برمجة بايثون للمبتدئين

هذا الكتاب مصمم للطلاب والمبتدئين الذين يرغبون في تعلم برمجة بايثون من الصفر. يقدم مقدمة شاملة ومبسطة حول أساسيات لغة البرمجة بايثون، بدءًا من المفاهيم الأساسية مثل أنواع البيانات، والتحكم في التدفق، وحتى تقنيات البرمجة المتقدمة مثل الكائنات والبرمجة الشيئية. يناسب هذا الكتاب جميع من يريد استكشاف عالم البرمجة باستخدام لغة بايثون، سواء كان ذلك لأغراض تعليمية أو تطوير المشاريع الصغيرة. بأسلوب سلس ومنظم، يساعد الكتاب القارئ في بناء أساس قوي في البرمجة باستخدام بايثون.

إعداد : علاء عرابي

GAUAB.com

الفصل الأول: مقدمة إلى بايثون

- ما هي لغة بايثون؟ ولماذا هي شائعة؟
- كيفية تثبيت بايثون وإعداد بيئة التطوير (Python IDE).
- أول برنامج بلغة بايثون: "مرحبًا بالعالم" (Hello World).

الفصل الثاني: الأساسيات

- المتغيرات وأنواع البيانات الأساسية (النصوص، الأرقام، القوائم، المجموعات).
- المدخلات والمخرجات (input/output).
- العمليات الحسابية والمنطقية.

الفصل الثالث: التحكم في تدفق البرنامج

- الجمل الشرطية (if-else).
- الحلقات التكرارية (for، while).
- العبارات الشرطية المختصرة (ternary operators).

الفصل الرابع: الدوال (Functions)

- تعريف الدوال واستخدامها.
- الدوال المضمنة (Built-in Functions).

الفصل الخامس: العمل مع البيانات

- القوائم (Lists) والمصفوفات (Tuples).
- القواميس (Dictionaries).
- المجموعات (Sets).
- التلاعب بالنصوص (Strings).

الفصل السادس: البرمجة الكائنية (OOP)

- المفاهيم الأساسية (Class، Object).
- إنشاء الأصناف والخصائص والدوال داخلها.
- الوراثة (Inheritance) والتعددية الشكلية (Polymorphism).

الفصل السابع: التعامل مع الملفات

- قراءة وكتابة الملفات النصية.
- التعامل مع الملفات بصيغة CSV.
- العمل مع ملفات JSON.

الفصل الثامن: التعامل مع الأخطاء والاستثناءات

- أهمية إدارة الأخطاء في البرمجة.
- استخدام try-except لإدارة الاستثناءات.

الفصل التاسع: المكتبات والأدوات المضمنة في بايثون

- التعريف بالمكتبات القياسية مثل `random`، `math`، و`datetime`.
- مقدمة إلى مكتبات شائعة مثل NumPy و Pandas.

الفصل العاشر: مشاريع تطبيقية صغيرة

- برنامج آلة حاسبة.
- لعبة بسيطة (مثل لعبة التخمين).
- أداة لتحليل النصوص (مثل حساب الكلمات).
- العمل مع ملفات CSV وتحليل بياناتها.

الفصل الحادي عشر: استخدام MySQL في بايثون

ما هي لغة بايثون؟

بايثون هي لغة برمجة عالية المستوى و سهلة التعلم والاستخدام، تم تطويرها في أواخر الثمانينيات وأُطلق أول إصدار منها عام 1991 بواسطة المبرمج الهولندي **غيدو فان روسوم**. تُعتبر لغة برمجة متعددة الاستخدامات، مما يعني أنه يمكن استخدامها في العديد من المجالات مثل تطوير الويب، تحليل البيانات، الذكاء الاصطناعي، التعلم الآلي، تطبيقات سطح المكتب، وغيرها.

لماذا لغة بايثون شائعة؟

تتمتع بايثون بشعبية هائلة حول العالم، وتأتي في صدارة لغات البرمجة المفضلة للعديد من المبرمجين والمطورين. هناك العديد من الأسباب التي تفسر شعبيتها:

1. سهولة التعلم والاستخدام

- تعتمد بايثون على تركيبة لغوية بسيطة تشبه اللغة الإنجليزية، مما يجعلها خيارًا مثاليًا للمبتدئين.
- الشيفرة البرمجية واضحة وقابلة للقراءة، حتى بالنسبة للأشخاص الذين ليس لديهم خبرة برمجية سابقة.

2. تعدد الاستخدامات

- يمكن استخدام بايثون في مجموعة واسعة من التطبيقات، مثل تطوير الويب (Django وFlask)، تحليل البيانات (Pandas وNumPy)، التعلم الآلي (TensorFlow وScikit-learn)، والبرمجة النصية.

3. وجود مكتبات وأطر عمل قوية

- تحتوي بايثون على مكتبة قياسية غنية ومكتبات إضافية تُسهّل تنفيذ المهام المختلفة، مثل معالجة البيانات، التعامل مع الشبكات، تطوير الألعاب، وغيرها.

4. مجتمع ضخم وداعم

- مجتمع بايثون من أكبر المجتمعات التقنية على الإنترنت. إذا واجهتك مشكلة، ستجد دائمًا من يقدم لك المساعدة، سواء عبر المنتديات أو مواقع مثل Stack Overflow.

5. مجانية ومفتوحة المصدر

- بايثون مفتوحة المصدر، مما يعني أنها مجانية تمامًا وتتيح للمطورين التعديل عليها حسب احتياجاتهم.

6. التكامل مع لغات وأدوات أخرى

- تتكامل بايثون بسهولة مع لغات البرمجة الأخرى مثل C وC++ وJava، مما يجعلها خيارًا مرنًا للمطورين.

7. مناسبة لتطبيقات المستقبل

- تُستخدم بايثون في العديد من المجالات المستقبلية مثل الذكاء الاصطناعي والتعلم الآلي وإنترنت الأشياء (IoT)، مما يزيد من أهميتها في السوق التكنولوجي.

أهم المجالات التي تُستخدم فيها بايثون

- **تطوير الويب:** باستخدام أطر مثل Django وFlask.
- **تحليل البيانات:** باستخدام مكتبات مثل Pandas وNumPy.
- **التعلم الآلي:** باستخدام مكتبات مثل TensorFlow وScikit-learn.
- **أتمتة المهام:** كتابة سكربتات لأداء المهام المتكررة.
- **تطوير الألعاب:** باستخدام مكتبات مثل Pygame.

كيفية تثبيت بايثون وإعداد بيئة التطوير (Python IDE)

الخطوة الأولى: التحقق من تثبيت بايثون على النظام

- في معظم أنظمة التشغيل (مثل Linux وmacOS)، قد يكون بايثون مثبتًا مسبقًا.

للتحقق من وجود بايثون، افتح الطرفية (Terminal) أو موجه الأوامر (Command Prompt) واكتب:

```
python --version
```

أو

```
python3 --version
```

- إذا ظهر رقم الإصدار، فهذا يعني أن بايثون مثبت بالفعل.

الخطوة الثانية: تنزيل بايثون وتثبيته

1. تنزيل بايثون

- انتقل إلى الموقع الرسمي لبايثون: https://www.python.org/.
- اختر النسخة المناسبة لنظام التشغيل الخاص بك (Windows أو macOS أو Linux).

2. تثبيت بايثون على Windows

- قم بتنزيل ملف التثبيت (.exe).
- أثناء التثبيت:
  - **حدد خيار "Add Python to PATH"**.
  - اختر تثبيتًا مخصصًا إذا أردت تغيير المسار الافتراضي.

بعد الانتهاء، تحقق من التثبيت بكتابة:

```
python --version
```

3. تثبيت بايثون على macOS

يمكنك استخدام Homebrew لتثبيت بايثون عبر الطرفية:

```
brew install python
```

4. تثبيت بايثون على Linux

- يمكن تثبيت بايثون باستخدام مدير الحزم الخاص بتوزيعتك. على سبيل المثال:

Ubuntu/Debian:

```
sudo apt update
sudo apt install python3
```

Fedora:

```
sudo dnf install python3
```

الخطوة الثالثة: تثبيت بيئة التطوير (Python IDE)

للكتابة وتنفيذ الشيفرة البرمجية بسهولة، يفضل استخدام بيئة تطوير متكاملة (IDE). إليك بعض الخيارات الشائعة:

1. PyCharm

- **الوصف:** واحد من أفضل IDEs لبايثون، يوفر أدوات ذكية لتحرير الشيفرة، وتصحيح الأخطاء، وإدارة المشاريع.
- **التنزيل:** https://www.jetbrains.com/pycharm/
- **التثبيت:** قم بتنزيل النسخة المناسبة لنظام التشغيل الخاص بك واتبع تعليمات التثبيت.

2. Visual Studio Code (VS Code)

- **الوصف:** محرر شيفرة خفيف ومجاني يدعم إضافات مخصصة مثل Python Extension.
- **التنزيل:** https://code.visualstudio.com/
- **الإعداد:** بعد تثبيت VS Code، قم بتثبيت الإضافة الخاصة ببايثون من Marketplace.

3. Jupyter Notebook

- **الوصف:** مناسب لتحليل البيانات والتعلم الآلي. يتيح لك كتابة الشيفرة وتنفيذها في واجهة متصفح.

التثبيت:

```
pip install notebook
jupyter notebook
```

4. IDLE (مضمنة مع بايثون)

- **الوصف:** محرر خفيف مدمج مع تثبيت بايثون، مثالي للمبتدئين.
- **الاستخدام:** ابحث عن "IDLE" بعد تثبيت بايثون وشغله.

الخطوة الرابعة: إعداد مدير الحزم Pip

- Pip هو أداة لتثبيت مكتبات بايثون.

للتحقق من وجوده:

```
pip --version
```

لتثبيت مكتبة، استخدم:

```
pip install library_name
```

مثال:

```
pip install numpy
```

الخطوة الخامسة: كتابة أول برنامج "Hello, World"

1. افتح الـ IDE أو الطرفية.

اكتب الكود التالي واحفظه في ملف باسم `hello.py`:

```
print("Hello, World!")
```

نفذ البرنامج:

```
python hello.py
```

الخطوة السادسة: نصائح إضافية

- استخدم Git لإدارة المشاريع البرمجية ومشاركة الشيفرة.
- استكشف المكتبات الشائعة مثل Pandas، NumPy، وMatplotlib لتعزيز مهاراتك.

- مارس البرمجة يوميًا لتطوير مستواك في بايثون.

إذا كنت بحاجة إلى مساعدة إضافية، يمكنني توضيح أي خطوة أو مساعدتك في إعداد بيئة تطويرك!

أول برنامج بلغة بايثون: "مرحبًا بالعالم" (Hello World)

يُعتبر برنامج "مرحبًا بالعالم" أبسط برنامج يمكن كتابته في أي لغة برمجة، وهو الخطوة الأولى لتعلم أساسيات البرمجة. في بايثون، يتميز هذا البرنامج بكونه قصيرًا وبسيطًا للغاية.

خطوات كتابة برنامج "مرحبًا بالعالم" بلغة بايثون

1. تثبيت بايثون (إذا لم يكن مثبتًا بالفعل)

- تأكد من أنك قمت بتثبيت بايثون باتباع التعليمات المذكورة هنا.

تحقق من التثبيت عن طريق كتابة الأمر التالي في الطرفية أو موجه الأوامر:

```
python --version
```

أو

```
python3 --version
```

2. كتابة البرنامج

1. افتح محرر نصوص بسيط مثل **Notepad** (على ويندوز) أو **TextEdit** (على macOS) أو أي محرر مفضل لديك مثل **Visual Studio Code** أو **PyCharm**.

اكتب السطر التالي:

```
print("مرحبًا بالعالم!")
```

2. احفظ الملف باسم **hello.py**. تأكد من أن الامتداد هو `.py` ليتم التعرف عليه كملف بايثون.

3. تشغيل البرنامج

- افتح الطرفية (Terminal) أو موجه الأوامر (Command Prompt).

انتقل إلى المجلد الذي حفظت فيه الملف. على سبيل المثال:

```
cd path/to/your/file
```

نفذ البرنامج باستخدام الأمر:

```
python hello.py
```

أو

```
python3 hello.py
```

4. النتيجة

عند تنفيذ البرنامج، ستظهر الجملة التالية على الشاشة:

```
مرحبًا بالعالم!
```

شرح الكود

```
print("مرحبًا بالعالم!")
```

- **print:** دالة مدمجة في بايثون تُستخدم لعرض النصوص أو القيم على الشاشة.
- **"مرحبًا بالعالم!":** النص المراد عرضه، ويجب أن يكون محاطًا بعلامتي اقتباس مزدوجتين (") أو مفردتين (').

تطوير البرنامج

بمجرد فهمك لهذا البرنامج البسيط، يمكنك تجربة إضافة المزيد:

عرض أكثر من جملة:

```
print("مرحبًا بالعالم!")
print("أنا أتعلم بايثون.")
```

قبول مدخلات من المستخدم:

```
name = input("ما اسمك؟ ")
print(f"مرحبًا يا {name}!")
```

لماذا "مرحبًا بالعالم"؟

هذا البرنامج البسيط يهدف إلى تعريفك بأبسط قواعد اللغة البرمجية وكيفية تشغيل البرامج، وهو أساس كل رحلة في تعلم البرمجة.

المتغيرات وأنواع البيانات الأساسية في بايثون

---

1. ما هو المتغير؟

- المتغير هو اسم يتم استخدامه لتخزين قيمة معينة في ذاكرة الجهاز.
- يمكن تغيير قيمة المتغير أثناء تنفيذ البرنامج.

يتم إنشاء المتغير في بايثون بمجرد إسناد قيمة له.

```
age = 25
name = "Ahmed"
is_student = True
```

---

2. قواعد تسمية المتغيرات

- يجب أن يبدأ المتغير بحرف أو شرطة سفلية (_) ولا يمكن أن يبدأ برقم.

```
✅ name | ✅ _age | ❌ 1st_name
```

- يمكن أن يحتوي على حروف، أرقام، وشرطات سفلية فقط.

```
✅ user_name | ❌ user-name
```

- لغة بايثون حساسة لحالة الأحرف (Case-Sensitive):
  - `Name` و`name` متغيران مختلفان.

---

3. أنواع البيانات الأساسية في بايثون

أ. النصوص (Strings)

- تستخدم النصوص لتخزين النصوص أو الكلمات.

يتم كتابة النصوص بين علامات اقتباس مفردة ' أو مزدوجة ".

```
name = "Ahmed"
greeting = 'Hello, world!'
```

```
print(name)
```

- **عمليات على النصوص:**

دمج النصوص:

```
full_name = "Ahmed" + " " + "Ali"
print(full_name)  # Ahmed Ali
```

طول النص:

```
text = "Hello"
print(len(text))  # 5
```

ب. الأرقام (Numbers)

- هناك نوعان رئيسيان من الأرقام:

الأعداد الصحيحة (Integers):
أعداد بدون كسور.

```
x = 10
y = -5
```

الأعداد العشرية (Floats):
أعداد تحتوي على كسور.

```
pi = 3.14
height = 1.75
```

عمليات رياضية:

```
a = 10
b = 3
print(a + b)  # الجمع
print(a - b)  # الطرح
print(a * b)  # الضرب
print(a / b)  # القسمة
print(a // b) # القسمة مع تجاهل الكسر
print(a % b)  # باقي القسمة
print(a ** b) # الأس
```

ج. القوائم (Lists)
القوائم هي مجموعة قابلة للتغيير يمكنها تخزين عناصر متعددة من أنواع بيانات مختلفة.

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4]
```

```
mixed = [1, "Ahmed", True]
print(fruits[0])  # apple
```

- **عمليات على القوائم:**

إضافة عنصر:

```
fruits.append("orange")
print(fruits)  # ["apple", "banana", "cherry", "orange"]
```

إزالة عنصر:

```
fruits.remove("banana")
print(fruits)  # ["apple", "cherry"]
```

د. المجموعات (Sets)

المجموعات هي نوع من البيانات يخزن عناصر غير مكررة وغير مرتبة.

```
my_set = {1, 2, 3, 3, 4}
print(my_set)  # {1, 2, 3, 4}
```

- **عمليات على المجموعات:**

إضافة عنصر:

```
my_set.add(5)
print(my_set)  # {1, 2, 3, 4, 5}
```

التحقق من وجود عنصر:

```
print(3 in my_set)  # True
```

---

4. التحويل بين أنواع البيانات

يمكنك تحويل نوع بيانات إلى آخر باستخدام دوال التحويل:

```
x = 10  # عدد صحيح
y = str(x)  # تحويل إلى نص
print(y)  # "10"

z = "25"
w = int(z)  # تحويل نص إلى عدد صحيح
print(w)  # 25
```

أمثلة عملية

```
# مثال على استخدام المتغيرات وأنواع البيانات
name = "Sara"
age = 22
hobbies = ["reading", "traveling", "coding"]

print(f"My name is {name}, I am {age} years old, and I love
{hobbies[2]}.")
```

المدخلات والمخرجات (Input/Output) في بايثون

---

1. المدخلات (Input)

- **المفهوم:**

يمكن للمستخدم إدخال البيانات أثناء تشغيل البرنامج باستخدام دالة `input()`.

الاستخدام:

```
name = input("ما هو اسمك؟ ")
print(f"مرحبًا، {name}!")
```

- الدالة `input()` دائمًا تعيد نصًا (String).
- يمكن تحويل المدخلات إلى نوع آخر مثل الأرقام باستخدام دوال التحويل (مثل `int()` أو `float()`).

مثال لتحويل المدخلات:

```
age = input("كم عمرك؟ ")
age = int(age)  # تحويل النص إلى عدد صحيح
print(f"هو عمرك {age} سنوات.")
```

---

2. المخرجات (Output)

- **المفهوم:**
  يتم عرض النتائج باستخدام دالة `print()`.

الاستخدام الأساسي:

```
print("مرحبًا بالعالم!")
```

- **الجمع بين النصوص والقيم باستخدام تنسيق النصوص:**

الطريقة التقليدية:

```
name = "علي"
print("مرحبًا، " + name + "!")
```

استخدام f-strings (أفضل طريقة):

```
name = "علي"
age = 25
print(f"مرحبًا، {name}! عمرك {age} سنوات.")
```

العمليات الحسابية والمنطقية

1. العمليات الحسابية

العمليات الأساسية:

```
a = 10
b = 3
print(a + b)  # الجمع
print(a - b)  # الطرح
print(a * b)  # الضرب
print(a / b)  # القسمة
print(a // b) # القسمة مع تجاهل الكسر
print(a % b)  # باقي القسمة
print(a ** b) # الأس
```

2. العمليات المنطقية (Logical Operations)

- **القيم المنطقية:**
  - `True` و `False` هما القيمتان المنطقيتان الرئيسيتان.
  - يتم استخدامهما مع العمليات المنطقية.
- **العمليات المنطقية الأساسية:**

العملية	الرمز	المثال
أكبر من	`>`	`5 > 3` → True
أصغر من	`<`	`5 < 3` → False
يساوي	`==`	`5 == 5` → True
لا يساوي	`!=`	`5 != 3` → True
أكبر من أو يساوي	`>=`	`5 >= 5` → True

أصغر من أو يساوي `<=` `3 <= 5` → True

العمليات المنطقية المركبة:

و (AND): جميع الشروط يجب أن تكون صحيحة.

```
print(5 > 3 and 4 > 2)  # True
print(5 > 3 and 4 < 2)  # False
```

أو (OR): يكفي أن يكون شرط واحد صحيحًا.

```
print(5 > 3 or 4 < 2)  # True
print(3 > 5 or 4 < 2)  # False
```

ليس (NOT): يعكس القيمة المنطقية.

```
print(not True)  # False
print(not False) # True
```

مثال عملي يدمج المدخلات والمخرجات مع العمليات الحسابية والمنطقية

```
# برنامج لحساب ما إذا كان المستخدم قد بلغ سن الرشد
age = int(input("كم عمرك؟ "))

if age >= 18:
    print("أنت بالغ!")
else:
    print("أنت قاصر!")
```

تطبيق عملي: الآلة الحاسبة البسيطة

```
# برنامج بسيط لتنفيذ العمليات الحسابية
print("مرحبًا بك في الآلة الحاسبة!")
a = float(input("أدخل الرقم الأول: "))
b = float(input("أدخل الرقم الثاني: "))
operation = input("اختر العملية (+, -, *, /): ")

if operation == "+":
    print(f"النتيجة: {a + b}")
elif operation == "-":
    print(f"النتيجة: {a - b}")
elif operation == "*":
    print(f"النتيجة: {a * b}")
elif operation == "/":
    if b != 0:
        print(f"النتيجة: {a / b}")
    else:
        print("خطأ: القسمة على صفر غير مسموحة!")
else:
    print("عملية غير صحيحة!")
```

الجمل الشرطية (if-else) في بايثون

---

1. ما هي الجمل الشرطية؟

الجمل الشرطية تُستخدم لتنفيذ مجموعة معينة من التعليمات إذا تحقق شرط معين. إذا لم يتحقق الشرط، يمكن تنفيذ تعليمات أخرى.

---

2. التركيب الأساسي لجملة if

```
if شرط:
    # تعليمات تُنفذ إذا تحقق الشرط
```

- **مثال:**

```
age = 18

if age >= 18:
    print("أنت بالغ!")
```

```

```

---

3. استخدام if-else

```
if :شرط
# تعليمات تُنفذ إذا تحقق الشرط
else:
    # تعليمات تُنفذ إذا لم يتحقق الشرط
```

- **مثال:**

```
age = 16

if age >= 18:
    print("أنت بالغ!")
else:
    print("أنت قاصر!")
```

---

4. استخدام if-elif-else

- **if**: للتحقق من الشرط الأول.
- **elif**: للتحقق من شروط إضافية.
- **else**: لتنفيذ التعليمات إذا لم يتحقق أي شرط.

```
if :1شرط
1# تعليمات تُنفذ إذا تحقق الشرط
elif :2شرط
2# تعليمات تُنفذ إذا تحقق الشرط
else:
    # تعليمات تُنفذ إذا لم يتحقق أي شرط
```

- **مثال:**

```
score = 75

if score >= 90:
    print("ممتاز!")
elif score >= 75:
```

```
    print("جيد جدًا!")
elif score >= 50:
    print("مقبول.")
else:
    print("راسب.")
```

---

5. الجمل الشرطية المتداخلة (Nested If)

يمكن وضع جملة شرطية داخل جملة شرطية أخرى.

```
مثال:

age = 20
nationality = "مصري"

if age >= 18:
    if nationality == "مصري":
        print("يمكنك التصويت في الانتخابات.")
    else:
        print("لا يمكنك التصويت لأنك لست مواطنًا.")
else:
    print("لا يمكنك التصويت لأنك قاصر.")
```

---

6. اختصار جملة (if-else (Ternary Operator

إذا كانت جملة if-else بسيطة، يمكن كتابتها في سطر واحد.

- **الصيغة:**

```
تعليمات_إذا_تحقق_الشرط if شرط else تعليمات_إذا_لم_يتحقق_الشرط
```

- **مثال:**

```
age = 20
status = "بالغ" if age >= 18 else "قاصر"
print(status)
```

---

7. أمثلة عملية على الجمل الشرطية

مثال 1: معرفة إذا كان العدد زوجيًا أو فرديًا

```
number = int(input("أدخل عددًا: "))

if number % 2 == 0:
    print("العدد زوجي.")
else:
    print("العدد فردي.")
```

مثال 2: حساب الخصم على المنتجات

```
price = float(input("أدخل سعر المنتج: "))

if price > 100:
    discount = 20  # 20 خصم%
else:
    discount = 10  # 10 خصم%

final_price = price - (price * discount / 100)
print(f"السعر النهائي بعد الخصم هو: {final_price}")
```

مثال 3: التحقق من كلمة المرور

```
password = input("أدخل كلمة المرور: ")

if password == "12345":
    print("تم تسجيل الدخول بنجاح!")
else:
    print("كلمة المرور خاطئة!")
```

الحلقات التكرارية (For و While) في بايثون

---

1. حلقة التكرار For Loop

1.1 التركيب الأساسي

```
for متغير in :مجموعة
تعليمات يتم تنفيذها لكل عنصر في المجموعة #
```

- **مثال:**

```
for i in range(5):
    print(i)
```

```
نطاق الأرقام:
range(n) ينتج أرقامًا من 0 إلى n-1.
range(start, end) ينتج أرقامًا من start إلى end-1.
```

مثال باستخدام النطاق:

```
for j in range(3, 6):
    print(j)
```

1.2 استخدامات الحلقة for

مثال: طباعة عناصر قائمة:

```
fruits = ['برتقال' ,'موز' ,'تفاح']
for fruit in fruits:
    print(fruit)
```

مثال: طباعة الأرقام المضاعفة:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num * 2)
```

---

2. حلقة التكرار While Loop

2.1 التركيب الأساسي

```
while شرط:
    # تعليمات يتم تنفيذها طالما الشرط صحيح
```

- **مثال:**

```
count = 0
while count < 5:
    print(count)
    count += 1
```

- **تعديل العدادات داخل الحلقة:**
  - `count += 1` يضيف 1 إلى المتغير `count`.
  - `count -= 1` ينقص 1 من المتغير `count`.

2.2 استخدامات الحلقة while

مثال: طباعة الأرقام حتى 10:

```
num = 1
while num <= 10:
    print(num)
    num += 1
```

مثال: طلب إدخال كلمة المرور حتى تكون صحيحة:

```
password = ""
while password != "12345":
    password = input("أدخل كلمة المرور: ")
print("تم تسجيل الدخول بنجاح!")
```

---

3. الحلقات التكرارية مع الجمل الشرطية
مثال: طباعة الأعداد الزوجية حتى 10:

```
num = 0
while num <= 10:
    if num % 2 == 0:
        print(num)
    num += 1
```

---

4. تكرار داخل جمل التكرار (Nested Loops)
مثال: طباعة جميع الأزواج الممكنة من الأرقام:

```
for i in range(3):
    for j in range(3):
        print(f"{i}, {j}")
```

---

5. استخدام break و continue

- **break**: لإنهاء الحلقة فورًا عندما يتحقق شرط معين.
- **continue**: لتجاوز الجولة الحالية من الحلقة عندما يتحقق شرط معين.

مثال break:

```
count = 0
while count < 10:
    if count == 5:
        break
    print(count)
    count += 1
```

مثال continue:

```python
for num in range(10):
    if num % 2 == 0:
        continue
    print(num)
```

---

6. استخدام حلقات مع النصوص
مثال: طباعة كل حرف من النصوص:

```python
text = "Python"
for char in text:
    print(char)
```

العبارات الشرطية المختصرة (Ternary Operators) في بايثون

---

1. تعريف العبارات الشرطية المختصرة (Ternary Operators)

العبارة الشرطية المختصرة هي طريقة قصيرة وفعالة لكتابة جمل الشرط (`if-else`) في سطر واحد.

---

2. الصيغة العامة للعبارات الشرطية المختصرة

```
تعليمات_إذا_تحقق_الشرط if شرط else تعليمات_إذا_لم_يتحقق_الشرط
```

---

3. أمثلة على العبارات الشرطية المختصرة

3.1 مثال بسيط

```python
age = 20
```

```
status = "بالغ" if age >= 18 else "قاصر"
print(status)  # سيتم طباعة "بالغ"
```

3.2 استخدام العبارات الشرطية مع القيم الرقمية

```
num1 = 15
num2 = 10
max_num = num1 if num1 > num2 else num2
print(max_num)  # سيتم طباعة "15"
```

3.3 العبارات الشرطية مع النصوص

```
username = "admin"
role = "مشرف" if username == "admin" else "مستخدم عادي"
print(role)  # سيتم طباعة "مشرف"
```

3.4 استخدام العبارات الشرطية مع القيم المنطقية

```
is_logged_in = True
message = "مرحبًا بك!" if is_logged_in else "يرجى تسجيل الدخول"
print(message)  # سيتم طباعة "مرحبًا بك!"
```

---

4. العبارات الشرطية داخل الدوال (Ternary Operators in Functions)

يمكن استخدام العبارات الشرطية المختصرة داخل الدوال:

```
def get_discount(price):
    discount = 20 if price > 100 else 10
    return discount

print(get_discount(150))  # سيتم طباعة "20"
```

---

5. فوائد العبارات الشرطية المختصرة

- **إيجاز الكود**: توفر طريقة فعالة لكتابة جمل الشرط في سطر واحد بدلاً من استخدام جمل `if-else` متعددة.
- **وضوح القراءة**: إذا تم استخدامها بشكل مناسب، تجعل الكود أكثر وضوحًا وسهولة في الفهم.

تعريف الدوال واستخدامها في بايثون

---

1. تعريف الدوال (Functions)

الدالة في بايثون هي كتلة من التعليمات البرمجية التي تؤدي وظيفة معينة وتُستخدم لإعادة تنظيم الكود وجعله أكثر قابلية لإعادة الاستخدام. يمكن تعريف الدالة بواسطة الكلمة الرئيسية `def`.

---

2. صيغة تعريف الدوال

```
def اسم_الدالة(parameters):
    # التعليمات التي تنفذ داخل الدالة
    return القيم
```

- **اسم الدالة**: هو الاسم الذي يطلق على الدالة.
- **parameters**: (اختياري) هو قيد (argument) يتم تمريره إلى الدالة.
- **القيم**: (اختياري) القيمة التي يتم إرجاعها من الدالة.

---

3. أمثلة على تعريف واستخدام الدوال

3.1 تعريف دالة بدون معلمات (Parameters)

```
def greet():
    return "مرحبًا!"

print(greet())  # سيتم طباعة "مرحبًا!"
```

3.2 تعريف دالة بمتغير (Parameter)

```
def greet_user(name):
```

```
    return f"Hello, {name}!"

print(greet_user("Ali"))  # Output: "Hello, Ali!"
```

3.3 تعريف دالة بإرجاع أكثر من قيمة (Multiple Return Values)

```
def add_and_subtract(a, b):
    return a + b, a - b

result = add_and_subtract(10, 5)
print(result)  # 5 ,15)" سيتم طباعة)"
```

3.4 دالة مع قيم افتراضية (Default Parameters)

```
def greet_user(name="زائر"):
    return f"مرحبًا، {name}!"

print(greet_user())  # زائر ،مرحبًا" سيتم طباعة!"
```

3.5 دالة تستخدم القيد (Arguments) متعدد

```
def calculate_sum(*args):
    return sum(args)

print(calculate_sum(1, 2, 3, 4, 5))  # 15" سيتم طباعة"
```

3.6 الدوال كأحد المعاملات

```
def apply_operation(a, b, operation):
    return operation(a, b)

# تعريف دالة للعمليات الحسابية
def multiply(x, y):
    return x * y

print(apply_operation(4, 5, multiply))  # 20" سيتم طباعة"
```

4. فوائد استخدام الدوال

- **إعادة استخدام الكود**: يمكن استخدام الدوال في أماكن متعددة دون الحاجة إلى كتابة التعليمات البرمجية من جديد.
- **وضوح الكود**: يسهم في تقسيم الكود إلى أجزاء منطقية يسهل التعامل معها.
- **اختبارية**: يسهل اختبار أجزاء معينة من الكود بعد عزلها في دوال منفصلة.

الدوال المضمنة (Built-in Functions) في بايثون

---

1. تعريف الدوال المضمنة

الدوال المضمنة هي دوال تم تضمينها في لغة بايثون، وهي توفر وظائف مختلفة تُستخدم بشكل شائع. هذه الدوال توفر وقت المطور وتساعد في تنفيذ العمليات الأساسية بكفاءة.

---

2. أمثلة على الدوال المضمنة في بايثون

2.1 دالة `print()`

تستخدم لطباعة النصوص أو القيم على الشاشة.

```
print("Hello, World!")  # طباعة النصوص
```

2.2 دالة `len()`

تعيد عدد العناصر داخل كائن مثل نصوص أو قوائم.

```
print(len("Python"))  # 6" سيتم طباعة"
print(len([1, 2, 3, 4, 5]))  # 5" سيتم طباعة"
```

2.3 دالة `max()`

تُستخدم لإيجاد أكبر قيمة بين القيم.

```
print(max(10, 20, 30))  # 30" سيتم طباعة"
```

2.4 دالة `min()`

تُستخدم لإيجاد أصغر قيمة بين القيم.

```
print(min(10, 20, 30))  # 10" سيتم طباعة"
```

2.5 دالة `sum()`

تُستخدم لإيجاد مجموع القيم.

```
print(sum([1, 2, 3, 4, 5]))  # 15" سيتم طباعة"
```

2.6 دالة `abs()`

تُستخدم للحصول على القيمة المطلقة لرقم معين.

```
print(abs(-10))  # 10" سيتم طباعة"
```

2.7 دالة `type()`

تعيد نوع البيانات الخاصة بالقيمة.

```
```

```
print(type(10))  # سيتم طباعة "<class 'int'>"
```

2.8 دالة `sorted()`

تُستخدم لترتيب القيم أو الكائنات.

```
print(sorted([3, 1, 4, 2]))  # 4 ,3 ,2 ,1]" سيتم طباعة]"
```

2.9 دالة `input()`

تُستخدم لطلب إدخال من المستخدم.

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

2.10 دالة `range()`

تُستخدم لإنشاء عدد من الأرقام التي تُستخدم في الحلقات.

```
for i in range(5):
    print(i)  # ستطبع الأرقام من 0 إلى 4
```

---

3. فوائد استخدام الدوال المضمنة

- **سهولة الاستخدام**: توفر الوقت والجهد عند تنفيذ العمليات الشائعة.
- **قابلة لإعادة الاستخدام**: يمكن الاستفادة من الدوال المضمنة دون الحاجة إلى إعادة كتابة الكود من جديد.
- **فعالية الأداء**: تُنفذ بشكل فعال وتدعم العديد من العمليات الأساسية.

القوائم (Lists) والمصفوفات (Tuples) في بايثون

---

1. القوائم (Lists)

القائمة هي نوع من الكائنات القابلة للتغيير في بايثون وتُستخدم لتخزين مجموعة من العناصر ذات الأنواع المختلفة. يمكن تعديل العناصر، مثل الإضافة، الحذف، أو التغيير.

أمثلة على القوائم

```
# تعريف قائمة
my_list = [1, 2, 3, "hello", 4.5]

# الوصول إلى عنصر في القائمة
print(my_list[0])  # "1" سيتم طباعة

# تعديل عنصر في القائمة #
my_list[1] = "world"
print(my_list)  # 1] سيتم طباعة, "world", 3, "hello", 4.5]

# إضافة عنصر إلى القائمة
my_list.append(6)
print(my_list)  # 1] سيتم طباعة, "world", 3, "hello", 4.5, 6]
```

---

2. المصفوفات (Tuples)

المصفوفة هي كائن غير قابل للتغيير (غير قابل للتعديل) في بايثون. تُستخدم لتخزين عناصر ثابتة لا يمكن تعديلها.

أمثلة على المصفوفات

```
# تعريف مصفوفة
my_tuple = (1, 2, 3, "hello")

# الوصول إلى عنصر في المصفوفة
print(my_tuple[0])  # "1" سيتم طباعة

# لا يمكن تعديل العناصر في المصفوفة #
# my_tuple[1] = "world"  # سيعطي خطأ

# يمكن استخدام المصفوفات عند الحاجة إلى بيانات ثابتة #
```

---

3. الفروقات بين القوائم والمصفوفات

- **التغيير**: القوائم قابلة للتغيير (mutable)، بينما المصفوفات غير قابلة للتغيير (immutable).
- **الأداء**: المصفوفات توفر أداءً أفضل عند التعامل مع كميات كبيرة من البيانات لأنها لا تتطلب إنشاء كائن جديد عند تغيير العناصر.
- **الاستخدام**: تُستخدم القوائم للتخزين الديناميكي، بينما تُستخدم المصفوفات عند الحاجة إلى مجموعة ثابتة من العناصر.

---

4. العمليات الشائعة على القوائم والمصفوفات

الإضافة: باستخدام `append()` للقوائم، أو إعادة تعيين مجموعة جديدة في المصفوفات.

```
my_list.append(10)  # قائمة
my_tuple = my_tuple + (10,)  # مصفوفة
```

الحذف: باستخدام `remove()` أو `del` للقوائم.

```
my_list.remove(2)  # حذف العنصر من القائمة
```

```
التكرار: القوائم يمكن تكرارها باستخدام الضرب (*).


new_list = my_list * 2  # سيتم تكرار القائمة مرتين
```

الدمج: المصفوفات يمكن دمجها عن طريق التكرار أو الضرب.

```
combined_tuple = my_tuple * 2  # دمج المصفوفة
```

القواميس (Dictionaries) في بايثون

---

1. تعريف القواميس (Dictionaries)

القاموس هو نوع بيانات غير متسلسل (غير قابل للترتيب) في بايثون يُستخدم لتخزين البيانات كزوج من المفاتيح والقيم. يمكن استخدام القواميس لتخزين مجموعة غير متكررة من البيانات حيث يتم تعيين قيمة لكل مفتاح.

أمثلة على القواميس

```
# تعريف قاموس
my_dict = {
    "name": "Ali",
    "age": 30,
    "city": "Dubai"
}

# الوصول إلى قيمة باستخدام المفتاح
print(my_dict["name"])  # سيتم طباعة "Ali"

# إضافة عنصر إلى القاموس
my_dict["email"] = "ali@example.com"
print(my_dict)  # سيتم طباعة {'name': 'Ali', 'age': 30, 'city': 'Dubai',
'email': 'ali@example.com'}

# تعديل عنصر في القاموس
my_dict["age"] = 31
print(my_dict)  # سيتم طباعة {'name': 'Ali', 'age': 31, 'city': 'Dubai',
'email': 'ali@example.com'}
```

---

2. العمليات الشائعة على القواميس

إضافة عناصر: باستخدام المفاتيح والقيم.

```
my_dict["phone"] = "123-456-7890"
```

حذف عناصر: باستخدام `del` أو `pop()`.

```
del my_dict["city"]  # حذف عنصر باستخدام المفتاح
my_dict.pop("email")  # حذف عنصر باستخدام `pop()`
```

التعديل: باستخدام المفتاح لتغيير القيمة.

```
my_dict["age"] = 32  # تعديل قيمة المفتاح "age"
```

التحقق من وجود عنصر: باستخدام `in`.

```
if "name" in my_dict:
    print("Name is present!")  # سيتم الطباعة
```

---

3. التكرار خلال القواميس

باستخدام `for` أو `items()` للحصول على المفاتيح والقيم.

```
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

---

4. المزايا والفوائد

- القواميس تقدم مرونة وسهولة في تخزين وتعديل البيانات بشكل غير متسلسل.
- يمكن استخدام القواميس في العديد من الحالات مثل تخزين إعدادات التطبيقات، وإدارة البيانات، وربط النصوص بالمفاتيح.

المجموعات (Sets) في بايثون

---

1. تعريف المجموعات (Sets)

المجموعة هي كائن غير متسلسل (غير قابل للترتيب) في بايثون يُستخدم لتخزين مجموعة فريدة من العناصر. لا تحتوي المجموعات على أي تكرار، ويمكن أن تحتوي على عناصر من نفس النوع أو أنواع مختلفة.

أمثلة على المجموعات

```
# تعريف مجموعة
my_set = {1, 2, 3, "hello", 4.5}
```

```
# عرض العناصر في المجموعة
print(my_set)  # سيتم طباعة مجموعة من العناصر الفريدة

# التحقق من وجود عنصر
print(2 in my_set)  # سيتم طباعة "True"

# إضافة عنصر إلى المجموعة
my_set.add(6)
print(my_set)  # سيتم طباعة مجموعة تحتوي على 6

# إزالة عنصر من المجموعة
my_set.remove(1)
print(my_set)  # سيتم طباعة مجموعة بدون العنصر 1
```

---

2. العمليات الشائعة على المجموعات

إضافة عناصر: باستخدام `add()`.

```
my_set.add(7)
```

إزالة عناصر: باستخدام `remove()` أو `discard()`.

```
my_set.remove(2)  # حذف عنصر
```

تكرار عناصر: يمكن استخدام الضرب (*) لتكرار المجموعة.

```
new_set = my_set * 2  # تكرار المجموعة مرتين
```

العمليات المنطقية: مثل الجمع (|)، التقاطع (&)، الفرق (-)، الفرق المتناظر (^).

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

print(set1 | set2)  # الاتحاد: {1, 2, 3, 4, 5}
```

```
print(set1 & set2)  # 3} :التقاطع}
print(set1 - set2)  # 2 ,1} :الفرق}
print(set1 ^ set2)  # 5 ,4 ,2 ,1} :الفرق المتناظر}
```

---

3. الفوائد والاستخدامات

- **إزالة التكرار**: المجموعات تضمن أن تكون العناصر فريدة.
- **الأداء**: المجموعات توفر أداءً سريعًا للعمليات مثل البحث، والإضافة، والإزالة.
- **عدم الترتيب**: العناصر في المجموعة غير مرتبة، مما يجعلها مثالية لاستخدامها في حالات حيث الأهمية تكون في التحقق من وجود عنصر أو إدارته بشكل سريع.

التلاعب بالنصوص (Strings) في بايثون

---

1. تعريف النصوص (Strings)

النصوص (Strings) في بايثون هي كائنات تستخدم لتمثيل النصوص النصية باستخدام النصوص المحاطة بأقواس مزدوجة أو فردية.

أمثلة على النصوص (Strings)

```
# تعريف نصوص
my_string = "Hello, World!"

# عرض النص
print(my_string)  # سيتم طباعة "Hello, World!"

# طول النص
print(len(my_string))  # 13 سيتم طباعة

# الوصول إلى نص معين باستخدام الفهرسة
print(my_string[0])  # سيتم طباعة "H"

# التقطيع (Slicing) للنصوص
print(my_string[0:5])  # سيتم طباعة "Hello"

# استبدال النصوص
print(my_string.replace("World", "Python"))  # سيتم طباعة "Hello, Python!"
```

```
# تقسيم النصوص إلى أجزاء
print(my_string.split(","))  # سيتم طباعة ['Hello', ' World!']
```

---

2. العمليات الشائعة على النصوص

الإدراج: باستخدام القوسين المزدوجين أو المفردين.

```
new_string = "Python" + " " + "Programming"
```

الإدراج الديناميكي: باستخدام القوسات مع النصوص الديناميكية.

```
name = "John"
greeting = f"Hello, {name}!"
print(greeting)  # سيتم طباعة "Hello, John!"
```

الاستبدال: باستخدام `replace()`.

```
modified_string = my_string.replace("World", "Universe")
```

التقسيم: باستخدام `split()`.

```
parts = my_string.split(" ")
```

التحقق من وجود نص معين: باستخدام `in`.

```
if "World" in my_string:
    print("Found!")  # سيتم الطباعة
```

---

3. العمليات الإضافية

إزالة المسافات البيضاء: باستخدام `strip()`.

```
trimmed_string = my_string.strip()
```

تحويل النص إلى نصوص صغيرة/كبيرة: باستخدام `lower()` أو `upper()`.

```
small_string = my_string.lower()
capitalized_string = my_string.upper()
```

الإدراج في مواقع معينة: باستخدام الفهرسة المعكوسة.

```
indexed_insert = my_string[:5] + " Python" + my_string[5:]
```

---

4. الفوائد والاستخدامات

- **التعامل مع النصوص**: النصوص في بايثون توفر الكثير من الطرق لمعالجة النصوص، مثل البحث، الاستبدال، والدمج.
- **المرونة**: النصوص تسهل إنشاء وتعديل النصوص النصوصية، مما يجعلها أداة قوية لمعالجة البيانات النصية.

المفاهيم الأساسية: الفئات (Class) والكائنات (Object) في بايثون

---

1. ما هي الفئات (Class)؟

الفئة (Class) هي مخطط أو قالب يمكننا من خلاله إنشاء كائنات. تحتوي الفئة على **خصائص (Attributes)** و**وظائف (Methods)** تصف سلوك الكائنات التي سيتم إنشاؤها منها.

---

2. ما هو الكائن (Object)؟

الكائن (Object) هو نسخة (Instance) من الفئة. يمكن أن يحتوي الكائن على بيانات وسلوك بناءً على تعريف الفئة.

---

3. تعريف الفئة وإنشاء كائن

مثال:

```
# تعريف فئة
class Person:
    # تعريف المُهيّئ (Constructor)
    def __init__(self, name, age):
        self.name = name  # خاصية الاسم
        self.age = age    # خاصية العمر

# تعريف دالة داخل الفئة
    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years
old."

# إنشاء كائن من الفئة
person1 = Person("Ali", 25)

# استدعاء خاصية ودالة
print(person1.name)          # طباعة "Ali"
print(person1.greet())       # طباعة "Hello, my name is Ali and I am 25
years old."
```

---

4. المُهيّئ (`__init__` Constructor):

- **المهمة:** يُستخدم لتهيئة خصائص الكائن عند إنشائه.
- يتم استدعاؤه تلقائيًا عند إنشاء كائن باستخدام الفئة.

مثال:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

my_car = Car("Toyota", "Corolla")
print(my_car.brand)  # طباعة "Toyota"
print(my_car.model)  # طباعة "Corolla"
```

---

5. الفرق بين الفئة والكائن

الفئة (Class)	الكائن (Object)
مخطط لإنشاء الكائنات.	نسخة من الفئة.
يحتوي على الخصائص والوظائف العامة.	يحتوي على بيانات مخصصة بناءً على الفئة.

---

6. التفاعل بين الكائنات والفئات

مثال:

```
class Animal:
    def __init__(self, species):
        self.species = species

    def sound(self):
        return "This is a generic sound."

# إنشاء كائنات متعددة
cat = Animal("Cat")
dog = Animal("Dog")

print(cat.species)  # طباعة "Cat"
print(dog.sound())  # طباعة "This is a generic sound."
```

---

7. الفوائد والاستخدامات

- **إعادة الاستخدام:** يمكن استخدام الفئات لإنشاء كائنات متعددة بنفس الخصائص والسلوك.
- **تنظيم الكود:** يساعد في تنظيم البرامج الكبيرة.
- **قابلية التوسع:** يسهل إضافة ميزات جديدة دون تعديل كبير.

إنشاء الأصناف (Classes) والخصائص (Attributes) والدوال (Methods) داخلها في بايثون

---

1. ما هي الأصناف والخصائص والدوال؟

- **الصنف (Class):** هو قالب لإنشاء الكائنات التي تشترك في خصائص وسلوكيات معينة.
- **الخاصية (Attribute):** هي متغير داخل الصنف يمثل حالة أو بيانات الكائن.

- **الدالة (Method):** هي وظيفة داخل الصنف تحدد سلوك الكائن.

---

2. إنشاء صنف مع خصائص ودوال

مثال بسيط:

```
# تعريف الصنف
class Student:
    # المُهيّئ (Constructor) لتعريف الخصائص
    def __init__(self, name, age, grade):
        self.name = name  # خاصية الاسم
        self.age = age    # خاصية العمر
        self.grade = grade  # خاصية الدرجة

# دالة لعرض معلومات الطالب
    def get_info(self):
        return f"Name: {self.name}, Age: {self.age}, Grade:
{self.grade}"

    # دالة لتحديث الدرجة
    def update_grade(self, new_grade):
        self.grade = new_grade
        return f"{self.name}'s grade has been updated to {self.grade}."

# إنشاء كائن من الصنف
student1 = Student("Ali", 20, "A")

# عرض معلومات الطالب
print(student1.get_info())  # طباعة: "Name: Ali, Age: 20, Grade: A"

# تحديث الدرجة
print(student1.update_grade("A+"))  # طباعة: "Ali's grade has been updated
to A+."
```

---

3. أنواع الخصائص

الخصائص العامة (Public Attributes): يمكن الوصول إليها من داخل وخارج الصنف.

```
```

```
student1.name = "Ahmed"  # تعديل خاصية عامة
print(student1.name)     # طباعة: "Ahmed"
```

الخصائص الخاصة (Private Attributes): تبدأ بشرطتين سفليتين (__) ولا يمكن الوصول إليها مباشرة من خارج الصنف.

```
class Example:
    def __init__(self):
        self.__private_attr = "This is private"

    def get_private(self):
        return self.__private_attr

obj = Example()
print(obj.get_private())  # طباعة: "This is private"
```

---

4. استخدام الدوال داخل الصنف

- **دوال الوصول (Getter Methods):** للحصول على قيمة خاصية معينة.
- **دوال التحديث (Setter Methods):** لتحديث قيمة خاصية معينة.

مثال:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.__price = price  # خاصية خاصة

# دالة للحصول على السعر
    def get_price(self):
        return self.__price

    # دالة لتحديث السعر
    def set_price(self, new_price):
        if new_price > 0:
            self.__price = new_price
        else:
            print("Invalid price!")
```

```
product1 = Product("Laptop", 1500)

# عرض السعر
print(product1.get_price())  # 1500 :طباعة

تحديث السعر #
product1.set_price(2000)
print(product1.get_price())  # 2000 :طباعة
```

---

5. الجمع بين الخصائص والدوال

مثال عملي:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"{amount} has been deposited. New balance:
{self.__balance}"
        return "Invalid deposit amount!"

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"{amount} has been withdrawn. Remaining balance:
{self.__balance}"
        return "Invalid withdrawal amount!"

# إنشاء كائن لحساب بنكي
account = BankAccount("Sara", 500)

# الإيداع والسحب
print(account.deposit(200))   # 200" :طباعة has been deposited. New
balance: 700"
print(account.withdraw(300))  # 300" :طباعة has been withdrawn. Remaining
balance: 400"
```

---

6. المزايا

- **تنظيم الكود:** تسهيل قراءة الكود وصيانته.
- **إعادة الاستخدام:** يمكن إنشاء عدة كائنات من نفس الصنف.
- **الكبسلة (Encapsulation):** حماية البيانات باستخدام الخصائص الخاصة والدوال.

الوراثة (Inheritance) والتعددية الشكلية (Polymorphism) في بايثون

---

1. الوراثة (Inheritance)

- **الوراثة** هي واحدة من أهم مفاهيم البرمجة الكائنية (OOP).
تسمح الوراثة بإنشاء صنف جديد (صنف فرعي) يعتمد على صنف موجود مسبقًا (صنف أساسي).
يُمكن للصنف الفرعي أن يرث الخصائص والدوال من الصنف الأساسي ويُضيف ميزات أو يُعدّلها.

مثال على الوراثة:

```
# تعريف الصنف الأساسي
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# تعريف الصنف الفرعي
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# إنشاء كائنات من الأصناف
dog = Dog("Buddy")
cat = Cat("Kitty")

print(dog.speak())  # طباعة: "Buddy says Woof!"
print(cat.speak())  # طباعة: "Kitty says Meow!"
```

ميزات الوراثة:

1. **إعادة الاستخدام:** يمكن استخدام كود الصنف الأساسي في الأصناف الفرعية دون تكرار.
2. **توسيع الوظائف:** يمكن للأصناف الفرعية إضافة وظائف أو تعديلها دون الحاجة إلى تعديل الصنف الأساسي.

---

2. التعددية الشكلية (Polymorphism)

- **التعددية الشكلية** تسمح باستخدام نفس الدالة أو الخاصية بطرق مختلفة بناءً على السياق.
  تُعتبر التعددية الشكلية أساسًا قويًا لتصميم البرامج بحيث يكون الكود أكثر مرونة وقابلية للتوسع.

مثال على التعددية الشكلية:

تطبيق التعددية من خلال الدوال:

```
# تعريف الأصناف
class Bird:
    def speak(self):
        return "Birds chirp!"

class Parrot(Bird):
    def speak(self):
        return "Parrots mimic sounds!"

class Crow(Bird):
    def speak(self):
        return "Crows caw!"

# استخدام التعددية الشكلية
def make_sound(bird):
    print(bird.speak())

# إنشاء كائنات
parrot = Parrot()
crow = Crow()

# استدعاء نفس الدالة ولكن بتصرف مختلف
make_sound(parrot)  # طباعة: "Parrots mimic sounds!"
make_sound(crow)    # طباعة: "Crows caw!"
```

تطبيق التعددية مع القوائم:

```
# قائمة تحتوي على كائنات من أصناف مختلفة
animals = [Dog("Buddy"), Cat("Kitty")]

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

الوراثة مع التعددية الشكلية:

مثال عملي:

```
class Vehicle:
    def move(self):
        return "The vehicle is moving."

class Car(Vehicle):
    def move(self):
        return "The car is driving on the road."

class Boat(Vehicle):
    def move(self):
        return "The boat is sailing on water."

class Plane(Vehicle):
    def move(self):
        return "The plane is flying in the air."

# التعددية الشكلية
vehicles = [Car(), Boat(), Plane()]

for vehicle in vehicles:
    print(vehicle.move())
```

النتيجة:

```
The car is driving on the road.
The boat is sailing on water.
The plane is flying in the air.
```

---

3. الوراثة متعددة المستويات (Multilevel Inheritance):

مثال:

```
class LivingBeing:
    def breathe(self):
        return "Breathing..."

class Mammal(LivingBeing):
    def feed_milk(self):
        return "Feeding milk to babies."

class Human(Mammal):
    def speak(self):
        return "Speaking a language."

human = Human()
print(human.breathe())    # طباعة: "Breathing..."
print(human.feed_milk())  # طباعة: "Feeding milk to babies."
print(human.speak())      # طباعة: "Speaking a language."
```

---

4. الوراثة المتعددة (Multiple Inheritance):

مثال:

```
class Father:
    def skill(self):
        return "Good at math."

class Mother:
    def skill(self):
        return "Good at art."

class Child(Father, Mother):
    pass

child = Child()
print(child.skill())  # طباعة: "Good at math." (لأنها الوراثة الأولى)
```

ملخص:

- **الوراثة**: تسهّل إعادة استخدام الكود وتنظيمه.
- **التعددية الشكلية**: تعطي مرونة في التصميم من خلال استخدام نفس الدوال بطرق مختلفة.

قراءة وكتابة الملفات النصية في بايثون

---

1. فتح الملفات باستخدام الدالة `open()`

للتعامل مع الملفات النصية في بايثون، نستخدم الدالة `open()`، التي تقبل وسيطين رئيسيين:

- **اسم الملف:** اسم الملف النصي المراد فتحه.
- **وضعية الفتح (Mode):** تحدد كيفية التعامل مع الملف (قراءة، كتابة، إضافة، إلخ).

أوضاع الفتح الشائعة:

الوضع	الوصف
`"r"`	قراءة فقط (الوضع الافتراضي).
`"w"`	كتابة، يحذف محتوى الملف إذا كان موجودًا.
`"a"`	إضافة، يكتب إلى نهاية الملف.
`"+r"`	قراءة وكتابة.

---

2. قراءة الملفات النصية

مثال: قراءة محتوى ملف

```
# فتح الملف للقراءة
with open("example.txt", "r") as file:
    content = file.read()  # قراءة كامل محتوى الملف
    print(content)
```

قراءة الملف سطرًا بسطر:

```
```

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # إزالة المسافات البيضاء
```

قراءة الملف كقائمة من الأسطر:

```
with open("example.txt", "r") as file:
    lines = file.readlines()  # إرجاع قائمة تحتوي على كل سطر
    print(lines)
```

---

3. كتابة الملفات النصية

مثال: الكتابة إلى ملف

```
# فتح الملف للكتابة
with open("example.txt", "w") as file:
    file.write("Welcome to Python Programming!\n")
    file.write("This is a new line.\n")
```

ملاحظة: إذا كان الملف موجودًا، سيتم حذف محتواه واستبداله.

إضافة محتوى إلى نهاية الملف:

```
with open("example.txt", "a") as file:
    file.write("This is an additional line.\n")
```

---

4. معالجة الأخطاء أثناء التعامل مع الملفات

التأكد من وجود الملف قبل قراءته:

```
import os

if os.path.exists("example.txt"):
```

```
    with open("example.txt", "r") as file:
        print(file.read())
else:
    print("The file does not exist.")
```

التعامل مع الاستثناءات:

```
try:
    with open("example.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found!")
except IOError:
    print("An error occurred while handling the file.")
```

---

5. مثال عملي: برنامج بسيط لحفظ واسترجاع الملاحظات

```
# حفظ ملاحظة في ملف
def save_note():
    note = input("Enter your note: ")
    with open("notes.txt", "a") as file:
        file.write(note + "\n")
    print("Note saved!")
```

```
 # قراءة كل الملاحظات
def read_notes():
    print("\nYour notes:")
    try:
        with open("notes.txt", "r") as file:
            for line in file:
                print(line.strip())
    except FileNotFoundError:
        print("No notes found.")

# البرنامج الرئيسي
while True:
    print("\n1. Save a note")
```

```
    print("2. Read notes")
    print("3. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        save_note()
    elif choice == "2":
        read_notes()
    elif choice == "3":
        break
    else:
        print("Invalid choice. Try again.")
```

---

6. نصائح مهمة:

- استخدم تعليمة `with open()` لأنها تغلق الملف تلقائيًا بعد الانتهاء من العمل.
- تحقق دائمًا من وجود الملف قبل قراءته.
- تأكد من وجود أذونات الكتابة عند محاولة تعديل الملفات.

التعامل مع الملفات بصيغة CSV في بايثون

ملفات CSV (Comma-Separated Values) تُستخدم لتخزين البيانات على شكل جدول حيث تكون القيم مفصولة بفاصلة. يمكن التعامل مع هذه الملفات بسهولة باستخدام مكتبة `csv` المدمجة في بايثون.

---

1. قراءة ملفات CSV

أ. قراءة الملف كسطور نصية

```
import csv

# قراءة ملف CSV وعرض محتوياته
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

مثال على ملف CSV:

```
Name,Age,City
Ali,25,Dubai
Lina,30,Cairo
Ahmed,28,Riyadh
```

ب. تخزين البيانات كقوائم

```
import csv

data = []

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        data.append(row)

print(data)
```

ج. قراءة الملف باستخدام رؤوس الأعمدة (`DictReader`)

```
import csv

# استخدام DictReader لقراءة البيانات كرؤوس وأعمدة
with open("data.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(f"Name: {row['Name']}, Age: {row['Age']}, City:
{row['City']}")
```

---

2. كتابة ملفات CSV

أ. كتابة البيانات كسطور

```
import csv
```

```
data = [
    ["Name", "Age", "City"],
    ["Ali", 25, "Dubai"],
    ["Lina", 30, "Cairo"],
    ["Ahmed", 28, "Riyadh"],
]

with open("output.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

print("Data written to output.csv")
```

ب. كتابة البيانات مع رؤوس الأعمدة (`DictWriter`)

```
import csv

data = [
    {"Name": "Ali", "Age": 25, "City": "Dubai"},
    {"Name": "Lina", "Age": 30, "City": "Cairo"},
    {"Name": "Ahmed", "Age": 28, "City": "Riyadh"},
]

with open("output.csv", "w", newline="") as file:
    fieldnames = ["Name", "Age", "City"]
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()  # كتابة رؤوس الأعمدة
    writer.writerows(data)

print("Data written to output.csv")
```

---

3. تعديل ملفات CSV

إضافة بيانات جديدة إلى ملف موجود

```
import csv

new_data = [
```

```
    ["Sara", 22, "Beirut"],
    ["Omar", 35, "Amman"],
]

with open("output.csv", "a", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(new_data)

print("New data appended to output.csv")
```

تعديل القيم داخل الملف

```
import csv

updated_data = []

# قراءة البيانات وتعديلها
with open("output.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        if row[0] == "Lina":
            row[1] = 32  # تحديث العمر
        updated_data.append(row)

# إعادة كتابة الملف
with open("output.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(updated_data)

print("Data updated in output.csv")
```

---

4. التعامل مع القيم المخصصة (مثل الفاصلة المنقوطة)

قراءة ملف CSV بفاصلة منقوطة (;)

```
import csv

with open("data_semicolon.csv", "r") as file:
    reader = csv.reader(file, delimiter=";")
    for row in reader:
```

```
        print(row)
```

كتابة ملف CSV بفاصلة منقوطة (;)

```
import csv

data = [
    ["Name", "Age", "City"],
    ["Ali", 25, "Dubai"],
    ["Lina", 30, "Cairo"],
    ["Ahmed", 28, "Riyadh"],
]

with open("output_semicolon.csv", "w", newline="") as file:
    writer = csv.writer(file, delimiter=";")
    writer.writerows(data)

print("Data written to output_semicolon.csv")
```

---

5. معالجة الأخطاء أثناء التعامل مع ملفات CSV

التحقق من وجود الملف

```
import os

filename = "data.csv"

if os.path.exists(filename):
    with open(filename, "r") as file:
        print(file.read())
else:
    print("File not found!")
```

التعامل مع استثناءات CSV

```
import csv
```

```
try:
    with open("data.csv", "r") as file:
        reader = csv.reader(file)
        for row in reader:
            print(row)
except FileNotFoundError:
    print("The file does not exist.")
except csv.Error as e:
    print(f"CSV error: {e}")
```

أفكار تطبيقية باستخدام ملفات CSV

1. **برنامج إدارة الطلاب:**
   - حفظ بيانات الطلاب (الاسم، العمر، الدرجة) في ملف CSV.
   - قراءة وتحديث بيانات الطلاب بسهولة.
2. **تحليل بيانات:**
   - قراءة بيانات مبيعات أو إحصائيات من ملف CSV وتحليلها باستخدام مكتبة مثل `pandas`.
3. **دمج ملفات CSV:**
   - جمع محتوى ملفات متعددة في ملف CSV واحد.

العمل مع ملفات JSON في بايثون

JSON (JavaScript Object Notation) هو تنسيق خفيف لتبادل البيانات يُستخدم على نطاق واسع. تدعم لغة بايثون التعامل مع JSON من خلال مكتبة `json` المدمجة.

---

1. ما هو JSON؟

JSON يعبر عن البيانات كأزواج من المفاتيح والقيم:

```
{
    "name": "Ali",
    "age": 25,
    "city": "Cairo"
}
```

---

2. مكتبة `json` في بايثون

استيراد المكتبة:

```
import json
```

---

3. قراءة ملفات JSON

أ. قراءة ملف JSON وتحويله إلى كائن بايثون

```
import json

# قراءة ملف JSON
with open("data.json", "r") as file:
    data = json.load(file)

print(data)  # طباعة البيانات كقاموس بايثون
```

ملف JSON:

```
{
    "name": "Ali",
    "age": 25,
    "skills": ["Python", "Java", "HTML"]
}
```

ب. الوصول إلى القيم داخل JSON

```
print(data["name"])     # Ali
print(data["skills"])   # ['Python', 'Java', 'HTML']
```

---

4. كتابة بيانات إلى ملف JSON

أ. كتابة قاموس إلى ملف JSON

```
import json

data = {
    "name": "Lina",
    "age": 30,
    "city": "Dubai",
    "skills": ["SQL", "C++", "CSS"]
}

with open("output.json", "w") as file:
    json.dump(data, file, indent=4)

print("Data written to output.json")
```

نتيجة ملف JSON:

```
{
    "name": "Lina",
    "age": 30,
    "city": "Dubai",
    "skills": ["SQL", "C++", "CSS"]
}
```

---

5. تحويل البيانات بين JSON وكائنات بايثون

أ. تحويل JSON إلى كائن بايثون (loads)

```
import json

json_string = '{"name": "Ahmed", "age": 28, "city": "Riyadh"}'
data = json.loads(json_string)

print(data)            # {'name': 'Ahmed', 'age': 28, 'city': 'Riyadh'}
print(data["city"])    # Riyadh
```

ب. تحويل كائن بايثون إلى JSON (dumps)

```python
import json

data = {
    "name": "Sara",
    "age": 22,
    "city": "Beirut"
}

json_string = json.dumps(data, indent=4)
print(json_string)
```

---

6. تعديل محتوى JSON

تحديث القيم وإعادة الكتابة

```python
import json

# قراءة الملف
with open("data.json", "r") as file:
    data = json.load(file)

# تعديل البيانات
data["age"] = 26
data["skills"].append("Django")

# كتابة التعديلات إلى الملف
with open("data.json", "w") as file:
    json.dump(data, file, indent=4)

print("Data updated successfully")
```

---

7. التعامل مع الأخطاء

التحقق من وجود الملف

```python
import os
```

```
filename = "data.json"

if os.path.exists(filename):
    with open(filename, "r") as file:
        data = json.load(file)
        print(data)
else:
    print("File not found!")
```

التعامل مع استثناءات JSON

```
import json

try:
    with open("data.json", "r") as file:
        data = json.load(file)
        print(data)
except json.JSONDecodeError as e:
    print(f"Error decoding JSON: {e}")
except FileNotFoundError:
    print("File not found!")
```

---

8. استخدام JSON مع واجهات برمجة التطبيقات (APIs)

إرسال طلب واستقبال JSON

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
data = response.json()

print(data)
```

نتيجة JSON المستلمة:

```
                    }
  ,userId": 1"
      ,id": 1"
```

```
,"title": "Sample Title"
".body": "This is the content of the post"
{
```

---

9. JSON المتداخل (Nested JSON)

الوصول إلى البيانات داخل JSON متداخل

```python
import json

nested_json = {
    "name": "Omar",
    "details": {
        "age": 35,
        "address": {
            "city": "Amman",
            "country": "Jordan"
        }
    }
}

# الوصول إلى القيم
print(nested_json["details"]["address"]["city"])  # Amman
```

---

10. أفكار تطبيقية

1. **إدارة بيانات الطلاب:**
   - تخزين واسترجاع بيانات الطلاب (الأسماء، الأعمار، الدرجات) باستخدام JSON.
2. **إنشاء إعدادات للتطبيق:**
   - حفظ إعدادات التطبيق في ملف JSON (مثل اللغة المفضلة، السمة).
3. **التكامل مع APIs:**
   - جلب البيانات أو إرسالها باستخدام JSON.

أهمية إدارة الأخطاء في البرمجة

إدارة الأخطاء (Error Handling) تُعد واحدة من العناصر الأساسية في تطوير البرمجيات، حيث تهدف إلى تحسين جودة البرامج وتجربة المستخدم. الأخطاء قد تحدث في أي وقت أثناء تشغيل البرنامج، سواء بسبب إدخال بيانات غير صحيحة من المستخدم، أو بسبب مشاكل تقنية، أو حتى أخطاء غير متوقعة في النظام.

أسباب أهمية إدارة الأخطاء

1. **تحسين استقرار البرنامج:**
   - إدارة الأخطاء بشكل صحيح تساعد في منع انهيار البرنامج عند حدوث خطأ. بدلًا من توقف البرنامج بشكل مفاجئ، يمكن التعامل مع الخطأ وإبقاء النظام قيد التشغيل.
2. **تحسين تجربة المستخدم:**
   - عند إدارة الأخطاء بشكل فعال، يمكن عرض رسائل واضحة وسهلة الفهم للمستخدم بدلاً من رسائل النظام الغامضة. هذا يجعل المستخدم يشعر بالثقة في البرنامج.
3. **سهولة تصحيح الأخطاء (Debugging):**
   - توفر إدارة الأخطاء سجلات (Logs) أو تفاصيل دقيقة حول الأخطاء التي تحدث، مما يساعد المطورين في التعرف على المشكلة وحلها بسرعة.
4. **تجنب المشاكل الأمنية:**
   - الأخطاء التي لا تتم إدارتها قد تُظهر معلومات حساسة للمستخدمين، مثل مسارات الملفات أو أسماء الجداول في قاعدة البيانات. إدارة الأخطاء تقلل من هذا الخطر.
5. **التعامل مع الأخطاء غير المتوقعة:**
   - البرامج لا تعمل دائمًا في بيئات مثالية. إدارة الأخطاء تسمح بالتعامل مع الظروف غير المتوقعة مثل انقطاع الاتصال أو نقص الموارد.

أنواع الأخطاء الشائعة

1. **أخطاء زمن التشغيل (Runtime Errors):**
   - تحدث أثناء تشغيل البرنامج مثل تقسيم رقم على صفر أو الوصول إلى عنصر غير موجود في قائمة.
2. **أخطاء منطقية (Logical Errors):**
   - تحدث عندما تكون النتائج غير صحيحة بسبب خطأ في منطق الكود.
3. **أخطاء التركيب (Syntax Errors):**
   - تحدث عند كتابة كود غير متوافق مع قواعد اللغة البرمجية.
4. **أخطاء موارد النظام:**
   - مثل عدم وجود مساحة كافية في الذاكرة أو فقدان اتصال الإنترنت.

أفضل ممارسات لإدارة الأخطاء

1. **استخدام هيكل Try-Catch:**
   - معظم اللغات البرمجية توفر أدوات مثل `try-catch` للتعامل مع الأخطاء.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("لا يمكن القسمة على صفر!")
```

2. **توفير رسائل خطأ واضحة:**
   - بدلاً من عرض رسالة مثل "Error 404"، يجب أن تكون الرسائل مفهومة مثل "الصفحة المطلوبة غير موجودة".
3. **استخدام سجلات الأخطاء (Logging):**
   - حفظ تفاصيل الأخطاء في سجلات لمساعدة المطورين في تحليل المشكلات.

```
import logging

logging.basicConfig(filename='error.log', level=logging.ERROR)
try:
    result = 10 / 0
except ZeroDivisionError as e:
    logging.error(f"An error occurred: {e}")
```

4. **إعادة المحاولة (Retries):**
   - في بعض الحالات، يمكن إعادة المحاولة عند فشل العملية بدلاً من إنهائها.

```
import time

for attempt in range(3):
    try:
        # محاولة الاتصال بالخادم
        connect_to_server()
        break
    except ConnectionError:
        print("فشل الاتصال. المحاولة مرة أخرى...")
        time.sleep(2)
```

5. **تجنب كشف التفاصيل الحساسة:**
   - لا تعرض معلومات داخلية مثل أسماء الملفات أو كلمات المرور عند حدوث خطأ.
6. **التعامل مع أخطاء الإدخال:**
   - تحقق دائمًا من صحة البيانات المدخلة من المستخدم.

```


age = input
```

```
("أدخل عمرك: ")
```

```
if not age.isdigit():
    print("يرجى إدخال رقم صالح.")
else:
    print(f"عمرك هو: {age}")
```

فوائد طويلة المدى

- **الاستدامة:** البرامج التي تدير الأخطاء بشكل جيد تكون أكثر استدامة وقابلة للصيانة.
- **الأداء الجيد:** يمكن تقليل التأثير السلبي للأخطاء على الأداء العام.
- **تحسين الأمان:** يساعد في حماية النظام من الاستغلال.
- **رضا المستخدم:** يؤدي إلى تجربة مستخدم إيجابية وموثوقة.

استخدام `try-except` لإدارة الاستثناءات في بايثون

إدارة الاستثناءات (Exceptions Handling) هي واحدة من الميزات القوية في لغة بايثون، حيث تتيح للمبرمجين التعامل مع الأخطاء التي قد تحدث أثناء تشغيل البرنامج دون التسبب في انهياره. يتم ذلك باستخدام البنية `try-except`.

مفهوم `try-except`

- **كتلة `try`:** تحتوي على الكود الذي قد يسبب خطأً.
- **كتلة `except`:** تحتوي على الكود الذي يتم تنفيذه إذا حدث استثناء (خطأ).

صيغة `try-except` الأساسية

```
try:
    # الكود الذي قد يتسبب في خطأ
except:
    # الكود الذي يتم تنفيذه إذا حدث خطأ
```

أمثلة على استخدام `try-except`

1. التعامل مع القسمة على صفر

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر!")
```

النتيجة:

```
خطأ: لا يمكن القسمة على صفر!
```

---

2. التعامل مع عدة أنواع من الأخطاء

```
try:
    num = int(input("أدخل رقمًا: "))
    result = 10 / num
except ValueError:
    print("خطأ: يرجى إدخال رقم صحيح.")
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر!")
```

سيناريوهات:

إذا أدخل المستخدم نصًا بدل الرقم:

```
خطأ: يرجى إدخال رقم صحيح.
```

إذا أدخل المستخدم الرقم 0:

```
خطأ: لا يمكن القسمة على صفر!
```

---

3. استخدام الكلمة المفتاحية `else`

يمكن استخدام `else` لتنفيذ كود إضافي إذا لم تحدث استثناءات.

```
try:
    num = int(input("أدخل رقمًا: "))
    result = 10 / num
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر!")
except ValueError:
    print("خطأ: يرجى إدخال رقم صحيح.")
else:
    print(f"النتيجة هي: {result}")
```

---

4. استخدام `finally`

يتم تنفيذ كتلة `finally` سواء حدث استثناء أو لم يحدث.

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("خطأ: الملف غير موجود.")
finally:
    print("تم الانتهاء من محاولة قراءة الملف.")
```

النتيجة:

إذا كان الملف غير موجود:

```
خطأ: الملف غير موجود.
تم الانتهاء من محاولة قراءة الملف.
```

---

5. استخدام المتغير في `except`

يمكنك استخدام متغير لتخزين تفاصيل الخطأ.

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"حدث خطأ: {e}")
```

النتيجة:

```
حدث خطأ: division by zero
```

فوائد استخدام `try-except`

1. **منع انهيار البرنامج:**
   - يحافظ على استمرارية البرنامج حتى عند حدوث أخطاء.
2. **تقديم رسائل خطأ واضحة:**
   - يساعد في تحسين تجربة المستخدم.
3. **سهولة تصحيح الأخطاء:**
   - يمكن تسجيل الأخطاء لمراجعتها لاحقًا.
4. **إدارة الأخطاء المتوقعة:**
   - مثل القسمة على صفر أو فقدان الملفات.

أفضل الممارسات عند استخدام `try-except`

1. **حدد الاستثناءات بدقة:**
   - تجنب استخدام `except` بدون تحديد نوع الاستثناء، لأن ذلك قد يخفي أخطاء غير متوقعة.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("خطأ: لا يمكن القسمة على صفر!")
```

2. **استخدم رسائل خطأ واضحة ومفهومة:**
   - تأكد من أن الرسائل توضح المشكلة للمستخدم.
3. **لا تبالغ في استخدام `try-except`:**
   - استخدمه فقط عند الحاجة، لأن الأخطاء البسيطة يمكن حلها بطرق أخرى.

التعريف بالمكتبات القياسية في بايثون: random ،math، وdatetime

تتضمن لغة بايثون مكتبات قياسية متعددة تقدم وظائف وأدوات جاهزة للاستخدام في مختلف التطبيقات. من بين هذه المكتبات المكتبة **math** للحسابات الرياضية، و**random** لتوليد القيم العشوائية، و**datetime** للتعامل مع الوقت والتاريخ.

---

1. مكتبة `math`

تُستخدم مكتبة **math** لإجراء العمليات والحسابات الرياضية المتقدمة.

أمثلة على الوظائف في مكتبة `math`:

الجذر التربيعي (Square Root):

```
import math

print(math.sqrt(16))  # 4.0 :الناتج
```

حساب القوى (Power):

```
print(math.pow(2, 3))  # 8.0 :الناتج
```

الثوابت الرياضية:

```
print(math.pi)  # 3.141592653589793 :الناتج
print(math.e)   # 2.718281828459045 :الناتج
```

الدوال المثلثية:

```

print(math.sin(math.pi / 2))  # 1.0 :الناتج
print(math.cos(0))           # 1.0 :الناتج
```

---

2. مكتبة `random`

تُستخدم مكتبة **random** لتوليد الأرقام والقيم العشوائية، وهي مفيدة في الألعاب والمحاكاة.

أمثلة على الوظائف في مكتبة `random`:

توليد عدد عشوائي بين 0 و1:

```
import random

print(random.random())  # 0.678936239 :مثال
```

توليد عدد صحيح عشوائي في نطاق معين:

```

print(random.randint(1, 10))  # 7 :مثال
```

اختيار عنصر عشوائي من قائمة:

```
options = ['علي' ,'محمد' ,'أحمد']
print(random.choice(options))  # محمد' :مثال'
```

خلط قائمة عشوائيًا:

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)  # 2 ,4 ,1 ,5 ,3] :مثال]
```

---

3. مكتبة `datetime`

تُستخدم مكتبة **datetime** للتعامل مع التاريخ والوقت، مثل عرض التاريخ الحالي، أو حساب الفروقات الزمنية.

أمثلة على الوظائف في مكتبة `datetime`:

عرض التاريخ والوقت الحالي:

```
from datetime import datetime

now = datetime.now()
print(now)  # 12:45:30.123456 19-01-2025 :مثال
```

تنسيق التاريخ والوقت:

```
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_date)  # 12:45:30 19-01-2025 :مثال
```

حساب الفرق بين تاريخين:

```

from datetime import timedelta

today = datetime.now()
future_date = today + timedelta(days=7)
print(future_date)  # 12:45:30.123456 26-01-2025 :مثال
```

إنشاء تاريخ ووقت محدد:

```
specific_date = datetime(2025, 1, 1, 10, 30)
print(specific_date)  # 10:30:00 01-01-2025 :الناتج
```

فوائد استخدام المكتبات القياسية

1. **توفير الوقت والجهد:**
   - لا تحتاج إلى كتابة دوال معقدة من البداية، فالمكتبات القياسية تقدم لك حلولًا جاهزة.
2. **سهولة الاستخدام:**
   - الدوال والوظائف في هذه المكتبات واضحة ومباشرة.
3. **التكامل مع بايثون:**

- هذه المكتبات مضمنة في بايثون، فلا حاجة لتثبيتها بشكل منفصل.

مقدمة إلى مكتبات شائعة في بايثون: NumPy وPandas

تعتبر مكتبتا **NumPy** و**Pandas** من أكثر المكتبات شيوعًا واستخدامًا في بايثون، خاصة في مجال تحليل البيانات والعلوم الحسابية. تقدم كل من المكتبتين أدوات قوية وفعالة للتعامل مع البيانات وتنظيمها وإجراء العمليات الحسابية عليها.

---

1. مكتبة NumPy

ما هي مكتبة NumPy؟

- **NumPy** هي مكتبة بايثون مخصصة للتعامل مع المصفوفات متعددة الأبعاد (**Arrays**) وتنفيذ العمليات الرياضية عليها بسرعة وكفاءة.
- تعتمد مكتبة NumPy على لغة C تحت السطح، مما يجعل العمليات الرياضية أسرع بكثير مقارنة بالقوائم العادية في بايثون.

أمثلة على استخدام NumPy:

تثبيت المكتبة:

```
pip install numpy
```

إنشاء مصفوفة NumPy:

```
import numpy as np

array = np.array([1, 2, 3, 4, 5])
print(array)  # 5 4 3 2 1] :الناتج]
```

إنشاء مصفوفة ذات أبعاد متعددة:

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix)
# :الناتج
```

```
[3 2 1]] #
6 5 4]  #]]
```

إجراء عمليات رياضية على المصفوفات:

```
array = np.array([1, 2, 3, 4, 5])
print(array + 10)  # 15 14 13 12 11] :الناتج]
print(array * 2)   # 10 8  6  4  2 ] :الناتج]
```

حساب المتوسط والانحراف المعياري:

```
print(np.mean(array))  # 3.0 :الناتج
print(np.std(array))   # 1.4142135623730951 :الناتج
```

---

2. مكتبة Pandas

ما هي مكتبة Pandas؟

- **Pandas** هي مكتبة مخصصة للتعامل مع البيانات المهيكلة مثل الجداول.
- تعتمد Pandas على الكائنين **Series** (عمود واحد) و**DataFrame** (جدول كامل).

أمثلة على استخدام Pandas:
تثبيت المكتبة:

```
pip install pandas
```

إنشاء DataFrame:

```
import pandas as pd

data = {
```

```
    "Name": ["Ali", "Sara", "Omar"],
    "Age": [25, 30, 22],
    "City": ["Riyadh", "Jeddah", "Dammam"]
}
df = pd.DataFrame(data)
print(df)
# الناتج:
#    Name  Age     City
# 0   Ali   25   Riyadh
# 1  Sara   30   Jeddah
# 2  Omar   22   Dammam
```

قراءة البيانات من ملف CSV:

```
df = pd.read_csv("data.csv")
print(df.head())  # عرض أول 5 صفوف من البيانات
```

إجراء عمليات على الأعمدة:

```
print(df["Age"].mean())  # متوسط الأعمار
print(df["City"].value_counts())  # عدد مرات تكرار المدن
```

تصفية البيانات:

```
filtered = df[df["Age"] > 25]
print(filtered)
# الناتج: الصفوف التي أعمارها أكبر من 25 فقط
```

إضافة عمود جديد:

```
df["Salary"] = [5000, 6000, 4500]
print(df)
# الناتج: تمت إضافة عمود "Salary" إلى الجدول
```

الفروقات بين NumPy وPandas

الميزة	NumPy	Pandas
الغرض الأساسي	العمليات الرياضية والمصفوفات	تحليل البيانات المهيكلة
الهيكل الأساسي	Arrays (مصفوفات)	DataFrames و Series
سهولة الاستخدام	مناسب للبيانات العددية فقط	يدعم بيانات عددية ونصية
التعامل مع الملفات	لا يدعم قراءة الملفات	يدعم قراءة CSV و Excel

---

- **NumPy** مثالية للعمل مع البيانات العددية الكبيرة وإجراء العمليات الحسابية المعقدة.
- **Pandas** أداة قوية لتحليل البيانات المهيكلة مثل الجداول وإجراء تصفية ومعالجة البيانات بسهولة.

من خلال استخدام هاتين المكتبتين، يمكن تنفيذ جميع المهام المتعلقة بتحليل البيانات وتنظيمها بكفاءة وفعالية.

برنامج آلة حاسبة باستخدام لغة بايثون

يمكن إنشاء برنامج بسيط لآلة حاسبة باستخدام لغة بايثون لتلبية العمليات الحسابية الأساسية (الجمع، الطرح، الضرب، القسمة). إليك الكود مع شرح لكل جزء:

الكود:

```
# تعريف دالة لكل عملية حسابية
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
```

```
        return "خطأ: لا يمكن القسمة على الصفر!"
    return x / y

# واجهة المستخدم
print("مرحبًا بك في برنامج الآلة الحاسبة!")
print("اختر العملية التي تريد تنفيذها:")
print("1. الجمع")
print("2. الطرح")
print("3. الضرب")
print("4. القسمة")

# أخذ اختيار المستخدم
choice = input("أدخل رقم العملية (1/2/3/4): ")

# التحقق من اختيار المستخدم
if choice in ('1', '2', '3', '4'):
    num1 = float(input("أدخل الرقم الأول: "))
    num2 = float(input("أدخل الرقم الثاني: "))

    if choice == '1':
        print(f"النتيجة: {num1} + {num2} = {add(num1, num2)}")
    elif choice == '2':
        print(f"النتيجة: {num1} - {num2} = {subtract(num1, num2)}")
    elif choice == '3':
        print(f"النتيجة: {num1} * {num2} = {multiply(num1, num2)}")
    elif choice == '4':
        print(f"النتيجة: {num1} / {num2} = {divide(num1, num2)}")
else:
    print("اختيار غير صالح، يرجى المحاولة مرة أخرى.")
```

شرح الكود:

1. **تعريف الدوال:**
   - `add(x, y)`: تجمع رقمين.
   - `subtract(x, y)`: تطرح الرقم الثاني من الأول.
   - `multiply(x, y)`: تضرب الرقمين.
   - `divide(x, y)`: تقسم الرقم الأول على الثاني مع التحقق من عدم القسمة على الصفر.
2. **واجهة المستخدم:**
   - يتم عرض قائمة بالعمليات الحسابية المتاحة.
   - يقوم المستخدم باختيار العملية عبر إدخال رقم العملية.
3. **إدخال القيم:**
   - يطلب البرنامج من المستخدم إدخال الرقمين المطلوب تنفيذ العملية عليهما.
4. **تنفيذ العملية:**
   - بناءً على اختيار المستخدم، يتم استدعاء الدالة المناسبة وعرض النتيجة.

5. **التحقق من المدخلات:**
   - يتحقق البرنامج من أن اختيار المستخدم صالح (بين 1 و4).

كيفية تشغيل البرنامج:

احفظ الكود في ملف Python (مثلاً: `calculator.py`).

شغل البرنامج باستخدام الأمر التالي في سطر الأوامر:

```
python calculator.py
```

لعبة التخمين البسيطة باستخدام بايثون

يمكننا إنشاء لعبة تخمين بسيطة حيث يحاول اللاعب تخمين رقم عشوائي يختاره البرنامج. هذه اللعبة طريقة ممتعة لتعلم كيفية استخدام الحلقات والجمل الشرطية في بايثون.

الكود:

```
import random

def guess_the_number():
    print("مرحبًا بك في لعبة التخمين!")
    print("سيتم اختيار رقم عشوائي بين 1 و 100.")
    print("حاول تخمين الرقم، وسنخبرك إن كنت قريبًا!")

# اختيار رقم عشوائي
    random_number = random.randint(1, 100)
    attempts = 0

    while True:
        try:
            # أخذ تخمين المستخدم
            guess = int(input("أدخل تخمينك: "))
            attempts += 1  # زيادة عدد المحاولات

            if guess < random_number:
                print("الرقم الذي اخترته أقل من الرقم الصحيح. حاول مرة أخرى.")
            elif guess > random_number:
                print("الرقم الذي اخترته أكبر من الرقم الصحيح. حاول مرة أخرى.")
```

```
            else:
                print(f"تهانينا! لقد خمّنت الرقم الصحيح {random_number} في
{attempts} محاولة.")
                break
        except ValueError:
            print("يرجى إدخال رقم صحيح.")

    print("):  شكراً للعبك!")

# تشغيل اللعبة
guess_the_number()
```

شرح الكود:

1. **استيراد مكتبة `random`:**
   - تُستخدم مكتبة `random` لاختيار رقم عشوائي.
2. **دالة `guess_the_number`:**
   - تحتوي على منطق اللعبة بالكامل.
3. **اختيار الرقم العشوائي:**
   - يتم اختيار رقم عشوائي باستخدام `random.randint(1, 100)`.
4. **الحلقة اللانهائية `while True`:**
   - تسمح للمستخدم بمحاولة التخمين حتى يحصل على الإجابة الصحيحة.
5. **التخمين والتحقق:**
   - يتحقق البرنامج من أن المدخل صحيح (رقم).
   - يتم مقارنة التخمين بالرقم العشوائي:
     - إذا كان التخمين أقل: يظهر رسالة "الرقم أقل".
     - إذا كان أكبر: يظهر رسالة "الرقم أكبر".
     - إذا كان التخمين صحيحًا: تنتهي اللعبة مع رسالة تهنئة.
6. **عدد المحاولات:**
   - يتم احتساب عدد المحاولات التي احتاجها اللاعب لتخمين الرقم الصحيح.
7. **التعامل مع الأخطاء:**
   - إذا أدخل المستخدم قيمة غير رقمية، يظهر رسالة خطأ دون إيقاف البرنامج.

كيفية تشغيل اللعبة:

- احفظ الكود في ملف Python (مثلاً: `guess_game.py`).

شغل البرنامج باستخدام الأمر:

```
python guess_game.py
```

تطوير اللعبة:

يمكنك إضافة تحسينات مثل:

- تحديد مستوى الصعوبة (تقليل أو زيادة نطاق الأرقام).
- إضافة خاصية تحديد عدد المحاولات المتاحة.
- حفظ أعلى درجات اللاعبين.

أداة لتحليل النصوص باستخدام بايثون (مثل حساب الكلمات)

يمكننا بناء أداة بسيطة لتحليل النصوص. هذه الأداة ستقوم بحساب عدد الكلمات، الجمل، الأحرف، وأحيانًا إظهار أكثر الكلمات استخدامًا في النص.

الكود:

```
def text_analyzer(text):
    # حساب عدد الأحرف
    total_characters = len(text)

    # حساب عدد الكلمات
    words = text.split()
    total_words = len(words)

    # حساب عدد الجمل
    sentences = text.split('.')
    total_sentences = len([s for s in sentences if s.strip() != ''])

    # إحصاء الكلمات الأكثر استخدامًا
    word_frequency = {}
    for word in words:
        word = word.lower().strip(",.?!")  # إزالة العلامات والتأكد من الحروف الصغيرة
        word_frequency[word] = word_frequency.get(word, 0) + 1

    # ترتيب الكلمات حسب التكرار
    sorted_words = sorted(word_frequency.items(), key=lambda x: x[1],
reverse=True)

    # عرض النتائج
    print(f"تحليل النص:")
```

```
    print(f"- عدد الأحرف: {total_characters}")
    print(f"- عدد الكلمات: {total_words}")
    print(f"- عدد الجمل: {total_sentences}")
    print("\nالكلمات الأكثر تكرارًا:")
    for word, freq in sorted_words[:5]:
        print(f"  {word}: {freq} ("مرة

# مثال على الاستخدام
text = """
Python is a popular programming language. Python is used in data
science, web development, and more.
It is easy to learn and powerful.
"""

text_analyzer(text)
```

شرح الكود:

1. **حساب عدد الأحرف:**
   - يتم استخدام دالة `len()` لحساب عدد الأحرف في النص.
2. **حساب عدد الكلمات:**
   - يتم تقسيم النص إلى كلمات باستخدام `split()`.
3. **حساب عدد الجمل:**
   - يتم تقسيم النص إلى جمل باستخدام `split('.')`، ثم يتم تجاهل الجمل الفارغة.
4. **إحصاء الكلمات الأكثر استخدامًا:**
   - يتم تنظيف كل كلمة من العلامات (مثل الفواصل والنقاط) وتحويلها إلى حروف صغيرة.
   - يتم إنشاء قاموس `word_frequency` لتخزين عدد مرات ظهور كل كلمة.
5. **ترتيب الكلمات حسب التكرار:**
   - يتم ترتيب القاموس بناءً على القيم (عدد التكرارات) باستخدام `sorted()`.
6. **عرض النتائج:**
   - يتم طباعة عدد الأحرف، الكلمات، الجمل، والكلمات الأكثر تكرارًا.

كيفية تشغيل البرنامج:

- انسخ الكود إلى ملف Python (مثلًا: `text_analyzer.py`).

شغل البرنامج باستخدام الأمر:

```
python text_analyzer.py
```

تطوير الأداة:

1. **إضافة خصائص جديدة:**
   - حساب عدد الأحرف بدون مسافات.
   - الكشف عن الكلمات الفريدة.
2. **توسيع التحليل:**
   - تحليل الجمل الطويلة أو الكلمات الأكثر طولًا.
   - دعم لغات أخرى غير الإنجليزية.
3. **واجهة مستخدم:**
   - إنشاء واجهة رسومية أو تطبيق ويب لهذه الأداة.

مثال على الإخراج:

```
تحليل النص:
- عدد الأحرف: 130
- عدد الكلمات: 18
- عدد الجمل: 3

الكلمات الأكثر تكرارًا:
python: 2 مرة
is: 2 مرة
and: 2 مرة
a: 1 مرة
popular: 1 مرة
```

استخدامات الأداة:

- تحليل المقالات أو النصوص الطويلة.
- فهم الأنماط اللغوية في النصوص.
- تحسين النصوص المكتوبة بناءً على التكرار والتحليل.

العمل مع ملفات CSV وتحليل بياناتها باستخدام بايثون

ملفات CSV (Comma-Separated Values) تُستخدم لتخزين البيانات على شكل جدول. بايثون توفر مكتبة مدمجة تُسمى `csv` للتعامل مع هذه الملفات، بالإضافة إلى مكتبات مثل **Pandas** لتحليل البيانات بشكل متقدم.

مثال: قراءة ملف CSV وعرض بياناته باستخدام مكتبة csv

كود بسيط لقراءة ملف CSV:

python
CopyEdit

```
import csv

# قراءة الملف وعرض البيانات
with open('data.csv', mode='r', encoding='utf-8') as file:
    reader = csv.reader(file)

    # قراءة العناوين
    headers = next(reader)
    print(f"العناوين: {headers}")

    # قراءة الصفوف
    for row in reader:
        print(f"صف البيانات: {row}")
```

شرح الكود:

1. يتم فتح الملف باستخدام `open()`.
2. يتم إنشاء كائن `reader` لقراءة محتويات الملف.
3. تُستخدم دالة `next()` لتخطي العناوين.
4. يتم عرض البيانات من كل صف في الملف.

تحليل ملف CSV باستخدام مكتبة Pandas

كود لتحليل البيانات:

```
import pandas as pd

# قراءة الملف باستخدام Pandas
df = pd.read_csv('data.csv')

# عرض أول 5 صفوف
print("أول 5 صفوف من البيانات:")
print(df.head())
```

```
# عرض معلومات عن البيانات
print("\nمعلومات عن البيانات:")
print(df.info())

# حساب متوسط قيمة عمود معين
if 'price' in df.columns:
    avg_price = df['price'].mean()
    print(f"\nمتوسط السعر: {avg_price}")
```

شرح الكود:

1. **قراءة الملف:** يتم استخدام `pd.read_csv()` لتحميل البيانات في DataFrame.
2. **عرض البيانات:** تعرض الدالة `head()` أول 5 صفوف.
3. **معلومات عن البيانات:** تعرض `info()` وصفًا عامًا عن الأعمدة والبيانات.
4. **تحليل البيانات:** مثال على حساب متوسط القيم في عمود **price**.

إنشاء ملف CSV جديد

كود لإنشاء وكتابة بيانات في ملف CSV:

```
import csv

# البيانات المراد حفظها
data = [
    ['Name', 'Age', 'Country'],
    ['Ali', 25, 'Egypt'],
    ['Sara', 30, 'Jordan'],
    ['John', 22, 'USA']
]

# إنشاء الملف
with open('output.csv', mode='w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)

    # كتابة البيانات
    writer.writerows(data)

print("تم إنشاء الملف بنجاح!")
```

مثال عملي: تحليل بيانات المبيعات

ملف CSV يحتوي على بيانات المبيعات:

```
Product,Category,Price,Quantity
Laptop,Electronics,1000,5
Smartphone,Electronics,700,8
Shoes,Fashion,50,20
T-shirt,Fashion,20,15
```

كود التحليل باستخدام Pandas:

```
import pandas as pd

# قراءة ملف CSV
df = pd.read_csv('sales.csv')

# حساب إجمالي الإيرادات لكل منتج
df['Revenue'] = df['Price'] * df['Quantity']

# حساب إجمالي الإيرادات لكل فئة
category_revenue = df.groupby('Category')['Revenue'].sum()

print("إجمالي الإيرادات لكل فئة:")
print(category_revenue)

# المنتج الأكثر مبيعًا
top_product = df.loc[df['Quantity'].idxmax()]
print(f"\nالمنتج الأكثر مبيعًا: {top_product['Product']} بكمية
{top_product['Quantity']}")
```

خلاصة:

1. مكتبة `csv` مناسبة للمشاريع البسيطة.
2. مكتبة **Pandas** قوية لتحليل ومعالجة البيانات.
3. يمكن استخدام هذه الأدوات لتحليل المبيعات، أداء الموظفين، أو أي بيانات مشابهة.

استخدامات ملفات CSV:

- إدارة بيانات الطلاب.
- تحليل تقارير المبيعات.
- تصدير أو استيراد بيانات بين الأنظمة.

استخدام MySQL في بايثون

MySQL هو نظام إدارة قاعدة بيانات شائع يُستخدم للتعامل مع البيانات وتخزينها واسترجاعها. يمكن استخدام MySQL مع بايثون عن طريق مكتبات مثل **MySQL-Connector** أو **MySQL-Python**.

إعداد MySQL مع بايثون

تثبيت مكتبة MySQL-Connector:

```
pip install mysql-connector-python
```

ارتباط MySQL ببايثون:

```
import mysql.connector

# الاتصال بقاعدة البيانات
db_connection = mysql.connector.connect(
    host="localhost",  # عنوان قاعدة البيانات
    user="username",   # اسم المستخدم
    password="password",  # كلمة المرور
    database="database_name"  # اسم قاعدة البيانات
)

cursor = db_connection.cursor()

# تنفيذ استعلام بسيط
cursor.execute("SELECT * FROM table_name")

# عرض النتائج
results = cursor.fetchall()
for row in results:
    print(row)

# إغلاق الاتصال
cursor.close()
db_connection.close()
```

إدراج بيانات في قاعدة البيانات

```
import mysql.connector

# الاتصال بقاعدة البيانات
db_connection = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="database_name"
)

cursor = db_connection.cursor()

# إدراج بيانات جديدة
sql = "INSERT INTO table_name (column1, column2) VALUES (%s, %s)"
values = ('value1', 'value2')

cursor.execute(sql, values)
db_connection.commit()  # حفظ التغييرات

# عرض عدد الأسطر المتأثرة
print(f"تم إدراج {cursor.rowcount} سطر")

cursor.close()
db_connection.close()
```

تحديث بيانات في قاعدة البيانات

```
import mysql.connector

# الاتصال بقاعدة البيانات
db_connection = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="database_name"
)
```

```
cursor = db_connection.cursor()

# تحديث بيانات
sql = "UPDATE table_name SET column1 = %s WHERE column2 = %s"
values = ('new_value', 'condition_value')

cursor.execute(sql, values)
db_connection.commit()  # حفظ التغييرات

print(f"تم تحديث {cursor.rowcount} سطر")

cursor.close()
db_connection.close()
```

حذف بيانات من قاعدة البيانات

```
import mysql.connector

# الاتصال بقاعدة البيانات
db_connection = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="database_name"
)

cursor = db_connection.cursor()

# حذف بيانات
sql = "DELETE FROM table_name WHERE column = %s"
value = ('value_to_delete',)

cursor.execute(sql, value)
db_connection.commit()  # حفظ التغييرات

print(f"تم حذف {cursor.rowcount} سطر")

cursor.close()
db_connection.close()
```

عرض البيانات من قاعدة البيانات

```
import mysql.connector

# الاتصال بقاعدة البيانات
db_connection = mysql.connector.connect(
    host="localhost",
    user="username",
    password="password",
    database="database_name"
)

cursor = db_connection.cursor()

# تنفيذ استعلام SELECT
cursor.execute("SELECT * FROM table_name")

# عرض النتائج
results = cursor.fetchall()
for row in results:
    print(row)

cursor.close()
db_connection.close()
```

أهم العمليات مع MySQL وPython:

1. القراءة (SELECT).
2. الإدراج (INSERT).
3. التحديث (UPDATE).
4. الحذف (DELETE).

bit.ly/3VujLCh

إعداد : علاء عرابي

GAUAB.com