưu hơn trong trường hợp này.

III. Chiến lược phòng ngừa và truy vết lỗi

Với rất nhiều và đa dạng các loại lỗi, vấn đề ở phần trước, việc chuẩn bị các chiến lượng phòng ngừa lỗi là hết sức cần thiết. Cần thực hiện ngay từ lúc bắt đầu viết mã nguồn để tạo ra mã nguồn gọn gàng, dễ đọc hiểu, dễ bảo trì. Tuy nhiên, việc phát sinh lỗi là hoàn toàn có thể xảy ra bất kỳ lúc nào. Do đó, quy trình truy vết lỗi là hết sức cần thiết để có thể khắc phục lỗi một cách tốt nhất.

III.1. Thực hiện đối chiếu danh sách lỗi toàn diện sau khi thực hiện viết mã nguồn

Kiểm tra hàm khởi tạo	**Kiểm tra cách truy xuất thuộc tính**
☐ Tất cả tham số bắt buộc đã được truyền vào *___init___*? ☐ *super.___init___* được gọi trong lớp kế thừa chưa? ☐ Thứ tự gọi *super* đúng chưa? ☐ Không dùng tham biến mặc định với giá trị là kiểu dữ liệu mutable?	☐ Tất cả instance variables đều dùng được truy xuất bằng *self.* chưa? ☐ Các biến ở phạm vi Private trong lớp đã được truy cập đúng cách chưa? ☐ Xác định rõ class và instance variables? ☐ Có vô ý chỉnh sửa class variables nào hay không?
Kiểm tra định nghĩa phương thức	**Kiểm tra việc kế thừa**
☐ Tất cả instance methods đều có tham số *self* chưa? ☐ Tên tham số đầu tiên của phương thức trong lớp là *self* (không phải là *obj, this*)? ☐ Các tham số trong phương thức được ghi đè đã đúng và trùng khớp giữa lớp kế thừa và lớp được kế thừa? ☐ Phương thức từ lớp được kế thừa có được gọi khi cần thiết chưa?	☐ Không có hiện tượng tham chiếu vòng tròn ☐ MRO (Method Resolution Order) được xử lý đúng chưa? ☐ Vấn đề đa kế thừa từ nhiều lớp với cấp độ khác nhau được giải quyết chưa? ☐ Cơ chế đa hình có hoạt động đúng không?
Kiểm tra về quy tắc thiết kế và quy chuẩn viết mã nguồn	**Kiểm tra về hiệu năng và sử dụng bộ nhớ**

☐ Quy chuẩn đặt tên đã tuân thủ *PascalCase* cho lớp, *snake_case* cho phương thức chưa? ☐ Nguyên tắc Đơn nhiệm vụ - Single Responsibility Principle được tuân thủ không? ☐ Đã thực hiện giảm sự kết nối chặt chẽ giữa các lớp (Loose Coupling) chưa? ☐ Đã sử dụng định dạng đúng của biến hoặc phương thức cho phạm vi private hoặc protected hoặc public chưa?	☐ Không có memory leaks từ Tham chiếu vòng tròn (Circular References)? ☐ Việc tạo các objects từ các lớp có tối ưu chưa? ☐ ___*slots*___ có được dùng khi cần thiết? ☐ Không xảy ra việc tạo cùng object nhiều lần không cần thiết trong các vòng lặp?

III.2. Quy trình truy vết lỗi

Có rất nhiều cách tiếp cận để truy vết lỗi. Dưới đây là một số cách tiếp cận dựa vào tính chất lỗi để truy vết:

III.2.1 Truy vết theo lỗi bắn ra từ chương trình:

- Lỗi bắn ra chương trình có thể làm dừng chương trình tại đó nếu không có xử lý bắt lỗi. Nếu có xử lý bắt lỗi thông thường lỗi sẽ được ghi lại vào trong file log của chương trình với cùng nội dung bắn lỗi.
- Đặc điểm của loại lỗi này là do framework hay chương trình biên dịch Python bắn ra có kèm theo chi tiết về lỗi. Đa phần sẽ dễ dàng xác định lỗi chính xác phát sinh từ đâu.
- Sử dụng truy vết ngược dựa trên lỗi để khắc phục trong tương quan với toàn bộ code bị ảnh hưởng. Nếu chỉ quan tâm đến dòng code lỗi mà không quan tâm đến toàn bộ lớp hoặc phương thức chứa dòng lỗi, có thể không khắc phục hoàn toàn lỗi phát sinh mà còn gây ra các lỗi khác không mong muốn.

Cùng phân tích một số dòng lỗi sau để có cái nhìn chi tiết hơn:

Ví dụ 1:

```
Traceback (most recent call last):
  File "<path_to_project>/Project/test.py", line 807, in <module>
    measure(ResultWithSlots)
  File "<path_to_project>/Project/test.py", line 798, in measure
    cls_instances = [cls(i, j) for i in range(count)]
                                        ^
NameError: name 'j' is not defined
```

Nhận xét: Từ lỗi trên có thể nhận ra các đặc điểm sau:

- Loại lỗi: **NameError**
- Thông báo lỗi: *name 'j' is not defined*, biến *j* chưa được định nghĩa
- Thông tin truy vết: Lỗi ở dòng code số **798**, thuộc file **test.py**. Xuất phát từ việc chạy hàm **measure(ResultWithSlots)** ở dòng code số **807** trong file **test.py**
- Hướng khắc phục: Giả định, dựa trên lỗi, dự định sẽ khai báo j bằng một giá trị nào đó. Tuy nhiên, cần xem xét biến *cls* đại diện cho một lớp nào đó có nhận vào bao nhiêu tham số khi khởi tạo và giá trị của mỗi tham số là gì để có thể khắc phục lỗi hoàn toàn.

Ví dụ 2:

```
Traceback (most recent call last):
  File "/Users/jrvn-hieu/Desktop/Practical/aivn-aio2024-mlops/test.py", line 808, in <
                                          module>
    measure(ResultWithSlots)
  File "/Users/jrvn-hieu/Desktop/Practical/aivn-aio2024-mlops/test.py", line 799, in
                                          measure
    cls_instances = [cls(i, j) for i in range(count)]
                     ^^^^^^^^^
TypeError: ResultWithSlots.__init__() takes 2 positional arguments but 3 were given
```

Nhận xét: Từ lỗi trên có thể nhận ra các đặc điểm sau:

- Loại lỗi: **TypeError**
- Thông báo lỗi: *ResultWithSlots.___init___ takes 2 positional arguments but 3 were given* - Phương thức init của lớp *ResultWithSlots* nhận vào 2 tham số, nhưng tham số đưa vào lại là 3 tham số.
- Thông tin truy vết: Lỗi ở dòng code số **799**, thuộc file **test.py**. Xuất phát từ việc chạy hàm **measure(ResultWithSlots)** ở dòng code số **808** trong file **test.py**
- Hướng khắc phục: Hãy kiểm tra lại tham số của lớp **ResultWithSlots** và truyền lại tham số cho đúng.

Ví dụ 3:

```
Traceback (most recent call last):
  File "/Users/jrvn-hieu/Desktop/Practical/aivn-aio2024-mlops/test.py", line 807, in <
                                          module>
    measure(ResultWithSlots)
  File "/Users/jrvn-hieu/Desktop/Practical/aivn-aio2024-mlops/test.py", line 802, in
                                          measure
    print(f"\nClass: {cls.__unknown_name__}")
                      ^^^^^^^^^^^^^^^^^^^^^
AttributeError: type object 'ResultWithSlots' has no attribute '__unknown_name__'
```

Nhận xét: Từ lỗi trên có thể nhận ra các đặc điểm sau:

- Loại lỗi: **AttributeError**
- Thông báo lỗi: *type object 'ResultWithSlots' has no attribute '___unknown_name___'* - đối tượng *ResultWithSlots* không có thuộc tính *___unknown_name___*.
- Thông tin truy vết: Lỗi ở dòng code số **802**, thuộc file **test.py**. Xuất phát từ việc chạy hàm **measure(ResultWithSlots)** ở dòng code số **807** trong file **test.py**
- Hướng khắc phục: Hãy kiểm tra lại thuộc tính lớp **ResultWithSlots** xem thuộc tính đã được định nghĩa chưa? Hay đang truy xuất sai tên thuộc tính của lớp.

III.2.2 Truy vết theo lỗi lô-gic nghiệp vụ:

Loại lỗi lô-gic nghiệp vụ này thông thường không bắn lỗi, hoặc có thể bắn lỗi tùy thuộc vào loại nghiệp vụ và cách bắt lỗi trong chương trình. Loại lỗi này bắt buộc phải có tài liệu về nghiệp vụ để đối chiếu, truy vết và tái hiện lỗi trong điều kiện tương tự để khắc phục hoàn toàn lỗi. Việc ghi log chi tiết trong chương trình sẽ giúp ích nhiều cho loại lỗi này.

III.2.3 Truy vết theo lỗi hiệu năng và sử dụng bộ nhớ:

Có nhiều cách, thư viện để kiểm tra hiệu năng của phương thức. Với mục tiêu dễ sử dụng, đủ thông tin, có thể sử dụng 3 thư viện kiểm tra thời gian xử lý và bộ nhớ sử dụng cho từng dòng code (ở môi trường development) là **line_profiler**, **memory_profiler**, scalene.

Thư viện **scalene** có một nhược điểm là chương trình của bạn phải chạy đủ lâu để theo dõi và ghi nhận dữ liệu. Trong bài này chúng ta sẽ dùng **line_profiler** và **memory_profiler**. Mặc dù hơi thủ công một chút chỉ theo dõi trên phương thức hoặc hàm, không dùng được cho biến hoặc lớp.

Sau đây sẽ giới thiệu cách sử dụng đơn giản kết hợp cả hai để theo dõi hiệu năng và sử dụng bộ nhớ.

```python
import logging
import random
from line_profiler import profile as time_profile
# from memory_profiler import profile as mem_profile

# ...một phần code của chương trình

@time_profile
# @mem_profile
def main():
    """Main function with line profiler only"""
    orders = [
        Order(product_ID=f"snack0{random.randint(1, 4)}", quantity=random.randint(1, 5)
                                                , price_per_unit=random.randint(11, 20)),
        Order(product_ID=f"snack0{random.randint(1, 4)}", quantity=random.randint(1, 5)
                                                , price_per_unit=random.randint(16, 25)),
        Order(product_ID=f"snack0{random.randint(1, 4)}", quantity=random.randint(1, 5)
                                                , price_per_unit=random.randint(31, 40)),
        Order(product_ID=f"snack0{random.randint(1, 4)}", quantity=random.randint(1, 5)
                                                , price_per_unit=random.randint(16, 25)),
    ]

    revenue = Revenue(orders=orders)
    cost = ProductCost(orders=orders)
    income = Income(revenue=revenue, cost=cost)
    result = income.compute()
    print(f"Income: {result}")
    return result

if __name__ == "__main__":
    main()
```

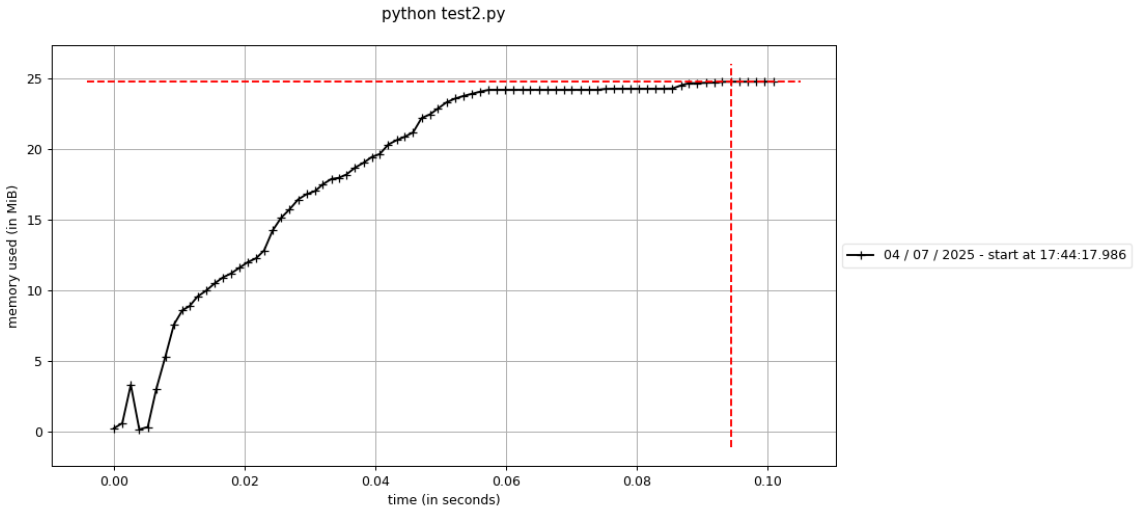
```bash
# Chạy chương trình theo thứ tự sau trên terminal
# Theo dõi hiệu năng từng dòng code
# -- Sau khi chạy lệnh dưới sẽ sinh ra file performance_and_memory_monitor.py.lprof
kernprof -l performance_and_memory_monitor.py

# Theo dõi sử dụng bộ nhớ
# -- Sau khi chạy lệnh dưới sẽ sinh ra dữ liệu theo dõi sử dụng bộ nhớ, chẳng hạn như:
                                                mprofile_20250702163954.dat
# -- [tuỳ chọn] có thể thay @time_profile bằng @mem_profile để xem luôn dữ liệu sử dụng
                                                 bộ nhớ theo dòng code trước khi chạy code
                                                bên dưới

mprof run --interval 0.01 python performance_and_memory_monitor.py

# Xem báo cáo hiệu năng
python3 -m line_profiler -rmt "performance_and_memory_monitor.py.lprof"

# Xem báo cáo sử dụng bộ nhớ với dữ liệu mới nhất
mprof plot
```

Kết quả như sau:

```
# Hiệu năng theo từng dòng code của hàm

Timer unit: 1e-06 s

Total time: 0.000651 s
File: test2.py
Function: main at line 92

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    92                                           @time_profile
    93                                           def main():
    94                                               """Main function with line
                                                 profiler only"""
    95         1          0.0      0.0      0.0      orders = [
    96         1         24.0     24.0      3.7          Order(product_ID=f"snack0{
                                                 random.randint(1, 4)}", quantity=random.
                                                 randint(1, 5), price_per_unit=random.
                                                 randint(11, 20)),
    97         1          5.0      5.0      0.8          Order(product_ID=f"snack0{
                                                 random.randint(1, 4)}", quantity=random.
                                                 randint(1, 5), price_per_unit=random.
                                                 randint(16, 25)),
    98         1          5.0      5.0      0.8          Order(product_ID=f"snack0{
                                                 random.randint(1, 4)}", quantity=random.
                                                 randint(1, 5), price_per_unit=random.
                                                 randint(31, 40)),
    99         1          4.0      4.0      0.6          Order(product_ID=f"snack0{
                                                 random.randint(1, 4)}", quantity=random.
                                                 randint(1, 5), price_per_unit=random.
                                                 randint(16, 25)),
   100                                               ]
   101
   102         1          2.0      2.0      0.3      revenue = Revenue(orders=orders)
   103         1          1.0      1.0      0.2      cost = ProductCost(orders=orders)
   104         1          1.0      1.0      0.2      income = Income(revenue=revenue,
                                                 cost=cost)
   105         1        591.0    591.0     90.8      result = income.compute()
   106         1         17.0     17.0      2.6      print(f"Income: {result}")
   107         1          1.0      1.0      0.2      return result
```

```
# Sử dụng bộ nhớ theo từng dòng code

Filename: test2.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    93     24.9 MiB     24.9 MiB           1   @mem_profile
    94                                         def main():
```

```
9     95                                                """Main function with line profiler
                                                        only"""
10    96     24.9 MiB      0.0 MiB           1       orders = [
11    97     24.9 MiB      0.0 MiB           1           Order(product_ID=f"snack0{random
                                                        .randint(1, 4)}", quantity=random.randint(1
                                                        , 5), price_per_unit=random.randint(11, 20)
                                                        ),
12    98     24.9 MiB      0.0 MiB           1           Order(product_ID=f"snack0{random
                                                        .randint(1, 4)}", quantity=random.randint(1
                                                        , 5), price_per_unit=random.randint(16, 25)
                                                        ),
13    99     24.9 MiB      0.0 MiB           1           Order(product_ID=f"snack0{random
                                                        .randint(1, 4)}", quantity=random.randint(1
                                                        , 5), price_per_unit=random.randint(31, 40)
                                                        ),
14   100     24.9 MiB      0.0 MiB           1           Order(product_ID=f"snack0{random
                                                        .randint(1, 4)}", quantity=random.randint(1
                                                        , 5), price_per_unit=random.randint(16, 25)
                                                        ),
15   101                                                ]
16   102
17   103     24.9 MiB      0.0 MiB           1       revenue = Revenue(orders=orders)
18   104     24.9 MiB      0.0 MiB           1       cost = ProductCost(orders=orders)
19   105     24.9 MiB      0.0 MiB           1       income = Income(revenue=revenue,
                                                    cost=cost)
20   106     24.9 MiB      0.0 MiB           1       result = income.compute()
21   107     24.9 MiB      0.0 MiB           1       print(f"Income: {result}")
22   108     24.9 MiB      0.0 MiB           1       return result
```



Hình 4: Trực quan hoá sử dụng bộ nhớ theo thời gian

III.3. Chiến lược làm theo mẫu có sẵn và kiểm thử

Sử dụng mẫu lớp:

```python
# Standard class template
class ClassName:
    """Class docstring"""

    class_variable = "default"  # Class variable

    def __init__(self, required_param, optional_param=None):
        """Constructor docstring"""
        super().__init__()  # If inheriting
        self.required_param = required_param
        self.optional_param = optional_param or []
        self._protected_var = "internal"
        self.__private_var = "secret"

    def public_method(self):
        """Public method docstring"""
        return self._protected_method()

    def _protected_method(self):
        """Protected method docstring"""
        return self.__private_var

    def __str__(self):
        """String representation"""
        return f"ClassName({self.required_param})"

    def __repr__(self):
        """Developer representation"""
        return f"ClassName(required_param={self.required_param!r})"
```

Dựa vào mẫu, lập trình viên có thể dễ dàng đối chiếu để kiểm tra việc sử dụng các thành phần trong class đã đúng hay chưa.

Viết kiểm thử trước hoặc sau khi viết code:

```python
def test_class():
    # Test constructor
    obj = ClassName("test")
    assert obj.required_param == "test"

    # Test methods
    result = obj.public_method()
    assert result is not None

    # Test inheritance
    if hasattr(obj, 'parent_method'):
        obj.parent_method()

```

```
    # Test edge cases
    try:
        ClassName()  # Should raise TypeError
        assert False, "Should have raised TypeError"
    except TypeError:
        pass
```

Nên luôn viết kiểm thử cho mỗi đoạn mã. Có thể bắt đầu với các test đơn giản kiểm tra đầu ra, sau đó mở rộng dần.

III.4. Mô tả nhanh code đã viết và tự đánh giá lại code

Trước khi code, hãy tạo checklist về lô-gic cần viết code. Sau khi viết xong, mô tả lại phần code đã viết dưới dạng tóm tắt gạch đầu dòng. Tiếp theo, tự đánh giá lại giữa code và checklist lô-gic xem đã đáp ứng được chưa.

```
# Mẫu tự mô tả

+ Mô tả yêu cầu nghiệp vụ:
[] Yêu cầu 1
[] Yêu cầu 2

+ Mô tả code đã viết:
- phần code dùng phương thức <method_a> để xử lý yêu cầu 1.
- dùng phương thức <method_b> để xử lý yêu cầu 2.

+ Phạm vi ảnh hưởng:
- Có ảnh hưởng đến phần nào khác không
- Ảnh hưởng đến lớp <class_a> do thay đổi lô-gic
```

III.5. Sử dụng công cụ hỗ trợ khi code

Như đã đề cập ở các phần trước đó, các nguyên tắc, quy chuẩn trong Python là rất nhiều. Liệu chúng ta có thể học thuộc hoặc làm quen đến mức nhớ hết tất cả được không? Câu trả lời là có thể. Tuy nhiên, để hỗ trợ cho quá trình này, các tiện ích, công cụ hỗ trợ ra đời.

Hiện tại có rất nhiều trình biên soạn code (code editor) chẳng hạn như: VS Code, Cursor, Sublime, Vim, Jetbrains Fleet, và Windsurf. Tất cả điều hỗ trợ plugins và/hoặc extension (phần mở rộng) để hỗ trợ cho quá trình viết code được tối ưu nhất có thể.

Và sau đây là một vài plugin được hỗ trợ trong VS Code và cũng có thể có trong các trình biên soạn code khác. Những plugin này giúp ích rất nhiều để giảm thiểu các lỗi trong quá trình lập trình với Python:

- Flake8: hỗ trợ về convention (quy chuẩn trình bày và viết code).

- Pylance: Gợi ý tham số, gợi ý hoàn thiện code, hỗ trợ thêm dòng import thư viện tự động.
- Pylint: hỗ trợ phát hiện lỗi, phản hồi về chất lượng code trong file đang mở.
- Prettier - Code formatter: giúp code bạn viết trông đẹp mắt và dễ đọc hơn.
- Và còn nhiều hơn nữa. Tham khảo trên Visual Studio Marketplace.

Một lưu ý là càng cài đặt nhiều plugins hoặc extension cho trình biên soạn thì trình biên soạn càng nặng. Nếu máy của bạn không đủ mạnh sẽ làm trình biên soạn chạy chậm hơn.

IV. Tài liệu tham khảo

[1] DoanNgocCuong, *Checklist toan dien cac loi trong python thuong gap*, Github DoanNgocCuong, 2025. [Online]. Available: https://github.com/DoanNgocCuong/BasicTasksLearning_OpenCodingHomework_AIVietnamAIO_20252026/blob/main/Module1_/20250618_TopBugOOP_IIAgent.md.

[2] PhamVanDuc, *Si kyic object oriented design principles*, Viblo PhamVanDuc, 2016. [Online]. Available: https://viblo.asia/p/so-luoc-object-oriented-design-principles-MdZGAQODGox.

[3] T. P. Editors, *Pep 0 – index of python enhancement proposals (peps)*, python.org, 2000. [Online]. Available: https://peps.python.org/#topics.

[4] L. P. Ramos, *Solid principles: Improve object-oriented design in python*, realpython.com, 2023. [Online]. Available: https://realpython.com/solid-principles-python/.

[5] M. Simionato., *The python 2.3 method resolution order*, python.org, Unknown. [Online]. Available: https://docs.python.org/3/howto/mro.html.

[6] Tutorialsteacher, *Python - public, protected, private members*, tutorialsteacher.com, Unknown. [Online]. Available: https://www.tutorialsteacher.com/python/public-private-protected-modifiers.

[7] refactoring.guru, *What's a design pattern?*, refactoring.guru, Unknown. [Online]. Available: https://refactoring.guru/design-patterns/what-is-pattern.

[8] oodesign.com, *Solid design principles*, oodesign.com, Unknown. [Online]. Available: https://www.oodesign.com/design-principles/.

Phụ lục

1. **Rubric:**

Mục	Kiến Thức	Đánh Giá
I.	- Kiến thức về tổng quan lỗi khi Lập trình OOP với Python. - Kiến thức về các lỗi thường gặp. - Kiến thức về chiến lược phòng ngừa và giảm thiểu lỗi.	- Nắm được tổng quan về lỗi khi lập trình OOP với Python. - Nắm được nội dung về các lỗi phổ biến. - Nắm được các chiến lược phòng ngừa và giảm thiểu lỗi.

2. **Source code:** Các bạn có thể download source code tại đây