الوحـدة 3: من الخوارزمية إلى البرنامج البسيط

Français : ***De l'algorithme au programme simple***

English : ***From Algorithms to simple program***

الهدف:

الجانب النظري	الجانب التطبيقي
الربط بين المفاهيم الخوارزمية وتطبيقها العملي في لغة برمجة بسيطة مع تقديم أساسيات البرمجة.	تمكين الطلاب من كتابة برامجهم الأولى وتجربة هياكل التحكم، المتغيرات والدوال (الوظائف)، وتطوير مهاراتهم في التصحيح (Debugging).

المحتوى:

1. **مقدمة**
   - تعريف البرنامج وعلاقته بالخوارزمية
   - مراحل تحويل خوارزمية إلى برنامج
2. **بيئات البرمجة:**
   - البيئات المرئية
   - البيئات النصية
3. **بيئة البرمجة Pycharm.**
4. **لغة بايثون Python:**
   - أساسيات لغة بايثون.
5. **الإجراءات والدوال (Procedure & Functions).**
   - الإجراءات والدوال في الخوارزميات
   - الإجراءات والدوال في لغة بايثون
   - مختلف أشكال الاستدعاء

1. مقدمة:

قبل أن نكتب أي برنامج، يجب أن نخطط له. الخوارزمية هي بمثابة المخطط أو الوصفة التي تحل المشكلة خطوة بخطوة. بعد ذلك، نقوم بتحويل هذه الخوارزمية إلى برنامج بلغة برمجة مثلPython ، C++، أو جافا. في هذا الدرس، سنتعلم كيفية الانتقال من التفكير المنطقي (الخوارزمية) إلى كتابة كود حقيقي.

بعد أن تطرقنا في الدرس السابق إلى الخوارزميات من حيث المفهوم والبنية، سنعالج في هذا الدرس مفهوم البرمجة، مع إبراز آليات تحويل الخوارزميات إلى برامج عملية قابلة للتنفيذ.

أ. تعريف البرنامج وعلاقته بالخوارزمية:

البرنامج هو مجموعة من التعليمات المكتوبة بلغة برمجة، تُنفَّذ من طرف الحاسوب بهدف حل مشكلة أو إنجاز مهمة محددة. وترتبط الخوارزمية بالبرنامج ارتباطًا وثيقًا، إذ تمثل الخوارزمية الوصف المنطقي المجرد لخطوات الحل، بينما يُعدّ البرنامج ترجمة عملية لهذه الخطوات إلى صيغة قابلة للتنفيذ على الحاسوب.

ب. مراحل تحويل خوارزمية إلى برنامج قابل للتنفيذ:

تمر عملية تحويل الخوارزمية إلى برنامج عبر مجموعة من المراحل الأساسية، يمكن تلخيصها فيما يلي:

1. **فهم المشكلة:** تتمثل هذه المرحلة في تحليل المشكلة المطروحة بدقة، من خلال تحديد:
   - المعطيات (المدخلات)
   - النتائج المنتظرة (المخرجات)
   - الشروط والقيود المرتبطة بها
2. **تصميم الخوارزمية:** يتم في هذه المرحلة وضع خطوات الحل بطريقة منطقية ومنظمة، مع إمكانية التعبير عنها باستخدام:
   - اللغة الطبيعية
   - شبه الكود(Pseudo-code)
   - المخططات الانسيابية
3. **كتابة البرنامج (الترميز) (Edition):** تُترجم الخوارزمية إلى لغة برمجة ما مثل (Python)، مع احترام قواعد اللغة المختارة (الصياغة، البنية، الكلمات المحجوزة).
4. **اختبار البرنامج وتصحيح الأخطاء (Debugging):** تهدف هذه المرحلة إلى التحقق من صحة عمل البرنامج، من خلال:
   - اكتشاف الأخطاء (النحوية والمنطقية، أثناء التنفيذ) وتصحيحها.
   - اختبار البرنامج على الحالات الخاصة (القيم الحدية) (***Edge cases***) ومعالجتها معالجات خاصة إذا اقتضى الأمر.
5. **تنفيذ البرنامج (Execution):** يتم تشغيل البرنامج على الحاسوب، مع إدخال المعطيات وملاحظة النتائج الناتجة.

تمثل هذه المراحل سلسلة مترابطة، حيث يُعدّ الانتقال المنهجي بينها شرطًا أساسيًا للحصول على برنامج صحيح وفعّال، مع إمكانية الرجوع إلى المراحل السابقة عند الحاجة إلى التصحيح أو التحسين.

وتُعدّ عملية اختبار البرامج وتصحيح أخطائها مرحلة أساسية جدا في تطوير البرمجيات، نظرًا لما قد يترتب عن الأخطاء البرمجية من عواقب خطيرة، خاصة في الأنظمة الحساسة. فقد أدّت بعض الأخطاء البرمجية في الواقع إلى كوارث حقيقية، مثل حادثة صاروخ ***Ariane 5 Flight 501*** الذي انفجر بعد ثوانٍ من إطلاقه بسبب خطأ في تحويل نوع بيانات، وكذلك حوادث جهاز العلاج الإشعاعي ***Therac-25*** التي تسببت في تعريض مرضى لجرعات إشعاعية قاتلة نتيجة أخطاء برمجية.

كما يمكن أن تؤدي الأخطاء في البرامج البنكية أو أنظمة الطيران إلى خسائر مالية جسيمة أو إلى تهديد حياة الإنسان. ومن ثمّ، فإن التأكد من صحة البرامج واختبارها بدقة يُعدّ ضرورة حتمية لضمان سلامة وموثوقية الأنظمة المعلوماتية.

ت. كتابة البرنامج:

لا تقتصر عملية كتابة البرامج على إتقان لغة البرمجة فحسب، بل تستلزم كذلك استخدام ما يُعرف بـ **بيئة التطوير المتكاملة** (***Integrated Development Environment – IDE***)، وهي منظومة برمجية توفر مجموعة من الأدوات الضرورية لبناء برنامج قابل للتنفيذ. إذ تُمكّن هذه البيئة المبرمج من التركيز على الجوانب المنطقية وتصميم الخوارزميات، مع تقليل الانشغال بالتفاصيل التقنية المرافقة، مما يسهم في تسريع عملية التطوير وتحسين دقتها ورفع مستوى موثوقيتها.

2. بيئات البرمجة (Programming Environments):

أ. مفهوم بيئة البرمجة:

بيئة البرمجة هي مجموعة من الأدوات البرمجية التي تمكّن المبرمج من كتابة البرنامج (الشيفرة) (*Code*) (*script*) واختباره (*testing*) وتصحيحه (*debugging*) وتنفيذه (*execution*).

ب. مكوّنات بيئة البرمجة:

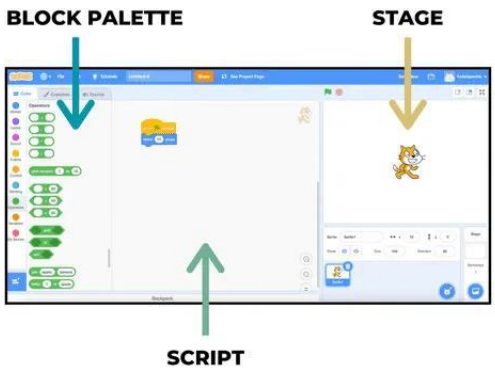
تتكوّن بيئة البرمجة من مجموعة من الأدوات التي تعمل معًا لمساعدة المبرمج على كتابة البرنامج وتشغيله والتأكد من صحته:

1. **محرر الشيفرة(Code Editor) :** أداة تُستخدم لكتابة الكود وتعديله.
2. **المترجم/المفسّر (Compiler / Interpreter) :** يقوم بتحويل الشيفرة إلى لغة يفهمها الحاسوب وتنفيذها.
3. **أدوات التصحيح(Debugger) :** تساعد على اكتشاف الأخطاء وإصلاحها.
4. **أداة التنفيذ (Run/Execution) :** تمكّن من تشغيل البرنامج ورؤية نتائجه.

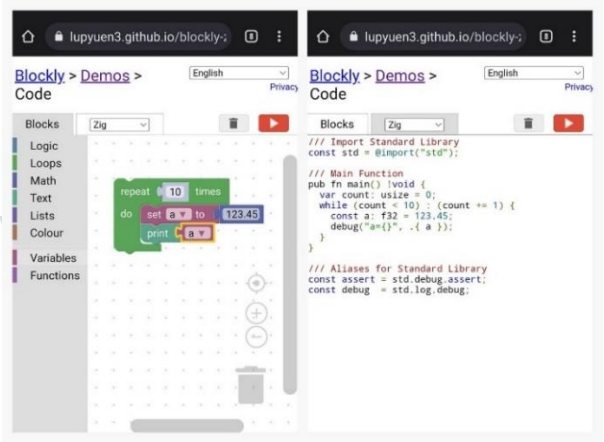
ت. أنواع بيئات البرمجة:

1. بيئات البرمجة المرئية (Visual Programming):

هي بيئات تعتمد على التمثيل الرسومي للأوامر باستخدام كتل (*Blocks*) بدل كتابة نصوص. حيث تعتمد على سحب كتل (باعتبارها أوامر) وإفلاتها وتركيبها فيما بينها من أجل تكوين البرنامج. وهي موجهة عادة للصغار وللمبتدئين من أجل تعلم مبادئ البرمجة. من بين أشهر هذه البيئات نذكر *Scratch* و *Blockly*.



واجهة Scratch



واجهة Blockly

يمكن تجربة هذه البيئات مباشرة عن طريق الانترنت (عن بعد) دون الحاجة إلى تنزيل البرامج:

- موقع *Scratch* : https://scratch.mit.edu/:
- موقع *Blockly*: https://blockly.games :

مميزاتها:

- سهلة التعلم مما يجعلها بيئة مفضلة لصغار السن للتعود على مفاهيم البرمجة.
- محدودة في إمكانياتها وغير مناسبة للتطبيقات المعقدة.

2. بيئات البرمجة النصية (Text-based Programming):

بيئة البرمجة النصية هي نظام برمجي متكامل يُتيح للمبرمج كتابة البرامج باستخدام لغة برمجة نصية (مثل *Python*، *Java*، *C++*، *JavaScript*) بدلاً من استخدام كتل رسومية، مع توفير الأدوات اللازمة لتطوير البرامج وتنفيذها وتصحيحها وتحليلها ضمن واجهة موحدة.

مكوناتها:

بالإضافة إلى المكونات المذكورة سابقا (المحرر، المترجم، المصحح، والمنفذ) فإن هذه البيئات تحتوي كذلك على أداة لإدارة المشاريع والملفات (*Project/File Management*) من أجل تنظيم ملفات المشروع، المكتبات، والوحدات البرمجية. كل هذه الأدوات يتم الوصول إليها عن طريق نافذة موحدة تسمى بـ"واجهة المستخدم" (*User Interface*).

خصائصها:

- تعتمد على كتابة الكود النصي بدقة.
- تمنح مرونة عالية لتطوير برامج معقدة ومشاريع كبيرة.
- غالبًا ما توفر أدوات تحليل الأداء ومراقبة الأخطاء.
- تسمح بدمج المكتبات الخارجية لتوسيع قدرات البرامج.
- مناسبة للمبرمجين الذين يرغبون في التحكم الكامل في البرنامج مقارنة بالبيئات المرئية.

أمثلة مشهورة:

- **البيئة "PyCharm":** مخصصة للغة البرمجة بايثون "*Python*".
- **البيئة "Visual Studio Code":** مخصصة لعدة لغات مثل "*C++*"، "*JavaScript*"، "*Python*"...
- **البيئة "Eclipse":** مخصصة للغتي "*Java*" و"*C++*".
- **البيئة "NetBeans":** مخصصة للغة "*Java*".
- **البيئة "Embarcadero":** مخصصة للغتي "*C++*" و "*Delphi*".

3. بيئة البرمجة Pycharm:

البيئة "*PyCharm*" هي بيئة تطوير متكاملة (*IDE : Integrated Developpment Enverronment*) طوّرتها شركة "*JetBrains*"، وهي مخصصة أساسًا لتطوير البرامج المكتوبة بلغة بايثون "Python "، لكنها ليست مقصورة عليها بالكامل:

الدعم الأساسي :

- لغة *Python* بكل إصداراتها .
- مكتبات *Python* المختلفة مثل*NumPy* :، *Pandas*، *Django*، *Flask*، وغيرها .

الدعم الإضافي :

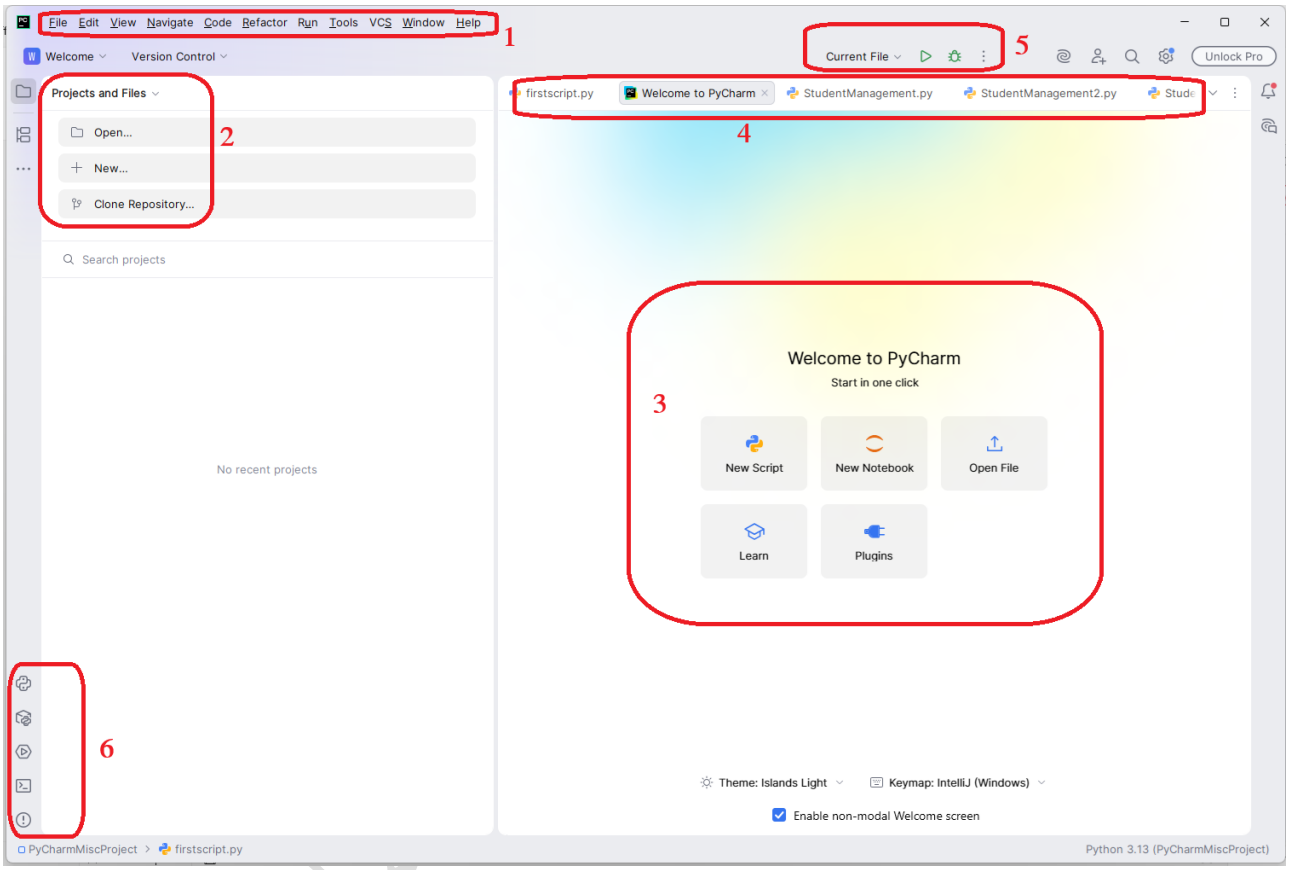
بعض اللغات الأخرى بشكل محدود عبر **الإضافات (*Plagins*)**:

- لغات : *HTML / CSS / JavaScript* المتخصصة في تطوير تطبيقات الويب باستخدام *Python + Frontend* .
- لغة: *SQL* لكتابة استعلامات قواعد البيانات داخل المشروع .

البيئة *Pycharm* مُصممة أساسًا للغة *Python*، ولا تُعد بيئة مناسبة لتطوير Java أو++C ، حتى مع إضافة مكونات إضافية، إذ يبقى دعمها لهما محدودًا مقارنة ببيئات متخصصة.

أ. النافذة الرئيسية لـ Pycharm:

عند فتح البرنامج تظهر الواجهة التالية:



النافذة الرئيسية لـ Pycharm

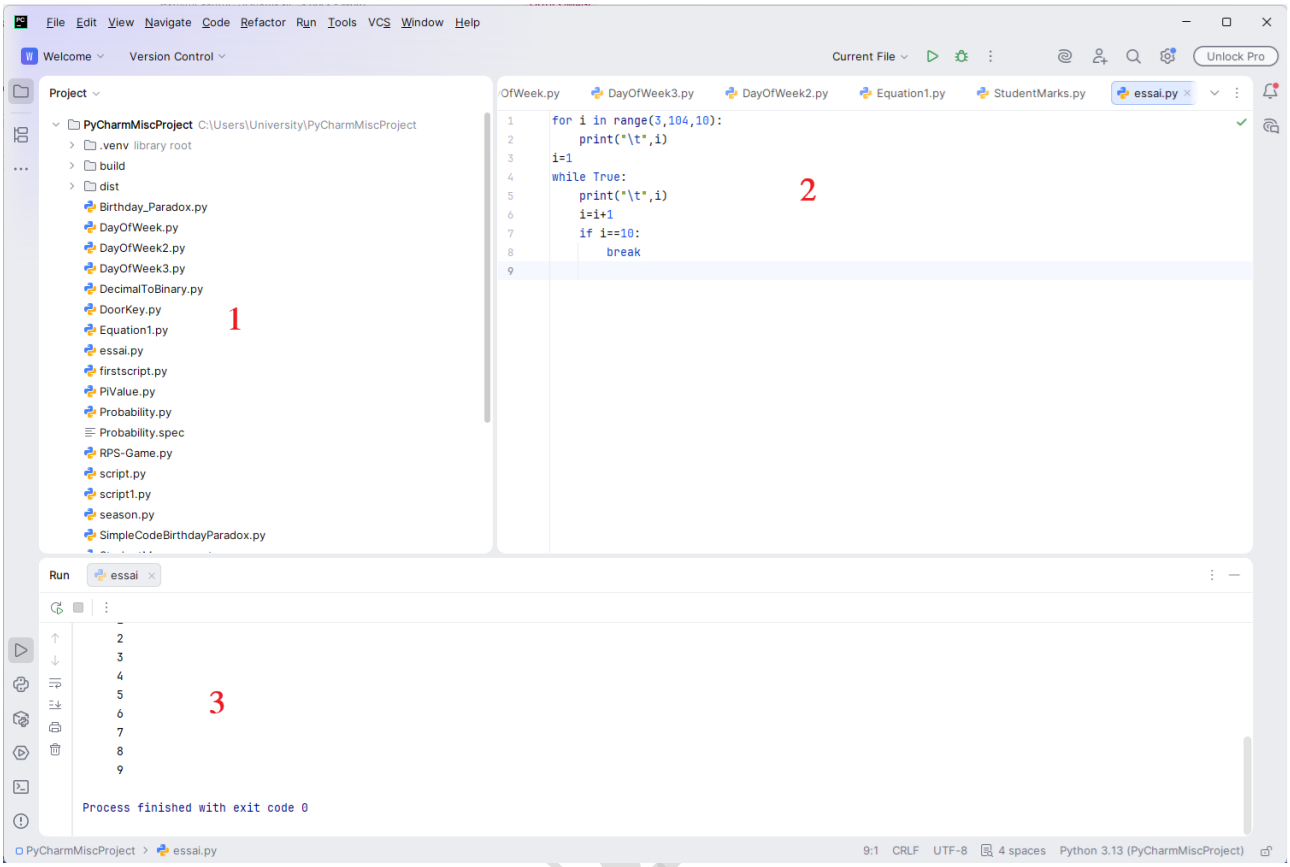
تسمى أيضا بـ"مساحة العمل" وهي الواجهة العامة التي يتم من خلالها التعامل مع البرنامج، وتحتوي على جميع الأدوات والمكونات الأخرى.

أهم مكونات النافذة:

- شريط القوائم "1" (*Menu bar*) : والتي تحتوي على كل ما يمكن القيام به في هذا البرنامج موزعة على عدة قوائم.
- زر التحكم في المشاريع والملفات "*Projects and Files*" رقم "2": التعامل مع المشاريع والملفات المكونة لها.
- أزرار التحكم في البرامج "3": إنشاء برنامج جديد أو فتح برنامج موجود.
- الملفات (البرامج) المفتوحة مؤخرا "4": قائمة بأسماء البرامج التي تم فتحها مؤخرا.
- أزرار التصحيح والتنفيذ "5": تحتوي على زر اختيار البرنامج الذي يتم تنفيذه عن الضغط على زر التنفيذ (يستحسن ترك الاسم "*Current file*" حتى يتم تنفيذ البرنامج المفتوح حاليا، بالإضافة إلى زر التنفيذ وزر التصحيح.
- زر فتح بعض النوافذ الأخرى "6" مثل نافذة الأوامر "*Command line*"، نافذة الخدمات "*Services*" نافذة الإضافات "*Packages*"، ...

ب. النافذة عند فتح برنامج:

عند فتح برنامج موجود أو إنشاء برنامج جديد تظهر النافذة التالية:



نافذة Pycharm عند فتح برنامج

أهم مكونات النافذة:

- قائمة البرامج "1": وهي تمثل قائمة بأسماء كل البرامج الموجودة داخل المشروع.
- كود البرنامج "2": وهي النافذة المخصص لكتابة كود البرنامج "محرر الشيفرة".
- نافذة النتائج "3": وهي النافذة التي تظهر فيها النتائج عند طلب تصحيح أو تنفيذ البرنامج.

4. لغة بايثون Python:

لغة بايثون (*Python*) هي لغة برمجة عالية المستوى (أنشئت سنة 1991 من طرف المبرمج الهولندي ***Guido van Rossum***) تُستعمل لكتابة البرامج والتطبيقات بطريقة سهلة وواضحة. ميزتها أنها قوية وسهلة التعلم وتُعد من أكثر اللغات استعمالًا في العالم اليوم (نظرا لاحتوائها على الكثير من المكتبات الجاهزة، ولأنها مناسبة لمجالات عديدة وبالأخص الذكاء الاصطناعي).

أساسيات لغة بايثون:

1) عموميات:

- **المتغيرات (variables):** هي أماكن في الذاكرة لتخزين المعلومات. في بايثون لا يوجد تصريح مسبق للمتغيرات كما في بعض اللغات مثل C أوJava أو في الخوارزميات، بل المتغيرة تُنشأ بمجرد إعطائها قيمة، وهذه القيمة هي التي تمنحها النمط.

مثال: التعليمة $A = 3$ تنشئ متغيرة A من نوع عدد صحيح قيمته 3.

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

- **التعليقات (Comments):**

التعليقات هي عبارات تُكتب داخل الشيفرة (بأي لغة طبيعية) لشرح وظيفة تعليمة أو مجموعة من التعليمات، وهي لا تُنفذ من طرف الحاسوب ولا تظهر للمستخدم، لكنها تُعد عنصرًا أساسيًا لتسهيل فهم منطق البرنامج، سواء بالنسبة للمبرمج نفسه أو لغيره.

لإدراج التعليقات نستعمل الرمز # عند بداية كل تعليق.

مثال:

```
# This is a comment
if m<=2:
    m = m + 12 # and this one too
    y -= 1
```

- **المسافة البادئة (التسنين) (Indentation):** تُستعمل لتنظيم الشيفرة باستخدام الفراغات (*Spaces* أو *Tabs*)، بحيث تُكتب كل تعليمة في سطر مستقل، وتكون التعليمات التابعة لنفس البنية مصطفّة في نفس العمود مع إزاحة إلى الداخل بالنسبة لبداية البنية، وتنتهي هذه البنية عند ظهور تعليمة جديدة في نفس مستوى بدايتها.

مثال:

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

لاحظ:

- التعليمتين (m = m + 12) و(y - = 1) تابعتين للتركيبة (if).
- التعليمة (k = y % 100) ليست تابعة لها.

2) العمليات والتعليمات:

شكل العمليات والتعليمات في بايثون يشبه إلى حد كبير شكل العمليات والتعليمات في لغتي *C* وJava.

- **العمليات الحسابية:** الجمع "+"، الطرح "-"، الضرب "*"، القسمة الحقيقية "/"، القسمة الصحيحة "//"، باقي القسمة "%"، الأس "**" (2**3 معناها 3 أس 2).
- **تعليمة الإسناد (*Affectation*) (*Assignment*):** للدلالة على تعليمة الإسناد في هذه اللغة نستعمل رمز المساواة " = " كما في اللغتين المذكورتين سابقا، مثال: m = m +12.

ملاحظة:

- للدلالة على المساواة نستعمل "==" وعلى عدم المساواة نستعمل " !=".
- يمكن اختصار التعليمات التي تتضمن تغيير قيمة متغيرة والاحتفاظ بالنتيجة في نفس المتغيرة على النحو التالي:
  - التعليمة: X = X + 2 يمكن كتابتها على الشكل: X += 2.

- التعليمة: $p = p * i$ يمكن كتابتها على الشكل: $p *= i$.
- التعليمة: $i = i - 1$ يمكن كتابتها على الشكل: $i -= 1$.
- التعليمة: $q = q/2$ يمكن كتابتها على الشكل: $q /= 2$.

– في بايثون 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.

- **تعليمة الإدخال (input) :** تعطي دائما نتيجة من نمط "سلسلة حروف".

مثال:

```
A=input("Enter the value of A: ")
B=int(input("Enter the value of B: "))
C=float(input("Enter the value of C: "))
V=bool(input("Enter a Boolean value: "))
```

لاحظ:

– تعليمة القراءة في بايثون بإمكانها إظهار نص من أجل توضيح ما هو مطلوب من المستخدم.

– المتغيرة A من نوع سلسلة حروف.

– المتغيرة B هي عدد صحيح، لذا توجب سبق تعليمة القراءة بالكلمة (الوظيفة) int التي ستقوم بتحويل ما يدخله المستعمل إلى عدد صحيح.

– المتغيرة C هي عدد حقيقي، لذا توجب سبق تعليمة القراءة بالكلمة $float$ التي ستقوم بتحويل ما يدخله المستعمل إلى عدد حقيقي.

– المتغيرة V هي عدد منطقي: الوظيفة $bool$ تحول ما كل ما يدخله المستعمل إلى قيمة $V=True$ ماعدا إذا لم يدخل أي قيمة فستكون $V=False$.

- **تعليمة الإخراج (print):** تسمح بإظهار قيمة متغيرة، نص عادي أو نص منسق.

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.

3) التركيبات:

أ. الشروط:

- **التركيبة "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:

- وجود تركيبة جديدة "*if-elif*" والتي هي في هذا المثال عبارة عن إدماج ثلاثة تركيبات من "*if*" البسيطة في تركيبة واحدة.
- وجود النقطتين ضروري عند نهاية كل سطر (*if*)، (*else*) أو (*elif*).

- **التركيبة "حسب" (match):**

```
match h:
    case 0: print("zero")
    case 1: print("one")
    case 2: print("two")
    case _: print("False value")
```

لاحظ:

- استعمال (**case** _) للدلالة على الحالات الأخرى (*else*).

ب. الحلقات:

- **الحلقة "من أجل"(for):**

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

لاحظ:

- استعمال كلمة (range) للدلالة على المجال من 0 إلى n. في هذه الحالة i تأخذ القيم من 0 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)
```

- **الحلقة "طالما" (while):**

لها نفس الشكل كما في الخوارزميات.

مثال:

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

ملاحظة:

- الحلقة "كرر" (*repeat*): 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 دائما (*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 وُجدت 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 "حاول" (try):

على غرار معظم لغات البرمجة الحالية، تتوفر لغة بايثون على التركيبة "try" والتي تستعمل من أجل معالجة الأخطاء (*Exception Handling*)، دون أن يتوقف البرنامج فجأة.

فمثلا عندما نريد قراءة عدد قد يخطئ المستخدم ويدخل حرفا. وبدون وجود هذه التركيبة فإن البرنامج سيتوقف عن العمل بصفة مفاجئة مما قد يتسبب في تلف البيانات.

من أجل تجنب مثل هذه الخطاء ينصح دائما باستعمال هذه التركيبة كلما ترك المجال للمستخدم من أجل إدخال قيمة، أو عند التعامل مع ملفات موجودة في الجهاز، أو عند استقبال معلومات من برامج أخرى محلية أو عبر الشبكات.

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

- قد يدخل المستخدم قيمة معدومة، وهي قيمة مقبولة (لأنها عدد صحيح) لكنها ستنتج خطأ عندما يحاول البرنامج طباعة نتيجة القسمة 10/x، في هذه الحالة كذلك يتوقف البرنامج عن تنفيذ هذه التعليمات ويقفز مباشرة لتنفيذ التعليمة الموجودة في الجزء "except" وبالتالي يستمر البرنامج في العمل دون أن يتوقف.

- يمكن أن تحتوي هذه التركيبة على عدة أجزاء "except" وهذا من أجل معالجة بعض أنواع الأخطاء كل على حده، كما يمكن أن تحتوي على جزء "else" ينفذ إذا لم يحدث خطأ، وجزء آخر "finally" ينفذ دائما.

مثال شامل:

```
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")
```

4) هياكل البيانات في بايثون:

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 (bool):**

مثال:

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

وهي تقبل العمليات المنطقية : *and*، *or*، *not*.

يمكن تحويل أي محتوى إلى قيمة منطقية باستعمال التعليمة *bool*، مثل: *print(bool("Algiers"))*، حيث تُرجع 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 تُكتب 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
```

أهم الدوال (الوظائف)المستعملة مع النصوص:

نفرض: txt = " python language "

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

جدول بأهم الدوال المستعملة مع النصوص

ملاحظة: يمكن التعرّف على العديد من الدوال الأخرى في PyCharm عند كتابة اسم متغير من نوع نصي متبوعًا بنقطة، حيث تظهر قائمة بالخيارات المتاحة. هذه الميزة متوفرة مع كل أنواع المتغيرات لمعرفة الدوال الخاصة بها.

ب. الهياكل المركبة (Complex):

- **القوائم (List):**

وهي مجموعة مرتبة من العناصر، قابلة للتغيير.

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

- لإضافة عنصر في الأخير : *append*
- لحذف عنصر: remove
- لإدخال عنصر : insert

وهناك العديد من الوظائف الأخرى يمكن معرفتها مباشرة من خلال واجهة Pycharm.

- **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 بسيطا (ذو بعد واحد) نستعمل القائمة:

مثال:

T = [10,20,30,40]

- مؤشر العناصر يبدأ من الصفر. مثلا: T[2] يدل على القيمة 30.
- على عكس معظم اللغات، الجداول في بايثون ليست محدودة مسبقا من ناحية الطول.

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

```
mx = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
 ]
```

- المؤشرات تبدأ من 0.
- المؤشر الأول يدل على رقم السطر والثاني على العمود:

mx[2,1] يدل على القيمة 8.

مثال: قراءة عناصر مصفوفة متكونة 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 وCol عمود.
- الحلقات للسماح للمستخدم بإدخال القيم التي يريدها.
- الدالة str لتحويل عدد إلى نص.

- **القاموس (السجل) (*Dictionary*) :**

القاموس هو هيكل بيانات يُمَكِن من الاحتفاظ بعدة معلومات مختلفة النمط في هيكل واحد. يحتفظ بالمعلومات على شكل أزواج (مفتاح-قيمة) (key-value) ،ويتميز بالوصول السريع للبيانات عن طريق المفتاح. وهو البديل لـ struct في لغة C أو 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.

5. الإجراءات والدوال (الوظائف) (Procedures and Functions):

في البرمجة والخوارزميات، الإجراءات والدوال هي أدوات أساسية لتقسيم البرنامج إلى وحدات صغيرة يمكن إعادة استخدامها، مما يسهل كتابة البرامج الكبيرة، تنظيمها، وتصحيح أخطائها.

أ. تعريف:

1. الإجراء (Procedure):

الإجراء هو وحدة برمجية تنفذ مجموعة من التعليمات، يمكن استدعاؤها أكثر من مرة، ولا ترجع قيمة ناتجة.

- الهدف: تنفيذ مهمة محددة داخل البرنامج .
- يمكن أن تأخذ معطيات (Parameters) كمدخل.
- لا ترجع قيمة بشكل مباشر (في بعض اللغات)

2. الدالة (الوظيفة) (Function):

الدالة هي وحدة برمجية تؤدي مهمة محددة لكنها ترجع قيمة يمكن استخدامها لاحقًا.

- الهدف: إعادة قيمة معينة بعد معالجة المعطيات.
- تأخذ معطيات (Parameters) وتعيد قيمة (Return Value).

3. مزايا استعمالهما:

- النمطية(*Modularity*): تقسيم المشكلة المعقدة إلى مشكلات فرعية بسيطة.

- **التجريد (*Abstraction*):** عزل جزء من البرنامج (الخوارزم) في وحدة مستقلة، له اسم يدل على وظيفته، والتعامل معه دون الاهتمام بالتفاصيل.
- **إعادة الاستخدام:** كتابة الكود مرة واحدة واستخدامه آلاف المرات.
- **سهولة الصيانة:** إذا حدث خطأ في الحساب، نقوم بتصحيحه في مكان واحد فقط (داخل الدالة أو الاجراء).

ب. الإجراءات والدوال في الخوارزميات:

1. الإجراء:

مثال1 : تبديل قيمتي متغيرتين (Permutation):

التصريح بالإجراء (*Procedure declaration*):

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

- المتغيرات X وY تسمى المتغيرات الوسيطة (المعاملات) (المتغيرات الشكلية) (arguments). وهي بمثابة مدخلات لهذا الإجراء.
- المتغيرة Z هي متغيرة محلية.

الاستدعاء (الاستعمال) (*Procedure call*):

بعد التصريح بهذا الإجراء، يمكن الآن استدعاء هذا الإجراء في كل مرة نحتاج فيها إلى التبديل بين قيمتي متغيرتين من نوع عدد صحيح.

لاحظ:

```
Algorithm  algo
    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
```

- التصريح بالإجراءات يكون بعد المتغيرات (أي أن الكود المكون للإجراء يدمج ضمن تصريحات الخوارزم).
- السطر *Permutation(A,B)* عبارة عن استدعاء لهذا الإجراء، يطبق على المتغيرتين A و B.
- يمكن تطبيق هذا الإجراء من أجل التبديل بين أي متغيرتين من نوع عدد صحيح.
- في الاستدعاء الثالث هذه الإجراء يقوم بالتبديل بين عنصرين داخل الجدول T.
- استعمال كلمة var قبل المتغيرتين X وY يدل على ما يسمى بالتمرير بالعنوان (بالمرجع)

تمرير المعاملات (Arguments) في الإجراءات:

هناك طريقتين لتمرير المعاملات في الإجراءات:

التمرير بالعنوان (بالمرجع) (Pass by address) (Pass by reference):

هو أسلوب في استدعاء الإجراء يتم فيه تمرير مرجع (reference) المتغير الفعلي إلى الإجراء، بحيث يصبح المعامل الشكلي (الوسيط) داخل الإجراء مرادفًا (Alias) لنفس موقع الذاكرة الخاص بالمتغير الأصلي. وبالتالي، فإن أي تعديل يُجرى على المعامل داخل الإجراء ينعكس مباشرة على المتغير الأصلي خارجه .

للدلالة على التمرير بالمرجع نضيف كلمة var قبل ذكر المتغيرات المعنية بهذا النوع من التمرير.

التمرير بالقيمة (Pass by value):

هو أسلوب يتم فيه تمرير نسخة من قيمة المتغير الفعلي إلى المعامل الشكلي داخل الإجراء. وبالتالي، فإن أي تعديل يُجرى على المعامل داخل الإجراء لا يؤثر على قيمة المتغير الأصلي خارجه .

لاحظ:

في المثال السابق لولا استعمال التمرير بالمرجع (استعمال كلمة var قبل ذكر المعاملات X وY لما حدث تبديل للمتغيرات الحقيقية في الخوارزم.

مثال2:

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

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

- قيمة X التي سيظهرها الإجراء هي 30.
- قيمة المتغيرة A التي سيظهرها الخوارزم هي 20، لم تتغير لأن التمرير كان بالقيمة فقط.

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 (Function call):

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

```
Algorithme  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 الحالية، فإنّ لغة بايثون لا تُميّز بشكل صريح بين مفهومي الدالة (***Function***) والإجراء (***Procedure***)كما هو الحال في بعض اللغات التقليدية. ففي التصور الكلاسيكي، تُعرَّف الدالة على أنها وحدة برمجية تُرجِع قيمة نتيجة معالجة معيّنة، في حين يُعرَّف الإجراء على أنه مجموعة من التعليمات تُنفَّذ دون إرجاع قيمة.

غير أنّ هذا التمييز يتلاشى في بايثون، حيث تعتمد اللغة مفهومًا موحّدًا يتمثل في الدالة، التي يمكن أن تُستخدم لإرجاع قيمة أو لتنفيذ تعليمات فقط. فعند غياب تعليمة return، تُرجِع الدالة ضمنيًا القيمة None، وهو ما يجعلها تؤدي عمليًا وظيفة الإجراء دون الحاجة إلى تعريف مستقل له.

وبناءً على ذلك، يمكن اعتبار مفهوم الإجراء في بايثون حالة خاصة من الدالة، مما يجعل الفصل بينهما أقرب إلى كونه تمييزًا بيداغوجيًا يُستعمل لأغراض تعليمية، أكثر من كونه تمييزًا تقنيًا حقيقيًا داخل اللغة.

- يتم التعريف بالدالة في لغة بايثون باستعمال الكلمة "def".
- الكلمة return التي تدل على القيمة التي ترجعها الدالة.
- يمكن التصريح بالدوال في بايثون 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، str، tuple، bool التمرير يكون بالقيمة، أما بالنسبة للأنماط: list، dict، set (أنماط قابلة للتغيير) فالتمرير يكون بالعنوان.**
- **إذا أردنا الاحتفاظ بقيمة المتغيرات ذات الأنماط: int، float، str، tuple، bool (استعمال التمرير بالمرجع على هذا الأنواع) يجب استعمال return بقيمة المتغيرات المراد الاحتفاظ بقيمها مع استعمال الطريقة الصحيحة لاستدعاء الدوال.**

ث. مختلف الأشكال الممكنة لاستدعاء الإجراءات والدوال:

كل برنامج رئيسي (main program) أو خوارزم يمكن أن يحتوي على عدة إجراءات أو دوال. هذه الإجراءات والدوال تعتبر أبناء مباشرة (direct descendants) للبرنامج الرئيسي. كما أن أي إجراء أو دالة يمكن أن يحتوي على أبناء مباشرين.

انطلاقًا من هذه البنية الهرمية، يمكن تمييز عدة أشكال لاستدعاء الإجراءات والدوال، نذكر منها:

1. **الاستدعاء من البرنامج الرئيسي (المباشر) (*Direct Call from Main Program*):** في هذا الشكل، يقوم البرنامج الرئيسي باستدعاء إجراء أو دالة مباشرةً لتنفيذ مهمة محددة (ويعد هذا الشكل أبسط أنواع الاستدعاء وأكثرها استعمالًا).

مثال: البرنامج الرئيسي يستدعي الدالة AverageCalculating() .

2. **الاستدعاء بين الإجراءات والدوال(*Inter-Procedure Calls*) :** يمكن لإجراء أو دالة أن تستدعي إجراءً أو دالةً أخرى، مما يسمح بتقسيم المشكلة إلى أجزاء أصغر وأكثر تنظيمًا. يساعد هذا النوع على تحقيق مبدأ التجزئة (Modularity) وإعادة استعمال الشيفرة.

مثال: الدالة DataProcess() تستدعي الدالة ReadData() ، ثم تستدعي DisplayResult() .

3. **الاستدعاء المتسلسل(*Sequential Calls*) :** حيث يتم استدعاء عدة إجراءات أو دوال بشكل متتابع لتنفيذ سلسلة من العمليات. هذا النمط شائع في البرامج التي تعتمد على مراحل معالجة واضحة.

مثال: main → read() → process() → display()

4. **الاستدعاء الشرطي(*Conditional Calls*) :** يتم استدعاء إجراء أو دالة 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 الإجراء أو الدالة عدة مرات داخل حلقة. لا يجب الخلط بين هذا النوع والاستدعاء الذاتي التكراري (العودي) (recusrsion).

مثال: استدعاء دالة readmarks() لكل طالب داخل حلقة.

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 حلات عامة تحسب بنفي الطريقة (Recusrsive cases).

مثال:

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

7. **الاستدعاء المتبادل(*Mutual Recursion*) :** حيث تقوم دالتان أو أكثر باستدعاء بعضها البعض، وهو أقلها استعمالا.

مثال: الدالة A تستدعيB والدالة B تستدعيA .

6. الخلاصة:

في هذا الدرس تناولنا مختلف المراحل التي تسمح بالانتقال من فكرة حل مشكلة إلى برنامج قابل للتنفيذ، حيث تم التطرق إلى:

- تعريف الخوارزمية والبرنامج وبيان العلاقة بينهما، مع عرض مراحل تحويل الخوارزمية إلى برنامج.
- التعرف على بيئات البرمجة بنوعيها: المرئية والنصية، مع تقديم بيئة PyCharm كأداة لتطوير برامج بايثون.
- دراسة أساسيات لغة Python من حيث المتغيرات، أنواع البيانات، والتعليمات الأساسية.
- فهم مفهوم الإجراءات والدوال في الخوارزميات وفي لغة بايثون، مع التمييز بينهما.
- استكشاف مختلف أشكال استدعاء الدوال والإجراءات، بما في ذلك الاستدعاء البسيط، المتداخل، والتكراري.

وبذلك يهدف الدرس إلى تمكين المتعلم من تحويل خوارزمية بسيطة إلى برنامج منظم ووظيفي باستخدام لغة بايثون.