ING1310 - El Libro

Juan Ignacio Sáa Hargous

15 de agosto de 2014

Índice general

Capítulo 1

Estructura de este libro

Cada capítulo estará dividido en secciones. En cada una de ellas habrá un tema a estudiar. Dentro de cada sección será posible encontrar consejos, que ayudarán a recalcar puntos importantes.

Consejo.

Lea atentamente cada sección del libro.

También encontrará destacados errores comunes de programación, estos están recopilados de las correcciones de pruebas y tareas de versiones anteriores del curso de Introducción a la Computación.

Error común de programación.

Copiar en una evaluación, esto lleva a reprobar el curso con nota 1.

Dentro de cada capítulo, encontrará ejemplos de código, estos tienen el siguiente formato:

```
import math

# función para encontrar el área de una esfera
def area(radio):
    a = 4 * math.pi * r**2
    return a
```

Ejemplo de código.

Al final del contenido de cada capítulo habrá una sección de ejercicios propuestos, estos estarán divididos en tres grupos:

- Básicos: ejercicios simples para captar la base de la materia nueva.
- Sugeridos: del nivel esperado para este curso.
- Desafiantes: para los que quieran dominar la materia en un nivel avanzado.

Capítulo 2

Algoritmos

2.1. Definición

Un algoritmo es una secuencia ordenada de instrucciones que permite realizar una tarea mediante pasos sucesivos que no generen dudas a quien deba ejecutarla.

Por ejemplo, un algoritmo para calcular el promedio de cinco notas podría ser el siguiente:

1. Sume las cinco notas.
2. Divida la suma calculada por cinco.
3. El resultado de la división es el promedio.

Este algoritmo resuelve nuestro problema, pero no es aplicable si cambia la cantidad de notas sobre las que queremos calcular el promedio. Para resolver el problema general, podríamos modificarlo de la siguiente manera:

1. Cuente todas las notas.
2. Sume todas las notas.
3. Divida la suma calculada por la cantidad de notas original.
4. El resultado de la división es el promedio.

Nuestro algoritmo ahora tiene un paso adicional, que lo hace más versátil, ya que ahora se puede aplicar para resolver más problemas, incluido el problema original.

Un algoritmo puede ser tan sencillo como una única instrucción o tan complejo como sea necesario. Ejemplos típicos son los manuales de ensamblado de muebles, automóviles e, incluso, transbordadores espaciales.

2.2. Representaciones de un algoritmo

Ya vimos que un algoritmo puede ser representado como una lista de pasos en un orden específico, esta es una forma eficiente para algoritmos sencillos y con una cantidad pequeña de pasos.

Otra forma de representación son los diagramas de flujo, descripciones en pseudocódigo, código de algún lenguaje de programación, estructuras secuenciales, etc.

Por ahora nos centraremos en la representación como secuencia de pasos y diagramas de flujo, ya que estos nos ayudarán a convertir un algoritmo en el código de un programa computacional que resuelva un problema específico.

2.3. Diagramas de flujo

Los diagramas de flujo son una forma sencilla de representar un algoritmo. Por ejemplo:

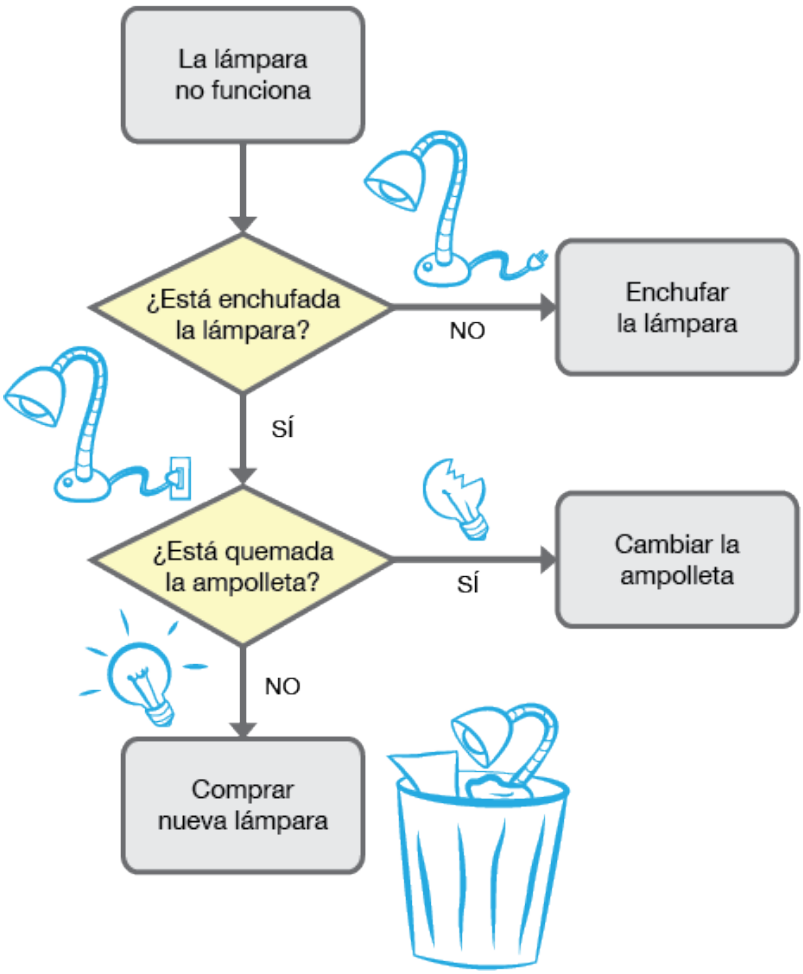


Diagrama de flujo para arreglar una lámpara.

Un diagrama de flujo está compuesto por los siguientes elementos:

- **Tareas**, representadas por rectángulos.
- **Transiciones**, representadas por flechas.

- **Decisiones**, representadas por rombos.

Además, un diagrama de flujo debe cumplir con las siguientes condiciones:

- Debe haber una única **tarea incial**, a la que no llegan transiciones.
- Las transiciones tienen un solo sentido.
- Desde una tarea puede existir a lo más una transición de salida.
- La pregunta de una decisión se debe poder responder con sí o no. Cada respuesta debe tener una única transición de salida.
- Pueden haber múltiples **tareas terminales**, estas se reconocen porque desde ellas no salen transiciones.

Otro ejemplo de diagrama de flujo:

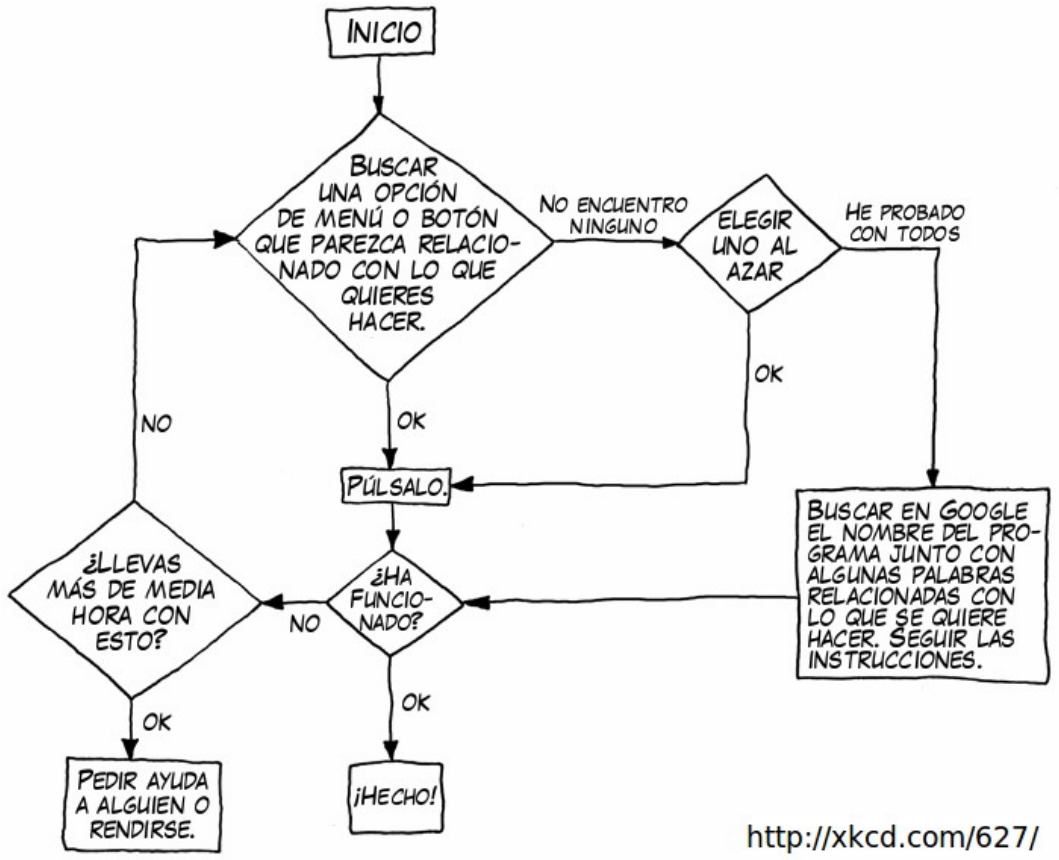


Diagrama de flujo para soporte técnico.

2.4. Variables

Una variable es un espacio en la memoria del computador donde se guardan datos durante la ejecución de un programa (y se pierden al terminar de ejecutarlo).

Toda variable debe tener un nombre (o identificador), para diferenciarla de las otras variables, y un tipo, para indicar qué valores puede guardar.

Puede que ya se hayan familiarizadon con variables en demostraciones matemáticas, donde es común ver declaraciones del tipo "sea $x \in \mathbb{R}$ tal que...", nosotros diríamos que existe una variable llamada x que pertenece al tipo de los números reales.

En general, los lenguajes de programación tienen varios tipos predefinidos para las variables más utilizadas (números y variables lógicas son los más comunes) y ofrecen formas de agregar nuevos tipos de datos para guardar información que puede ser más compleja (como un libro, una persona o una universidad).

Siguiendo con nuestro ejemplo para calcular el promedio de algunas notas, podemos incluir variables en su ejecución. Para esto modificaremos el proceso de la siguiente forma: pediremos al usuario que ingrese las notas una a una, indicando con un cero que ha terminado de ingresarlas.

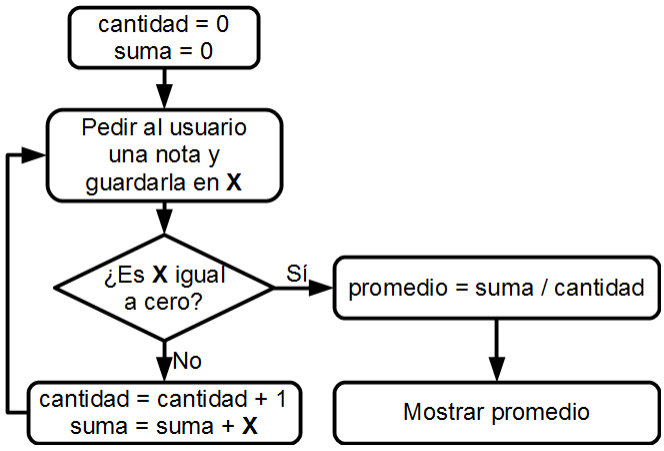


Diagrama de flujo para calcular un promedio.

En el diagrama de flujo anterior existen 4 variables: `cantidad`, `suma`, `X` y `promedio`. De estas, `X` es la única que se pide directamente a la persona que ejecuta el algoritmo. Las otras tres variables se usan para cálculos internos y permiten entregar el resultado final.

2.5. Ejercicios

2.5.1. Básicos

1. Dibuje el diagrama del flujo del algoritmo para hacer hielo en el *freezer* de un refrigerador. ¿Qué pasa si este está desenchufado? ¿Qué pasa si no hay cubetas de hielo?

2.5.2. Sugeridos

1. Dibuje un diagrama de flujo para determinar si una secuencia de números está ordenada en orden ascendente. Pida los números al usuario uno a uno.

2. Una compañía de seguros quiere que usted dibuje un diagrama de flujo de su proceso de cálculo de prima para un seguro de vida, y para decidir si debe asegurar a un cliente o rechazarlo.

   La prima base es de $100, pero se modifica según ciertas condiciones:

   - Si el asegurado tiene menos de 35 años, la prima baja en un 15 %
   - Si el asegurado tiene más de 50 años, la prima sube en un 15 %
   - Si el asegurado fuma, sube en 20 %
   - Si practica deportes extremos, sube en 30 %
   - Si presenta diagnósticos de enfermedades crónicas, sube en 20 %

   Sin embargo, la compañía de seguros rechazará asegurar a quienes cumplan alguna de las siguientes condiciones:

   - Si tiene más de 75 años
   - Si tiene más de 60 años, fuma, y tiene enfermedades crónicas
   - Si tiene menos de 20 años, y practica deportes extremos

2.5.3. Desafiantes

1. Dibuje un diagrama de flujo para determinar si un número ingresado por el usuario es primo o compuesto.

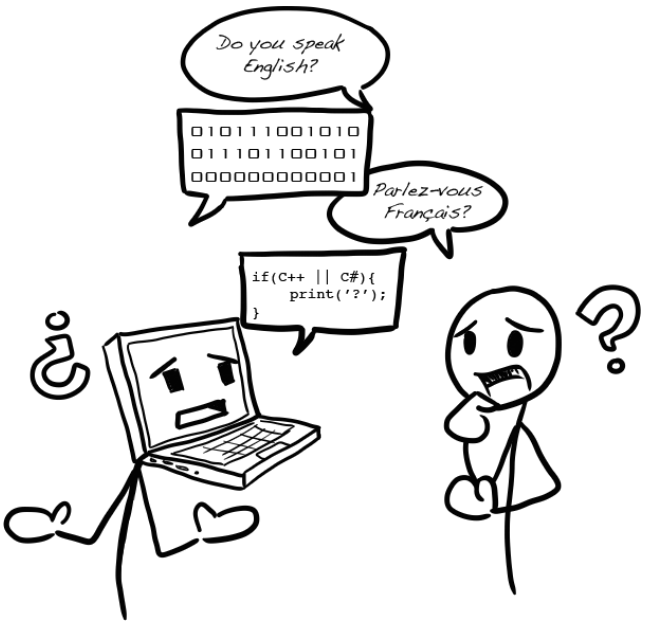
Capítulo 3

Lenguajes de programación

3.1. Necesidad

Hemos visto que un algoritmo se puede representar como una lista de pasos o como un diagrama de flujo. Dentro de un algoritmo pueden haber variables para guardar valores necesarios en el cálculo de una solución. También permiten interactuar con el usuario, pidiéndole que ingrese valores y mostrándole mensajes.

Aunque ambas notaciones son simples de leer y entender por una persona, nos interesa que un computador pueda entender un algoritmo y ejecutarlo. Para esto necesitamos un lenguaje común que sea comprensible tanto para las personas que lo escriben como para la máquina que lo ejecuta. Estos son los lenguajes de programación.



3.2. Un poco de historia

Los lenguajes de programación son anteriores a los computadores, se usaron para programar telares (desde 1801) y pianolas (desde 1876). Más adelante, entre 1930 y 1940, Alan Turing desarrolló máquinas teóricas que permitían codificar y ejecutar algoritmos.

En los años 40 se desarrollaron los primeros computadores digitales. Estos eran programados directamente en instrucciones de procesador, es decir, en binario. El operador ingresaba en la máquina las instrucciones a ejecutar sin ninguna validación de por medio.

El siguiente paso fueron los lenguajes ensamblador. Una ventaja de estos es que pueden ser leídos por un operador, pero es necesario convertirlos en instrucciones de máquina antes de poder ejecutarlos. Este proceso lo lleva a cabo un *compilador*, que es un programa especializado en convertir el código escrito por una persona en lenguaje de máquina.

```
ID1F    DEC     0               # DOWNLINK ID RECEIVED
DEL31   DEC     0               # DOWNLINK INPUT COMPLETE
FLAGT   DEC     0               # MEMORY TEST
FLAG1   DEC     0               # 20MS BRANCH CONTROL
DEL2    OCT     0               # STAGED FLAG.
DEL5    OCT     0               # ATTHLD LOCK FLAG.
PI      DEC     3.14159266B3
6K13    DEC     -1.59286B1      # RADAR RANGE RATE TO FPS AT 13
DVGXX   DEC     0B13            # VEL TO GAIN IN EXT DELTA V
DVGXY   DEC     0
DVGXZ   DEC     0
DEL20   DEC     0               # LOGIC FLAG FOR ENG CONTROL
DEL21   OCT     0               # LUNAR SURFACE FLAG.
2J      DEC     1.9970B7        # COTAN OF DESIRED LOS AT TPI
DEL42   OCT     0               # LUNAR ALINE COMPLETE FLAG
DEL45   EQU     DEL42           # RR RANGE/RANGE RT. UPDATE FLAG
K55     OCT     377777          # S.F. FOR HDOT DISPLAY
MU3     DEC     0               # P-ITER COUNTER.
RD3DTL  EQU     MU3             # RD3DOT LOWER LIMIT
5K17    DEC     -.01B-2         # YD3DOT LOWER LIMIT
MU6     DEC     0               # STAGING COUNTER
1K37    DEC     15.B17          # ACCEL CAL DURATION
MU8     DEC     0               # ULLAGE COUNTER
MU10    DEC     0               # GYRO CALIBRATE DURATION
1K9     DEC     5.0B17          # ULLAGE COUNTER LIMIT
1K30    DEC     150.B17         # GYRO CALIBRATE DURATION
2K17    DEC     5.B17           # NO OF P-ITERATIONS-3
MU19    DEC     0               # MARK COUNTER
4K23    DEC     62.0B17         # STAGING TIME DELAY
S623    DEC     0B3             # EX SELECTION IN G. S
1K4     DEC     .1B0            # DISPLAY INTERPOLATION
1K24    DEC     .87E-3B1        # SINGULARITY THRESHOLD
```

Ejemplo de código ensamblador. Fragmento del código original del Apollo 11.
`https://code.google.com/p/virtualagc/source/browse/trunk/FP8/FP8.s?r=258`

Más adelante se trató de modificar la sintaxis de los lenguajes para que fuese más fácil leerlos y utilizarlos para escribir programas más complejos. Dentro de los siguientes lenguajes, destaca BASIC como uno de los más adoptados:

```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

Ejemplo de código en BASIC.

Los lenguajes de programación han ido evolucionando con el tiempo. Hoy en día existen miles de lenguajes de programación, entre ellos se encuentran: Basic, C++, COBOL, Fortran, Haskell, Java, Lisp, MATLAB, Perl, Python, Scala, TeX, etc.

En este curso estudiaremos Python como lenguaje de programación. Python fue diseñando por Guido van Rossum, su primera versión apareción en 1991 y se sigue actualizando hasta hoy.

Capítulo 4

Empezando con el código

4.1. Lenguaje a utilizar

En este libro utilizaremos el lenguaje de programación Python, en su versión 2.7. Este se puede descargar desde su página oficial[1] para utilizar directamente. Tenga en cuenta que más adelante utilizaremos módulos extra para trabajar con vectores numéricos y gráficos, que tendrá que instalar por separado.

Es recomendable, para los que trabajen en Windows, instalar Python(x, y). Esta es una distribución gratuita que incluye un editor de código más amigable, junto con todas las extensiones necesarias para este curso. Se puede descargar desde su página oficial[2].

[1] http://python.org/
[2] http://code.google.com/p/pythonxy/

4.2. Variables

Una variable es un espacio en la memoria del computador que se utiliza para almacenar datos de forma temporal (esto quiere decir que los datos se guardan mientras el programa se está ejecutando y que se pierden al cerrarlo).

Toda variable tiene un identificador (o nombre) y puede guardar datos de distintos tipos[3]. El nombre de una variable puede estar compuesto por letras, números y el síbolo de _ (guión bajo), con la restricción de que el primer símbolo de un identificador no puede ser un número.

Ejemplos de nombres válidos para variables:

- `variable`
- `color42`
- `x`
- `cantidadDeOjos`
- `_persona`
- `i`
- `color1`
- `numero_de_dia`
- `n`

Ejemplos de nombres **no** válidos para variables:

- `1variable`
- `hola!`
- `?`
- `42`
- `#personas`
- `color de ojos`

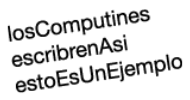
De los ejemplos anteriores podemos deducir que un identificador no puede contener espacios ni otros símbolos. Además, en Python no está permitido el uso de letras fuera del alfabeto inglés, esto significa que no se pueden usar ñ, ß, ç ni letras acentuadas.

Como regla general, no pueden existir dos variables con el mismo nombre, pero es importante tomar en cuenta que **Python diferencia mayúsculas de minúsculas**, por lo que `nota`, `nOtA` y `Nota` son tres nombres distintos.

Consejo.

Utilice el sistema **camelCase** para nombrar sus variables. Este consiste en escribir en mayúscula la primera letra de cada palabra del nombre de la variable, exceptuando la primera que va en minúscula. Por ejemplo:

```
cantidadDeAlumnos = 42
pajarosPorArbol = 3
nivelDeHambre = 'Alto'
```



Error común de programación.

Utilizar un número para comenzar el nombre de una variable.

[3]Existen otros lenguajes, como Java, donde las variables tienen un tipo explícito y solamente pueden guardar valores del tipo al que pertenecen.

Las únicas excepciones a los identificadores posibles son las **palabras reservadas** de cada lenguaje. La lista completa de palabras reservadas en Python es la siguiente:

- `and`
- `as`
- `assert`
- `break`
- `class`
- `continue`
- `def`
- `del`
- `elif`
- `else`
- `except`
- `exec`
- `finally`
- `for`
- `from`
- `global`
- `if`
- `import`
- `in`
- `is`
- `lambda`
- `not`
- `or`
- `pass`
- `print`
- `raise`
- `return`
- `try`
- `while`
- `with`
- `yield`

4.3. Tipos de datos

En Python, el tipo de una variable está implícito y se deduce del valor que guarda. Si tenemos una variable `nota` que guarda el valor `6.7`, podemos deducir que su tipo es numérico.

Existen 5 tipos de datos predefinidos en Python: número, string, lista, tupla y diccionario. Los números pueden ser enteros o reales. Un string se utiliza para guardar texto. Las listas, tuplas y diccionarios se utilizan para guardar conjuntos de valores.

Adicionalmente, existen dos constantes `True` y `False`, que se utilizan para representar al verdadero y falso de la lógica proposicional.

Es importante tener en cuenta el tipo de dato de cada variable, porque de este dependerán las operaciones que se puedan realizar con ella. Por ejemplo, es posible dividir dos variables de tipo numérico, pero no es posible dividir dos variables de tipo string.

4.4. Asignación de valor a una variable

Para dar valor a una variable, se escribe el nombre de la variable, seguido de un signo igual y, por último, se escribe el valor que tomará. Por ejemplo:

```
nombre = "Bernardita Francisca"
apellido = 'Fuentes Barros'
hayFiesta = True
cantidad_de_preguntas = 5
notaDeAprobacion = 3.95
```

De este ejemplo, podemos notar algunas reglas importantes. Los nombres de variable se escriben directamente, mientras que un texto para guardar (un `string`) se debe encerrar en comillas. Estas pueden ser comillas simples (`'`) o dobles (`"`). Los números con decimales se deben escribir utilizando un punto para separar la parte decimal.

También es posible copiar el valor de una variable a otra. Por ejemplo:

```
cantidadDeLibros = 7
cantidadDePersonas = cantidadDeLibros
```

En el fragmento de código anterior, la variable `cantidadDePersonas` toma el valor `7`, ya que este era el valor de la variable `cantidadDeLibros`. Es muy importante recordar que una variable se puede utilizar al lado derecho de una asignación solamente si ya tenía un valor asignado previamente.

El valor asignado a una variable se guarda en esta hasta que se le asigne un valor distinto. El valor de una variable solamente se puede cambiar a través de una asignación.

Error común de programación.

Invertir el orden de asignación, escribiendo el nombre de la variable que toma un nuevo valor al lado derecho.

4.5. Comentarios

Aunque el propósito de un lenguaje de programación es describir un algoritmo de tal forma que un computador lo pueda interpretar correctamente, es posible que el código sea complejo o que su propósito no sea obvio de deducir. Para mitigar este problema se pueden escribir comentarios en el código.

En Python, un comentario empieza con el signo de almohadilla o gato (`#`) y termina al final de la línea[4]. El computador ignorará todos los comentarios que encuentre en el código, su objetivo es ayudar a las personas que leen o escriben el código. Por ejemplo:

```
# Vamos a programar una colección de libros
cantidad = 5 # la variable cantidad indica el total de libros registrados
# la siguiente variable indica cuántos libros están prestados actualmente
prestados = 2
```

[4]En otros lenguajes de programación es posible escribir comentarios de varias líneas.

Cuando un comentario es corto es recomendable escribirlo a la derecha de la línea que se está comentando. Para los comentarios de mayor longitud, es recomendable escribirlos arriba del código que explican.

Consejo.

Escriba comentarios que expliquen el objetivo final de un trozo de código, tal como *Calculamos el dígito verificador de un RUT.*
No comente cosas que se puedan deducir inmediatamente de leer el código, como:

```
# Sumamos uno a la variable x
x = x + 1
```

4.6. Sentencias y expresiones

Una sentencia es la unidad básica del código y, en Python, ocupa una línea[5]. Asignar un valor a una variable es un ejemplo de sentencia.

Una expresión es una parte de una sentencia que se debe resolver para obtener un resultado. Se utilizan mayormente para asignar valores a variables y para calcular si alguna condición es verdadera o falsa.

Las expresiones más simples de entender son las aritméticas, por ejemplo, en el siguiente código:

```
total = 2 + 3
```

Declaramos una variable llamada `total` y le asignamos como valor el resultado de la expresión `2+3`, con lo que guarda el valor `5`.

4.6.1. Operadores matemáticos

Los operadores matemáticos nos permiten trabajar con números. En Python están definidas las siguientes operaciones:

- Cambio de signo: `-`.
- Suma: `+`.
- Resta: `-`.
- Multiplicación: `*`.
- Exponenciación: `**`, eleva el primer operando al segundo, el resultado de `2**3` es `8`.
- División: `/`, se debe tener cuidado con el tipo de las variables. Si ambas son enteras, el resultado será entero, truncando la parte decimal; mientras que si una o ambas son con decimales, el resultado tendrá decimales.

[5] En otros lenguajes de programación existen sentencias que ocupan varias líneas

- División entera: `//`, entrega la parte entera del resultado de la división, descartando la parte decimal independientemente de si sus variables numéricas son enteras o tenían decimales. El resultado de `15.5//3` es `5`.
- Módulo: `%`, resto de la división entera.

Error común de programación.

Olvidar que la división de dos números enteros tiene como resultado un número entero, el código:

```
print 23/5
```

Imprime `4` en la consola.

Es posible combinar varios operadores en una misma expresión, utilizando entre ellos variables y números. También se pueden agrupar operaciones entre paréntesis. Ejemplos de uso:

```
a = 2
b = 3
c = a*b + 5  # c toma el valor 11
b = a**c     # b toma el valor 2048
d = b%10     # d toma el valor 8
e = (2*3 + 5**c)/a + b*3 - d  # e toma el valor 24420201
```

Error común de programación.

Utilizar paréntesis de tipos distintos a `(` y `)` para agrupar expresiones. Los símbolos `[`, `]`, `{` y `}` tienen otro propósito en Python.

4.6.2. Operadores lógicos

Los operadores lógicos tienen en común que su respuesta siempre es `True` (verdadera) o `False` (falsa). Las operacioes lógicas son:

- Conjunción: `and`, tiene resultado `True` si ambos operandos son verdaderos y `False` en otro caso.
- Disyunción: `or`, tiene resultado `True` si uno o ambos operandos son verdaderos y `False` si ambos son falsos.
- Negación: `not`, invierte el valor de verdad del operando.

Al igual que los otros operadores, es posible combinar varios en una misma expresión, utilizando entre ellos variables y números. También se pueden agrupar operaciones entre paréntesis. Ejemplos de uso:

```
a = True
b = False
c = a and b # c toma el valor False
d = a or b  # d toma el valor True
e = (a and (b or c or d)) or (not a) # e toma el valor True
```

Consejo.

Si no está seguro del orden en el que se aplicarán los operadores, agrupe operaciones con paréntesis hasta que no quede ambigüedad.

4.6.3. Operadores de comparación

Al igual que los operadores lógicos, la respuesta de los operadores de comparación es `True` o `False`. Las operaciones de comparación son:

- Igual: `==`.
- Distinto[6]: `!=`.
- Menor que: `<`.
- Menor o igual que: `<=`.
- Mayor que: `>`.
- Mayor o igual que: `>=`.

Error común de programación.

Utilizar un solo signo `=` para comparar. Recuerde que `==` se utiliza para comparar y que `=` se utiliza para asignar un valor.

Error común de programación.

Invertir el orden de los símbolos de algunos operadores. Recuerde que `=>`, `=<` y `=!` no existen en Python.

Los operadores de comparación se utilizan muchas veces en conjunto con los operadores lógicos, por ejemplo en:

```
edad = 19
dinero = 2500
puedeEntrar = edad >= 18 and dinero >= 5000
```

[6]También se puede utilizar `<>`, pero no se recomienda.

4.6.4. Operadores de texto

Las variables de tipo string son un caso especial y tienen dos operaciones importantes:

- Concatenación: `+`, une dos elementos de tipo string, manteniendo el orden entre ellos.
- Multiplicación por un entero: `*`, al multiplicar un string por un número entero se obtiene un nuevo string que consiste en el texto del string que se está multiplicando repetido tantas veces como el valor del número. Si el número es negativo o cero, el resultado será un string vacío.

Ejemplos de uso

```
a = 'h'
b = 'ola'
c = a + b # c toma el valor 'hola'
d = b * 4 # d toma el valor olaolaolaola
e = a + a + a + b*2 + c # e toma el valor 'hhholaolahola'
```

4.6.5. Operadores de asignación abreviados

Dentro del código es común cambiar el valor de una variable basándose en su valor original. Por ejemplo duplicar el valor de una variable, dividir su valor por 10 o sumarle una unidad. Con lo que hemos visto hasta aquí podemos realizar estas operaciones de la siguiente forma:

```
# valores originales
a = 123
b = 456
c = 789
# modificamos los valores en base al original
a = a * 2   # a toma el valor 246
b = b / 10  # b toma el valor 45
c = c + 1   # c toma el valor 790
```

Para estas variables el código es simple de leer, pero podemos encontrarnos en una situación como:

```
largoCamino = largoCaminos + 5
```

¿Notó que no se utiliza la misma variable a ambos lados de la asignación? Si esto es por diseño de nuestro código no hay problema, pero si queríamos modificar el valor de una variable en base a sí misma y utilizamos otra con un nombre similar tendremos problemas. Para evitar estos errores y para no tener que escribir el nombre de la variable dos veces se usan los operadores de asignación abreviados:

```
largoCamino += 5
```

En este caso la variable `largoCamino` aumenta su valor en 5 (estamos asumiendo que la variable ya tenía un valor inicial definido previamente y que es de tipo numérico).

Para utilizar un operador de asignación abreviado se escribe primero el nombre de la variable, luego el símbolo de la operación a realizar seguido de un signo igual y por último el valor que se utilizará para el segundo operando de la operación. Por ejemplo:

```
# valores originales
a = 123
b = 456
c = 789
# modificamos los valores en base al original
a *= 2   # a toma el valor 246
b /= 10  # b toma el valor 45
c += 1   # c toma el valor 790
```

Los operadores que pueden utilizarse para una asignación abreviada son: `+=`, `-=`, `*=`, `/=`, `%=`, `**=` y `//=`.

4.7. Salida

Habiendo visto variables y expresiones, ya podemos empezar a escribir algunos programas. Por ejemplo, podríamos escribir:

```
x = 23
y = 42
z = y ** x
```

En el programa anterior calculamos el valor de 42^{23}, pero al ejecutar este programa nadie puede ver el resultado. La **salida** de un programa consiste en los resultados que este produce. Existen distintas formas de generar esta salida, puede ser en texto o dibujos a través de una impresora, alguna señal inalámbrica a través de una antena, texto o imágenes en una pantalla, sonidos, señales eléctricas que lleguen a otra máquina, etc.

Nos concentraremos, por ahora, en mostrar mensajes de texto en la pantalla del computador. Más específicamente en la consola de Python. Para mostrar un mensaje en la consola utilizamos la instrucción `print`. Esta es una instrucción básica de Python y una palabra reservada, es decir, no la podemos utilizar como nombre de alguna variable. Después de escribir `print` debemos indicar lo que queremos mostrar, esto puede ser un texto (envuelto entre comillas), un número o el nombre de alguna variable. Por ejemplo:

```
x = 23
y = 42
z = y ** x
print 'El resultado es:'
print z
```

Ahora el programa sí muestra mensajes al usuario, en particular muestra:

```
El resultado es:
216139269415798008294225812728452218888
```

Nótese que el resultado de cada instrucción `print` se muestra en una línea por separado. Si queremos que la siguiente instrucción se muestre en la misma línea, debemos escribir una coma después de lo que queremos imprimir. Por ejemplo:

```
x = 23
y = 42
z = y ** x
```

```
print 'El resultado es:',
print z
```

Muestra como salida:

```
El resultado es: 216139269415798008294225812728452218888
```

Nótese que ahora la salida está en una sola línea, pero que `print` agrega un espacio en blanco entre las dos instrucciones.

También es posible resumir las dos instrucciones anteriores en una sola de la siguiente forma:

```
x = 23
y = 42
z = y ** x
print 'El resultado es:', z
```

La instrucción `print` permite que le pasemos muchas cosas a mostrar, separadas por comas.

Error común de programación.

Escribir el nombre de una variable entre comillas cuando se quiere imprimir su valor.

```
x = 'hola'
print 'x' # imprime x
print x   # imprime hola
```

Recuerde que todo lo que está entre comillas es texto y se muestra tal como está.

4.8. Entrada

Ahora ya podemos escribir un programa que calcule el área de un círculo, sabiendo su radio. Por ejemplo:

```
radio = 23
area = 3.14 * radio**2
print 'El área del círculo es:', area
```

Pero si quisiéramos calcular el área de un círculo de otro radio tendríamos que escribir otro programa. Para resolver esto podemos pedir al usuario que ingrese un valor, guardarlo en una variable y utilizarlo. Para esto se utiliza la función `raw_input()`. Por ejemplo:

```
print 'Escribe tu nombre.'
nombre = raw_input()
print 'Hola, ', nombre
```

Al ejecutar este programa, se mostrará el texto `Hola,` seguido del nombre que haya escrito el usuario del programa.

La función `raw_input()` lee lo que escribe el usuario hasta que este escribe un salto de lína (apreta la tecla *Enter* en el teclado) y entrega el texto escrito como un `string` para que podamos guardarlo en una variable.

Si queremos convertir el texto escrito por el usuario en un número tenemos que utilizar la función `int()` o la función `float()`, dependiendo de si queremos transformarlo a un número entero o con decimales respectivamente. Con esto podemos reescribir el programa para que calcule el área de cualquier círculo:

```
print 'Escribe el radio del círculo.'
radioComoString = raw_input()
radio = float(radioComoString)
area = 3.14 * radio**2
print 'El área del círculo es:', area
```

Error común de programación.

No convertir un `string` a número antes de utilizarlo en alguna operación, por ejemplo:

```
print 'Escribe un número.'
x = raw_input()
print 'El triple de', x, 'es', 3*x
```

Si el usuario hubiese escrito `5`, nuestro programa mostraría:

```
El triple de 5 es 555
```

Es posible también escribir un mensaje para el usuario adentro de los paréntesis de la función `raw_input()`. Con esto no es necesario utilizar un `print` para indicar al usuario lo que debe ingresar, por ejemplo:

```
radio = float(raw_input('Escribe el radio del círculo.'))
area = 3.14 * radio**2
print 'El área del círculo es:', area
```

4.9. Ejercicios

4.9.1. Básicos

1. Escriba un programa que escriba `Hola, mundo.` en la consola.
2. Escriba un programa que calcule 15^{15} y muestre el resultado en la consola.

4.9.2. Sugeridos

1. Escriba un programa que calcule 15^{15} y muestre el resultado en la consola. Su programa debe tener una sola línea de código.

2. Escriba un programa que pida un número al usuario y muestre `True` si este es par o `False` si es impar.

3. Escriba un programa que pida al usuario los valores de radio y altura de un cono y muestre en consola el volumen del cono. Recuerde que el volumen de un cono se puede calcular como $V = \frac{\pi \cdot r^2 \cdot h}{3}$, donde r es el radio de la base y h es la altura.

4. Escriba un programa que pregunte al usuario su año de nacimiento y el año actual. Imprima la edad del usuario asumiendo que ya estuvo de cumpleaños este año.

5. Escriba un programa que pida al usuario un año e imprima `True` si este es bisiesto o `False` si no lo es. Un año es bisiesto si es divisible por cuatro, a menos que sea divisible por 100. Dentro de los años divisibles por 100, solamente son bisiestos los que son divisibles por 400.

4.9.3. Desafiantes

1. Escriba un programa que pida al usuario todas sus notas en un curso de la universidad a su elección y calcule su promedio final. Utilice el programa del curso para determinar el cálculo de la nota.

2. Escriba un programa que pida al usuario todas sus notas en un curso de la universidad a su elección, exceptuando la nota del examen. Calcule y muestre la nota que necesita el alumno en el examen para aprobar. Utilice el programa del curso para determinar el cálculo de la nota.

Capítulo 5

Generación de números aleatorios

5.1. Necesidad

Para simular procesos del mundo real, necesitamos poder generar valores aleatorios con facilidad. Por ejemplo para determinar el resultado de lanzar un dado, sacar una carta de una baraja o determinar a qué hora llega el diario a una casa por la mañana.

Para resolver esto Python provee funciones que permiten generar valores aleatorios[1] con distintas distribuciones de probabilidad.

5.2. La librería `random`

Para generar números aleatorios en Python, debemos importar la librería correspondiente. Para esto escribimos, al principio de nuestro archivo de código, la siguiente línea:

```
from numpy import random
```

En este libro utilizaremos la librería `random` incluida en `NumPy`. Tenga en cuenta que existen otras implementaciones de `random`, que se comportan ligeramente diferente a esta y que no usaremos.

Quienes hayan instalado solamente la versión básica de Python, deberán descargar e instalar ahora el módulo `Numpy`, este es gratuito y se puede descargar desde su página oficial[2].

5.3. Generar números enteros

Para generar un número entero al azar utilizamos la función `random.randint(max)`, esta nos entrega como resultado un número entre 0 y `max`, sin incluir `max`. Por ejemplo, para obtener un número entre 0 y 7 y guardarlo en una variable `x`, podemos utilizar el siguiente código:

[1] En realidad existe un algoritmo definido que genera estos números, por lo que son pseudoaleatorios.
[2] `http://www.numpy.org/`

```
from numpy import random
x = random.randint(8)
```

Si queremos generar números aleatorios entre dos valores conocidos, tenemos que especificar el mínimo y el máximo como parámetros de la función `random.randint(min, max)`. El mínimo se incluye dentro de los resultados posibles, pero el máximo no. De esta forma, para generar un número al azar y que el resultado sea 4, 5, 6, 7, 8 o 9, el código sería:

```
from numpy import random
resultado = random.randint(4, 10)
```

Error común de programación.

Olvidar que la cota superior de `random.randint()` no se incluye en el conjunto de resultados posibles.

5.4. Generar números decimales

Para generar un número aleatorio con decimales, Python (de nuevo a través de NumPy) provee la función `random.random()`, esta nos entrega como resultado un número entre 0 y 1. Si queremos generar números con decimales en un rango determinado, tenemos varias opciones. Por ejemplo, si queremos obtener un número entre 1 y 7 con decimales, podríamos generarlo de varias maneras:

```
from numpy import random

# opción 1, generar la parte entera entre 1 y 6, luego sumar decimales entre 0 y 1
forma1 = random.randint(1, 7) + random.random()

# opción 2, sumar la base y el resultado de una multiplicación que dará entre 0 y 6
forma2 = 1 + random.random()*6
```

Error común de programación.

Tratar de poner un número como parámetro en `random.random()`. Esto genera un arreglo (tema que se ve más adelante) de con tantos número como se indique. Pruebe ejecutar el código:

```
from numpy import random
x = random.random(5)
print x
```

5.5. Calcular eventos con distinta probabilidad

Es común tener que generar eventos con distinta probabilidad de aparición. Por ejemplo, si queremos representar el clima de un día de primavera, podemos suponer que será soleado el 80 % de los casos, estará nublado en un 10 % de los casos, con lluvias suaves en un 7 % y con lluvias fuertes en el 3 % restante. Para pasar esto a código, generamos un número al azar entre 0 y 1 y revisamos en qué categoría entra su resultado.

```
from numpy import random

resultado = random.random()
if resultado < 0.8:
    print 'Está soleado.'
elif resultado < 0.9:
    print 'Esta nublado.'
elif resultado < 0.97:
    print 'Está lloviendo suavemente.'
else:
    print 'La lluvia es fuerte.'
```

5.6. Ejercicios

5.6.1. Básicos

1. Escriba un programa que reparta dos cartas al azar de una baraja inglesa al usuario. Es posible que las dos cartas sean iguales.

2. Escriba un programa que simule el lanzamiento de dos dados de 6 caras. Su programa debe mostrar el resultado de cada dado y la suma de ellos.

5.6.2. Sugeridos

1. Se quiere simular una máquina tragamonedas. La máquina tiene 3 rodillos con 10 símbolos distintos en cada uno. El jugador gana si los tres son iguales y pierde si hay dos o tres distintos. Escriba un programa que simule un juego en la máquina, mostrando el resultado de los tres rodillos en la consola. Además imprima en la consola `True` si el jugador gana o `False` si pierde.

5.6.3. Desafiantes

1. Escriba un programa que simule una máquina tragamonedas. Una de las más conocidas es la *Liberty Bell*, que está compuesta por 3 rodillos con 10 símbolos pintados en cada rodillo. En total tenía 5 símbolos diferentes: herraduras de caballos, diamantes, espadas, corazones y una Campana de la Libertad. Cada vez que el jugador insertaba una moneda, los rodillos giraban y si todos terminaban con el mismo símbolo, el jugador ganaba. Decida el valor a pagar por cada símbolo y la cantidad de veces que aparece cada símbolo en cada rodillo, asegurando que cada rodillo tenga por lo menos una copia de cada símbolo.

Capítulo 6

Control de flujo - condicionales

6.1. Introducción

Hasta ahora nuestros programas han sido lineales, es decir, realizan siempre la misma secuencia de instrucciones sin la posibilidad saltarse algunas de ellas o de ejecutar algunas operaciones dependiendo de la entrada que leemos.

Dentro de los diagramas de flujo vimos que era posible tomar desiciones utilizando preguntas con respuestas de sí o no. En Pyhton (y en todos los otros lenguajes de programación) existen estructuras para controlar la ejecución condicional de algunas instrucciones.

6.2. El bloque `if`

El bloque más básico para el control de flujo en Pyhton se llama `if` y su forma es la siguiente:

```
if condición:
    # instrucciones a ejecutar si la condición es verdadera
    # pueden existir muchas instrucciones a ejecutar
```

Donde `condición` es una variable lógica, es decir, que guarda `True` o `False` o cualquier expresión que tenga como resultado `True` o `False`. Después de la condición vienen dos puntos que indican el inicio del bloque.

Error común de programación.

Escribir la palabra reservada `if` con mayúscula. Recuerde que en Python se diferencian mayúsculas de minúsculas, por lo que `if` es una palabra reservada del lenguaje, mientras que `If`, `iF` e `IF` son posibles nombres de variables.

Dentro del bloque de un `if` puede haber cualquier número y tipo de instrucciones, incluyendo asignaciones, impresión de mensajes en la consola, pedir datos al usuario, etc. Estas instrucciones se ejecutan si y solo si la condición es verdadera. Por ejemplo:

```
print 'Escribe un número.'
x = float(raw_input())
if x < 0:
    print 'Tu número es negativo.'
    x = -x
    print 'Tu número ahora es positivo y vale', x
```

Todas las líneas dentro del bloque de un `if` deben mantener la misma indentación (la misma distancia desde el margen izquierdo). Si esta alineación se rompe el código no se podrá ejecutar. El bloque `if` termina cuando la indentación vuelve al margen original.

```
a = int(raw_input('Escriba un número: '))
if a%2==0:
    print 'El número es impar.'
    print 'Esta línea también es parte del if.'
    print 'Todavía estamos dentro del if.'
print 'Esta línea está fuera del if.'
```

Se debe ser cuidadoso al crear nuevas variables dentro del bloque de un `if`, por ejemplo:

```
a = int(raw_input('Escriba un número: '))
if a>100:
    x = 5
print 'La nueva variable vale', x
```

En el código anterior, la variable `x` existe solamente si la condición del `if` es verdadera, es decir, si el usuario ingresa un número mayor que 100. En otro caso, la variable `x` no existirá cuando se trate de imprimir lo que nos llevará al siguiente error:

```
Traceback (most recent call last):
  File "C:\Users\User\Desktop\ejemplos.py", line 4, in <module>
    print 'La nueva variable vale', x
NameError: name 'x' is not defined
```

6.3. El bloque `else`

Es común encontrar casos donde, dependiendo de una condición, queremos ejecutar un trozo de código u otro. Por ejemplo, para determinar si un número es par:

```
x = int(raw_input('Escriba un número: '))
if x%2 == 0:
    print 'Tu número es par.'
if x%2 != 0:
    print 'Tu número es impar.'
```

En este caso es simple escribir manualmente la negación lógica de la condición original, pero si nuestro código fuera algo del tipo `if a and b or (c and not d) or x < -7`, escribir la negación de la condición puede causar errores. Para evitar esto existe el bloque `else`.

Un bloque `else` puede existir solamente a continuación de un bloque `if` y su contenido se ejecutará si y solo si la condición del `if` es falsa. Es importante notar que un bloque `else`

nunca lleva una condición propia, siempre utiliza la negación de la condición del `if` al que complementa.

De esta forma, nuestro ejemplo se puede reescribir de la siguiente manera:

```
x = int(raw_input('Escriba un número: '))
if x%2 == 0:
    print 'Tu número es par.'
else:
    print 'Tu número es impar.'
```

Error común de programación.

Escribir una condición en un bloque `else`. El siguiente código no funciona:

```
x = int(raw_input('Escriba un número: '))
if x%2 == 0:
    print 'Tu número es par.'
else x%2 != 0:
    print 'Tu número es impar.'
```

6.4. Combinación de `if` y `else` encadenados

En varias situaciones, dependiendo del valor de una variable queremos elegir entre más de dos opciones. Es posible encadenar varios bloques `if-else` utilizando el bloque `elif` para resolver esto de la siguiente forma:

```
x = int(raw_input('Escriba un número: '))
if x < 0:
    print 'Tu número es negativo.'
elif x > 0:
    print 'Tu número es positivo.'
else:
    print 'Tu número es cero.'
```

También es posible reescribir el código anterior sin utilizar el bloque `elif`, agrupando los bloques de la siguiente forma:

```
x = int(raw_input('Escriba un número: '))
if x < 0:
    print 'Tu número es negativo.'
else:
    if x > 0:
        print 'Tu número es positivo.'
    else:
        print 'Tu número es cero.'
```

6.5. Ejercicios

6.5.1. Básicos

1. Escriba un programa que pida un número al usuario y muestre un mensaje indicando si es divisible por 7 o no.

2. Escriba un programa que pida al usuario su edad y le diga si puede o no comprar bebidas alcohólicas.

6.5.2. Sugeridos

1. Escriba un programa que pida cuatro números al usuario y muestre en la consola el mayor y el menor de ellos.

2. Escriba un programa que pida un número entre 1 y 7 al usuario y escriba en la consola "Hola"tantas veces como indique el número[1]. Si el usuario escribe un número fuera del rango indicado, escriba un mensaje al usuario diciéndole que se equivocó.

3. Para avanzar en Monopoly, se lanzan dos dados de 6 caras y se mueve tantas casillas como indiquen estos. Pero si el resultado de la tirada es dobles, es decir, ambos dados tienen el mismo valor, el jugador lanza nuevamente y vuelve a avanzar. Si en la segunda tirada también obtiene dobles, vuelve a tirar una tercera vez. Si en la tercera tirada vuelve a obtener dobles, va a la cárcel y pierde su turno. Escriba un programa que genere una tirada para avanzar en Monopoly, indicando el resultado de cada tirada de dados, cada vez que se obtenga dobles y mostrando un mensaje final indicando el total avanzado o si el jugador va a la cárcel.

6.5.3. Desafiantes

1. Se quiere simular una carrera de autos. Se sabe que la pista es recta y mide 5 kilómetros. En la carrera participan 3 autos, que pueden empezar en cualquier punto de la pista, ambos parten en reposo. Cada auto tiene una aceleración constante y una velocidad máxima que no puede superar. Escriba un programa que pida al usuario la posición inicial, aceleración y velocidad máxima de cada auto. Luego escriba qué auto llega primero a la meta. Si el usuario ingresa una posición inicial fuera de la pista o una aceleración o velocidad máxima negativa, debe terminar el programa escribiendo un mensaje en la consola indicando al usuario en qué se equivocó.

2. Escriba un programa que pida al usuario todas las notas que obtuvo en el curso y determine si aprueba o reprueba. Revise el programa del curso para determinar la cantidad de notas a pedir, cuántas se eliminan, qué ponderaciones tiene cada una y qué condiciones son necesarias para aprobar.

[1] Recuerde que debe resolver este problema sin utilizar ciclos.

Capítulo 7

Control de flujo - ciclos

7.1. Necesidad

Ya conocemos las bases para construir programas simples, con flujos que incluyen decisiones. Pero todavía nos falta algo importante, y es para lo que los computadores son más usados: tareas repetitivas.

Hasta ahora la única forma que teníamos de repetir una parte del programa era escribir el código varias veces (o copiarlo y pegarlo varias veces). Esto funciona cuando conocemos la cantidad exacta de repeticiones, pero genera programas muy extensos y difíciles de mantener.

Los ciclos nos permitirán escribir programas que repitan secciones del código: es lo que en los diagramas de flujo dibujábamos como flechas que vuelven hacia atrás en el diagrama.

7.2. El ciclo `while`

El bloque `while`, que traducido significa literalmente **mientras**, se construye muy parecido al `if`, de la siguiente manera:

```
while condición:
    # sentencias a ejecutar mientras la condición sea verdadera
```

Donde `condición` es una expresión que tiene como resutaldo `True` o `False`. Las sentencias que se encuentren indentadas "adentro" del ciclo se ejecutarán repetitivamente mientras la evaluación de `condición` resulte verdadera.

Dentro del bloque se puede escribir cualquier tipo de sentencia, incluyendo construcciones como `if` o incluso otros `while`. Se pueden declarar variables, asignar valores, imprimir mensajes, etc. Por ejemplo:

```
print 'Ingrese un número positivo.'
x = int(raw_input())
while x <= 0:
    print '¡El número', x, 'no es positivo! Ingrésalo de nuevo.'
    x = int(raw_input())
print '¡El número', x, 'sí es positivo!'
```

En otras palabras, cuando un programa llega a un `while` se revisa si la condición es verdadera o falsa. En el caso que sea verdadera, se ejecuta todo lo que está adentro del

bloque y se vuelve al principio, donde se revisa la condición nuevamente. El bloque se sigue ejecutando hasta que la condición sea falsa.

Si la condición del ciclo es falsa la primera vez que se revisa, el bloque no se ejecutará ninguna vez, de la misma forma que un `if` cuya condición sea falsa.

Es muy importante recordar que para que un programa termine, las condiciones de todos y cada uno de sus ciclos deben hacerse falsas en algún momento. Por ejemplo, el siguiente programa no terminará nunca:

```
print 'Inicio del programa.'
while 2 < 5:
    print 'Hola'
print 'Esto no se va a imprimir nunca.'
```

Error común de programación.

Olvidar cambiar el valor de la variable que determina la condición del `while`, generando un ciclo infinito como:

```
# intentando imprimir los primeros 10 cuadrados
i = 1
while i <= 10:
    print i**2
```

7.3. Las sentencias `break` y `continue`

Hay dos sentecias adicionales que nos pueden ayudar en la ejecución de los `while`, estas son las sentencias `break` y `continue`.

`break` simplemente termina la ejecución del ciclo, o, como significa literalmente, la *rompe*. Esta sentencia habitualmente se usa dentro de un `if` en alguna parte del bloque de código. Su gracia es que rompe el ciclo exactamente en el lugar donde está puesta, saltándose las instrucciones que faltan para terminar el bloque, y continuando el flujo del programa en lo que venga después del ciclo. Un ejemplo:

```
valor = 0 # damos un valor inicial por si acaso
while True: # este ciclo parece que no va a terminar nunca
    valor = raw_input('Dame un valor: ')
    print 'Escribe 1 si quieres usar el valor ' + str(valor)
    decision = int(raw_input())
    if decision == 1:
        break
    print 'OK, voy a preguntar de nuevo...'
print 'Bien, el valor final es", valor
```

Por otro lado, `continue` sirve para continuar desde el principio el ciclo, es decir, se salta las sentencias que quedan para el fin del bloque y vuelve directamente hasta la condición del ciclo, que vuelve a evaluarse para decidir si se debe ejecutar de nuevo el bloque de código.

Por ejemplo, para imprimir todos los múltiplos de 7 entre 1 y 100, podemos usar el siguiente ciclo `for`:

```
n = 0
while n < 100:
    n += 1
    if n %7 != 0:
        continue
    print n, 'es múltiplo de 7.'
```

Error común de programación.

Poner una instrucción `continue` antes de cambiar la variable de control de un ciclo `while`. Con esto la variable no cambiará su valor si se ejecuta el `continue` y el ciclo no terminará nunca. Por ejemplo:

```
# intentando imprimir solamente los números impares
i = 1
while i < 100:
    if n %2 == 0:
        continue
    print n
    n += 1
```

7.4. Ejercicios

7.4.1. Básicos

1. Escriba un programa que imprima en la consola los números enteros del 1 al 10.000.

2. Escriba un programa que imprima en la consola los números enteros del 1 al 10.000, junto con el total de la suma acumulada hasta cada número. Las primeras líneas que deberá mostrar su programa son:

```
1 1
2 3
3 6
4 10
5 15
6 21
```

7.4.2. Sugeridos

1. Escriba un programa que imprima los números divisibles por 7 menores a 1.000.000. Pruebe resolver esto con `if` y sin `if`.

2. Escriba un programa que pida al usuario un número y dibuje un cuadrado utilizando asteriscos (*), con tantos asteriscos por lado como indique el usuario. Exija al usuario que ingrese un número positivo. Resuelva esto para dibujar cuadrados con y sin relleno. Por ejemplo, si el usuario ingresa 5, usted deberá imprimir alguno de los siguientes dos cuadrados:

```
*****          *****
*   *          *****
*   *          *****
*   *          *****
*****          *****
```

7.4.3. Desafiantes

1. Escriba un programa que imprima los números del 1 al 10.000, uno por línea. Si un número es divisible por 3 a su lado se debe imprimir `Fizz`. Si es divisible por 5 a su lado se debe imprimir `Buzz`. Si es divisible por ambos debe decir `FizzBuzz`.

2. Escriba un programa que pida un número al usuario y escriba un mensaje en la consola indicando si este es primo o compuesto.

3. La ruleta es un juego de apuestas que se encuentra en la gran mayoría de los casinos. Este juego permite apostar a un número, un color, una docena, columna o a los pares o impares. Cada tipo de apuesta paga de forma diferente, según la siguiente tabla:

Tipo de apuesta	**Pago**
Color, par o impar	1 a 1
Columna o docena	2 a 1
Pleno (un número)	35 a 1

La ruleta tiene los números del 0 al 36, ordenados como se muestra en el siguiente tablero:

Escriba un programa que simule un juego de ruleta, el juego debe preguntar al usuario el dinero que trae inicialmente y luego ofrecer rondas de juego mientras el jugador tenga dinero y quiera seguir jugando.

En cada ronda se le pregunta al usuario el valor a apostar (se debe validar que sea menor o igual a la cantidad de dinero actual del jugador), el modo de juego (columna, número, docena o color) y los detalles de este (a qué columna, número, docena o color apuesta). Luego se lanza una bolita aleatoriamente y se informa al jugador si gana o pierde la apuesta.

Capítulo 8

Funciones

8.1. Necesidad

Escribiendo programas encontramos frecuentemente que debemos ejecutar el mismo trozo de código más de una vez. Sabemos que esto se puede resolver utilizando ciclos para la mayoría de los casos. Pero siguen habiendo instancias donde estamos obligados a escribir un trozo de código varias veces en el mismo programa.

En estos casos nos conviene utilizar una función. Estas se comportan de la misma manera que en matemáticas, toman algunos datos de entrada y producen una respuesta en base a ellos.

8.2. Definición

En Python, una función se declara con la palabra reservada `def`, seguida del nombre de la función, a continuación un par de paréntesis donde se indican los nombres de los parámetros que pueda necesitar y finalmente dos puntos. En las siguientes líneas va el cuerpo de la función, indentado de la misma forma que un `if` o un `while`. Por ejemplo:

```
def mayor(x, y):
    if x > y:
        return x
    else:
        return y
```

En el cuerpo de una función puede encontrarse cualquier parte del código que ya hemos visto. Se pueden declarar nuevas variables, realizar cálculos sobre ellas, utilizar estructuras de control de flujo como `if` o `while`.

Toda función debe tener un nombre. Este sigue las mismas reglas que tienen los nombres de variables (solo letras, números y el símbolo `_`, no puede empezar con un número). **No puede haber más de una función con el mismo nombre, ni tampoco una función con el mismo nombre que una variable.** En el caso de declarar varias funciones con el mismo nombre, solamente la última definición se utiliza.

Error común de programación.

Declarar dos o más funciones con el mismo nombre.

Error común de programación.

Declarar una función con el mismo nombre de una variable o una variable con el mismo nombre de una función.

Para utilizar una función, primero debemos definirla. Luego, simplemente escribimos el nombre de la función, seguido de paréntesis, y dentro de los paréntesis escribimos los parámetros que esta necesita[1]. Por ejemplo:

```
def mayor(x, y):
    if x > y:
        return x
    else:
        return y

a = int(raw_input('Escriba un número: '))
b = int(raw_input('Escriba otro número: '))

m = mayor(a, b)
print 'El mayor es', m

print 'Entre 2 y 8, el mayor es', mayor(2, 8)
```

[1]Dentro de este libro ya hemos utilizado varias funciones que vienen predefinidas en Python, como `raw_input()`, `int()` y `str()`.

Nótese que los nombres de los parámetros de una función son independientes de las variables que se pasan a esta para ejecutarla. También es posible llamar a una función pasando constantes como parámetros.

8.3. Retorno

La ejecución de una función termina cuando se llega a una línea que empieza con la palabra reservada `return` o cuando se llega al final del bloque que define su cuerpo, lo que pase primero. Todo el código que sea parte de una función pero que se encuentre después de una sentencia de retorno no se ejecuta. Por ejemplo, al ejecutar el siguiente código:

```
def f(x):
    print 'Empezando la función'
    return x + 3
    print 'Continuando la función'
    return x + 7
    print 'Fin de la función'

a = f(2)
print a
```

En la consola se mostrará:

```
Empezando la función
5
```

Error común de programación.

Escribir código en una función después de una sentencia de retorno.

En muchos lenguajes de programación es obligatorio especificar el tipo de dato que retornará una función. En Python el tipo de dato de cada variable está implícito en el valor que esta guarda, por lo que no se deja explícito el tipo de retorno de una función.

Es posible que una misma función retorne valores de distintos tipos dependiendo de algún parámetro o de alguna condición interna. Por ejemplo:

```
def funcionEspecial(x):
    if x < 5:
        return 42
    elif x == 28:
        return 'Hola'
    else:
        return False
```

El uso de estas funciones no está recomendado, ya que al recibir la respuesta tendríamos que comprobar su tipo antes de operar con ella. En la gran mayoría de los casos esto haría que el código fuera innecesariamente complejo.

Consejo.

Asegure que cada función retorne siempre datos del mismo tipo, esto hará más cómodo su uso.

Por último, es posible que una función no retorne ningún valor. Por ejemplo, podríamos tener una función que permita saludar a un usuario varias veces:

```
def saludar(nombre, repeticiones):
    print 'Inicio del saludo'
    i = 0
    while i < repeticiones:
        print 'Hola,', nombre
        i += 1
    print 'Fin del saludo'
```

En este caso la función simplemente imprimiría mensajes en la consola saludando al usuario varias veces. No existe ninguna sentencia de retorno en este caso.

Cada vez que tratemos de asignar el resultado de una función que no tenga retorno a una variable, la variable tomará el valor `None`. Esta es una constante predefinida en Python que indica que la variable no está guardando nada, ni siquiera un 0 o un `False`.

Consejo.

Escriba un comentario arriba de cada función en su código, detallando el tipo de respuesta que esta entregará. Por ejemplo:

```
# Retorna un número con el área de un cuadrado para el lado recibido
def areaCuadrado(lado):
    return lado**2
```

8.4. Parámetros

Dentro de las funciones que hemos visto, la mayoría necesitan parámetros para calcular un valor final. Es posible tener funciones sin parámetros por ejemplo:

```
from numpy import random

def lanzarDado():
    return random.randint(1, 7)

def saludar():
    print 'Hola, soy tu computador.'
    print 'I don't hate you.'
```

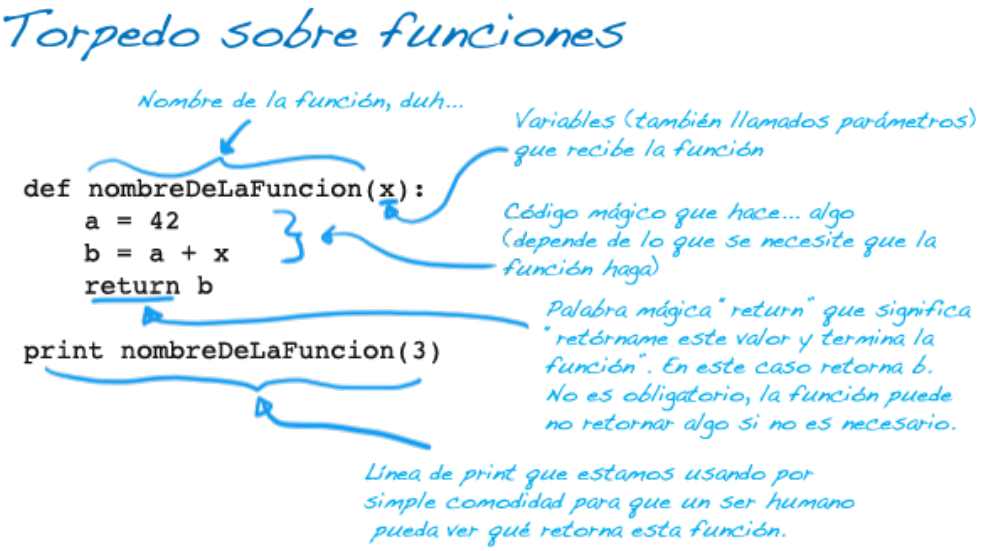
Las funciones que necesitan parámetros deben declararlos en su definición dentro de los paréntesis que van a continuación de nombre, separados por comas. Todos los parámetros de una misma función deben tener nombres diferentes. Dentro del cuerpo de la función se utilizan como variables que ya tienen un valor definido.

Al llamar a una función se le deben pasar tantos parámetros como esta especifique en su definición. Este paso de valores puede ser a través de variables o de valores constantes, escritos directamente en el código. Es mucho más común utilizar variables para dar valor a los paramétros de una función.

El orden en el que se pasan los parámetros influye al ejecutar la función. Por ejemplo:

```
def elevar(a, b):
    return a**b

print elevar(2, 6) # a toma el valor 2. b toma el valor 6.
print elevar(6, 2) # a toma el valor 6. b toma el valor 2.
```



Consejo.

Use nombres autoexplicativos para los parámetros de sus funciones.
Leer `revisarDatos(nombre, edad, altura, peso)` es mucho más cómodo que tratar de interpretar `revisarDatos(a, b, c, d)`.

8.5. Parámetros con valor por defecto

Es posible tener funciones que no necesiten usar parámetros para realizar su objetivo, pero que al recibir algún parámetro ejecutan algo ligeramente distinto. Por ejemplo, podríamos tener una función que salude al usuario de forma genérica, pero que al recibir el nombre del usuario lo incluya en el saludo. Hasta ahora esto lo podemos hacer definiendo dos funciones distintas:

```
def saludarGenerico():
    print 'Buenos días.'

def saludarConNombre(nombre):
    print 'Buenos días, ' + nombre + '.'
```

Para no tener que definir dos funciones que hacen lo mismo, podemos definir que algunos de los parámetros de la función tengan algún valor predefinido que se usa cuando no se pasan valores al ejecutarla:

```
def saludar(nombre = 'persona genérica'):
    print 'Buenos días, ' + nombre + '.'

saludar()
saludar('Diego')
```

Al ejecutar la función sin parámetros, se usa el valor por defecto del parámetro `nombre`, es decir, se imprime `Buenos días, persona genérica.`. Por otro lado, al indicar un nombre específico, se usa ese valor para el parámetro, en el ejemplo `Buenos días, Diego.`.

Incluso, podríamos revisar si a la función se le pasan parámetros o no, agregando un poco más de lógica:

```
def saludar(nombre = ''):
    if nombre == '':
        print 'Buenos días.'
    else:
        print 'Buenos días, ' + nombre + '.'
        print 'Tu nombre tiene', len(nombre), 'letras.'

saludar('Paula')
```

Se debe tener cuidado al definir un valor por defecto a los parámetros de una función, ya que los parámetros que se pasen al ejecutar la función se asignarán **en orden** a los primeros parámetros de la definición. Por ejemplo, al ejecutar:

```
def calcular(x=2, y=15, z=-1):
    if x%2==0:
        return y-5
    elif y > z:
        return z + x
    else:
        return x + y - z

print calcular()
print calcular(3)
print calcular(3, -5)
print calcular(3, -5, 6)
```

Se imprime:

```
10
2
-1
-8
```

8.6. Alcance, variables locales y globales

Hemos visto que las funciones pueden utilizar parámentros. También hemos visto que el nombre de un parámetro es independiente del nombre de la variable que se usa para darle valor. Podemos resumir esto en el siguiente ejemplo:

```
def elevar(a, b):
    return a**b

x = 3
y = 5
z = elevar(x, y)
a = 4
b = 7
c = elevar(a, b)
print z, c
```

En el código anterior, se imprimirá en la consola `243 16384`. Nótese que la función `elevar` se ejecuta correctamente sin importar los nombres de las variables que se le pasan como parámetros.

Con las variables que hemos visto, se pasa una copia del valor para asignar en el parámetro correspondiente. Por lo que el parámetro se puede modificar sin cambiar la variable original[2]. Por ejemplo, al ejecutar el código:

[2]Esto cambiará cuando veamos listas.

```
# Suma dos números, asumiendo que b es positivo
def sumar(a, b):
    while b > 0:
        a += 1
        b -= 1
    return a

a = 4
b = 5
c = sumar(a, b)
print a, b, c
```

Veremos que en la consola se imprime `4 5 9`. Esto es porque cambiar el valor de los parámetros `a` o `b` no cambia el valor de las variables `a` y `b`, ya que los parámetros toman una copia del valor de la variable original.

Dentro de una función podemos declarar nuevas variables, por ejemplo:

```
from numpy import random

def ruletaDeLaSuerte():
    x = random.random()
    respuesta = ''
    if x < .3:
        respuesta = 'Tendrás buena suerte.'
    elif x < .6:
        respuesta = 'Tendrás mala suerte.'
    elif x < .7:
        respuesta = 'Tendrás muy buena suerte.'
    elif x < .8:
        respuesta = 'Tendrás muy mala suerte.'
    elif x < .9:
        respuesta = 'Se cumplirán todos tus deseos.'
    else:
        respuesta = 'Te pasarán todas las cosas malas porque',
        respuesta += 'no compartiste *esa* foto en Facebook.'
    return respuesta

print 'Tu suerte para hoy es:'
print ruletaDeLaSuerte()
```

En este caso, `x` (y también `respuesta`) es una **variable local**, por lo que solamente se puede utilizar adentro de la función donde está declarada. Cualquier intento de utilizarla afuera de la función generará un error parecido a:

```
Traceback (most recent call last):
  File "C:\Users\User\Desktop\ejemplos.py", line 4, in <module>
    print x
NameError: name 'x' is not defined
```

Es posible, pero no recomendable, utilizar dentro de una función variables que esta no tiene definidas como parámetros. Estas son llamadas **variables globales**. Las variables

globales van más allá del alcance de este curso, pero puede leer sobre ellas en `http://gettingstartedwithpython.blogspot.com/2012/05/variable-scope.html` (en inglés).

Consejo.

No declare funciones adentro de otras funciones. La gran mayoría de las veces no van a hacer lo que usted quería decir.

8.7. Librerías

Habiendo visto que las funciones permiten reutilizar el código con facilidad, el paso siguiente es convertir en funciones los trozos de código que más se utilizan, por ejemplo, para pedir un número al usuario en un rango de valores definido:

```
# Pide un número entero al usuario entre los valores
# mínimo y máximo recibidos como parámetros.
def pedirNumero(minimo, maximo):
    print 'Escribe un número entre', minimo, 'y', maximo
    numero = int(raw_input())
    while numero < minimo or numero > maximo:
        print 'Error, valor fuera de rango.'
        print 'Escribe un número entre', minimo, 'y', maximo
        numero = int(raw_input())
    return numero
```

Ahora, teniendo esta función definida (y posiblemente otras más también), podemos utilizarla en varios de nuestros programas. Para esto tenemos dos opciones: podemos simplemente copiarla al inicio de cada nuevo programa que escribamos o podemos crear una librería de funciones comunes.

Una librería es simplemente un archivo de código con varias funciones definidas. Hemos usado ya la librería `numpy` para generar números aleatorios. Para crear una nueva librería escribimos las funciones que consideremos importantes o útiles en un archivo y lo guardamos con extensión `.py` al igual que cualquier otro archivo de código en Python.

Para importar las funciones definidas en una librería debemos asegurarnos que su arhivo de código se encuentre en el mismo directorio que el archivo de código donde la queremos usar. Luego debemos utilizar la instrucción `from import`. Suponiendo que tenemos la función del ejemplo anterior guardada en el archivo `utilidades.py`. En el archivo de código donde queramos utilizarla debemos escribir:

```
from utilidades import pedirNumero
x = pedirNumero(23, 42)
print 'El número ingresado es,' x
```

Nótese que escribimos explícitamente el nombre de la función que queremos importar desde nuestra librería. Si queremos traer más de una función podemos escribir una línea de `import` por cada una o separar los nombres de las funciones por comas:

```
from utilidades import pedirNumero, calcularIVA, calcularIntereses
```

Si queremos importar todas las funciones desde una librería, simplemente escribimos un asterisco en lugar de enumerar las funciones que queremos traer:

```
from utilidades import *
```

Error común de programación.

Definir una función y luego importar una librería que tiene una función con el mismo nombre. Como siempre se usa la última definición de la función, en este caso se utilizaría la que viene en la librería.

8.8. Ejercicios

8.8.1. Básicos

1. Escriba una función que calcule el área de un círculo, recibiendo como parámetro el radio de este. Recuerde que el área se puede calcular como $A = \pi \cdot r^2$.

2. Escriba una función que calcule el perímetro de un círculo, recibiendo como parámetro el radio de este. Recuerde que el perímetro se puede calcular como $P = 2 \cdot \pi \cdot r$.

8.8.2. Sugeridos

1. Escriba una función que reciba un número como parámetro y retorne `True` o `False` dependiendo de si este es primo o compuesto.

2. Escriba una función que reciba como parámetros 4 números y entregue como respuesta el mayor de ellos. Tome en cuenta que algunos o todos podrían tener el mismo valor.

3. Escriba una función que reciba como parámetro un número entero postivo y entregue como resultado el número que se obtiene al invertir el orden de los dígitos del primero.

4. Escriba una función que genere un valor al azar, de manera que el resultado sea 4, 8, 15, 16, 23 o 42.

8.8.3. Desafiantes

1. Escriba una función para calcular coeficientes binomiales. El coeficiente binomial entre n y k está definido como: $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$. Valide que $0 \leq k \leq n$, para números que no cumplan esta propiedad retorne `False`.

2. Escriba su propia librería de funciones matemáticas, incluya funciones para calcular factoriales, funciones estadísticas y determinar si un número es primo, compuesto, perfecto, amigable, triagular, etc.

Capítulo 9

Listas

9.1. Necesidad

Ya sabemos como guardar datos en nuestras variables, pero todavía tenemos algunas limitaciones en nuestros programas. Por ejemplo, supongamos que queremos pedirle al usuario todas las notas de una prueba del curso y ordenarlas para mostrarlas de mayor a menor. Tendríamos que saber de antemano la cantidad de notas, para poder guardar cada una en una variable. Luego, tendríamos que declarar tantas variables como notas se quieran guardar. Finalmente, la maraña de `if` y `else` que tendríamos que generar para poder ordenarlas correctamente sería enorme.

Para resolver este problema podemos usar listas, que nos permitirán guardar muchos datos en una única variable.

9.2. Definición

Una lista es una colección de elementos, que pueden ser de distintos tipos. Una lista, al igual que cualquier otra variable, tiene un nombre y este debe seguir las mismas reglas que los nombres de variables y funciones.

Para usar una lista por primera vez en Python, escribimos su nombre, seguido del signo igual y después un par de corchetes (los símbolos `[]`). Dentro del los corchetes podemos enumerar los elementos iniciales de la lista. Si queremos crear una lista vacía, dejamos los corchetes vacíos. Por ejemplo:

```
lista1 = [] # lista vacía
lista2 = [4, 8, 15, 16, 23, 42] # lista con números
lista3 = ['Daniela', 'Sofía', 'Germán'] # lista con palabras
lista4 = [2, 'Hola', 3.5, 'Rojo', False, True, 0] # lista mixta
```

Consejo.

Utilice solamente un tipo de datos para cada una de sus listas. Esto hará que revisarlas y operar con sus elementos sea más fácil.

Para revisar cuántos elementos se están guardando en una lista podemos utilizar la función `len(lista)`. Pasando como parámetro el nombre de la lista a revisar. Por ejemplo:

```
numeros = [4, 8, 15, 16, 23, 42]
print 'La lista tiene', len(numeros), 'elementos.'
```

Error común de programación.

Recuerde usar solamente los símbolos de `[` y `]` para acceder a los elementos de las listas.
Los símbolos de `(` y `)` se usan para definir los parámetros de una función o para agrupar expresiones.
Los símbolos de `{` y `}` se usarán más adelante para trabajar condiccionarios.

9.3. Acceso a los elementos

Para acceder a un elemento en una lista, se usa el operador `[]`. Se escribe luego del nombre de la variable que contiene la lista, y entre los corchetes ponemos el número de la posición a la que queremos acceder. **Las posiciones de una lista se enumeran desde cero en adelante**. Si una lista tiene 4 elementos, estos estarán en las posiciones 0, 1, 2 y 3. Por ejemplo:

```
lista = [4, 8, 15, 16, 23, 42]
lista[2] = 1 # Reemplazamos el tercer elemento de lista por un 1
# Usamos un elemento como variable para calcular otro valor
lista[2] = lista[0] + 10
lista[1+2] = 0 # Entre los corchetes puede ir cualquier operación de tipo int
print 'El primer elemento es', lista[0]
```

Error común de programación.

Olvidar que el primer elemento de una lista está en la posición cero.

Además es posible acceder a sus elementos utilizando índices negativos, donde el índice -1 indica el último elemento de la lista, -2 el penúltimo y así sucesivamente. Por ejemplo:

```
lista = [4, 8, 15, 16, 23, 42]
x = (lista[0] + lista[-1])/2
print 'El promedio entre el primero y el último es', x
```

Es posible imprimir una lista completa utilizando la instrucción `print`, en la consola se mostrarán todos los elementos de la lista, separados por comas y envueltos en un par de corchetes. Por ejemplo, el código:

```
x = [2,4,3,2,1,"hola"]
print x
```

Mostrará en la consola el texto:

```
[2, 4, 3, 2, 1, 'hola']
```

Debemos tener cuidado al acceder a los elementos de una lista, ya que si tratamos de usar una posición que no existe vamos a producir un error. Para una lista con N elementos, los valores posibles para usar como índices de la lista están en el rango $-N \leq \textit{índice} < N$.

Error común de programación.

Usar un índice fuera del rango permitido para acceder a los elementos de una lista.

9.4. Recorrer una lista

Muy frecuentemente vamos a querer hacer algo con cada uno de los elementos de una lista. Por ejemplo, si queremos imprimir todo el contenido de una lista podemos ejecutar lo siguiente:

```
x = ['hola', 'hoy', 'es', 'jueves']
i = 0 # índice del elemento a mostrar
while i < len(x):
    print x[i]
    i += 1
```

9.5. El ciclo `for`

Una forma más simple de recorrer los elementos de una lista consiste en utilizar un ciclo `for` de la siguiente forma:

```
numeros = [4, 8, 15, 16, 23, 42]
for num in numeros:
    print 'El cubo de', num, 'es', num**3
```

Un ciclo `for` no utiliza una variable auxiliar (la que usábamos para indicar el índice del elemento en un ciclo `while`), lo que mejora la legibilidad del códdigo, pero no nos permite saber en qué posición de la lista estamos en cada ejecución.

Otra aplicación común es sumar todos los números de una lista:

```
numeros = [4, 8, 15, 16, 23, 42]
suma = 0
for num in numeros:
    suma += num
print 'La suma total es', suma
```

Error común de programación.

Confundir la variable de un ciclo `for` con el índice de los elementos de una lista. El código:

```
numeros = [4, 3, 1, 2, 3, 0]
for i in numeros:
    print numeros[i]
```

Mostrará en la consola:

```
3
2
3
1
2
4
```

9.6. Generadores de listas

Como ya vimos, podemos crear listas manualmente escribiendo todos sus elementos, separados por comas, dentro de un par de corchetes y asignando esto a una variable. También podemos crear una lista vacía sin escribir nada dentro de los corchetes:

```
listaVacia = []
```

Existen funciones que entregan como retorno listas con valores específicos, nos concentraremos en las siguientes:

- `range(max)`: genera una lista con números enteros desde `0` hasta `max-1`.
- `range(min, max)`: genera una lista con números enteros desde `min` hasta `max-1`.
- `range(min, max, incremento)`: genera una lista con números enteros empezando en `min`, incrementando cada número en `incremento` y terminando en un número estrictamente menor a `max`.

Ejemplos:

```
lista1 = range(10)          # equivalente a lista1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
lista2 = range(10, 17)      # equivalente a lista2 = [10, 11, 12, 13, 14, 15, 16]
lista3 = range(10, 30, 3)   # equivalente a lista3 = [10, 13, 16, 19, 22, 25, 28]
```

Es común utilizar un ciclo `for` en conjunto con `range()` para repetir un trozo de código una cantidad conocida de veces. Por ejemplo, para imprimir el texto `hola` 100 veces en la consola:

```
for i in range(100):
    print 'hola'
```

9.7. Métodos de listas

Un método es muy similar a una función, permite ejecutar una secuencia de código predefinida y puede utilizar parámetros. La diferencia entre ambos es que un método se debe aplicar a una variable[1], mientras que una función se ejecuta por su cuenta. Algunos métodos útiles de las listas son:

- `append(x)`: agrega el elemento `x` al final de la lista, agrandando en 1 el tamaño de la lista.
- `insert(i, x)`: agrega el elemento `x` en la posición `i` de la lista, desplazando los elementos siguientes en una posición y agrandando en 1 el tamaño de la lista
- `extend(L)`: agrega todos los elementos de la lista `L` al final de la lista actual, agrandando la lista actual en `len(L)` elementos.
- `remove(x)`: elimina la primera aparición del elemento `x` de la lista, achicando el tamaño de la lista en 1.
- `pop()`: elimina el último elemento de la lista, achicando el tamaño de la lista en 1.
- `pop(i)`: elimina el elemento en la posición `i` de la lista, achicando el tamaño de la lista en 1.
- `index(x)`: entrega el valor del índice que ocupa el primer elemento de la lista igual a `x`.
- `count(x)`: entrega la cantidad de veces que se encuentra el elemento `x` dentro de la lista.
- `sort()`: ordena los elementos de la lista de menor a mayor. Si la lista contiene elementos de varios tipos, primero ordena los tipos por orden alfabético y luego ordena los elementos de cada tipo.
- `reverse()`: invierte el orden de los elementos de la lista.

Ejemplos de uso:

[1] Técnicamente se aplican a objetos, pero eso va más allá del alcance de este curso.

```
lista = [2, 4]

lista.append(-10)
lista.append(20)
lista.append(42)
print len(lista)      # imprime 5
print lista           # imprime [2, 4, -10, 20, 42]

lista.sort()
print lista           # imprime [-10, 2, 4, 20, 42]

lista2 = [5, 2, -1]
lista.extend(lista2)
print lista           # imprime [-10, 2, 4, 20, 42, 5, 2, -1]

lista.sort()
lista.reverse()
print lista           # imprime [42, 20, 5, 4, 2, 2, -1, -10]

lista.remove(2)
print lista           # imprime [42, 20, 5, 4, 2, -1, -10]

lista.insert(3, 7)
print lista           # imprime [42, 20, 5, 7, 4, 2, -1, -10]
```

9.8. Sublistas

Es posible obtener sublistas (trozos continuos de elementos de una lista) indicando la posición inicial (incluida) y la posición final (no incluida), separadas por dos puntos (`:`). Ambas posiciones son opcionales, si no se especifica la posición inicial se asume 0 y si no se especifica la posición final se asume `len()` de la lista. Por ejemplo:

```
x = [4, 8, 15, 16, 23, 42]

a = x[2:4]   # a = [15, 16]
b = x[3:]    # b = [16, 23, 42]
c = x[:3]    # c = [4, 8, 15]
d = x[:]     # d es una copia de x (d = [4, 8, 15, 16, 23, 42])
```

También es posible utilizar la notación de sublistas para modificar, agregar y quitar elementos de una lista, por ejemplo:

```
# modificar
x = [4, 8, 15, 16, 23, 42]

x[5] = 0          # x = [4, 8, 15, 16, 23, 0]
x[0:2] = [1, 7]   # x = [1, 7, 15, 16, 23, 0]

# agregar
```

```
x = [4, 8, 15, 16, 23, 42]

x[1:1] = ['a', 'b'] # x = [4, 8, 'a', 'b', 15, 16, 23, 42]
x[6:6] = 'c'        # x = [4, 8, 'a', 'b', 15, 16, 'c', 23, 42]

# quitar
x = [4, 8, 15, 16, 23, 42]

x[2:3] = [] # x = [4, 8, 16, 23, 42]
x[:2] = []  # x = [16, 23, 42]
```

9.9. Listas de listas

Es posible escribir una lista donde cada uno de los elementos guardados sea una lista. Esto se puede hacer directamente al crear la lista o se puede construir al ejecutar el código. Por ejemplo:

```
# lista con 3 elementos, cada elemento es una lista
pares = [[4, 8], [15, 16], [23, 42]]

# lista compleja
lista = []
for i in range(10):
    elemento = []
    for j in range(10):
        elemento.append(i+j)
    lista.append(elemento)
```

Para acceder a los elementos internos de una lista de este tipo debemos indicar el índice que ocupa en cada nivel. Por ejemplo:

```
pares = [[4, 8], [15, 16], [23, 42]]

print pares[1] # imprime [15, 16]
print pares[2][-1] # imprime 42
```

9.10. Listas, tipos por referencia y funciones

Las listas se comportan de manera distinta a los otros tipos de datos que hemos visto. Los tipos `int`, `float`, `str` y `bool` son **tipos básicos**, estos se trabajan **por valor** y al asignar uno en otro se guarda una copia del valor de la variable, por lo que el código:

```
a = 3
b = a
a = 5
print a, b
```

Muestra en consola el mensaje `5, 3`. Al asignar el valor de `a` en `b`, se guarda una copia del valor original y al modificar la variable original, la nueva queda con el valor antiguo.

Por otro lado, las listas son **tipos por referencia**[2]. Esto quiere decir que al asignar una lista en otra, no se copia el contenido sino que ambas son solo un acceso directo para los mismos datos. De esta manera, al ejecutar el código:

```
a = [3, 7]
b = a
b[0] = 5
print a, b
```

Se mostrará en la consola el texto `[5, 7] [5, 7]`. Ya que al modificar la lista a través de la variable `b`, cambiamos el valor de la única copia de los datos (hubiese pasado lo mismo si el código hubiera dicho `a[0] = 5`) y vamos a acceder al valor nuevo a través de cualquiera de las dos variables.

Para sacar una copia de los elementos de una lista, de manera que al modificar la lista original no cambien los elementos de la nueva podemos utilizar ciclos. Por ejemplo:

```
original = [4, 8, 15, 16, 23, 42]
# creamos una lista vacía
copia = []
# copiamos los elementos uno por uno
for i in original:
    copia.append(i)

a[0] = 33
print a # imprime [33, 8, 15, 16, 23, 42]
print b # imprime [4, 8, 15, 16, 23, 42]
```

Al pasar variables como parámetros a una función también se aplica la diferencia entre los tipos por valor y los por referencia. Con los tipos básicos se pasa como parámetro una copia del valor y al modificarlo adentro de la función no cambia el valor de la variable original. Por ejemplo, al ejecutar:

```
def f(x):
    x = x + 5

a = 7
f(a)
print a
```

Veremos que en la consola se muestra `7`, ya que el cambio de valor que se realiza en la función afecta a una variable local que tiene una copia del valor recibido como parámetro.

Si una función recibe como parámetro una lista, se pasa una referencia a los elementos de esta. Por lo que si modificamos los elementos de la lista adentro de la función, cambian también en la lista original. Por ejemplo, al ejecutar:

```
def f(x):
    x[0] = x[0] + 5

a = [-7, 15, -1]
f(a)
print a
```

[2] Al igual que los objetos de una clase.

Se mostrará en la consola `[-2, 15, -1]`, ya que al editar los elementos de la lista a través de la referencia recibida como parámetro, se modifican también los elementos de la lista original.

9.11. La clase string

Los strings en Python se delimitan por comillas simples o dobles (esto permite escribir comillas de un tipo dentro de un string que está delimitado por comillas del otro tipo directamente, como `'La película "El Aro" es de terror.'` sin la necesidad de escaparlas con \).

Los strings se pueden trabajar como listas para obtener el carácter de alguna posición, para recorrerlos con un ciclo `for` y para calcular substrings. Por ejemplo:

```
linea = 'La casa es roja.'
print linea[5]      # imprime una letra 's'

for letra in linea:
    print letra

print linea [5:9] # imprime'sa e'
```

Además, la clase string provee varios métodos útiles. Algunos de estos[3] son:

- `find(x)`: entrega el valor del índice que ocupa el substring igual a `x` dentro del string.
- `find(x, inicio)`: entrega el valor del índice que ocupa el substring igual a `x` dentro del string, buscando desde la posición `inicio` en adelante.
- `lower()`: retorna una copia del string original con todas las letras en minúscula, no modifica el string original.
- `upper()`: retorna una copia del string original con todas las letras en mayúscula, no modifica el string original.
- `replace(old, new)`: retorna una copia del string original reemplazando todas las apariciones del substring `old` por `new`, no modifica el string original.
- `split()`: retorna una lista con las palabras del substring original que estén delimitadas por espacios.
- `split(separador)`: retorna una lista con las palabras del substring original que estén delimitadas por el substring `separador`.

9.11.1. Caracteres especiales

Es común querer mostrar en la consola caracteres especiales, como una comilla o un salto de línea. Como Pyhton permite delimitar un string con comillas simples (`'`) o dobles (`"`), es simple generar un string que utilice un tipo de comillas simplemente eligiendo las otras para delimitarlo. Por ejemplo:

[3] Para ver la lista completa, puede revisar la documentación oficial en `http://docs.python.org/2/library/stdtypes.html#string-methods`.

```
a = "Bernardo O'Higgins fue un prócer de la patria."
b = 'El libro "El Hobbit" es mejor que la película.'
```

Pero si queremos generar un string que contenga ambos tipos de comillas nos encontramos con un problema. Este puede ser resuelto de varias formas: podríamos generar un string para cada sección con un solo tipo de comillas y luego concatenarlos para generar el que queríamos. O podríamos usar **secuencias de escape** para indicar que se debe incluir una comilla dentro de un string. Esto consiste en escribir la secuencia `\'` para insertar una comilla simple o `\"` para insertar una comilla doble. Por ejemplo:

```
a = '\"La vida de O\'Higgins\" es una película.'
print a
```

Mostrará en la consola el texto:

```
"La vida de O'Higgins" es una película.
```

Las secuencias de escape utilizadas de forma más frecuente son:

- `\'`: inserta una comilla simple.
- `\"`: inserta una comilla doble.
- `\t`: inserta una tabulación, es útil para alinear datos en columnas.
- `\n`: inserta un salto de línea (equivale a apretar la tecla "Enter").
- `\r`: inserta un retorno de carro (herencia de las máquinas de escribir). En Windows, el salto de línea se representa con la secuencia `\r\n`.

Ahora tenemos un nuevo problema, el carácter `\` se usa para indicar que a continuación viene una secuencia de escape. Por lo que se debe interpretar como un único símbolo este carácter y el que viene a continuación. ¿Cómo mostramos el carácter `\`? Simplemente lo escapamos. De esta forma, el código:

```
a = 'No hay que confundir el símbolo \\ con el de división (/).'
print a
```

Muestra en consola:

```
No hay que confundir el símbolo \ con el de división (/).
```

9.12. Ejercicios

9.12.1. Básicos

1. Escriba un programa que pida números al usuario hasta que este ingrese cero. Guarde los números en una lista. Cuando el usuario termine de escribir los números, imprímalos en la consola, uno en cada línea.

9.12.2. Sugeridos

1. Escriba un programa que pida al usuario una cantidad de notas a ingresar, repitiendo la pregunta hasta que ingrese un número positivo. Luego pida que ingrese esa cantidad de notas una por una, validando que estén entre 1 y 7, y guárdelas en una lista. Finalmente, imprima la moda, media y mediana de las notas ingresadas.

2. Escriba una función que reciba una lista como parámetro y retorne una nueva lista con los elementos de la lista original en algún orden aleatorio.

3. Escriba un programa que reparta 5 cartas de un mazo a cada jugador, asegurando que las cartas no se repitan. Muestre en cosola las cartas que recibe cada uno. Luego, declare como ganador al que recibió la carta más alta.

4. Escriba un programa que pida al usuario que escriba su nombre y luego lo muestre con las mayúsculas correspondientes a la escritura correcta. Por ejemplo, si el usuario escribe `FrAnCiScA EUgEnIA mOrALEs`, su programa debe corregirlo e imprimir `Francisca Eugenia Morales`.

5. Escriba un programa para jugar al gato. Guarde el tablero en una lista de listas, donde cada elemento final es una casilla del tablero. Permita jugar a dos jugadores preguntando alternadamente en que posición desea jugar cada uno. Debe validar que la casilla donde quieren jugar esté vacía. El programa termina cuando hay un ganador o cuando el tablero se llena, declarando un empate.

9.12.3. Desafiantes

1. Escriba un programa para jugar póquer que reparta 5 cartas de un mazo a cada jugador, asegurando que las cartas no se repitan. Muestre en cosola las cartas que recibe cada uno. Luego, compare las cartas que recibe cada uno y declare al ganador. Puede revisar el detalle de las reglas en `http://es.wikipedia.org/wiki/Poquer`.

2. Escriba una función que reciba como parámetro un tablero de sudoku (una lista con 9 elementos, donde cada elemento es una lista con 9 números enteros) y retorne `True` o `False` dependiendo de si el tablero está bien o mal resuelto.

Capítulo 10

Tuplas y diccionarios

10.1. Tuplas

Una tupla es una colección **inmutable** de elementos, que pueden ser de distintos tipos. Al igual que cualquier otra variable, tiene un nombre y este debe seguir las mismas reglas que los nombres de variables y funciones. Se diferencias de las listas en que sus elementos no están dentro de corchetes `[` y `]`. Por ejemplo:

```
tupla = 2, 3, 'hola'
```

Para imprimir los elementos de una tupla utilizamos la misma notación que para los elementos de una lista. Esto incluye también la posibilidad de utilizar índices negativos. Por ejemplo:

```
tupla = 2, 3, 'hola'
print tupla[0]   # Imprime 2
print tupla[-1] # Imprime hola
```

También es posible imprimir una tupla directamente, en este caso se mostrarán todos sus elementos separados por comas, envueltos en un par de paréntesis para diferenciarlos de una lista. Por ejemplo:

```
tupla = 2, 3, 'hola'
print tupla # imprime (2, 3, 'hola')
```

Es imposible agregar nuevos elementos a una tupla. Esto implica que todos los elementos contenidos en una tupla se deben especificar en su creación. Además es imposible cambiar el valor de cualquiera de los elementos de una tupla.

Error común de programación.

Tratar de cambiar el valor de un elemento en una tupla, por ejemplo:

```
tupla = (2, 3, 'hola')
tupla[0] = 5 # error, las tuplas son inmutables
```

Para revisar cuántos elementos tiene una tupla utilizamos la función `len()` igual que para una lista o un `str`.

10.2. Operaciones con tuplas

Las tuplas permiten dos operaciones importantes: multiplicación por un número entero y concatenación de tuplas.

Al multiplicar una tupla por un número entero obtenemos como resultado una nueva tupla con los elementos de la original repetidos tantas veces como el valor del número. Si el número es negativo se trata como si fuera cero. Por ejemplo:

```
tupla = 2, 3, 'hola'

tupla2 = tupla*3
print tupla2 # imprime (2, 3, 'hola', 2, 3, 'hola', 2, 3, 'hola')
print -2*tupla # imprime ()

print tupla # imprime (2, 3, 'hola')
```

Para concatenar tuplas utilizamos el operador `+`, de la misma manera que con variables de tipo `str`. Por ejemplo:

```
t1 = 2, 3, 'hola'
t2 = 1, -2, 0

t3 = t1 + t2
print t3 # imprime (2, 3, 'hola', 1, -2, 0)

t4 = t2 + t1
print t4 # imprime (1, -2, 0, 2, 3, 'hola')
```

Recuerde que en Python los decimales de un número se separan de la parte entera con un punto, siguiendo el estándar de numeración inglés. Note la diferencia en el siguiente código:

```
n1 = 6.4
n2 = 6,4

print 'El doble de n1 es', n1*2 # imprime El doble de n1 es 12.8
print 'El doble de n2 es', n2*2 # imprime El doble de n2 es (6, 4, 6, 4)
```

Error común de programación.

Usar comas para separar la parte decimal de un número. Esto genera una tupla.

10.3. Asignaciones con tuplas

Es posible asignar los valores de una tupla a varias variables al mismo tiempo. Esto se llama desencapsulación de datos. Por ejemplo:

```
tupla = 2, 3, 'hola'
a, b, c = tupla
print b # imprime 3
print c # imprime hola
```

Para poder realizar esta asignación múltiple es necesario que al lado izquierdo se encuentren tantas variables como elementos tenga la tupla.

Usando esta notación también es posible intercambiar valores sin necesidad de variables auxiliares. Por ejemplo:

```
a = 25
b = 'hola'
a, b = b, a # implícitamente se desencapsula el lado derecho
print a # imprime hola
print b # imprime 25
```

10.4. Diccionarios

Un diccionario es una colección similar a una lista, donde los índices de los elementos no son necesariamente números. Se diferencias de las listas y de las tuplas porque están delimitados por llaves `{` y `}`. Por ejemplo:

```
libro = {'titulo': 'El Hobbit', 'paginas': 308}
```

Tenemos un diccionario llamado `libro` que guarda dos valores, si queremos imprimirlos usamos la misma notación que para acceder a los elementos de una lista, pero escribiendo dentro de los corchetes el string del índice correspondiente. Por ejemplo:

```
libro = {'titulo': 'El Hobbit', 'paginas': 308}
print libro['titulo'], 'tiene', libro['paginas'], 'páginas.'
```

Los nombres que usamos para los índices pueden ser cualquier valor que se pueda guardar en un `str` y siguen las mismas reglas que estos. Esto signifca que son sensibles a las mayúsculas y minúsculas.

Al declarar un diccionario con todos sus elementos de forma explícita (como el ejemplo anterior), escribimos dentro de las llaves los nombres de los índices que vamos a utilizar. Los índices se deben escribir dentro de comillas. Luego se escriben dos puntos y el valor a guardar en ese índice. El valor a guardar puede ser de cualquiera de los tipos de datos que ya hemos visto (`bool`, `int`, `str`, lista, tupla, diccionario).

Es posible cambiar el valor de uno de los elementos del diccionario usando una asignación de la misma manera que en una lista. Por ejemplo:

```
libro = {'titulo': 'El Hobbit', 'paginas': 308}
libro['paginas'] = 523
```

También es posible asignar más elementos utilizando índices que no se habían asignado anteriormente. Por ejemplo:

```
libro = {'titulo': 'El Hobbit', 'paginas': 308}
libro['editorial'] = 'Minotauro'
libro['autor'] = 'John Tolkien'
```

La cantidad de elementos de un diccionario se puede calcular con la función `len()`. Esta entrega la cantidad de valores que guarda, sin contar los índices como valores. Por ejemplo:

```
libro = {'titulo': 'El Hobbit', 'paginas': 308}
print len(libro) # imprime 2
```

10.4.1. Índices de un diccionario

Hemos visto hasta ahora que un diccionario acepta `str` como índices. También es posible utilizar números, tuplas o los valores `True` y `False` como índices. No es posible usar listas como índices para un diccionario. Por ejemplo:

```
d = {'a':25, 1:42, 2:'hola', False:-22, (4,6):45}
print (d['a'] + d[False]) * d[2] # imprime holaholahola
```

Consejo.

Use solamente `str` y números enteros como índices para un diccionario.

10.4.2. Recorrer los elementos de un diccionario

Es posible recorrer un diccionario utilizando un ciclo `for`, pero se debe tener que cuenta que este revisa los índices del diccionario. Por ejemplo:

```
d = {'a':25, 1:42, 2:'hola', 'nombre':'Daniela'}

for i in d:
    print i
```

Imprimiría en la consola:

```
a
1
2
nombre
```

Si queremos imprimir el valor de cada elemento de un diccionario, simplemente utilizamos los índices que nos entrega el ciclo para acceder a cada posición de la siguiente forma:

```
d = {'a':25, 1:42, 2:'hola', 'nombre':'Daniela'}

for i in d:
    print d[i]
```

Python también provee facilidades para obtener los índices y los valores de un diccionario en listas, con las que ya estamos acostumbrados a trabajar. Para esto se usan los métodos `keys()` y `values`. Por ejemplo:

```
d = {'a':25, 1:42, 2:'hola', 'nombre':'Daniela'}

indices = d.keys()
valores = d.values()

print indices # imprime ['a', 1, 2, 'nombre']
print valores # imprime [25, 42, 'hola', 'Daniela']
```

10.5. Diccionarios como encapsuladores de datos

Podemos usar un diccionario para representar las distintas características de alguna cosa[1] en una sola variable. Por ejemplo, podríamos guardar la sigla, el nombre y la cantidad de alumnos de un curso de la universidad:

```
curso = {'sigla':'ING1310', 'nombre':'Introducción', 'alumnos':92}
print 'El curso', curso['nombre'],
print 'tiene', curso['alumnos'], 'alumnos.'
```

Aprovechando esto y guardando los diccionarios en una lista, podríamos generar el catálogo de cursos de la universidad:

```
catalogo = []
seguir = 1
while seguir != 0:
    sigla = raw_input('Sigla del curso: ')
    nombre = raw_input('Nombre del curso: ')
    cant = int(raw_input('Cantidad de alumnos del curso: '))
    curso = {'sigla': sigla, 'nombre': nombre, 'alumnos': cant}
    catalogo.append(curso)
    print 'Escriba 0 para terminar, otro número para seguir:'
    seguir = int(raw_input())

cant = 0
suma = 0
for curso in catalogo:
    cant += 1
    suma += curso['alumnos']
print 'En promedio hay', suma/cant, 'alumnos por curso.'
```

10.6. El operador `in`

El operador `in` permite comprobar si un elemento pertenece a una colección de manera fácil. Este se puede aplicar a listas, tuplas y diccionarios, aunque con estos útlimos funciona distinto. Al aplicar `in` a una lista o tupla, obtenemos como resultado `True` si el valor especificado se encuentra en la colección. Al aplicar `in` a un diccionario, obtenemos `True` si el valor buscado está dentro del conjunto de **índices** del diccionario. Por ejemplo:

[1] Esto es muy similar a clases y objetos, para los que saben algo del tema.

```
l = [-2, 4, 'casa']
t = (5, 6, -3, 'hola', 42, 78)
d = {'casa':'grande', 'b':'gato', 1:25, 2:42}

lContiene42 = 42 in l # False
tContiene42 = 42 in l # True
dContiene42 = 42 in l # False (no es un índice)
lContieneCasa = 'casa' in l # True
tContieneCasa = 'casa' in l # False
dContieneCasa = 'casa' in l # True (sí es un índice)
```

10.7. Ejercicios

10.7.1. Básicos

1. Escriba una función que reciba dos tuplas como parámetros. Donde cada tupla sea un par de números representando una coordenada del plano cartesiano. Su función debe retornar la distancia entre los dos puntos.

10.7.2. Sugeridos

1. Escriba una agenda donde pueda guardar el nombre, teléfono y mail de varias personas. Luego implemente una búsqueda por nombre que entregue todos los datos de una persona o diga que la persona no está en la lista. Finalmente, implemente una búsqueda por mail que imprima todos los datos de una persona o diga que el mail no se encuentra en la agenda.

2. Escriba un programa para manejar un local de comida rápida. Para esto pida al usuario que ingrese los elementos del menú, indicando para cada uno su nombre, precio y cantidad de calorías. Debe guardar el menú del local en una lista, donde cada elemento es una tupla con los valores de un ítem.

   A continuación maneje la caja del local. Muestre al usuario una lista de los platos ofrecidos, mostrando también sus precios pero sin mostrar la cantidad de calorías de cada uno. Pida al usuario que elija uno o más platos del menú y muestre a continuación el total a pagar y la cantidad total de calorías que contienen.

Capítulo 11

Archivos

11.1. Necesidad

Los programas que hemos escrito hasta ahora piden datos al usuario, trabajan con ellos, calculan cosas y muestran mensajes. El problema que tienen es que al ejecutarlos nuevamente tenemos que ingresar todos los parámetros nuevamente a mano. No tienen memoria persistente.

Los archivos dentro de un computador guardan información de forma permanente. Al apagar y volver a prender la máquina siguen ahí. Hemos escrito archivos de código (con la extensión `.py`) y también sabemos que podemos escribir librerías de funciones. Ahora trabajaremos con archivos de datos, externos a nuestro código.

11.2. Conceptos comunes

Un archivo es una sección dentro de la memoria persistente del computador. Estos pueden estar ubicados en un disco duro, CD, memoria flash (pendrives), cintas magnéticas, etc. Todo archivo tiene un nombre y una extensión, como `foto.jpg`, `sonata.mp3`, `video.mkv`, `funciones.py` o `tesis.doc`.

El contenido de un archivo depende del tipo de este y cada tipo de archivo se puede abrir con un programa determinado. De esta manera podemos abrir un archivo `mp3` en un reproductor de música para escuchar una canción o abrir un archivo `docx` en Word para escribir un informe.

Nosotros trabajaremos con archivos de texto simple (o texto plano). Estos normalmente usan la extensión `txt`, pero también podrían utilizar otra como `py`, `out`, `in`, `conf`, etc. La ventaja de este tipo de archivos es que son muy fáciles de leer y de escribir y que la gran mayoría de los lenguajes de programación (incluyendo Python) proveen formas simples de trabajar con ellos.

11.3. Lectura de archivos

Para leer el contenido de un archivo de texto, usamos la función `open(nombreArchivo)`. Esta función necesita como parámetro un string con el nombre del archivo que vamos a leer

y nos entrega como respuesta una variable de tipo `file`[1]. Para leer el contenido del archivo, llamamos al método `readline()` de nuestra variable, este nos entrega un string con el texto de cada línea del archivo con un carácter `'\n'` al final. Por ejemplo:

```
archivo = open('input.txt')
primeraLinea = archivo.readline()
segundaLinea = archivo.readline()

print 'La primera línea dice', primeraLinea
```

También es posible leer todas las líneas de un archivo utilizando un ciclo `for` de la siguiente forma:

```
archivo = open('input.txt')
i = 1
for linea in archivo:
    print 'La línea', i, 'del archivo tiene el texto:', linea
    i += 1
```

Después de terminar de leer el contenido de un archivo, es recomendable cerrarlo. Con esto indicamos al sistema operativo que el archivo ya está disponible para que otro programa lo pueda utilizar. Para hacer esto simplemente ejecutamos el método `close()` del archivo que abrimos de la siguiente forma:

```
archivo = open('notas.txt')
linea1 = archivo.readline()
linea2 = archivo.readline()
archivo.close()
```

11.4. Escritura de archivos

Para escribir un archivo de texto, debemos abrirlo en modo de escritura, para esto utilizamos la función `open(nombreArchivo, modoApertura)`, pasando un segundo parámetro con el texto `'w'`[2] indicando que vamos a escribir en el archivo.

El método `write()` escribe al archivo el texto que pasamos como parámetro, sin saltos de línea (estos se deben ingresar manualmente con la secuencia `\n` o `\r\n` dependiendo del sistema operativo). Para finalizar la escritura debemos llamar al método `close()`. Por ejemplo:

```
archivo = open('salida.txt', 'w')
archivo.write('Nuevo archivo de texto.\r\n')
archivo.write('Este archivo tiene dos líneas.')
archivo.close()
```

Si el archivo en el que queremos escribir nuestros datos no existe no hay problema. Al ejecutar `close()` se creará con el contenido que escribimos.

[1] *File* significa archivo en inglés.
[2] Del inglés *write*, escribir.

Error común de programación.

Olvidar cerrar un archivo después de terminar de escribir en él.

11.5. Rutas de acceso

Cuando abrimos un archivo tenemos varias opciones para escribir su nombre. Si el archivo está en la misma carpeta que nuestro programa, podemos simplemente escribir el nombre del archivo a leer. Si está en otra carpeta del computador, podemos escribir la ruta completa (también se le dice ruta absoluta) desde el disco donde se encuentra hasta el nombre del archivo. Por último, si sabemos la ubicación del archivo relativa a la carpeta donde está nuestro programa, podemos escribir la ruta relativa del archivo. Por ejemplo:

```
# el arhivo está en la misma carpeta que el programa
archivo1 = open('input.txt')

# ruta absoluta al archivo en Windows
archivo2 = open('C:\Users\Francisca\Desktop\notas.txt')
# ruta absoluta al archivo en Linux o Mac
archivo2 = open('/home/Pedro/datos.txt')

# ruta relativa en Windows
archivo2 = open('Carpeta de datos\notas.txt')
# ruta relativa en Linux o Mac
archivo2 = open('Carpeta de datos/Subcarpeta 1/datos.txt')
```

Consejo.

Utilice rutas de archivo relativas a la ubicación de su programa. El uso de rutas absolutas limita el funcionamiento de su programa a una arquitectura específica (si anda bien en Windows, no funcionará en Mac o Linux).

11.6. Ejercicios

11.6.1. Básicos

1. Escriba un programa que pida al usuario el nombre de un archivo. Luego escriba en la consola todo el contenido de este.

11.6.2. Sugeridos

1. Escriba una agenda donde pueda guardar el nombre, teléfono y mail de varias personas. Luego implemente una búsqueda por nombre que entregue todos los datos de una

persona o diga que la persona no está en la lista. Finalmente, implemente una búsqueda por mail que imprima todos los datos de una persona o diga que el mail no se encuentra en la agenda.

Su programa deberá guardar los datos de los contactos de la agenda en un archivo de datos, en algún formato a su elección. Cada vez que el programa se inicie se deben leer todos los contactos desde este archivo, para que el usuario pueda buscar en ellos. Al terminar su programa deberá reescribir este archivo para guardar los contactos nuevos que se hayan ingresado.

Capítulo 12

Matrices y arreglos

12.1. La librería NumPy

NumPy, del inglés *Numeric Python*, es una librería que provee facilidades para trabajar con arreglos y matrices de números. Se puede descargar gratuitamente desde su página oficial[1] y para poder utilizarla se debe importar[2]. Asuma que todos ejemplos que se darán a continuación están precedidos por la línea

```
from numpy import *
```

12.2. Arreglos

Un arreglo es una estructura de datos[3] muy similar a una lista, pero que está diseñada para trabajar solamente con números como elementos (pueden ser `int` o `float`). Un arreglo es equivalente a un vector en álgebra lineal.

Para crear un nuevo arreglo debemos llamar a la función `array()`[4] y pasarle como parámetro una lista con los valores que queremos guardar inicialmente. Por ejemplo:

```
lista = [2, 3, 29]
a = array(lista)
b = array([2, 7, -1])
# Aquí aprovechamos que la función range() retorna una lista
c = array(range(23))
```

[1] `http://www.numpy.org/`

[2] Esto ya lo hemos hecho anteriormente para generar números aleatorios con `random`.

[3] Es una clase, si quiere saber más de este tema revise en `https://docs.python.org/2/tutorial/classes.html`.

[4] Técnicamente el constructor de la clase `array`.

Error común de programación.

Pasar muchos números como parámetros a `array()` en lugar de una lista. Para diferenciar:

```
x = array([2, 7, -1]) # bien
y = array(2, 7, -1)   # error
```

Al igual que las listas, los arreglos pueden ser de varias dimensiones. Podemos crear un arreglo multidimensional pasando al constructor una lista de varias dimensiones:

```
a = array([[2,2],[3,3],[4,4]])
```

Existen varias funciones que permiten crear arreglos con valores iniciales conocidos. Algunas de estas son:

- `arange(max)`: equivalente a `array(range(max))`.
- `arange(min, max)`: equivalente a `array(range(min, max))`.
- `arange(min, max, incremento)`: equivalente a ejecutar la composición de funciones `array(range(min, max, incremento))`.
- `zeros(cantidad)`: crea un arreglo con tantos ceros como se indique en `cantidad`.
- `zeros((x, y))`: crea un arreglo de dos dimensiones con `x` filas e `y` columnas. El uso de doble paréntesis es obligatorio, ya que el único parámetro recibido es una tupla.
- `ones(cantidad)`: crea un arreglo con tantos unos como se indique en `cantidad`.
- `linspace(min, max, cantidad)`: crea un arreglo con `cantidad` valores, repartidos equitativamente entre `min` y `max` (incluyendo ambos límites).
- `random.random(cantidad)`: crea un arreglo con `cantidad` valores aleatorios entre 0 y 1.

Para acceder a los elementos de un array se usa la misma notación de índices que se usa en las listas y tuplas. También es posible utilizar la notación `arreglo[inicio:fin]` para obtener un subarreglo del original, pero se debe tener en cuenta que se recibe una referencia a la sección pedida y que los cambios en esta se reflejarán en el arreglo original. Por ejemplo:

```
a = arange(2,14,2)
print a               # imprime [ 2 4 6 8 10 12]
print a[1] * a[-1]    # imprime 48
b = a[1:-1]
print b               # imprime [ 4 6 8 10]
b[2] = -1
print b               # imprime [ 4 6 -1 10]
print a               # imprime [ 2 4 6 -1 10 12]
```

Si queremos sacar una copia de un arreglo debemos utilizar el método `copy()` de la siguiente forma:

```
a = arange(2,14,2)
c = a.copy()
c[2] = 23
print a                 # imprime [ 2 4 6 8 10 12]
print c                 # imprime [ 2 4 23 8 10 12]
```

Error común de programación.

Tratar de utilizar la notación de sublistas `[:]` para obtener una copia de un arreglo.

12.2.1. Operaciones con arreglos

Es posible operar entre un arreglo y un número con las operaciones de suma, resta, multiplicación, división, exponenciación y módulo. Al realizar cualquiera de estas operaciones, el resultado será un nuevo arreglo resultante de aplicar la operación indicada a cada uno de sus elementos con el número utilizado. Por ejemplo:

```
a = array([4, 8, 15, 16, 23, 42])
print 2*a   # imprime [ 8 16 30 32 46 84]
print a/2   # imprime [ 2 4 7 8 11 21]
print 2/a   # imprime [0 0 0 0 0 0]
print a**4  # imprime [ 256 4096 50625 65536 279841 3111696]
print a%7   # imprime [4 1 1 2 2 0]
```

También es posible utilizar estas operaciones entre dos arreglos. Para esto ambos deben tener la misma cantidad de elementos. El resultado de cualquiera de estas operaciones será un nuevo arreglo donde cada uno de sus elementos se obtiene de aplicar la operación indicada a los elementos de esa misma posición en los arreglos originales. Por ejemplo:

```
a = array([4, 8, 15, 16, 23, 42])
b = array([-3, 2, 12, 7, 0, 12])
print b*a   # imprime [-12 16 180 112 0 294]
print a/b   # imprime [-2 4 1 2 0 6]
print b/a   # imprime [-1 0 0 0 0 0]
print a**b  # imprime [ 0 64 129746337890625 268435456 1 230539333248]
```

Por último, NumPy provee una serie de funciones predefinidas que pueden recibir arreglos como parámetros. En el caso de pasar un arreglo a estas, el resultado será un nuevo arreglo obtenido al aplicar la función a cada uno de los componentes del arreglo original. Algunas[5] de estas funciones son:

- `exp(x)`: exponencial e^x.
- `sin(x)`: seno.
- `cos(x)`: coseno.
- `tan(x)`: tangente.
- `log(x)`: logaritmo.
- `log10(x)`: logaritmo en base 10.
- `sqrt(x)`: raíz cuadrada $\sqrt{(}x)$.
- `radians(x)`: convierte un ángulo de

[5] Para una lista completa puede ver `http://docs.scipy.org/doc/numpy/reference/routines.math.html`.

grados a radianes.

- `degrees(x)`: convierte un ángulo de radianes a grados.
- `absolute(x)`: valor absoluto.

Es posible combinar estas funciones con las otras operaciones de arreglos que ya hemos visto. Por ejemplo:

```
a = linspace(0, 10, 30)
b = linspace(-5, 6, 30)
c = linspace(3, 19, 30)

d = 4*sin(a) + (c - sqrt(a)*b)**3 - log10(c) + exp(4)*b/5
```

12.2.2. Métodos de arreglos

Además de todas las operaciones que podemos ejecutar con arreglos, estos proveen métodos para obtener datos sobre su contenido. Algunos[6] de estos son:

- `fill(x)`: todos los elementos del array toman el valor `x`.
- `copy()`: entrega una copia del array. Recuerde que la notación `arreglo[:]` entrega una referencia al mismo arreglo y no una copia diferente.
- `resize(forma)`: cambia las dimensiones del array, recibe una tupla con el valor de cada dimensión. El array no puede estar siendo referenciado por otra variable.
- `transpose()`: retorna el array transpuesto.
- `flatten()`: retorna una copia del array en 1 dimensión.
- `all()`: retorna verdadero si todos los elementos son verdaderos o distintos de cero.
- `any()`: retorna verdadero si algún elemento es verdadero o distinto de cero.
- `min()`: retorna el mínimo.
- `max()`: retorna el máximo.
- `argmin()`: retorna el índice del elemento mínimo.
- `argmax()`: retorna el índice del elemento máximo.
- `sum()`: retorna la suma de los elementos.
- `cumsum()`: retorna un array con la suma acumulada de los elementos hasta cada posición.
- `mean()`: retorna el promedio de los elementos.
- `var()`: retorna la varianza de los elementos.
- `std()`: retorna la desviación estándar de los elementos.

[6] Puede leer una lista completa en `http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods`.

- `prod()`: retorna la multiplicación de los elementos.
- `cumprod()`: retorna un array con la multiplicación de los elementos hasta cada posición.

Ejemplos de uso:

```
a = random.random(10)
print 'El mayor es', a.max(), 'y está en el lugar', a.argmax()
print 'La suma de sus elementos es', a.sum()
print 'Su desviación estándar es', a.std()
```

12.3. Matrices

Una matriz es muy similar a un arreglo de dos dimensiones. Podemos crear una matriz mueva utilizando `matrix()` y pasando como parámetro una lista de listas, donde cada lista interna tiene la misma cantidad de elementos.

Para acceder a los elementos de una matriz se usa una notación similar a la de listas, pero pasando una tupla con las coordenadas a obtener. Por ejemplo:

```
m = matrix([[23, 4], [4, 5], [6, 7]])

print m[2,1] # imprime 7
print m[0,0] # imprime 23
print m[1,0] # imprime 4
```

12.3.1. Operaciones con matrices

Es posible aplicar las operaciones de suma y resta a dos matrices de las mismas dimensiones. El resultado de estas operaciones será una nueva matriz con las mismas dimensiones que las originales.

Al operar entre una matriz y un número, la operación se realiza con cada uno de los elementos internos de la matriz, de la misma forma que en los arreglos.

Las operaciones con funciones matemáticas que vimos para trabajar con arreglos también sirven para trabajar con matrices. Por ejemplo:

```
a = matrix([[23, 4], [4, 5], [6, 7]])
b = matrix([[-7, 8], [1, 0], [2, 3]])

c = a + 2*b - cos(a)/2 + log10(a+2*b)
print c
```

12.3.2. Multiplicación de matrices

Es posible multiplicar dos matrices para obtener como resultado otra matriz. Para poder hacer esto se debe cumplir que el número de columnas de la primera sea igual al número de filas de la segunda. El resultado de multiplicar una matriz con a filas y b columnas por una

matriz de b filas y c columnas será una matriz de a filas y c columnas. La multiplicación de matrices no es conmutativa[7].

Dadas dos matrices $A = (a_{ij})_{m\times n}$ y $B = (b_{ij})_{n\times p}$, definimos su multiplicación como una nueva matriz $C = A * B = (c_{ij})_{m\times p}$, donde cada elemento $c_{i,j}$ está definido por $c_{ij} = \sum_{r=1}^{n} a_{ir}b_{rj}$.

En Python, para multiplicar dos matrices utilizando esta definición, se usa el operador `*`. El operador de potencias permite elevar una matriz a un número entero siguiendo la definición de potencia como multiplicación abreviada de factores iguales. Para poder calcular las potencias de una matriz, esta debe ser cuadrada (debe tener la misma cantidad de filas que de columnas). Por ejemplo:

```
a = matrix([[23, 4, 4, 5], [4, 5, 8, 7], [2, 3, 6, 7]])
b = matrix([[-7, 8, 7], [1, 0, 5], [2, 3, -1], [2, 3, 4]])
c = a*b
d = b*a
e = c**4
```

La matriz identidad es el elemento neutro en la multiplicación de matrices. Es decir, al multiplicarla por otra matriz, el resultado es la otra matriz. Esta matriz es cuadrada y está compuesta por unos en la diagonal principal y ceros en las otras posiciones. En Python se pueden generar utilizando la función `identity(size)` como parámetro para el constructor de `matrix()`. Por ejemplo:

```
identidad3 = matrix(identity(3))
```

Siguiendo con esta línea, dada una matriz A, podemos definir su inversa A^{-1} como aquella matriz que al multiplicarla por la original da como resultado la matriz identidad ($A * A^{-1}1 = I$). No toda matriz tiene una inversa (por ejemplo, una matriz con puros ceros no es invertible). Para obtener la inversa de una matriz en Python utilizamos la notación `matriz.I`. Por ejemplo:

```
a = matrix([[23, 4, 4, 5], [4, 5, 8, 7], [2, 3, 6, 7]])
inversa = a.I
```

La división de matrices no está definida, pero Python permite utilizar el operador `/` entre matrices de las mismas dimensiones. El resultado de esto es una nueva matriz del mismo tamaño que las anteriores, donde cada elemento es el resultado de la división entre los elementos que ocupaban esa posición en las matrices originales.

12.3.3. Resolver sistemas lineales de ecuaciones

Un sistema de ecuaciones lineales con n variables se puede escribir como una matriz cuadrada A de n filas y n columnas, que multiplicada por una matriz x de n filas y 1 columna con las incógnitas del sistema da como resultado otra matriz b de n filas y 1 columna.

Una forma de resolver este sistema es encontrar el inverso de la matriz A y multiplicarlo por b, lo que nos dará como resultado el valor de cada una de nuestras incógnitas. Para que el sistema tenga una solución, la matriz A debe ser invertible.

[7]Para más información sobre la multiplicación de matrices puede leer `http://es.wikipedia.org/wiki/Multiplicaci%C3%B3n_de_matrices`.

En Python, para resolver un sistema de ecuaciones lineales, podemos utilizar la función `linalg.solve(A, b)`, pasando como parámtros las matrices A y b del sistema de ecuaciones. Por ejemplo:

```
A = matrix([[1, 2, 3], [5, -7, 3], [4, 7, 8]])
b = matrix([[1], [-5], [2]])
x = linalg.solve(A, b)
```

Nótese que la matriz `b` está escrita de forma tal que tiene 1 columna y 3 filas. La notación para esto exige tener una lista de listas. Para simplificar la notación, podríamos escibir una matriz de una fila con muchas columnas y luego trasponerla (algo así como girarla y espejarla). Para obtener la traspusta de una matriz en Python utilizamos la notación `matriz.I`. Por ejemplo:

```
b = matrix([[1, -5, 2]])
bTraspuesta = b.T
```

12.4. Ejercicios

12.4.1. Básicos

1. Escriba una función que reciba un arreglo como parámetro y retorne su módulo.

12.4.2. Sugeridos

1. Escriba una función que reciba dos arreglos de tres elementos cada uno y retorne su producto punto.

2. Escriba una función que reciba dos arreglos de tres elementos cada uno y retorne un arreglo con su producto cruz.

3. Escriba una función que reciba una matriz y retorne `True` si es invertible o `False` si no lo es. Una matriz cuadrada es invertible si y solo si su determinante es distinto de cero. Una matriz no cuadrada no es invertible.

4. Escriba un programa que lea una lista de notas desde un archivo de texto, donde están escritas una en cada línea. Calcule e imprima el promedio y la desviación estándar de ellas.

Capítulo 13

Gráficos con `matplotlib`

13.1. La librería `matplotlib`

Matplotlib es una librería que permite generar gráficos en Python. Se puede descargar gratuitamente desde su página oficial[1] y para poder utilizarla se debe importar al archivo de código donde escribamos el código. Asuma que todos ejemplos que se darán a continuación están precedidos por la línea

```
from matplotlib import pyplot
```

13.2. Gráficos cartesianos

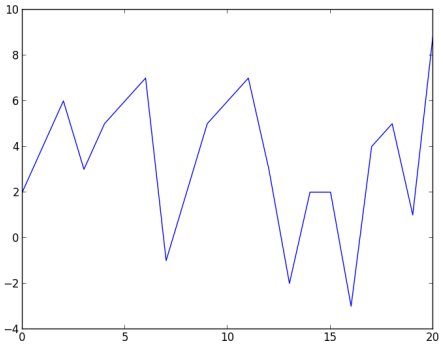
El método base para graficar es `pyplot.plot()`. Este puede recibir uno, dos o tres parámetros y realizará cosas distintas en cada ocasión. El primer parámetro debe ser una lista con valores numéricos o un `array`.

Si llamamos a `pyplot.plot()` con un solo parámetro, graficará esa secuencia de valores directamente. Por ejemplo:

```
valores = [2,4,6,3,5,6,7,-1,2,5,6,7,3,-2,2,2,-3,4,5,1,9]
pyplot.plot(valores)
pyplot.show()
```

Genera el gráfico:

[1] `http://matplotlib.org/`



El método `pyplot.show()` se utiliza para indicar que ya hemos terminado de generar todos los elementos del gráfico y permite mostrar el gráfico generado en una ventana nueva.

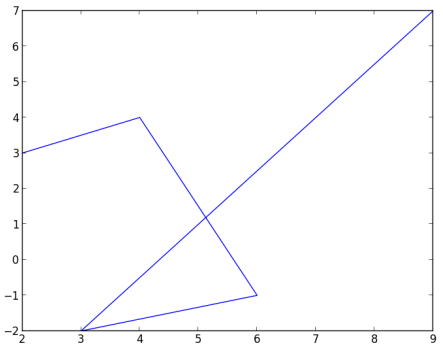
Error común de programación.

Olvidar escribir `pyplot.show()` para mostrar el gráfico y preguntarse por qué no se dibuja nada.

Con dos parámetros, `pyplot.plot(cx, cy)`, debe recibir dos listas con números (o dos `array`) con la misma cantidad de elementos. El primer parámetro indica las coordenadas `x` y el segundo las coordenadas `y` a graficar. Por ejemplo:

```
pyplot.plot([2,4,6,3,9], [3,4,-1,-2,7])
pyplot.show()
```

Genera el gráfico:

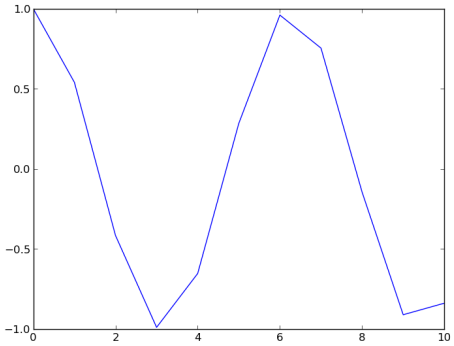


13.3. Graficar funciones

Con elementos de tipo `array` es posible calcular funciones matemáticas fácilmente. Esto es útil si queremos, por ejemplo, graficar la función coseno entre 0 y 10:

```
x = arange(11)
pyplot.plot(x, cos(x))
pyplot.show()
```

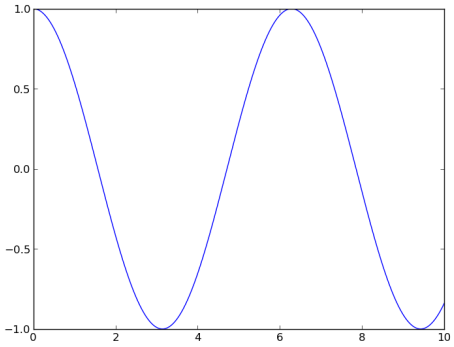
Genera el gráfico:



Si este gráfico no se parece mucho a la función que esperábamos, se debe a que la cantidad de puntos graficados no es suficiente para que se vea suave. Para lograr una mejor vista de la función es necesario generar más puntos para graficar. Vimos que generando elementos de tipo `array` con la función `linspace(min, max, cant)` obtenemos secuencias de números con la cantidad de puntos que queramos. Por lo que podemos obtener un mejor dibujo con el siguiente código:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, cos(x))
pyplot.show()
```

Genera el gráfico:

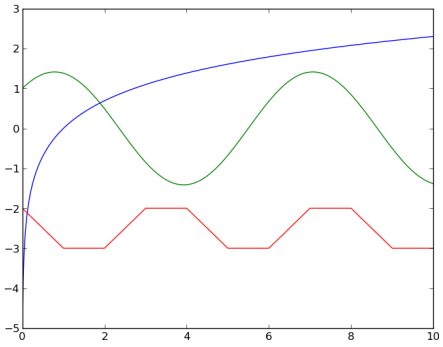


13.4. Graficar múltiples series en un mismo gráfico

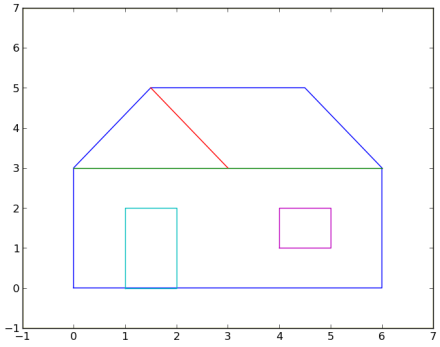
Es posible graficar más de una serie por gráfico. Para esto debemos ejecutar la función `pyplot.plot(cx, cy)` tantas veces como series queramos graficar y, finalmente, ejecutar la función `pyplot.show()` una sola vez. Por ejemplo:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, log(x))
pyplot.plot(x, sin(x) + cos(x))
pyplot.plot(range(11), [-2,-3,-3,-2,-2,-3,-3,-2,-2,-3,-3])
pyplot.show()
```

Genera el gráfico:



Con esto ya podemos generar dibujos como:



13.5. Dar formato a una serie

Cada serie (o línea) que agregamos a un gráfico toma un color distinto asignado automáticamente. Podemos asignar estos colores a mano utilizando un tercer parámetro al ejecutar la función `pyplot.plot(cx, cy, formato)`. Depués de las dos listas con números (o dos `array`) con la misma cantidad de elementos, pasamos un string con las opciones de formato que queremos utilizar. Para los colores básicos las opciones son:

- `'b'`: azul.
- `'r'`: rojo.
- `'m'`: magenta.
- `'k'`: negro.
- `'g'`: verde.
- `'c'`: cyan.
- `'y'`: amarillo.
- `'w'`: blanco.

Es posible también indicar un color en formato RGB[2] pasando una secuencia hexadecimal del tipo `#aabbcc` indicando el color deseado.

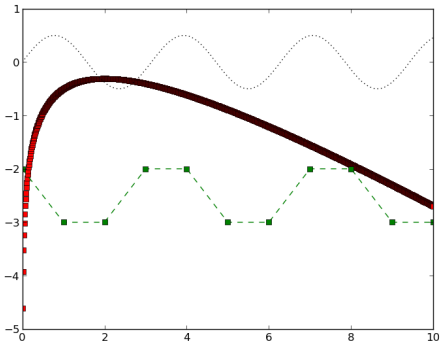
Además del color, es posible cambiar el estilo de la línea, para marcar los puntos con un círculo, usar líneas punteadas o alguna otra combinación. Algunas[3] de estas opciones son:

- `'-'`: línea continua.
- `'--'`: línea segmentada.
- `':'`: línea punteada.
- `'o'`: un círculo en cada punto.
- `'s'`: un cuadrado en cada punto.
- `'^'`: un triángulo en cada punto.

Estos formatos se pueden combinar entre ellos. Por ejemplo:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, log(x)-x/2, 'rs')
pyplot.plot(x, sin(x) * cos(x), ':k')
pyplot.plot(range(11), [-2,-3,-3,-2,-2,-3,-3,-2,-2,-3,-3], '--sg')
pyplot.show()
```

Genera el gráfico:



13.6. Opciones de bordes, leyenda y títulos

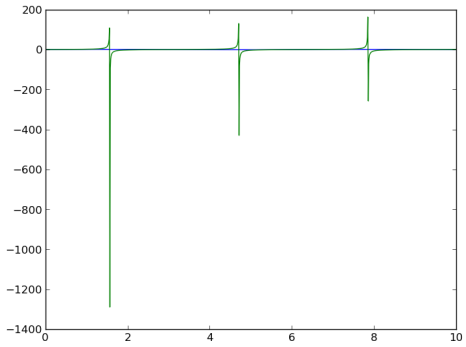
Es posible indicar qué sección del plano queremos que aparezca en nuestro gráfico. Esto es útil ya que la opción por defecto es mostrar el menor rectángulo que contenga todos los puntos que queremos graficar, distorsionando la escala de los ejes. Por ejemplo:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, sin(x))
pyplot.plot(x, tan(x))
pyplot.show()
```

Genera el gráfico:

[2] Detalles en `http://es.wikipedia.org/wiki/RGB`.

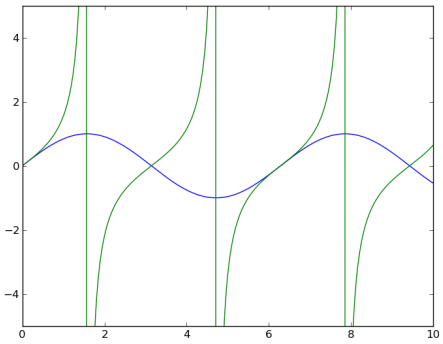
[3] La lista completa está en `http://matplotlib.org/1.3.1/api/pyplot_api.html#matplotlib.pyplot.plot`.



Para mostrar una sección específica del plano donde estamos graficando, debemos utilizar la función `pyplot.axis([xmin, xmax, ymin, ymax])` antes de ejecutar `pyplot.plot()`, indicando el valor mínimo y máximo en los ejes X e Y. Aplicando esto:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, sin(x))
pyplot.plot(x, tan(x))
pyplot.axis([0, 10, -5, 5])
pyplot.show()
```

Se genera el gráfico:

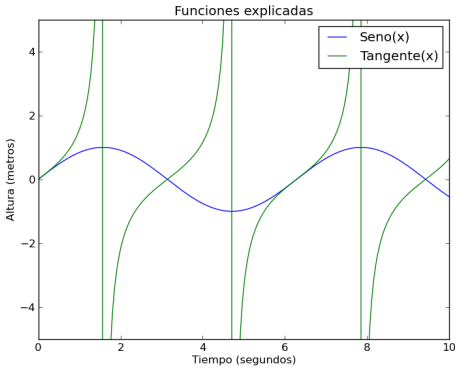


Nuestros gráficos también se pueden decorar con un título, con texto para el eje X y para el eje Y y con una leyenda explicando cada serie. Para esto tenemos que utilizar distintas funciones. Por ejemplo:

```
x = linspace(0, 10, 1000)
pyplot.plot(x, sin(x))
pyplot.plot(x, tan(x))
pyplot.axis([0, 10, -5, 5])
pyplot.xlabel('Tiempo (segundos)')
pyplot.ylabel('Altura (metros)')
# Debe mantener el orden de las series
```

```
pyplot.legend(['Seno(x)', 'Tangente(x)'])
pyplot.title('Funciones explicadas')
pyplot.show()
```

Genera el gráfico:



13.7. Varios gráficos en una misma ventana

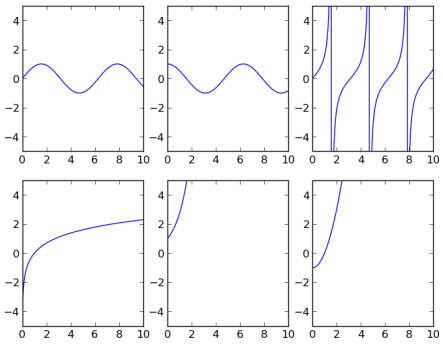
Podemos mostrar también varios gráficos en una sola ventana. En este caso se organizarán en una cuádrícula rectangular. Para posicionar cada gráfico en esta cuadrícula, debemos indicar mediante la función `subplot(nFilas, nColumnas, nPlot)` la cantidad de filas y columnas que tiene la cuadrícula, además de la posición del gráfico dentro de la cuadrícula. **Las posiciones en esta cuadrícula están numeradas desde 1 en adelante**. Por ejemplo:

```
x = linspace(0, 10, 1000)
pyplot.subplot(2, 3, 1)
pyplot.plot(x, sin(x))
pyplot.axis([0, 10, -5, 5])
pyplot.subplot(2, 3, 2)
pyplot.plot(x, cos(x))
pyplot.axis([0, 10, -5, 5])
pyplot.subplot(2, 3, 3)
pyplot.plot(x, tan(x))
pyplot.axis([0, 10, -5, 5])

pyplot.subplot(2, 3, 4)
pyplot.plot(x, log(x))
pyplot.axis([0, 10, -5, 5])
pyplot.subplot(2, 3, 5)
pyplot.plot(x, exp(x))
pyplot.axis([0, 10, -5, 5])
pyplot.subplot(2, 3, 6)
pyplot.plot(x, x**2 - 1)
pyplot.axis([0, 10, -5, 5])
```

```
pyplot.show()
```

Genera el gráfico:



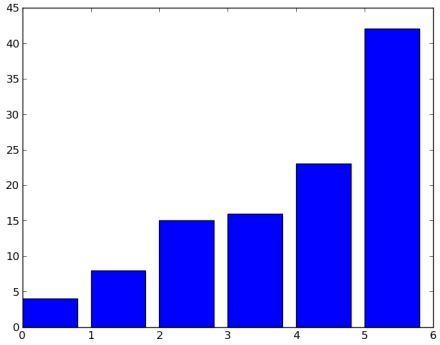
13.8. Otros tipos de gráficos

13.8.1. Gráficos de barras

Un gráfico de barras está compuesto por una secuencia de valores, donde cada valor representa el alto de una columna. Para generar uno debemos tener una secuencia de números de las columnas que queremos graficar, además de una secuencia de números con las alturas de cada una y pasarlas como parámetro a la función `pyplot.bar(columnas, valores)`. Por ejemplo:

```
cols = range(6)
valores = [4, 8, 15, 16, 23, 42]
pyplot.bar(cols, valores)
pyplot.show()
```

Genera el gráfico:

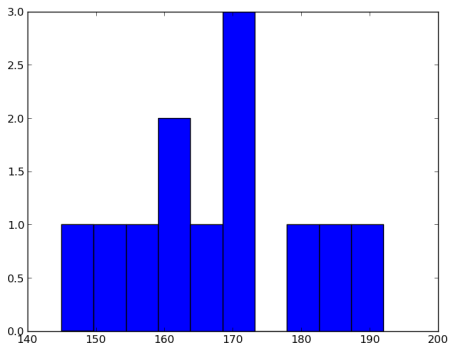


13.8.2. Histogramas

Un histograma es un tipo de gráfico utilizado para agrupar valores según el rango de valores en el que se encuentran. Son muy utilizados para estadística. Para generar uno de estos solamente necesitamos una lista o arreglo de números para pasar como parámetro a la función `pyplot.hist(valores)`. Por ejemplo:

```
alturas = [150,184,164,192,182,170,171,162,163,145,157,170]
pyplot.hist(alturas)
pyplot.show()
```

Genera el gráfico:

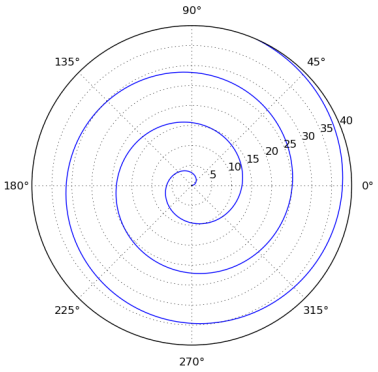


13.8.3. Gráficos en coordenadas polares

Hay gráficos que son más simples de dibujar utilizando coordenadas polares en lugar de cartesianas. En el plano polar, cada punto está definido por su distancia al origen y el ángulo que forma con el eje. Para graficar en coordenadas polares, se utiliza la función `pyplot.polar(angulo, radio)`, pasando como parámetros una lista con los ángulos y una lista con los radios de los puntos a graficar. Por ejemplo:

```
t = linspace(0, 20, 1000)
r = 2*t
pyplot.polar(t, r)
pyplot.show()
```

Genera el gráfico:



13.9. Ejercicios

13.9.1. Básicos

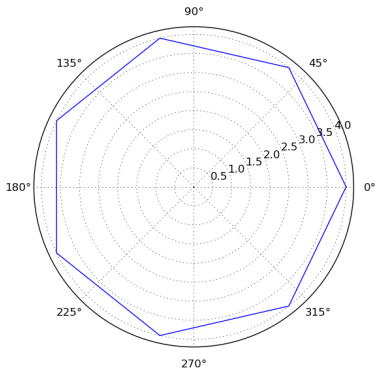
1. Escriba programas que grafiquen cuadrados o rectángulos donde el largo de los lados esté dado por el usuario.

2. Escriba un programa que grafique triángulos, donde el usuario ingrese las coordenadas de las tres esquinas del triángulo.

13.9.2. Sugeridos

1. Escriba un programa que lea las notas de un curso en una prueba desde un archivo `notas.txt`, donde cada línea contenga un rut y una nota, separados por un espacio. Luego grafique un histograma con la distribución de notas entre cada par de números enteros consecutivos.

2. Escriba un programa que pida al usuario un radio y un número entero mayor o igual a 3. Luego genere un gráfico con un polígono regular con el número de lados indicado por el usuario inscrito en un círculo del radio especificado. Por ejemplo si el usuario indica un radio 4 y un número de lados 7, deberá generar:



Capítulo 14

Recursividad

14.1. Definición

Para entender el concepto de recursividad, lo primero que se debe hacer es comprender el concepto de recursión. Por otra parte, para entender la recursión, antes se debe conocer la recursividad.

Ejemplo de imagen recursiva (`http://es.xkcd.com/strips/auto-descripcion/`).

14.2. Recursividad

A nivel de código, una función se dice recursiva si puede resolver algunos casos de su input por sí misma y para resolver el resto de los posibles inputs se llama a ella misma dentro de su código.

Por ejemplo, la secuencia de Fibonacci se define de la siguiente manera:

$$F(x) = \begin{cases} 0 & , x = 0 \\ 1 & , x = 1 \\ F(x-1) + F(x-2) & , x > 1 \end{cases}$$

Podríamos tratar de escribir una función que calcule la serie de Fibonacci usando ciclos y variables auxiliares, pero el código se empezaría a complicar. Una forma más simple de calcular la serie de Fibonacci es aplicar recursividad de la siguiente forma:

```
def Fibonacci(x):
    # Caso base
    if x <= 1:
        return x
    # Llamada recursiva
    return Fibonacci(x-1) + Fibonacci(x-2)
```

La lógica de la función en Python que calcula los términos de la serie funciona de la siguiente manera:

- Define un **caso base**: para los valores de x menores a `1`.
- Utiliza su misma definición para resolver los casos más complicados.

Analizando con detalle una llamada a esta función, una llamada que calcule el valor de `Fibonacci(4)` realizará muchas llamadas a la misma función para calcular los valores que necesita de la siguiente forma:

```
Fibonacci(4)
├─ Fibonacci(3)
│  ├─ Fibonacci(2)
│  │  ├─ Fibonacci(1)
│  │  └─ Fibonacci(0)
│  └─ Fibonacci(1)
└─ Fibonacci(2)
   ├─ Fibonacci(1)
   └─ Fibonacci(0)
```

14.2.1. Necesidad del caso base

En una función recursiva el caso base es imprescindible, ya que sin este un programa que ejecute la función no terminaría nunca. Por ejemplo, si escribimos una función para calcular una cuenta regresiva:

```
def cuentaRegresiva(x):
    print x
    cuentaRegresiva(x-1)
```

Y la ejecutamos pasando como parámetro el valor 5 veremos en la consola:

```
5
4
3
2
1
0
-1
-2
.
.
.
# muchas líneas más tarde
```

```
.
.
.
RuntimeError: maximum recursion depth exceeded
```

Esto pasa porque la función indica que siempre debe ejecutarse con un valor menor y esta secuencia de llamadas a la misma función no termina nunca. El error que aparece al final, `RuntimeError: maximum recursion depth exceeded` se debe a que Python tiene como límite un máximo de 1000 lamadas recursivas[1].

Para corregir nuestra función debemos definir uno o más casos base, que se resuelven sin necesidad de recursividad. Luego debemos asegurar que toda llamada a la función que ejecute recursión llegue siempre a alguno de estos casos. En nuestro ejemplo podríamos definir que cuando la cuenta llegue a 0 se debe terminar la función:

```
def cuentaRegresiva(x):
    if x == 0:
        print '¡Despegue!'
        return
    print x
    cuentaRegresiva(x-1)
```

Esta función todavía puede quedarse pegada si se ejecuta pasando como parámetro un número negativo. Podemos corregir esto cambiando la condición del caso base de `if x == 0:` a `if x <= 0:`, para que la función termine instantámente con un número negativo.

14.3. Algortimos de ordenación recursivos

Existen algortimos de ordenación recursivos. Estos tienen la ventaja de ordenar más rápido[2] una lista de elementos, con la desventaja de ser más complejos de entender.

14.3.1. Quick sort

Este algoritmo ordena una lista posicionando correctamente un elemento cada vez y luego ejecutándose para cada sublista que no contenga ese elemento.

Para empezar su definición, debemos determinar los casos base: una lista vacía o con 1 elemento está ordenada. Teniendo los casos base identificados, podemos definir nuestro algortimo recursivo:

- Tomamos un elemento cualquiera de la lista y lo llamamos pivote.
- Recorremos la lista, dejando los elementos menores al pivote a su izquierda y el resto a su derecha.
- Aplicamos quick sort a las sublistas a la izquierda y a la derecha del pivote.

A nivel de código, podemos verlo de la siguiente forma:

[1] Esto se puede cambiar importando `sys` y ejecutando la función `sys.setrecursionlimit(nuevoMaximo)`, pero no es buena idea.

[2] Dependiendo del algoritmo esto puede ser para todos los casos o solo para algunos.

```
def quicksort(lista, iz, der)
    # revisar casos base
    if der - iz + 1 <= 1:
        return

    # elegir un pivote (en este caso, el elemento central
    pivote = lista[(iz + der) / 2]
    i = iz, j = der
    # mover los elementos a la izquierda o derecha del pivote
    while i < j:
        while x[i] < pivote:
            i += 1
        while x[j] > pivote:
            j -= 1
        if i < j:
            temp = x[i]
            x[i] = x[j]
            x[j] = temp
            i += 1
            j -= 1

    # aplicar el paso recursivo
    quicksort(lista, iz, j)
    quicksort(lista, j + 1, der)
```

En esta implementación, se utiliza como pivote el elemento central del subarreglo a ordenar. Los parámetros `iz` y `der` indican los índices del subarreglo que estamos ordenando dentro de cada ejecución.

14.3.2. Merge sort

Merge sort es otro algoritmo de ordenación recursivo. También toma como casos base que las listas de largo 0 y de largo 1 están ordenadas. Su funcionamiento general dice lo siguiente:

- Ordena la primera mitad de la lista.
- Ordena la segunda mitad de la lista.
- Mezcla las dos mitades ya ordenadas.

La parte de mezclar las dos listas es la más importante del algoritmo. Podemos verlo en código aquí:

```
def mergeSort(lista, desde, hasta):
    largo = hasta - desde
    ordenada = []
    if largo == 1:
        ordenada.append(lista[desde])
    else:
        # Ordenamos mitades
```

```
        a = mergeSort(lista, desde, desde + largo/2)
        b = mergeSort(lista, desde + largo/2, hasta)
        # Mezclamos
        enA = 0
        enB = 0
        for i in range(largo):
            if enB==len(b) or (enA<len(a) and a[enA]<b[enB]):
                # - ya se acabó b
                # - no se ha acabado a, y el de a es menor que el de b
                ordenada.append(a[enA])
                enA += 1
            else:
                ordenada.append(b[enB])
                enB += 1
    return ordenada
```

14.4. Ejercicios

14.4.1. Básicos

1. Analice el código de la siguiente función que suma dos enteros positivos:

```
def suma(x, y):
    if y == 0:
        return x
    return suma(x+1, y-1)
```

   a) ¿Cuál es el caso base?

   b) ¿Qué pasa si se ejecuta con un `x` negativo?

   c) ¿Qué pasa si se ejecuta con un `y` negativo?

   d) ¿Cómo la cambiaría para que funcione con números enteros positivos y negativos?

2. Escriba una función recursiva que multiplique dos enteros positivos. Para esto no puede utilizar el operador `*`, solamente sumas y llamadas recursivas a la misma función.

3. Escriba una función recursiva que calcule la suma de todos los números enteros desde 1 hasta un `n` entero positivo.

14.4.2. Sugeridos

1. Escriba una función recursiva que imprima un triángulo formado por asteriscos en la consola. La función debe recibir dos parámetros, la cantidad de asteriscos a imprimir en el nivel actual y el ancho máximo del triángulo. Por ejemplo, al ejecutar `imprimirTriangulo(1, 5)` se debe mostrar:

```
*
**
```

```
***
****
*****
****
***
**
*
```

2. Escriba una función recursiva que permita determinar si un string es palíndromo[3] o no. No debe usar ciclos ni métodos de string como `reverse()`.

14.4.3. Desafiantes

1. Escriba un programa que genere un archivo de texto con el nombre `copia.py`, el contenido del archivo copiado debe ser el código del programa que se está ejecutando.

[3] Se lee igual de izquierda a derecha y de derecha a izquierda, como radar.

Capítulo 15

Simulación

15.1. Introducción

Muchas veces, se quiere usar herramientas computacionales para predecir el comportamiento de algún *sistema* en el mundo real. Algunos de ellos se pueden resolver matemáticamente (por ejemplo, ¿cuánto se deformará un resorte bajo una fuerza determinada?, o ¿cuánto demora en caer una piedra desde 1km de altura?). Sin embargo, otros son muy difíciles o prácticamente imposibles de resolver directamente (por ejemplo, ¿cuánta fuerza puede soportar una vértebra con un tumor?, o ¿cuánto se demora la gente en llegar de su casa a su trabajo si se corta la luz en todo Santiago?).

Para esto, usaremos **simulación**. Podemos dividir la simulación en dos grandes clases: determinística y estocástica. La simulación determinística es la que simula un modelo que no depende de ninguna variable aleatoria, es decir, podemos conocer todos los factores que influyen, con certeza absoluta. Por otro lado, la simulación estocástica se basa en modelos que cuentan con variables aleatorias, que es imposible conocer de antemano. Obviamente, no podemos simular un sistema si no sabemos nada sobre algún factor importante, por eso, en general, conocemos, o podemos estimar algo sobre las variables aleatorias: su distribución. Con esto, podremos generar datos *parecidos* a los que veríamos en la realidad. Por ejemplo, sabemos que entre las 7am y las 8am llegan en promedio 600 personas a una estación de metro. No sabemos si llegan todas juntas, o si llegan en grupos de 10 cada minuto, o de alguna otra forma. Entonces, podemos generar la variable aleatoriamente de modo que se comporte *parecido* a la realidad.

En esta unidad nos enfocaremos en simulaciones estocásticas, es decir, con variables aleatorias.

15.2. Modelo de una Simulación

En la realidad, un proceso cualquiera normalmente se ve afectado por miles y miles de factores. ¿Cómo camina la gente por el paseo Ahumada? Depende de la temperatura, del viento, de cómo durmió Andrés Ramírez anoche, de cuánto se demoró Miguel Antúnez en tomar desayuno, ... en fin, es imposible considerar todas las variables. Por eso, es necesario crear un buen **modelo** del proceso.

Para crear un modelo, primero hay que pensar qué se quiere simular. Una vez respondida

esa pregunta, tenemos que decidir qué queremos medir en la simulación (qué datos queremos obtener al simular este proceso). Con esas cosas en mente, nos fijaremos en 4 componentes del proceso para definir un buen modelo: los elementos o actores que participan en la simulación, el tiempo, los eventos que influyen en el estado del sistema, y los parámetros que determinan cómo ocurren los eventos.

15.2.1. Elementos o Actores

Los elementos son los participantes en la dinámica de la simulación. Pueden ser cosas concretas, personas, conceptos, etc. Los elementos pueden influir en el estado de la simulación. También pueden contener el estado, o una parte de él. Su comportamiento podría depender de alguna variable o estado del sistema, e incluso podrían cambiarlo durante la simulación.

Para modelar los elementos en la simulación, hay que fijarse qué cosas influyen en ella, cómo influyen, de qué depende su modo de influencia, etc.

15.2.2. Tiempo

Los procesos que simularemos toman cierto tiempo. Son sistemas dinámicos que evolucionan. Es importante analizar bien cómo modelar el tiempo en la simulación.

En primer lugar, no realizaremos simulaciones infinitas: necesitamos saber hasta cuándo simular. Sea un límite temporal (simular una semana de duración, unas horas, algunos milisegundos), de condiciones (hasta atender 1000 clientes, hasta que se vendan todos los productos, hasta que se caiga el puente), u otro tipo de límite.

También, hay que analizar la escala del tiempo y la resolución con la que lo mediremos. Si simulamos el movimiento de los glaciares y su evolución en los siglos, probablemente no vamos a querer simular con precisión de microsegundos, sino años o décadas. Si vamos a simular colisiones de partículas subatómicas, no nos interesa saber qué pasará en un plazo de 10 años, sino que querremos medir con mucha precisión el tiempo, pero por un intervalo muy corto. Así, depende del proceso a simular qué medidas de tiempo son más apropiadas.

15.2.3. Eventos

¿Qué eventos pueden ocurrir durante la simulación? Es necesario identificar cuáles de ellos son relevantes, y afectan el estado del modelo. Todos los demás podemos ignorarlos. Por ejemplo, no es importante si un pasajero de un bus recibe una llamada telefónica, pero sí es importante cuando sale un bus del terminal.

En general, los eventos serán interacciones entre dos elementos, y como resultado del evento podría cambiar el estado de los elementos, e incluso el estado de otras cosas. Además, los eventos normalmente son de interés para las mediciones que queramos realizar durante la simulación.

15.2.4. Parámetros

Los parámetros determinan cómo se comportarán los elementos, o la ocurrencia de los eventos. Son parte del modelo, y son constantes durante toda la simulación. Ejemplos típicos de parámetros son: distribuciones de probabilidad, datos obtenidos de mediciones del mundo real, suposiciones o estimaciones del modelo.

15.3. Programando una Simulación

Las simulaciones de tiempo discreto, como las usaremos en este curso, normalmente se programan según el análisis realizado de los 4 elementos descritos anteriormente: elementos, tiempo, eventos y parámetros.

Los elementos se guardan en variables, siguiendo una estructura ordenada. Podemos usar diccionarios para guardar distintos atributos de cada elemento (por ejemplo, al simular una clínica podemos guardar la enfermedad del paciente, la hora a la que ingresó, el tipo de cobertura de salud que tiene, etc.).

El tiempo se refleja en un ciclo, donde en cada iteración se avanza el "reloj" en un intervalo determinado, por ejemplo 1 segundo, 3 horas, o 10 años. Esta es la resolución con que podremos medir el tiempo, y todos los eventos ocurrirán en alguno de estos instantes. Dentro de este ciclo de tiempo ocurrirán todos los eventos, e irán cambiando las variables del estado del sistema, y los elementos.

Los eventos, según cómo ocurran, deberemos decidirlo dentro del ciclo del tiempo. Si son eventos probabilísticos, tendremos que evaluar las probabilidades de que ocurra en cada instante, y según una variable aleatoria gatillar, o no, el evento.

Los parámetros, por último, irán incluidos en todos las partes de la simulación, pues determinarán la forma en que se comporten los elementos, el paso del tiempo, y la ocurrencia de los eventos.

15.4. Ejercicio de ejemplo - Supermercado

15.4.1. Enunciado

Hay un supermercado con 5 cajas. Está abierto las 24 horas, y queremos simular todo un día, partiendo con todo vacío (sin clientes adentro, ni en las filas de las cajas).

Cada minuto, llegan entre 0 y 3 clientes. Cada cliente llevará entre 5 y 20 productos. El tiempo que demora el cliente, entre que llega al supermercado, y se pone en la fila de la caja es de 1 minuto por producto, más entre 3 y 6 minutos adicionales.

Al momento de elegir la caja, el cliente elige la caja que tenga menos productos en su fila. Esto es, la suma de la cantidad de productos que tenga cada cliente que está en la fila.

La atención de un cliente en la caja toma 3 minutos, más un tiempo entre 0.5 y 1.5 minutos por producto.

Nota: todos los rangos son con distribución uniforme.

Al final del día, entregue los siguientes datos:

- El total de clientes que entraron al supermercado.
- La cantidad de clientes que, al final del día, estaban en la sala de ventas (entraron pero no alcanzaron a elegir todos los productos ni a ponerse en la cola de alguna caja).
- Para cada una de las cajas, muestre, además:
  - El total de clientes atendidos.
  - El total de productos que pasaron por la caja.
  - El promedio del tiempo de atención a clientes.
  - El largo de la fila de esa caja al terminar el día.

15.4.2. Análisis

Elementos

Rápidamente identificamos 2 elementos: clientes y cajas. Los clientes llegan, demoran un tiempo, se ponen en la caja, etc. Las cajas atienden a los clientes, tienen una cola, se demoran, etc.

En esta simulación en particular, los productos del supermercado influyen solamente según su cantidad en cada compra. No nos interesa su precio, tamaño, stock ni ningún otro dato de ellos, por lo que no es necesario considerarlos como elementos.

No hay más elementos relevantes.

Tiempo

Vemos que la mayoría de las cosas está especificada en minutos. El rango de los tiempos permitiría efectuar nuestra simulación en segundos, o en intervalos de 10 segundos, de 30, o de un minuto. Elegiremos minutos.

La simulación debe terminar después de un día, es decir 1440 minutos.

Eventos

Los eventos que pueden ocurrir son:

- Llega un cliente al supermercado y empieza a elegir productos.
- Un cliente termina de elegir productos y se pone en una cola de caja.
- Una caja termina de atender a un cliente y queda disponible.
- Una caja disponible empieza a atender a un cliente en la cola.

Cada uno de estos eventos cambia el estado del sistema, afectando a los actores involucrados. Al llegar un cliente, podemos calcular (aleatoriamente) cuánto se va a demorar. Sabremos, entonces, que si llegó en el minuto X de la simulación, y demorará Y minutos en elegir productos, cuando el "reloj" marque $X + Y$ minutos, el cliente se pondrá en la fila de una caja.

De la misma manera, cuando una caja empieza a atender a un cliente, en el minuto X, y calculamos que demorará Y, al llegar el minuto $X + Y$ sabemos que la caja terminará con ese cliente y puede atender a otro.

Parámetros

Los parámetros son todos los datos que son dados: que hay 5 cajas, que los clientes llegan entre 0 y 3 por minuto, que cada uno lleva entre 5 y 20 productos, etc. Estos estarán incorporados en el código de la simulación, determinando el comportamiento de los elementos y la ocurrencia de los eventos.

15.4.3. Solución

```
from numpy import random

clientesComprando = []
cajas = []
for i in range(5):
    caja = {
        'fila': [],
        'productosVendidos': 0,
        'clientesAtendidos': 0,
        'tiempoFinal': None,
        'tiemposAtencion': []
    }
    cajas.append(caja)

totalClientes = 0

# t indica el minuto actual
for t in range(24*60):
    # Llegan clientes
    for n in range(random.randint(4)):
        productos = random.randint(5, 21)
        cliente = {
            'productos': productos,
            'tiempoFin': t + productos + random.randint(3, 7)
        }
        clientesComprando.append(cliente)
        totalClientes += 1

    for cliente in clientesComprando:
        # Cliente termina de comprar
        if cliente['tiempoFin'] == t:
            clientesComprando.remove(cliente)
            # Buscamos la caja con menos productos
            min = cajas[0]
            menorTotal = 0
            for cliente in cajas[0]['fila']:
                menorTotal += cliente['productos']
            for caja in cajas:
                total = 0
                for cliente in caja['fila']:
                    total += cliente['productos']
                if total < menorTotal:
                    menorTotal = total
                    min = caja
            min['fila'].append(cliente)
```

```
    for caja in cajas:
        # Caja termina de atender un cliente
        if caja['tiempoFinal'] == t:
            caja['tiempoFinal'] = None
            caja['clientesAtendidos'] += 1
            cantidadProductosCaja  = caja['fila'][0]['productos']
            caja['productosVendidos'] += cantidadProductosCaja
            caja['fila'].pop(0)

        # Caja empieza a atender un cliente
        if caja['tiempoFinal'] == None and len(caja['fila']) > 0:
            cliente = caja['fila'][0]
            tiempoVariable = random.random())*cliente['productos']
            tiempoAtencion = 3 + int((0.5 + tiempoVariable)
            caja['tiempoFinal'] = t + tiempoAtencion
            caja['tiemposAtencion'].append(tiempoAtencion)

print 'En total llegaron', totalClientes, 'clientes.'
print len(clientesComprando), 'no terminaron de comprar.'

i = 1
for caja in cajas:
    print 'Resumen caja', i
    print '  Atendió', caja['clientesAtendidos'], 'clientes.'
    print '  Vendió', caja['productosVendidos'], 'productos.'
    print '  Quedaron', len(caja['fila']), 'clientes en la fila.'
    t = sum(caja['tiemposAtencion'])/len(caja['tiemposAtencion'])
    print '  Promedio de tiempo en atender', t

    i += 1
```

Capítulo 16

Temas avanzados

16.1. Funciones como variables

Ya hemos visto que para definir una función en Python se utiliza la palabra reservada `def`, después se indica el nombre de la función, seguida por un par de paréntesis, dentro de los cuales hay una lista con los nombres de los parámetros que la función necesita.

Sabemos también que no pueden existir dos variables con el mismo nombre. De la misma manera, no pueden existir dos funciones con el mismo nombre. Un detalle no menor de esto es que no puede haber una variable que tenga el mismo nombre que una función. Por ejemplo, si ejecutamos el siguiente trozo de código:

```
def f(x):
    return x + 5

f = 15
a = f(2)
```

Obtendremos el error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Este error se debe a que al escribir la línea `f = 15` indicamos que `f` es un número y ya no una función y no se puede utilizar una variable numérica como una función.

Aprovechando que los nombres de las funciones se utilizan como variables, es posible guardar una función en una variable, por ejemplo en:

```
def sucesor(x):
    return x + 1

siguiente = sucesor
a = siguiente(23) # a toma el valor 24
```

En el fragmento de código anterior, la variable `siguiente` es efectivamente otro nombre para la misma función que definimos como `sucesor` y se puede utilizar de la misma forma.

Esto abre nuevas posibilidades en Python, ya que si podemos guardar funciones en variables, también podríamos pasar estas variables que guardan funciones como parámetros a otras funciones.

16.2. Map

La función `map(funcion, lista)` permite generar una lista de elementos a partir de una lista de valores iniciales y de una función que trabaje con un único parámetro. Por ejemplo, para generar una lista con los cubos de los números del 0 al 9, podemos escribir:

```
def cubo(x):
    return x**3

base = range(10)
listaDeCubos = map(cubo, base)
print listaDeCubos # imprime [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

La función `map` permite ahorrar tiempo al escribir el código, ya que con ella no es necesario escribir explícitamente el ciclo que genera nuestra lista de cubos. Como podemos ver aquí:

```
def cubo(x):
    return x**3

base = range(10)
listaDeCubos = []
for i in base:
    listaDeCubos.append(cubo(i))
print listaDeCubos # imprime [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

16.3. Reduce

La función `reduce(funcion, lista)` genera un único resultado a partir de una lista. Para utilizar `reduce` necesitamos una función que trabaje con 2 parámetros. El resultado que se obtiene consiste en ejecutar la función con los dos primeros valores de la lista. Luego se calcula el valor de la función pasando como parámetros el resultado obtenido anteriormente y el tercer elemento de la lista y se sigue así hasta operar con todos los valores. Por ejemplo:

```
def promedio(x, y):
    return (x+y)/2.0

base = [4, 8, 10, 0, 7]
resultado = reduce(promedio, base)
print resultado # imprime 5.5
```

El resultado anterior se calcula de la siguiente forma:

- Se calcula `promedio` entre 4 y 8 (los dos primeros elementos de la lista), obteniendo 6.
- Se calcula `promedio` entre el 6 obtenido anteriormente y 10, obteniendo 8.

- Se calcula `promedio` entre el 8 obtenido anteriormente y 0, obteniendo 4.
- Se calcula `promedio` entre el 4 obtenido anteriormente y 7, obteniendo 5,5. Este es el resultado final.

16.4. Filter

La función `reduce(funcion, lista)` permite obtener un subconjunto de una lista original, utilizando una función que retorne `True` o `False` para el parámetro que recibe. Por ejemplo, si queremos obtener solamente los números pares de una lista, podemos escribir:

```
def esPar(x):
    return x%2 == 0

base = range(15)
resultado = filter(esPar, base)
print resultado # imprime [0, 2, 4, 6, 8, 10, 12, 14]
```

Es importante recordar que las funciones `map`, `reduce` y `filter` se pueden usar para trabajar con listas de cualquier tipo de datos, no solamente para listas de números. Por ejemplo, si tenemos una lista de diccionarios, donde cada diccionario representa una persona guardando su nombre, edad y dinero. Podríamos escribir un programa que imprima los nombres de las personas que pueden ingresar a una fiesta para mayores de 18 que cobra una entrada de $5000 de la siguiente forma:

```
def puedeEntrar(persona):
        if persona['edad'] < 18:
                return False
        if persona['dinero'] < 5000:
                return False
        return True

def nombre(persona):
        return persona['nombre']

personas = [
        {'nombre': 'Juan', 'edad':20, 'dinero':1000},
        {'nombre': 'Ana', 'edad':22, 'dinero':5000},
        {'nombre': 'Pedro', 'edad':16, 'dinero':7000},
        {'nombre': 'Elisa', 'edad':21, 'dinero':4000},
        {'nombre': 'Claudio', 'edad':15, 'dinero':9000},
        {'nombre': 'Julia', 'edad':23, 'dinero':2000},
        {'nombre': 'Diego', 'edad':19, 'dinero':8000},
        {'nombre': 'Carmen', 'edad':18, 'dinero':6000}
]

entran = filter(puedeEntrar, personas)
nombres = map(nombre, entran)
print nombres # imprime ['Ana', 'Diego', 'Carmen']
```

Nótese que al utilizar las funciones `map` y `filter`, no es necesario escribir ningún ciclo de forma explícita en el código.

16.5. Lambdas

El operador `lambda` permite definir funciones sin la necesidad de darles un nombre. Esto permite simplificar aún más la notación de `map`, `reduce` y `filter` al no necesitar definir una función explícita para realizar la operación.

La sintaxis para declarar una función lambda es:

```
lambda listaDeParametros : expresion
```

Por ejemplo, para generar los cuadrados de los números del 0 al 9, podemos escribir:

```
cuadrados = map(lambda x : x**2, range(10))
print cuadrados # imprime [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

16.6. Listas por comprensión

Las listas por comprensión (o *comprehensions* en inglés) son una forma de declarar listas usando una notación similar a la de conjuntos en matemáticas. Por ejemplo, el conjunto de los cuadrados menores a 12 se puede definir como $C = \{x^2 : x \in \mathbb{N} \wedge x < 12\}$.

Podemos crear una lista en Python que contenga estos mismos números utilizando un ciclo o con el operador `map` de la siguiente forma:

```
c1 = []
for i in range(12):
    c1.append(i**2)

c2 = map(lambda x : x**2, range(12))
```

Tambien podemos declarar esta misma lista por comprensión de la siguiente manera:

```
c3 = [i**2 for i in range(10)]
```

La declaración de una lista por comprensión consiste en indicar la transformación que se va a realizar sobre cada elemento de otra lista, que se recorre con un ciclo `for`. Esta notación es particularmente cómoda cuando se tiene un string con varios números separados por espacios y se quiere convertir en una lista de números:

```
linea = '4 8 15 16 23 42'
numeros = [int(i) for i in linea.split()]
```

La función que aplicamos a cada elemento de la lista original debe recibir un parámetro y retornar un valor. La lista que usamos de base para la generación puede tener cualquier tipo de contenido. Por ejemplo, para dejar una lista de nombres con la primera letra en mayúscula y el resto en minúsculas podemos aplicar:

```
nombres = ['JUAN', 'ana', 'PeDRo', 'ElIsA', 'ClAuDiO']
corregidos = [n[0].upper()+n[1:].lower() for n in nombres]
print corregidos # imprime ['Juan', 'Ana', 'Pedro', 'Elisa', 'Claudio']
```

16.7. Ordenar listas según un criterio específico

Hemos visto que las listas en Python proveen el método `sort()` para ordenar su contenido. Este utiliza los operadores `<`, `==` y `>` para ordenar los elementos de la lista de menor a mayor.

Es posible también pasar una función de comparación como parámetro a `sort()` para que el ordenamiento se realice según nuestro criterio. Para esto, la función indicada debe recibir dos parámetros y retornar un número entero como respuesta. El número retornado debe ser cero si los dos parámetros son iguales según nuestro orden, negativo si el primero va antes o positivo si el primero va después.

Trabajando con listas de números, la función de comparación que los ordena de menor a mayor es:

```
def compara(x, y):
    return x - y
```

Si queremos utilizarla para ordenar una lista con ella de forma explícita, debemos pasarla como parámetro a `sort()` de la siguiente forma:

```
def compara(x, y):
    return x - y

lista = [3, 4, -2, 1, 0, -5, 6, 7, 3]
lista.sort(compara)
```

Si queremos ordenar los números de menor a mayor, pero poniendo los pares primero y los impares después, podemos usar la siguiente función:

```
def f(x, y):
    if x%2==0 and y%2==0:
        return x-y
    if x%2==0:
        return -1
    if y%2==0:
        return 1
    return x-y

lista = [4, 6, 3, 4, 2, -7, 0, 3, 2, 1]
lista.sort(f)
print lista # imprime [0, 2, 2, 4, 4, 6, -7, 1, 3, 3]
```

16.8. Ejercicios

16.8.1. Básicos

1. Escriba un programa que lea una matriz desde un archivo. Cada línea del archivo representa una fila de la matriz, con los números separados por espacios. Utilize `map` o comprensiones para generar la matriz de números.

16.8.2. Sugeridos

1. Escriba un programa que pida una lista de palabras al usuario. Luego ordénelas según su largo de mayor a menor e imprima la lista ordenada.

2. Escriba un programa que trabaje con una lista de diccionarios. Cada diccionario representa a una persona y debe guardar los siguientes datos: nombre, altura y peso. Su programa debe imprimir los nombres de las personas de la lista, ordenados según su índice de masa corporal. El índice de masa corporal (IMC) se calcula como el peso en kilos dividido en el cuadrado de la altura en metros de una persona.