Conceptes bàsics de programació i Python, orientat a 1r de Batxillerat

Enric X. Martín Rull
Manel Velasco García
Juny 2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Estructura del curs:

PART 1. El llenguatge Python

PART 2. Conceptes d’algorísmia

PART 3. Utilització de llibreries

PART 4. Exercicis tipus

PART 2. Conceptes d'algorísmia

1. Conceptes d'algorísmia
2. Estructura del codi
3. Estructura de les dades
4. Creació de classes
5. Qüestions d'estil

2.1. Conceptes d’algorísmia

Abans de posar-nos a fer un programa, val la pena fer-nos una sèries de preguntes i un petit anàlisi del que anem a fer:

- quin problema anem a resoldre?
- quines dades d’entrada tindrà el nostre programa?
- podem definir totes les dades possibles a l’entrada?
- hi ha interacció amb usuaris?
- utilitzem dades guardades a fitxers?
- què s’ha de mostrar a pantalla?
- com acaba el programa?
- ...

2.1. Conceptes d’algorísmia

- quin és el resultat de l’execució del programa?
- quina relació hi ha entre l’entrada i la sortida del programa? la podem representar d’alguna manera?
- la solució al problema que volem resoldre, quina estructura té? és una seqüència? va prenent decisions? repeteix operacions?
- si la solució (nombre de passos) és molt llarga, la podem dividir en blocs més petits?
- …

2.1. Conceptes d'algorísmia

- si hi ha conjunts d'operacions que es repeteixen, les podem agrupar en blocs que anirem cridant (funcions)?
- Les dades que utilitzarà el programa, tenen alguna estructura concreta? són una llista de valors? estan ordenades?
- ...

2.1. Conceptes d’algorísmia

La resposta a aquestes preguntes (text, gràfica, diagrames de fluxe, etc.) ens ajudarà a tenir clara l’estructura del programa i els recursos que necessitarà:

- us de pantalla (text, gràfic)
- us de fitxers (quíns, on)
- entrada per part de l’usuari (què, com)

- dades internes a usar, tipus, estructures

- proves a fer per comprovar el funcionament del programa (joc de proves)

2.1. Conceptes d'algorísmia

Anomenem **algorisme** una seqüència precisa d'operacions o pasos que resolen un problema en un temps determinat (que ha de ser finit).

L'algorisme és el mètode i és **independent** del llenguatge de programació o de la màquina que l'executarà.

2.1. Conceptes d'algorísmia

Per especificar un algorisme cal fer un esforç de **formalització**.

Exemple. Volem obrir la porta de casa nostra

Agafem el clauer
Busquem la clau de la porta de casa (sabem la forma o el color)
Posem aquesta clau al pany
Girem en sentit antihorari una o dues voltes (recordem com va el pany)
Empenyem o estirem la porta segons com estigui muntada
Treiem la clau del pany

2.1. Conceptes d’algorísmia

Per especificar un algorisme cal fer un esforç de **formalització**.

Exemple. Volem obrir la porta de casa nostra

Seqüència

Agafem el clauer
Busquem la clau de la porta de casa (sabem la forma o el color)
Posem aquesta clau al pany
Girem en sentit antihorari una o dues voltes (recordem com va el pany)
Empenyem o estirem la porta segons com estigui muntada
Treiem la clau del pany

2.1. Conceptes d'algorísmia

Fixem-nos que tenim una certa quantitat de coneixements i **condicions d'abans** de començar a obrir:

{ tenim el clauer **i**
la clau de casa és al clauer **i**
sabem quina és la clau (per forma, color, és la única…) **i**
sabem que el pany és d'una volta o de dues, **o**
sabem que normalment el tanquem amb una o dues voltes, **i**
sabem que la porta obre cap a dins o cap a fora
}

Precondicions: P (poden implicar us de memòria).

2.1. Conceptes d’algorísmia

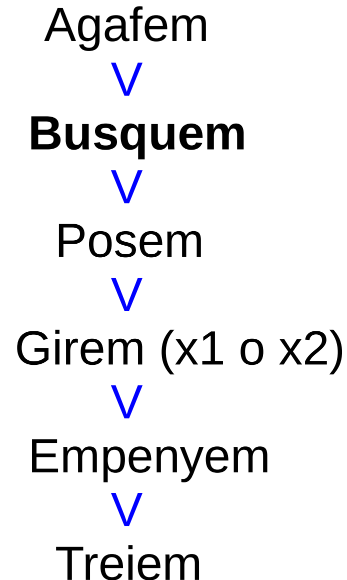
Podem definir una **condició d’èxit**, (el que s’espera al final) que indica que ha acabat l’operació:

{ la porta està oberta i
la clau està treta
}

Postcondicions: Q

2.1. Conceptes d'algorísmia

Al definir la seqüència podem avaluar els àtoms d'activitat I si alguns d'ells tenen més complexitat que els altres, es repeteixen, etc.



2.1. Conceptes d'algorísmia

Desplegament de **Busquem**:

P: precondicions, condicions d'abans que s'han de complir:

{ tenim el clauer **i**
la clau de casa és al clauer **i**
sabem quina és la clau (per forma, color, és la única…)
}

També té una **condició d'èxit, Postcondició, Q**
{
la clau de la porta és a la nostra ma
}

2.1. Conceptes d’algorísmia

De quines operacions consta la **cerca** (Busquem) de la clau?

- Despleguem el clauer per poder veure totes les claus
- Mirem la clau de més a l’esquerra.
- **Si no** és la de casa, mirem la següent a la seva dreta. **Si si** que ho és, l’agafem amb la ma i hem acabat
- **repetim** el pas anterior fins trobar-la.

Atenció! Veiem que la condició lògica de “la clau de casa és al clauer” ens garanteix que el procediment on anem repetint un pas acabarà.

2.1. Conceptes d’algorísmia

Es poden definir tres **estructures** lògiques que poden resoldre qualsevol problema integrades en un algorisme:

- seqüència
- alternativa
- iteració

2.1. Conceptes d'algorísmia

Seqüència

{P}	(precondicions)
S1	
S2	(accions)
…	
SN	
{Q}	(postcondicions)

2.1. Conceptes d’algorísmia

Exemple: donat un nombre, volem calcular el seu doble més tres.

{ Precondició P: x és un nombre real }

y = 2 * x
y = y + 3

{Postcondicions Q: y és un nombre real, y = 2x + 3 }

(funcions no línials com log, sqrt, poden trencar la postcondició: y és real)

2.1. Conceptes d'algorísmia

Alternativa

{P} (precondicions)
si B_1 -> S1
si B_2 -> S2
…
si B_N -> SN
{Q} (postcondicions)

Per cada cas **cal** complir: {P ∧ Bi}
Si
{Q}

I si (B_1 v B_2 v B_3 v … v B_N) **no** és cert per {P} **falten** casos.

2.1. Conceptes d'algorísmia

Exemple: donat un nombre, volem calcular el seu valor absolut.

{ Precondició P: x és un nombre real }

si $x>0$ -> $y = x$
si $x<0$ -> $y= -x$

{Postcondicions Q: y és un nombre real, $y = | x |$ }

UPC
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB

2.1. Conceptes d’algorísmia

Podem **verificar** que pels dos casos es compleix la postcondició.

(1) { P: x és un nombre real $\wedge$ $x>0$ }
$y = x$
{Q: y és un nombre real, $y = | x |$ }

(2) { P: x és un nombre real $\wedge$ $x<0$ }
$y = -x$
{Q: y és un nombre real, $y = | x |$ }

En canvi veiem que: $(x>0)$ v $(x<0)$ **no és cert per tot P**: x és un nombre real, donat que ens deixem el cas $x=0$. Seria millor fer la primera condició: $x >=0$.

2.1. Conceptes d'algorísmia

Iterativa

```
{P}
mentre B fer
    S
fimentre
{Q}
```

- Mentre es dongui que l'expressió booleana B és certa, s'anirà executant la seqüència S d'operacions.
- Aquesta seqüència i l'acabament de la condició B, hauran de portar la Precondició P a la Postcondició Q.
- Si S no modifica l'estat de B, no acabarà mai.

2.1. Conceptes d'algorísmia

Exemple. Volem fer el producte de tots els nombres entre 1 i N (factorial).

{ Precondicions P: N és un nombre natural, N>0 ; F = 1; i = 1 }

```
mentre ( i <= N )
        F = F * i
        i = i + 1
fimentre
```

{Postcondició Q: F = N ! }

La condició booleana de manteniment dins del bucle, és, B: i<=N

2.1. Conceptes d'algorísmia

Desplegament del funcionament de la iteració:

Volta	i (a l'entrar)	F (a l'entrar)	i (al sortir)	F (al sortir)	N
1	1	1	2	1	5
2	2	1	3	2	5
3	3	2	4	6	5
4	4	6	5	24	5
5	5	24	6	120	5
6	6	Al valdre i=6 acaba i surt del mentre			

Les condicions que s'han de complir a l'inici i final de cada iteració s'anomenen **invariant** del bucle. En aquest cas es compleixen dues condicions:

I : (1 <= i <= N+1) i F = (i - 1) !

Quan acaba la condició de permanència B, I implica que es compleix Q.

2.2. Estructura del codi

Seqüència

Exemple en Python: mostrar el valor de la funció $f(x) = x^3 + x^2 - x + 4$ en el punt x=2.0

```
x=2.0                            # definim variable x i li donem valor d’entrada
resultat=0.0                     # definim variable resultat pel final

resultat = x**3                  # fem les operacions
resultat = resultat + x**2       # fem les operacions
resultat = resultat -x           # fem les operacions
resultat = resultat + 4          # fem les operacions

print ( resultat)                # mostrem resultat
```

2.2. Estructura del codi

Alternativa

Exemple d’alternatives en Python:

```
if (condició 1):
	S1
if (condició 2):
	S2
if (condició 3):
	S3
```

Amb aquesta estructura es poden executar **diverses** o **totes** les seqüències S1,S2, S3 segons si hi ha més d’una condició certa.

```
if (condició 1):
	S1
elif (condició 2):
	S2
elif (condició 3):
	S3
```

Amb aquesta estructura només es pot executar **una** de les seqüències, serà la primera que trobi la seva condició certa.

2.2. Estructura del codi

Iteració: cerca

L'estructura habitual de cerca d'un element és la següent:

```
while ( not final ):
    element = obtenir_següent ( )
    final = processar ( element )
resultat = element
```

En cas de que no estigui garantida la presència de l'element és:

```
while ( not final ) and (queden_elements) :
    element = obtenir_següent ( )
    final = processar ( element )
if (final):
    resultat = element
else:
    resultat = ERROR
```

2.2. Estructura del codi

Iteració: cerca

Exemple en Python: Donada una llista , mirar si hi ha un nombre major que 50.

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

n = len (Llista)
i=0
trobat = False

while ( i < n ) and (trobat==False) :
    trobat  = ( Llista [ i ] > 50)
    i = i + 1

if (trobat):
    print (“ si hi ha un nombre major de 50”)
```

2.2. Estructura del codi

Iteració: recorregut (cal passar per tots els elements)

Exemple en Python: Donada una llista de nombres, mirar quants n'hi ha majors que 50.

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

n = len (Llista)
i=0
nombre_majors = 0

while ( i < n ) :
if ( Llista [ i ] > 50):
        nombre_majors = nombre_majors + 1
        i = i + 1
print (“ Hi ha “, nombre_majors, “ més grans de 50”)
```

2.2. Estructura del codi

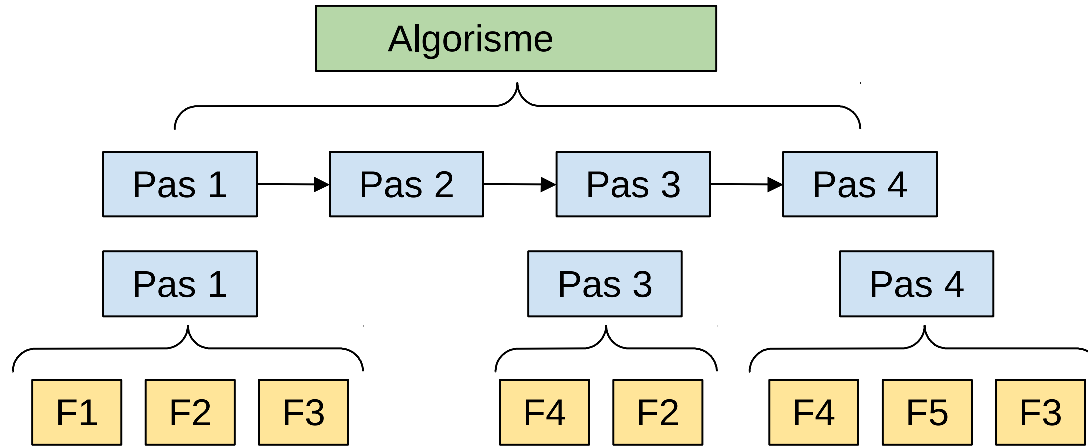
Iteració: recorregut (cal passar per tots els elements)

Alternativa per recorregut: **FOR**

```
Llista = [12, 34, 11, 25, 5, 67, 57, 46, 2, 1, 3 ]

nombre_majors = 0

for element in Llista :
    if ( element > 50):
        nombre_majors = nombre_majors + 1

print (“ Hi ha “, nombre_majors, “ més grans de 50”)
```

2.2. Estructura del codi

Estructuració dels passos previs al codi: disseny descendent.



Permetrà dividir tasques i reaprofitar codi.

2.2. Estructura del codi

Recursivitat

Hi ha algorismes que de forma nativa neixen com a recursius.

Exemple, el factorial de N.

$$f(1) = 1$$
$$f(N) = N \times f(N-1)$$

2.2. Estructura del codi

Recursivitat

Exemple en Python

```
#Definició de funció
def factorial_recursiu (n):
    if n==1:
        return 1
    else:
        return n * factorial_recursiu (n-1)

#Programa
s = input (“Entra un nombre: “)
x = int (s)
print ( “El factorial de ”, x,” és ”, factorial_recursiu (x) )
```

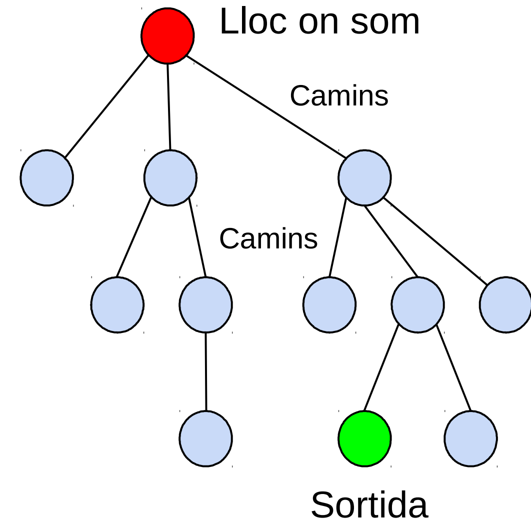
2.2. Estructura del codi

Recursivitat

Sortir del laberint

Recursivitat múltiple

*funcio **Sortir_laberint** (Habitació)*
si *Estem_a_la_Sortida (Habitació)*
Acabar ( )
sinó
***per** C **en** Camins_possibles (Habitació)*
Sortir_laberint (Seguir_Camí (C))



2.3. Estructura de les dades

Taules, matrius i tuples

Una taula és un tipus de dades estructurat, homogeni, amb accés directe als elements i en principi de dimensió fixa.

TaulaVisitants[12]

0	0	0	0	0	0	0	0	0	0	0	0

On cada element serà el comptador de persones de un dels mesos. Hi podrem accedir fent x = TaulaVisitants[3]

2.3. Estructura de les dades

Taules, matrius i tuples

Una matriu és un tipus de dades estructurat, homogeni, amb accés directe als elements i en principi de dimensió fixa. Es considera multidimensional*.

*A nivell lògic, a memòria, no.

TaulaVisitants[12][10]

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

A les files i columnes hi tindrem els comptadors de visitants dels diferents mesos i anys. Hi accedirem així: x = TaulaVisitants [2][5]

2.3. Estructura de les dades

Taules, matrius i tuples

Una **tupla** és un tipus de dades estructurades, amb accés directe, de dimensió fixa, però de tipus **heterogenis**. Veiem un exemple amb el tipus tupla de Python, on guardem 3 dades: dues cadena i una numèrica:

Alumne = (“Lucía”, 13, “2n ESO”)

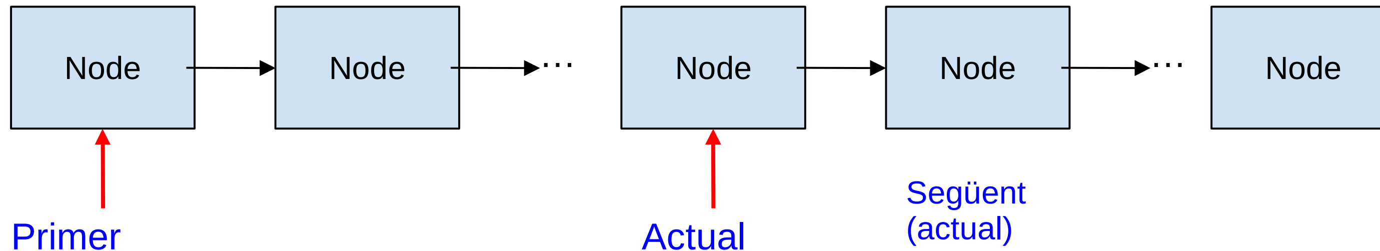
En Python podem crear llistes de tuples i després treballar amb elles, fer cerques o diferents ordenacions.

Grup = [(“Lucía”, 13, “2n ESO”), (“Mar”, 14, “2n ESO”), (“Lee”, 14, “3r ESO”), (“Jonathan”, 16, “4t ESO”)...]

2.3. Estructura de les dades

Llistes, piles i cues

Des del punt de vista d'organització, una **llista** és una estructura on tenim un element que és el primer (tindrem un apuntador a ell), un element que és l'actual (serà un apuntador que va canviant) i una opcionalment una operació que donat un element ens pot dir quin és el seu següent.

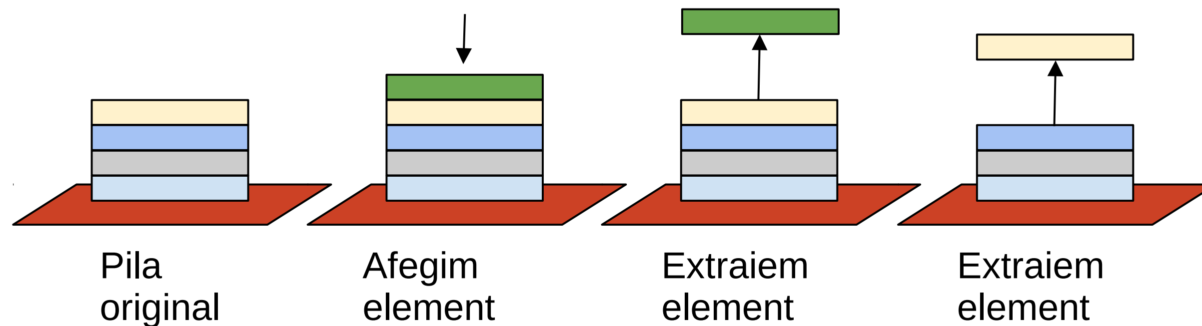


Python permet accedir a llistes com si fossin taules.

2.3. Estructura de les dades

Llistes, piles i cues

Una **pila** és una estructura on la inserció (push) i l'extracció (pop) d'elements es fa sempre per dalt (top).

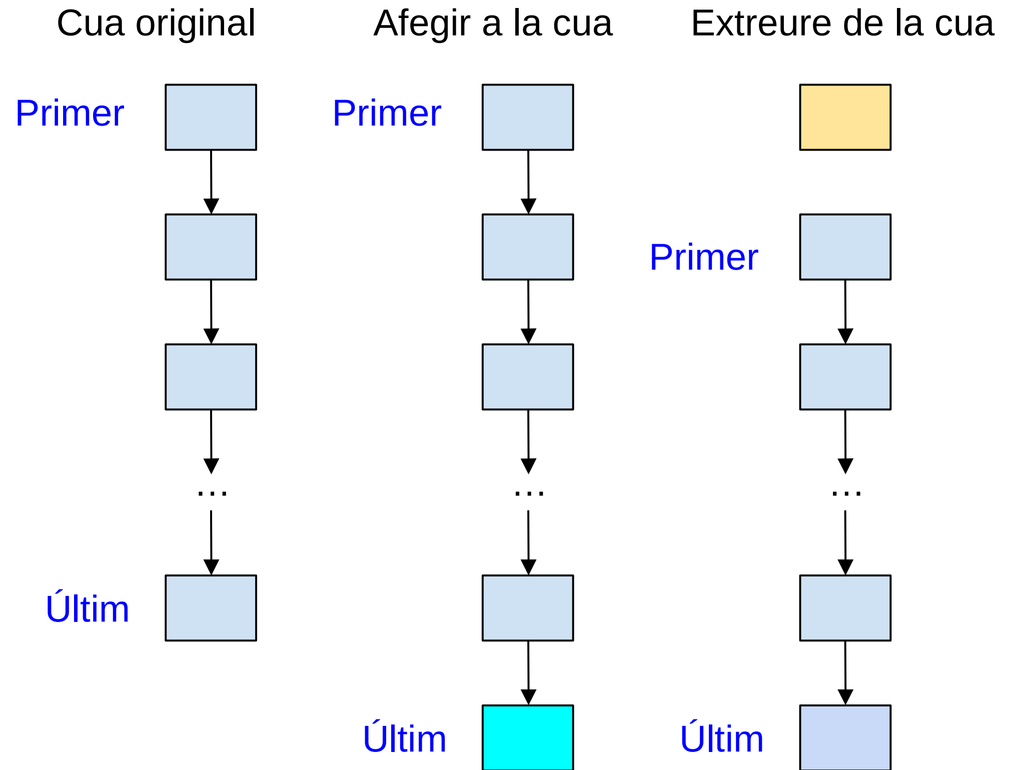


Python no suporta piles però es poden crear a partir de llistes.

2.3. Estructura de les dades

Llistes, piles i cues

Una **cua** es gestiona a partir del primer (és el primer a sortir) i l'últim (que deixarà de ser-ho a l'arribar un nou element).



Python no suporta cues però es poden crear a partir de llistes.

2.3. Estructura de les dades

Llistes, piles i cues

Aquests tres tipus són els principals “Tipus abstractes de dades” TAD, que es fan servir a la informàtica.

Altres són els arbres o els grafs.

La implementació de l’estructura de dades del TAD i les seves funcions associades les veurem a continuació, fent-ne un encapsulament.

2.4. Creació de classes

Els llenguatges de programació permeten crear **classes** o **objectes** on sota un concepte concret, podem agrupar-hi estructures de dades i les operacions per treballar-hi.

Exemple: implementació d'una cua a partir de llistes.

class declara la nova classe,

def són les diferents funcions,

__init__ és crida un cop a l'inici.

```
class cua():

    def __init__(self):
        self.dades=[]

    def afegir(self, element):
        self.dades.append(element)

    def extreure(self):
        return self.dades.pop(0)

    def buida(self):
        return len(self.dades)==0
```

2.4. Creació de classes

Exemple:

Utilització de les classes.

```
class cua():

    def __init__(self):
        self.dades=[]

    def afegir(self, element):
        self.dades.append(element)

    def extreure(self):
        return self.dades.pop(0)

    def buida(self):
        return len(self.dades)==0


c = cua()
c.afegir("Joan")
c.afegir("Maria")
c.afegir("Kevin")

while not c.buida():
    print (c.extreure())
```

2.4. Creació de classes

Exemple 2:

Creació de llistes genèriques a partir de la classe llista de Python.

```
class llista ():

    def __init__(self):
        self.dades=[]
        self.actual=0
        self.num=0

    def seleccionar (self, nombre):
        self.actual = nombre

    def afegir(self, element):
        self.dades.insert(self.actual,element)
        self.num=self.num+1

    def extreure(self):
        self.num=self.num-1
        return self.dades.pop(self.actual)

    def consultar(self):
        return (self.dades[self.actual])

    def buida(self):
        return len(self.dades)==0

    def quants(self):
        return self.num
```

2.4. Creació de classes

Exercici 2.4

Després de veure la implementació als exemples de les classes cua i llista…

Dissenya una classe pila amb les funcions:

- apilar (element)
- desapilar ( ) que retorna un element
- buida ( ) que retorna un booleà indicant si està buida o no.

Tot respectant el funcionament lògic de la pila.

Prova que funciona bé.

2.5. Qüestions d'estil

Estructura general dels fitxers de codi

(1)
 Comentaris inicials: Nom del programa, autor, data versió.
(2)
importació de mòduls o funcions (si cal)
(3)
declaració de funcions pròpies que farem servir (si cal)
declaració de variables que farem servir

(4)
programa que desenvolupem

2.5. Qüestions d’estil

Noms de les variables i estructures

Els noms de les variables seran generosos, s’entén millor el codi si les variables tenen noms que fan referències a les seves funcions. Millor “radi”, “àrea”, “entrada” que no a,b,c… etc.

Identació del codi

Hi ha llenguatges com C que els blocs de codi (generats per if, while o for) s’han de posar entre claus ‘ { ‘ iniciar, ‘ } ’ tancar. El llenguatge Python obliga a que els blocs estiguin tabulats.

```
while (condició):
    bloc de codi
    del while
continuació_programa
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d’Informàtica de Barcelona
UPC
FIB

2.5. Qüestions d'estil

Comentaris

A banda del comentari inicial sobre l'autor, data i versió, serem generosos posant comentaris a les funcions, a parts del codi interessants, per indicar on comença o acaba un bloc, etc.

Expressions lògiques

Tot i que nosaltres podem conèixer perfectament com s'avaluen les expressions lògiques al llenguatge de programació que utilitzem, expressions molt llargues poden portar a confusió o ser font d'errors

2.5. Qüestions d'estil

Expressions lògiques

Exemple. Volem saber si un punt al pla R^2 està al primer o tercer quadrant.

```
if x>0.0 and y>0.0 or x<0.0 and y<0.0:
        print ("SI")
else:
        print (“NO”)
```

Millor escriure:

```
if  ( x>0.0 and y>0.0 )  or  ( x<0.0 and y<0.0 ):
        print ("SI")
else:
        print (“NO”)
```

2.5. Qüestions d'estil

Llargada i profunditat dels blocs

Si el codi de dins d'un bucle while, for, o el codi del cas d'un if o else, és tan llarg que no cap en una pantalla, això dificultarà la seva comprensió (anar pujant i baixant). És probable que algun tros es pugui extreure a nivell lògic cap a funcions.

Si els blocs de codi van aprofundint en excés, o s'acumulen massa avaluacions de casos i condicions, serà impossible fer la seva verificació i dificulta la comprensió:

```
if (condicio1):
    while (condicio2):
        if (cas1):
            codi
        elif (cas2):
            while (condicio3):
                for recorregut_estructura:
                    etc…
```