1- أنواع المتغيرات:

ا- الأرقام :

- int الاعداد الصحيحة -2 1
- floatالاراقام الكسرية او المضروبة ب 10 مرفوعة لقوة ما 5E4 مثلا او 2.0

ب- العمليات على الأرقام :

- الجمع +
- الطرح –
- الضرب *
- التقسيم /
- التقسيم مع الدالة الأرضية //
- الباقي من التقسيم (MOD) %
- القوة او الجذر **
- الأولوية للاقواس ثم للضرب والتقسيم ثم من اليسار لليمين

ج- تعريف المتغيرات :

- لا يمكن بدأ اسم المتغير برقم
- لا يمكن ان يحتوي على فراغ بل نستخدم _
- لا يمكن استخدام هذه الرموز :"',<>/?|\()!@#$%^&*~-+
- لا يمكن استخدام الكلمات المحجوزة مثل int list...

لغة البايثون سهلة التعامل مع المتغيرات من الأنواع المختلفة أي لا تحتاج الى تعريف نوع المتغيرفي نفس الوقت يجب الحذر لكي لا يكون نوع المتغير غير المطلوب يمكن استخدام الدالة type() لمعرفة نوع المتغير

د- النصوص (strings): لغة البايثون تتعامل مع النصوص على انها سلاسل احرف لذلك يمكننا الوصول لاي حرف من الكلمة بسهولة

يمكن تعريف النصوص ضمن "" او '' لتعطي إمكانية لاستخدام هذه الإشارات صمن النص مثلا "I'm taha"

لو استخدمنا ' لبقيت الكلمات بعد الحرف الأول خارج النص وتسبب خطأ

يمكن استخدام الدالة print() لطباعة النصوص والتعامل معها

نستخدم \n ضمن النص لبدأ سطر جديد

الدالة len() تعطي عدد مكونات النص

للوصول الى أي جزء من النص يتم تعريف النص كمتغير واستخدام رقم الحرف المراد الوصول اليه ضمن []

والعد يبدأ من 0 او من -1 للعد من اليمين

A= "taha elhariri"

Out[2]: A[0]

Out[2]: ‘t’

A[-1]

Out[2]: ‘i’

يمكن التعامل مع اكثر من جزء من النص عن طريق استخدام :

A[:3]

Out[2]: ‘tah’

هنا من البداية حتى المحتوى الثالث ولكن الثالث لا يتم استدعاؤه

A[5:]

Out[2]: ‘ elhariri’

A[2:-5]

Out[2]: 'ha elh'

الرقم الثالث ضمن الاقواس يدل على الخطوة

و- التعامل مع المتغيرات ضمن النصوص:

- استخدام % : print("I'm going to inject %s text here, and %s text here." %('s1','s2'))
- %s تقابل الدالة str() للتحويل الى نص
- %r تقابل الدالة repr() للتحويل نص مع إبقاء علامات التنصيص
- \t لابقاء فراغ بمقدار tab
- %d للطباعة الأرقام الصحيحة فقط
- %5.8f لطباعة الأرقام مع الفواصل حيث 5 هي عدد الأرقام قبل الفاصلة و8 بعدها ويمكن تغيير هذه الاراقام

د- القواميس(Dictionary)

```
my_dict = {'key1':'value1','key2':'value2'}
```

- يمكن وضع اكثر من نوع فيه
- يمكن التعديل على القيم
- لا يمكن وضع متغيرات
- يمكن كتابته فارغ
- يمكن كتابة اكثر من قاموس
- النتيجة عن keys() هي عبارة عن opject فالافضل تحويلها الى list

للتوصل الى قيم keys,value على حدة :

```
L1= [ ]
L2= [ ]
For x,y in my_dict.items():
L1.append(x)
L2.append(y)
```

Lists

عناصر القائمة محاطة بأقواس مربعة

عناصرها مرتبة ، لاستخدام الفهرس للوصول إلى العنصر

القائمة قابلة للتغيير => إضافة ، حذف ، تحرير

عناصر القائمة يمكن تكرارها

يمكن أن تحتوي القائمة على أنواع بيانات مختلفة

يمكن وضع اكثر من واحدة متداخلة

```
mylist=["one","tow","one",1,100.5,True]
print(mylist)
']one', 'tow', 'one', 1, 100.5, True[
```

Tuples

العناصر محاطة بأقواس

يمكنك إزالة الأقواس إذا أردت

عناصرها مرتبة ، لاستخدام الفهرس للوصول إلى العنصر

غير قابل للتغيير => لا يمكنك الإضافة أو الحذف

عناصر القائمة لا يمكن تكرارها

يمكن أن يكون لدى Tuple أنواع بيانات مختلفة

```
mytupleone=("osama","ahmad")
mytupletwo="osama","ahmad"

print(mytupleone)
('osama', 'ahmad')

print(mytupletwo)
('osama', 'ahmad')

print(type(mytupleone))
<class 'tuple'>
print(type(mytupletwo))
<class 'tuple'>
```

Set

عناصر المجموعة محاطة بأقواس مجعدة

العناصر غير مرتبة وغير مفهرسة

الفهرسة والتقطيع لا يمكن القيام به

تحتوي المجموعة على أنواع بيانات ثابتة فقط (أرقام ، سلاسل ، مجموعات) listو Dict ليست كذلك

عناصر القائمة لا يمكن تكرارها

```
setone={"osama","ahmad",100}
print(setone)
#{'ahmad', 'osama', 100}
#print(setone[0])
#error
```

Boolean

القيم المنطقية هي كائنان ثابتان خطأ + صحيح.

Files

Opening:pwd:

يعطي المسار

Open()

فتح الملف

.read()

قراءة الملف ولكن لا يمكن القراءة مرتين

.seek(0)

للقراءة مرة ثانية

.readlines()

قراءة الملفات الكبيرة

.close()

للاغلاق .من اجل الامان

‘W’

‘W+’

مسموح فقط الكتابة

‘a’

مسموح الفتح والاضافة وينشئ ملف ان لم يكن موجود

‘-a’

مسموح الاضافة

لكن تتطلب :

%%writefile

For line in oppen (‘test.txt’)

Print(line)

طريقة ثانية لفتح الملف والفراءة بدون استدعاء

2- العمليات المنطقية:

- == هل يساوي
- =! لا يساوي

- =< اصغر او يساوي
- < اصغر

الناتج من هذه العمليات 1,0 True false

- يمكن ربط اكثر من عملية باستخدام and or او اكثر من علامة 1<2<3 النتيجة True

3- الجمل في بايثون:

- الجمل الشرطية if else:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

الانتباه لوجود tab في السطر التالي للكلمة المفتاحية والبدأ قبله يعني الخروج من الجملة

- الجملة التكرارية for:

```
for item in object:
    statements to do stuff
```

يمكن استخدام عداد ارقام او احرف او قائمة ...

```
list1 = [1,2,3,4,5,6,7,8,9,10]
for num in list1:
  print(num)
```

يمكن استخدام الجمل الشرطية والتكرارية بشكل متداخل مثال : طباعة الاعداد الزوجية والفردية :

```
for num in list1:
  if num % 2 == 0:
    print(num)
  else:
    print('Odd number')
```

يمكن استخدام نص كمتغير في كل دورة :

```
for letter in 'This is a string.':
  print(letter)
```

يمكن إضافة اكثر من متغير في كل دورة :

```
list2 = [(2,4),(6,8),(10,12)]
for (t1,t2) in list2:
  print(t1)
```

يمكن التكرار ضمن مجال والتحكم بسهولة في المعطيات :

```
for a in range(10):
  print(a)
```

النتيجة فقط حتى الرقم 9

range(start,stop,step)

يمكن تعيين البداية والنهاية والخطوة حيث النهاية غير مضمنة

- الجملة التكرارية while :

الفرق بينها وبين for انها تنتهي بانتهاء الشرط المضمن لا بتحديد مجال

```
while case 1:
   code statements
else:
   final code statements
```

يمكن استخدامها بطريقة مشابهة ل for :

```
x = 0

while x < 10:
   print('x is currently: ',x)
   print(' x is still less than 10, adding 1 to x')
   x+=1
```

استخدام العبارات الإضافية break, continue, pass:

Break تقوم بالخروج من الجملة التي تتضمنها

Continue تكمل الجملة دون إتمام الدورة

Pass تكمل دون فعل شيئ لتفادي وجود جملة شرطية فارغة

```
number = 0

for number in range(10):
   if number == 5:
      break, continue, pass    # one here

   print('Number is ' + str(number))

print('Out of loop')
```

الاختلاف بين هذه العبارت هنا يظهر انه عند استخدام continue ستكون النتيجة بدون طباعة الرقم 5 اما باستخدام pass سيكون وكأنه تم اهمال الجملة الشرطية اما عند استخدام break فسوف يتوقف عند الدورة الخامسة

- استخدام الجمل التكرارية لانشاء قائمة :

```
lst = [x for x in 'word']

['w', 'o', 'r', 'd']
```

يمكن إضافة جملة شرطية :

```
lst = [x for x in range(11) if x % 2 == 0]

[0, 2, 4, 6, 8, 10]
```

يمكن تعريف المتغير الضمني كقائمة باستخدام الجملة التكرارية واستخدامه مباشرة لتعريف قائمة جديدة وذلك يقلل من استهلاك الذاكرة :

```
lst = [ x**2 for x in [x**2 for x in range(11)]]
```

مثال : تحويل قائمة بدرجات الحرارة السليسوس الى فهرنهايت :

```
celsius = [0,10,20.1,34.5]

fahrenheit = [((9/5)*temp + 32) for temp in celsius ]
```

4- بعض الدوال المستخدمة :

- range() يمكن استخدامها لتوليد ارقام ضمن مجال محدد البدأ والنهاية والخطوة
list(range(0,101,10))
- enumerate() تستخدم لانشاء حزم تحتوي على المدخلات ضمن الاقواس مع ترتيبها مثال :

```
a=enumerate('abcde')

list(a)

Out[137]: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

يمكن أيضا تحديد البدأ بالترقيم :

```
months = ['March','April','May','June']

list(enumerate(months,start=3))
[(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

- zip(list1,list2) تستخدم لدمج قائمتين بحسب الترتيب أي يصبح العنصر الأول من كل قائمة في حزمة والثاني وهكذا :

```
list(zip(mylist1,mylist2))

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

استخدام هذه الدالة لعكس قيم قاموس ما:

```
def switcharoo(d1,d2):
   dout = {}

   for d1key,d2val in zip(d1,d2.values()):
```

سيتم وضع القيم مكان المفاتيح وبالعكس dout[d1key] = d2val

return dout

- in تستخدم كعملية منطقية جوابها true false لتحديد ما اذا كان العنصر موجودا في القائمة مثلا او لا :

'x' in ['x','y','z']

True

- max(), Min() للوصول الى العنصر الأكبر او الأصغر ضمن القائمة
- مكتبة random تحتوي على عدد من الدوال مثل randint , shuffle

from random import shuffle

shuffle(mylist) هذه الدالة سوف تقوم بالتعديل مباشرة على القائمة الاصلية دون ارجاع أي قيمة كمتغير

from random import randint

randint(a,b) تعطي رقم صحيح عشوائي ضمن المجال

- Input() لادخال قيمة من المستخدم input('Enter Something into this box: ')

النتيجة تكون نص دائماً

مثال:

لعبة تخمين العدد:

- نجعل البرنامج ينتج رقم عشوائي بين 0 وال 100
- نطلب من المستخدم ادخال رقم ضمن هذا المجال
- اذا كان الرقم خارج المجال يطلب منه إعادة الادخال مرة أخرى
- في المرة الأولى للتخمين اذا كان العدد المخمن قريب للعدد العشوائي 10 او اقل يكتب cold
- اذا كان العدد المخمن بعيد عن العدد العشوائي 10او اكثر يكتب warm
- في المرات التالية اذا كان التخمين الأخير اقرب من التخمين السابق يكتب warmer والا يكتب colder
- في حالة تخمين العدد يكتب لقد نجحت مع عدد مرات التخمين

"""# Let's use while loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"

```
# On a player's first turn, if their guess is
# within 10 of the number, return "WARM!"
# further than 10 away from the number, return "COLD!"
# On all subsequent turns, if a guess is
# closer to the number than the previous guess return "WARMER!"
# farther from the number than the previous guess, return "COLDER!"
# When the player's guess equals the number, tell them they've
guessed correctly and how many guesses it took!"""

from random import randint
a= randint(0,101)
# print(a)
b=0
s=1
d1=0
d2=0
while a!=b:
    b= int(input("inter a number"))
    d2=d1
    d1=abs(a-b)

    if b<0 or b>100:
            print("OUT OF BOUNDS")
            continue
    if s==1:

        if d1>=10:
            print("cooold")
        elif d1<=10:
            print("waaarm")
    else:
        if d1<d2:
            print("warmer")
        else:
            print("colder")
    if a==b:
        print("you've guessed correctly and it took {} guesses !".format(s))
        break
    s=s+1
```

5- الدوال functions:

هو عبارة تستخدم لحفظ عملية معينة يتكرر استخدامها ضمن البرنامج وتستعمل باستدعاء اسم الدالة مع المعطيات اللازمة مثلا ()randint

يتم تعريف الدالة عن طريق الكلمة المفتاحية def ويليها اسم الدالة والمعطيات بين قوسين

وفي الاسطر التالية يتم كتابة ما يراد تطبيقه من عمليات والناتج التي تعطيه هذه الدالة

```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

مثال لطباعة ترحيب :

```
def greeting(name):
    print('Hello %s' %(name))
```

يتم استدعاء هذه الدالة عن طريق كتابة الاسم والمدخلات :

```
greeting(“taha”)
```

سيكون الناتج :

```
hello taha
```

في هذه الحالة البرنامج لا يقوم بارجاع او حفظ أي قيمة أي لا يمكننا حفطه ضمن متغير

A= greeting(“taha”) فارغا aهذا سيعطي خطأ او سيبقى محتوى

اذا كان هناك حاجة لحفظ المتغيرات يمكن ان نستخدم return

```
def add_num(num1,num2):
    return num1+num2
```

في هذه الحالة يمكن حفظ الناتج ضمن متغير

مثال دالة لاختبار هل العدد اولي او لا

```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If never mod zero, then prime
        print(num,'is prime!')
```

ملاحظة كلمة else هنا هي بمحاذاة for لأننا نريد اختبار جميع الأرقام قبل طباعة النتيجة اذا كان العدد اولي

مثال: دالة تأخذ معطيات كقائمة وتعطي true اذا كانت القائمة تحتوي على 007 بالترتيب :

```
def is_007(l):
    for i in range(len(l)-2):
        if l[i]==l[i+1] and l[i]==0:
            if l[i+2]==7:
                return True
    else:
        return False
```

6-دالة map تستخدم لتطبيق دالة ما على اكثر من عنصر مثال:

دالة لحساب مربع عدد ما:

```
def square(num):
    return num**2
```

لو اردنا تطبيق هذه الدالة على قائمة من الاعداد سيظهر خطأ لذلك يمكن استخدام دالة map على الشكل التالي :

```
my_nums = [1,2,3,4,5]
list(map(square,my_nums))
```

يمكن استخدام هذه الدالة أيضا مع اكثر من متغير بالشكل التالي

```
a = [1,2,3,4]
b = [5,6,7,8]
list(map(lambda x,y:x+y,a,b))
```

سيتم تطبيق العملية على كل عنصر من المدخلات

6- دالة filter تستخدم لكي ننشئ قائمة بالمدخلات التي تعطي قيمة true فقط ضمن دالة معينة :
دالة لايجاد الاعداد الزوجية :

```
def check_even(num):
    return num % 2 == 0
```

اذا كان لدينا قائمة باعداد ونريد منها فقط الاعداد الزوجية :

```
nums = [0,1,2,3,4,5,6,7,8,9,10]
```

يمكن وضع الدالة المراد تطبيقها مع المدخلات ضمن دالة :filter

```
List(filter(check_even,nums))
```

النتيجة [0, 2, 4, 6, 8, 10]

7- دالة lambda
يمكن اختصار الكود المستخدم لايجاد مربع عدد بهذا الشكل :

```
def square(num): return num**2
```

فعليا تتيح هذه الدالة اختصارا اخر للانشاء الدوال البسيطة او المعقدة كالتالي:

```
square = lambda num: num **2
```

حيث يمكن استدعاء الدالة الجديدة هكذا :

```
square(m)
```

8- مفهوم local and global:
يستخدم لتحديد مدى تاصير المتغيرات ضمن الجمل أي ان عمل متغير ما او قيمته هي ضمن مجال معين
مثال:

```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'
    def hello():
        global name ضمن هذه الدالة عندما نغير المتغير الى غلوبال يتم اخذ القيمة المعرفة في بداية الكود
        print('Hello '+name)لو لم يتم تعريفها في السطر السابق لكانت القيمة هنا هي قيمة المتغير الأقرب

    hello()

greet()
```

9- *args and **kwargs :
نستخدم كلمة *args في حال كانت معطيات الدالة قابلة للتغير لتجنب الأخطاء فيمكن ادخال مدخل واحد او اكثر حسب رغبة المستخدم :

```
def myfunc(*args): يمكن كتابة أي كلمة أخرى ولكن المتعارف عليه هو هذا
    return sum(args)*.05
```

```
myfunc(40,60,20)
```

def myfunc(*spam): مماثل تماما للسابق

```
    return sum(spam)*.05

myfunc(40,60,20)
```

اما بالنسبة ل **kwargs فهي تنشئ فهرس باستخدام المدخلات :

```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

النتيجة: My favorite fruit is pineapple

ويمكن الاستخدام في نفس الوقت للجملتين ضمن الدالة الواحدة بشرط الترتيب :

```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
myfunc(fruit='cherries',juice='orange','eggs','spam')
```

هذه صحيحة

هذه تعطي خطأ

البرمجة كائنية التوجه

الكائن Objects

كل ما هو موجود في لغة البايثون هو كائن يمكننا استخدام الدالة type() لمعرفة نوع هذا الكائن مثلاً

```
print(type(1))
<class 'int'>
```

الصنف class

يمكننا تعريف الكائنات باستخدام الكلمة المفتاحية class وهي في العادة تستخدم للكائنات ذات الخصائص المشتركة

```
# Create a new object type called Sample
class Sample: تعريف الصنف
   pass

# Instance of Sample
x = Sample()  انشاء الكائن ضمن هذا الصنف

print(type(x))
<class '__main__.Sample'>
```

لكل كائن ضمن الصنف خصائص ودوال يمكن الوصول اليها

الخصائص Attributes

يتم استدعاء الخصائص بالشكل التالي self.attribute = something اما تعريفها ضمن الصنف فيتم عن طريق الكلمة الخاصة __init__()

```
class Dog:  تعريف صنف باسم معين
   def __init__(self,breed): تحديد خصائص هذا الصنف
      self.breed = breed

sam = Dog(breed='Lab') انشاء كائن
frank = Dog(breed='Huskie')
sam.breed استدعاء الخاصية
'Lab' : النتيجة
```

يوجد صفات خاصة بالصنف مثلا ان الكلاب من الثديات يمكن اضافتها خارج الكلمة المفتاحية __init__()

```
class Dog:

   # Class Object Attribute
   species = 'mammal'  التعامل معها مشابه للبقية

   def __init__(self,breed,name):
      self.breed = breed
      self.name = name
sam = Dog('Lab','Sam')
sam.species
'mammal'
```

الفرق هنا يكمن عند تعريف الكائن لا نحتاج الى تعريف هذه الخاصية

العمليات Methods

هي الدوال المضمنة في الصنف وتستخدم لتطبيق عملية معينة على الكائن بحسب الخصائص المعينة له

```
class Circle:
```

```
  pi = 3.14  هذه خاصية لكل الدوائر لذلك يتم تعريفها خارج أي دالة

  # Circle gets instantiated with a radius (default is 1)
  def __init__(self, radius=1): 1 تعريف القطر الافتراضي الى
    self.radius = radius
    self.area = radius * radius * Circle.pi  حساب المساحة الافتراضية

  # Method for resetting Radius
  def setRadius(self, new_radius): دالة لادخال قطر جديد
    self.radius = new_radius
    self.area = new_radius * new_radius * self.pi حساب المساحة الجديدة

  # Method for getting Circumference
  def getCircumference(self): دالة لحساب محيط الدائرة
    return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
Radius is:  1
Area is:  3.14
Circumference is:  6.28
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

التركة Inheritance

يتم هذا الامر حين نعرف صنف جديد تابع لصنف اخر مثلا صنف الحيوانات يمكن ان يكون تحته صنف الكلاب كما في المثال التالي:

```
class Animal:  تعريف الصنف الأول
  def __init__(self):
    print("Animal created")  ستعمل تلقائيا بعد انشاء الكائن

  def whoAmI(self): انشاء دوال
    print("Animal") يتم طبعاتها عند  استدعاء الدالة

  def eat(self):
    print("Eating")
```

```
class Dog(Animal):  انشاء الصنف جديد مع توريث خصائص ودوال الصنف القديم
  def __init__(self):
    Animal.__init__(self)  بدون هذا السطر لن يتم استيراد الخصائص في صنف الحيوانات
    print("Dog created")

  def whoAmI(self): ضمن هذا السطر يتم تحديث الدالة الموجودة في الصنف القديم دون ان تؤثر عليها فقط في الصنف
الجديد ستعمل هذه الدالة وليس الدالة القديمة
    print("Dog")

  def bark(self): يمكن إضافة دالة جديدة أيضا كهذه
    print("Woof!")
d = Dog()
Animal created خاصية الصنف الأول
Dog created خاصية الصنف الجديد
d.whoAmI()          الدالة المحدثة
Dog
d.eat() الدالة من الصنف القديم
Eating
d.bark() الدالة من الصنف الجديد
Woof!
```

يمكننا أيضا استدعاء الخصائص من الصنف الأول في الصنف الثاني

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

class Bear(Animal):
  def __init__(self,name,legs=4,hibernate='yes'):
    Animal.__init__(self,name,legs) هنا سيتم تعريف الخصائص الموجودة في صنف الحيوانات ضمن صنف الدببة
    self.hibernate = hibernate
```

يمكن تطبيق هذا الامر في حال اردنا استيراد خصائص من صنفين او اكثر الى صنف ثالث

```
class Car:
  def __init__(self,wheels=4): نعرف صنف للسيارات يكون الوضع التلقائي 4 عجلات
    self.wheels = wheels
class Gasoline(Car): تعريف صنف محركات الوقود
  def __init__(self,engine='Gasoline',tank_cap=20):
    Car.__init__(self)
    self.engine = engine
    self.tank_cap = tank_cap
    self.tank = 0

  def refuel(self): دالة تعبئة الوقود
    self.tank = self.tank_cap


class Electric(Car): تعريف صنف سيارات الكهرباء
  def __init__(self,engine='Electric',kWh_cap=60):
    Car.__init__(self)
```

```
    self.engine = engine
    self.kWh_cap = kWh_cap
    self.kWh = 0

  def recharge(self): دالة الشحن
    self.kWh = self.kWh_cap
```
الان يمكننا تعريف صنف سيارة هجينة تعمل بكلا المحركين
```
class Hybrid(Gasoline, Electric):
  def __init__(self,engine='Hybrid',tank_cap=11,kWh_cap=5):
    Gasoline.__init__(self,engine,tank_cap)
    Electric.__init__(self,engine,kWh_cap)
```
في هذه الحالة اذا تم تعريف كائن من نوع مهجن فسيحمل صفات كائن السيارة والكهرباء والوقود في نفس الوقت
عند استدعاء الدوال المضمنة في الصنف فنحن فعليا نقوم بامر مشابه فيما لو قمنا بكتابتها هكذا
```
Hybrid.recharge(a)
```
هناك أيضا دالة تبين لنا تفاصيل عملية التركة في الأصناف وهي Method Resolution Order (MRO)
```
class A:
  num = 4

class B(A):
  pass

class C(A):
  num = 5

class D(B,C):
  pass
```
الان عند استدعاء القيمة سيقوم بجلب اخر قيمة تم تعيينها في الأصناف وهي 5 وليس 4 D.num
يمكن أيضا معرفة محتوى الصنف هكذا D.mro()
```
Out[152]: [__main__.D, __main__.B, __main__.C, __main__.A, object]
```
الدالة super() هي دالة مضمنة تتبع التنسيق السابق في استيراد الخصائص أي انها تستورد اخر دالة تم تعريفها او اخر خاصية في الخصائص ذات الاسم المشترك
```
class A:
  def truth(self):
    return 'All numbers are even'

class B(A):
  pass

class C(A):
  def truth(self):
    return 'Some numbers are even'
class D(B,C):
  def truth(self,num):
    if num%2 == 0:
      return A.truth(self)  كما ذكرنا سابقاً يتم استدعاء الامر بهذه الطريقة من الدالة الأولى
    else:
      return super().truth() اما في هذه الحالة فسيتم استيراد اخر ما تم تعريفه
```

تعدد الاشكال Polymorphism

هو قابلية ان تتشارك الكائنات من أصناف مختلفة نفس اسم الخاصية

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

  def speak(self):
    return self.name+' says Woof!'

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

  def speak(self):  تعريف نفس اسم الخاصية في الصنف السابق
    return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak()) استدعاء نفس اسم الخاصية والنتيجة مختلفة لكل كائن
print(felix.speak())
Niko says Woof!
Felix says Meow!
```

يمكن رؤية الفرق بشكل أوضح في المثال التالي :

```
def pet_speak(pet): تعريف دالة لاستدعاء الدالة من الصنف
  print(pet.speak())

pet_speak(niko) ستحتلف النتيجة باختلاف نوع الكائن المدخل
pet_speak(felix)
Niko says Woof!
Felix says Meow!
```

هناك حالات يكون فيها حاجة لتعريف دالة ما ضمن الصنف الأول وتضمين خطـ في حال عدم وجود هذه الدالة ضمن الصنف الجديد

```
def speak(self):          # Abstract method, defined by convention only
    raise NotImplementedError("Subclass must implement abstract method") هنا سيظهر رسالة خطأ في
                                                                          حال عدم تعريف الدالة ضمن الصنف الجديد
print(fido.speak())
 File "C:\Users\tahae\trCovid 19\untitled1.py", line 14, in speak
  raise NotImplementedError("Subclass must implement abstract method")
NotImplementedError: Subclass must implement abstract method  يتم طباعة ما بين القوسين هنا
```

بعض العمليات الخاصة في الأصناف class

هناك بعض الدوال الخاصة التي لا يتم استدعاؤها كما يتم استدعاء الدوال التي ذكرناها سابقا مثل d.bark() بل هي مرتبطة بكلمات خاصة في اللغة مثال :

```
class Book:
  def __init__(self, title, author, pages):  نعرف الصنف والخصائص كما كنا نفعل من قبل
    print("A book is created")
    self.title = title
```

```
        self.author = author
        self.pages = pages

    def __str__(self): هذه الدالة تعمل عند استخدام أوامر الطباعة
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pages)

    def __len__(self): هذه الدالة نعمل مع دالة len التي استخدمناها من قبل
        return self.pages

    def __del__(self): هذه الدالة تستخدم لحذف الكائن
        print("A book is destroyed")
book = Book("Python Rocks!", "Jose Portilla", 159) عند تعريف الكائن ستتم طباعة ما تم تضمينه في  الخصائص
مباشرة
A book is created
print(book) هذه الأوامر الثلاث هي التي تقوم باستدعاء الدوال الخاصة
print(len(book))
del book
Title: Python Rocks!, author: Jose Portilla, pages: 159 نتيجة امر الطباعة
159 نتيجة امر الطول
A book is destroyed نتيجة امر حذف
```

مثال على استخدام الصنف انشاء صنف حساب بنكي يحتوي اسم المستخدم وقيمة المال المودع مع دالتين للسحب والايداع

```
class account:

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

    def deposit(self,quantity):

        self.balanc=quantity+self.balanc
        print("deposit is done your balance is {}".format(self.balanc))

    def withdraw(self,quantity):
        if quantity<self.balanc:
            self.balanc=self.balanc-quantity
            print("withdraw is done your balance is {}".format(self.balanc))
        else:
            print("cann't do this. your balance is {}".format(self.balanc))
    def __str__(self):
        return "account owner {} balance is {}".format(self.name,self.balanc)
b=account("taha",120)

print(b)
account owner taha balance is 120

b.deposit(50)
deposit is done your balance is 170
```

b.withdraw(10)
withdraw is done your balance is 160

b.withdraw(1000)
cann't do this. your balance is 160

print(b)
account owner taha balance is 160

الوحدات والحزم Modules, Packages

الوحدات هي طريقة لحفظ الدوال والعبارات في لغة بايثون واستدعاؤها حال لزومها باستخدام كلمة import

import astropy استدعاء الوحدة مباشرة

import astropy.table استدعاء وحدة من حزمة

data = astropy.table.Table.read('my_table.fits') استخدام الدالة المضمنة في الصنف المطلوب

from astropy.table import * يمكن استخدام النجمة لاستدعاء كل المحتوى ضمن الوحدة دفعة واحدة

data = Table('my_table.fits')

الحزم هي طريقة لحفظ وترتيب الوحدات للوصول اليها بشكل اسرع وخاصة في المكتبات ذات الوحدات المتعددة

لتحميل الحزم نستخدم pip install في comand prompt

هنا نستطيع تحميل أي حزمة للعمل عليها ضمن بيئة بايثون

مثال pip install colorama هذا السطر يقوم بتثبيت المكتبة

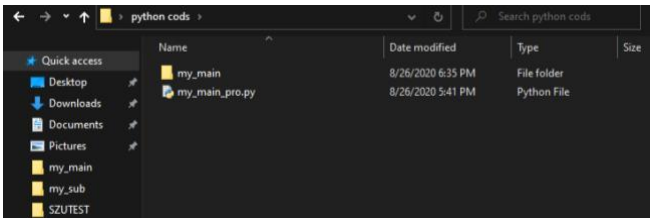
انشاء الحزم والوحدات والتعامل معها

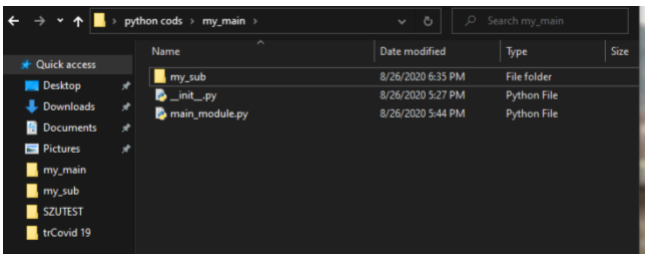
في البداية نفتح المجلد المراد العمل ضمنه ونقوم بانشاء ملف البرنامج الأساسي my_main_pro.py

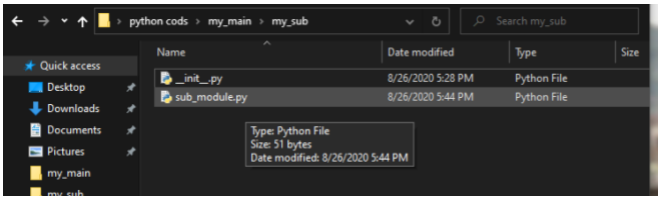
في نفس المجلد يمكن انشاء مجلد للحزم ولكن بشرط ان يحتوي على ملف فارغ باسم __init__.py لربطه معالبرنامج الأساسي

ضمن مجلد الحزمة هذا يمكن انشاء مجلد حزمة فرعية مع نفس الشرط بوجود ملف __init__.py

ضمن كل ملف من ملفات الحزم يمكن إضافة القدر المراد من الوحدات وضمن هذه الوحدات يتم تعريف الدوال







بعد انشاء الملفات والمجلدات بهذا الشكل يمكن تعريف الدوال ضمن الوحدات واستدعاؤها في البرنامج الأساسي كالتالي

```
from my_main import main_ module  استدعاء الوحدة من الحزمة
main_ module.main_fun() تشغيل الدالة الموجودة ضمن الحزمة
from my_main.my_sub import sub_ module استدعاء الوحدة الفرعية من الحزمة الفرعية
sub_ module.sub_fun() تشغيل الدالة الموجودة ضمن الوحدة الفرعية
```

في بقية لغات البرمجة هناك الدالة الأساسية main() التي يتم ضمنها تشغيل البرنامج الأساسي اما في بايثون فالامر مختلف عندما يتم تشغيل ملف برنامج ما يتم تعيين الخاصية __name__ لتصبح __main__ اما اذا كان الامر مجرد استدعاء دالة من الملف فلن تعمل هذه الخاصية لذلك يمكننا وضع جملة شرطية للتحقق من تشغيل الملف مباشرة او استيراد في ملف اخر

مثال:

```
---------- الملف الأول
def func():  الملف الأول يحتوي على دالة
   print("function in the first one")
print("the top of the first script") سيم تشغيل هذا السطر في حال تشغيل الملف
if __name__=="__main__":
   print("first one is working directly") اذا تم تشغيل الملف مباشرة فسيعمل هذا السطر
else:
   print("first on is imported") سيعمل هذا السطر ان كان الملف يعمل مستوردا ضمن ملف اخر
---------- الملف الثاني
import first في هذا السطر نقوم باستيراد الملف الأول
print("the top of second") سيعمل هذا السطر في بداية تشغيل الملف في كل الحالات
first.func()  تشغيل الدالة المعرفة في الملف الأول
if __name__=="__main__":
   print("second one is working directly") سيعمل هذا السطر في حالة التشغيل المباشر
else:
   print("second on is imported") سيعمل هذا السطر ان كان الملف يعمل مستوردا ضمن ملف اخر
```

عند تشغيل الملف الأول مباشرة ستكون النتيجة :

```
the top of the first script
first one is working directly
```

اما في حال تشغيل الملف الثاني :

```
the top of the first script
first on is imported
the top of second
function in the first one
second one is working directly
```

الاسطر المعلمة بالاصفر ستعمل بمجرد استدعاء الملف

ضمن مجلد الحزم قلنا اننا يجب ان ننشأ ملف باسم __init__ للدلالة ان هذا الملف هو حزمة في بايثون ويمكن ان يكون هذا الملف فارغ ولكن ماذا لوقام المستخدم باستيراد جميع ما في الحزمة عن طريق from example import * سيتم هنا استيراد جميع ما في الحزمة والحزم الفرعية وقد يتطلب ذلك وقتا ويكون امر غيرمرغوب لان الكثيرمنها لن تستخدم لذلك الحل هو كتابة السطر التالي ضمن الملف __init__ :

```
__all__ = ["firstOne", "secondOne", "therdOne"……]
```

في هذه الحال سيتم استدعاء الحزم او الوحدات المضمنة في العبارة فقط

استكشاف الأخطاء ومعالجتها Errors and Exception Handling:

نستخدم هذه الطريقة لطباعة جملة تختصر لنا نوع الخطأ او تنبهنا لوجود خطأ دون إيقاف الكود او الانهاء غير الطبيعي حيث يمكن متابعة الأقسام التالية دون أي مشاكل ونستخدم فيها الجمل المفتاحية try: except: else: على الشكل التالي

```
try:
```

```
   f = open('testfile','w') هنا نقوم بكتابة الكود المراد اختباره او الكود الأصلي
   f.write('Test write this')
except IOError: في هذا السطر يمكن كتابة نوع الخطأ المراد كتابة رسالة بشأنه او يمكن المتابعة بدون كتابة نوع الخطأ وحينها
                تعمل على كل الأخطاء على السواء او يمكن إضافة عبارة مستقلة لكل نوع من أنواع الأخطاء
   print("Error: Could not find file or read data") هذا السطر سيعمل في حال وجود خطأ فقط
else:
   print("Content written successfully") اذا لم يكن هناك خطأ سيعمل هذا الجزء من الكود
   f.close()
```

مثال اخر:

```
try:
   x=int(input("inter a number : "))
   y=x*2
   print(y)
except :
   print("you have to inter a number")
else:
   print("don!!")
```

```
inter a number : 5
10
don!!
```

```
inter a number : h
you have to inter a number
```

يمكننا استخدام عبارة :finally لتشغيل امر ما في كل الأحوال مثال:

```
try:
      val = int(input("Please enter an integer: "))
   except:
      print("Looks like you did not enter an integer!")

   finally:
      print("Finally, I executed!")
   print(val)
```

```
Please enter an integer: 5
Finally, I executed! تم تشغيلها بالرغم من عدم حدوث  أي خطأ
5 طباعة  القيمة بشكل طبيعي
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!                    Except تمت الطباعة بالرغم من وجود خطا أي ان الكود لم ينتهي عند عبارة
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)  حدث هذا  الخطأ لأن القيمة لم يتم تعريفها
<ipython-input-8-cc291aa76c10> in <module>()                                 بسبب حدوث الخطأ
----> 1 askint()

<ipython-input-6-c97dd1c75d24> in askint()
    7     finally:
    8         print("Finally, I executed!")
----> 9     print(val)
```

UnboundLocalError: local variable 'val' referenced before assignment

لتفادي حدوث مثل هذا الامر يمكن استخدام while

سيستمر البرنامج بالعمل دائما :while True

```
try:
   val = int(input("Please enter an integer: "))
except:
   print("Looks like you did not enter an integer!")
   continue المفترض ان يعود البرنامج من البداية عند الوصول الى هذا السطر ليسمح للمستخدم بإدخال رقم من جديد
else:
   print("Yep that's an integer!")
   break بعد ادخال رقم وعدم حدوث خطأ يجب ان ينتهي البرنامج هنا
finally:
   print("Finally, I executed!")
print(val)
```

Please enter an integer: five هنا تم ادخال احرف بدل الرقم

Looks like you did not enter an integer! بعد حدوث الخطأ تمت طباعة هذه العبارة وإعادة البرنامج الى البداية

Finally, I executed! المفترض ان تعمل عبارة الاتمام بشكل صحيح ولا يتم طباعة هذه الجملة ولكن عبارات التوقف والاتمام والمتابعة لن تؤثر في هذه العبارة ونلاحظ ان السطر الأخير لم تتم طباعته

Please enter an integer: 3

Yep that's an integer! هنا أيضا بسبب وجود عبارة التوقف لم يتم طباعة السطر الأخير أي انها تعمل على العبارات العادية ولا تعمل على ما ضمن عبارة finally

Finally, I executed!

اختبار الوحدات Unit Testing

أحيانا يجب اختبارالكود قبل التطبيق العملي حتى من ناحية قابلية الفهم لقارئه او من ناحية المتغيرات والاخطاء التي يمكن ان تحدث

هناك مكتبات خاصة تعمل على اختبار الكود وإعطاء ملخص وطريقة تحسين الكود مع درجة من 10 للتقييم منها مثلا :

- pylint
- pyflakes
- pep8

وهناك بعض المكتبات التي تعمل بشكل أوسع وتسمح بكتابة كود اختبار خاص مثل :

- unittest
- doctest

هذه المكتبات تحتاج لتثبيت كي تعمل:

```
! pip install pylint إشارة التعجب هنا لتشغيل الكود وكأنه يعمل من موجه الأوامر
```

هذا السطر يعمل من واجهة الكانسول وليس سطر برمجي يمكن انشاء شبيه له ذكرنا ذلك في %%writefile simple1.py التعامل مع الملفات

```
a = 1 سنقوم بانشاء كود بسيط الان ضمن الملف السابق
b = 2
print(a)
print(B)
```

سنقوم بتشغيل الكود الان عن طريق مكتبة pylint

```
! pylint simple1.py
```

********** Module simple1 ستكون النتيجة كالتالي

```
simple1.py:5:0: C0305: Trailing newlines (trailing-newlines)
simple1.py:1:0: C0114: Missing module docstring (missing-module-docstring)
simple1.py:1:0: C0103: Constant name "a" doesn't conform to UPPER_CASE naming style (invalid-
name)
simple1.py:2:0: C0103: Constant name "b" doesn't conform to UPPER_CASE naming style
(invalid-name)
simple1.py:4:6: E0602: Undefined variable 'B' (undefined-variable)
```

تعطي في البداية بعض المعلومات عن إضافة اسطر فارغة واستخدام قواعد الكتابة المناسبة والدوال والحزم وبعدها بيان الأخطاء مثلا هنا الخطا في ان المتغيرغير معرف

```
--------------------------------------------------------------------

Your code has been rated at -12.50/10 (previous run: -2.50/10, -10.00)
```

التقييم النهائي

يمكن تحسين الكود عبر كتابته بهذا الشكل :

```
%%writefile simple1.py
"""
A very simple script.
"""

def myfunc():
    """
    An extremely simple function.
    """
    first = 1
    second = 2
    print(first)
    print(second)

myfunc()
```

```
! pylint simple1.py
```

```
********** Module simple1
C: 14, 0: Final newline missing (missing-final-newline)

--------------------------------------------------------------------

Your code has been rated at 8.33/10 (previous run: -12.50/10, +20.83)
```

هذه المكتبة تعطي نتائج جيدة في حال كان الكود بيسطاً اما في حال كان اعقد من ذلك فالامر مختلف

```
%%writefile simple2.py
"""
A very simple script.
"""

def myfunc():
    """
    An extremely simple function.
    """
    first = 1
    second = 2
    print(first)
```

```
    print('second')

myfunc()
```

```
! pylint simple2.py
```

```
********** Module simple2
C: 14, 0: Final newline missing (missing-final-newline)
W: 10, 4: Unused variable 'second' (unused-variable)
```
في هذه الحالة يعطي تنبيه انه هناك متغيرات لم تستخدم ولا يتم التعرف الى اننا نريد طباعتها في السطر الأخير من الدالة

```
Your code has been rated at 6.67/10 (previous run: 6.67/10, +0.00)
```

في الحالات المعقدة نسبياً نقوم بكتابة كود اختبار باستخدام المكتبات مثل unittest

```
%%writefile cap.py  لننشأ ملف بسيط يحوي دالة للتحويل الى احرف كبيرة
def cap_text(text):
    return text.capitalize()
```

بعد ذلك نقوم بانشاء ملف اخر لاختبار هذا الكود :

```
%%writefile test_cap.py
import unittest  نقوم باستيراد المكتبة المراد العمل عليها
import cap       نقوم باستيراد الكود المراد اختباره

class TestCap(unittest.TestCase):  ننشأ صنف ونقوم بتحديد النمط المراد اختباره هنا (نتذكر بحث الأصناف والتوارث او التركة )

    def test_one_word(self):        نقوم بانشاء دالة لاختبار كلمة واحدة
        text = 'python'             ادخال كلمة للاختبار
        result = cap.cap_text(text) تطبيق الدالة على هذه الكلمة
        self.assertEqual(result, 'Python') == فحص فيما اذا كانت النتيجة مطابقة لما نتوقعه تماما مثل استخدامنا ل

    def test_multiple_words(self):
        text = 'monty python'
        result = cap.cap_text(text)
        self.assertEqual(result, 'Monty Python')

if __name__ == '__main__':  هذا السطر لتشغيل الملف فور استدعائه
    unittest.main()         هذه هي الدالة التي تقوم بتشغيل اختبار المكتبة
```

لعبة black jack

```
import random   استيراد هذه المكتبة لتوليد القيم العشوائية سنستخدمها لخلط الأوراق

suits = ('Hearts', 'Diamonds', 'Spades', 'Clubs') تعريف هذه المتغيرات لادخالها في ما بعد وهي تمثل قيم وأسماء الورق

ranks = ('Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King', 'Ace')

values = {'Two':2, 'Three':3, 'Four':4, 'Five':5, 'Six':6, 'Seven':7, 'Eight':8, 'Nine':9, 'Ten':10, 'Jack':10,
          'Queen':10, 'King':10, 'Ace':1}

playing = True هذا المتغير لبدأ اللعبة في الدالة الأخيرة يتم التحكم بقيمته ضمن بعض الدوال سيتم التطرق اليها

class Card:  تعريف صنف الورقة وهو يحوي نوع اسم الورقة ونوعها وقيمتها
```

```
    def __init__(self,suit,rank,value):
        self.suit=suit
        self.rank=rank
        self.value=value


    def __str__(self): عندما نقوم بطباعة الورقة نحتاج لمعرفة الاسم والنوع
        return f"{self.rank} of {self.suit}"

class Deck: نعرف هذا الصنف لانشاء مجموعة ورق كاملة


    def __init__(self): ما يحتويه هذا الصنف هو قائمة
        self.deck = []
        for suit in suits: يتم تعبئة هذه القائمة بالقيم التي قمنا بتعريفها في بداية البرنامج
            for rank in ranks: لكل نوع من الأنواع نقوم بإدخال جميع أسماء الورق وقيمتهم
                self.deck.append(Card(suit,rank,values[rank])) بما ان القيمة تم تعريفها كقاموس يمكن بسهولة الوصول اليها عن طريق اسم الورقة


    def __str__(self): هذه الدالة لطباعة مجموعة الورق كاملة
        d="" نقوم بانشاء نص فارغ
        for i in range(len(self.deck)) : بعدها نقوم بتعبأة هذا النص بكل ورقة على حدى
            d=d+(f"{self.deck[i].rank} of {self.deck[i].suit} \n") نقوم باضافة كل ورقة على سطر جديد
                هنا لا يمكننا طباعة كل ورقة على حدى لأننا نحتاج ان نقوم بحفظ ما تتم طباعته او كتابته ضمن الدالة


        return f"{d}"


    def shuffle(self): هذه الدالة تقوم بخلط او الغاء ترتيب المجموعة بشكل عشوائي
        random.shuffle(self.deck)


    def deal(self): هذه الدالة لسحب الورقة الأولى من المجموعة
```

```
        عندما نقوم بسحب ورقة فاننا نقوم بحذفها من المجوعة واضافتها الى يد المستخدم لذلك يجب ان card=self.deck.pop()
                                                                تعطي الدالة في النهاية قيمة ليتم استخدامها لاحقا
        return card
class Hand:                                                     تعريف صنف اليد لاستخدامه للاعبين
    def __init__(self):                                         يكون ضمن هذا الصنف الأوراق ومجموع قيمها ولا نحتاج الى تعريف هذه القيم في البداية لان ذلك يتم
                                                                ضمن اللعبة
        self.cards = []
        self.value = 0
        self.aces = 0                                           سنستخدم هذا المتغير لتحديد فيما اذا كان هناك ورقة من نوع قص في اليد
    def add_card(self,card):                                    دالة سحب ورقة
        self.cards.append(card)                                 ستتم إضافة الورقة التي تم ازالتها من مجموعة الورق الى قائمة الورق في يد المستخدم
        if card.rank == 'Ace':                                  اذا كانت هذه الورقة من نوع قص
            self.aces +=1                                       سيتم رفع عدد هذا النوع واحد
        self.value += card.value                                ستتغير قيمة مجموع اليد بإضافة قيمة الورقة المسحوبة
    def adjust_for_ace(self):                                   دالة تحديد قيمة القص اما 1 او 11
        if self.aces !=0 and self.value<=11:                    اذا كانت ورقة القص موجودة في اليد وقيمة مجموع اليد اقل من 11 سيتم رفع
                                                                قيمة القص من 1 الى 11
            self.value+=10                                      تكون قيمة القص 1 لذلك نضيف 10 لتصبح 11
            self.aces -=1                                       نقوم بانزال عدد القص 1 لكي لا يتم تكرار العملية مرة أخرى
class Chips:                                                    تعريف صنف النقود
    def __init__(self,total=100):                               يتم تعريف القيمة الكلية ويخصم منها القيمة الموضوعة للعب
        self.total = total
        self.bet = 0
    def win_bet(self):                                          اذا فاز اللاعب تضاف القيمة الموضوعة للعب  الى القيمة الكلية
        self.total += self.bet
    def lose_bet(self):                                         اذا خسر اللاعب ستخصم قيمة اللعب من القيمة الكلية
```

```
    self.total -= self.bet
def take_bet(chips): دالة تحديد قيمة اللعب


  while True:
    try: نستخدم استكشاف الأخطاء لنضمن ان اللاعب ادل قيمة عددية
      chips.bet = int(input('How many chips would you like to bet? '))
    except ValueError:
      print('Sorry, a bet must be an integer!') ستتم طباعة هذا السطر في حال ادخال المستخدم أي نوع غير الأرقام
    else:
      if chips.bet > chips.total: اذا لم يكن هناك خطأ في الادخال يبقى علينا التحقق من القيمة اذا كانت ضمن القيمة الكلية
                                  ام لا
        print("Sorry, your bet can't exceed",chips.total)
      else:
        break      اذا كانت القيمة المدخلة اصغر من القيمة الكلية يستم الخروج من الحلقة التكرارية
                   في كل الحالات السابقة سيتم تكرار العملية حتى ادخال قيمة مقبولة


def hit(deck,hand): دالة السحب


  hand.add_card(deck.deal()) سيتم استدعاء امر السحب من مجموعة الورق والذي يعطي قيمة الورقة المسحوبة ويرسل
                             هذه القيمة الى دالة إضافة الورقة الى يد المستخدم


def hit_or_stand(deck,hand): دالة لتخيير المستخدم في السحب مرة أخرى او التوقف
  global playing جعل هذا المتغير عالمي أي ان أي تغيير ضمن هذه الدالة سيؤثر على القيمة التي تم تعريفها في بداية
                 البرنامج


  while True:
    x = input("Would you like to Hit or Stand? Enter 'h' or 's' ") يتم سؤال المستخدم عن اختياره بحسب الاحرف


    if x[0].lower() == 'h': اذا اختار المستخدم السحب يستم استدعاء دالة السحب
      hit(deck,hand)
    elif x[0].lower() == 's': اذا اختار اللاعب التوقف عن السحب يتم استدعاء دالة تحديد قيمة القص وطباعة جملة ان
                              اللاعب الاخر سوف يلعب
      hand.adjust_for_ace()
```

```
        print("Player stands. Dealer is playing.")
        playing = False سيتم تحويل قيمة هذا المتغير لكي يتم ارجاعها اذا أراد اللاعب اللعب مرة أخرى
    else:
        print("Sorry, please try again.") اذا تم ادخال أي حرف اخر غير الحرفين المطلوبين يتم طباعة هذه العبارة وإعادة الحلقة التكرارية
        continue
    break لمنع تكرار الحلقة الا في حال عد ادخال حرف صحيح يمكن إعادة استدعاء الدالة بدل من ذلك
def show_some(player,dealer): دالة لاظهار بعض أوراق الخصم تحتاج الى مدخلين هما اللاعب والخصم سيتم تعريفهم في وقت لاحق
    print(f"\nDealer's Hand:\n card hidden \n {dealer.cards[1]} ") طباعة ورقة مخفية وورقة ظاهرة من أوراق الخصم
    print(f"\nPlayer's Hand:", *player.cards, sep='\n') طباعة جميع أوراق اللاعب استخدام إشارة النجمة لانه يمكن ان يتغير عدد أوراق اللاعب في كل مرة فالنجمة تعني الجميع والفاصلة بين طل قيمة من القيم هي سطر جديد

def show_all(player,dealer): دالة اظهار جميع الأوراق
    print("\nDealer's Hand:\n ", *dealer.cards, sep='\n')
    print("Dealer's Hand =",dealer.value)
    print(f"\nPlayer's Hand:\n", *player.cards, sep='\n')
    print("Player's Hand = ",player.value)
def player_busts(player,dealer,chips): ناتي الان لتعريف دوال للفوز والخسارة والتعادل
    show_all(player,dealer) سيتم اظهار جميع الأوراق
    print("Player busts!") هذه الدالة ستعمل في حال اصبح مجموع يد اللاعب اعلى من 21 او في حال كانت قيمة يد الخصم اعلى
    chips.lose_bet() سيخسر اللاعب ويتم خصم القيمة من القيمة الكلية

def player_wins(player,dealer,chips): تعريف دالة الفوز
    print("Player wins!")
    chips.win_bet()

def dealer_busts(player,dealer,chips): دالة خسارة الخصم
    show_all(player,dealer) يتم استدعاء هذه الدالة مرة واحد لان خسارة شخص تعني فوز الاخر
    print("Dealer busts!")
    chips.win_bet()
```

```
def dealer_wins(player,dealer,chips): دالة فوز الخصم
    print("Dealer wins!")
    chips.lose_bet()

def push(player,dealer): دالة التعادل لا يتم فيها خصم قيم
    print("Dealer and Player tie! It's a push.")

while True:          ناتي  لكتابة كود اللعبة
    print('Welcome to BlackJack! Get as close to 21 as you can without going over!\n\
    Dealer hits until she reaches 17. Aces count as 1 or 11.') طباعة جملة الترحيب مع معلومات بسيطة عن اللعبة
    deck = Deck() انشاء مجموعة ورق
    deck.shuffle() خلط مجموعة الورق

    player_hand = Hand()  انشاء يد اللاعب
    player_hand.add_card(deck.deal())سحب ورقتين للاعب
    player_hand.add_card(deck.deal())

    dealer_hand = Hand() انشاء يد الخصم
    dealer_hand.add_card(deck.deal()) سحب ورقتين للخصم
    dealer_hand.add_card(deck.deal())

    player_chips = Chips() انشاء محفظة للاعب
    take_bet(player_chips)  استدعاء دالة السؤال عن قيمة اللعبة
    show_some(player_hand,dealer_hand) دالة اظهار بعض الأوراق

    while playing : المتغير  هنا تكون قيمته 1 في البداية
        hit_or_stand(deck,player_hand) سؤال اللاعب فيما اذا كان يريد السحب او التوقف
        show_some(player_hand,dealer_hand) اظهار يد اللاعب بعد السحب
        if player_hand.value > 21: اذا كانت قيمة يد اللاعب اكبر من 21  سيخسر اللاعب ويتم استدعاء دالة خسارة اللاعب
                                   والخروج من الحلقة التكرارية
```

```
        player_busts(player_hand,dealer_hand,player_chips)
        break

if player_hand.value <= 21:اما اذا كانت قيمة يد اللاعب اقل او اصغر من 21 سيكون الدور للخصم

    while dealer_hand.value < 17: اذا كانت قيمة يد الخصم اقل من 17 فيستمر بالسحب
        hit(deck,dealer_hand)

    show_all(player_hand,dealer_hand) اظهار حميع الأوراق للاعب والخصم


    if dealer_hand.value > 21: اذا كانت قيمة يد الخصم بعد السحب اكثرتماما من 21 فسوف يخسر
        dealer_busts(player_hand,dealer_hand,player_chips)


    elif dealer_hand.value > player_hand.value: اما اذا كانت اقل او يساوي 21 و اكبر من قيمة يد اللاعب فسوف يفوز
        dealer_wins(player_hand,dealer_hand,player_chips)

    elif dealer_hand.value < player_hand.value: اما اذا كانت قيمة يد اللاعب اكثر فسوف يفوز اللاعب
        player_wins(player_hand,dealer_hand,player_chips)

    else:
        push(player_hand,dealer_hand)       اذا كانت كل الاحتمالات غير متحققة فذلك يعني التعادل

print("\nPlayer's winnings stand at",player_chips.total) طباعة المبلغ المتبقي في المحفظة او القيمة الكلية

new_game = input("Would you like to play another hand? Enter 'y' or 'n' ") سؤال المستخدم فيما اذا كان يريد ان يكمل اللعب او لا

if new_game[0].lower()=='y': اذا كان يريد اكمال اللعب
    playing=True سيتم تغيير هذا المتغير الى القيمة صحيح لانها تبدلت عندما توقف اللاعب عن السحب في اخر مرة
```

بهذا سوف تعود الحلقة التكرارية للعمل continue

else:

print("Thanks for playing!")

سيقوم البرنامج بالخروج ممن الحلقة التكرارية break

دالة reduce() :

تستخدم هذه الدالة لتطبيق عملية ما على جميع عناصر قائمة ما على الشكل التالي :

يتم في البداية اخذ اول عنصرين من القائمة وتطبيق العملية عليهم والنتيجة سيتم تطبيق العملية عليها مع القيمة الثالثة حتى الانتهاء من القائمة

هذا الامر لاستدعاء الحزمة الخاصة بالدالة from functools import reduce

lst =[47,11,42,13]

سيتم هنا جمع الأرقام في القائمة السابقة حيث يتم جمع اول رقمين ثم بعدها النتيجة مع الرقم الثالث وهكذا reduce(lambda x,y: x+y,lst)

يمكن تطبيق هذه الدالة لايجاد القيمة الأكبر ضمن قائمة ما

reduce(lambda a,b: a if (a > b) else b ,lst)

دالتي all() and any() :

هذه الدوال تعطي قيم منطقية بحسب المدخلات حيث تعطي الدالة الاولة قيمة صحيح في حال كانت كل المدخلات صحيحة فقط اما الثانية فتعطي صحيح اذا كانت واحدة من المدخلات على الأقل صحيحة

```
lst = [True,True,False,True]
all(lst)
False
any(lst)
True
```

دالة complex() :

تستخدم هذه الدالة لتوليد الأرقام العقدية التي تتكون من قسمين قسم حقيقي وقسم تخيلي وتكتب بالشكل ai+bj او لتحويل الكتابة الى عدد عقدي على الشكل التالي :

```
complex(2,3)
(2+3j)
complex('12+2j')
(12+2j)
```

Decorators:

هي طريقة تستخدم لترتيب وتنظيم الدوال او لانشاء دوال مضمنة ضمن دوال أخرى وذلك لكون لغة البايثون قادرة على التعامل مع الدوال على انها كائن او مدخلات لدوال أخرى.

في البداية يجب تذكر مفهوم العالمية والمحلية لان الدوال أيضا تعامل معاملة المتغيرات من حيث هذا المنطق فهناك دوال محلية ودوال عالمية

s = 'Global Variable' تعريف متغير عالمي

def check_for_locals(): انشاء دالة لطباعة المتغيرات المحلية

print(locals()) سيكون فارغاً بداية بسبب عدم وجود أي متغير محلي

print(globals()) سيقوم بطباعة جميع المتغيرات العاليمية بما فيها المتغيرات الموجودة أصلا ضمن بايثون على شكل قاموس

print(globals().keys()) هنا يمكننا طباعة أسماء المتغيرات

globals()['s'] هنا يمكننا الوصول الى المتغير الذي قمنا بتعريفه سابقا

سنقوم الان بانشاء دالة بسيطة لفهم كيفية التعامل مع الدوال والتنسيقات :

def hello(name='Jose'):

```
    return 'Hello '+name
```
نقوم الان بوضع الدالة ضمن متغير وهنا سيكون هناك نسخة جديدة ومطابقة ومنفصلة عن الدالة الأولى تماما مثل greet = hello المتغير العادي

يمكن التعامل مع الدالة الجديدة بلا أي فرق يذكر greet()

عند حذف الدالة الاصلية لا تتأثر الدالة الجديدة del hello

الان نقوم بتعريف الدوال بداخل دوال أي دوال محلية تماما كالمتغير المحلي :

```
def hello(name='Jose'):
    print('The hello() function has been executed')

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    print(greet())
    print(welcome())
    print("Now we are back inside the hello() function")
```

الان في حال استدعاء الدالة سيكون الناتج كاالتالي:

```
The hello() function has been executed
        This is inside the greet() function
        This is inside the welcome() function
Now we are back inside the hello() function
```

أي يمكننا استدعاء الدوال المعرفة ضمن الدالة داخل الدالة نفسها

لا يمكننا الوصول الى الدوال المعرفة داخل دالة بالاستدعاء الطبيعي للدالة وستعتبر غير موجودة.

عوضا عن ذلك يمكن كتابة وتنسيق الدالة الأساسية لتعطي الدالة المرادة في الوقت المناسب كالتالي مثلا:

```
def hello(name='Jose'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Jose':
        return greet
    else:
        return welcome
```

في هذه الحالة يمكننا استدعاء الدالة المضمنة ببساطة بطريقتين اما حفظ الدالة في متغير او استخدام قوسين تماما كالمتغيرات

```
x = hello()
<function __main__.hello.<locals>.greet>
```

في هذه الحالة أيضا ستكون النتيجة متشابهة hello()()

الان سنقوم بانشاء المنسق Decorator:

في هذه الدالة يتم تشغيل كود معين وبعدها يتم استدعاء الدالة المطلوبة وبعدها تشغيل كود اخر :def new_decorator(func)

```
    def wrap_func():
        print("Code would be here, before executing the func")
```

```
        func()

        print("Code here will execute after the func()")

    return wrap_func

def func_needs_decorator(): هذا مثال على دالة سيتم استعاؤها ضمن المنسق
    print("This function is in need of a Decorator")
func_needs_decorator = new_decorator(func_needs_decorator) هكذا سيتم تعديل الدالة وتنسيقها


@new_decorator يمكننا أيضا تطبيق نفس العملية السابقة مباشرة هكذا
def func_needs_decorator():
    print("This function is in need of a Decorator")
```

ستكون النتيجة عند استدعاء الكود في الحالتين السابقتين كالتالي :

```
func_needs_decorator()
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

التكرار والانشاء Iterators and Generators:

الهدف الأساسي من هذه الدوال هو انشاء متغيرات ودوال حسب الحاجة عوضا عن حفظ كل شيئ في الذاكرة

بالنسبة لدوال الانشاء ستسمح لنا بتطبيق العمليات بالقدر المطلوب الى الوصول الى الكلمة المفتاحية yield

القضية الأهم هنا هو اننا عوضا عن ادخال القيم في كل مرة المنشئ يحتفظ بالقيمة الأخيرة عند اخر تشغيل له ويستعملها عند استدعائه لانه لا يعمل كالدالة العادية ينفذ الامر ويخرج بل يتحول الى كائن يحمل قيمة معينة .

سنقوم بانشاء دالة لحساب مكعب عدد ما :

```
def gencubes(n):
    for num in range(n):
        yield num**3 في هذه الدالة سيتم تكرار العملية من الصفر الى العدد المدخل في الدالة
```

لطباعة النتيجة نكتب الدالة على الشكل التالي :

```
for x in gencubes(10):
    print(x)
```

سيتم الان طباعة مكعبات الاعداد من 0 الى 9

هناط دالتين يمكن التعامل معهما الانشاء هم التالي والتكرار next() and iter()

```
def simple_gen(): ننشئ دالة انشاء بسيطة
    for x in range(3):
        yield x
g = simple_gen() نحفظ الدالة في متغير
print(next(g)) الان باستخدام التالي يمكننا الوصول الى كل قيمة في الدالة واحدة تلي الأخرى
```

ولكن عند انتهاء العدد المعرف ضمن الدالة الأولى سيظهر لنا خطأ يقول بان التكرار قد تما إيقافه بسبب كلمة yield

الامر مشابه لما يحصل عند محاولة طباعة النصوص

```
s = 'hello'
for let in s:
    print(let)
```

هذا لا يعني ان النصوص هي بالفعل منشئات انما تدعم هذه الخاصية اذا استخدمنا الدالة iter()

```
s_iter = iter(s) بعد ذلك يمكننا تطبيق دالة التالي عليها
```

next(s_iter)
'h'

مجموعة الحزم Collections Module:

هي حزم مضمنة في بايثون توفر بعض السهولة للتعامل مع أنواع البيانات المختلفة

- العداد Counter:

هو صنف قاموس فرعي يوفر تعداد للبيانات على شكل مفاتيح وقيم

يتم استيراده بالطريقة التالية : from collections import Counter

lst = [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

ستوفر هذه الدالة العنصر مع عدد تكراراه في السلسلة على شكل قاموس Counter(lst)

الامر نفسه ينطبق على النصوص Counter({1: 6, 2: 6, 3: 4, 12: 1, 21: 1, 32: 1, 223: 1})

s = 'How many times does each word show up in this sentence word times each each word'

يمكن استخدامه أيضا لعد الكلمات بعد فصل النص words = s.split()

Counter(words)

c = Counter(words)

يمكن استخدام هذه الدالة لمعرفة الأكثر تكراراً c.most_common(2)

لمعرفة مجموع العناصر sum(c.values())

لحذف محتوى العداد c.clear()

لانشاء قائمة بالمحتوى دون تكرار list(c)

التحويل الى حزم set(c)

التحويل الى قاموس عادي dict(c)

التحويل الى قائمة تحتوي على العنصر وعدد التكرارات في حزم غير قابلة للتعديل c.items()

ارجاع القائمة السابقة الى عداد Counter(dict(list_of_pairs))

الأقل استخداما بحسب المتغير ان c.most_common()[:-n-1:-1]

لحذف القيم السالبة والصفرية c += Counter()

- القاموس التلقائي defaultdict:

مماثل تقريبا للقاموس العادي الا انه لا يعطي أي خطأ لقيمة غير موجودة

يتم استدعاؤه كالتالي from collections import defaultdict

عند تعريف قاموس فارغ واستدعاء قيمة غير موجودة d = {}

الوضع الطبيعي ان يعطي تنبيه بعدم وجود هذه القيمة ضمن القاموس d['one']

لتفادي هذا الخطأ نحول القاموس العادي الى قاموس تلقائي d = defaultdict(object)

عندها سيتم انشاء القيمة ككائن في القاموس d['one']

يمكن أيضا انشاء قيمة تلقائية للقيم الجديدة d = defaultdict(lambda: 0)

- القاموس المرتب OrderedDict:

هو مشابه للقاموس العادي الا انه حساس للترتيب أي اننا عندما نضع نفس المحتوى في قاموسين مختلفين بترتيب مختلف سناخذ نتيجة صحيح لو حاولنا ان نقارن بينهم بينما القاموس المرتب سيعطي النتيجة صحيح فقط في كال كان الترتيب أيضا نفسه :

```
d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'
d2 = {}
d2['b'] = 'B'
d2['a'] = 'A'
print(d1==d2)
```

قاموسين عاديين بترتيب مختلف مع نفس المحتوى

Dictionaries are equal?

بينما لو جربنا نفس الامر على القاموس المرتب True

يتم استدعاء القاموس بهذا الشكل from collections import OrderedDict

```
print('Dictionaries are equal?')
d1 = OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d2 = OrderedDict()
d2['b'] = 'B'
d2['a'] = 'A'
print(d1==d2)

Dictionaries are equal?
النتيجة خطأ لعدم توافق الترتيب أيضا False
```

- تسمية الحزم namedtuple

الهدف الأساسي هو الوصول للعناصر بتسميتها بدل الأرقام التي قد نخطئ بتحديدها

```
t = (12,13,14)
هذا هو الوضع الطبيعي لاستدعاء القيم t[0]
12

نقوم باستدعاء الدالة الان from collections import namedtuple
ننشئ تسمية الحزم الان وهذا يعني اننا ننشئ صنف من نوع ما Dog = namedtuple('Dog','age breed name')
                                                    يحتوي الخصائص التالية
sam = Dog(age=2,breed='Lab',name='Sammy')
frank = Dog(age=2,breed='Shepard',name="Frankie")
عند استدعاء الكائن سيتم طباعة الخصائص مباشرة ويمكن الوصول لخاصية تماما كالاصناف sam
Dog(age=2, breed='Lab', name='Sammy')
sam.age
ولكن ما يميز هذه الدالة عن الأصناف هو بقاء قابلية الاستدعاء بالأرقام او الأماكن ضمن الحزم 2
sam[0]
2
```

التاريخ والوقت datetime

```
import datetime استيراد المكتبة

t = datetime.time(4, 20, 1) انشاء متغير وقت

print(t)
print('hour  :', t.hour)
print('minute:', t.minute)
print('second:', t.second)
print('microsecond:', t.microsecond)
print('tzinfo:', t.tzinfo)
04:20:01
hour  : 4
minute: 20
second: 1
microsecond: 0
tzinfo: None
today = datetime.date.today() يمكن أيضا جلب معلومات اليوم الحالي ووضعها في متغير
print(today)
print('ctime:', today.ctime())
```

```
print('tuple:', today.timetuple())
print('ordinal:', today.toordinal())
print('Year :', today.year)
print('Month:', today.month)
print('Day  :', today.day)
```

```
2020-09-04
ctime: Sun Sep  4 00:00:00 2020
tuple: time.struct_time(tm_year=2020, tm_mon=9, tm_mday=4, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=6, tm_yday=250, tm_isdst=-1)
ordinal: 737672
Year : 2020
Month: 9
Day  : 4
```

```
d1 = datetime.date(2015, 3, 11) يمكن أيضا تعريف تواريخ
print('d1:', d1)

d2 = d1.replace(year=1990) لاستبدال قيمة او قيم
print('d2:', d2)
```

```
d1: 2015-03-11
d2: 1990-03-11
```

يمكن أيضا اجراء عمليات على التواريخ d1-d2
datetime.timedelta(days= 9131)

المصحح Debugger:
هو دالة مضمنة في بايثون تمكننا من مراقبة ما يحدث في الكود في كل خطوة وتحليلها لمعرفة الأخطاء

```
import pdb استيراد المصحح

x = [1,3,4]
y = 2
z = 3

result = y + z
print(result)
pdb.set_trace() يمكننا انشاء المصحح في أي سطر نريد وبعدها نقوم بالتنقل سطراً سطراً
نستخدم next بعد تشغيل البرنامج للانتقال الى الخط التالي حتى انهاء البرنامج
result2 = y+x
print(result2)

> c:\users\tahae\trcovid 19\untitled1.py(12)<module>()
     8 result = y + z
     9 print(result)
    10
    11 result2 = x*y
---> 12 print(result2)


next
[1, 3, 4, 1, 3, 4]
--Return--
None
```

توقيت الكود Timing your code:
يمكننا من معرفة الوقت اللازم لتنفيذ السطر البرمجي

```
import timeit
timeit.timeit('"-".join(str(n) for n in range(100))', number=10000) ندخل السطر البرمجي مع ادخال عدد المرات
0.21865416520477374 التي سيتم اجراء الامر بها

%timeit "-".join(str(n) for n in range(100))  يمكن تنفيذ الامر بهذا الشكل أيضا

20.4 µs ± 269 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

التعبيرات العادية Regular Expressions:
هي طريقة للبحث ومطابقة النصوص باستخدام مكتبة re

```
import re  استيراد المكتبة
patterns = ['term1', 'term2'] انشاء قائمة للبحث
text = 'This is a string with term1, but it does not have the other term.' النص الذي سيتم البحث ضمنه
for pattern in patterns:
   print('Searching for "%s" in:\n "%s"\n' %(pattern,text))
      if re.search(pattern,text): البحث عن مطابقة النص
```

```
    print('Match was found. \n')
  else:
    print('No Match was found.\n')
```
الان لو قمنا بانشاء متغير وحفظ نتيجة البحث فيه
```
match = re.search(pattern,text)
```
هذا المتغير يعطينا نتيجة صحيح او خطا بحسب نتيجة البحث وأيضا يعطينا معلومات عن النتيجة مثلا مكان البداية والنهاية
```
match.end()
match.start()
```
يمكننا أيضا استخدام خيارات الفصل للنصوص
```
split_term = '@' المكان الذي نريد فصل النص عنده

phrase = 'What is the domain name of someone with the email: hello@gmail.com' النص المطلوب
re.split(split_term,phrase)  امر الفصل
['What is the domain name of someone with the email: hello', 'gmail.com'] النتيجة مطابقة لامر الفصل في النص العادي
re.findall('match','test phrase match is in middle') هذا الامر يمكننا من إيجاد جميع التطابقات على حدى على شكل قائمة
```
توفر هذه المكتبة أنماط عديدة للبحث والكتابة

'sd*',	للبحث عن اس بدون دي او مع دي او اكثر
'sd+',	للبحث عن اس متبوعة بدي واحدة او اكثر
'sd?',	للبحث عن اس متبوعة بدي واحدة او غير متبوعة ب دي
'sd{3}',	للبحث عن اس متبوعة بثلاث دي
'sd{2,3}',	للبحث عن اس متبوعة ب دي من المجال 2 الى 3
'[sd]',	للبحث عن دي او اس
's[sd]+'	للبحث عن اس متبوعة بدي او اس او اكثر
's[^sd]'	للبحث عن اس غير متبوعة باس او دي
'[a-z]+',	للبحث عن سلسلة احرف صغيرة
'[A-Z]+',	للبحث عن سلسلة احرف كبيرة
'[a-zA-Z]+',	للبحث عن سلسلة احرف دون اعتبار الكبير والصغير
'[A-Z][a-z]+'	للبحث عن حرف كبير متبوع باحرف صغيرة
r'\d+',	سلسلة ارقام
r'\D+',	سلسلة من غير الأرقام
r'\s+',	سلسلة من الفراغات
r'\S+',	سلسلة من غير الفراغات
r'\w+',	سلسلة احرف واراقام
r'\W+'	سلسلة من غير الاحرف والأرقام

الكائنات النصية StringIO:

تمكننا هذه المكتبة من حفظ الملفات النصية على شكل كائن في ذاكرة البايثون لاستعماله في الكود على عكس أوامر الفتح التقليدية التي تحتفظ بالملف في الذاكرة الاصلية

```
import io  استيراد المكتبة

message = 'This is just a normal string.'  انشاء نص بسيط

f = io.StringIO(message)  تحويل النص باستخدام المكتبة

f.read() يمكن الان تطبيق عدة عمليات على النص

f.write(' Second line written to file like object')

40 نتيجة السطر السابق هي مكان انتهاء الكتابة أي موقع اخر حرف تم إدخاله
```

لذا نستخدم هذه الدالة لتصفير المؤشر f.seek(0)

اغلاق الملف f.close()

واجهات المستخدم GUI

- التفاعل Interact

في البداية نقوم باستيراد بعض المكتبات اللازمة

```
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
```

الان نقوم بتعريف دالة بسيطة لاستدعائها ضمن التفاعل

```
def f(x):
   return x
interact(f, x=10,); 30 الى 10- في هذه الدالة سنقوم بانشاء شريط لتحديد قيمة المتغير في الدالة السابقة يبدأ من
                                                    وتكون القيمة الافتراضية 10
interact(f, x=True); اما في هذه الحالة عند كتابة صحيح او خطأ او سيقوم بانشاء مربع اختيار وتغيير قيمة المتغير
                                                    الى صحيح وخطأ بحسب الإشارة
interact(f, x='Hi there!'); اما اذا تم ادخال نص فسيتم انشاء مربع نص
@interact(x=True, y=1.0) يمكن استخدام دالة التفاعل أيضا كمنظم لدوال تحتوي عدة متغيرات
def g(x, y):
   return (x, y) في هذه الحالة سيظهر شريط تمرير مع مربع اختيار
 def h(p, q):
   return (p, q)
interact(h, p=5, q=fixed(20)); في هذه الحالة يمكننا تثبيت احد القيم وإبقاء الأخرى متغيرة
IntSlider(min=-10,max=30,step=1,value=10) يمكن التحكم بخيارات الشريط أيضا
interact(f, x=widgets.IntSlider(min=-10,max=30,step=1,value=10)); وهذه هي القيم الافتراضية ويجب
                                                    ان تكون اعداد صحيحة
interact(f, x=(10,50,0.1)); يمكن كتابة الدالة بهذا الشكل للحصول على قيم عشرية
interact(f, x={'one':1,'two':2}); يمكننا أيضا الحصول على قوائم اختيار مع القيم المقابلة لها للطباعة
interact(f, x=['one','two']); او طباعة القيمة نفسها من الصندوق هكذا
interact(f, x=(0,4,2));يمكن استخدام الشريط أيضا بهذا الشكل
interact(f, x=(0.0,10.0, 0.01));

@interact(x=(0.0,20.0,0.5))
def h(x=5.5):
   return x
def f(x:True): يمكن أيضا تعريف الدالة واستخدامها مباشرة في التفاعل
   return x
interact(f); الان سيظهر لدينا مربع اختيار
```

أحيانا لا نحتاج الى اظهار النتيجة مباشرة بل يتم اظهاراها ضمن استدعاء بدالة أخرى او في مكان اخر لذلك نيتخدم خاصية الاظهار على الشكل التالي

```
from IPython.display import display  استدعاء الخاصية

def f(a, b): تعريف متغير بسيط لاستخدامه
   display(a + b) دالة الاظهار لاستدعائها في الوقت المطلوب
   return a+b النتيجة التي يتم حفظها كقيمة للمتغير
w = interactive(f, a=10, b=20) الان نقوم بحفظ الشريط كمتغير نوعه واجهة مستخدم من النوع شريط
w.children يمكن الوصول الى المحتويات هكذا
(IntSlider(value=10, description='a', max=30, min=-10),
```

```
 IntSlider(value=20, description='b', max=60, min=-20),
 Output())
display(w) ويتم عرضه باستخدام دالة العرض
w.kwargs للوصول الى المدخلات الحالية وهي قيم متغيرة في كل مرة
w.result لاظهار الناتج النهائي
```

- اساسيات القطع Widget Basics

تحتوي عناصر التفاعل مع المستخدم خاصيات مضمنة بداخلها `import ipywidgets as widgets`

مثلا يتم اظهار هذا الشريط مباشرة عند استدعائه بقيم تلقائية دوم الحاجة الى امر العرض `widgets.IntSlider()`

ويمكن أيضا استخدام الامر اظهار لعرضها بعد حفظها في متغير `from`

```
IPython.display import display
w = widgets.IntSlider()
display(w)
```

عندما يتم عرض الشريط مرة أخرى فان أي تغيير يتم عليه سيؤثر على الشريط السابق لذلك نقوم باغلاق الشريط عند انتهاء العمل عليه عبر الامر اغلاق

```
w.close()
w.value = 100 يمكن تعيين قيمة الشريط يدويا او عرضها مباشرة
w.value
w.keys هذا الامر يقوم بعرض الخصائص التي يحتوي عليها المتغير لمعرفة أيها سيتم التعامل معه او استدعاؤه
widgets.Text(value='Hello World!', disabled=True) يمكننا مثلا تعيين قيمة وجعل مربع النص غير قابل للتعديل او معطل
```

يمكن أيضا ربط الخصائص المتشابهة في القطع التفاعلية المختلفة كربط القيمة في مربع النص والقيمة في الشريط

```
a = widgets.FloatText() مربع نص يتم ادخال قيمة عددية كسرية داخله
b = widgets.FloatSlider()
display(a,b)
```

نستخدم هذه الدالة لربط القيمة في كل من القطعتين السابقتين

```
mylink=widgets.jslink((a,'value'),(b,'value'))
mylink.unlink() يمكننا أيضا فك الارتباط بهذا الامر
```

- قوائم القطع التفاعلية

لعرض جميع النوافذ التفاعلية المتاحة في المكتبة يمكننا استخدام هذا الامر لطباعتها

```
for item in widgets.Widget.widget_types.items():
  print(item[0][2][:-5])
```

النوافذ التي تتعامل مع الأرقام هي كالتالي

```
widgets.IntSlider( شريط الاعداد الصحيحة
  value=7,
  min=0,
  max=10,
  step=1,
  description='Test:',
  disabled=False,  وهذه هي الخصائص التي يمكن استخدامها ضمنه
  continuous_update=False,
  orientation='horizontal',
  readout=True,
  readout_format='d'
)
widgets.FloatSlider( شريط الاعداد الكسرية
  value=7.5,
  min=0,
```

```
    max=10.0,
    step=0.1,
    description='Test:',
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True,
    readout_format='.1f',
)
widgets.IntRangeSlider( شريط الأرقام ضمن مجال
    value=[5, 7], يتم اختيار مجال كقيمة خرج
    min=0,
    max=10,
    step=1,
    description='Test:',
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True,
    readout_format='d',
)
widgets.FloatRangeSlider( شريط قيمة مجال كسرية مشابه للسابق
    value=[5, 7.5],
    min=0,
    max=10.0,
    step=0.1,
    description='Test:',
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True,
    readout_format='.1f',
)
widgets.IntProgress( شريط التقدم او التحميل
    value=7,
    min=0,
    max=10,
    step=1,
    description='Loading:',
    bar_style='', # 'success', 'info', 'warning', 'danger' or '' يمكن تحديد شكل ولون  الشريط ونمطه
    orientation='horizontal'
)
widgets.FloatProgress( شريط مشابه للسابق مع الاعداد الكسرية
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description='Loading:',
```

```
    bar_style='info',
    orientation='horizontal'
)
widgets.BoundedIntText(مربع نص للاعداد الصحيحة
    value=7,
    min=0, يمكن تغيير القيمة اما بالكتابة او عن طريق النقر
    max=10,
    step=1,
    description='Text:',
    disabled=False
)
widgets.BoundedFloatText( مشابه للسابق مع الاعداد الكسرية
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description='Text:',
    disabled=False
)
widgets.IntText( مشابه للسابق دون وجود تحديد للقيمة الدنيا والعليا
    value=7,
    description='Any:',
    disabled=False
)
widgets.FloatText( مثل السابق للاعداد الكسرية
    value=7.5,
    description='Any:',
    disabled=False
)
```

النوافذ التي تتعامل مع القيم المنطقية

```
widgets.ToggleButton( مربع زر
    value=False,
    description='Click me',
    disabled=False,
    button_style='', # 'success', 'info', 'warning', 'danger' or ''
    tooltip='Description',
    icon='check'
)
widgets.Checkbox( مربع اختيار
    value=False,
    description='Check me',
    disabled=False
)
widgets.Valid( نافذة اشعارات خطأ وصحيح
    value=False,
    description='Valid!',
)
```

النوافذ التي تتعامل مع الاختيارات يمكن استخدام القوائم فيها او القواميس حسب الغاية المطلوبة

```
widgets.Dropdown( مربع اختيارات
   options=['1', '2', '3'], يمكن استخدام قاموس كما ذكرنا سابقا
   value='2', القيمة التلقائية
   description='Number:',
   disabled=False,
)
widgets.RadioButtons( قائمة اختيارات لا تقبل التعدد
   options=['pepperoni', 'pineapple', 'anchovies'],
   # value='pineapple', القيمة يمكن ان تكون محددة تلقائيا او لا
   description='Pizza topping:',
   disabled=False
)
widgets.Select( قائمة اختيار مباشرة
   options=['Linux', 'Windows', 'OSX'],
   value='OSX',
   # rows=10, لتحديد عدد الاسطر الظاهرة
   description='OS:',
   disabled=False
)
widgets.SelectionSlider( شريط اختيارات
   options=['scrambled', 'sunny side up', 'poached', 'over easy'],
   value='sunny side up',
   description='I like my eggs ...',
   disabled=False,
   continuous_update=False,
   orientation='horizontal',
   readout=True
)
import datetime استيراد مكتبة الوقت
dates = [datetime.date(2015,i,1) for i in range(1,13)] انشاء قائمة لاخذ الأشهر منها
options = [(i.strftime('%b'), i) for i in dates] من القائمة السابقة نضع التاريخ وناخذ منه الاسم المختصر للشهر
widgets.SelectionRangeSlider( شريط اختيار مجال نصي
   options=options, نضع الخيارات هنا من القائمة
   index=(0,11), طول شريط الاختيار
   description='Months (2015)', تعريف
   disabled=False
)
widgets.ToggleButtons( مربعات اختيارات
   options=['Slow', 'Regular', 'Fast'],
   description='Speed:',
   disabled=False,
   button_style='', # 'success', 'info', 'warning', 'danger' or ''
   tooltips=['Description of slow', 'Description of regular', 'Description of fast'],
   # icons=['check'] * 3
)
widgets.SelectMultiple(نافذة اختيار متعدد
   options=['Apples', 'Oranges', 'Pears'],
   value=['Oranges'],
```

```
    # rows=10,
    description='Fruits',
    disabled=False
)
```

النوافذ التي تتعامل مع النصوص

```
widgets.Text(نافذة مربع  نص
    value='Hello World',          القيمة الموجودة
    placeholder='Type something', ما يكتب عند كون المربع فارغ
    description='String:',
    disabled=False
)
widgets.Textarea( مساحة كتابة
    value='Hello World',
    placeholder='Type something',
    description='String:',
    disabled=False
)
```

يمكن أيضا استخدام الطريقة التالية في حال يوجد ضرورة للتعامل مع النص المكتوب في التعريف او ما شابه

```
widgets.HBox([widgets.Label(value="The $m$ in $E=mc^2$:"), widgets.FloatSlider()])

widgets.HTML( نافذة نص من نوع اتش تي ام ال
    value="Hello <b>World</b>",
    placeholder='Some HTML',
    description='Some HTML',
)
widgets.HTMLMath( لكتابة المعادلات الرياضية والرموز
    value=r"Some math and <i>HTML</i>: \(x^2\) and $$\frac{x+1}{x-1}$$",
    placeholder='Some HTML',
    description='Some HTML',
)
file = open("images/WidgetArch.png", "rb") لعرض صورة ما يجب فتح الصورة وحفظها في متغير
image = file.read() وبعدها قراءة الصورة
widgets.Image( نافذة عرض الصورة
    value=image,
    format='png',
    width=300,
    height=400,
)
widgets.Button(نافذة الضغط تختلف عن الزر الذي تم ذكره في النوافذ ذات القيم المنطقية
    description='Click me',
    disabled=False,
    button_style='', # 'success', 'info', 'warning', 'danger' or ''
    tooltip='Click me',
    icon='check'
)
```

احداث النوافذ Widget Events
هناك نوافذ لا تحفظ قيمة انما توضع لتسجيل النقرات عليها مثل الازرار

```
from IPython.display import display
button = widgets.Button(description="Click Me!") الحدث الذي يتم التعامل معه هو ضغط الزر
display(button)

def on_button_clicked(b): نعرف دالة للتعامل مع الحدث
   print("Button clicked.")

button.on_click(on_button_clicked) عندما يتم الضغط على الزر فعليا هذه الدالة يتم استدعاؤها فلذا نقوم بوضع الدالة
                                   المراد تشغيلها ضمنها

text = widgets.Text() هناك امر مشابه في قائمة النص وهو عند الضغط على زر ادخال
display(text)

def handle_submit(sender): نعرف دالة للقيام بامر ما
   print(text.value)

text.on_submit(handle_submit) وبنفس الطريقة السابقة يتم استدعاء الامر عند استدعاء الامر من خلال النافذة
                              يلزم في بعض الأحيان تسجيل جميع التحركات التي نقوم بها ضمن النافذة ويتم ذلك كالتالي
int_range = widgets.IntSlider() نقوم بتعريف شريط بسيط
display(int_range)

def on_value_change(change):
   print(change['new']) سيتم حفظ القيم عند أي تغير على شكل قاموس والكلمة جديد هي التي يمكن استخدامها لطباعة التغير

int_range.observe(on_value_change, names='value')
                                   يلزم في بعض الأحيان ربط اكثر من قطعة ببعضها لذلك يتم استخدام أوامر ربط او عدم الربط
import traitlets      استيراد المكتبة
انشاء سطر تعريف نصي
caption = widgets.Label(value = 'The values of slider1 and slider2 are synchronized')

انشاء شريطين تمرير
slider1 = widgets.IntSlider(description='Slider 1')
slider2 =  widgets.IntSlider(description='Slider 2')

انشاء ارتباط
l = traitlets.link((slider1, 'value'), (slider2, 'value'))

العرض
display(caption, slider1, slider2) هنا التحرك في أي من الشريطين يعني التاثر للشريط الاخر مباشرة

dl = traitlets.dlink((slider1, 'value'), (slider2, 'value')) اما لو تم استخدام هذا الامر فالتغير في الشريط الثاني لا يؤثر
                                                            على الشريط الأول اما تغير الشريط الأول فهو يؤثر على الشريط الثاني
l.unlink() يمكن فصل الارتباط بهذا الشكل
                                   يمكننا أيضا تغيير قيمة ما بحسب التغير في قيمة نافذة أخرى
caption = widgets.Label(value='The values of range1 and range2 are synchronized')
slider = widgets.IntSlider(min=-5, max=5, value=1, description='Slider')
```

```
def handle_slider_change(change):
   caption.value = 'The slider value is ' + (
      'negative' if change.new < 0 else 'nonnegative'
   ستتغير الكتابة فوق الشريط بتغير قيمة الشريط الجديدة ويمكن عرض نوع المدخلات والقيمة السابقة ومصدر المدخلات (
   print(change.owner,"\n",change.name)

slider.observe(handle_slider_change, names='value')

display(caption, slider)
يمكن ربط القيم بدون تأخر بالاستيراد من الكرينال عبر الربط المباشر للنوافذ
caption = widgets.Label(value = 'The values of range1 and range2 are synchronized')

range1 = widgets.IntSlider(description='Range 1')
range2 = widgets.IntSlider(description='Range 2')

l = widgets.jslink((range1, 'value'), (range2, 'value'))
display(caption, range1, range2)
widgets.jsdlink يمكن استخدام الربط الجزئي أيضا هنا
l.unlink() نستخدم الفصل أيضا بنفس الأوامر في الحالات السابقة

في بعض الحالات نريد ان نقوم باظهار التغير المباشر للقيم وفي بعضها فقط القيمة النهائية
import traitlets
a = widgets.IntSlider(description="Delayed", continuous_update=False)
b = widgets.IntText(description="Delayed", continuous_update=False)
c = widgets.IntSlider(description="Continuous", continuous_update=True)
d = widgets.IntText(description="Continuous", continuous_update=True)

traitlets.link((a, 'value'), (b, 'value')) سنقوم بانشاء شريطين ومربعين نص وربطهم ببعضهم الأول سيظهر التغير بعد
افلات المؤشر او انتهاء الكتابة في مربع النص بينها الثاني سيكون التغير مباشر التطبيق على البقية
traitlets.link((a, 'value'), (c, 'value'))
traitlets.link((a, 'value'), (d, 'value'))
widgets.VBox([a,b,c,d]) سنقوم بعرضهم ضمن مربع عرض
```

- التنسيق والانماط

  هناك خاصيتان للعمل عليهم في هذا القسم هما المكان والشكل المكان يتوفر في اغلب النوافذ بينما الشكل يتوفر فقط في البعض

  الخصائص المتوفرة في تنسيق المكان

  Sizes
  height
  width
  max_height
  max_width
  min_height
  min_width

  Display
  visibility
  display
  overflow

overflow_x
overflow_y

Box model
border
margin
padding

Positioning
top
left
bottom
right

Flexbox
order
flex_flow
align_items
flex
align_self
align_content
justify_content

عند انشاء شريط على سبيل المثال يكون مكانه تلقائي ويمكن ببساطة تغيير هذا المكان

```
import ipywidgets as widgets
from IPython.display import display

w = widgets.IntSlider()
display(w)
w.layout.margin = 'auto' الحواف
w.layout.height = '75px' الارتفاع بالبكسل
```

يمكن أيضا نقل خصائص ومكان احد الكائنات للاخر تماما كنسخ محتوى متغير

```
x.layout = w.layout
```

اما بالنسبة للشكل فمثلا للازرار تتوفر الخصائص التالية

```
'primary'
'success'
'info'
'warning'
'danger'
```

يتم تعريف هذه الأنماط في button_style

```
import ipywidgets as widgets

widgets.Button(description='Ordinary Button', button_style='')
widgets.Button(description='Danger Button', button_style='danger')
```

يمكن أيضا التغيير على الألوان

```
b1 = widgets.Button(description='Custom color')
b1.style.button_color = 'lightgreen'
b1
```

يمكن القاء نظرة على ما يمكن فعله عن طريق

```
b1.style.keys
s1 = widgets.IntSlider(description='Blue handle')
s1.style.handle_color = 'lightblue' يمكن أيضا تغيير لون المؤشر
s1
```
بعض الخصائص المتوفرة للنوافذ هي

```
Button
button_color
font_weight

IntSlider, FloatSlider, IntRangeSlider, FloatRangeSlider
description_width
handle_color

IntProgress, FloatProgress
bar_color
description_width
```
اما البقية فتحتوي فقط على description_width

- استخدام المخرجات

```
out = widgets.Output() نقوم بانشاء نافذة فارغة
out ونقوم بعرضها
with out: من خلال هذا الامر يتم التعامل وعرض المراد في هذه النافذة
  for i in range(10):
    print(i, 'Hello world!')
from IPython.display import YouTubeVideo يمكننا حتى عرض فيديو من اليوتيوب في هذه النافذة
with out:
  display(YouTubeVideo('eWzY2nGfkXk'))
```

- نافذة التشغيل

```
play = widgets.Play( نقوم بتعريف النافذة وتغذيتها بمعطيات أولية
  # interval=10,كلما ازدادت هذه القيمة كلما قلة سرعة الحركة
  value=50,
  min=0,
  max=100,
  step=1,
  description="Press play",
  disabled=False
)
slider = widgets.IntSlider()
widgets.jslink((play, 'value'), (slider, 'value'))
widgets.HBox([play, slider])
```

- نافذة ادخال تاريخ

```
widgets.DatePicker(
  description='Pick a Date',
  disabled=False
)
```

- نافذة اختيار لون

```
widgets.ColorPicker(
  concise=False,في هذه الحالة يقوم بكتابة اسم اللون في مربع نصي
  description='Pick a color',
```

```
    value='blue',
    disabled=False
)
```

- نافذة لاستقبال المعلومات من أداة تحكم الألعاب

```
widgets.Controller(
    index=0,
)
```

- صناديق العرض

```
widgets.Box(items)
widgets.HBox(items)
items = [widgets.Label(str(i)) for i in range(4)]
left_box = widgets.VBox([items[0], items[1]])
right_box = widgets.VBox([items[2], items[3]])
widgets.HBox([left_box, right_box])
```

- مربعات قابلة للطي

```
accordion = widgets.Accordion(children=[widgets.IntSlider(), widgets.Text()])
accordion.set_title(0, 'Slider')
accordion.set_title(1, 'Text')
accordion
```

- نوافذ تراتبية

```
tab_contents = ['P0', 'P1', 'P2', 'P3', 'P4']
children = [widgets.Text(description=name) for name in tab_contents]
tab = widgets.Tab()
tab.children = children
for i in range(len(children)):
    tab.set_title(i, str(i))
tab                         سيتم انشاء قوائم وفي كل قائمة مربع نص
tab.selected_index يمكن تحديد أي النوافذ يتم التعامل معها عن طريق هذا الامر
 يمكن أيضا تضمين النوافذ والمربعات القابلة للطي مع بعضها
tab_nest = widgets.Tab()
tab_nest.children = [accordion, accordion]
tab_nest.set_title(0, 'An accordion')
tab_nest.set_title(1, 'Copy of the accordion')
tab_nest
```