Pontificia Universidad Católica de Chile
Departamento de Ciencias de la Computación
Escuela de Ingeniería

Introducción a la Programación
IIC1103

Segundo semestre 2013

Índice general

Capítulo 1

Prólogo

Este documento tiene como objetivo resumir los contenidos del curso **IIC1103 Introducción a la Programación** dictado el segundo semestre del año 2013. Su objetivo no es de ninguna forma reemplazar la asistencia a clases o laboratorios. Tampoco tiene como fin ser la única fuente de estudio para el alumno, puesto que los contenidos son lo suficientemente amplios como para requerir bastante tiempo para lograr dominarlos en profundidad.

Se recomienda fuertemente al lector complementar los contenidos del documento con ejercitación de su parte. Esta puede ser rehacer las tareas planteadas durante el semestre, los laboratorios o incluso ejercicios propuestos de internet.

También es importante recordarle al alumno que este texto fue realizado por otro ser humano, por lo que puede contener errores menores. Sin embargo, todo el código utilizado aquí ha sido previamente testeado. Cualquier feedback o crítica puede ser enviada al mail **assoto@uc.cl**[1].

Para finalizar, es necesario recordarle al lector que la mejor forma de preparar una evaluación en este curso es programar, ya que el lector debe ser capaz de resolver por sus propios medios los problemas que requieran de la modelación y posterior desarrollo de un algoritmo.

[1] Desearía aprovechar este espacio para agradecer a Esteban Andrés Salgado por regalarme su portada =).

Capítulo 2

Conceptos previos

2.1. Introducción a los algoritmos

Antes de comenzar a programar debemos comprender el concepto de algoritmo. Un algoritmo es un conjunto finito de pasos definidos y ordenados que nos permiten realizar una actividad de manera metódica. Veamos un ejemplo básico de algoritmo.

Ejemplo 2.1 Diseñe un algoritmo que decida cargar un teléfono celular si le queda poca batería.

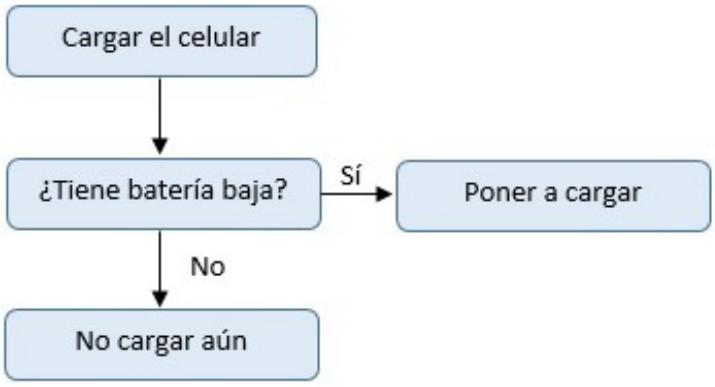

Figura 2.1: algoritmo para decidir cuando cargar un telefono celular.

Es fundamental modelar la solución de un problema mediante un algoritmo antes de comenzar a programar, pues el algoritmo es independiente del lenguaje de programación. Una vez modelado el problema, sólo debemos adaptarlo al lenguaje en el que deseamos escribir la respuesta para poder realizar lo pedido.
Cabe destacar que este documento no tiene por fin enseñar algoritmos ya que es un tema perteneciente a otros cursos. Se propone al lector que averigue un poco acerca de ellos para facilitar la modelación de los problemas planteados. Veamos a continuación otros dos ejemplos ilustrativos.

Ejemplo 2.2 Diseñe un algoritmo que filtre el ingreso de personas a un casino. Una persona puede entrar a un casino si tiene más de 18 años y si paga su entrada.

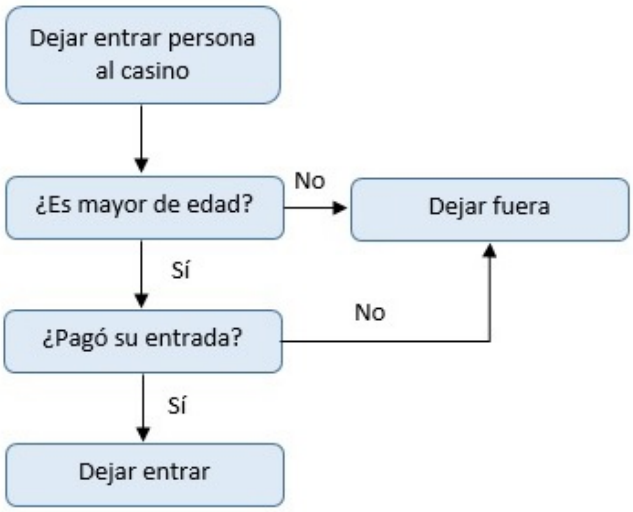

Figura 2.2: algoritmo para filtrar el ingreso de personas a un casino.

Antes de continuar es necesario conocer el concepto de **variable**. En computación una variable es un espacio en memoria reservado para almacenar información que puede o no ser conocida. En general a las variables les daremos un nombre y le asignaremos valores según corresponda.

Ejemplo 2.3 Diseñe un algoritmo que dado un número natural n positivo, retorne la suma de los n primeros números naturales.

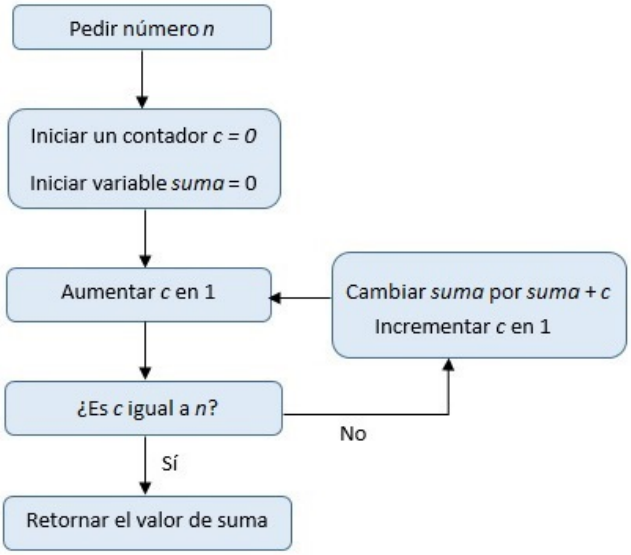

Figura 2.3: algoritmo para retornar la suma de los primeros n naturales.

Para finalizar la sección, cabe destacar que la modelación de los problemas será repasada antes de dar solución a los ejemplos y ejercicios planteados en este texto, con el fin de mostrar la importancia de programar en base a un modelo previo.

2.2. Creando un nuevo proyecto en Python

A continuación detallaremos cómo iniciar un nuevo proyecto en Python 3 usando el IDLE, el entorno de programación disponible en http://www.python.org/download.
Para esto abrimos IDLE. Luego, para crear un nuevo proyecto debemos presionar ctrl+n o ir a la pestaña File y abrir una ventana nueva:

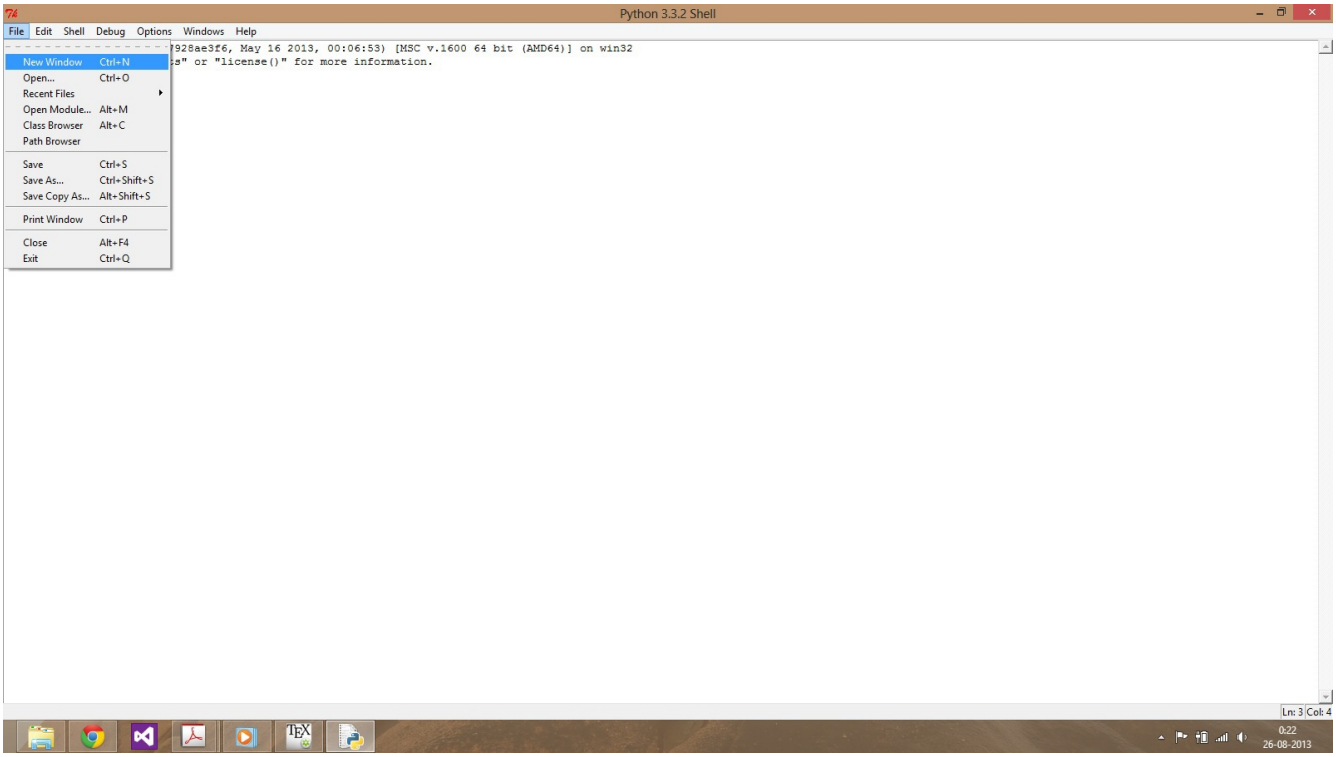

Figura 2.4: creando un nuevo proyecto en el IDLE.

Se nos abrirá una nueva ventana en la que debemos escribir el código de nuestros programas. Luego al guardar algún archivo escrito en esta ventana generaremos un archivo de extensión .py.
Nuestro primer proyecto en Python será el famoso *"Hello world!"*[1], que consiste en un programa que imprime en pantalla "Hello world!". Para esto escribimos:

```
print("Hello world!")
```

Y luego para ejecutarlo presionamos f5 o Run Module en la pestaña **Run**.

[1]En general, este programa es el primero que se escribe al aprender un nuevo lenguaje

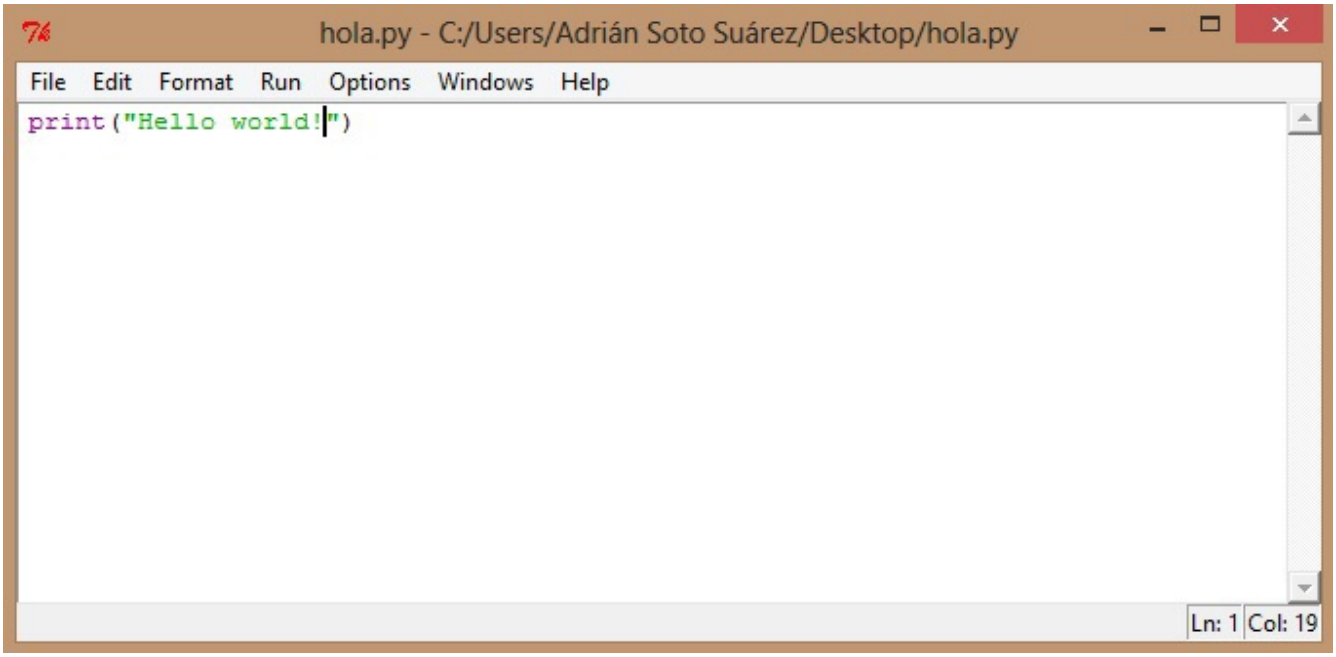

Figura 2.5: primer programa en Python.

Capítulo 3

Tipos de datos, expresiones y control de flujo

3.1. Tipos de datos

En Python, todo elemento tiene un tipo. Este tipo determina las acciones que podemos realizar sobre el elemento o bien el dominio de los valores posibles que puede tomar. Para declarar y asignarle un valor a una variable en Python debemos usar el simbolo =. Entre los tipos de valores posibles destacan varios.

- *int*: una variable de tipo int representa un número entero o en inglés "*integer*". Puede tomar valores enteros y es posible realizar operaciones aritméticas sobre él. Por ejemplo[1]:

```
# Recordar que al usar el simbolo # es una linea comentada que no influye en el programa
a=2
b=5
c=a+b # La variable c tiene el valor 7
d=5/2 # La variable d tiene el valor 2.5
e=b%a # La variable e tiene el valor 1, que es el resto de 5/2
```

- *float*: una variable de tipo float representa un número racional. Su nombre proviene de "*floating point*" (punto flotante en español) y es posible realizar operaciones aritméticas sobre él. El siguiente código muestra algunos procedimientos básicos:

```
a=2.5
b=5.1
c=a+b # La variable c tiene el valor 7.6
d=5/2 # La variable d tiene el valor 2.5
e=5//2 # La variable e tiene el valor 2, correspondiente a la division entera entre 5 y 2
f=2**4 # La variable f tiene el valor 16, que es 2 elevado a 4.
```

- *str*: los objetos de tipo *str* son los tipos llamados strings. Los strings sirven para representar texto mediante una "cadena de caracteres". Es importante notar que la concatenación de dos strings se hace con el operador + y que todo texto entre comillas representará strings. Como por ejemplo:

[1] El código en Python no llevará tilde puesto que se cae en algunos sistemas operativos. La razón de esto escapa a los conocimientos del curso.

```
texto1 = "hola"
texto2 = "chao"
texto3 = texto1 + texto2 # Esta variable tiene valor "holachao"
```

- *bool*: los objetos de tipo bool son variables que pueden tomar sólo dos valores **True** o **False**. Son de mucha importancia, pues pueden representar condiciones de término o de fin de un programa o una parte de este. Es útil saber lo siguiente:

  - Para saber si una variable a es igual a otra variable b debo escribir a==b. Esto es válido tanto para tipos *int*, *float* y *str* entre otros.
  - Para saber si una variable a es mayor o igual a otra variable b debo escribir a >= b.
  - Para saber si una variable a es menor o igual a otra variable b debo escribir a <= b.
  - Para saber si una variable a es mayor a otra variable b debo escribir a>b.
  - Para saber una variable a es menor a otra variable b debo escribir a<b.

  Existen además tres operadores booleanos que debemos conocer:

  - *and*: representa el operador lógico $\wedge$.
  - *or*: representa el operador lógico $\vee$.
  - *not*: cambia el valor de verdad de una variable booleana.

  Ejemplos de variables booleanas serían los siguientes:

```
variable1 = 2>5 # Esta variable tiene valor False
variable2 = 3<5 # Esta variable tiene valor True
variable3 = True # Esta variable tiene valor True
variable4 = 5==5 # Esta variable tiene valor True ya que 5 es igual a 5
text = "hola"
var5 = text == "hola" # Esta variable es True
var6 = 5=="5" # Esta variable es False pues el entero 5 es distinto al string (o caracter) 5
var7 = 5 >=4 # Esta variable es True pues 5 es mayor o igual a 4
dist = 5!=4 # Esta variable es True pues 5 es distinto de 4
nodist = not(5!=4) # Esta variable es False pues negamos el valor de 5!=4
hola = 5>4 and 5<4 # Esta variable es False pues and exige que todas las clausulas sean True
chao = 5>4 or 5<4 # Esta variable es True pues or exige que una de las clausulas sea True
```

Cabe destacar la existencia del tipo None que es el tipo nulo, es decir, es una variable aún no inicializada. También recordemos que existen varios otros tipos. Se propone al lector buscarlos.

3.2. Comandos Básicos

Python incluye algunos comandos básicos que debemos conocer antes de comenzar a programar. Estos son funciones que vienen implementadas en Python. Cabe destacar que el concepto de función es bastante amplio y se detallará con profundidad en los siguientes capítulos del texto. Por ahora, nos bastará con saber que una función es una entidad que recibe ciertos parámetros, y en base a eso, ejecuta ciertas acciones que pueden terminar con el retorno de un valor. Las funciones más básicas (y que necesitamos hasta ahora) son las siguientes:

- *print*: print es una función que recibe como parámetro un string y lo imprime en la consola. También recibe parámetros que no sean strings y los imprime, como *int* y *float* pero es importante saber que si queremos escribir texto y concatenarle una variable que no es string, esta variable debe ser tratada como string mediante *str*(). Veamos esto en un ejemplo:

```
print ("hola") # Esto imprime en consola hola
print ("Mi nombre es: "+"Miguel") # Esto imprime en consola Mi nombre es: Miguel
a = 3
print (a) # Esto imprime en consola 3
print ("El numero es: "+a) # Esto genera error de tipos
print ("El numero es: "+str(a)) # Esto imprime en consola El numero es: 3
```

- *input*: input es una función que recibe como parámetro un string y pide otro string en consola. Este puede ser asignado a una variable. Si quisieramos pedir números, debemos transformar el string ingresado en número mediante *int*() o *float*(). Se debe realizar lo mismo al recibir un *bool*(). Ejemplos de esto escritos en Python son:

```
a = input ("Diga un nombre : ")
b = int( input ("Diga un numero : "))
c = int( input ("Diga otro numero : "))
print ("La suma de los numeros de "+a+" es: "+str(b+c))
```

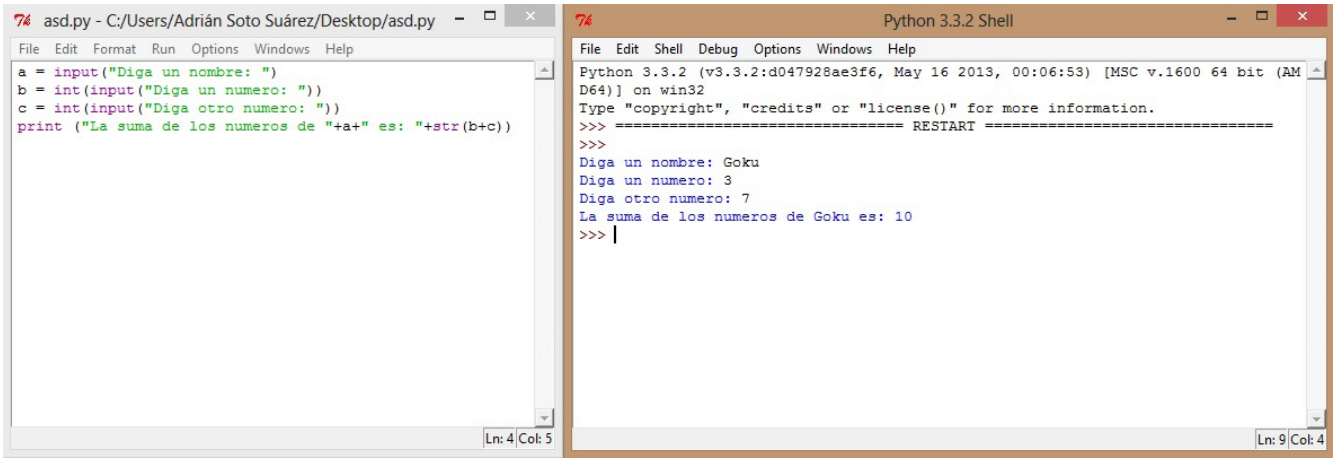

Figura 3.1: resultado del ejemplo de *intput*.

3.2.1. Operadores

Para los tipos *float* e *int* existen operadores entre dos números a y b:

- $+$: entrega la suma de los números a y b.
- $-$: entrega la resta de los números a y b.
- $*$: entrega la multiplicación los números a y b.
- $/$: entrega la división de los números a y b.
- $//$: entrega la división entera de los números a y b.

- %: entrega el resto de la división de los dos números a y b.
- $**$: entrega el resultado de elevar a por b.

3.3. Programas básicos

Con los conocimientos adquiridos hasta el momento ya es posible hacer programas básicos que interactúen con el usuario pedir un input y devolver algo en consola. Los dos ejemplos presentados a continuación ilustran de buena forma lo que somos capaces de hacer:

Ejemplo 3.1 Escriba un programa que pida al usuario el lado de un cuadrado y entregue el área, el perímetro y el valor de su diagonal.

En este problema debemos hacernos cargo de tres cosas:

- Recibir el valor del input y guardarlo como un *float*.
- Hacer las operaciones aritméticas respectivas.
- Imprimir en pantalla los resultados pedidos.

Notamos que el primer requerimiento lo hacemos con la función *input*. El segundo requerimiento se logra al hacer las operaciones aritméticas pertinentes y almacenarlas en variables. El tercer requerimiento hace uso de la función *print*. La solución propuesta es:

```
lado = float (input ("Ingrese el lado del cuadrado: "))
perimetro = lado*4
area = lado**2
diagonal = lado*(2**(1/2))
print ("El perimetro es: "+str( perimetro ))
print ("El area es: "+str( area ))
print ("La diagonal es: "+str( diagonal ))
```

Ejemplo 3.2 Usted está en un curso en el que tiene 3 pruebas que ponderan un 20 % cada una, y un examen quepondera el 40 % restante. El curso se aprueba si el promedio no aproximado del curso es mayor o igual que 4.0. Escriba un programa que reciba las 4 notas e imprima *True* si usted aprueba el curso y *False* en caso contrario.

Para resolver el ejemplo debemos recibir las notas correspondientes y realizar las operaciones aritméticas pertinentes. Aquel valor se compara a 4.0 para ser asignada la respuesta a una variable booleana. Luego esta se imprime en consola.

```
n1 = float (input ("Ingrese la nota de la prueba 1: "))
n2 = float (input ("Ingrese la nota de la prueba 2: "))
n3 = float (input ("Ingrese la nota de la prueba 3: "))
ex = float (input ("Ingrese la nota del examen : "))
pasaste = (0.2* n1 +0.2* n2 +0.2* n3 +0.4* ex ) >=4.0
print (pasaste)
```

Notemos que el programa anterior es equivalente a escribir el siguiente código:

```
n1 = float (input ("Ingrese la nota de la prueba 1: "))
n2 = float (input ("Ingrese la nota de la prueba 2: "))
n3 = float (input ("Ingrese la nota de la prueba 3: "))
```

```
ex = float (input ("Ingrese la nota del examen : "))
print ((0.2*n1 + 0.2*n2 + 0.2*n3 + 0.4*ex) >= 4.0)
```

3.4. If, elif y else

En el ejemplo 3.2, se obtenía como resultado una impresión en pantalla de una variable booleana, la que podía ser *True* o *False*. En general, no se desea recibir una impresión en pantalla de la variable booleana, sino que se desea un mensaje distinto dependiendo del valor de la variable. En los lenguajes de programación por lo general existe una manera de ejecutar cierto grupo de acciones dependiendo si una sentencia se cumple o no, es decir, si una variable booleana asociada a la sentencia es verdadera o falsa. En particular, en Python la sintaxis es la siguiente:

```
if <variable y/o expresion booleana>:
  #Ejecutar aqui las acciones indentandolas
```

Lo que hace el código de arriba es ejecutar las acciones declaradas en el *if* si se cumple la condición o si la variable booleana es *True*. Notemos que tras la variable y/o expresión booleana hay dos puntos (:) y que las acciones a ejecutar deben estar indentadas, es decir, desplazada 4 espacios dentro respecto al if declarado (también se puede hacer click en tab). El ejemplo 3.2 puede ser replanteado de la siguiente manera:

```
n1 = float (input ("Ingrese la nota de la prueba 1: "))
n2 = float (input ("Ingrese la nota de la prueba 2: "))
n3 = float (input ("Ingrese la nota de la prueba 3: "))
ex = float (input ("Ingrese la nota del examen : "))
prom = 0.2*n1 + 0.2*n2 + 0.2*n3 + 0.4*ex
if prom >=4.0:
  print("Felicitaciones, aprobaste el curso !")
# Esto imprime "Felicitaciones, aprobaste el curso!" si la variable prom es mayor o igual a 4.0
```

Cabe destacar que se considera "dentro de un if" todo lo que esté indentado. Puede haber un *if* dentro de otro *if*, y las instrucciones de este deben estar indentadas según corresponda.

Ejemplo 2.3 Cree un programa que dado un número, determine si es par e imprimir en consola si lo es. Además si es par, elévelo al cuadrado. Al término del programa, despídase con un. "Adios mundo!" independiente de si el número era par o impar.

Es importante saber que en esta oportunidad, si se cumple la condición de ser par se deben ejecutar dos instrucciones:

- Imprimir en consola que es par.
- Elevarlo al cuadrado.

Luego de esto, hay que escribir un "Adios mundo!" fuera del *if*, ya que esto debe ser independiente de la condición de paridad.

```
a = int(input("Ingrese el num: "))
var = a%2 == 0
if var: #Es posible poner una variable tipo bool
  print("Es par")
  print(a**2) # Estas lineas se ejecutan solo si a es par
print ("Adios mundo!") # Notemos que no esta indentado
# Independiente de la paridad de a se ejecutara la linea
```

Ahora bien, nos interesará también ejecutar acciones cuando no se cumpla la condición pero si se cumpla otra. Para esto existe la entidad elif, la que viene después de un if. Si la condición del if no se cumple, pero si se cumple

la condición del elif, se ejecutará el grupo de instrucciones que está indentadas en el elif. Después de la condición deben ir los dos puntos (:) y además el elif no debe estar indentado respecto al if original. Es posible además no poner uno, sino que varios elif.
También nos interesa tener la entidad else, que viene después de un if o elif. Se ejecuta si ninguna de las condiciones anteriores fue ejecutada y no recibe una condición (variable tipo *bool*) pero si debe ir acompañada de dos puntos (:). Debemos tener en cuenta que:

- Un *if* puede ir sin acompañarse de un *elif* o un *else*.
- Un *else* o un *elif* debe estar precedido por un *if* o un *elif*.
- Es posible anillar estas entidades, unas dentro de otras.

Ejemplo 3.4 Diseñee un programa simple que decida cargar el celular si le queda poca batería.

```
#Es llevar a Python la idea del ejemplo 2.1
bat = input ("Bateria baja?\n1) Si\n2)No\n") #\n es un salto de linea
if bat == 1:
  print ("Cargar celular")
else :
  print ("No cargar ")
```

Ejemplo 3.5 Diseñe un programa que reciba dos números a y b y una operación aritmética (representada por un número) entre las cuatro básicas. Si la operación no es correcta decir que no fue posible realizarla. Imprimir en consola los resultados.

```
a = float(input("Ingrese el primer numero: "))
b = float(input("Ingrese el segundo numero: "))
op = input("Ingrese la operacion :\ n1) Suma \n2) Resta \n3) Mult \n4) Div\n")
if op =="1":
  s = a+b
  print ("La suma es: "+str(s))
elif op =="2":
  r = a-b
  print ("La resta es: "+str(r))
elif op =="3":
  m = a*b
  print ("La multiplicacion es: "+str(m))
elif op =="4":
  d = a/b
  print ("La division es: "+str(d))
else:
  print("No fue posible realizar operacion")
  print(str(a)+" ? "+str(b))
print("Adios!")
```

Antiejemplo 3.1 Pedir 3 números que representan cada uno lados de un triángulo. Ver si el triángulo es equilátero, isósceles o escaleno.

Un error muy frecuente es escribir el siguiente código:

```
a = float (input("Ingrese el primer lado: "))
b = float (input("Ingrese el segundo lado: "))
```

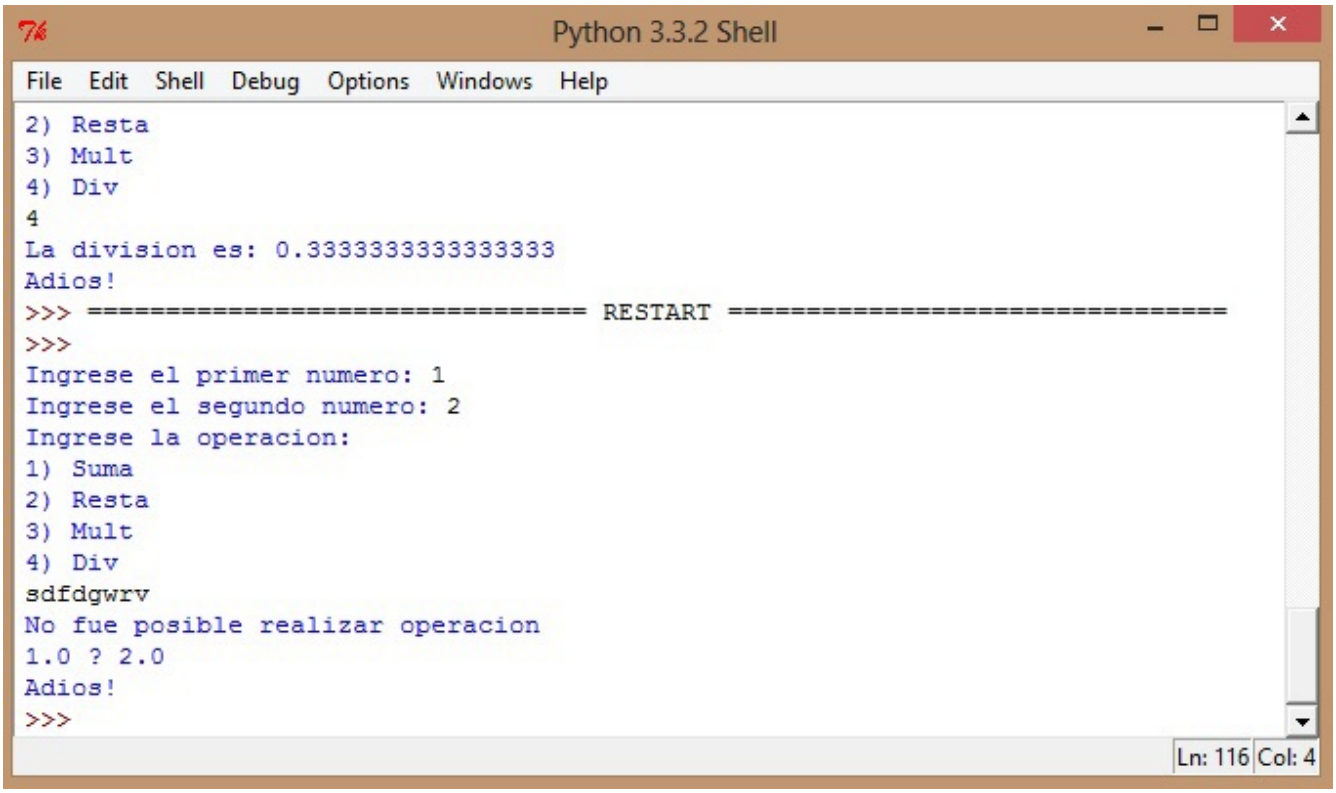

Figura 3.2: ejemplo 3.5, la calculadora.

```
c = float (input("Ingrese el tercer lado: "))
if a==b and b==c:
  print ("Equilatero")
if a==b or b==c or a==c:
  print ("Isosceles")
else :
  print("Escaleno")
```

Lamentablemente este programa no es correcto, ya que si nuestro input es el de un triángulo equilátero, como por ejemplo a=1, b=1 y c=1, el programa imprime en pantalla equilátero e isósceles, lo que no es correcto. Esto sucede porque si se cumple la primera condición, no se excluye de la ejecución al segundo *if*, ya que eso lo haría un *elif* (recordemos que el *elif* se ejecuta solamente si no se cumplió la condición anterior, en cambio un if lo haría siempre). Una solución correcta sería:

```
a = float (input("Ingrese el primer lado: "))
b = float (input("Ingrese el segundo lado: "))
c = float (input("Ingrese el tercer lado: "))
if a==b and b==c:
  print ("Equilatero")
elif a==b or b==c or a==c: #En vez de if pusimos elif
  print ("Isosceles")
else :
  print("Escaleno")
```

Ejemplo 3.6 Pida al usuario 3 números que representan los coeficientes de una ecuación cuadrática a, b, c. Imprima en pantalla:

- Que no es cuadrática si $a = 0$
- Que tiene dos soluciones reales si $b^2 - 4ac \geq 0$.
- Que tiene una solución real si $b^2 - 4ac = 0$.
- Que no tiene soluciones reales si $b^2 - 4ac \leq 0$.

Al realizar este problema debemos comprobar si es o no es cuadrática. En el caso de que lo sea, debemos hacer un análisis de sus posibles soluciones:

```
a = float (input ("Ingrese el a de la ecuacion: "))
b = float (input ("Ingrese el b de la ecuacion: "))
c = float (input ("Ingrese el c de la ecuacion: "))
if a !=0:
  if b**2 - 4*a*c >= 0:
    print ("Tiene dos soluciones reales")
  elif b **2 - 4*a*c == 0:
    print ("Tiene una solucion real")
  elif b **2 - 4*a*c <= 0:
    print ("Tiene soluciones complejas")
  else :
    print ("No es cuadratica")
```

3.5. Loops

Al momento de resolver un problema, nos intereserá ejecutar acciones mientras se cumpla una condición. En Python podemos realizar esto con la entidad *while* y *for*, las que ejecutarán las instrucciones que tengan indentadas una y otra vez mientras se sigan cumpliendo determinadas condiciones.

3.5.1. While

La entidad *while* repite las instrucciones indentadas mientras una expresión booleana sea *True*. Es posible poner también una variable de tipo *bool* dentro.

```
while <Condicion y/o variable booleana>:
  #Instrucciones indentadas que seran ejecutadas
```

Es importante notar que podemos poner un while dentro de otro while respetando la indentación, y también es posible poner las entidades condicionales *if*, *elif*, *else*. Al momento de usar un *while* en un programa se debe ser cuidadoso, puesto que si la expresión booleana nunca llega a ser falsa el programa puede quedar en un loop infinito y no terminar.

Ejemplo 3.7 Hacer un programa que imprima los números del 1 al 10 en consola.

Para realizar este programa, debemos tener un variable (que llamaremos contador) que diga el número de iteración en la que está el loop y se imprima en pantalla según corresponda. Cuando el contador supere el 10, saldrá del loop.

```
count = 1
while count <= 10:
  print(count)
  count = count + 1 #count aumenta en cada iteracion
#Cuando count sea 11, no volvera a entrar en el loop y se termina el programa
```

Obs: otra opción válida para hacer esto es lo siguiente:

```
count = 1
seguir = True
while seguir:
  print (count)
  count = count +1
  if count == 11:
    seguir = False
```

Ejemplo 3.8 Pedir al usuario un natural n e imprimir en pantalla el factorial de dicho número.

Para realizar este programa, además de llevar un contador de iteración, debemos tener una variable que lleve el valor del factorial:

```
n = int(input("Ingrese el numero n: "))
count = 1
mult = 1
while count <=n:
  mult = mult*count
  count = count + 1
print ( mult )
```

3.5.2. For

Un *for* es una entidad que recorre de manera determinada los elementos de una lista y ejecuta las instrucciones que tenga indentadas. **Si bien el capítulo de listas es posterior, y en él aprenderemos a usar bien la entidad for**, es posible entender lo básico para usar *for* como un loop. Posee la siguiente sintaxis:

```
for <variable local> in <lista>:
  #Grupo de instrucciones a ejecutar
  #Pueden usar o no a la variable local
```

Es necesario antes conocer antes la función *range*. *range* representa una secuencia de números y es usada frecuentemente para inicar loops en donde se conoce el número de iteraciones mediante un for[2]. Se inicializa de la siguiente manera:

```
a = range(<numero inicial>,<numero final>)
b = range(<numero inicial>,<numero final>,<intervalo>)
```

Donde número inicial es el nímero en que parte la secuencia, número final es el sucesor del último número de la secuencia, y si se inicializa con *intervalo*, el intervalo entre número y número es ese (Si no está inicializado con *intervalo*, avanza de 1 en 1). Por ejemplo:

[2]Esta definición fue extraída de la documentación oficial de Python

- *range*(1,10) tiene los números 1, 2, 3, 4, 5, 6, 7, 8, 9.
- *range*(1,10,2) tiene los números 1, 3, 5, 7, 9.
- *range*(1,10,3) tiene los números 1, 4, 7.

Así, el siguiente *for*:

```
for i in range (1 ,10):
  #Acciones indentadas
```

hace nueve iteraciones, una para cada valor que toma i. Empezará en 1, luego valdrá 2, e incrementará hasta llegar a 9.

Ejemplo 3.9.1 Imprimir en pantalla los números pares entre 1 y 10 excluyéndolos.

```
for i in range (2 ,10 ,2):
  print (i)
```

Ejemplo 3.9.2 Imprimir en pantalla los números impares entre 1 y 10 sin excluirlos.

```
a= range (1 ,10 ,2)
for i in a: #Notemos que inicializamos el range antes de usarlo en el for
  print (i)
```

Ejemplo 3.10 Pedir un número natural n e imprimir en pantalla los numeros primos entre 1 y n.

Recordemos que un número es primo si solamente es divisible por 1 y por si mismo. Para determinar si un número n es primo podemos dividirlo desde 2 hasta $n-1$. Podemos llevar una variable booleana (será *True* si el número es primo) que por defecto sea *True* y que cuando el resto de la división entre n y algún número i entre 2 y $n-1$ sea 0 (*i.e.* n es divisible en i) se haga *False*. Podemos llevar un loop que seleccione de 1 en 1 el número n a comprobar si es primo, y dentro de este loop, otro que vaya dividiendo por todos los x entre 2 y $n-1$. Cuando salga del loop interior, si la variable booleana sigue siendo verdadera, imprimirá el número. Luego debe continuar con el loop extrerior para comprobar todos los números restantes, luego de reseteada la variable *bool* a *True*.

```
n = int(input("Ingrese el numero n: "))
esPrimo = True
for i in range (2,n +1):
  for x in range (2,i):
    if i %x ==0:
      esPrimo = False
      #Notar que la linea a continuacion esta afuera del loop interno , pero dentro del exterior
  if esPrimo :
    print (i)
  esPrimo = True #Debemos resetear la esPrimo antes de volver al loop exterior
```

3.5.3. Break y continue

A veces no deseamos declarar variablas tipo bool adicionales para continuar o parar con un loop. Para esto existen dos entidades llamadas *break* y *continue*. Al ejecutar un *break* dentro del loop, este terminará, y no ejecuta ninguna de las instrucciones dentro del loop que estaban debajo del break. En cambio *continue* hace que el loop pase a la siguiente iteración saltandose todas las lineas dentro del loop que estuviesen después del *continue*.

Ejemplo 3.11 Escriba el programa que imprime los primeros 10 naturales en pantalla usando *break* y *continue*.

```
count = 1
while True:
  print(count)
  count += 1 #Esto es lo mismo que escribir count = count +1
  if count <=10:
    continue
  else:
    break
```

Notemos que con buen manejo de variables de tipo *bool* es posible realizar lo mismo que haria un *break* o un *continue*. Queda a criterio del programador la víaa que utilizará cuando necesite usar loops para resolver un problema.

Ejemplo 3.12 Diseñe un programa que reciba dos números a y b y una operación aritmética entre las cuatro básicas representada como un número. Si la operación no es correcta decir que no fue posible realizarla. Imprimir en consola los resultados. Luego de esto, preguntarle al usuario si desea realizar otra operación. Si responde "si", ejecutar nuevamente el programa.

Nuevamente es un ejemplo conocido, pero esta vez debemos añadir un loop que nos mantenga en el programa hasta que deseemos salir. Hay varias formas de hacer esto, pero la propuesta es la siguiente:

```
s="si"
while True :
  if s!="si":
    break
  a = float (input("Ingrese el primer numero a: "))
  b = float ( input (" Ingrese el segundo numero b: "))
  op = input ("Ingrese la operacion:\ n1) Suma \n2) Resta \n3) Mul\n4) Div\n")
  if op =="1":
    s = a+b
    print ("La suma es: "+str(s))
  elif op =="2":
    r = a-b
    print ("La resta es: "+str(r))
  elif op =="3":
    m = a*b
    print ("La mult es: "+str(m))
  elif op =="4":
    d = a/b
    print ("La div es: "+str(d))
  else :
    print ("Ingrese operacion valida")
  s = input ("Desea seguir?")
```

Capítulo 4

Funciones y módulos

4.1. Funciones

Ya habíamos hablado anteriormente de funciones como entidades que reciben parámetros y realizan un conjunto de acciones. Sin embargo debemos ahondar un poco más. Una función es una sección apartada del codigo que ejecuta ciertas acciones en base a parámetros entregados. Luego puede retornar un valor o no retornar nada. Una función por si sola no hace nada, y la idea es utilizarla en el programa al llamarla. Una ventaja muy importante que tienen las funciones es que si queremos reutilizar el código de una parte del programa, cambiando sólo ciertos parámetros, es posible crear una función, y la llamamos las veces que necesitemos cambiando los parámetros de entrada. La sintaxis es como se sigue:

```
def <nombre funcion> (parametro1 , parametro2 , ... , parametroN):
  # Aqui el codigo indentado
  #...
return <valor de salida> # Puede o no tener return
```

Luego basta con llamar a la función con su nombre en el código. Cabe destacar que para usar una función debe ser declarada antes del código en el que se va a utilizar, aunque si otra función usa una función, esto no necesariamente se debe cumplir. Repitamos un ejemplo hecho anteriormente, pero ahora aplicando funciones.

Ejemplo 4.1 Pedir un número natural n e imprimir en pantalla los números primos entre 1 y n.

Se repite el concepto hecho en el capítulo anterior, pero esta vez para determinar si un número es primo se aplicará una función:

```
def esPrimo (num):
#num es el parametro de entrada que representa el numero que queremos saber si es primo
#Es una variable local y no influye fuera de la funcion
  primo = True
  for i in range(2, num):
    if num %i ==0:
    primo = False
  return primo #Retorna True si un numero num es primo

n = int(input("Ingrese un numero: "))
for i in range(2,n):
  if esPrimo(i): # Llamamos a la funcion en cada iteracion
  print (i)
```

Ejemplo 4.2 Cree dos funciones que dado el lado de un cuadrado, una retorne su perímetro, otra su área, y otra el valor de su diagonal.

```
def perimetro (lado):
  return 4*lado
def area (lado):
  return lado**2
def diagonal (lado):
  return lado *(2**0.5)
#Ahora las usaremos en un programa
l = float(input("Ingrese el lado: "))
a = perimetro (l)
b = area (l)
print (a)
print (b)
print (diagonal (l)) #Aqui no asignamos diagonal (l) a una variable
```

Cabe destacar que una función retorna un valor si y sólo si tiene **return**. Si esta finaliza con un print y no posee return, esta función no se puede asignar a una variable:

```
def unaFunc(s):
  print(s+" algun string")

def otraFunc(s):
  a = s+" algun string"
  return a

#valor = unaFunc("hola") Esto tira error!!
valor = otraFunc("hola") Esto es posible
```

Es necesario recalcar que además al llegar al return no se sigue ejecutando ningún otro código dentro de la función aunque este exista código más abajo.

4.2. Módulos

Un módulo en Python es una colección de funciones y constantes que pueden ser importadas desde cualquier otro programa. En general, cuando creamos un programa, y lo guardamos con extensión *.py*, estamos creando un módulo. Python trae varios implementados. Entre los más destacados están random y math. Para usar un módulo implementado por Python se debe escribir lo siguiente:

```
from <Nombre del modulo> import <Nombre de lo que deseamos importar>
```

Obs. También es posible:

```
import <Nombre del modulo>
#Luego para usar la funcion usamos <Nombre del modulo>.<Nombre de la funcion>
```

Si el módulo que queremos usar es uno de los que guardamos nosotros se debe hacer de la misma manera, pero el programa en el que convocamos al módulo debe estar guardado en la misma carpeta que el módulo que estamos importando. Podemos importar de él las funciones definidas. Luego podemos usar la entidad importada. Queda a criterio del lector buscar los módulos que necesite y averiguar las funciones pertinentes. Veamos un ejemplo:

Ejemplo 4.3 Usando el módulo math, dado cierto radio, calcular el área y el perímetro de un círculo.

Se realizará con funciones. Usaremos la constante *Pi* implementada en el módulo *math*:

```
from math import pi
def perimetroCirc(radio):
  return 2*pi*radio
def areaCirc(radio):
  return pi*(radio**2)

r = float(input("Ingrese el radio: "))
print(perimetroCirc(r))
print(areaCirc(r))
```

En síntesis, cuando tenemos un programa de Python guardado en un.py, podemos usar funciones que estén en otros archivos .py si **importo** la función. Esto es análogo a dividir nuestro programa en varias hojas, y cuando importo funciones, las pego en la hoja que las está, logrando así un programa más grande. Además en general, cuando tengamos proyectos grandes, es aconsejable dividirlo en varios módulos, para que su posterior edición sea más sencilla.

4.3. Main

Cuando nosotros tenemos un módulo que aparte de funciones tiene código que se ejecuta como en el ejemplo 4.3, si llegamos a importar ese módulo, aunque no lo queramos, en nuestro programa se ejecutará el código que no fue declarado dentro de una función.
Supongamos que tenemos un módulo con el código del ejemplo 4.3 llamado Circ, y además tenemos otro programa en que importamos la función perimetroCirc. El resultado sería el siguiente:

Antes de ejecutar el codigo de nuestro programa (un simple "hello world") se ejecuta el código del módulo, en que pedía un radio e imprimía en consola el perímetro y el área del círculo asociado. Para esto, debemos agregar un condicional en el módulo Circ que incluye todo el código "ejecutable" (es decir, el código no definido como función). En general, cuando tengamos un módulo, y queramos mantener código ejecutable, pero queremos que se ejecute sólo si lo estamos abriendo desde el mismo módulo, este debe ir indentado en el siguiente if:

```
if __name__ == "main":
  #Codigo ejecutable
```

Así, el módulo Circ quedaría:

```
from math import pi
def perimetroCirc(radio):
  return 2*pi*radio
def areaCirc(radio):
  return pi*(radio**2)
if __name__ == "main":
  r = float(input("Ingrese el radio: "))
  print (perimetroCirc(r))
  print (areaCirc(r))
```

Y si importamos en algón otro programa el módulo Circ.py, no se ejecutará aquel código (el programa de la figura 3.1 imprimiría sencillamente un "hello world").

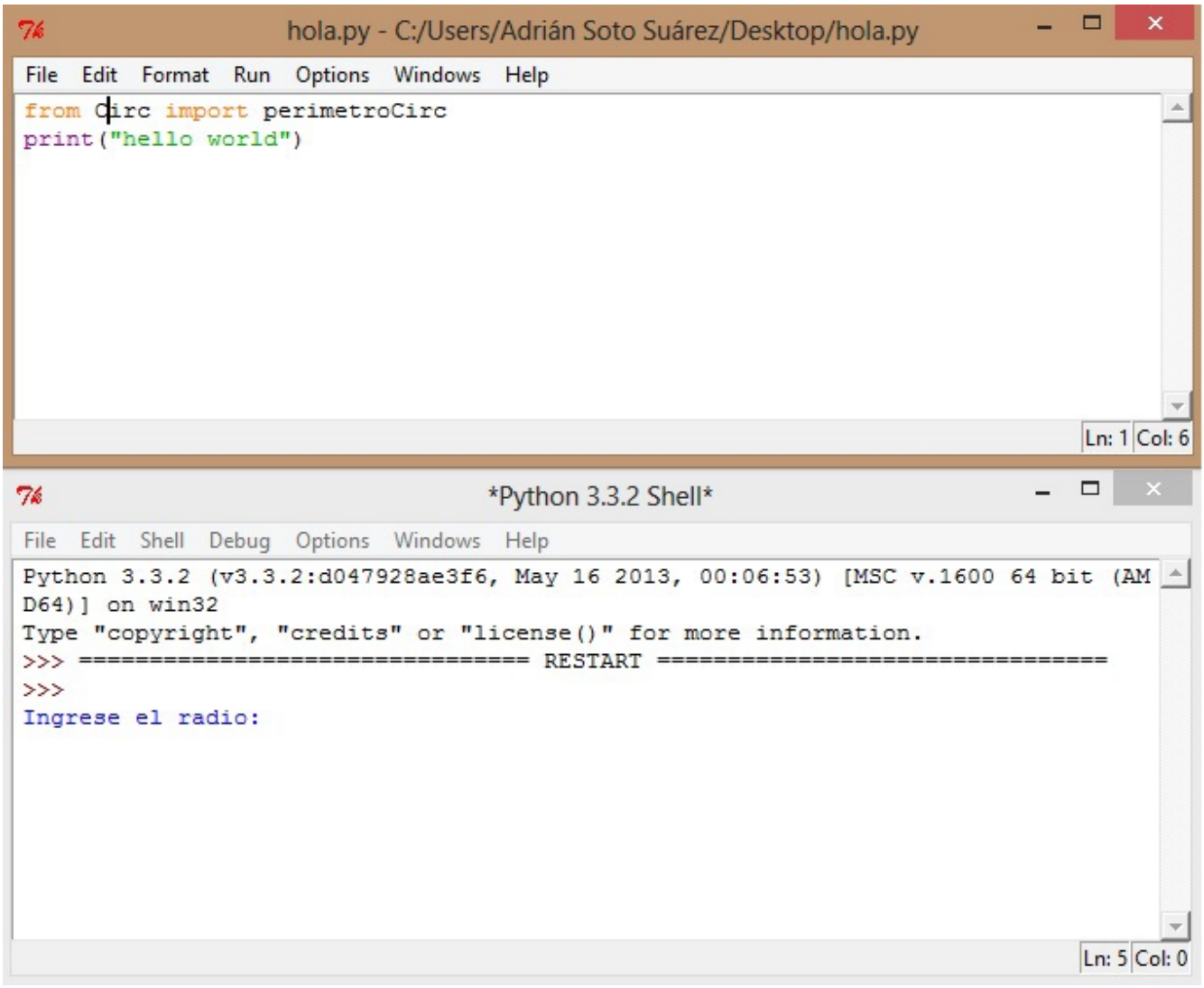

Figura 4.1: problemas con módulos.

Obs. Es una buena práctica de programación no escribir código ejecutable en un módulo si sólo queremos definir en él funciones. Si deseamos hacer un programa relativamente grande, el ejecutable debe ir en un módulo separado que importe todos los demás módulos en los que están declaradas las funciones por usar.

Capítulo 5

Strings

5.1. Declaración y manejo de strings

Hasta ahora hemos aplicado conceptos sobre todo aritméticos y matemáticos simples. Pero ¿Qué hay acerca del procesamiento de texto en Python?. Ya hemos introducido un tipo llamado string que correspone a la representación de una cadena de caracteres. Este capítulo busca ilustrar parte de las funciones más importante a la hora de trabajar con strings.

Una variable se declara de tipo string cuando se le asigna texto entre " " o bien se le asigna el retorno de una función de tipo string. Por ejemplo:

```
s = "Hola, soy un string" #Declaracion de un string

s2 = input("Ingrese su texto") #Recordemos que input guarda como
#string el texto ingresado por el usuario.
```

Para el manejo de string es posible usar utilidades que ya vienen implementadas en python:

- Para saber la cantidad de caracteres de un string usamos la función len:

  ```
  s = "mensaje"
  a = len(s) #Esta variable tiene el valor 7
  ```

- Para acceder al i-ésimo caracter de un string es posible hacer lo siguiente:

  ```
  s = "Hola, soy un string"
  a = s[0] #Esta variable tiene el valor "H"
  b = s[1] #Esta variable tiene el valor "o"
  ```

  Notemos que el carater en la primera posición es el 0 y que la última posición es la len(s)−1.

- Para saber si un caracter o un string forma parte de un string:

  ```
  s = "Hola, soy un string"
  var = "a" in s #Variable booleana que toma el valor True
  #Pues el caracter a esta en el string.
  ```

```
var2 = "Hola" in s #Variable booleana que toma el valor True
var3 = "Chao" in s #Variable booleana de valor False
```

- Para saber el orden según la tabla ascii de dos strings podemos usar $<$ o $>$. Un string es "mayor que otro" si el orden alfabético **según la tabla ASCII**[1] es posterior. Notar que si se desea el orden alfabético se desea hacer normalmente se debe hacer convirtiendo todo a minúscula o mayúscula.

```
s = "Alameda"
s2 = "Providencia"
a = s<s2
b = s>s2
#a toma el valor True
#b toma el valor False
```

  variable será **True** si *s1* está alfabéticamente antes que *s2* de lo contrario será **False**

- Para extraer parte de un string y almacenarlo (en otros lenguajes esto se hace con una función llamada **substring**) debemos hacer *string*[x:y] que toma el string desde la posición x hasta la y-1. Por ejemplo:

```
s = "Soy un string"
a = s[0:5]
#a tiene el valor "Soy u"

b = s[3:7]
#b tiene el valor " un "

c = s[:7]
#c tiene el valor "Soy un "

d = s[5:]
#d tiene el valor "n string"

e = s[1:-2]
#e tiene el valor "oy un stri"
```

  Notamos que c, d son especiales. Las dos primeras porque carecen de un término. Python asume que el usuario intentó decir que el termino faltante era 0 y len(s) respectivamente. En cuanto a e, que tiene un número negativo, esto se toma como la posición desde derecha a izquierda.

5.2. Funciones de strings

Existen en Python ciertas funciones que nos permiten realizar varias funciones con los datos de tipo string. Estas funciones requieren una instancia de un string para ser usadas de la forma:

```
string.funcion(parametros)
```

Algunas de las funciones se presentan a continuación. Se propone al lector que averigue en la documentación oficial de Python acerca de otras que logre necesitar:

[1] Esta tabla se explica en secciones posteriores.

- **replace**(string1,string2): Reemplaza en un string el valor string1 si está contenido por el valor de string2.

```
s = "hola hola chao hola"
s.replace("chao","hola")
#Ahora el valor de s es "hola hola hola hola"
```

- **isalpha**(): Función que retorna True si el string contiene sólo caracteres alfabéticos:

```
s = "hola "
a = s.isalpha()
#a posee el valor True

s="123"
a = s.isalpha()
#a posee el valor False
```

- **count**(string1): Función que retorna un int que cuenta cuantas veces string1 aparece dentro del string.

```
s = "hola hola chao hola"
a = s.count("a")
b = s.count("hola")
c = s.count("chao")
# a es 4
# b es 4
# c es 0
```

- **find**(string1): Encuentra la primera aparición en el string de string1. Si no existe retorna -1.

```
s = "hola"
a = s.find("a")
s2 = "babaroa"
b = s2.find("b")
c = s2.find("ro")
# a es 3
# b es 0
# c es 4
```

- **upper**(): Retorna un string con todos los caracteres alfabéticos en mayúscula.

```
s = "hola"
a = s.upper()
# a es "HOLA"
```

- **lower**(): Retorna un string con todos los caracteres alfabéticos en minúscula.

```
s = "Hola"
a = s.lower()
# a es "hola"
```

- **strip**(): Elimina los espacios de los lados.

```
s = " Hola tengo espacios "
a = s.strip()
# a es "Hola tengo espacios"
#Si, es mentira, porque a no tiene espacios a los lados :(
```

5.3. El tipo chr y la tabla ASCII

En Python no existe el tipo *chr* por si solo, sin embargo es posible trabajar con strings de largo 1, es decir, podemos representar caracteres. Un caracter es la representación de 8-bits[2] asociado a una tabla llamada ASCII. Esta tabla para cada número entre 0 y el 255 posee una representación en forma de caracter.
En Python para obtener el número asociado a un caracter usamos la función **ord**(char). En cambio, para obtener el caracter asociado a número usamos la función **chr**(int). Por ejemplo:

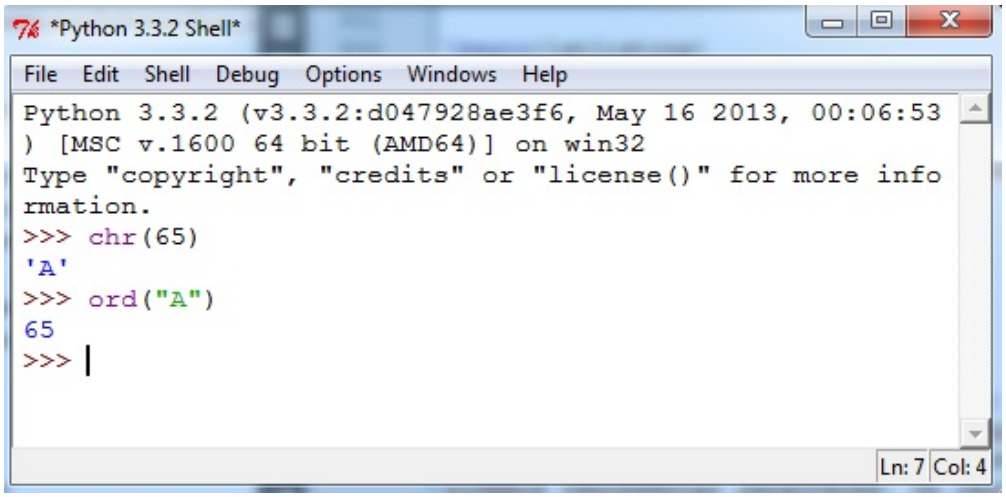

Figura 5.1: ejemplo de **ord**() y **chr**().

Es necesario notar que en la tabla ASCII:

- Las letras en mayúscula de la **A** a la **Z** están entre los números 65 y 90.
- Las letras en minúscula de la **a** a la **z** están entre los números 97 y 122.

Ejemplo 5.1 Reciba un string por parte del usuario y compruebe si este posee sólo caracteres alfabéticos. Luego, si se cumple lo anterior, preguntele al usuario si desea pasar todo el texto a mayúscula o minúscula.

Para realizar el ejercicio utilizaremos las funciones que detallamos anteriormente:

```
s = input("Ingrese su texto: ")
if s.isalpha():
  s2 = input("Que desea hacer?\n1) Mayusculas\n2) Minusculas\n")
  if s2 == "1":
    s = s.upper()
    print(s)
  elif s2 == "2":
```

[2] 8 bits es un número de 8 dígitos en binario, que forma la unidad byte

```
s = s.lower()
print(s)
```

Es necesario notar que debemos asignarle a s el valor de la manera **s = s.lower()** o bien **s = s.upper()** ya que estas funciones no cambian el valor de s sino que retornan un nuevo elemento con el valor correspondiente a cambiar todas a mayúsculas o minúsculas. Para profundizar en el manejo de strings es necesario pasar al siguiente capítulo correspondiente a listas.

Capítulo 6

Listas

6.1. Listas Unidimensionales

Las listas corresponden a un tipo de estructura de datos que son capaces de almacenar variables instanciadas. Informalmente, podemos ver a los arreglos como casilleros en los que guardamos valores, y cada vez que agregamos un valor a la lista se agrega un casillero con una información correspondiente dentro. Su declaración más básica es de la forma:

```
miLista = []
```

En donde miLista corresponde a una lista unidimensional capaz de almacenar datos. Su función fundamental es **append(valor)**, que adhiere al final de la lista a *valor*. Por ejemplo:

```
miLista = []
miLista.append(1)
miLista.append(2)
miLista.append(3)
miLista.append(4)
miLista.append(5)
```

Agregamos a una lista vacía los números enteros 1, 2, 3, 4, 5. Si ahora imprimimos la lista:

```
miLista = []
miLista.append(1)
miLista.append(2)
miLista.append(3)
miLista.append(4)
miLista.append(5)

print(miLista)
```

El resultado se muestra en la figura 5.1:

Para conocer el largo (número de elementos que hay almacenados en la lista) al igual que en strings ocupamos la función len(lista).

```
miLista = []
miLista.append(1)
```

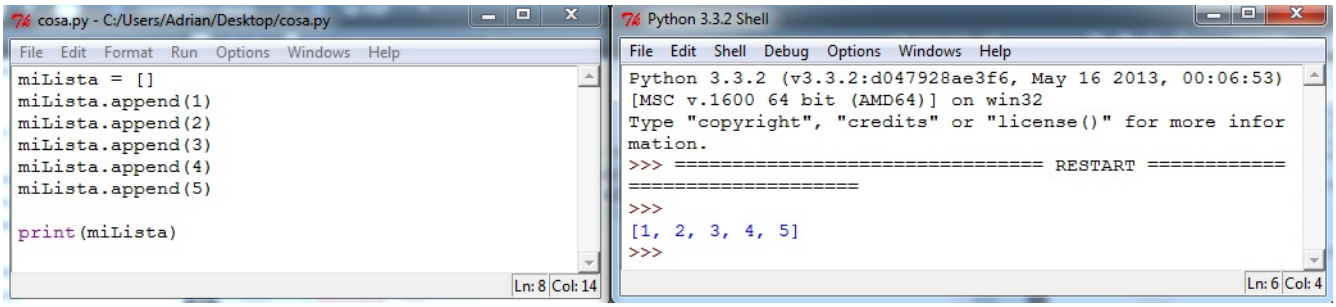

Figura 6.1: adherir elementos a una lista.

```
miLista.append(2)
miLista.append(3)
miLista.append(4)
miLista.append(5)

a = len(lista)

#El valor de a es 5 pues tiene 5 elementos almacenados
```

Una vez inicializada la lista podemos acceder al i-ésimo elemento de la misma manera que a un string, es decir **lista[i]**:

```
miLista = []
miLista.append(1)
miLista.append(2)
miLista.append(3)
miLista.append(4)
miLista.append(5)

print(miLista[0])
print(miLista[3])
```

Donde cabe destacar que el primer elemento está en la posición 0 y el último en la posición $len(lista) - 1$. El resultado de lo anterior es el siguiente:

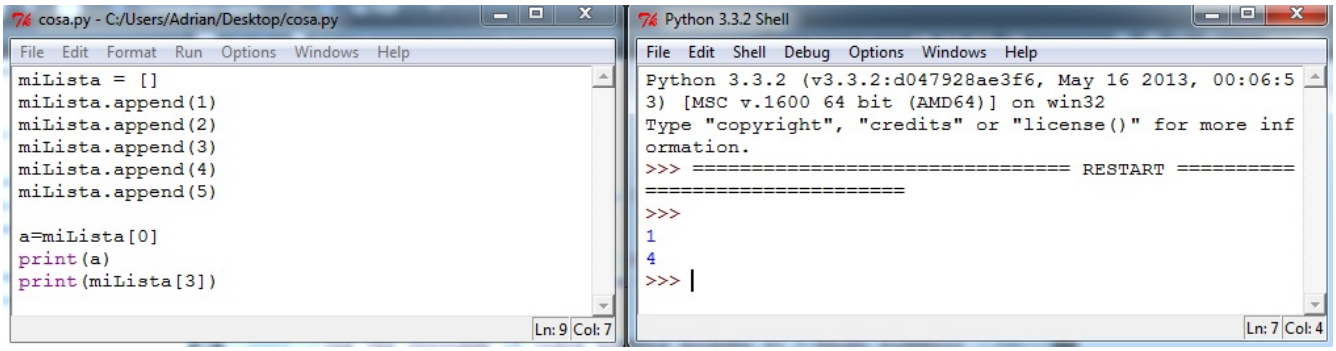

Figura 6.2: acceder al i-ésimo elemento.

A diferencia de otros lenguajes, es posible adjuntar a una lista valores de distintos tipos. Por ejemplo en una lista podemos guardar valores float, int, string entre los otros que conocemos. Cabe destacar que la lista **es una estructura dinámica** ya que su largo no es fijo. Bajo este contexto no se debe confundir a la lista con otra estructura de datos llamada **arreglo**, la que tiene un comportamiento similiar, pero su largo (cantidad de "casilleros" para almacenar información) es fijo después de su creación, sin embargo, es posible modificar lo que tenga el término i-ésimo del arreglo.
En Python existe además otro tipo llamado **tupla**, el que consta de información que no necesariamente es del mismo tipo, pero tiene un largo fijo. Además no es posible modificar los valores de la tupla luego de que esta fue creada. Queda a criterio del lector buscar información sobre estas dos estructuras de datos mencionadas.

Ejemplo 6.1 Cree una bitácora en Python en el que sea posible pedir al usuario la fecha de un suceso y un string que diga el suceso que desea guardar.

Para esto haremos un menú para ver si el usuario quiere agregar una nota o verlas todas.

```
bitacora = []
op = 0
while op != "3":
    op = input("Que desea hacer\n1) Agregar nota\n2) Leer Notas\n3) Salir\n")
    if op == "1":
        #Se supone que el usuario ingresa bien estos datos
        year = input("Year?: ")
        month = input("Month?: ")
        day = input("Day?: ")
        suceso = input("Ingrese el suceso: ")
        s = year+"/"+month+"/"+day+": "+suceso
        bitacora.append(s)
    if op == "2":
        for i in range(0,len(bitacora)):
            print(bitacora[i])
    elif op!="3" and op!="2" and op!="1":
        print("Ingreso equivocado, \intente de nuevo")
```

6.2. Utilidades y funciones para listas unidimensionales

Aparte de **len()** y **append()** existen otras funciones y utilidades que hay que tener presentes sobre listas:

- Para obtener parte de una lista debemos usar lista[x:y] de la misma forma que en strings. Esto **no modifica** la lista original.

  ```
  l1=[1,2,3,4,5,6,7,8,9,10]
  l2=l1[5:]
  #l2 tiene los valores: [6,7,8,9,10]
  ```

- Para transformar un string a una lista de caracteres podemos usar **list()**:

  ```
  s="hola"
  lista = list(s)
  #lista tiene los valores: ["h","o","l","a"]
  ```

  Esto último también funciona para pasar un valor de tipo *range()* a una lista de números. Esto se muestra en

la figura 5.3.

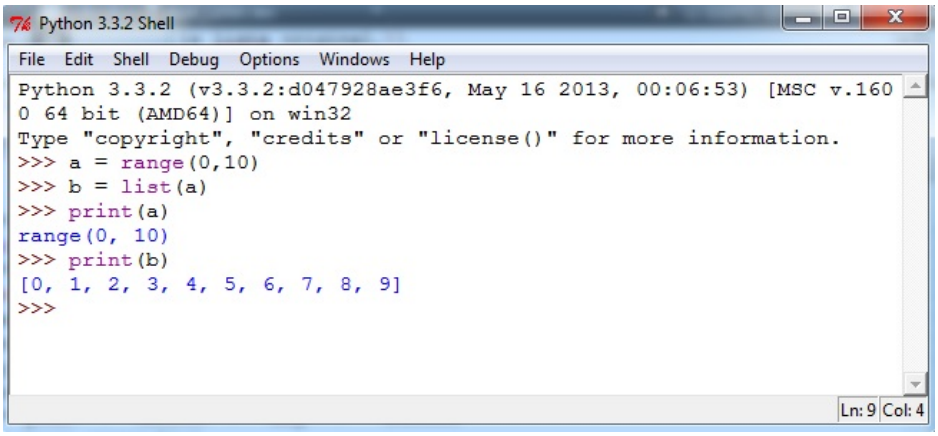

Figura 6.3: range vs lista.

- **lista.remove(elemento)**: Elimina la primera aparación de "elemento" en la lista. Si no existe se nos lanza un error.

```
s="hola"
lista = list(s)
lista.remove("h")
#lista ahora tiene los valores: ["o","l","a"]
```

- Si queremos borrar el i-ésimo elemento de la lista debemos usar **del lista[i]**:

```
a = [1,1,2,3,4,5,2,10]
del a[len(a)-2]
#Ahora a es [1, 1, 2, 3, 4, 5, 10]
```

- Si tenemos dos listas l1 y l2, unimos las listas con el operador +. Un ejemplo es:

```
a = [1,2,3]
b = [4,5,6]
c = a+b
#c es [1, 2, 3, 4, 5, 6]
```

- **lista.index(elemento)**: retorna el entero en el que está la primera aparición de "elemento" en la lista. Si "elemento" no existe en la lista obtendremos error.

```
a = "holah"
b = a.index("h")
#b es 0
```

- **lista.count(elemento)**: retorna la cantidad de veces que "elemento" se encuentra en la lista.

```
a = [1,1,2,2,2,3,4,5,5,5]
b = a.count(1)
c = a.count(3)
d = a.count(5)
e = a.count(6)

#b es 2
#c es 1
#d es 3
#e es 0
```

- **string.split(string2)**: retorna una lista de string en la que que cada se borró todas las apariciones de string2, y se hizo una lista con cada elemento que no fue eliminado. El string original no se ve modificado.

```
s="Soy|un|string|con|rayas"
b = s.split("|")
#b es ['Soy', 'un', 'string', 'con', 'rayas']
s2="Yo no se que soy"
b2 = s2.split("no")
#b es ['Yo ', ' se que soy']
```

6.3. For

Ya habíamos usado el for anteriormente como una herramienta para realizar iteraciones usando "range". Ahora que conocemos las listas, podemos decir que este es un tipo "iterable", es decir, podemos realizar acciones para cada uno de los valores almacenados. Cuando hacemos:

```
for <variable local> in <lista>:
  #Grupo de instrucciones a ejecutar
  #Pueden usar o no a la variable local
```

Lo que hacemos es tomar uno a uno los valores almacenados en lista, le damos el nombre de "variable local" mientras esté en la iteración y ejecutamos acciones. Así:

```
s=["palabra1","palabra2",...,"palabran"]
for palabra in s:
  print(palabra)

#Este programa imprime desde palabra1 a palabran
```

Imprime todas las palabras de la lista s asignándoles momentaneamente el nombre *palabra* mientras pasen por la iteración. Notemos que *palabra* va tomando los valores desde "palabra1" hasta "palabran". Esto último es muy distinto a:

```
s=["palabra1","palabra2",...,"palabran"]
for i in range(0,len(s)):
  print(i)

#Este programa imprime los numeros desde 0 hasta n-1
#Notamos que son n palabras
```

Ya que aquí la variable local i toma los valores numéricos entre 0 y $n-1$. Si quisieramos imprimir las palabras usando range en el for deberíamos:

```
s=["palabra1","palabra2",...,"palabran"]
for i in range(0,len(s)):
  print(s[i])

#Este programa vuelve a imprimir las palabras
```

6.4. Ordenamiento

Python posee una función de ordenamiento llamada **sort()** que es capaz de ordenar alfabéticamente (**según la tabla ASCII**) una lista de strings y de menor a mayor un grupo de números. ¿Pero si queremos hacer otro tipo de ordenamiento?.
Existen diversos algoritmos que sirven para ordenar. A continuación se van a detallar dos de estos algoritmos esperando que el lector busque otros y mientras tanto descubra por si mismo como implementarlos en Python.

- **Bubble Sort**: Algoritmo de ordenamiento que se para en un término y ve el término siguiente. Si están en orden incorrecto los voltea. Es necesario recorrer desde el principio hasta el final una lista muchas veces para lograr una lista completamente ordenada.
- **Insertion Sort**: Se toma el primer elemento y se fija. Luego se escoje el menor de todos los demás y se úbica antes o después (dependiendo si es mayor o menor) del término original. Luego tenemos dos elementos ordenados. Ahora se vuelven a tomar los restantes, y se escoge el menor y se ordena respecto a los primeros dos (que teníamos fijos y ya estaban ordenados). En general tenemos elementos ordenados y otros que faltan por ordenar. Se toma el menor de los que falta por ordenar y se ubica en su posición correspondiente en los elementos ya ordenados.Se hace esto hasta obtener una lista completamente ordenada.

6.5. Listas bidimensionales

Las listas bidimensionales son listas de listas. Para inicializar una es posible crear una lista vacía e ingresar a ella más listas. Es posible hacer listas multidimensionales pero en este texto se detallaran hasta las listas bidimensionales:

```
s=[]
a=[1,2]
b=[3,4]
s.append(a)
s.append(b)

#Ahora s tiene los valores [[1,2],[3,4]]
```

Sea *listaBid* una lista bidimensional. Supongamos que esta se compone de n listas. Si quiero acceder al i-ésimo elemento de la j-ésima lista con $0 \leq j < n$ debo hacer lo siguiente:

```
listaBid[j][i]
```

Notamos que me muevo primero a la j-ésima lista, y en esta busco la i-ésima variable. Esta es posible almacenarla en una variable. Notemos que es posible interpretar estos arreglos como matrices, ya que cada arreglo interior puede

corresponder a una fila, y si todos estos arreglos interiores tienen el mismo largo, este largo corresponde al número de columna[1].

6.6. Valores por referencia

Supongamos que tenemos el siguiente código:

```
a = [1,2,3]
b = a
```

Luego si edito un valor en b, este **también se editará en a**, ya que al decir "$b = a$" son ahora la misma lista. Esto sucede porque las listas son tipos por referencia, es decir, no nos importan los valores, sino que sus direcciones en memoria. Hasta ahora habíamos trabajado sólo con tipos por valor. Este concepto es muy amplio y se detalla en otros cursos, por ende, ahora sólo debemos quedarnos con que el código anterior es problemático. Una solución es:

```
a = [1,2,3]
b = []
for i in a:
  b.append(i)
```

Esto también pasa cuando creo una lista bidimensional a partir de la misma lista:

```
a = [x,x,x]
b.append(a)
b.append(a)
b.append(a)

#Hasta ahora el valor de a es [[x,x,x],[x,x,x],[x,x,x]]
```

Si edito uno de los tres arreglos interiores, se editarán los tres:

```
a = [x,x,x]
b.append(a)
b.append(a)
b.append(a)
b[0][0] = "o"

#Hasta ahora el valor de a es [[o,x,x],[o,x,x],[o,x,x]]
```

Para solucionar este problema:

```
b=[]
for i in range(0,3):
  a=[]
  for j in range(0,3):
    a.append("x")
  b.append(a)
```

Esto último funciona ya que al inicializar en cada iteración el valor de a otra vez como vacío, no estoy agregando a b el mismo arreglo.

[1]Es posible que el usuario los quiera leer al revés, por lo que es instructivo mencionar que hay diversas formas de interpretar las listas bidimensionales y la que se muestra aquí es sólo una guía.

Ejemplo 6.2 Sea M una matriz de 2x2. Cree una función que pida como parámetro un arreglo bidimensional que represente a M y retorne su determinante.

Interpretamos la matriz bidimensional como [[a,b],[c,d]] donde el determinante es $ad - bc$.

```
def determinante(m):
  a1 = (m[0][0])*(m[1][1])
  a2 = (m[0][1])*(m[1][0])
  det = a1-a2
  return det

a = float(input("Ingrese a: "))
b = float(input("Ingrese b: "))
c = float(input("Ingrese c: "))
d = float(input("Ingrese d: "))
primeraFila = [a,b]
segundaFila = [c,d]
miMatriz = [primeraFila,segundaFila]
print(determinante(miMatriz))
```

Capítulo 7

Diccionarios

7.1. Funcionamiento de un diccionario

En computación, un diccionario es una estructura de datos que nos permite insertar, eliminar y buscar elementos en ella. No nos asegura que lo que estemos guardando se mantenga ordenado, pero si nos asegura que la busqueda de un valor sea extremadamente rápida.
Lo que se almacena en un diccionario es una llave junto a su valor. En el diccionario buscaremos un elemento por su llave y se nos retornará el valor. Por ejemplo yo puedo almacenar la tupla $(1, "miPalabra")$, es decir la llave 1, junto al valor "miPalabra". Si yo le pregunto al diccionario el valor asociado a la llave 1, este me contestará "miPalabra". Sin embargo si yo le pregunto por "miPalabra" el diccionario no sabrá contestarme.
El diccionario que veremos a continuación se conoce habitualmente como una tabla de hash. Si bien su construcción y funcionamiento se explican en cursos superiores, a grandes rasgos su funcionamiento es el siguiente:

- Cuando inicializo un diccionario, reservo un determinado número de buckets (casilleros) en mi memoria. Cada uno tiene un número de bucket asociado.
- Cuando quiero almacenar una llave ingreso su valor a una función $h(x)$ que me retorna un número entero, el que me indicará el número del bucket en que guardaré el valor.
- Si el bucket estaba ocupado, comienzo a generar una lista con todos los valores que se van acumulando. Mientras más valores acumule en un casillero, más lento se volverá mi diccionario.
- Cuando quiero buscar el elemento por su llave, calculo su función $h(x)$ que me dirá el casillero en el que está. Luego me muevo en la lista respectiva hasta encontrar el valor exacto. Luego lo retorno.
- Cuando quiero eliminar un elemento, realizo la misma acción que en el punto anterior, sólo que en vez de retornar el elemento, lo elimino.

En este capítulo no implementaremos nuestro propio diccionario, sino que aprenderemos a usar el que viene implementado por Pyrhon.

7.2. Uso del diccionario

En Python tenemos dos formas de instanciar un nuevo diccionario. La primera de ellas es usando la función **dict()**, que genera un diccionario vacío:

```
miDiccionario = dict()
#La variable miDiccionario tiene almacenado un nuevo diccionario.
```

La otra forma es declarar directamente el valor de las llaves con sus respectivos valores asignados:

```
miDiccionario = {1: 'James', 2: 'Kirk', 3:'Lars',4:'Cliff'}
```

Independiente de la forma en la que creemos nuestro diccionario[1], la forma de insertar, buscar y eliminar es la misma:

- Para insertar una llave k en nuestro diccionario asociado al valor *value* debemos hacer lo siguiente:

  ```
  miDiccionario[key] = value
  ```

- Para eliminar la llave k con su respectivo valor asociado, debemos hacer lo siguiente

  ```
  del miDiccionario[key]
  ```

- Para buscar un valor *value* por su respectiva llave k, debemos hacer lo siguiente:

  ```
  a = miDiccionario[k]
  #Ahora la variable a tiene asignado el valor de value.
  ```

- Si queremos obtener una lista con todas las llaves del diccionario la podemos obtener así:

  ```
  listaKeys = list(miDiccionario.keys())
  #Ahora en listaKeys tenemos una lista con todas las llaves.
  ```

- Por último, si queremos consultar si una determinada llave k forma parte del diccionario, lo podemos preguntar así:

  ```
  var = k in miDiccionario
  #Si k forma parte de miDiccionario, var toma el valor True.
  #Si k no forma parte de miDiccionario, val toma el valor False.
  ```

Ejemplo 7.1 Dada una lista de strings que contiene el número de alumno y el nombre de un grupo de alumnos separados con un — como por ejemplo:

```
#'123|Juan'
```

Cree una función que reciba dicha lista y retorne un diccionario diccionario que como llave reciba un número de alumno y su value sea el nombre del alumno. Luego para cada llave, imprima su value.

Primero definimos la función pedida:

```
def listaToDic(lista):
    diccRet = dict()
    for i in lista:
```

[1]Los ejemplos a continuación suponen que tenemos instanciado un diccionario llamado miDiccionario.

```
        s = i.split('|')
        a = int(s[0])
        b = s[1]
        diccRet[a]=b
    return diccRet
```

Luego, dada una lista llamada "lista" hacemos lo siguiente:

```
d = listaToDic(lista)
for i in list(d.keys()):
    print (d[i])
```

Capítulo 8

Clases

8.1. Programación orientada a objetos

La programación orientada a objetos es el paradigma de programación que se basa en la interacción de **objetos**. Un objeto es una entidad dentro de un programa que posee un comportamiento característico según su clase.
Cuando queremos modelar comportamientos del mundo real en un programa desearemos trabajar con objetos y no sólo con los tipos implementados en Python. Desearemos tener nuestros propios tipos con funciones específicas para darles un cierto comportamiento deseado.
Es necesario entonces poder instanciar objetos dentro de un programa de Python para modelar mejor los comportamientos de la realidad. Para esto necesitamos hacer uso de **Clases**. Las clases son la herramienta que la programación orientada a objetos (OOP) usa para trabajar.

8.2. Crear una clase

8.2.1. Declarar una clase

Crear una clase en Python es análogo a crear una planilla en la que se describe el comportamiento que tendrán los objetos provenientes de aquella clase. Para definir una clase debemos escribir lo siguiente:

```
class <Nombre de la clase>:
  <Aqui va el contenido de la clase indentado>
```

El contenido indentado corresponde a los **atributos** de la clase y las funciones características de la clase, que llamaremos **métodos**, dentro de los cuales está el **constructor**, que permite instanciar objetos de la clase creada.

8.2.2. Atributos de la clase

Cada clase debe contener ciertas variables que definen su comportamiento. Por ejemplo al crear una clase *Persona* esta debe poseer una variable tipo *string* que almacene su nombre, otra de tipo *int* que almacene su edad, una lista que almacene el banco y un número de cuenta, entre varias otras variables.
Estas variables pueden ser declaradas indentadas dentro de la clase. Siguiendo con el ejemplo de las personas:

```
class Persona:
  rut = ""
  edad = 0
```

```
  datosBancarios = []
  <Resto de los datos>
```

Notamos que los atributos tienen valores de base que no indican, puesto que cuando creemos una persona, estos atributos no deben tener un valor particular, por lo que debemos entregarlos cada vez que inicialicemos un nuevo objeto de cierta clase. Es necesario entonces conocer el método **constructor**.

8.2.3. Constructor

El constructor es un método que recibe como parámetros las variables con las que inicializaremos cada nuevo objeto. Estos parámetros se asignan a los atributos. Debe ser declarado de la siguiente forma:

```
class <Nombre de la Clase>:

  def __init__(self, parametroIngresado_1,..., parametroIngresado_n):
    self.atributo_1 = parametroIngresado_1
    self.atributo_2 = parametroIngresado_2
    #...
    self.atributo_n = parametroIngresado_n
    <Contenido adicional del constructor>

  <Contenido de la clase>
```

Es necesario notar que:

- Los atributos toman el valor de los parametros ingresados y no necesariamente deben tener el mismo nombre.
- Los parámetros no necesariamente tienen que ser asignados a atributos. Se puede trabajar dentro del constructor con ellos (como en una función normal) y asignar algún otro valor a los atributos.
- Los atributos de la clase que van a recibir una asignación de algún valor deben venir con un *self.* como se mostró anteriormente.
- Si a un atributo al que se le asigna un valor dentro del constructor no es definido en la clase, este se creará automáticamente, por lo que los atributos asignados dentro del constructor **no requieren ser declarados en la clase**.

Si continuamos con el ejemplo de las personas:

```
class Persona:

  def __init__(self, nombre, edadIngresada, banco, numeroCuenta):
    self.nombre = nombre
    self.edad = edadIngresada
    lista = []
    lista.append(banco)
    lista.append(numeroCuenta)
    self.datosBancarios = lista

  <Contenido adicional de la clase>
```

Es necesario destacar que un objeto de la clase persona posee como atributos (variables que lo caracterizan) a *nombre*, *edad* y *datosBancarios*. Los atributos definidos no requieren tener el mismo nombre de los parámetros que recibe la función. Se muestra en el ejemplo que el método recibe un *"nombre"* que se asigna al atributo *"nombre"*. Pero también el método recibe una *"edadIngresada"* a pesar de que el atributo de la clase se llame *"edad"*.

8.2.4. Instanciar un objeto en el programa

Para instanciar un objeto de cierta clase en el programa se hace de la siguiente manera:

```
class miClase:
  <Contenido de la clase>

#Inicio del programa

miObjeto = miClase(parametro_1,...,parametro_n)
```

En donde *miObjeto* es un objeto de la clase *miClase*. Es necesario destacar que el constructor recibe a *self* y los parámetros, sin embargo, para instanciar un objeto en el programa, debemos ingresar sólo los parámetros.
Si queremos usar en el programa un atributo del objeto tenemos que escribir lo siguiente:

```
class miClase:
  <Contenido de la clase>


#Inicio del programa

miObjeto = miClase(parametro_1,...,parametro_n)

var = miObjeto.atributoX
#Asignamos el valor del "atributoX" a la variable var

#...

miObjeto.atributoX = valor
#El atributoX de miObjeto toma un nuevo valor
```

Para ejemplificar, continuaremos el ejemplo de las personas:

```
class Persona:

  def __init__(self, nombre, edadIngresada, banco, numeroCuenta):
    self.nombre = nombre
    self.edad = edadIngresada
    lista = []
    lista.append(banco)
    lista.append(numeroCuenta)
    self.datosBancarios = lista

  #<Contenido adicional de la clase>

#Inicio del programa

#Esta linea crea un nuevo objeto tipo persona
miPersona = Persona("Juan", 30, "Banco Estado","00000001")
print(miPersona.nombre)
print(miPersona.edad)
print(miPersona.datosBancarios)
```

El resultado de lo anterior es el siguiente:

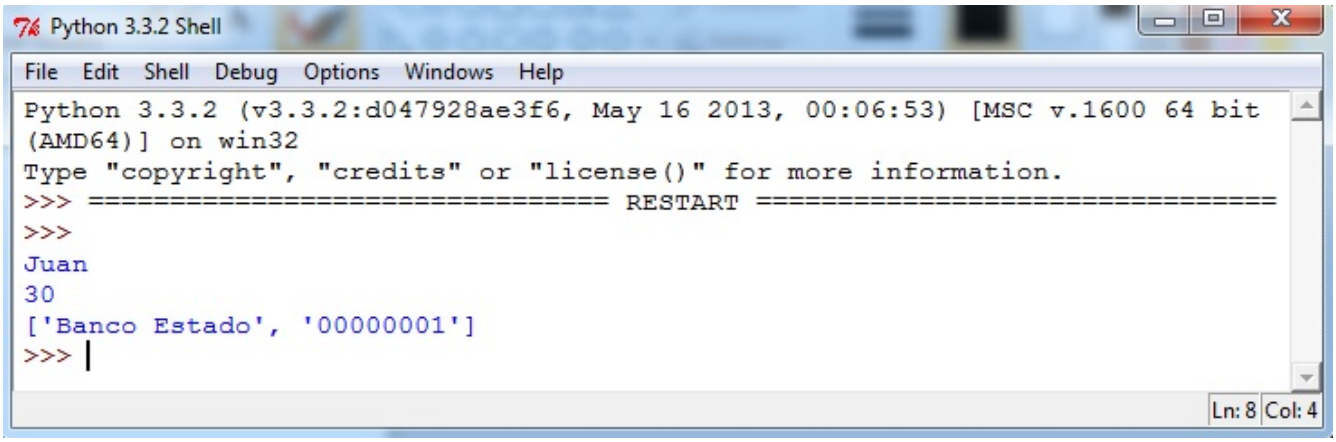

Figura 8.1: imprimir los datos de "Juan".

8.2.5. Self

Cada método definido en una clase debe recibir un primer parámetro que puede llevar cualquier nombre, pero es una buena práctica llamarlo *self*. Este parámetro nos sirve para cuando deseemos referenciar en alguno de los métodos definidos en la clase a los atributos que contiene el objeto, o al objeto mismo. En el constructor, cuando queríamos asignar valores al atributo del objeto debíamos utilizar el self. Si no hubiesemos usado el self, el programa crearía una variable que no quedaría almacenada en el objeto. En el ejemplo anterior, la variable *lista* no queda almacenada en el objeto persona, pero si se almacenará el atributo *datosBancarios*.

8.2.6. Métodos

Los métodos son funciones que se definen indentadas en la declaración de la clase como las que ya conocemos, pero la diferencia es que cuando queramos llamarlas en el programa, necesitamos de un objeto de su clase para ser utilizadas. Su definición es de la siguiente manera:

```
class <Nombre de la Clase>:

  def __init__(self, parametroIngresado_1,..., parametroIngresado_n):
    self.atributo_1 = parametroIngresado_1
    self.atributo_2 = parametroIngresado_2
    #...
    self.atributo_n = parametroIngresado_n
    <Contenido adicional del constructor>

  def metodo_1(self, parametro_1,...,parametro_n):
    <Contenido del metodo>

  def metodo_2(self, parametro_1,...,parametro_n):
    <Contenido del metodo>

  #...
```

Cuando se haga un llamado a un método se debe haber antes instanciado un objeto de aquella clase. Luego hay que escribir *objeto.método(parámetros)* como se muestra a continuación:

```
class miClase:

  <Contenido de la clase>

  def metodo_1(self, parametro_1,...,parametro_n):
    <Contenido del metodo>


  #...

miObjeto = miClase(parametro_1,...,parametro_n)
miobjeto.metodo_1(parametroIngresado_1,...,parametroIngresado_2)
```

Continuaremos con el ejemplo de las personas. Definirimos un método que representa el cumpleaños de la persona, otro que cambie los datos bancarios, y por último, un método que imprime todos los datos en la consola:

```
class Persona:

  def __init__(self, nombre, edadIngresada, banco, numeroCuenta):
    self.nombre = nombre
    self.edad = edadIngresada
    lista = []
    lista.append(banco)
    lista.append(numeroCuenta)
    self.datosBancarios = lista

  def cumpleFeliz(self): #Notamos que este metodo no recibe parametro
    self.edad = self.edad + 1

  def cambioDeBanco(self,bancoNuevo,cuentaNueva):
    self.datosBancarios[0] = bancoNuevo
    self.datosBancarios[1] = cuentaNueva

  def imprimirDatos(self):
    print("Nombre: "+self.nombre)
    print("Edad: "+str(self.edad))
    print("Banco: "+self.datosBancarios[0])
    print("Cuenta: "+self.datosBancarios[1])

miPersona = Persona("Juan",30,"Banco Estado","00000001")
miPersona.imprimirDatos()
miPersona.cumpleFeliz()
miPersona.cambioDeBanco("Banco Santander","00000002")
miPersona.imprimirDatos()
```

El resultado de lo anterior se puede apreciar en la figura 6.2. El contenido de clases es más extenso, por lo que es recomendado para el lector ahondar en esta materia.

Ejercicio Propuesto 8.1 Recientemente fueron lanzados al mercado nuevos videojuegos de la saga Pokemon. Como usted no quiere ser menos, se ha propuesto comenzar a programar su propio Pokemon. Para esto debe comenzar modelando las clases **Pokemon** y **Ataque Pokemon**. La primera debe contener como atributos la vida máxima del Pokemon, su vida actual, su tipo y una lista de ataques (que contendrá objetos de clase Ataque Pokemon). La clase Ataque Pokemon debe contener el tipo del ataque y el daño que realiza. Luego debe ser capaz de programar

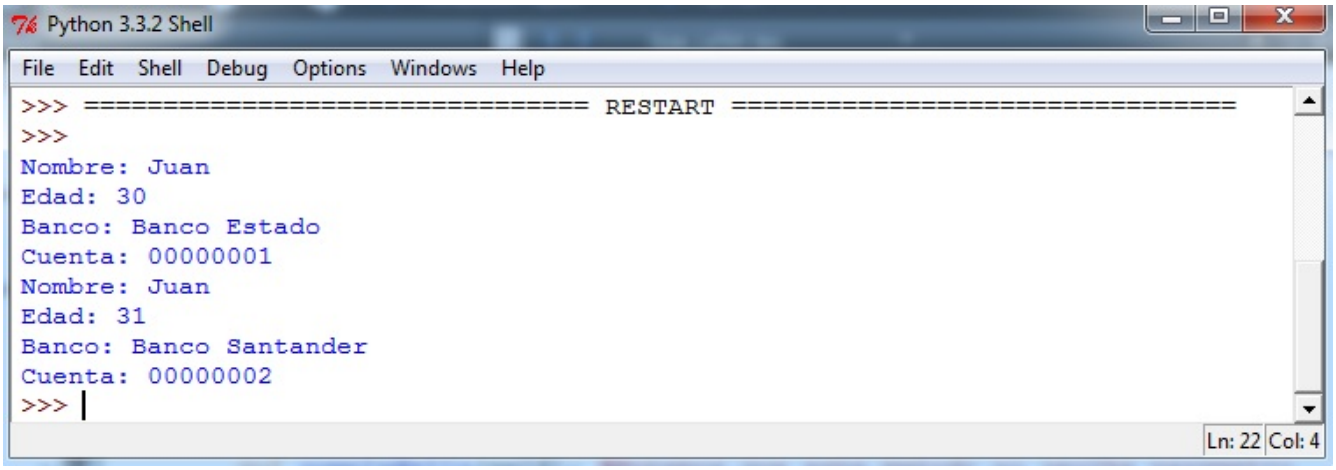

Figura 8.2: utilización de métodos.

una batalla Pokemon[1]. Todo lo demás queda a criterio del programador. Se propone hacer el juego lo más fluido posible.

[1]Se recomienda fuertemente que la batalla sea modelada con la materia de clases, ya sea como método o una clase propia. Si no se conoce el juego puede buscar en la web algunas referencias.

Capítulo 9

Archivos

9.1. ¿Por qué guardar datos?

Hasta ahora todo lo que hemos programado no es persistente en el tiempo. Es decir, una vez cerrado el programa, toda interacción con el usuario no tendrá efecto en las posteriores ejecuciones. Pero ¿Qué pasa si queremos guardar permanentemente los datos entregados por nuestro programa?. Para esto se hace necesario hacer uso de manejo de archivos. En este texto nos enfocaremos a leer archivos de texto plano (archivos con extensión .txt).

9.2. Ruta de un archivo

La ruta (o path) de un archivo es su ubicación en el computador. Esta puede ser una ruta relativa o absoluta. Una ruta absoluta indica una ruta independiente de la ubicación del .py que estamos ejecutando, mientras que una ruta relativa depende de la ubicación del .py que ejecutamos. Para el manejo de rutas se hace necesario importar el módulo os. Si el lector desea ahondar en este tema, puede buscar información acerca de esto en internet. Nosotros sólo trabajaremos con archivos que se encuentren en la misma carpeta que nuestro .py mediante rutas relativas.

9.3. Leer un archivo de texto

Para leer un archivo de texto necesitamos hacer uso de la función **open** en modo lectura. Esta función open recibe como parámetros el path del archivo (ruta en donde se encuentra ubicado) y el modo (lectura o escritura). Esta función retorna un objeto tipo *file* que debemos almacenar en una variable. Para abrir un archivo en modo lectura debemos escribir lo siguiente:

```
miArchivo = open(<Nombre del archivo>,"r")
```

En donde **Nombre del archivo** es el nombre del archivo que queremos cargar que se encuentra en la misma carpeta que nuestro programa. Este debe incluir la extensión .txt Ahora bien para leer el objeto tipo *file* que cargamos podemos usar la función *readline* o *readlines*

- **readline**: Esta función siempre lee la primera línea del archivo ya cargado (Identifica una línea segun los saltos de línea) y la retorna como string(generalmente con el fin de almacenarlo en una variable). Esta línea se elimina del archivo cargado (No del archivo original) con el fin de seguir usándola para cargar el archivo.
- **readlines**: Esta función lee todas líneas, las retorna como una lista de strings (generalmente para almacenarlas en otra variable) y las borra del archivo cargado. Los string retornados en la lista si tienen los salto de línea. Para extraerlos es posible cortarlos como ya lo vimos en el capítulo de strings. Es recomendado usar esta

función ya que deja todo almacenado en una sola lista, y es más sencillo trabajar sobre listas que sobre objetos tipo *file*.

Ejemplo 9.1 Dado el siguiente archivo de extensión .txt llamado *archivo1.txt*:

```
Hola,
soy
un
archivo.
```

Imprima en el shell de Python el contenido del archivo como un solo string sin saltos de línea. Cabe destacar que el archivo tiene un espacio al final de las tres primeras líneas, por lo que no hay que agregarlo.

Según lo descrito anteriormennte, una solución es la que se presenta a continuación. El resultado del shell se puede ver en la figura 7.1:

```
miArchivo = open("archivo1.txt","r")
s=""
lineasDelArchivo = miArchivo.readlines()
print(lineasDelArchivo) #Esto sirve para ver que valor toma la variable
for casillero in lineasDelArchivo:
  s = s + casillero[0:len(casillero)-1]
print(s)
#El salto de linea \n es un caracter
```

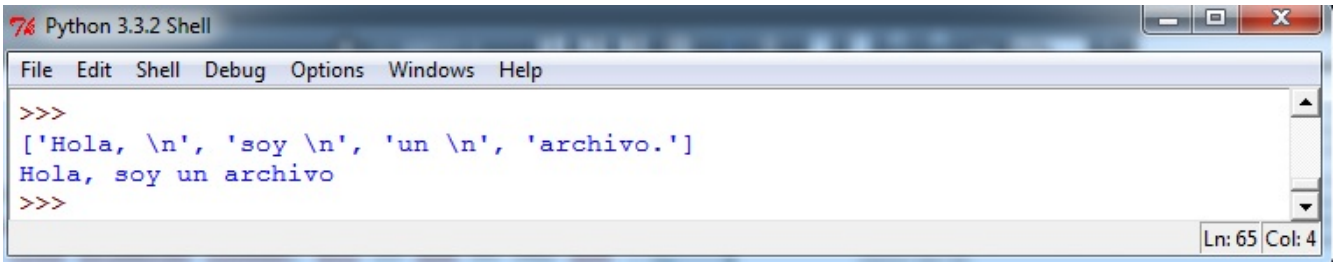

Figura 9.1: lectura de archivos.

9.4. Escribir un archivo .txt

9.4.1. Modo write

Para escribir un archivo .txt en modo write debemos crear un objeto tipo file con la función open con el parámetro de modo como "w":

```
miArchivo = open(<Nombre del archivo>,"w")
```

Si el archivo no existe, Python lo creará. Si el archivo ya existe **se borrará y se comenzará a escribir uno nuevo**. Ahora para comenzar la escritura debemos usar la función write. En el siguiente ejemplo:

```
miArchivo = open("archivo.txt","w")
miArchivo.write(string)
```

Estamos escribiendo un archivo desde el principio que se ubica en la misma carpeta de nuestro programa y se llama *archivo.txt*. Estamos escribiendo el string que ingresamos como parámetro a la función *write*. Si queremos saltar de línea debemos usar el caracter de salto de línea.
Cada vez que terminemos de escribir el archivo debemos usar la función *close()* para cerrar el objeto tipo *file* que instanciamos.

```
miArchivo = open("archivo.txt","w")
miArchivo.write(string)
miArchivo.close()
```

Ejemplo 9.2 Escriba el archivo del ejemplo 7.1:

```
Hola,
soy
un
archivo.
```

Usando lo la función write. Recuerde que el archivo se llama "archivo1.txt" y debe estar contenido en la misma carpeta que nuestro programa.

Una solución propuesta es la siguiente:

```
miArchivo = open("archivo8000.txt","w")
miArchivo.write("Hola, \n")
miArchivo.write("soy \n")
miArchivo.write("un \n")
miArchivo.write("archivo.")
miArchivo.close()
```

9.4.2. Modo append

Funciona de la misma manera que el modo write, pero la diferencia es que si el archivo ya existe no lo crea otra vez, y comienza a escribir sobre él. Para esto se debe usar la función *open* con el parámetro de modo como "a":

```
miArchivo = open("archivo.txt","a")
miArchivo.write(string)
miArchivo.close()
```

Capítulo 10

Recursión

10.1. Definir una función recursiva

En ciencias de la computación, la recursión es un método de resolución de problemas basado en la capacidad de una función de llamarse a si misma en su definición, con el objetivo de divir un problema grande, en problemas pequeños. Para su finalización es necesario establecer una condición de término de la recursión.

En términos generales una recursión se escribe como:

```
def funcion_recursiva(var_1,...,var_n):
    if var_booleana:
        #Caso base
    else:
        #Llamada recursiva
```

Ejemplo 10.1 La función factorial de n ($n!$) se defina para los naturales como 1 si $n = 0$ y $n(n-1)!$ en otro caso. Defina la función recursiva en Python que calcule el factorial de un número:

Usamos la llamada recursiva si n no es 0:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n - 1)
```

10.2. Ejemplos de funciones recursivas

10.2.1. Quicksort

Quicksort es un ordenamiento recursivo que divide una lista en sublistas y se ordenan de la siguiente forma:

- Elegir un elemento de la lista que se denominará pivote.
- Dejar todos los elementos menores que él a un lado, y todos los mayores que el pivote al otro.
- Obtenemos dos sublistas. Una de todos los elementos de la izquierda, y otra de todos los elementos de la derecha.

- Repetir recursivamente el proceso para cada sublista mientras tenga más de un elemento.

Una forma propuesta de implementar este algoritmo en Python es la siguiente:

```
def sort(lista):
    menores = []
    iguales = []
    mayores = []

    if len(lista) > 1:
        pivot = lista[0]
        for x in lista:
            if x < pivot:
                menores.append(x)
            if x == pivot:
                iguales.append(x)
            if x > pivot:
                mayores.append(x)
        return sort(menores)+sort(iguales)+sort(mayores)
        #El operador + une las tres listas. Ocupamos la misma funcion en cada sublista
    else:
        return lista
```

Notamos que *sort* es una función que recibe como parámetro una lista. En el inicio se generan tres listas auxiliares que almacenarán los menores, los mayores y los iguales al pivote, que siempre será el primer elemento de cada una de las sublistas. Luego de llenar estas tres listas auxiliares con los elementos correspondientes, debemos volver a llamar a la función sort en cada una de ellas. En el ejemplo anterior, debemos notar que lista es el parámetro ingresado, por lo que la línea que dice *"return lista"*, esta retornando los valores de las sublistas menores, iguales o mayores, cuando posean sólo un elemento. Es una buena idea mirar algún video que muestre este ordenamiento de manera gráfica para que quede más claro.

10.2.2. Permutación de Strings

Este problema consiste en tomar un string e imprimir todas sus posibles permutaciones. Es decir, todos los strings que se pueden formar usando los caracteres del string original.

```
#Funcion que elimina el i-esimo caracter de un string.
def eliminar(s,i):
    if i == 0:
        return s[1:]
    elif i == len(s)-1:
        return s[:len(s)-1]
    else:
        return s[:i]+s[i+1:]

def permutar(s,perm):
    if len(perm) == 1:
       #Si el largo del string perm es 1, la recursion termina.
        print(s+perm)
    else:
        for i in range(0, len(perm)):
           #Toma el caracter y lo pasa
            s1=s+perm[i]
            s2=eliminar(perm,i)
            permutar(s1,s2)
```

```
def llamar_permutar(s):
    permutar("",s)
```

En este problema usamos una estrategia habitual. Se define una función recursiva que recibe dos parámetros, y otra función que sirve para iniciar la recursión, que fija uno de los parámetros de la función recursiva y recibe el input. La función *permutar* recibe un string ya permutado `s` y otro por permutar `perm`. Dentro del *for* de la función *permutar*, lo que se hace es tomar de a uno los caracteres por permutar, y concatenarlos con el strings ya permutado. Por ejemplo si el input es `abc`, en la primera llamada de permutar esta función se llamará recursivamente tres veces:

```
#La llamada de permutar("","abc") llama a:
permutar("a","bc")
permutar("b","ac")
permutar("c","ab")
```

Para cada uno de estas llamadas a permutar se vuelve a ejecutar el caso recursivo:

```
#La llamada de permutar("a","bc") llama a:
permutar("ab","c")
permutar("ac","b")
#La llamada de permutar("b","ac") llama a:
permutar("ba","c")
permutar("bc","a")
#La llamada de permutar("c","ab") llama a:
permutar("ca","b")
permutar("cb","a")
```

Notamos que todas las llamadas anteriores ejecutan el caso base de la recursión, y cada una imprime respectivamente:

```
abc
acb
bac
bca
cab
cba
```

10.2.3. Mergesort

Es un ordenamiento recursivo que se basa en la estrategia "Dividir para conquistar". La idea es dividir una lista en trozos e ir ordenando por segmentos del mismo tamaño. Por ejemplo si queremos ordenar la siguiente lista de menor a mayor:

```
l = [23,4,30,7,10,11,0]
```

La dividimos:

```
[23][4][30][7][10][11][0]
```

Y ordenamos de a pares:

```
[4,23][7,30][10,11][0]
```

Luego formamos las siguientes sub-listas:

```
[4,7,23,30][0,10,11]
```

Para hacer esto se van tomando los primeros elementos de las sublistas anteriores. Por ejemplo para ordenar este segmento:

```
#Tenemos dos sub-listas:
[4,23][7,30]
#Creamos una lista resultante vacia:
[4,23][7,30]
[]
#Pasamos a ella el elemento menor entre 4 y 7:
[23][7,30]
[4]
#Luego entre 23 y 7:
[23][30]
[4,7]
#Luego entre 23 y 30:
[][30]
[4,7,23]
#Luego el elemento restante:
[]
[4,7,23,30]
```

Una implementación del algoritmo es la siguiente:

```
#Ordenamiento recursivo con mergesort
def merge_sort(lista):
    if len(lista) <=1:
        return lista
    else:
        lista_izquierda = []
        lista_derecha = []
        medio = len(lista)//2
        for i in range(0,medio):
            lista_izquierda.append(lista[i])
        for j in range(medio,len(lista)):
            lista_derecha.append(lista[j])

        lista_izquierda = merge_sort(lista_izquierda)
        lista_derecha = merge_sort(lista_derecha)
        return merge(lista_izquierda,lista_derecha)

def merge(lista_izquierda,lista_derecha):
    lista_retorno = []
    while len(lista_izquierda)>0 or len(lista_derecha)>0:

        if len(lista_izquierda)>0 and len(lista_derecha)>0:
            if lista_izquierda[0] <= lista_derecha[0]:
                lista_retorno.append(lista_izquierda[0])
                del lista_izquierda[0]
            else:
                lista_retorno.append(lista_derecha[0])
                del lista_derecha[0]
```

```
        elif len(lista_izquierda)>0:
            lista_retorno.append(lista_izquierda[0])
            del lista_izquierda[0]

        elif len(lista_derecha)>0:
            lista_retorno.append(lista_derecha[0])
            del lista_derecha[0]

    return lista_retorno
```

La función `merge_sort` es recursiva. Va dividiendo en sublistas la lista original para luego ejecutarse a si misma en estas dos sublistas. Este paso se hace hasta dejar las sublistas de tamño 1. Luego las listas se van juntando nuevamente quedando ordenadas según la función `merge`.

Capítulo 11

Simulación

11.1. ¿Para qué simular?

Hasta ahora habíamos utilizado la programación como una herramienta funcional. Dado un tipo input, generábamos un programa capaz de computar algo en base a él para luego generar un output. Si bien esto es útil, podemos hacer muchas más cosas con la programación, como por ejemplo simular un el comportamiento sistema mediante el uso de variables aleatorias sin tener que implementarlo en el mundo real. Con esto, podemo conocer los puntos débiles del sistema, nos permite realizar un análisis de sensibilidad a las variables involucradas y también tomar decisiones en torno a las estadísticas obtenidas.
Para simular no hay una receta absoluta, simplemente debemos aplicar lo aprendido hasta ahora para intentar resolver un problema con cierto grado de seguridad. A grandes rasgos lo que debemos hacer es:

- Identificar las entidades que realizan acciones de interes en el sistema.
- Hacerlos interactuar a través del tiempo.
- Llevar estadísticas almacenadas en variables.

Para tener una mayor seguridad de los resultados obtenidos, es posible ejecutar la simulación varias veces, y calcular promedios de las estadísticas recogidas.

11.2. Ejemplos de simulación

11.2.1. Ataque Zombie

Un ataque zombie afecta a una ciudad aislada. La cura tardará 10 días en ser elaborada, pero sólo es factible de usar si hay menos de un cuarto de la población afectada. En la ciudad viven 200000 personas y existen 500 zombies. Cada zombie tiene una probabilidad p (número entre 0 y 1) de realizar un ataque por día. En cada ataque son infectadas una o dos personas (el número de infectados es aleatorio). Nos interesa saber que decisión tomar (correr o intentar buscar la cura) según el valor de p.

```
import random

class ZombieAttack:
    def __init__(self,p):
        self.personas = 200000
```

```
        self.zombies = 500
        self.probabilidad = p
        self.bitacora = []

    def iteracion(self,j):
        c=0
        for i in range(self.zombies):
            a = random.uniform(0,1)
            if a<self.probabilidad:
                b = random.randint(1,2)
                if self.personas - b >0:
                    self.personas -= b
                    self.zombies += b
                    c += b
                else:
                    self.zombies += self.personas
                    c += self.personas
                    self.personas = 0
        s = "En el dia "+str(j+1)+" fueron infectadas "+str(c)+" personas"
        self.bitacora.append(s)


    def simulacion(self):
        for i in range(10):
            self.iteracion(i)


    def estadisticas(self):
        print("Zombies: ",self.zombies)
        print("Personas: ",self.personas)
        for s in self.bitacora:
            print(s)
        if self.personas/4 > self.zombies:
            print("Elaborar cura!")
        else:
            print("Run to the hills!")

p = int(input("Ingrese la probabilidad: "))
a = ZombieAttack(p)
a.simulacion()
a.estadisticas()
```

Capítulo 12

Ejercicios

12.1. Cola de supermercado

Modele una cola de un servicio con una sola caja. Los clientes llegan en un tiempo aleatorio entre 7 y 12 minutos. La caja tarda en atender al cliente entre 10 y 15 minutos. La caja está abierta durante 480 minutos y debe correrla por 5 días. Queremos saber cuantos clientes en promedio no son atendidos en un día, cuanto se demora un cliente en promedio y el promedio de gente atendida por día.

```
import random

class cliente:
    def __init__(self):
        self.tes = 0
    def cambiarTES(self,tiempo):
        self.tes = tiempo

class servicio:
    def __init__(self):
        self.clienteActual = None

    def pasarCliente(self, unCliente):
        tiempo = random.randint(10,15)
        unCliente.cambiarTES(tiempo)
        self.clienteActual = unCliente

class simulacion:
    def __init__(self):
        self.serv = servicio()
        self.clientesAtendidos = 0
        self.tiempoEnAtencion = 0
        self.listaDeEspera = []

        primercliente = random.randint(7,12)
        self.siguienteCliente = primercliente #Tiempo en el que llegara el siguiente cliente
```

```
    def ejecutar(self):
        minutos = 480
        while minutos >= 0:
            if self.siguienteCliente >0:
                self.siguienteCliente -= 1

            if self.siguienteCliente == 0:
                self.listaDeEspera.append(cliente())
                t = random.randint(7,12)
                self.siguienteCliente = t #Nuevo contador que espera la llegada de un nuevo cliente

            if (self.serv).clienteActual != None:
                ((self.serv).clienteActual).tes -= 1
                if ((self.serv).clienteActual).tes == 0:
                    self.clientesAtendidos +=1
                    ((self.serv).clienteActual) = None

            if (self.serv).clienteActual == None:
                if len(self.listaDeEspera)>0:
                    ti = (self.serv).pasarCliente(self.listaDeEspera[0])
                    del self.listaDeEspera[0]
            minutos-=1
        print("Quedo en espera: "+str(len(self.listaDeEspera)))
        print("Se atendieron: "+str(self.clientesAtendidos))
        return str(len(self.listaDeEspera)) + "|" + str(self.clientesAtendidos)

dia = 1
sumListaEspera = 0
sumClientesAtendidos = 0
sumPromTiempo = 0
for i in range(0,5):
    print("### Dia: "+str(dia)+" ###")
    miSim = simulacion()
    s = miSim.ejecutar()
    l = s.split("|")
    a = int(l[0])
    b = int(l[1])
    sumListaEspera += a
    sumClientesAtendidos += b
    sumPromTiempo += 480/b
    dia+=1

print(sumListaEspera/5)
print(sumClientesAtendidos/5)
print(sumPromTiempo/5)
```

12.2. Banco de la Plaza

Hoy en día los fraudes bancarios han crecido exponencialmenete. Muchas veces los clientes de un banco no se daban cuenta porque no tenían acceso a revisar su estado de cuenta a través de internet. Para dar solución a este problema, el Departamento de Ciencia de la Computación te ha encargado crear un programa en Python que permita llevar la administración de las cuentas corrientes de los clientes del ***Banco de la Plaza***.

El programa debe permitir almacenar clientes y sus transacciones bancarias: nombre, rut, género, balance y lista_transacciones. El atributo lista_transacciones a su vez está compuesto por tuplas de 4 elementos cada una: nombre_transacción, fecha_transacción, descripción_transacción y monto_transaccion,de esta forma un cliente puede tener muchas transacciones registradas. El formato de la fecha es dd-mm-yyyy, donde dd es día, mm es mes y yyyy es año, ejemplo 21-03-2014. El atributo balance determina el monto inicial que un cliente tiene en su cuenta bancaria. Los tipos de transacción válidos son: Deposito, Giro y Compra.

El programa debe permitir crear nuevos clientes y administrar su información. Por lo tanto, se debe contar con el siguiente menú principal:

```
Bienvenido al Sistema del Banco de la Plaza
Seleccione una accion:
[1] Crear cliente
[2] Mostrar clientes
[3] Buscar clientes
[4] Agregar transaccion a un cliente
[5] Salir
```

La opción [1] del menú principal debe permitir al usuario crear un nuevo cliente.

La opción [2] del menú principal debe permitir desplegar todos los clientes que están registrados en el banco. Por ejemplo, si se han registrado 3 clientes, se debe mostrar la información completa de cada cliente:

```
Seleccione cliente:
[1] Julio Meneses, 1.456.273-5, Masculino,
    [(Deposito, 15-01-2010, para gastos comunes,1000),
     (Compra, 11-03-2012, zapatos Merrel,50000)]
[2] Fabiola Retamal, 19.264.198-4, Femenino, [(Giro, 05-19-2013, para prestamo,17000),
     (Giro, 11-04-2014, compra cartera,350000)]
[3] Alicia Aliaga, 18.984.853-3, Femenino,
      [(Compra, 12-03-2014, cartera Luis Griton,500)]
[4] Salir
```

La opción [3] Buscar del menú principal debe mostrar el siguiente sub-menú:

```
Ingrese el tipo de busqueda:
[1] Por nombre cliente
[2] Por nombre de cliente y tipo de transaccion
[3] Por fecha
[4] Salir
```

La opción [1] debe preguntar por el nombre del cliente y deplegar el registro completo del cliente.

La opción [2] debe preguntar por el nombre del cliente y el tipo de transaccion (Deposito, Giro o Compra). De esta forma, se deplegarán todas las transacciones de cierto tipo de un cliente determinado.

La opción [3] debe mostrar la información de todos los clientes que tengan transacciones en la fecha ingresada. Es decir, el nombre del cliente, el rut y la tupla de la transacción con la fecha buscada

La opción [4] del menú principal debe permitir agregar una nueva transaccion a un cliente. Pare ellos, se debe deplegar todos los clientes, seleccionar uno de ellos y finalmente agregar la transaccion. Ejemplo, si se selecciona esta opción, se despliegan los cliente hasta ahora registrados:

```
Seleccione cliente:
[1] Julio Meneses, 1.456.273-5, Masculino,
    [(Deposito, 15-01-2010, para gastos comunes,1000),
     (Compra, 11-03-2012, zapatos Merrel,50000)]
[2] Fabiola Retamal, 19.264.198-4, Femenino, [(Giro, 05-19-2013, para prestamo,17000),
     (Giro, 11-04-2014, compra cartera,350000)]
[3] Alicia Aliaga, 18.984.853-3, Femenino,
      [(Compra, 12-03-2014, cartera Luis Griton,500)]
[4] Salir
2
Ingrese tipo transaccion: Compra
Ingrese fecha transaccion: 01-04-2014
Ingrese descripcion: Perfume Carolina Lesera
Ingrese monto de la transaccion: 110000
Se ha agregado una transaccion al cliente Fabiola Retamal:
    [(Giro, 05-19-2013, para prestamo,17000),
     (Giro, 11-04-2014, compra cartera,350000),
     (Compra, 01-04-2014, Perfume Carolina Lesera,11000)]
```

La opción [5] del menú principal permite salir completamente del programa de Sistema Banco de la Plaza.

Se te pide que crees la clase Cliente que debe contar con los siguientes atributos:

- Un string para el nombre del cliente
- Un string para el rut del cliente
- Un string para la fecha de la transaccion
- Un entero para el balance del cliente
- Una lista para las transacciones del cliente (recuerda que cada transacción será una tupla con (tipo transaccion, fecha transaccion, descripcion,monto))

Además la clase cliente debe contar al menos con los siguientes métodos:

- `agregar_transaccion(self, nuevatransaccion)`: Este método permite agregar una nueva transaccion al listado de transacciones que ya tiene un cliente.
- `verifica_balance(self)`: Este método retorna True si el cliente aún tiene balance positivo, y False caso de tener balance negativo.
- `actualiza_balance(self,monto)`: Este método actualiza el balance de un cliente utilizando el parámetro monto que recibe el método.

Tu programa principal debe contar con al menos las siguientes funciones:

- `mostrar_menuprincipal()`: Esta función solo depliega el menu principal.
- `mostrar_clientes(lista_clientes)`: Esta función debe deplegar todos los clientes del banco.

- `buscar_cliente(lista_clientes, cliente_a_buscar)`: Esta función recibe la lista_clientes y el cliente_a_buscar. La función debe mostrar toda la información que contenga el cliente_a_buscar.
- `buscar_transaccion_enCliente(lista_clientes, transaccion_a_buscar,cliente_a_buscar)`: Esta función recibe la lista_clientes, la transaccion_a_buscar y cliente_a_buscar. La función debe mostrar el nombre, rut del cliente_a_buscar y además todas las transacciones que sean del tipo transaccion_a_buscar de ese cliente.
- `buscar_fechaTransaccion(lista_clientes, fecha_a_buscar)`: Esta función recibe la lista_clientes y la fecha_a_buscar. La función debe mostrar todos los clientes que tengan transacciones en la fecha_a_buscar.

En el caso que se esté buscando y no se encuentren coincidencias, se debe mostrar el mensaje "no existen coincidencias".

```
class Cliente:
    def __init__(self,_nombre,_rut,_sexo,_balance,_trainicial):
        self.nombre = _nombre
        self.rut = _rut
        self.sexo = _sexo
        self.balance = _balance
        self.tr = []
        (self.tr).append(_trainicial)

    def agregar_transaccion(self, nuevatransaccion):
        #Recibe la tupla lista
        (self.tr).append(nuevatransaccion)

    def verifica_balance(self):
        if self.balance>=0:
            return True
        return False

    def actualiza_balance(self,monto):
        self.balance = monto

    def printCliente(self):
        print(self.nombre+", "+self.rut+", "+self.sexo+",")
        print(self.tr)


def mostrar_menuprincipal():
    a = int(input("Seleccione una accion:\n[1] Crear cliente\n[2] Mostrar clientes\n[3] Buscar cli
    return a

def mostrar_menubusqueda():
    a = int(input("Ingrese el tipo de busqueda:\n[1] Por nombre cliente\n[2] Por nombre de cliente
    return a

def buscar_cliente(lista_clientes,cliente_a_buscar):
    for cliente in lista_clientes:
        #Recibe nombre de cliente
        if cliente.nombre == cliente_a_buscar:
            return cliente
        return None

def realizar_busqueda(num, lista_clientes):
    count = 1
    matched = False

    if num==1:
        nombre = input("Ingrese nombre cliente: ")
        micliente = buscar_cliente(lista_clientes,nombre)
        if micliente!=None:
            matched = True
            print("["+str(count)+"] ",end="")
            micliente.printCliente()
            count+=1

    elif num==2:
```

```
        nombre = input("Ingrese nombre cliente: ")
        tx = input("Ingrese tipo de transaccion: ")
        matched = buscar_transaccion_enCliente(lista_clientes,tx,nombre)


    elif num==3:
        fecha = input("Ingrese fecha: ")
        matched = buscar_fechaTransaccion(lista_clientes,fecha)


    if not(matched) and num!=4:
        print("No existen coincidencias")
    print("")




def mostrar_clientes(lista_clientes):
    count = 1
    for cliente in lista_clientes:
        print("["+str(count)+"] ",end="")
        cliente.printCliente()
        count+=1

def buscar_fechaTransaccion(lista_clientes,fecha_a_buscar):
    aux = []
    matched = False
    count = 1
    imprimir = False
    for cliente in lista_clientes:
        for tx in cliente.tr:
            if fecha_a_buscar in tx[1]:
                matched = True
                imprimir = True
                aux.append(tx)
        if imprimir:
            print("["+str(count)+"] "+cliente.nombre+", "+cliente.rut+", "+cliente.sexo+",")
            print(aux)
            count+=1
        aux = []
        imprimir = False
    return matched

def buscar_transaccion_enCliente(lista_clientes,transaccion_a_buscar,cliente_a_buscar):
    aux = []
    imprimir = False
    matched = False
    count = 1
    for cliente in lista_clientes:
        if cliente.nombre == cliente_a_buscar:
            for tran in cliente.tr:
                if transaccion_a_buscar in tran[0]:
```

```
                    matched = True
                    imprimir = True
                    aux.append(tran)
        if imprimir:
            print("["+str(count)+"] "+cliente.nombre+", "+cliente.rut+", "+cliente.sexo+",")
            print(aux)
            count+=1
        aux = []
        imprimir = False
    return matched

def crea_clientes():
    nom = input("Ingrese nombre del cliente: ")
    rut = input("Ingrese RUT del cliente: ")
    g = input("Ingrese genero del cliente: ")
    b = int(input("Ingrese balance: "))
    tx = input("Ingrese transaccion: ")
    tx2 = tx.split(",")
    bal = (tx2[3]).replace(" ","")
    bal2 = int(bal)
    tupla = (tx2[0],tx2[1],tx2[2],bal2)
    cl = Cliente(nom,rut,g,b,tupla)
    return cl

def agregar_tx(lista_clientes):
    print("Seleccione cliente: ")
    mostrar_clientes(lista_clientes)
    n3 = int(input())
    cli = clientes[n3-1]
    tipo = input("Ingrese tipo de transaccion: ")
    fec = input("Ingrese fecha de transaccion: ")
    de = input("Ingrese descripccion: ")
    m = int(input("Ingrese monto de la transaccion: "))
    if tipo == "Giro" or tipo == "Compra":
        aux = cli.balance
        cli.actualiza_balance(aux - m)
        if not(cli.verifica_balance()):
            print("No posee suficiente saldo")
            cli.actualiza_balance(aux)
        else:
            a = (tipo,fec,de,m)
            cli.agregar_transaccion(a)
            print("Se ha agregado una transaccion al cliente "+str(cli.nombre))
            print(cli.tr)
            print("")
    else:
        aux = cli.balance
        cli.actualiza_balance(aux + m)
        a = (tipo,fec,de,m)
        cli.agregar_transaccion(a)
        print("Se ha agregado una transaccion al cliente "+str(cli.nombre))
        print(cli.tr)
        print("")
```

```
#Programa
print("Bienvenido al Sistema de Banco de la Plaza")
clientes = []
numero = mostrar_menuprincipal()
while numero!=5:
    if numero == 1:
        cl = crea_clientes()
        clientes.append(cl)
        print("El cliente "+cl.nombre+" fue ingresado correctamente")
        print("")

    elif numero == 2:
        mostrar_clientes(clientes)
        print("")

    elif numero == 3:
        n2 = mostrar_menubusqueda()
        realizar_busqueda(n2,clientes)
        print("")

    elif numero == 4:
        agregar_tx(clientes)
    numero = mostrar_menuprincipal()



print("Gracias por elegir el Sistema de Banco de la Plaza!")
```

12.3. Algebraco

Algebraco[1] es una aplicación que sirve para manipular expresiones algebraicas. Internamente, Algebraco representa las expresiones aritméticas como:

- un nímero entero, o
- una lista de tres elementos, donde tanto el primer como el segundo elemento son expresiones, y el tercer elemento es un operador. Los operadores permitidos son los strings "+" , "-" , "*" y "/".

Así, `4`, `[1,[3,4,‘‘+’’],‘‘-’’]` y `[[5,5,‘‘/’’],[2,1,‘‘-’’],‘‘+’’]` son expresiones permitidas en Algebraco que corresponden a 4, (1-(3+4)) y ((5/5) + (2-1)) respectivamente.

1. Escribe una función `evalua`, que tome como argumento una expresión de Algebraco y retorne el número al cual esta expresión es igual. Por ejemplo,
   `evalua(4)` debe retornar 4, `evalua([1,[3,4,‘+’],‘-’])` debe retornar -6, `evalua([[11,7,‘-’],[4,3,‘+’],‘*’])` debe retornar 28.

   Ayuda: Para saber si la variable a es un int, puedes preguntar `if type(a) == int`.

2. Informalmente, las *subexpresiones* de una expresión `E` son partes de `E` que al mismo tiempo son expresiones. Por ejemplo, si `E = [1,[2,3,‘‘-’’],‘‘+’’]`, las subexpresiones de `E` son `[1,[2,3,‘‘-’’],‘‘+’’]`, `1`, `[2,3,‘‘-’’]`, `2`, y `3`. Por definición, `E` es siempre una subexpresión de `E`.
   Escribe una función `cuenta` que reciba una expresión y un número, tal que `cuenta(exp,n)` retorne el número de subexpresiones de `exp` que, al evaluarse, tienen el valor n. Por ejemplo, `cuenta([1,1,‘‘+’’],1)` retorna 2 porque hay dos subexpresiones de `[1,1,‘‘+’’]`, en particular 1 y 1, que son iguales a 1.

```
#Funcion recursiva que evalua una expresion.
def evaluate(exp):
    if type(exp)==int:
        return exp
    else:
        if exp[2] == '+':
            return evaluate(exp[0]) + evaluate(exp[1])
        elif exp[2] == '-':
            return evaluate(exp[0]) - evaluate(exp[1])
        elif exp[2] == '*':
            return evaluate(exp[0]) * evaluate(exp[1])
        elif exp[2] == '/':
            return evaluate(exp[0]) / evaluate(exp[1])

#Funcion recursiva que retorna una lista con todas las subexpresiones.
def subexp(exp,l):
    l.append(exp)
    if type(exp[0]) == int:
        l.append(exp[0])
    else:
        subexp(exp[0],l)
    if type(exp[1]) == int:
        l.append(exp[1])
    else:
        subexp(exp[1],l)
```

[1] Examen 2013-2

```
#Funcion que llama a subexp.
def call_subexp(exp):
    l = []
    subexp(exp,l)
    return l

#Funcion que retorna el numero de subexpresiones iguales a n.
def count_subexp(exp,n):
    l = call_subexp(exp)
    count=0
    for expr in l:
        if evaluate(expr)==n:
            count+=1
    return count



#Algunas pruebas...
l = [1,[[1,1,'+'],4,'+'],'-']
print(evaluate(l))

a = call_subexp(l)
print(a)

print(count_subexp([[5,5,"/"],[2,1,"-"],"+"],1))
```