البرمجة كائنية التوجه في بايثون (oop)

احصل على أحدث نسخة من الملف

Adnan Al-Jamous

**يمكنك الرجوع إلى هذا المرجع كلما احتجت لتذكُّر أو تطبيق مفاهيم البرمجة كائنية التوجه في بايثون، وهو يغطي جميع الجوانب بدءاً من إنشاء الكائنات وحتى استخدام مفاهيم متقدمة مثل الوراثة المتعددة والفئات الأساسية المجردة. **

الفهرس

الفهرس 1
البرمجة كائنية التوجه (OOP) في بايثون 3
1. المقدمة 3
2. مفهوم OOP في بايثون 3
3. الكائنات والفئات 3
1.الكائن (Object) 3
2.الفئة (Class) 4
إنشاء الفئات والمثيلات 4
4. السمات والدوال فيOOP 5
4.1 سمات المثيل(Instance Attributes) 5
4.2دوال المثيل(Instance Methods) 5
5. السمات العامة للفئة (Class Attributes) ودوال الفئة 5
6. الدوال الكلاسيكية والدوال الثابتة 6
6.1الدوال الكلاسيكية(Class Methods) 6
6.2الدوال الثابتة(Static Methods) 6
7. الدوال السحرية(Magic Methods) 6
أمثلة على الدوال السحرية: 6
8. الوراثة(Inheritance) 7
تعريف الوراثة 7
مثال على الوراثة: 7
9. الوراثة المتعددة(Multiple Inheritance) 7
9.2كيفية تطبيق الوراثة المتعددة 8
9.3ترتيب البحث(MRO - Method Resolution Order) 8
9.4مزايا الوراثة المتعددة 9
10. التعددية الشكلية(Polymorphism) 9
10.1مفهوم التعددية الشكلية 9
10.2التعددية مع الأنواع المدمجة 9
10.3التعددية في الفئات (الكلاسات) 10

10.4مزايا التعددية الشكلية..... 11
11. التغليف(Encapsulation) 11
مفهوم التغليف 11
أنواع الوصول للسمات والأساليب:..... 11
مثال على التغليف: 11
12. المُعالجات (Getters) والمُعدّلات(Setters) 12
مفهوم Getters وSetters 12
مثال توضيحي: 12
13. الزخرفةproperty@ 12
مفهومproperty@ 12
مثال توضيحي: 12
الفوائد:..... 13
14. الفئات الأساسية المجردة(Abstract Base Classes - ABC) 13
مفهوم الفئات الأساسية المجردة..... 13
خطوات الاستخدام: 13
الفوائد:..... 14
15. التجريد(Abstraction) 14
مفهوم التجريد بالعموم..... 14
مثال توضيحي باستخدام الفئات الأساسية المجردة:..... 15
16. المفاهيم الأساسية والخلاصة..... 18
المفاهيم الأساسية فيOOP: 18
الخلاصة: 19

فيما يلي مرجع شامل للبرمجة الكائنية التوجه (OOP) في بايثون، يجمع كافة المفاهيم الأساسية مع الأمثلة والتوضيحات المفصلة. يُمكن اعتبار هذا الملف مرجعاً متكاملاً للمبتدئين والمطورين على حد سواء.

البرمجة كائنية التوجه (OOP) في بايثون

---

1. المقدمة

البرمجة الكائنية التوجه (Object-Oriented Programming - OOP) هي أسلوب في كتابة الكود يعتمد على تقسيم البرنامج إلى كائنات (Objects) تحتوي على بيانات (Attributes) ووظائف (Methods). يُساعد هذا النمط في تنظيم الكود، تحسين قابليته للقراءة، وإمكانية إعادة استخدامه، مما يجعله خياراً مناسباً للمشاريع المتوسطة والكبيرة.

---

2. مفهوم OOP في بايثون

- **تعريف:OOP**
يُبنى البرنامج على أساس الفئات (Classes) التي تُستخدم كقوالب لإنشاء الكائنات(Objects) ، حيث يحتوي كل كائن على بيانات وسلوكيات محددة.
- **أسباب استخدام:OOP**

  - تحسين تنظيم البرامج.
  - تسهيل قراءة الكود.
  - تعزيز قابلية إعادة الاستخدام.

- **أنماط البرمجة في بايثون:**

  - الإجرائي :(Procedural) يعتمد على الإجراءات والخطوات.
  - الوظيفي :(Functional) يعتمد على الدوال والعمليات الرياضية.
  - الكائني :(OOP) يعتمد على الكائنات والفئات.

ملاحظة :في بايثون، كل شيء هو كائن (أرقام، نصوص، قوائم، دوال...).

---

3. الكائنات والفئات

1.الكائن (Object)

- **التعريف:**
الكائن هو **مثيل** لفئة معينة، يحتوي على **البيانات (السمات)** و**السلوكيات (الدوال)**.

- **أمثلة:**

إنسان :يمتلك خصائص مثل الاسم والعمر وسلوكيات مثل المشي والتحدث.

سيارة :لها خصائص مثل اللون والطراز وسلوكيات مثل القيادة والتوقف.

- **متى نستخدم الكائنات؟**
نستخدم الكائنات عندما يكون العنصر لديه بيانات دائمة وسلوكيات معينة، أو عندما يحتاج إلى التفاعل مع كائنات أخرى أو إعادة الاستخدام والتوسع. أما إذا كانت العملية بسيطة ولا تتطلب حالة دائمة، فقد يكون استخدام الدوال العادية أكثر كفاءة.

2.الفئة (Class)

- **التعريف:**
الفئة هي **المخطط أو القالب (Blueprint)** المستخدم لإنشاء الكائنات، حيث تحدد السمات والدوال المشتركة لجميع الكائنات المستندة إليها.
- **مثال:**

Car:فئة تصف السيارات بشكل عام، من حيث الخصائص (مثل اللون والطراز والسرعة) والسلوكيات (كالقيادة والتوقف).

مثال بسيط: حلوى المعمول

- **الفئة:(Class)**
هي مثل قالب المعمول الذي يحدد الشكل والحجم.
- **الكائن:(Object)**
هو كل حبة معمول تُصنع من هذا القالب. كل حبة تُعتبر كائنًا منفردًا تشترك مع الآخرين في نفس التصميم الأساسي.

باختصار:

- **الفئة = (Class) التصميم أو القالب.**
- **الكائن = (Object) التطبيق الفعلي للتصميم.**

ويمكن القول بأن الشيء هو نموذج من الفئة وأن الفئة هي إطار للكائن

إنشاء الفئات والمثيلات

```
class Member:
    def __init__(self):
        print("A New Member Has Been Added")

member_one = Member()  # عند إنشاء كائن جديد ستظهر الرسالة
```

ملاحظة:
الكائنات في OOP أكثر مرونة وتنظيماً مقارنة بالقواميس، إذ يتم ربط البيانات بالدوال.

---

4. السمات والدوال فيOOP

4.1 سمات المثيل(Instance Attributes)

- تُعرّف داخل دالة __init__باستخدام selfوتختلف من كائن لآخر.

```
class Member:
    def __init__(self, first_name, middle_name, last_name, gender):
        self.fname = first_name
        self.mname = middle_name
        self.lname = last_name
        self.gender = gender
```

4.2دوال المثيل(Instance Methods)

- تُعرّف داخل الفئة وتأخذ selfكأول معامل لتتيح الوصول إلى سمات الكائن.

```
class Member:
    def __init__(self, first_name, middle_name, last_name):
        self.fname = first_name
        self.mname = middle_name
        self.lname = last_name

    def get_full_name(self):
        return f"{self.fname} {self.mname} {self.lname}"
```

---

5. السمات العامة للفئة (Class Attributes) ودوال الفئة

- **السمات العامة:**
  تُعرّف داخل الفئة خارج دالة __init__وتكون مشتركة بين جميع الكائنات.

```
class Member:
    not_allowed_names = ["Hell", "Shit", "Baloot"]
    users_num = 0

    def __init__(self, first_name, middle_name, last_name, gender):
        self.fname = first_name
        self.mname = middle_name
        self.lname = last_name
        self.gender = gender
        Member.users_num += 1
```

- **دوال الفئة:**
تستخدم السمات العامة وتُنفذ عمليات مثل التحقق من البيانات أو تحديث عداد المستخدمين.

```
    def full_name(self):
        if self.fname in Member.not_allowed_names:
            raise ValueError("Name Not Allowed")
        return f"{self.fname} {self.mname} {self.lname}"

    def delete_user(self):
        Member.users_num -= 1
        return f"User {self.fname} Is Deleted."
```

---

6. الدوال الكلاسيكية والدوال الثابتة

6.1الدوال الكلاسيكية(Class Methods)

- تُعرّف باستخدام @classmethodوتأخذ معامل clsبدلاً من self، مما يعني أنها تعمل على مستوى الفئة وليس الكائن.
- لا تحتاج إلى إنشاء كائن من الفئة.

6.2الدوال الثابتة(Static Methods)

- تُعرّف باستخدام @staticmethodولا تأخذ أي معاملات خاصة.
- مرتبطة بالفئة لكنها مستقلة عن حالة الكائن.

```
class Member:
    not_allowed_names = ["Forbidden", "Restricted", "Banned"]
    users_num = 0

    @classmethod
    def show_users_count(cls):
        print(f"We Have {cls.users_num} Users In Our System.")

    @staticmethod
    def say_hello():
        print("Hello From Static Method")

    باقي تعريفات الدوال والسمات كما في الأمثلة السابقة #
```

---

7. الدوال السحرية(Magic Methods)

الدوال السحرية هي دوال خاصة تبدأ وتنتهي بشرطتين (__)، وتستخدم لتخصيص سلوك الكائنات عند استخدامها مع دوال مدمجة مثل print()وlen().

أمثلة على الدوال السحرية:

- __init__: تهيئة الكائن.
- __str__: تحويل الكائن إلى نص.

- __len__: حساب طول الكائن.

```
class Skill:
    def __init__(self):
        self.skills = ["Html", "Css", "Js"]

    def __str__(self):
        return f"هذه مهاراتي => {self.skills}"

    def __len__(self):
        return len(self.skills)

profile = Skill()
print(profile)       # يستخدم __str__
print(len(profile))  # يستخدم __len__

profile.skills.append("PHP")
profile.skills.append("MySQL")
print(len(profile))  # بعد الإضافة
```

---

8. الوراثة(Inheritance)

تعريف الوراثة

الوراثة هي آلية تُمكن فئة جديدة (الفئة المشتقة) من اكتساب الخصائص والسلوكيات (الأساليب والبيانات) من فئة موجودة مسبقاً (الفئة الأصلية).
تساعد الوراثة في:

- **إعادة استخدام الكود :**استخدام الكود الموجود دون إعادة كتابته.
- **التوسعة :**إضافة أساليب أو خصائص جديدة أو تعديل الأساليب القديمة بما يتناسب مع احتياجات الفئة المشتقة.
- **تقليل التكرار وتحسين التنظيم :**يجعل الكود أكثر نظافة وسهولة في الصيانة.

مثال على الوراثة:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Dog barks
```

---

9. الوراثة المتعددة(Multiple Inheritance)

9.1مفهوم الوراثة المتعددة

الوراثة المتعددة تُتيح للفئة المشتقة أن ترث من أكثر من فئة أساسية واحدة، مما يسمح لها بالوصول إلى خصائص وأساليب عدة فئات في آن واحد. وهذا يوفر مرونة في التصميم ويسمح بإعادة استخدام الكود بطريقة أكثر كفاءة.

9.2كيفية تطبيق الوراثة المتعددة

أ. تعريف الفئات الأساسية

نقوم بتعريف فئتين أساسيتين BaseOneو BaseTwo، تحتوي كل منهما على مُهيئ (__init__) وأساليب خاصة (func_one()و func_two()):

```
class BaseOne:
    def __init__(self):
        print("Base One")

    def func_one(self):
        print("One")


class BaseTwo:
    def __init__(self):
        print("Base Two")

    def func_two(self):
        print("Two")
```

ب. تعريف الفئة المشتقة

الفئة Derivedترث من BaseOneو BaseTwo، مما يسمح لها بالوصول إلى جميع أساليبهما.

```
class Derived(BaseOne, BaseTwo):
    pass
```

ج. إنشاء كائن من الفئة المشتقة واستدعاء الأساليب

عند إنشاء كائن من Derived، سيتم استدعاء المُهيئات من كلا الفئتين الأساسية، ويمكن استدعاء الأساليب الموروثة:

```
my_var = Derived()

# طباعة الدوال الموروثة
print(my_var.func_one)
print(my_var.func_two)

# استدعاء الأساليب الموروثة
my_var.func_one()  # ناتج: "One"
my_var.func_two()  # ناتج: "Two"
```

9.3ترتيب البحث(MRO - Method Resolution Order)

يستخدم بايثون ترتيب البحث في الدوال **(MRO)**لتحديد أي فئة تُستدعى منها الدالة عندما تتواجد عدة وراثات. يمكن التحقق من ترتيب البحث باستخدام:

```
print(Derived.mro())
```

مثال إضافي على MRO في التسلسل الهرمي للوراثة:

```
class Base:
    pass

class DerivedOne(Base):
    pass

class DerivedTwo(DerivedOne):
    pass
```

9.4مزايا الوراثة المتعددة

- **إعادة استخدام الكود :**إمكانية الوصول إلى أساليب من فئات متعددة دون تكرار الكود.
- **إضافة وظائف متعددة :**إمكانية دمج وظائف متعددة داخل فئة واحدة بسهولة.
- **مرونة أكبر في التصميم :**يسمح بتنظيم هيكلي أكثر مرونة، مما يسهل تطوير التطبيقات المعقدة.

---

10. التعددية الشكلية(Polymorphism)

10.1مفهوم التعددية الشكلية

التعددية الشكلية تعني القدرة على استخدام **نفس اسم الأسلوب (Method)** في فئات (Classes) مختلفة بحيث يتم تنفيذها بطرق متعددة تختلف حسب طبيعة الكائن. هذا يسمح لنا بتطبيق نفس الواجهة (interface) بأساليب تتناسب مع متطلبات كل فئة.

10.2التعددية مع الأنواع المدمجة

حتى في العمليات الأساسية على الأنواع المدمجة في لغة البرمجة نلاحظ مثالاً على التعددية:

- **استخدام العامل :(+)**

عند جمع الأعداد

```
n1 = 5
n2 = 3
print(n1 + n2)  # 8 :الناتج
```

:عند دمج النصوص

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)  # الناتج: "Hello World"
```

- **استخدام الدالة:()len**

تعمل الدالة ()lenعلى قياس طول السلسلة أو القائمة، وتتصرف بشكل مختلف حسب نوع البيانات.

الاستنتاج:

من الأمثلة السابقة يتضح أن العامل +يُستخدم بطرق مختلفة حسب نوع البيانات المدخلة، مما يعكس مبدأ التعددية الشكلية.

10.3التعددية في الفئات (الكلاسات)

في البرمجة الكائنية، يمكننا تطبيق التعددية الشكلية من خلال تصميم فئة أساسية تحتوي على أسلوب عام يُجبر الفئات المشتقة على تنفيذه بطرقها الخاصة. لنأخذ المثال التالي:

أ. تعريف الفئة الأساسية

نعرف فئة أساسية Animalتحتوي على أسلوب ()speakدون تنفيذ محدد (أو برفع استثناء) للإشارة إلى ضرورة إعادة تعريفه في الفئات الفرعية:

```
class Animal:
    def speak(self):
        raise NotImplementedError("يجب على الفئات المشتقة تنفيذ هذه الدالة")
```

ب. تعريف الفئات المشتقة وتنفيذ الأسلوب بشكل مختلف

كل فئة مشتقة تقوم بتنفيذ الأسلوب ()speakبطريقة تناسب طبيعتها:

```
class Dog(Animal):
    def speak(self):
        print("Bark")  # الكلب ينطق "Bark"

class Cat(Animal):
    def speak(self):
        print("Meow")  # القطة تنطق "Meow"
```

ج. استخدام التعددية الشكلية

يمكننا الآن إنشاء قائمة تحتوي على كائنات من الفئات المختلفة واستدعاء الأسلوب ,()speakحيث يُنفذ كل كائن النسخة الخاصة بفئته:

```
animals = [Dog(), Cat()]

for animal in animals:
    animal.speak()
```

ما يحدث في هذا المثال؟

- عند استدعاء ()speakعلى كائن من فئة Dogيتم طباعة."Bark"

- وعند استدعاء speak()على كائن من فئة Catيتم طباعة."Meow"

10.4مزايا التعددية الشكلية

- **إعادة استخدام الكود:**
يمكن استخدام نفس اسم الأسلوب عبر فئات متعددة مع اختلاف التنفيذ، مما يقلل من تكرار الكود.
- **المرونة:**
كل فئة يمكنها أن تنفذ الأسلوب بما يتناسب مع طبيعتها دون الحاجة لتغيير الواجهة العامة.
- **سهولة الصيانة:**
يساهم ذلك في الحفاظ على كود نظيف ومرن، مما يسهل تعديله وتطويره فيما بعد.

باختصار، التعددية الشكلية تُمكّننا من كتابة كود أكثر تنظيماً ومرونة حيث يمكننا استدعاء نفس الأسلوب لكائنات مختلفة وتطبيق السلوك المناسب لكل منها تلقائيًا.

---

11. التغليف(Encapsulation)

مفهوم التغليف

التغليف هو إخفاء التفاصيل الداخلية للكائن عن العالم الخارجي، بحيث يُمكن الوصول إلى البيانات عبر واجهات أو أساليب محددة فقط.
يساعد التغليف في:

- حماية البيانات من التلاعب غير المصرح به.
- تنظيم الكود وتحديد مستويات الوصول.

أنواع الوصول للسمات والأساليب:

1. **العامة :(Public)** يمكن الوصول إليها من أي مكان.
2. **المحمية :(Protected)** يُفضل عدم تعديلها خارج الفئة أو الفئات المشتقة (يُشار إليها ببادئة _).
3. **الخاصة :(Private)** لا يمكن الوصول إليها مباشرة من خارج الفئة (يُشار إليها ببادئتين __).

مثال على التغليف:

```
class Account:
    def __init__(self, balance):
        self.__balance = balance  # سمة خاصة

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
```

```
        return self.__balance

account = Account(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

---

12. المُعالجات (Getters) والمُعدّلات(Setters)

مفهوم Getters وSetters

- **المُعالج :(Getter)** أسلوب يُستخدم لاسترجاع قيمة سمة خاصة.
- **المُعدّل :(Setter)** أسلوب يُستخدم لتعيين أو تغيير قيمة سمة خاصة.

تُستخدم هذه الأساليب لضمان الوصول الآمن إلى السمات والتحقق من القيم المدخلة.

مثال توضيحي:

```
class Member:
    def __init__(self, name):
        self.__name = name  # سمة خاصة

    def get_name(self):  # Getter
        return self.__name

    def set_name(self, new_name):  # Setter
        self.__name = new_name

one = Member("Ahmed")
# محاولة الوصول المباشر (غير مستحسنة) – يمكن الوصول للسمة باستخدام one._Member__name
print(one._Member__name)   # Output: Ahmed (أو قد يكون تم تغييرها)
print(one.get_name())      # استخدام Getter لاسترجاع القيمة
one.set_name("Abbas")      # استخدام Setter لتغيير القيمة
print(one.get_name())      # Output: Abbas
```

---

13. الزخرفة@property

مفهوم@property

تُستخدم الزخرفة @property لتحويل أسلوب (Method) إلى سمة (Attribute) يمكن الوصول إليها دون الحاجة لاستدعاء الدالة كطريقة، مما يُحسّن من قابلية قراءة الكود ويُخفي منطق الحساب.

مثال توضيحي:

```
class Member:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    @property
    def age_in_days(self):
        return self.age * 365

one = Member("Ahmed", 40)
print(one.name)         # Output: Ahmed
print(one.age)          # Output: 40
print(one.age_in_days)  # Output: 14600 (يحسب العمر بالأيام)
```

الفوائد:

- تحسين قابلية القراءة.
- إخفاء منطق الحساب.
- إضافة مرونة للوصول إلى البيانات.

---

14. الفئات الأساسية المجردة(Abstract Base Classes - ABC)

مفهوم الفئات الأساسية المجردة

الفئة الأساسية المجردة هي فئة تحتوي على أساليب مجردة (abstract methods) لا تحتوي على تنفيذ، وتُلزم الفئات المشتقة بتنفيذ تلك الأساليب.
يُستخدم هذا المفهوم لفرض بنية محددة على الفئات المشتقة.

خطوات الاستخدام:

1. **استيراد المكتبات اللازمة:**

```
from abc import ABCMeta, abstractmethod
```

2. **تعريف الفئة الأساسية المجردة:**

```
class Programming(metaclass=ABCMeta):
    @abstractmethod
    def has_oop(self):
        pass

    @abstractmethod
    def has_name(self):
        pass
```

3. **تعريف الفئات المشتقة:**

```
class Python(Programming):
    def has_oop(self):
        return "Yes"

class Pascal(Programming):
```

```
    def has_oop(self):
        return "No"
    def has_name(self):
        return "Pascal"
```

4. **إنشاء الكائنات واستخدام الأساليب:**

```
one = Pascal()
print(one.has_oop())   # Output: No
print(one.has_name())  # Output: Pascal
```

الفوائد:

- فرض بنية محددة.
- تحسين التنظيم الهيكلي للكود.
- إخفاء التفاصيل الداخلية.

---

15. التجريد(Abstraction)

مفهوم التجريد بالعموم

التجريد هو تقليل التعقيد عن طريق إخفاء التفاصيل غير الضرورية والتركيز على الواجهة أو الوظيفة الأساسية فقط.
يساهم التجريد في:

- تبسيط الاستخدام.
- تحسين قابلية الصيانة دون التأثير على المستخدم النهائي.
- يُساعد في إخفاء التفاصيل الداخلية وتعزيز وضوح الهيكل البرمجي.

أدوات تحقيق التجريد:

1. **الفئات المجردة:(Abstract Classes)**

- فئة لا يمكن إنشاء كائن منها مباشرة.
- تُستخدم لتحديد سلوك مشترك وتوفير إطار عمل للفئات الفرعية.
- تحتوي على دوال مجردة (بدون تنفيذ) ودوال عادية (بالتنفيذ).

2. **الدوال المجردة:(Abstract Methods)**

- دوال تُعلن داخل الفئة المجردة بدون تنفيذ.
- تُجبر الفئات الفرعية على إعادة تعريفها لتوفير التنفيذ المناسب.

3. **الواجهات:(Interfaces)**

- تحدد مجموعة من الدوال التي يجب على الفئات التي تطبقها تنفيذها.
- تُستخدم لتحقيق التجريد دون تحديد تفاصيل التنفيذ.

مثال توضيحي باستخدام الفئات الأساسية المجردة:

```
from abc import ABC, abstractmethod

# تعريف الواجهات باستخدام الفئات المجردة
class Moveable(ABC):
@    abstractmethod
    def move(self):
        pass

class Steerable(ABC):
@    abstractmethod
    def steer(self, direction):
        pass

class Fuelable(ABC):
@    abstractmethod
    def refuel(self, amount):
        pass

# الفئة المجردة Vehicle# تجمع الواجهات السابقة
class Vehicle(Moveable, Steerable, Fuelable):
@    abstractmethod
    def move(self):
        pass

@    abstractmethod
    def steer(self, direction):
```

```
        pass

@    abstractmethod
    def refuel(self, amount):
        pass

    # دالة عادية لجميع المركبات
    def show_status(self):
        print("Vehicle status: operational")

# الفئة Car# تنفذ جميع الدوال المجردة المعرفة في Vehicle
class Car(Vehicle):
    def move(self):
        print("The car moves on roads.")

    def steer(self, direction):
        print(f"The car steers to the {direction}.")

    def refuel(self, amount):
        print(f"The car is refueled with {amount} liters.")

# إنشاء كائن من الفئة Car# واستخدامه
car = Car()
car.move() # الناتج: The car moves on roads.
car.steer("left") # الناتج:  The car steers to the left.
car.refuel(50) # الناتج: The car is refueled with 50 liters.
car.show_status() # الناتج: Vehicle status: operational
```

بهذا المثال نكون قد غطينا أدوات تحقيق التجريد:

- **الفئات المجردة** لتحديد إطار عمل مشترك.
- **الدوال المجردة** التي تُجبر الفئات المشتقة على تنفيذ السلوك المطلوب.
- **الواجهات** لتعريف مجموعة من الدوال التي يجب تنفيذها، مما يُسهم في بناء أنظمة برمجية مبسطة وسهلة الصيانة.

---

16. المفاهيم الأساسية والخلاصة

المفاهيم الأساسية فيOOP:

1. **الفئة:(Class)**
القالب أو المخطط الذي يُستخدم لإنشاء الكائنات ويحدد السمات والأساليب.
2. **الكائن:(Object)**
مثيل للفئة يحمل بياناته وسلوكياته.
3. **السمات:(Attributes)**
المتغيرات التي تحتفظ بالبيانات الخاصة بالكائن، ويمكن أن تكون عامة، محمية، أو خاصة.
4. **الأساليب:(Methods)**
الدوال المعرفة داخل الفئة التي تعمل على بيانات الكائن.
5. **الوراثة:(Inheritance)**
آلية تمكن الفئات المشتقة من استيراث الخصائص والسلوكيات من الفئات الأصلية.
6. **التعددية الشكلية:(Polymorphism)**
إمكانية استخدام نفس الاسم للأسلوب مع تنفيذات مختلفة بحسب الكائن.
7. **التغليف:(Encapsulation)**
إخفاء التفاصيل الداخلية وتحديد كيفية الوصول إلى البيانات.
8. **الدوال السحرية:(Magic Methods)**
دوال خاصة تبدأ وتنتهي بشرطتين (مثل __init__, __str__, __len__) تُستدعى تلقائياً في حالات معينة.
9. **التجريد:(Abstraction)**
إخفاء التعقيدات الداخلية والتركيز على الواجهة الأساسية.
10. **الفئات الأساسية المجردة:(Abstract Base Classes)**
تُستخدم لفرض بنية معينة على الفئات المشتقة.

الخلاصة:

البرمجة كائنية التوجه (OOP) هي طريقة لكتابة الشيفرة البرمجية تُسهّل قراءتها وصيانتها، رغم أنها قد تُضيف تعقيدًا في البرامج البسيطة، إلا أنها تُعد الخيار الأمثل للمشاريع المتوسطة والكبيرة. وتدعمها معظم لغات البرمجة الحديثة. فيما يلي تلخيص لأهم أربعة مفاهيم تشكّل أساس هذا النمط من البرمجة:

1. **الوراثة (Inheritance) :**
تُتيح إنشاء فئات جديدة مستندة إلى فئات موجودة مسبقًا، مما يعزز إعادة استخدام الكود وتوسيع الوظائف.
2. **التجريد (Abstraction) :**
يساعد في تبسيط التعامل مع الكائنات عن طريق إخفاء التفاصيل المعقدة، مما يتيح للمبرمجين التركيز على الوظائف الأساسية دون الانشغال بالتفاصيل الداخلية.
3. **التعددية الشكلية (Polymorphism) :**
تسمح باستخدام نفس الواجهات أو الأساليب على أنواع متعددة من الكائنات، بحيث يمكن لكل كائن أن ينفذ تلك الأساليب بطريقة خاصة به.
4. **التغليف (Encapsulation) :**
يوفر آلية لحماية البيانات من التلاعب غير المصرح به، من خلال تجميع البيانات والأساليب التي تتعامل معها داخل وحدة واحدة (الفئة) وتحديد مستويات الوصول إليها.

هذه المفاهيم تساهم جميعها في بناء برمجيات أكثر تنظيمًا وقابلية للصيانة والتطوير.