

## الوحدة 3: من الخوارزمية إلى البرنامج البسيط

مقدمة: ( تعريف البرنامج وعلاقته بالخوارزمية).

يُعدّ التمييز بين الخوارزمية والبرنامج من الأساسيات المهمة في مجال الإعلام الآلي؛ إذ تمثل الخوارزمية الفكرة المجردة أو الخطوات المنطقية لحل مشكلة معينة، بينما يُعدّ البرنامج ترجمةً لهذه الخوارزمية إلى إحدى لغات البرمجة مثل Python أو Delphi أو Java أو Lisp أو Pascal أو C ، حتى يتمكن الحاسوب من فهمها وتنفيذها. ومن هذا المنطلق، تبقى الخوارزمية مستقلة عن لغة البرمجة، ويمكن تجسيدها في عدة برامج باختلاف اللغات المستعملة. كما ينبغي أن تُصاغ الخوارزمية بطريقة واضحة وسهلة القراءة، مع الحرص على عدم تعقيدها أو إطالتها بشكل مفرط، وفي حال كبر حجمها يُستحسن تقسيمها إلى خوارزميات فرعية، مع إضافة تعليقات توضيحية تسهّل فهمها.

### 1. مراحل تحويل خوارزمية إلى برنامج قابل للتنفيذ:

تمر عملية تحويل الخوارزمية إلى برنامج قابل للتنفيذ عبر مراحل أساسية ومتراصة، تبدأ **بفهم المشكلة** فهماً دقيقاً من خلال تحديد المدخلات والمخرجات والقيود المرتبطة بها. بعد ذلك، يتم **تصميم خوارزمية** واضحة ومنظمة تمثل خطوات الحل، ويمكن التعبير عنها باللغة الطبيعية أو شبه الكود أو المخططات الانسيابية.

في المرحلة الموالية، **تُترجم هذه الخوارزمية إلى برنامج** باستخدام لغة برمجة مناسبة، مع الالتزام بقواعدها من حيث البنية والصياغة. ثم تأتي **مرحلة اختبار البرنامج وتصحيح الأخطاء**، حيث يتم التحقق من سلامة عمله من خلال اكتشاف الأخطاء النحوية والمنطقية ومعالجتها، إضافة إلى اختبار الحالات الخاصة لضمان دقة النتائج.

بعد التأكد من صحة البرنامج، يتم **تشغيله** فعلياً بإدخال المعطيات وملاحظة المخرجات. وتجدر الإشارة إلى أن هذه المراحل ليست منفصلة، بل يمكن الرجوع إلى أي مرحلة سابقة عند الحاجة إلى التصحيح أو التحسين. وتُعد مرحلة الاختبار من أهم مراحل تطوير البرمجيات، لما قد تسببه الأخطاء من نتائج خطيرة، خاصة في الأنظمة الحساسة، مما يجعل التأكد من صحة البرامج أمراً ضرورياً لضمان موثوقيتها.

## II. بيئات البرمجة

لا تقتصر عملية كتابة البرامج على إتقان لغة البرمجة فقط، بل تتطلب أيضًا استخدام ما يُعرف ببيئة التطوير المتكاملة (Integrated Development Environment – IDE)، وهي نظام برمجي يوفر مجموعة من الأدوات الأساسية لبناء البرامج القابلة للتنفيذ. وتساعد هذه البيئة المبرمج على التركيز على الجوانب المنطقية وتصميم الخوارزميات، مع تقليل الانشغال بالتفاصيل التقنية، مما يساهم في تسريع عملية التطوير ورفع دقتها وزيادة موثوقيتها.

### 1. مكونات بيئة البرمجة (IDE) :

تتكوّن بيئة البرمجة من مجموعة من الأدوات التي تعمل بشكل متكامل لمساعدة المبرمج على كتابة البرنامج وتشغيله والتأكد من صحته، حيث تسهّل هذه الأدوات عملية التطوير وتقلّل من الأخطاء وتسريع الإنجاز.

ومن أهم مكوناتها:

1. **محرر الشيفرة (Code Editor)** يُستخدم لكتابة الكود البرمجي وتعديله بسهولة .
2. **المترجم/المفسر (Compiler / Interpreter)** يقوم بتحويل الشيفرة البرمجية إلى لغة يفهمها الحاسوب ثم تنفيذها .
3. **أدوات التصحيح (Debugger)** تساعد على اكتشاف الأخطاء البرمجية وإصلاحها .
4. **أداة التشغيل (Run/Execution)** تمكّن من تشغيل البرنامج ومشاهدة نتائجه .
5. **إدارة المشروع (Project Manager)** تساعد على تنظيم ملفات المشروع ومكوناته .
6. **الإكمال التلقائي والمساعدة (Auto-completion & Help)** تسهّل كتابة الكود بسرعة ودقة وتقتراح الأوامر المناسبة .

وبذلك توفر بيئة البرمجة إطارًا متكاملًا يجمع بين أدوات الكتابة، الترجمة، التصحيح، والتنفيذ في مكان واحد، مما يساعد المبرمج على التركيز على الجانب المنطقي لتصميم الحلول البرمجية بدل الانشغال بالتفاصيل التقنية.

## 2. أنواع بيئات البرمجة:

### أ. بيئات البرمجة المرئية ( Visual Programming )

هي نوع من بيئات البرمجة تعتمد على أسلوب بصري بدل الكتابة النصية، حيث يتم إنشاء البرامج باستعمال كتل (Blocks) تمثل أوامر برمجية، ويقوم المستخدم بسحبها وإفلاتها وتركيبها معًا لبناء تسلسل من التعليمات. وتُعد هذه البيئات مناسبة جدًا للمبتدئين والأطفال، لأنها تساعد على فهم مبادئ البرمجة بطريقة سهلة وتفاعلية. ومن أشهر هذه البيئات **Scratch** و **Blockly**.

يمكن استخدامها مباشرة عبر الإنترنت دون الحاجة إلى تثبيت أي برامج، عبر المواقع التالية:

Scratch: <https://scratch.mit.edu/> .

Blockly: <https://blockly.games> .

#### مميزاتها:

- بسيطة وسهلة التعلم، مما يجعلها مثالية لتعليم أساسيات البرمجة للمبتدئين .
- لكنها تبقى محدودة الإمكانيات ولا تسمح بتطوير مشاريع برمجية معقدة.

### ب. بيئة البرمجة النصية (Text-based Programming Environment)

هي نظام برمجي متكامل يتيح للمبرمج كتابة البرامج باستخدام لغات برمجة نصية مثل: **Python**، **Java**، **C++**، **JavaScript** بدلاً من استخدام عناصر رسومية مثل السحب والإفلات كما في (Scratch)

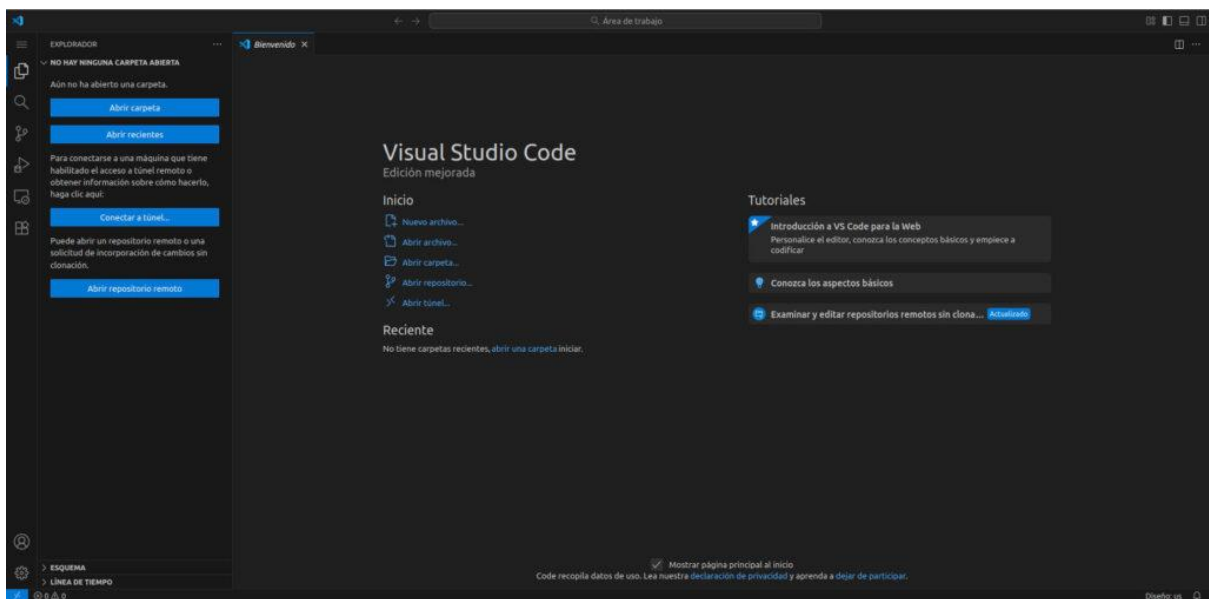
#### خصائصها

- تعتمد على كتابة الكود النصي بدقة .
- توفر مرونة عالية لتطوير البرامج الكبيرة والمعقدة .
- تحتوي على أدوات لتحليل الأداء ومراقبة الأخطاء .
- تسمح بإضافة مكتبات خارجية لتوسيع قدرات البرنامج .
- تمنح المبرمج تحكمًا كاملاً مقارنة بالبيئات المرئية.

## أمثلة مشهورة:

- PyCharm: بيئة مخصصة لتطوير برامج Python
- Visual Studio Code: يدعم عدة لغات مثل C++ و JavaScript و Python.
- Eclipse: يُستخدم بشكل رئيسي مع Java و C++.
- NetBeans: مخصص لتطوير تطبيقات Java.
- Embarcadero RAD Studio: يدعم لغتي C++ و Delphi.

## 3. بيئة البرمجة Visual studio code



هو محرر أكواد مجاني وقوي من تطوير شركة Microsoft، يُستخدم لكتابة وتحرير الشيفرة البرمجية ويدعم عددًا كبيرًا من لغات البرمجة. يتميز بخفته وسرعته، إضافةً إلى مرونته العالية وإمكانية تخصيصه حسب احتياجات المطور.

## الميزات:

- مجاني وخفيف: يعمل بكفاءة على معظم الأجهزة .
- يدعم لغات متعددة: مثل JavaScript و Python و Java و C++ وغيرها .
- إضافات قوية: (Extensions) لتوسيع الوظائف بسهولة .
- إكمال تلقائي ذكي: (IntelliSense) يساعد على كتابة الكود بسرعة ودقة .
- تكامل مع Git: لإدارة المشاريع وتتبع التعديلات .

- واجهة بسيطة وقابلة للتخصيص :مناسبة للمبتدئين والمحترفين .
- أدوات تصحيح (Debugging) لتجربة واكتشاف الأخطاء داخل الكود .
- تكامل مع أدوات حديثة :مثل Docker و Azure.

باختصار VS Code : هو خيار ممتاز لأي مبرمج يبحث عن محرر قوي، سريع، وقابل للتخصيص.

### III. لغة بايثون Python :

هي لغة برمجة عالية المستوى ومفتوحة المصدر تم تطويرها بواسطة Guido van rossum أوائل التسعينات من مميزات أنها سهلة القراءة والكتابة لسهولة قواعدها وبساطتها، مما يجعلها مثالية للمبتدئين والمطورين.

#### 1. أساسيات لغة بايثون:

##### أ. عموميات:

• **المتغيرات:** هي عبارة عن مساحات يتم حجزها في ذاكرة الحاسوب بهدف تخزين البيانات أثناء تشغيل البرنامج. عند تعريف المتغير و إسناد قيمة له، يقوم مفسر لغة بايثون بتحديد نوع هذا المتغير بناءً على هذه القيمة تلقائياً وقت التشغيل. في لغة البايثون يجب إسناد قيمة للمتغير أثناء تعريفه.

##### تنقسم المتغيرات في بايثون إلى:

- **Integer (int):** الأعداد الصحيحة
- **Float:** الأعداد العشرية
- **String (str):** النصوص وتوضع بين علامتي تنصيص
- **Boolean (bool):** القيم المنطقية True أو False

```
x = 10      # عدد صحيح
y = 3.14    # عدد عشري
name = "Ali" # نص
valid = True # منطقي
```

• **التعليقات:** التعليقات هي عبارات تُكتب داخل الشيفرة البرمجية بلغة طبيعية بهدف شرح وظيفة تعليمة أو مجموعة من التعليمات. لا يتم تنفيذ هذه التعليقات من طرف الحاسوب، ولا تظهر للمستخدم عند تشغيل البرنامج، لكنها تساعد المبرمج على فهم الكود وتنظيمه، سواء لنفسه أو لغيره من المبرمجين.

**مثال:**

```
# هذا التعليق يشرح ان البرنامج يطبع رسالة
print(" Hello")
```

## • المسافة البادئة (التسنين / Indentation) :

المسافة البادئة هي أسلوب يُستخدم لتنظيم الشيفرة البرمجية عن طريق الفراغات (Spaces) أو علامات الجدولة (Tabs). يتم من خلالها كتابة كل تعليمة في سطر مستقل، مع إزاحة التعليمات التابعة لنفس البنية إلى الداخل بحيث تكون مصطفة في نفس العمود. تساعد المسافة البادئة على توضيح بنية البرنامج وجعله أكثر قابلية للقراءة والفهم، خاصة في تحديد الكتل البرمجية مثل الشروط والحلقات والدوال. وتنتهي هذه البنية عند الرجوع إلى نفس مستوى بداية الكتلة.

### مثال:

```
If 5>2:
```

```
    print("5 أكبر من 2")
```

• السطر الذي يحتوي على print يوجد فيه مسافة بادئة

هذا يعني أنه تابع لجملته الشرط if.

• إذا لم نضع المسافة البادئة سيعطي البرنامج خطأ .

## ب. العمليات و التعليمات:

تتشابه العمليات والتعليمات في بايثون مع لغتي C و Java، وتشمل:

### • العمليات الحسابية:

- الجمع " + "
- الطرح " - "
- الضرب " \* "
- القسمة الحقيقية " / "
- القسمة الصحيحة " // "
- باقي القسمة " % "
- الأس " \* \* "

- **تعليمية الإسناد (Assignment) :**

تُستخدم " = " لإسناد القيم للمتغيرات: مثال:  $m=m+12$

- **المقارنات:**

- المساواة نستعمل ==

- عدم المساواة نستعمل !=

- **الاختصارات في العمليات:**

يمكن اختصار بعض العمليات على النحو التالي:

|        |                 |
|--------|-----------------|
| $x+=2$ | # بدل $x=x+2$   |
| $p*=i$ | # بدل $p=p*i$   |
| $i-=1$ | # بدل $i=i-1$   |
| $q/=2$ | # بدل $q=q / 2$ |

- **إسناد عدة قيم:**

يمكن إسناد عدة قيم لعدة متغيرات في نفس الوقت ، مثل:  $A, B = 10, 20$  والتي تعني وضع القيمة 10 في المتغيرة A والقيمة 20 في المتغيرة B.

- **تعليمية الإدخال (input):**

تعليمية الإدخال هي أمر يُستخدم في البرمجة لكي نسمح للمستخدم بإدخال بيانات أثناء تشغيل البرنامج.  
**ملاحظة:** كل ما يتم إدخاله بواسطة `input()` يُعتبر نصًا (**string**) حتى لو كان رقمًا.

النوع يكون دائمًا سلسلة نصية (**string**) حتى لو كتبت رقما # `A=input("Enter the value of A: ")`

يأخذ إدخال المستخدم ثم يحوله إلى عدد صحيح # `B=int(input("Enter the value of B: "))`

يحول الإدخال إلى عدد عشري # `C=float(input("Enter the value of C: "))`

تحول كل ما يدخله المستخدم الى قيمة # `TrueV=bool(input("Enter a Boolean value: "))`

ماعدًا اذا لم يخل أي قيمة فستكون False

- **تعليلة الإخراج print():** تعليلة الإخراج print() تُستخدم لعرض المعلومات على الشاشة.

**مثال:**

```
print("\n\tThe solution is : ",X)
a=12
b=20
print("sum of a & b is", a+b)
print("f"sum of {a} and {b} = {a+b} !")
```

**لاحظ**

- وضع الرمز \n داخل السلسلة النصية يؤدي إلى القفز إلى السطر الموالي.
- وضع الرمز \t داخل السلسلة النصية يؤدي إلى ترك مسافة فارغة قبل كتابة السلسلة.
- يمكن كتابة نتيجة عبارة رياضية.
- الحرف "f" في السطر الأخير يدل على أن النص منسق، وبالتالي يمكن إدخال المتغيرات فيه لإظهار قيمها.

**ت. التركيبات:**

**أ. الشرط:**

- **التركبة "if" (إذا):**

```
if m<=2 :
    m = m + 12
    y -= 1
    k = y % 100
```

```
if m<=2:
    m = m + 12
    y -= 1
else:
    m = m + 1
    k = y % 100
```

```
if h == 0:
    print("\tSaturday")
elif h == 1:
    print("\tSunday")
elif h == 2:
    print("\tMonday")
else:
    print("\t\tFalse value")
```

### لاحظ:

- تم استعمال بنية جديدة "if-elif-else"، وهي تمثل دمج عدة شروط "if" بسيطة في تركيب واحد أكثر تنظيماً.
- يجب وضع النقطتين (:) في نهاية كل سطر من: "if"، "elif"، "else" لأنها ضرورية في الصياغة.

### ب. الحلقات:

#### • الحلقة "for" (من أجل):

```
for i in range(0,n):  
    print("\t",Courses[i],":",Marks[i])
```

### لاحظ:

- تُستعمل الدالة (range) لتحديد مجال القيم؛ فعند الكتابة من 2 إلى n فإن المتغير i يأخذ القيم من 2 إلى n-1، ويمكن كتابتها على الشكل: for i in range(n).
- الكتابة for i in range(1, 10, 3) تعبر عن القيم من 1 إلى 9 باستعمال الخطوة (step) 3.
- المجال (range) يمكن أن يكون سلسلة حروف أو قائمة كما في المثال:

```
str1="ALGERIA"  
for c in str1:  
    print(c)  
Week=["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday"]  
for Day in Week:  
    print(Day)
```

### مثال:

```
N=int(input("N ="))  
F=1  
while N>1:  
    F*=N  
    N-=1  
print(F)
```

### ملاحظة:

- الحلقة "repeat" (كرر) غير موجودة في بايثون لكن يمكن تحقيق مبدئها عن طريق الحلقة while كما يلي:

```
while True:
    print("\t",i)
    i=i+1
    if i==10:
        break
```

### لاحظ:

- الصيغة "while True" شرطها محقق دائماً (True) وبالتالي فالدخول إلى داخل الحلقة مضمون من أول مرة.
- عندما يتحقق الشرط i تساوي 10 تنفذ التعليمة break والتي تسمح بالخروج من الحلقة.
- التعليمة break ليست خاصة بالحلقة while بل يمكن استعمالها مع الحلقة for كذلك.
- إذا وجدت التعليمة break داخل حلقة متداخلة ضمن حلقات أخرى، فإنها تؤدي إلى إنهاء تنفيذ الحلقة التي تحتويها فقط، بينما يستمر تنفيذ باقي الحلقات الخارجية بشكل عادي.

### ت. تركيبة معالجة الأخطاء "try" (حاول):

في بايثون، تُستعمل البنية **try** لمعالجة الأخطاء (Exception Handling) بطريقة تمنع توقف البرنامج بشكل مفاجئ.

فعند إدخال المستخدم لقيمة غير صحيحة (مثل حرف بدل عدد)، قد يحدث خطأ يؤدي إلى توقف البرنامج، وربما فقدان بعض البيانات.

لذلك ينصح باستخدام هذه البنية (**try**) في الحالات التالية:

- عند إدخال البيانات من طرف المستخدم
- عند التعامل مع الملفات المخزنة في الجهاز
- عند استقبال بيانات من برامج أخرى أو عبر الشبكات

وذلك بهدف تجنب الأخطاء وضمان استمرار البرنامج بشكل سليم.

### مثال: قراءة عدد صحيح:

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except:
    print("try again please")
```

– إذا أدخل المستخدم قيمة صحيحة فإن التعليمات الموجودة في الجزء "try" هي التي

تتفد. أما إذا أخطأ فستنفذ التعليمة الموجودة  
في الجزء "except".

- قد يقوم المستخدم بإدخال القيمة 0، وهي عدد صحيح ومقبول، لكنها ستنتج خطأ عندما يحاول البرنامج طباعة نتيجة القسمة 10/x، عند حدوث هذا الخطأ، لا يتوقف البرنامج نهائياً، بل ينتقل مباشرة إلى جزء **except** لمعالجة المشكلة، وبهذا يستمر تنفيذ البرنامج بشكل عادي دون انقطاع.

- يمكن أن تتضمن بنية "try" عدة أجزاء "except" لمعالجة أنواع مختلفة من الأخطاء كل على حده. كما يمكن أن تحتوي على جزء "else" ينفذ في حالة عدم وقوع أي خطأ، وجزء "finally" ينفذ دائماً سواء حدث خطأ أم لا.

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("enter an integer value")
except ZeroDivisionError:
    print("division by zero")
else:
    print("success")
finally:
    print("always printed")
```

مثال:

هياكل البيانات في البايثون:

أ. الأنماط (الهياكل) البسيطة (Primitive):

• الأعداد الصحيحة (int) :

مثال:

```
A=10

B=int(input(" "))
```

• الأعداد الحقيقية (العشرية) (float) :

مثال:

```
A=3.14
B=float(input(" "))
```

• الأعداد المنطقية (البوليانية) (bool) :

مثال:

```
A= True
B=False
```

وهي تقبل العمليات المنطقية: not, and, or.

يمكن تحويل أي محتوى إلى قيمة منطقية باستعمال التعليمة bool، مثل: print(bool("Algiers")) حيث ترجع هذه التعليمة القيمة True إذا كان المحتوى غير صفري وغير فارغ.

• النصوص (سلاسل الحروف) (str):

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  |
|-------|----|----|----|----|----|----|----|
| Text= | A  | l  | g  | e  | r  | i  | A  |
| index | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

- النص (String) هو سلسلة من الحروف أو الرموز تكتب بين " " أو ' ' مثل: "Hello" أو 'Hello'.

- لا يمكن تغيير حرف داخل النص مباشرة، لكن يمكن إنشاء نص جديد:

```
text = "Algeria"
text[0] = "a" # خطأ
text = "a" + text[1:] # صحيح
```

- يمكن الوصول إلى الحروف بمؤشر موجب أو سالب:

```
print(text[0]) # A
print(text[-1]) # a
```

–يمكن تقطيعها وتكرارها ودمجها:

```
print(text[0:3]) # Alg
```

```
print(text[2:]) # geria
```

```
text=text+" " +text*3 #Algeria Algeria Algeria Algeria
```

### أهم الدوال (الوظائف) المستعملة مع النصوص:

إليك جدول بأهم الدوال المستعملة مع النصوص. نفرض: txt = " python language "

| النتيجة            | دورها                                                 | اسم الدالة                   |
|--------------------|-------------------------------------------------------|------------------------------|
| "LANGUAGE PYTHON"  | تحويل الحروف إلى حروف كبيرة                           | txt.upper()                  |
| "python language"  | تحويل الحروف إلى حروف صغيرة                           | txt.lower()                  |
| "Python language " | تحويل الحرف الأول إلى حرف كبير والباقي إلى حروف صغيرة | txt.capitalize()             |
| "Python language"  | حذف المسافة في بداية نص أو في نهايته                  | txt.strip()                  |
| "Java language"    | لتعويض محتوى بمحتوى آخر                               | txt.replace("Python","Java") |
| True               | للتأكد إن كانت كل الحروف أبجدية                       | txt.isalpha                  |
| False              | للتأكد إن كانت كل الحروف أرقام                        | txt.isdigit                  |
| True               | للتأكد إن كانت كل الحروف هي أبجدية وأرقام             | txt.isalnum                  |

### ب. الهياكل المركبة (Complex):

#### • القوائم (List) :

```
numbers = [1, 2, 3, 4]
print(numbers[0]) # الوصول لعنصر
numbers.append(5) # اضافة عنصر
```

وهي مجموعة مرتبة من العناصر، قابلة للتغيير.

–إضافة عنصر في الأخير: append

–لحذف عنصر: remove

–لإدخال عنصر: insert

وهناك العديد من الوظائف الأخرى يمكن معرفتها مباشرة من خلال واجهة Visual Studio code.

#### • الصفوف (Tuple) :

وهي قائمة غير قابلة للتغيير، مثل: t = (1, 2, 3)

### • المجموعات (Set) :

هي مجموعة غير مرتبة من عناصر غير مكررة مثل:  $S = \{1,2,3\}$ .

### • المصفوفات والجداول (Matrices):

#### الجداول البسيطة:

إذا كان الجدول بسيطاً (ذو بعد واحد) نستعمل القائمة:

**مثال:**

$T = [10,20,30,40]$

– مؤشر العناصر يبدأ من الصفر. مثلاً  $T[2]$  يدل على القيمة 30.

– على عكس معظم اللغات، الجداول في بايثون ليست محدودة مسبقاً من ناحية الطول.

**المصفوفات :** المصفوفات هي عادة جداول ذات بعدين (أسطر وأعمدة):

– المؤشرات تبدأ من 0.

– المؤشر الأول يدل على رقم السطر والثاني على العمود:

$mx[2,1]$  يدل على القيمة 8.

**مثال:** قراءة عناصر مصفوفة مكونة من عدد من الأسطر وعدد من الأعمدة:

```
Row=int(input("Number of rows:"))
Col=int(input("Number of columns:"))
M=[]
M=[[0 for j in range(Col)] for i in range(Row)]
for i in range(Row):
    for j in range(Col):
        M[i][j]=int(input("M["+str(i)+","+str(j)+"]="))
print(M)
```

– السطر الأول والثاني لقراءة عدد الأسطر

وعدد الأعمدة.

– السطر الثالث: إنشاء مصفوفة فارغة.

– السطر الرابع: تحويل المصفوفة إلى

مصفوفة ذات Row سطر و Col عمود.

– الحلقات للسماح للمستخدم بإدخال القيم

التي يريدها.

– الدالة str لتحويل عدد إلى نص.

## • القاموس (السجل) (Dictionary):

القاموس (Dictionary) هو هيكل بيانات يسمح بتخزين مجموعة من المعلومات المختلفة في نفس البنية. يتم تنظيم البيانات على شكل أزواج (مفتاح - قيمة) (key-value) ، مما يتيح الوصول السريع إلى القيم باستخدام المفاتيح. ويُعتبر بديلاً لمفهوم "struct" في لغة C و "record" في باسكال.

### مثال:

```
Person = {"name": "Ali", "age": 20}
print(Person ["name"])    # الوصول للقيمة باستخدام المفتاح
Person ["age"] = 21        # تعديل قيمة
Person ["city"] = "Algiers" # إضافة مفتاح جديد
del Person ["name"]        # حذف مفتاح
```

### مميزاته:

- الوصول إلى القيم عن طريق المفتاح : السطر 2.
- يمكن تعديل القيم: السطر 3.
- يمكن إضافة أو حذف مفتاح (ميزة غير موجودة في معظم اللغات الأخرى): السطرين 4 و 5.

## الإجراءات والدوال (الوظائف) (Procedures and Functions)

في البرمجة والخوارزميات، تُعدّ الإجراءات والدوال من الأدوات الأساسية التي تساعد على تقسيم البرنامج إلى أجزاء صغيرة قابلة لإعادة الاستخدام، مما يسهّل كتابة البرامج الكبيرة وتنظيمها وتصحيح الأخطاء.

### أولاً: التعريف

#### 1. الإجراء (Procedure):

الإجراء هو وحدة برمجية تُستخدم لتنفيذ مجموعة من التعليمات، ويمكن استدعاؤه عدة مرات داخل البرنامج دون أن يُرجع قيمة ناتجة بشكل مباشر.

- الهدف منه هو تنفيذ مهمة محددة داخل البرنامج .
- يمكن أن يستقبل معطيات (Parameters) كمدخلات .

- لا يعيد قيمة مباشرة (في بعض لغات البرمجة).

## 2. الدالة (الوظيفة) (Function)

الدالة هي وحدة برمجية تُستخدم لتنفيذ مهمة محددة، لكنها تختلف عن الإجراء في أنها تُرجع قيمة يمكن الاستفادة منها لاحقًا داخل البرنامج.

الهدف منها هو معالجة معطيات معينة وإرجاع نتيجة.

تستقبل معطيات (Parameters) كمدخلات.

تُرجع قيمة (Return Value) بعد تنفيذ العملية.

- مزايا استعمال الإجراءات والدوال:

- النمطية (Modularity): تساعد على تقسيم المشكلة الكبيرة والمعقدة إلى أجزاء أصغر وبسيطة، مما يجعل فهم البرنامج أسهل.

- التجريد (Abstraction): يتم وضع جزء من البرنامج في وحدة مستقلة تحمل اسمًا يعبر عن وظيفتها، بحيث يمكن استخدامها دون الحاجة لمعرفة تفاصيل تنفيذها.

- إعادة الاستخدام: يمكن كتابة الكود مرة واحدة ثم استخدامه مرات عديدة في أماكن مختلفة داخل البرنامج.

- سهولة الصيانة: في حال وجود خطأ، يمكن تصحيحه في مكان واحد فقط (داخل الدالة أو الإجراء) دون الحاجة لتعديل البرنامج كاملاً.

ب. الإجراءات و الدوال في الخوارزميات:

### 1. الاجراء:

```

Procedure Permutation(var X, Y:
integer)
Var      Z: integer
Begin
  Z←X
  X←Y
  Y←Z
End
  
```

**مثال 1:** تبديل قيمتي متغيرين (Permutation)

التصريح بالإجراء: (Procedure Declaration)

- المتغيران X و Y يُسمَّيان معاملات (Parameters) أو

متغيرات شكلية، وهما يمثلان مدخلات الإجراء .

- المتغير Z هو متغير محلي يُستعمل داخل الإجراء فقط لتنفيذ عملية التبديل .

## الاستدعاء (Procedure Call) :

بعد تعريف الإجراء ، يمكن استدعاؤه في أي وقت نحتاج فيه إلى تبديل قيم متغيرين من نوع عدد صحيح.

### ملاحظات مهمة:

- يتم التصريح بالإجراء بعد تعريف المتغيرات،

أي ضمن جزء التصريحات في الخوارزمية .

Algorithm ago;

T: array[1:100]of integer

A,B,C: integer

Procedure Permutation(..)

...

end

Begin

...

Permutation(A,B)

Permutation(C,A)

Permutation(T[4],T[6])

...

End

- التعليمة مثل Permutation(A, B): تمثل استدعاءً

للإجراء لتبديل القيمتين الموجودتين في المتغيرين A و B

- يمكن استخدام هذا الإجراء لتبديل أي متغيرين

من نفس النوع (مثل الأعداد الصحيحة) .

- يمكن أيضًا تطبيقه على عناصر داخل مصفوفة (جدول)،

مثل تبديل عنصرين داخل جدول T.

- استعمال الكلمة var قبل المتغيرين X و Y يدل على التمرير

بالمرجع (By Reference) ، أي أن التغيير الذي يحدث

داخل الإجراء يؤثر مباشرة على القيم الأصلية للمتغيرات خارج الإجراء.

## تمرير المعاملات (Arguments) في الإجراءات:

توجد طريقتان أساسيتان لتمرير المعاملات عند استدعاء الإجراءات:

### 1. التمرير بالمرجع: (Pass by Reference / Pass by Address)

هو أسلوب يتم فيه تمرير عنوان (مرجع) المتغير الأصلي إلى الإجراء، بحيث يصبح المعامل

الشكلي داخل الإجراء مرتبطاً مباشرة بنفس موقع الذاكرة.

• أي تغيير يحدث على المعامل داخل الإجراء يؤثر مباشرة على المتغير الأصلي خارج الإجراء .

• يُستخدم للدلالة عليه بإضافة الكلمة var قبل المعاملات .

• يُستعمل عندما نريد تعديل القيم الأصلية للمتغيرات

### 2. التمرير بالقيمة: (Pass by Value)

هو أسلوب يتم فيه تمرير نسخة من قيمة المتغير إلى الإجراء، وليس المتغير نفسه.

- التعديلات التي تتم داخل الإجراء لا تؤثر على المتغير الأصلي خارج الإجراء .
- يُستخدم عندما لا نحتاج إلى تغيير القيم الأصلية .

### ملاحظة:

في مثال تبديل القيم (Permutation) ، لو لم يتم استخدام التمرير بالمرجع (أي بدون كلمة **var**)، فلن يتم تبديل القيم الحقيقية للمتغيرات خارج الإجراء، لأن الإجراء سيعمل فقط على نسخ من القيم وليس على القيم الأصلية.

### مثال 2:

```
Algorithm algo2;
  A : integer
Begin
  A ← 20
  Change(A)
  Write(A)
End
```

```
Procedure Change(X: integer)
Begin
  X ← X + 10
  Write(X)
End
```

### الشرح:

- عند استدعاء **Change(A)** يتم تمرير نسخة من قيمة **A** (أي 20) إلى المتغير **X**.
- داخل الإجراء: تصبح قيمة **X = 30** ويتم عرضها .
- بعد انتهاء الإجراء: تبقى قيمة **A** الأصلية كما هي 20 .
- عند طباعة **A** في النهاية، ستظهر 20 لأنها لم تتغير .

### 2. الدالة:

#### مثال : حساب مساحة دائرة:

لحساب مساحة دائرة باستخدام دالة، يتم تمرير قيمة نصف القطر كمدخل، وتقوم الدالة بإرجاع المساحة المحسوبة كقيمة ناتجة.

#### التصريح بالدالة:

```
Function CircleAire(R: real): real
Begin
  CircleAire ← 3.14 * R * R
End
```

### الاستدعاء :

استدعاء الدالة لا يكون بطريقة مباشرة كما في الإجراءات بل يتم بإسنادها إلى متغيرة من نفس نوع القيمة التي ترجعها الدالة.

```
Algorithm Algo ;  
  R1, R2, R3, S1, S2, S3 : Real  
  Function CircleAire(...)  
    ...  
  End  
Begin  
  ...  
  S1 ← CircleAire(R1)  
  S2 ← CircleAire(R2)  
  S3 ← CircleAire(R3)  
  ...  
End
```

### ت. الإجراءات والدوال في لغة بايثون:

على غرار معظم لغات البرمجة الحديثة، لا تميّز لغة بايثون بشكل صريح بين مفهومي الدالة (Function) والإجراء (Procedure) كما هو الحال في بعض اللغات التقليدية. ففي المفهوم الكلاسيكي، تُعرّف الدالة على أنها وحدة برمجية تُرجع قيمة بعد تنفيذ معالجة معينة، بينما يُعرّف الإجراء على أنه مجموعة من التعليمات تُنفّذ دون إرجاع قيمة.

لكن في بايثون يختفي هذا الفرق، حيث تعتمد اللغة مفهومًا موحدًا هو الدالة، والتي يمكن استخدامها إما لإرجاع قيمة أو لتنفيذ تعليمات فقط. فعند عدم استعمال التعليمة return، تُرجع الدالة تلقائيًا القيمة None، وبذلك تؤدي نفس وظيفة الإجراءات دون الحاجة لتعريف مستقل.

بناءً على ذلك، يمكن اعتبار الإجراءات في بايثون حالة خاصة من الدالة، ويُعد التفريق بينهما مجرد تمييز تعليمي (بيداغوجي) أكثر منه تمييزًا تقنيًا داخل اللغة.

## ملاحظات:

- يتم تعريف الدالة في بايثون باستخدام الكلمة def.
  - تستعمل الكلمة return لتحديد القيمة التي تُرجعها الدالة.
  - يمكن تعريف الدوال في البايثون في أي مكان داخل البرنامج.
- لاحظ الاختلاف بين طريقة استدعاء الدالة Permut واستدعاء الإجراء (دالة بدون إرجاع) Welcome.

```
def Permut(X, Y) :  
    Z=X  
    X=Y  
    Y=Z  
    return X,Y  
...  
A,B = Permut(A,B)
```

```
def Welcome(name) :  
    print(f"Say Welcome to {name} !")  
...  
welcome("Nabil") # this is the call
```

## ملاحظات مهمة:

- في بايثون طريقة تمرير المعاملات يتعلق بنمط المتغيرات والعمليات المطبقة عليها. فبالنسبة للأنماط: int, float, bool, tuple, str التمرير يكون بالقيمة، أما بالنسبة للأنماط: list, dict, set ) أنماط قابلة للتغيير ( فالتمرير يكون بالعنوان.
- إذا أردنا الاحتفاظ بقيمة المتغيرات ذات الأنماط: int, float, bool, tuple, str (استعمال التمرير بالمرجع على هذا الأنواع) يجب استعمال return بقيمة المتغيرات المراد الاحتفاظ بقيمتها مع استعمال الطريقة الصحيحة لاستدعاء الدوال.

## ث. مختلف الأشكال الممكنة لاستدعاء الإجراءات والدوال:

يمكن لأي برنامج رئيسي (Main Program) أو خوارزمية أن يحتوي على عدة دوال أو إجراءات، وتُعتبر هذه الأخيرة أبناء مباشرين (Direct Descendants) للبرنامج الرئيسي. كما يمكن لكل دالة أو إجراء أن يحتوي بدوره على دوال أخرى، مما يُنشئ بنية هرمية منظمة.

انطلاقاً من هذه البنية، يمكن التمييز بين عدة أنواع من استدعاء الدوال والإجراءات، نذكر منها:

## 1) الاستدعاء المباشر من البرنامج الرئيسي (Direct Call from Main Program):

يقوم البرنامج الرئيسي باستدعاء دالة أو إجراء مباشرة لتنفيذ مهمة معينة، وهو أبسط أنواع الاستدعاء وأكثرها استعمالاً.

**مثال:** استدعاء الدالة `AverageCalculating()` من البرنامج الرئيسي.

```
def AverageCalculating(a, b):  
    return (a + b) / 2  
  
# البرنامج الرئيسي  
result = AverageCalculating(10, 20)  
print(result)
```

## 2) الاستدعاء بين الدوال والإجراءات (Inter-Procedure Calls):

يمكن لدالة أو إجراء استدعاء دالة أخرى، مما يساعد على تقسيم المشكلة إلى أجزاء أصغر وأكثر تنظيماً. يساعد هذا النوع على تحقيق مبدأ التجزئة (مبدأ Modularity) وإعادة استعمال الشيفرة.

**مثال:** الدالة `DataProcess()` تستدعي الدالة `ReadData()` ثم تستدعي `DisplayResult()`.

```
def ReadData():  
    return [10, 20, 30]  
  
def DisplayResult(data):  
    print("النتيجة:", data)  
  
def DataProcess():  
    data = ReadData()  
    DisplayResult(data)  
  
# البرنامج الرئيسي  
DataProcess()
```

لدينا دالة اسمها `DataProcess()`  
لكنها لا تقوم بكل العمل وحدها، بل:

- تستدعي `ReadData()` لقراءة البيانات
- ثم تستدعي `DisplayResult()` لعرض النتيجة

### 3) الاستدعاء المتسلسل (Sequential Calls):

يتم استدعاء عدة دوال بشكل متتابع لتنفيذ سلسلة من العمليات. هذا النمط شائع في البرامج التي تعتمد على مراحل معالجة واضحة.

مثال:  $\text{main} \rightarrow \text{read}() \rightarrow \text{process}() \rightarrow \text{display}()$

### 4) الاستدعاء الشرطي (Conditional Calls):

يتم استدعاء دالة أو إجراء بناءً على تحقق شرط معين. وهذا النمط يربط بين مفهومي التحكم في التدفق (Control Flow) واستدعاء الدوال.

مثال:

```
if m >= 0:
    PositiveProcess()
else:
    NegativeProcess()
```

### 5) الاستدعاء التكراري داخل حلقة (Iterative Calls via Loops):

يتم استدعاء نفس الدالة عدة مرات داخل حلقة. لا يجب الخلط بينه وبين الاستدعاء التكراري (Recursion).

### 6) الاستدعاء الذاتي (التكراري) (Recursive Calls):

في هذا الشكل، تقوم الدالة أو الإجراء باستدعاء نفسها لحل مشكلة يمكن تقسيمها إلى مسائل أصغر مشابهة. ومن شروط نجاحها وجود حالة أساسية لإيقاف التكرار، (Base case) ووجود حالات عامة تحسب بنفي الطريقة.

مثال:

- حساب العامل (factoria):  $n! = n * (n-1)!$  مع العلم أن  $0! = 1$ .
- حساب متتالية فيبوناتشي:  $F_n = F_{n-1} + F_{n-2}$  مع العلم أن  $F_0 = 1$  و  $F_1 = 1$ .

## 7) الاستدعاء المتبادل (Mutual Recursion):

تقوم دالتان أو أكثر باستدعاء بعضهما البعض، وهو أقل الأنواع استعمالاً.

**مثال:** الدالة A تستدعي B، و الدالة B تستدعي A.

### الخلاصة:

يهدف هذا الدرس إلى توضيح المسار الذي يمر به حل أي مشكلة، بدايةً من الفكرة وصولاً إلى برنامج قابل للتنفيذ. وفي هذا الإطار تم التطرق إلى عدة نقاط أساسية، من بينها:

- التعرف على مفهوم الخوارزمية والبرنامج، وفهم العلاقة بينهما، مع شرح كيفية تحويل الخوارزمية إلى برنامج عملي .
- استكشاف بيئات البرمجة بنوعيتها (المرئية والنصية)، مع استخدام بيئة **Visual studi code** كأداة لتطوير تطبيقات بايثون .
- دراسة الأساسيات في لغة **Python** مثل المتغيرات، أنواع البيانات، والتعليمات الأساسية .
- توضيح مفهوم الدوال والإجراءات في كل من الخوارزميات وبايثون، مع إبراز الفرق بينهما .
- التعرف على مختلف طرق استدعاء الدوال، مثل الاستدعاء البسيط، المتداخل، والتكراري .

في النهاية، يهدف هذا الدرس إلى تمكين المتعلم من الانتقال من خوارزمية بسيطة إلى برنامج منظم وفعال باستخدام لغة بايثون.