Python - Guia de clases

Ing. Pedro Muñoz del Rio

pmunoz@gmail.com

Lima, Perú

El curso para el cuál este documento ha sido creado es una introducción para programadores al lenguaje de programación Python y es parte de una serie de cursos cuyo fin es introducir al lector en el desarrollo de sofware mediante esta potente herramienta.

Los objetivos del curso son familiarizar al estudiante con los principales elementos del lenguaje y enseñar a pensar de forma "pythonica", de tal manera que pueda empezar a elaborar sus propios programas.

Al terminar las clases, el alumno debe conocer y comprender los componentes fundamentales del lenguaje de programación y estar en capacidad de pensar y resolver problemas utilizando Python.

La presente Guía es un complemento a las clases dictadas de forma presencial. Se ha redactado para servir como hoja de ruta en el transcurso de las sesiones y como estructura del contenido a dictarse.

Índice

Lista de Gráficos

Capítulo 1

Programar

1.1. Programación en Python

El objetivo de este curso es enseñar a utilizar como herramienta el lenguaje de programación Python para usuarios con conocimientos previos de programación.

1.1.1. Codificación de las instrucciones

Para que una computadora lleve a cabo alguna acción, debe leer las instrucciones guardadas en la memoria y ejecutarlas en el CPU, tal como se muestra en la imagen 1.1[1].

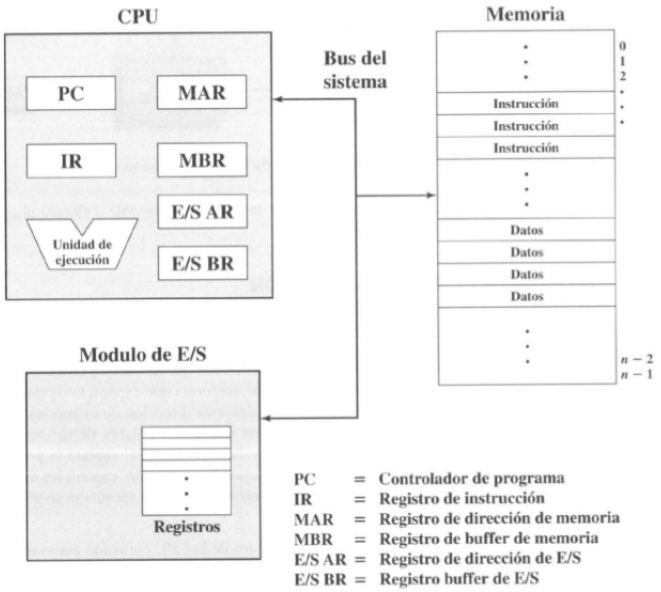

Figura 3.2. Componentes del computador: esquema de dos niveles.

Gráfico 1.1: Componentes del computador

Estas instrucciones están codificadas en código binario (ceros y unos), también llamado *lenguaje binario*. No solo las operaciones aritméticas, sino también los símbolos utilizados por las computadoras son representados por el estándar ASCII (American Standard Code for Information Interchange).

[1] [5] pp. 61

La correspondencia entre secuencias de bits y caracteres determinada por la tabla es arbitraria. La letra "a", por ejemplo, se codifica con la secuencia de bits 01100001 y la letra "A"se codifica con 01000001. Todo texto se puede codificar como una secuencia de bits, por ejemplo el texto "Hola" es:

01001000 01101111 01101100 01100001

En la pantalla de texto (el famoso "modo DOS" o más propiamente dicho "Interfaz de Linea de Comandos") se ven los símbolos ASCII originales, pero en las pantallas gráficas, ahora utilizadas de forma masiva, lo que se ven son símbolos gráficos formados por pixeles, donde el color de cada pixel está dado por una combinación de ceros y unos.

1.1.2. Programar

Tal como se mencionaba en la sección anterior, las instrucciones para que una computadora realice alguna acción están almacenadas en memoria. La memoria es una especie de "almacén" con cajones numerados donde se almacenan instrucciones para el CPU. Cada "cajón", está numerado o mejor dicho, tiene su propia dirección de memoria.

La CPU lee las direcciones memoria en forma secuencial (a menos que una instrucción le indique lo contrario) y ejecuta lo que cada instrucción le señala. Cada instrucción describe acciones simples como sumar dos direcciones de memoria, copiar una dirección en otra, empezar a leer a partir de una determinada dirección y así por el estilo.

Combinando instrucciones se pueden lograr resultados complejos, y esas instrucciones son un programa. La secuencia de instruccciones que la CPU puede ejecutar se llaman *código de máquina*, el cuál codifica instrucciones en secuencia de unos y ceros que hacen al CPU ejecutar determinadas acciones.

1.1.3. Lenguaje ensamblador

En el inicio de los tiempos, cuando las computadoras eran gigantes que ocupaban pisos enteros, hechas de cables y tubos de vacio, hacer un programa implicaba volver a cablear, donde cada cable en un panel de conectores significaba un uno y el cable desconectado era un cero. Para evitar esta labor tediosa, se crearon abreviaciones para conjuntos de instrucciones, estos *códigos mnemotécnicos* son abreviaturas fáciles de recordar.

Por ejemplo un programa que busque obtener el promedio de tres números, sería como sigue:

```
SUM #10, #11, #13
SUM #12, #13, #13
DIV #13, 3, #13
FIN
```

Donde SUM suma los números en las posiciones 10 y 11 y los coloca en la posición 13, luego suma el número en la dirección 12 al número en la dirección 13 y los coloca en la posición 13, luego se divide el número en la dirección 13 entre 3 y se almacena el resultado en la dirección 13, obteniéndose el

resultado.

Este código es llamado *lenguaje ensamblador*, y el programa que lo traduce a código máquina es el *ensamblador*. Debido a que las instrucciones varían por cada familia de procesadores, el lenguaje ensamblador es diferente y los programas escritos para un tipo de CPU no funcionarán en otro.

1.1.4. Lenguajes de programación

Para evitar este carnaval de lenguajes ensambladores y además simplificar la programación, se crearon los lenguajes de alto nivel, los cuales son iguales en todas las plataformas donde se ejecutan, de ahí su calificativo de "alto nivel". En contraste el código de máquina y los lenguajes ensambladores son lenguajes de bajo nivel por su dependencia de un hardware específico.

Por ejemplo, en Python el programa que divide tres valores es:

```
media = (10 + 11 + 12)/3
```

Mucho más sencillo que sus equivalente en lenguajes de bajo nivel.

1.1.5. Compiladores e interpretes

Para que los lenguajes de alto nivel tengan independencia de la plataforma, se ejecutan mediante programas llamados compiladores e interpretes, los cuáles se encargan de los detalles del hardware y presentan la misma interfaz a un programa escrito en el mismo lenguaje en otra plataforma.

Por ejemplo, se puede escribir un programa Python en una Machintosh y volverlo a ejecutar en una PC con Windows o en un servidor con FreeBSD y obtener el mismo resultado.

En palabras de [4]: [2]

> *Un **compilador** lee completamente un programa en un lenguaje de alto nivel y lo traduce en su integridad a un programa de código de máquina equivalente. El programa de código de máquina resultante se puede ejecutar cuantas veces se desee, sin necesidad de volver a traducir el programa original.*
>
> *Un **intérprete** actúa de un modo distinto: lee un programa escrito en un lenguaje de alto nivel instrucción a instrucción y, para cada una de ellas, efectúa una traducción a las instrucciones de código de máquina equivalentes y las ejecuta inmediatamente. No hay un proceso de traducción separado por completo del de ejecución. Cada vez que ejecutamos el programa con un intérprete, se repite el proceso de traducción y ejecución, ya que ambos son simultáneos.*

De lo escrito, se puede deducir que un programa compilado es en general más veloz que un programa interpretado, debido a que el intérprete debe ejecutarse siempre que se corre el programa. Por otro lado, los lenguajes interpretados han sido diseñados para ser más flexibles y dinámicos, automatizando la gestión de la memoria y liberando al programador de tareas que dificultan su trabajo.

[2] [4] pp. 14

1.1.6. Algoritmos

Dos programas que resuelvan el mismo problema de la misma forma, así estén escritos utilizando diferentes lenguajes de programación, están siguiendo ambos el mismo algoritmo. Un algoritmo es en esencia, una secuencia de pasos para conseguir un objetivo.

Por ejemplo, el algoritmo de la media de tres números es:

1. Solicitar el valor del primer número,
2. Solicitar el valor del segundo número,
3. Solicitar el valor del tercer número,
4. Sumar los tres números y dividir el resultado por 3,
5. Mostrar el resultado.

Esta secuencia de instrucciones nos permite conseguir el objetivo en cualquier lenguaje de programación, idependientemente de las instrucciones que se necesiten. Son procedimientos que se pueden ejecutar, siguiendo la línea de razonamiento, sin necesidad de tener una computadora, utilizando papel o una calculadora.

No todo conjunto de instrucciones es un algoritmo, para tener esa condición un procedimiento debe cumplir las siguientes condiciones: [3]

1. Ha de tener cero o más datos de entrada.
2. Debe proporcionar uno o más datos de salida como resultado.
3. Cada paso del algoritmo ha de estar definido con exactitud, sin la menor ambiguedad.
4. Ha de ser finito, es decir, debe finalizar tras la ejecución de un número finito de pasos, cada uno de los cuales ha de ser ejecutable en tiempo finito.
5. Debe ser efectivo, es decir, cada uno de sus pasos ha de poder ejecutarse en tiempo finito con unos recursos determinados (en nuestro caso, con los que proporciona una computadora).

Además de lo anterior, se busca que los algoritmos sean eficientes, es decir, que logren su objetivo lo más rápido posible.

1.2. ¿Qué es Python?

Python es un lenguaje de programación de alto nivel, fácil de aprender y de uso profesional con una sintaxis legible y ordenada. Posee además un entorno interactivo que permite hacer pruebas con velocidad y despejar dudas sobre el lenguaje. Su entorno de ejecución detecta muchos errores y proporciona información para resolverlos. En cuanto a sus características como lenguaje, Python puede programarse funcionalmente u orientado a objetos y posee ricas estructuras de datos que facilitan la labor del programador.

[3] [4] pp. 21

Python es muy utilizado en ámbitos científicos y de ingeniería debido a las características mencionadas y a la disponibilidad de librerías matemáticas y científicas de gran calidad.

Python puede ser ejecutado en múltiples plataformas, incluyendo las más comunes como Windows, Linux y MacOS. Se pueden llevar a cabo programas de escritorio, desarrollo web, utilitarios, juegos, etc.

Actualmente Python es utilizado por diversas empresas nacionales y extranjeras para escribir utilitarios, sistemas de información y desarrollos web.

Un ejemplo de programa Python es el siguiente:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


a = float(raw_input('Dame un numero:'))
b = float(raw_input('Dame otro numero:'))
c = float(raw_input('Y ahora, uno mas:'))
media = (a + b + c) / 3
print 'La media es', media
```

1.2.1. Utilización

Python es un lenguaje de programación interpretado, dinámico y multiparadigma (POO, estructurado). Se caracteriza por tener una baja curva de aprendizaje, ser multiuso y permitir una programación ordenada y elegante.

Su visión está expresada en el Zen de Python:

> Bello es mejor que feo.
> Explícito es mejor que implícito.
> Simple es mejor que complejo.

Para empezar a experimentar con Python se puede utilizar el intérprete interactivo desde la consola:

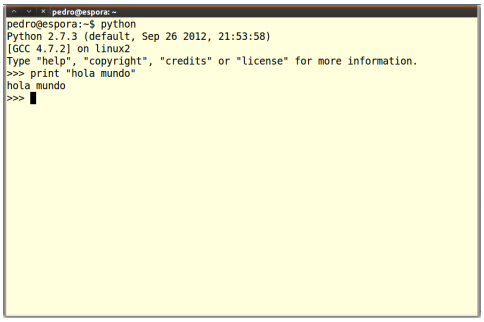

Gráfico 1.2: Consola python

Si se desea ejecutar varias lineas de código sin tener que tipear una tras otra en la consola interactiva, se guarda el código en un archivo de texto que termine en .py y se ejecuta utilizando el comando python.

1.3. Variables

Las variables son espacios de memoria a los que asignamos arbitrariamente valores que van a permanecer en la variable mientras no se cierre el programa. Por ejemplo, si deseamos calcular el volumen de un cubo u otro volumen similar, tenemos que multiplicar tres lados del sólido.

```
>>> 123*14.5*12
21402.0
>>> 123*14.5*11
19618.5
>>>
```

Solo para cambiar un lado, se tienen que introducir de nuevo todos los números en el problema. Si estos números se hubieran guardado en una variable, volver a llevar cabo el cálculo hubiera sido más sencillo.

```
>>> a = 123
>>> b = 14.5
>>> a*b*10
17835.0
>>>
```

Se han creado dos variables, a y b, en las cuales se almacenan dos números que no cambian y solo se debe escribir el nuevo número.

Al asignarle valor a una variable, el orden es importante, el valor se asigna de derecha a izquierda.

variable = expresión

Se debe recalcar que en un lenguaje de programación como Python, el símbolo = no significa "igual a" sino, "se asigna a".

Para poner nombres a las variables se siguen las siguientes reglas:

> El nombre de una variable es su identificador. Hay unas reglas precisas para construir identificadores. Si no se siguen, diremos que el identificador no es válido. Un identificador debe estar formado por letras minúsculas, mayúsculas, dígitos y/o el carácter de subrayado (_), con una restricción: que el primer carácter no sea un dígito.
>
> Hay una norma más: un identificador no puede coincidir con una palabra reservada o palabra clave. Una palabra reservada es una palabra que tiene un significado predefinido y es necesaria para expresar ciertas construcciones del lenguaje.

Las palabras reservadas de Python son: **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 y yield**.

1.4. Tipos básicos

Los tipos básicos son en esencia tres: números, cadenas y valores booleanos.

- Números: 5 (entero), 4.7 (flotante), 4 + 6j (complejo).
- Cadenas: "Clase de programación".
- Booleanos: True y False

Los comentarios se escriben utilizando el símbolo #:

```
# Este es un comentario
a = "Esta es una linea de programa"
```

Las variables se asignan de forma dinámica y dependen del tipo de dato que se les asigna, por ejemplo, una variable puede ser entera, luego asignarsele un valor de coma flotante y por último una cadena y es válido:

```
>>> a = 5
>>> a
5
>>> a = 6.4
>>> a
6.4
```

```
>>> a = "Lima"
>>> a
'Lima'
>>>
```

1.4.1. Cadenas

Existen tres tipo de cadenas: Las cadenas normales, las cadenas unicode y las cadenas "crudas" (raw en inglés).

```
>>> print "a"
a
>>> c = r"\n"
>>> c
'\\n'
>>> print c
\n
>>>
>>> d = u"g"
>>> d
u'g'
>>> print d
g
>>>
```

En Python se pueden concatenar y multiplicar cadenas.

```
>>> 4 * "cad"
'cadcadcadcad'
>>> "cadena1" + " " + "cadena 2"
'cadena1 cadena 2'
>>>
```

1.4.2. Bool

Python tiene un tipo de datos especial que permite expresar sólo dos valores: cierto y falso en la forma True o False. Son los valores booleanos, llamados así en honor del matemático inglés George Bool, que desarrolló un sistema algebraico basado en estos dos valores y tres operaciones: la conjunción, la disyunción y la negación. Python ofrece soporte para estas operaciones con los operadores lógicos.

Se estudiará con más profundidad este tipo de datos al aprender sobre los operadores lógicos y de comparación en la siguiente clase.

1.5. Operadores aritméticos

Operador	Descripción	Ejemplo
+	Suma	r = 3 + 2 # r es 5
-	Resta	r = 4 - 7 # r es -3
-	Negación	r = -7 # r es -7
*	Multiplicación	r = 2 * 6 # r es 12
**	Exponente	r = 2 ** 6 # r es 64
/	División	r = 3.5 / 2 # r es 1.75
//	División entera	r = 3.5 // 2 # r es 1.0
%	Módulo	r = 7 % 2 # r es 1

Si en una operación utilizamos un número flotante y un entero el resultado será flotante.

1.6. Ejercicios

Traducir las siguientes expresiones matemáticas a Python y evaluarlas. Utilizar el menor número de paréntesis posible.

1. $2 + (3 \cdot (6/2))$
2. $\frac{4+6}{2+3}$
3. $(4/2)^5$
4. $(4/2)^{5+1}$
5. $(-3)^2$
6. $-(3^2)$

Evaluar los polinomios:

1. $x^4 + x^3 + 2x^2 - x$

2. $x^4 + x^3 + \frac{1}{2}x - x$

Evaluar estas expresiones y sentencias en el orden indicado:

1. a = 'b'
2. a + 'b'
3. a + 'a'
4. a * 2 + 'b' * 3
5. 2 * (a + 'b')
6. 'a' * 3 + '/*' * 5 + 2 * 'abc' + '+'
7. 2 * '12' + '.' + '3' * 3 + 'e-' + 4 * '76'

Capítulo 2

Estructuras de control

2.1. Operadores

En Python se utilizan los operadores aritméticos para llevar cabo operaciones matemáticas. Los operadores existentes son: +, -, *, /, %, **.

```
>>> 4 + 5
9
>>> 12 - 9
3
>>> 5 * 6
30
>>> 24 / 4
6
>>> 9 % 4
1
>>> 3 ** 2
9
>>>
```

2.2. Operadores lógicos y de comparación

Para programar existen tres operadores lógicos: la "y lógica" o conjunción (and), la "o lógica" o disyunción (or) y el "no lógico" o negación (not).

Cada operador tiene su propia tabla de verdad.

El operador and da como resultado True si y sólo si son verdaderos sus dos operandos. La tabla del operador and es:

Izquierdo	Derecho	Resultado
True	True	True
True	False	False
False	True	False
False	False	False

El operador or proporciona True si cualquiera de sus operandos es True, y False sólo cuando ambos operandos son Falses. La tabla del operador or es:

Izquierdo	Derecho	Resultado
True	True	True
True	False	True
False	True	True
False	False	False

El operador not es un operador unario y devuelve True si el operando es False y viceversa. La tabla del operador not es:

Izquierdo	Resultado
True	False
False	True

Al combinar valores verdaderos y falsos se pueden formar expresiones lógicas que den como resultado True o False.

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> False or True
True
```

```
>>> not False
True
>>> not True
False
>>>
```

Al utilizarse en conjunto los operadores lógicos tiene diferente precedencia, por ejemplo.

```
>>> True or False and True
True
>>> True and False or True
True
>>>
```

Existen varios operadores que devuelven valores booleanos (True o False) al operar, entre ellos están los operadores de comparación los cuales son: >, <, >=, <=, ==, !=.

```
>>> 4 > 5
False
>>> 6 < 7
True
>>> 5 >= 5
True
>>> 4 <= 3
False
>>> 4 <= 4
True
>>> 8 == 8
True
>>> 5 != 6
True
```

2.3. Sentencias condicionales

La sentencia condicional 'if' (si) sirve para bifurcar el flujo de un programa. ¿Qué significa esto?

Un programa sigue un flujo secuencial, una instrucción se ejecuta después de la anterior, pero al llegar a un if, se puede decidir si ejecutar un bloque de instrucciones u otro, dependiendo de las condiciones definidas en el programa. Al llegar a este punto, ejecuta esta(s) accion(es) sólo si la condición es cierta.

Este tipo de sentencia se denomina condicional y en Python se escribe de la siguiente forma:

```
if condition:
    action
    action
    ...
```

2.3.1. Sentencia if

Como ejemplo consideremos la división de dos números, se puede calcular siempre y cuando el divisor sea diferente de cero. Si hacemos un programa que divide dos números, debemos evitar que el divisor sea cero para lo cual se utiliza una sentencia if.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

a = float(raw_input('Valor del divisor: '))
b = float(raw_input('Valor del dividendo: '))

if a != 0:
    x = b/a
    print 'Solucion: ', x
```

Como se puede apreciar, la sentencia if es seguida por una condición que indica "a debe ser diferente de cero". Si la condición es verdadera, se ejecuta el consiguiente bloque de código, si es falsa no se ejecuta y termina el programa.

En el ejemplo se pueden ver dos importantes reglas sintácticas de Python:

1. La condición de la sentencia if es seguida por dos puntos (:).
2. Las líneas de código que se ejecutan luego del if tienen que seguir todas una misma identación (o sangrado según el texto).

El programa no tiene errores debido al if, pero si el divisor es cero no devuelve nada, un comportamiento confuso para el usuario. Para evitarlo podemos incluir un segundo if y el programa quedaría como sigue:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


a = float(raw_input('Valor del divisor: '))
b = float(raw_input('Valor del dividendo: '))


if a != 0:
    x = b/a
    print 'Solucion: ', x


if a == 0:
    print "No se puede realizar una division entre cero."
```

En el segundo ejemplo se incluye una segunda sentencia if la que comprueba si a es cero, de ser así, escribe en la pantalla que no se puede realizar la división.

Este segundo if, si bien agrega una funcionalidad útil al programa, es ineficiente debido a que se va a ejecutar el condicional sea cual sea el valor de a. Para solucionar este problema se necesita que si la condición no se cumple, se ejecute otro bloque de código, para este se utiliza la sentencia else.

2.3.2. En caso contrario (else)

La palabra "else" significa, en inglés, "sino" o en "caso contrario". Al igual que en el if, los bloques de código en una sentencia else requieren ser identados.

El código de la división entre cero utilizando un else es:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


a = float(raw_input('Valor del divisor: '))
b = float(raw_input('Valor del dividendo: '))


if a != 0:
    x = b/a
```

```
    print 'Solucion: ', x
else:
    print "No se puede realizar una division entre cero."
```

Si en vez de discernir si un numéro es o no cero, se buscará utilizar más condiciones posibles de una expresión lógica entonces la sentencia else se reemplaza por la sentencia elif.

2.3.3. Condicionales múltiples (elif)

En diversos problemas, las alternativas no serán solo dos sino se deberá analizar diferentes casos. En este escenario se utiliza la sentencia elif (else if en otros lenguajes) para discernir que condición es la que se cumple.

Por ejemplo, si se desea determinar cuál es el mayor de tres números, se puede utilizar el siguiente algoritmo (en una gran cantidad de problemas, como este, no existe un solo algoritmo como solución, sino que pueden pensarse e implementarse diversos algoritmos que solucionen el problema, la respuesta de un programador no es siempre igual a la de otro):

1. Comparar el primer valor con el segundo y el tercero, si es mayor que ambos entonces es el mayor número.
2. Si lo anterior no se cumple entonces se compara el segundo valor con los otros dos, si es mayor que ambos es el número más grande.
3. Si todo lo anterior no es cierto, se compara el tercer número.

Este algoritmo se puede implementar utilizando solo sentencias if, pero es poco eficiente ya que si la primera condición resulta ser cierta, igual evaluaría los otros casos cuando ya no es necesario hacerlo. Para hacerlo más eficiente utilizamos la sentencia elif.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


a = int(raw_input('Dame el primer numero: '))
b = int(raw_input('Dame el segundo numero: '))
c = int(raw_input('Dame el tercer numero: '))


if a >= b and a >= c:
    maximo = a
elif b >= a and b >= c:
    maximo = b
elif c >= a and c >= b:
```

```
    maximo = c
print 'El maximo es', maximo
```

Los algoritmos que se pueden expresar con **if**, **if-else** y **elif** se pueden expresar solo con **if** pero pierden legibilidad y eficiencia.

2.4. Sentencias iterativas

Si se desea no solo elegir entre bloques de código a ejecutar, sino ejecutarlos varias veces, se utilizan sentencias iterativas, las que como su nombre indica, iteran un bloque de código n veces.

Python permite indicar que deseamos que se repita un trozo de programa de dos formas distintas: mediante la sentencia while y mediante la sentencia for.

2.4.1. La sentencia while

En inglés, "while" significa "mientras". La sentencia while se usa así:

```
while condicion:
    action
    action
    .
    .
    .
    action
```

Y su significado es:

Mientras se cumpla esta condición, repetir las siguientes acciones.

Las sentencias que implican repetición se denominan *bucles*. En un bucle, un conjunto de instrucciones son repetidas una cantidad de veces. En el caso del **While**, el bloque de código se repite hasta que determinada condición sea falsa.

Por ejemplo, en el siguiente programa se imprimen números de 0 al 4 utilizando un bucle **while**:

```
i=0
while i < 5:
    print i
    i += 1
```

```
print 'Hecho'
```

En este ejemplo se puede ver que la sentencia **while** termina en dos puntos luego de la condición, la identación indica que líneas pertenecen al bucle **while** y la última línea no pertenece al bucle y se ejecuta cuando el **while** ha terminado.

Cálculo de la sumatoria

Como ejemplo del uso de un bucle, se mostrará como se puede calcular una sumatoria. La sumatoria matemática consiste en sumar una cantidad de veces determinada expresión. Por ejemplo si se desean sumar los mil primeros números naturales:

$$\sum_{i=0}^{1000} i$$

O, lo que es lo mismo, el resultado de $1 + 2 + 3 + \cdots + 999 + 1000$.

En egeneral la primera idea que le suele venir a quienes aprenden a programar es sumar (1 + 2 + 3 + ... + 999 + 1000) e imprimir esa suma en la pantalla, un procedimiento tedioso e inútil. Se podría sumar también los números a una variable, lo que sería otra forma tediosa y mecánica de llevar a cabo la operación.

Para hacer de forma concisa la suma se utilizará un bucle while:

```
sumatorio = 0
while condicion :
    sumatorio += numero
print sumatorio
```

2.4.2. El bucle for-in

Hay otro tipo de bucle en Python: el bucle **for-in**, que se puede leer como "para todo elemento de una serie, hacer. . . ". Un bucle **for-in** presenta el siguiente aspecto:

```
for variable in serie de valores:
    action
    action
    .
    .
    .
    action
```

Por ejemplo:

```
for nombre in ['Pepe', 'Ana', 'Juan']:
    print 'Hola,', nombre
```

Los nombres entre corchetes y separados por comilla son una lista de nombres. Las listas se estudiarán más adelante en el curso, ahora solo es necesario saber que el bucle **for-in** ha iterado sobre sobre los nombres, la variable *nombre* ha tomado el valor de cada uno de forma ordenada, de izquierda a derecha.

Para poner un ejemplo, se puede pensar en el problema de elevar una serie de números a una potencia:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


numero = int(raw_input('Dame un numero: '))


for potencia in [2, 3, 4, 5]:
    print numero, 'elevado a ',potencia, ' es ', (numero ** potencia)
```

En caso que se desee iterar en grandes números (de 1 a 100 por ejemplo), se utiliza la función range(), tal como se muestra en el siguiente ejemplo:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
```

En el ejemplo se ve como range() crea una lista cuyos primer y último valor son el primer parámetro y el anterior al segundo parámetro. En caso que no se coloque un tercer parámetro el intervalo entre los elementos de la lista será 1, si se coloca un tercer parámetro se utilizará como intervalo, como se puede ver en el sgundo ejemplo con la función range().

Para mostrar un ejemplo más complejo, se calcularán las raices de un número:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
numero = float(raw_input('Dame un numero: '))

for n in range(2, 101):
    print 'la raiz', n, '-esima de ', numero,' es ', (numero**(1.0/n))
```

Para calcular una sumatoria, tal como se hizo con el while, se puede utilizar el siguiente código:

```
sum = 0
for i in range(1, 1001):
    sum += i
print sum
```

Para calcular si un número es o no primo:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

num = int(raw_input('Dame un numero: '))

creo_que_es_primo = True

for divisor in range(2, num):
    if num % divisor == 0:
        creo_que_es_primo = False

if creo_que_es_primo:
    print 'El numero', num, 'es primo'
else:
    print 'El numero', num, 'no es primo'
```

2.4.3. Rotura y salto en bucles: break y continue

En diversos programas es necesario terminar la ejecución de un bucle antes de que termine, por ejemplo, en el anterior programa donde se calcula si un número es o no primo, a pesar de que se haya encontrado que el número no es primo, el programa se sigue ejecutando. Para evitar este malgasto

de recursos se utiliza la sentencia **break** (romper en inglés).

En el siguiente programa se calculará si un número es o no primo, pero se utilizará break para terminar el bucle en cuanto se encuentre que el número es primo.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


num = int(raw_input('Dame un numero: '))


creo_que_es_primo = True


for divisor in range(2, num):
    if num % divisor == 0:
        creo_que_es_primo = False
        break


if creo_que_es_primo:
    print 'El numero', num, 'es primo'
else:
    print 'El numero', num, 'no es primo'
```

Otra sentencia importante para controlar la ejecución de un bloque es **continue** (continuar en inglés). Esta sentencia a diferencia de **break** no termina la ejecución de un bucle, solo pasa a la siguiente iteración sin ejecutar el resto del código dentro del bucle.

Por ejemplo, si se tienen una serie de números y se desean imprimir solo los números pares se puede dar la siguiente solución (hay otras soluciones posibles, pero se utiliza esta para propositos de ejemplo):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


for num in [2,5,3,89,24,16,17,31]:
    if num % 2 != 0:
        continue
    print num
```

En este ejemplo se puede ver como **continue** sirve para no ejecutar la sentencia siguiente en el bucle. Si en el if se encuentra que num no es par, se pasa a la siguiente iteración, en caso contrario el if no hace nada y se imprime el número en la pantalla.

2.5. Ejercicios

Verifique los resultados de las siguientes expresiones:

1. True == True != False
2. (1 < 3) and (4 > 5)
3. (4 > 5) or (10 != 10)
4. (True or (2 == 1 + 4)) == False

1. Escriba un programa que lea un número flotante por teclado y muestre por pantalla el mensaje "El número es negativo." sólo si el número es menor que cero.
2. Escriba un programa que lea un número flotante por teclado y muestre por pantalla el mensaje "El número es positivo." sólo si el número es mayor que cero.
3. Escriba un programa que lea la edad de dos personas y diga quién es más joven, la primera o la segunda. Ten en cuenta que ambas pueden tener la misma edad. En tal caso, hazlo saber con un mensaje adecuado.
4. Escriba un programa que, dado un número entero, muestre por pantalla el mensaje "El número es par." cuando el número sea par y el mensaje "El número es impar." cuando sea impar.
5. Un capital de C euros a un interés del x por cien anual durante n años se convierte en $C * (1 + \frac{x}{100})^n$ euros. Diseñe un programa Python que solicite la cantidad C, el interés x y el número de años n y calcule el capital final sólo si x es una cantidad positiva.
6. Escriba un programa que lea dos números de la pantalla y devuelva el mayor de ambos, si ambos son iguales debe escribir "Ambos números son iguales".
7. Escribe un programa que calcule el máximo de 5 números enteros.
8. Escribe un programa que, dado un número real que debe representar la calificación numérica de un examen, proporcione la calificación cualitativa correspondiente al número dado. La calificación cualitativa será una de las siguientes: "Jalado" (nota menor que 10.5), "Aprobado" (nota mayor o igual que 10.5, pero menor que 15), "Notable" (nota mayor o igual que 15, pero menor que 17), "Sobresaliente" (nota mayor o igual que 17, pero menor que 20), "Matrícula de Honor" (nota 20).
9. Implementa un programa que muestre todos los múltiplos de 6 entre 6 y 150, ambos inclusive.
10. Implementa un programa que muestre todos los múltiplos de n entre n y m * n, ambos inclusive, donde n y m son números introducidos por el usuario.
11. Implementa un programa que muestre todos los números potencia de 2 entre 20 y 230, ambos inclusive.
12. Escriba un programa que calcule $\sum_{i=n}^{m} i$ donde n y m son números enteros que deberá introducir el usuario por teclado.

13. Modifica el programa anterior para que si n ¿m, el programa no efectúe ningún cálculo y muestre por pantalla un mensaje que diga que n debe ser menor o igual que m.

14. El factorial de n se denota con n!, escriba un programa que pida el valor de n y muestre por pantalla el resultado de calcular n!.

15. Escriba un programa que muestre la tabla de multiplicar de un número introducido por teclado por el usuario.

16. Escribe un programa que muestre, en líneas independientes, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

17. Escribe un programa que muestre, en líneas independientes y en orden inverso, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

18. Escribe un programa que muestre los números pares positivos entre 2 y un número cualquiera que introduzca el usuario por teclado.

19. Escribe un programa que pida el valor de dos enteros n y m y calcule el sumatorio de todos los números pares comprendidos entre ellos (incluyéndolos en el caso de que sean pares).

20. Escribe un programa que calcule el máximo común divisor (mcd) de dos enteros positivos. El mcd es el número más grande que divide exactamente a ambos números.

21. Escribe un programa que vaya leyendo números y mostrándolos por pantalla hasta que el usuario introduzca un número negativo. En ese momento, el programa mostrará un mensaje de despedida y finalizará su ejecución.

22. Escribe un programa que vaya leyendo números hasta que el usuario introduzca un número negativo. En ese momento, el programa mostrará por pantalla el número mayor de cuantos ha visto.

Capítulo 3

Tipos de datos estructurados

Los tipos de datos utilizados hasta el momento han sido números (enteros y flotantes), listas y cadenas (que son una forma especial de lista). Los números son tipos de datos *escalares* mientras que la cadenas y listas son tipos de datos *secuenciales*. Un escalar es un dato atómico, único mientras que un dato secuencial se compone de una sucesión de elementos, tal como una cadena es una sucesión de caracteres. Los datos secuenciales son tipos de datos estructurados.

3.1. Cadenas

Una cadena es una sucesión de caracteres encerradas en comillas simples o dobles las que cuales puden ser manipuladas por los operadores y funciones de Python. Una cadena es inmutable, es decir, una vez creada en memoria ya no puede ser modificada.

3.1.1. Escapes

En un programa las cadenas no solo incluyen los caracteres que se ven en pantalla sino también caracteres especiales que no tienen una representación visible. Por ejemplo los *saltos de línea* se muestran en pantalla como eso y no como un carácter visible. Los saltos de línea se incluyen en la cadena y producen un efecto al ser impresa, por ejemplo:

```
>>> a='cadena\n de ejemplo'
>>> a
'cadena\n de ejemplo'
>>> print a
cadena
 de ejemplo
>>>
```

Al mostrar la cadena, se produce un salto de línea después de *cadena*. Este salto de línea se codifica en dos caracteres: la barra invertida \ y la letra n. La barra invertida se llama carácter de escape e indica que el siguiente carácter es especial. Los saltos de línea tienen sus propios códigos en los

estándares de codificación de texto como ASCII y Unicode. Por ejemplo en ASCII el código de `\n` es 10:

```
>>> ord('\n')
10
>>>
```

Algunos caracteres de control son `\n` que genera un salto de línea, `\t` que tabula el texto de forma horizontal (8 caracteres), `\v` que los tabula vertical y `\b` que retrocede el texto un espacio.

```
>>> print "Hola\ncadena"
Hola
cadena
>>> print "Tabular el\ttexto"
Tabular el        texto
>>> print "Tabula\vvertical"
Tabula
      vertical
>>> print "Retrocede \bun espacio"
Retrocedeun espacio
>>>
```

Si se quiere imprimir una comilla en un texto con el mismo tipo de comillas, se debe utilizar (escapar) con una barra invertida.

```
>>> print "Comillas \" en una cadena"
Comillas " en una cadena
>>> print 'Comillas \' en una cadena'
Comillas ' en una cadena
>>>
```

3.1.2. Longitud e indexación

Para medir la longitud de una cadena y/o una lista se utiliza la función *len*.

```
>>> len('Hola Mundo')
10
>>> len('Hola\nMundo')
10
```

3.1.3. Subcadenas

Para acceder a cada carácter de una cadena se utilizan los corchetes (como en una lista). El primer elemento tiene índice cero. Si se desea obtener una subcadena se colocan el índice del primer y último carácter con dos puntos de por medio (lo mismo sirve para las listas).

```
>>> a = "Clase de Python"
>>> a[14]
'n'
>>> a[4:11]
'e de Py'
>>>
```

3.1.4. Recorrido de cadenas

Debido a que son listas, las cadenas se pueden recorrer con un bucle **for-in**.

```
>>> for c in "Cadena":
...     print c
...
C
a
d
e
n
a
>>>
```

La palabra reservada **in** sirve también para identificar si un elemento está dentro de una lista o cadena, por ejemplo:

```
>>> 'a' in 'Hola'
True
>>> 'a' in 'Python'
False
>>>
```

Una forma alternativa de recorrer una lista es mediante su índice. Se itera sobre un rango del tamaño de la lista y se referencia cada elemento.

```
>>> cad = "Ejemplo"
>>> for i in range(len(cad)):
...     print i, cad[i]
...
0 E
1 j
2 e
3 m
4 p
5 l
6 o
>>>
```

La variable i toma los valores de range(len(a)), en este caso los valores comprendidos entre 0 y 6, ambos inclusive. Con a[i] se accede a cada uno de ellos.

3.1.5. Comparación de cadenas

Al igual que en las listas, para comparar cadenas se emplea los operadores '==', '!=', '<', '>', '<=', '>='. Los operadores '<', '>', '<=', '>=' se utilizan comparando de izquierda a derecha cada carácter entre si.

```
>>> 'Python' == 'Python'
True
>>> a = 'Cadena'
>>> b = 'Cadena'
>>> a == b
True
```

```
>>> x = "Linux"
>>> y = "Windows"
>>> x == y
False
>>>
>>> 'Martes' != 'Lunes'
True
>>> 'A' > 'a'
False
>>> 'Abcdefg' < 'xyz'
True
>>>
```

3.2. Listas

Tal como se ha mencionado, las cadenas son tipos especiales de listas, el tipo secuencial de datos más utilizado en Python. En una lista se pueden almacenar diferentes tipos de datos como números, cadenas e incluso otras listas. Para representar las listas se utilizan elementos separados por comas encerrados entre corchetes.

```
>>> [1, 2, 3, 4,5, 6]
[1, 2, 3, 4, 5, 6]
>>> ['hola', 'mundo']
['hola', 'mundo']
>>> a = [1, 6, 3, -5, 10.6]
>>> a
[1, 6, 3, -5, 10.6]
>>> b = ['lista', ['otra', 'lista']]
>>> c = [2+4, 5-2, 3*'cad']
>>> c
[6, 3, 'cadcadcad']
>>>
```

Muchas de las operaciones y funciones que operan sobre las cadenas también funcionan en las listas. La función *len* devuelve la longitud de una lista, el operador + las concatena, el operador * las multiplica un número de veces, se puede hacer referencia a un elemento de la lista por su índice, el

operador de corte obtiene un un fragmento de la lista:

```
>>> a = [1, 6, 3, -5, 10.6]
>>> len(a)
5
>>> b = ['lista', ['otra', 'lista']]
>>> a + b
[1, 6, 3, -5, 10.6, 'lista', ['otra', 'lista']]
>>> 4*a
[1, 6, 3, -5, 10.6, 1, 6, 3, -5, 10.6, 1, 6, 3, -5, 10.6, 1, 6, 3, -5, 10.6]
>>> a[2]
3
>>> a[1:3]
[6, 3]
>>> a[2:]
[3, -5, 10.6, 3]
>>> a[:4]
[1, 6, 3, -5]
>>> [1]*10
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

El bucle **for-in** recorre los elementos de una lista:

```
>>> for i in [1, 5, -3]:
...     print i
...
1
5
-3
>>>
```

3.2.1. Comparación de listas

Los operadores de comparación de las cadenas también trabajan con las listas. Tanto los operadores de igualdad (==) como el de desigualdad (!=) siguen las siguientes reglas:

- si las listas son de talla diferente, resolviendo que las listas son diferentes;
- y si miden lo mismo, comparando elemento a elemento de izquierda a derecha y resolviendo que las dos listas son iguales si todos sus elementos son iguales, y diferentes si hay algún elemento distinto.

```
>>> [1, 2, 3] == [1, 2]
False
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] == [1, 2, 4]
False
>>>
```

Los operadores '<', '>', '<=', '>=' se utilizan comparando de izquierda a derecha cada elemento entre si, al igual que en las cadenas. Por ejempo, al evaluar la expresión [1, 2, 3] < [1, 3, 2] se empieza por comparar los primeros elementos de ambas listas. Como no es cierto que $1 < 1$, pasamos a comparar los respectivos segundos elementos. Como $2 < 3$, el resultado es True, sin necesidad de efectuar ninguna comparación adicional.

```
>>> [1, 2, 3] < [1, 3, 2]
True
>>> [10, 20, 30] > [1, 2, 3]
True
>>> [1, 2, 3] < [1, 2]
False
>>>
```

3.2.2. Modificar, agregar y eliminar elementos a listas

Para modificar el contenido de una lista se pueden modificar sus elementos, agregar elementos o eliminarlos.

```
>>> a = [4,7,1]
>>> a[2] = 5
```

```
>>> a
[4, 7, 5]
>>>
```

Al agregar elementos a una lista la hacemos crecer, una forma es sumándole otra lista pero si se desea agregar un elemento que no está en una lista se necesita utilizar la función *append*. Además de la diferencia entre agregar una lista y un elemento fuera de una lista, otra diferencia entre + y append es que la segunda no crea una nueva lista sino *modifica* la lista original. Debido a esta diferencia en el trabajo en memoria, append es más eficiente.

Una lista puede ser representada por una variable que apunte a esta lista, debido a que es un puntero, si se asigna esta variable a otra variable, ambas apuntaran a la misma lista, de tal manera que cualquier modificación repercutirá en ambas variables. Si se quiere comprobar si dos variables representan la misma posición en memoria, se utiliza el operador **is**.

```
>>> a = [9,3,6]
>>> a
[9, 3, 6]
>>> a.append(12)
>>> a
[9, 3, 6, 12]
>>> >>> b = a
>>> b
[9, 3, 6, 12]
>>> a.append(5)
>>> b
[9, 3, 6, 12, 5]
>>> a is b
True
>>>
```

Para eliminar elementos de una lista se utiliza el método **pop** con el índice del elemento a eliminar. También se puede utilizar el operador de corte para eliminar un fragmento de la lista.

```
>>> a = range(1,10)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.pop(5)
```

```
6
>>> a
[1, 2, 3, 4, 5, 7, 8, 9]
>>> b = a
>>> a.pop(2)
3
>>> b
[1, 2, 4, 5, 7, 8, 9]
>>> a
[1, 2, 4, 5, 7, 8, 9]
>>>
```

El borrado de elementos de una lista es peligroso cuando se mezcla con el recorrido de las mismas. Por ejemplo un programa que elimina los elementos negativos de una lista.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

a = [1, 2, -1, -4, 5, -2]

for i in range(0,len(a)):
    if a[i] < 0:
        a.pop(i)

print a
```

Este programa no funciona bien y da un error:

```
Traceback (most recent call last):
  File "solo_positivos.py", line 9, in <module>
    if a[i] < 0:
IndexError: list index out of range
```

El mensaje de error nos dice que tratamos de acceder a un elemento con índice fuera del rango de índices válidos. ¿Cómo es posible, si la lista tiene 6 elementos y el índice i toma valores desde 0 hasta 5? Al eliminar el tercer elemento (que es negativo), la lista ha pasado a tener 5 elementos, es decir, el índice de su Ítimo elemento es 4. Pero el bucle "decidió" el rango de índices a recorrer antes de borrarse ese elemento, es decir, cuando la lista tenía el valor 5 como índice del útimo elemento.

Cuando tratamos de acceder a a[5], Python detecta que estamos fuera del rango válido. Es necesario que el bucle "actualice" el valor del último índice válido con cada iteración:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

a = [1, 2, -1, -4, 5, -2]

i=0
while i < len(a):
    if a[i] < 0:
        a.pop(i)
    i += 1

print a
```

Al ejecutar el programa se ve que el -4 no ha sido eliminado. Inicialmente la lista era [1, 2, -1, -4, 5, -2], al eliminar el elemento a[2] de la lista original, i valía 2 y la lista se convierte en [1, 2, -4, 5, -2]. Después del borrado, se incrementó i y eso hizo que la siguiente iteración considerara el posible borrado de a[3], pero en ese instante -4 estaba en a[2], así que "se lo salta". La solución es sencilla: sólo se ha de incrementar i en las iteraciones que no producen borrado alguno:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

a = [1, 2, -1, -4, 5, -2]

i=0
while i < len(a):
    if a[i] < 0:
        a.pop(i)
    else:
        i += 1

print a
```

3.2.3. Pertenencia a una lista

Al igual que en las cadenas, para saber si un elemento pertenece a una lista se utiliza el operador **in**.

```
>>> a = [4, 9, 1, -34, 7]
>>> 4 in a
True
>>> 100 in a
False
>>>
```

3.2.4. Listas por comprensión (List Comprehensions)

En muchas ocasiones para generar una lista se puede utilizar una característica de Python llamada "listas por comprensión", la cual permite crear listas en base a un programa "in line" dentro de la lista.

Por ejemplo, el siguiente programa escoge solo los números pares de una lista generada mediante range() y los agrega a una lista.[6]

```
numbers = range(10)
size = len(numbers)
evens = []
i = 0
while i < size:
    if i % 2 == 0:
        evens.append(i)
    i += 1
evens
[0, 2, 4, 6, 8]
```

En vez de llevar a cabo todo este procedimiento utilizando una variable para iterar (i) y hacer trabajar al interprete en cada iteración, se puede utilizar el siguiente código:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
>>>
```

3.2.5. Copiado de una lista

Si se asigna una lista a otra variable, lo único que se consigue es crear una nueva referencia a esa lista. Para copiar una lista se utilizan [:] o list().

```
>>> a = [3,4,6,8,3,6]
>>> b = a
>>> b
[3, 4, 6, 8, 3, 6]
>>> a is b
True
>>> b = a[:]
>>> a is b
False
>>> b = list(a)
>>> a is b
False
>>>
```

3.3. Tuplas

Todo lo explicado sobre listas es válido para las tuplas, con la excepción de que las tuplas no pueden modificarse. Para declarar una tupla se utilizan paréntesis, en caso se desee una tupla de un solo elemento se agrega una coma para diferenciarlo de un elemento entre paréntesis.

```
>>> t = (1,2,3,4)
>>> t[2]
3
>>>  type(t)
<type 'tuple'>
>>> t1 = (1,)
>>> type(t1)
<type 'tuple'>
>>> type(t)
```

Las tuplas no pueden modificarse una vez creadas pero son más ligeras que las listas. Para convertir de una lista a una tupla y viceversa se utilizan los comandos **list** y **tuple**.

3.4. Diccionarios

Los diccionarios son estructuras de datos donde una "clave" hace referencia a un valor (como en una hash table). Para declararlos se utilizan llaves y para hacer referencia a los valores se colocan las claves entre corchetes. Como clave se puede utilizar cualquier valor inmutable: cadenas, números, tuplas, booleanos. Los valores en un diccionario no están ordenados.

Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

```
>>> d = {2: 123}
>>> d[2]
123
>>> d[3] = 456
>>> d
{2: 123, 3: 456}
>>> d["hola"] = 9887
>>> d
{2: 123, 3: 456, 'hola': 9887}
>>> d[2] = [100, 200, 300]
>>> d
{2: [100, 200, 300], 3: 456, 'hola': 9887}
>>>
>>> 2 in d
True
>>> 5 in d
False
>>>
```

Como se puede apreciar, si se quiere agregar un nuevo par clave-valor, se asigna el valor a la clave en el diccionario y si se quiere comprobar si existe una clave, se utiliza el operador **in**.

3.5. Ejercicios

1. Represente dos barras invertidas seguidas en una cadena.
2. Escriba un programa que lea una cadena y muestre el número de espacios en blanco que contiene.

3. Escriba un programa que lea una cadena y muestre el número de letras mayúsculas que contiene.

4. Escriba una programa que lea una cadena y muestra en pantalla el mensaje "Contiene dígito" si contiene algún dígito y "No contiene dígito" en caso contrario.

5. Escriba un contador de palabras. Al programa se le ingresa una oración y este encuentra el número de palabras (tip: el número de palabras es igual al número de espacios en blanco más uno).

6. Escriba un programa que devuelva el tamaño de la primera palabra de una oración.

7. Escriba un programa que reciba una cadena y un entero k y devuelva cuantas palabras tienen longitud k.

8. Escriba un programa que muestre la cantidad de dígitos que aparecen en una cadena introducida por teclado.

9. Escriba un programa que muestre la cantidad de números que aparecen en una cadena introducida por teclado (un número puede estar compuesto de uno o más dígitos).

10. Escriba un programa que indique si una cadena introducida por el usuario está bien formada como identificador de variable. Si lo está, mostrará el texto "Identificador válido" y si no, "Identificador inválido".

11. Un texto está bien parentizado si por cada paréntesis abierto hay otro más adelante que lo cierra. Escribaa un programa que lea una cadena y responda si está bien o mal parentizada.

12. Implementa un programa que lea de teclado una cadena que representa un número binario. Si algún carácter de la cadena es distinto de '0' o '1', el programa advertirá al usuario de que la cadena introducida no representa un número binario y pedirá de nuevo la lectura de la cadena.

13. Escriba un programa que lea tres cadenas y muestre el prefijo común más largo de todas ellas.

14. Escriba un programa que almacene en una variable a la lista obtenida con range(1,4) y a continuación, la modifique para que cada componente sea igual al cuadrado del componente original. El programa mostrará la lista resultante por pantalla.

15. Escriba un programa que, dada una lista a cualquiera, sustituya cualquier elemento negativo por cero.

16. Escriba un programa que elimine de una lista todos los elementos de valor par y muestre por pantalla el resultado.

17. Escriba un programa que elimine de una lista todos los elementos de índice par y muestre por pantalla el resultado.

18. Se dispone de una cadena que contiene una frase cuyas palabras están separadas por un número arbitrario de espacios en blanco. ¿Se puede"estandarizar" la separación de palabras en una sola línea Python? Por estandarizar quiere decir que la cadena no empiece ni acabe con espacios en blanco y que cada palabra se separe de la siguiente por un unico espacio en blanco.

19. Cree un diccionario cuyas claves sean cuatro regiones del Perú y sus valores sean las capitales de cada región. Itere ese diccionario e imprima la clave y valores de la forma: "La capital de X es Y". Para iterar sobre el diccionario utilice iteritems() o keys().

Capítulo 4

Funciones y Módulos

En los capítulos previos se han utilizado funciones en diversas oprtunidades. Instrucciones como len(), range(), round(), y otras son funciones predefinidas mientras que otras deben importarse de módulos antes de ser utilizadas como *sin* y *cos* del módulo *math*.

Una función debe llevar a cabo una sola tarea de forma eficiente y en caso necesario recibir los parámetros que necesite para cumplir su objetivo y devolver el resultado. Es mala práctica solicitar datos dentro de una función o confundir la devolución de los datos con la impresión del resultado en algún medio externo como la pantalla.

Las funciones no solo vienen definidas por librerías, el programador puede crear sus propias funciones que le permitan resolver un problema una vez y utilizar esa solución n cantidad de veces.

4.1. Uso de las funciones

Al llamar a una función (también se le dice activar o invocar) se le pasan cero o más parámetros separados por comas y encerrados entre paréntesis. Una función puede devolver un valor o no devolver nada. En caso de que no se especifique un valor de retorno, la función devuelve **None**, equivalente a **Null** de otros lenguajes.

```
>>> round(14.56678, 2)
14.57
>>> round(14.56678)
15.0
>>> from math import cos
>>> cos(90)
-0.4480736161291701
>>>
```

Si una función devuelve un resultado, este es de un tipo determinado por lo que al utilizar funciones al programar se debe tener cuidado en operarlo de acuerdo a su tipo.

```
>>> 1 + abs(-10)
11
>>> 3*cos(14)+123
123.4102116546235
>>> range(20,30,1+1)
[20, 22, 24, 26, 28]
>>> 12 + str(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

4.2. Definición de funciones

Para definir una función se sigue la siguiente sintaxis:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def cuadrado(x):
    """Esta es una funcion que calcula el cuadrado"""
    return x ** 2

def suma(x, y):
    return x + y

print cuadrado(4)
print suma(5, 6)
```

La línea que empieza con **def** es la cabecera de la función y el fragmento de programa que contiene los cálculos que debe efectuar la función se denomina cuerpo de la función. El valor que se pasa a una función cuando es invocada se denomina parámetro o argumento. Las porciones de un programa que no son cuerpo de funciones forman parte del programa principal: son las sentencias que se ejecutarán cuando el programa entre en acción. El cuerpo de las funciones sólo se ejecutará si se producen las

correspondientes llamadas.[1]

Los nombres de las funciones deben ser únicos, no solo con respecto a otras funciones sino también a los nombres de las variables. En una función se pueden escribir todo tipo de programas, incluyendo bucles, condicionales, llamadas a otras funciones entre otras.

Si se desea saber si una persona es menor o mayor de edad, se puede utilizar la siguiente función:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


def es_mayor_de_edad(edad):
    if edad < 18:
        resultado = False
    else:
        resultado = True
    return resultado


print es_mayor_de_edad(16)
```

Un problema más complejo es escribir una función que responda si un número dado es o no *perfecto*. Se dice que un número es perfecto si es igual a la suma de todos sus divisores excluído él mismo. Por ejemplo, 28 es un número perfecto, pues sus divisores(excepto él mismo) son 1, 2, 4, 7 y 14, que suman 28.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


def es_perfecto(n):
    sumatorio = 0


    for i in range(1, n):
        if n % i == 0:
            sumatorio += i


    if sumatorio == n:
```

[1] [4] p. 212

```
        return True
    else:
        return False


print es_perfecto(28)
print es_perfecto(18)
```

Además de números se pueden pasar listas o cadenas como argumentos, en el siguiente ejemplo se suman todos los elementos de una lista:

```
def sumatorio(lista):
    s=0
    for numero in lista:
        s += numero
    return s


print sumatorio([1,2,3,4,6])
```

Debido a que en Python todo nombre de variable es tan solo un apuntador a un objeto, en Python se pueden asignar funciones con el operador =, tal como se haría con una variable.

```
>>> def funcion():
...     print "Soy una funcion"
...
>>> funcion()
Soy una funcion
>>> x = funcion
>>> x()
Soy una funcion
>>>
```

Por último, en Python, al no ser un lenguaje tipado, se pueden aplicar las mismas operaciones a diferentes tipos de datos, por ejemplo una función que sume dos parámetros puede hacer esa operación con números, listas, cadenas y en general cualquier conjunto de objetos que soporten esa operación.

4.2.1. Parámetros

Las funciones pueden tener varios parámetros y parámetros por defecto, por ejemplo:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


def area_rectangulo(altura, anchura=10):
    return altura * anchura


print area_rectangulo(3, 4)
print area_rectangulo(3)
print area_rectangulo(anchura=5, altura=6)
```

Se puede ver que se pueden pasar los parámetros en cualquier orden, mencionando en forma explícita el nombre de cada uno.

Parámetros arbitrarios

Una función puede recibir 'n' cantidad de parámetros, donde n es un número desconocido de argumentos. Estos argumentos llegaran a la función en forma de tupla. Para definir los parámetros arbitrarios, se coloca el nombre de la tupla precedida por un asterisco (*).

```
# -*- coding: utf-8 -*-


def n_parametros(param1, *n_params):
    print "Parametro 1 es: ", param1


    print "Parametros arbitrarios: ", n_params


print n_parametros(100, "Hola", 200, True, [1,2,3])
```

También se pueden recibir parámetros arbitrarios como pares de clave=valor, en este caso el nombre del parámetro debe estar precedido por dos asteriscos (**).

```
# -*- coding: utf-8 -*-


def n_parametros(param1, *n_params, **dict_params):
    print "Parametro 1 es: ", param1
```

```
    for param in n_params:
        print param

    # Los argumentos arbitrarios tipo clave, se recorren como diccionarios
    for clave in dict_params:
        print "El valor de la clave ", clave, " es ", dict_params[clave]


n_parametros(100, "Hola", 200, True, [1,2,3], val1="valor 1",
                   val2="valor 2", num1=123)
```

Parámetros empaquetados

El caso contrario al anterior es una función que espera una lista fija de parámetros y solo recibe una lista de valores. En este caso un asterisco (*) debe preceder al nombre de la lista o tupla pasada como parámetro.

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [1500, 10]
print calcular(*datos)

datos = [1500]
print calcular(100, *datos)
```

En caso se pase un diccionario, el parámetro debe estar precedido por dos asteriscos (**).

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = {"descuento": 10, "importe": 1500}
print calcular(**datos)
```

4.3. Parámetros por valor y referencia

En el paso por referencia lo que se pasa como argumento es una referencia o puntero a la variable, es decir, la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido en si. En el paso por valor, por el contrario, lo que se pasa como argumento es el valor que contenía la variable.

La diferencia entre ambos estriba en que en el paso por valor los cambios que se hagan sobre el parámetro no se ven fuera de la función, dado que los argumentos de la función son variables locales a la función que contienen los valores indicados por las variables que se pasaron como argumento. Es decir, en realidad lo que se le pasa a la función son copias de los valores y no las variables en si.

Si quisiéramos modificar el valor de uno de los argumentos y que estos cambios se reflejaran fuera de la función tendríamos que pasar el parámetro por referencia.

En C los argumentos de las funciones se pasan por valor, aunque se puede simular el paso por referencia usando punteros. En Java también se usa paso por valor, aunque para las variables que son objetos lo que se hace es pasar por valor la referencia al objeto, por lo que en realidad parece paso por referencia.

En Python también se utiliza el paso por valor de referencias a objetos, como en Java, aunque en el caso de Python, a diferencia de Java, todo es un objeto (para ser exactos lo que ocurre en realidad es que al objeto se le asigna otra etiqueta o nombre en el espacio de nombres local de la función).

Sin embargo no todos los cambios que hagamos a los parámetros dentro de una función Python se reflejarán fuera de esta, ya que hay que tener en cuenta que en Python existen objetos inmutables, como las tuplas, por lo que si intentáramos modificar una tupla pasada como parámetro lo que ocurriría en realidad es que se crearía una nueva instancia, por lo que los cambios no se verían fuera de la función.

(...)

En resumen: los valores mutables se comportan como paso por referencia, y los inmutables como paso por valor.

[2]

Si no se desea modificar un objeto mutable pasado como parámetro, se puede pasar una copia de este a la función o copiarlo dentro de la función, de tal manera que nunca sea modificado.

```
L = [1, 2]
changer(X, L[:])


def changer(a, b):
```

[2] [1] p. 40

```
    b = b[:]        # Copy input list so we don't impact caller
    a = 2
    b[0] = 'spam'   # Changes our list copy only
```

4.4. Variables locales y globales

En el cuerpo de las funciones se pueden definir y utilizar variables, las cuáles no son iguales a las variables que se crean en el cuerpo principal del programa. Cuando se define una variable dentro de una función, esta variable no puede ser utilizada fuera de la función ya que no existe fuera del ámbito donde fue creada.

Las variables que solo existen en el cuerpo de la función se llaman *variables locales* y las que existen en el cuerpo del programa son llamadas variables globales. También los parámetros de una función son considerados variables locales. al acabar la ejecución de una función, todas sus variables locales dejan de existir.

Las variables globales, a diferencia de las locales son accesibles desde cualquier parte del código, incluyendo una función, excepto si dentro de la función hay una variable con el mismo nombre, en cuyo caso se utiliza la variable local y no la variable global.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

cad = 'cad1'
c = 'cadena2'

def imprime_saludo(saludo):
    cad = saludo
    print cad, c

def prueba(f):
    f('Jelou')

prueba(imprime_saludo)
print cad
```

Si se referencia una variable global desde una función, esta es utilizada, pero si se le asigna un valor, la variable es local si relación con la variable global del mismo nombre.

```
>>> def func():
...     print x
...
>>> x = 10
>>> func()
10


>>> def func2():
...     x = 10
...     print x
...
>>> x = 30
>>> func2()
10
>>>
```

Si se desea modificar una variable global dentro de una función se utiliza la palabra **global**, que es lo más parecido a una declaración que se puede encontrar en Python, con la diferencia que no es una declaración de tipo o tamaño sino de **namespace**.

```
>>> def func():
...     global x
...     x = 34
...
>>> x = 10
>>> func()
>>> x
34
>>>
```

Aún cuando la variable no hubiera sido utilizada fuera de la función, al declararla con global pasa a estar en el ámbito global del módulo en que se ejecuta.

4.5. Buenas prácticas con funciones

- La función debe ser en lo posible independiente del exterior. Utilizar argumentos y devolver valores es con frecuencia la mejor manera de mantener la modularidad de la función.
- Solo usar variables globales cuando sea necesario.
- No cambiar los parámetros mutables a menos que eso sea explícito en la funcionalidad del método.
- Una función debe tener una y sola una funcionalidad.
- En lo posible una función debe ser pequeña. Keep it simple and short.

4.6. Módulos

Al escribir funciones se simplifican los programas, y no solo se simplifican sino que además esa función puede ser utilizados en otros programas, como en el siguiente ejemplo:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


def area_rectangulo(altura, anchura):
    return altura * anchura
```

A este programa se le dará el nombre de *area_rectangulo.py* y será utilizado desde el programa *calculo.py*, que es el siguiente:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


from area_rectangulo import area_rectangulo


print area_rectangulo(1,4)
```

Cada archivo de texto con código Python es un Módulo, los cuales proveen una forma de particionar el espacio de nombres (namespace). Todo objeto en Python "vive" en su respectivo módulo y para ser utilizado en otro debe ser explícitamente importado.

4.6.1. Arquitectura de un programa en Python

En un programa python, a menos que sea muy sencillo, van a existir varios archivos con código y configuraciones las cuales serán importadas por otros archivos del programa.

El script de mayor nivel (el primero en ser ejecutado) contiene el flujo de control del programa y utiliza el resto de módulos (archivos .py) para obtener librerías y configuraciones. A su vez cada módulo puede importar lo que necesite de otros módulos. En Python no existen tipos de archivos diferentes para programas y librerías como en otros lenguajes.

En la siguiente imagen se puede apreciar la arquitectura de un programa en Python. [3]

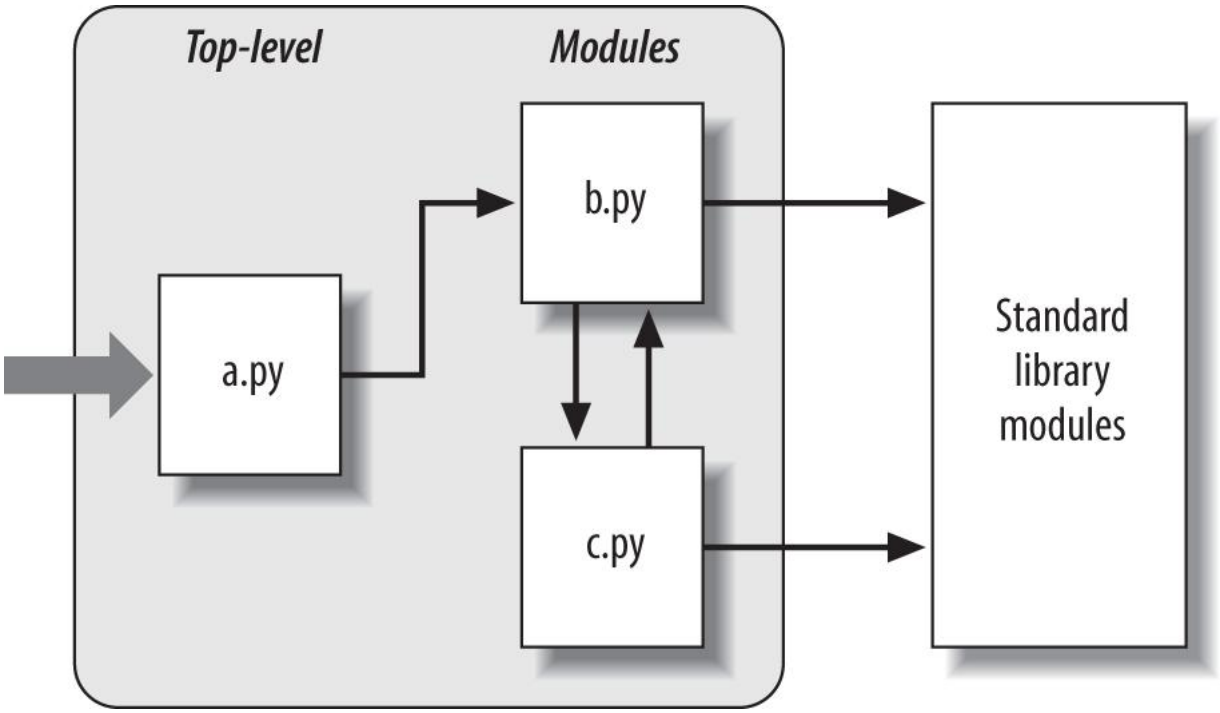

Gráfico 4.1: Arquitectura de un programa en Python

[3] [3] p. 672

4.7. Ejercicios

1. Escriba una función llamada raiz_cubica que devuelva el valor de la raíz cúbica de x.
2. Escriba una función llamada area_circulo que, a partir del radio de un círculo, devuelva el valor de su área. Utiliza el valor 3.1416 como aproximación de pi o importe el valor de pi que se encuentra en el módulo math.
3. Escriba una función que convierta grados Farenheit en grados centígrados. (Para calcular los grados centígrados se restan 32 a los grados Farenheit y se multiplica el resultado por cinco novenos.)
4. Implementa una función que reciba una lista de números y devuelva la media de dichos números. Tenga cuidado con la lista vacía (su media es cero).
5. Diseñe una función que calcule la multiplicación de todos los números que componen una lista.
6. Define una función que, dada una cadena x, devuelva otra cuyo contenido sea el resultado de concatenar 6 veces x consigo misma.
7. Escriba una función que, dada una lista de cadenas, devuelva la cadena más larga. Si dos o más cadenas miden lo mismo y son las más largas, la función devolverá una cualquiera de ellas.
8. Escriba una función que reciba dos listas y devuelva los elementos comunes a ambas, sin repetir ninguno (intersección de conjuntos).
9. Escriba una función que, dada una lista de números, devuelva otra lista que sólo incluya sus números impares.
10. Escriba una función que reciba una número indeterminado de números y devuelva el promedio.
11. Escriba una función que reciba un número indeterminado de pares clave=valor y los imprima en pantalla.

Capítulo 5

Programación orientada a objetos

5.1. POO en Python

En Python no solo se puede trabajar con el paradigma estructurado, además tiene completo soporte de la programación orientada a objetos (POO). En la POO los conceptos del mundo real se modelan a través de clases y objetos y los programas consisten en transacciones entre objetos.

Utilizar clases es opcional, se puede trabajar de forma estructurada mediante funciones o incluso solo con un script, dependiendo de la disponibilidad de tiempo con la que se trabaje. Programar utilizando POO requiere mayor tiempo de planeamiento y de preparación antes de empezar a implementar las funcionalidad buscadas, por lo que es más adecuado para proyectos de largo alcance y no para scripts que requieran resolver problemas en el acto.

5.2. Clases y Objetos

Una clase es la plantilla mediante la cual se crean objetos de un tipo, pero cada objeto puede tener valores diferentes en sus atributos. Como las funciones y los módulos, las clases son otra manera de empaquetar datos y reglas de negocio, incluso definen su propio namespace.

Un objeto es una entidad que agrupa estados y funcionalidades. Las variables (atributos) del objeto son las que definen el estado y las funcionalidades se implementan mediante funciones propias del objeto, también conocidas como métodos. Cada objeto mantiene su propio namespace pero obtiene todas sus variables y métodos de la clase que le sirve de plantilla.

La sintáxis para declarar una clase es:

```
>>> class NuevaClase:
        def metodo(self):
            print "Estamos en una nueva clase"


>>> objeto = NuevaClase()
```

```
>>> objeto.metodo()
Estamos en una nueva clase
>>>
```

Todo método de una clase recibe como parámetro implícito el argumento **self** (equivalente al **this** de C++), que es el objeto que llama al método, con el cual referencia a los atributos y métodos propios de la clase.

```
>>> class NuevaClase:
        def metodo(self):
            print "Estamos en una nueva clase"


>>> objeto = NuevaClase()
>>> objeto.metodo()
Estamos en una nueva clase
>>>
```

Para crear atributos en una clase se les asigna un valor, al igual que otras variables en Python.

```
>>> class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print self.data


>>>
```

También se pueden crear atributos en un objeto fuera de la clase, lo usual es hacerlo en el constructor pero no es obligatorio.

```
>>> obj = FirstClass()
>>> obj.atributo = 'Hola'
>>> obj.atributo
'Hola'
>>>
```

Al igual que las funciones y las variables, las clases viven en los módulos y pueden ser importadas de

otro módulo mediante una sentencia **import**.

Se puede hacer lo mismo con las funciones. No es una capacidad que se utilice con frecuencia, pero demuestra como las clases en Python son atributos en namespaces.

```
>>> def uppername(obj):
...     return obj.atributo.upper()
...
>>> uppername(obj)
'HOLA'
>>> FirstClass.metodo = uppername
>>> obj2 = FirstClass()
>>> obj2.atributo = 'HOLA'
>>> obj2.metodo()
'HOLA'
>>>
```

Más allá de la sintáxis, el sistema de clases de Python en realidad solo consiste en búsquedas de atributos en un árbol de objetos y un primer argumento especial para las funciones.

5.2.1. Atributos compartidos

En una clase se pueden tener atributos cuyo valor sea heredado por todos los objetos. Si modificamos la variable en un objeto, esta es modificada solo en el objeto correspondiente, pero si se modifica en la clase, el valor cambia en todos los objetos siempre que no hayan cambiado el valor del atributo.

```
>>> class SharedData:
        spam = 42


>>> x = SharedData()
>>> y = SharedData()
>>> x.spam, y.spam
(42, 42)
>>> x.spam += 1
>>> x.spam, y.spam
(43, 42)
>>> SharedData.spam = 99
```

```
>>> x.spam, y.spam
(43, 99)
>>>
```

5.2.2. Estándares

En Python el estándar para nombrar un módulo es empezar el nombre con una letra minúscula, y para las clases empezar con una letra mayúscula. No es obligatorio pero es la forma estándar de programar.

Por ejemplo si tenemos un módulo persona este se llamaría **persona.py** y si se tiene una clase Persona en este módulo su nombre empezará con mayúscula, en caso sea un nombre compuesto de varias palabras, cada una empezará con mayúscula.

```
>>> class Persona:
        pass


>>> class PersonaNatural(Persona):
        pass


>>>
```

5.2.3. Herencia

Una de las características inherentes a la POO es la herencia. Una clase (subclase) puede obtener todos los atributos y métodos de otra clase (superclase) y modificarlos, de tal manera que no es necesario cambiar o copiar la superclase. Al heredar una clase de otra se pueden generar enteras jerarquías de clases donde unas especializan la labor de otras reutilizando su código y agregando lo que se considere conveniente para la funcionalidad de la nueva clase. Mientras más se baja en la jerarquía, más específico es el software.

```
>>> class SecondClass(FirstClass):
        def display(self):
            print 'Current value = "%s"' % self.data


>>>
```

Cuando una subclase vuelve a implementar un método heredado de la superclase, lo está reemplazando. El otro método, **setdata** sigue funcionando igual que en **FirstClass**. Nada que se haga en

una subclase afecta a la superclase.

Cuando se hereda una subclase, esta puede reemplazar los métodos de la superclase y los métodos utilizados serán los de la subclase.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-


class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __repr__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)


class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)


if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    frank = Manager('Frank Wayne', 'Manager', 600000)
    print bob
    print sue
    print bob.lastName(), sue.lastName()
    sue.giveRaise(.10)
    print sue
    frank.giveRaise(.15)
```

```
    print frank
```

En el ejemplo, se puede apreciar como la clase Manager vuelve a implementar la función giveRaise() de la clase Person y para hacerlo se limita a llamar a la función giveRaise() original y sumarle un valor al parámetro que se le pasa. Una subclase no solo puede redefinir métodos, además puede agregar métodos propios.

Como toda clase es también un objeto, se pueden invocar las funciones directamente de las clases, siempre y cuando se pase de forma explícita el objeto sobre la cual se aplicará.

Herencia Múltiple

Al crear una subclase, esta puede heredar de varias superclases:

```
>>> class C1:
...       def metodo(self):
...           print "Estamos en el metodo 1"
...
>>> class C2:
...       def metodo(self):
...           print "Estamos en el metodo 2"
...
>>> class C3(C1,C2):
...      pass
...
>>> obj = C3()
>>> obj.metodo()
Estamos en el metodo 1
>>>
```

El orden de precedencia en la búsqueda en el árbol de atributos de la clase es de izquierda a derecha, con los métodos que aparecen en las clases más a la izquierda reemplazando a los métodos de igual nombre en las clases más a la derecha.

Clases abstractas

En las clases abstractas se definen métodos que no se van a utilizar en las instancias de la clase, solo existen para definir los métodos que tendrán las subclases. Al codificar un método de una clase abstracta se invoca una excepción para que al llamarlo se genere un error.

```
>>> class Super:
...     def delegate(self):
...         self.action()
...     def action(self):
...         raise NotImplementedError('action must be defined!')
...
>>> x = Super()
>>> x.action()


>>> x.action()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in action
NotImplementedError: action must be defined!
>>> x.delegate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in delegate
  File "<stdin>", line 5, in action
NotImplementedError: action must be defined!
>>>
```

Para las instancias de la subclase, el resultado es el mismo si no se reimplementa el método en la subclase.

```
>>> class Sub(Super): pass
...
>>> X = Sub()
>>> X.delegate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in delegate
  File "<stdin>", line 5, in action
NotImplementedError: action must be defined!
```

```
>>>


>>> class Sub(Super):
...     def action(self): print 'spam'
...
>>> X = Sub()
>>> X.delegate()
spam
>>>
```

En versiones de Python superiores a las 2.6 existe una sintaxis especial para declarar clases abstractas:

```
>>> from abc import ABCMeta, abstractmethod
>>> class Super:
...     __metaclass__ = ABCMeta
...     @abstractmethod
...     def method(self):
...         pass
...
>>> x = Super()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Super with abstract methods method
>>>
```

La ventaja de esta sintaxis es que no se puede instanciar objetos de una clase abstracta.

5.2.4. Poliformismo

En el ejemplo anterior se podría agregar el siguiente código al final:

```
if __name__ == '__main__':


    print '--All three--'
    for obj in (bob, sue, tom):
        obj.giveRaise(.10)
```

```
        print obj
```

En este ejemplo se puede ver como sin importar que objeto es igual se llama a la misma función y esta cumple ejecuta su código, con diferente resultado en el caso de un objeto Manager que de un objeto Person.

5.2.5. Atributos especiales de las clases

En las clases y objetos existen atributos especiales como __class__ que es la clase del objeto de la cual se llama, __name__ que es el nombre de la clase, __dict__ que es un diccionario con los atributos de una clase, entre otros atributos.

```
>>> from persona import Person
>>> bob = Person('Bob Smith')
>>> bob
[Person: Bob Smith, 0]
>>> print bob
[Person: Bob Smith, 0]
>>> bob.__class__
<class persona.Person at 0xb6a0a17c>
>>> bob.__class__.___name__
'Person'
>>> list(bob.__dict__.keys())
['pay', 'job', 'name']
>>> dir(bob)
['__doc__', '__init__', '__module__', '__repr__', 'giveRaise',
'job', 'lastName', 'name', 'pay']
>>>
>>> for key in bob.__dict__:
        print key, '=>', bob.__dict__[key]


('pay', '=>', 0)
('job', '=>', None)
('name', '=>', 'Bob Smith')
>>> for key in bob.__dict__:
        print key, '=>', getattr(bob, key)
```

```
('pay', '=>', 0)
('job', '=>', None)
('name', '=>', 'Bob Smith')
>>>
```

5.2.6. Sobrecarga de operadores

En Python las clases pueden redifinir la funcionalidad de los operadores del lenguaje como +, -, *, [], etc de tal manera que las clases se adapten mejor al estilo de codificación común en Python.

Cada operador tiene asignado un método especial que lo implementa en una clase. Por ejemplo, si una clase implementa el método __**add**__, este método es llamado cada vez que el objeto aparezca en una suma + y el valor de retorno del método se convierte en el resultado de la operación. Si la operación no es definida en una clase, intentar utilizarla va a producir una excepción.

En general es una característica que es utilizada por creadores de APIs para otros desarrolladores, no por desarrolladores que implementen un producto para el usuario final.

```
>>> class ThirdClass(SecondClass):
        def __init__(self, value):
            self.data = value
        def __add__(self, other):
            return ThirdClass(self.data + other)
        def __str__(self):
            return '[ThirdClass: %s]' % self.data

>>> a = ThirdClass('abc')
>>> a.display()
Current value = "abc"
>>> print a
[ThirdClass: abc]
>>> b = a + 'xyz'
>>> b.display()
Current value = "abcxyz"
>>> print b
[ThirdClass: abcxyz]
```

```
>>>
```

5.2.7. Constructores

Los atributos de las clases, al igual que las variables, pueden ser asignados a voluntad en cualquier punto del código, pero si se intenta utilizar una variable antes de que se le haya asignado un valor, la operación va a producir una excepción. Para evitar este hecho se utilizan los constructores, de tal manera que al instanciar un objeto se creen los atributos requeridos y se ejecuten las operaciones necesarias.

En Python el constructor se llama **__init__**. Es parte de la familia de métodos sobrecargados de Python, estos métodos se heredan en cada clase y se escriben con doble guión bajo a cada lado para hacerlos distintos. Python los ejecuta de forma automática cuando una instancia que los soporta llama a la operación correspondiente, son sobre todo una alternativa a una simple ejecución de un método. Son opcionales, si se omiten la operación no es soportada. Si no se implementa el método **__init__**, al instanciarse un objeto este se creará vacio.

La sintaxis del constructor es:

```
>>> class ClaseEjemplo:
        def __init__(self, argumento1):
            self.atributo = argumento1
            print self.atributo + ' fue bien recibido'

>>> ejem1 = ClaseEjemplo('Parametro')
Parametro fue bien recibido
>>>
```

Se puede heredar de alguna clase conocida y agregarle atributos y funciones:

```
>>> class ListaMejor(list):
        def __init__(self):
            self.nombre = 'meee'
        def setNombre(self, nombre):
            self.nombre = nombre
        def getNombre(self):
            return self.nombre

>>> lista = ListaMejor()
```

```
>>> lista
[]
>>> lista.nombre
'meee'
>>> lista.setNombre('Lista1')
>>> lista.getNombre()
'Lista1'
```

5.3. Ejercicios

1. Cree una clase que implemente una pila, sus métodos (pop, push) y devuelva cuantos items quedan en la pila.

2. Cree una clase que implemente una cola, y sus métodos y devuelva cuantos items quedan en la cola.

3. Crea una clase que implemente el tipo "Número imaginario". La clase debe soportar la suma y la resta, así como la impresión mediante print.

4. Crea una clase AutoCarreras que herede de una clase Auto sus atributos (número de llantas, de puertas, caballos de fuerza, velocidad, gasolina en el tanque) y métodos (agregarGasolina, arrancarMotor) y agregue nuevos atributos y métodos.

5. Crea una clase abstracta llamada FiguraGeometrica con los atributos nombre, num_lados y los métodos devolver_nombre() y calcular_area() y extiendala mediante las clases Rectángulo, Círculo, Triángulo y reimplemente las funciones de la superclase en cada una de las subclases.

6. Crea una clase Refresco y hereda tres clase de esta (Gaseosa, JugoFrutas y AguaMineral). En cada subclase reprograma el constructor para que al instanciar la clase se vea el nombre de la clase en pantalla.

7. Crea una clase NumeroRacional que tenga como atributo un numero racional (p. ej 5.5) e implemente los operadores de suma, resta, multiplicación, división y representación (__add__, __sub__, __mul__, __div__,__repr__).

8. Defina una clase *EnsaladaFrutas* con una atributo *frutas* que sea inicialmente ['melones', 'piñas', 'manzanas'] y un atributo *raciones* cuyo valor inicial sea 4. Escriba un método *__init__* que reciba los argumentos *ingredientes* (una lista de cadenas) y *num_raciones* (un entero) y guarde los valores obtenidos en los atributos y *frutas* y *raciones* (las que queden).

9. En el ejercicio anterior (*EnsaladaFrutas*) agregue un método *__str__* que imprima el número de raciones restantes y las frutas en la ensalada de frutas. La cadena debe verse como: "2 raciones en la ensalada de ['platanos', 'manzanas']"

10. En el ejercicio anterior escriba un método *agregar()* que obtenga una cadena de texto y la agregue al final de la lista *frutas*. Agregue además un método *servir()* que reduzca en uno la cantidad de raciones e imprima "Disfrute", si ya no hay raciones debe imprimir "Disculpe".

Capítulo 6

Excepciones

6.1. Las Excepciones

Las excepciones son eventos que pueden alterar el flujo de un programa. En Python las excepciones son lanzadas automáticamente cada vez que produce un error y pueden ser capturadas por el código del programador. Existen cuatro formas de procesar las excepciones, las que se estudiarán en el presente capítulo.

Se utilizan excepciones para manejar eventos no necesariamente contemplados en el flujo del programa. Por ejemplo si un programa se hace cargo de la comunicación con un web service en un servidor, si se cae la conexión debe haber una excepción que gestione esa eventualidad y cierre de forma ordenada las tareas que se estaban llevando a cabo.

El manejador de excepciones es la sentencia **try**, la cual deja una "marca" y algún código. En algún punto del programa se produce una condición que levanta una excepción y es llamado el código que se dejo para el caso de una excepción.

6.1.1. Captura de excepciones

Una excepción se genera cuando se comete algún error en el flujo del programa:

```
>>> def fetcher(obj, index):
...     return obj[index]
...
>>> x = 'spam'
>>> fetcher(x, 3)
'm'
>>> fetcher(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

El mensaje impreso es una excepción del tipo **IndexError** y además se ve el stack del programa que genero el error. Si se desea hacer algo diferente al manejo por defecto de excepciones de Python se utiliza **try/except**.

```
>>> try:
...     fetcher(x, 4)
... except IndexError:
...     print 'got exception'
...
got exception
>>>
```

En el segundo ejemplo, Python salta hacia el manejo de excepción definido por el programador, y no interrumpe la ejecución del programa. En ejemplos reales, no solo se captura la excepción sino se ejecuta código para resolver el problema presentado.

Se pueden capturar diversas excepciones en un try y agregar una sentencia **else** para el caso de que no se produzca ninguna excepción.

Las formas posibles de la sentencia try son:

Forma	Interpretación
except:	Captura las excepciones que no estén en otro except.
except name:	Captura una excepción específica.
except name as value:	Captura la excepción y la asigna a una variable.
except (name1, name2):	Captura cualquiera de las excepciones listadas.
except (name1, name2) as value:	Captura cualquiera y la asigna a una variable.
else:	Se ejecuta si no se presenta ninguna excepción.
finally:	Se ejecuta así se presenten o no excepciones.

Otro ejemplo:

```
def imprimir(x, y):
    print x + y
```

```
try:
    imprimir([0, 1, 2], 'spam')
except TypeError:
    print 'Hello world!'
    print 'resuming here'
```

Un ejemplo final, de mayores proporciones:

```
# File try2.py (Python 3.X + 2.X)

sep = '-' * 45 + '\n'
print(sep + 'EXCEPTION RAISED AND CAUGHT')

try:
    x = 'spam'[99]
except IndexError:
    print('except run')
finally:
    print('finally run')
    print('after run')

print(sep + 'NO EXCEPTION RAISED')
try:
    x = 'spam'[3]
except IndexError:
    print('except run')
finally:
    print('finally run')
    print('after run')

print(sep + 'NO EXCEPTION RAISED, WITH ELSE')
try:
    x = 'spam'[3]
except IndexError:
```

```
    print('except run')
else:
    print('else run')
finally:
    print('finally run')
    print('after run')



print(sep + 'EXCEPTION RAISED BUT NOT CAUGHT')
try:
    x = 1 / 0
except IndexError:
    print('except run')
finally:
    print('finally run')
    print('after run')
```

6.1.2. Levantamiento de excepciones

En Python no solo se pueden capturar excepciones producidas por fallos en el programa, sino también ser levantadas a propósito por el programa utilizando la sentencia **raise**.

```
>>> try:
...     raise IndexError
... except IndexError:
...     print 'got exception'
...
got exception
>>>
```

La sentencia raise en caso no sea manejada, genera una excepción que es tratada como todas las demás por Python.

```
>>> raise IndexError
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IndexError
>>>
```

6.1.3. Excepciones definidas por el usuario

Se pueden crear nuevas excepciones adicionales a las predefinidas por Python heredando una clase de la clase **Exception**.

```
>>> class AlreadyGotOne(Exception): pass
...
>>> def grail():
...     raise AlreadyGotOne()
...
>>> try:
...     grail()
... except AlreadyGotOne:
...     print 'got exception'
...
got exception
>>>
```

Al ser clases las excepciones, pueden ser heredadas y modificadas:

```
>>> class NuevaExcepcion(Exception):
...     def __str__(self): return 'Esta es una nueva excepcion.'
...
>>> raise NuevaExcepcion
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NuevaExcepcion: Esta es una nueva excepcion.
>>>
```

6.1.4. Terminar acciones

Después de un bloque **try** puede ser incluida la sentencia **finally**, la cuál se ejecuta siempre después del código incluido en el try así se levante o no una excepción. Es útil debido a que se va a ejecutar sin importar si el programa cometió o no un error, aportando así robustez a la ejecución.

```
>>> x = 'spam'
>>> def func():
...     try:
...         fetcher(x, 4)
...     finally:
...         print 'after fetch'
...     print('after try?')
...
>>> func()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in func
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

Debido al error, no se tiene el mensaje "after try" porque después del error el programa termina, pero si se ejecuta la sentencia incluida en el bloque finally. Si el programa no cometiera un error, ambas sentencias print se ejecutarían.

Se utiliza el **except** para capturar errores y **finally** para que estos errores no eviten la ejecución de código necsario para el programa. Por ejemplo, si el programa se comunica con un servidor del cuál recibe data, en caso la conexión se corte se utiliza la sentencia except para capturar el error y guardarlo en un log y **finally** para guardar la data recibida hasta el momento y cerrar de forma ordenada el programa.

6.1.5. Uso de las excepciones

En Python las excepciones son utilizadas sobre todo para:

- Gestión de errores: al producirse un problema Python levanta una excepción y si esta es ignorada por el programa, Python reacciona deteniendo la ejecución e imprimiendo el error.
- Notificación de eventos: se pueden utilizar las excepciones para definir que hacer en caso de que se produzca un evento.

- Manejo de casos especiales.

6.2. Ejercicios

1. Escriba un programa que itere sobre una lista utilizando un rango superior a la cantidad de elementos de la lista. Cuando se produzca el error, este debe ser capturado por una excepción e impresos sus datos.

2. Del problema anterior, agregue una sentencia **finally** que imprima la cantidad de elementos de la lista.

3. Escriba un programa que genere tres excepciones: IndexError, NameError y TypeError. Capture las tres excepciones e imprima los datos.

4. En el ejercicio anterior, levante una excepción adicional (mediante raise) y captúrela mediante un except genérico.

5. Escribe una función *ups()* que levante un error de tipo IndexError cuando es llamada. Escriba otra función desde la que se llame a *ups()* dentro de una estructura try/except y capture el error.

6. Implemente una excepción llamada MiError la cual debe tener un atributo data cuyo valor sea 42. Esta excepción será levantada desde la función *ups()* y en la función que la llama la excepción debe ser capturada y el dato mostrado.

Capítulo 7

Decoradores

7.1. Decoradores

Los decoradores (decorators) son una característica avanzada de Python donde una función tiene la capacidad de modificar otras funciones. Para entender los decoradores, primero se hará una revisión de otros conceptos de Python.

7.1.1. Ámbito

En Python las funciones crean un ámbito cada vez que son llamadas, cada fuhción tiene su propio espacio de nombres (namespace). Esto significa que cuando dentro de una función se llama a una variable, esta se busca primero en la función.

```
>>> a = "Variable global"
>>> def func1():
...     print locals()
...
>>> print globals()
{'a': 'Variable glonal', 'func1': <function func1 at 0xb73dabc4>,
'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__': None}
>>> func1()
{}
>>>
```

Dentro de una función se puede acceder a las variables globales siempre y cuando no haya otra variable del mismo nombre.

```
>>> cadena_a = "Variable global"
>>> def func1():
...     print cadena_a
...
>>> func1()
Variable global
>>>
```

En este ejemplo, al no encontrar Python una variable local cadena_a, utiliza la variable global. En cambio si se asigna la variable dentro de la función, se crea una nueva variable cuyo ámbito es func1().

```
>>> cadena_a = "Variable global"
>>> def func1():
...     cadena_a = 'text1'
...
>>> def func1():
...     cadena_a = 'prueba'
...     print locals()
...
>>> func1()
{'cadena_a': 'prueba'}
>>> cadena_a
'Variable global'
>>>
```

7.1.2. Vida de una variable

Las variables no solo viven en un espacio de nombres, también tienen tiempo de vida.

```
>>> def func1():
...     x = 1
...
>>> func1()
>>> print x
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Una vez la función es ejecutada, las variables locales desaparecen y ya no hay forma alguna de volver a invocarlas.

7.1.3. Argumentos y parámetros de una función

En una función los parámetros son variables locales:

```
>>> def func1(x):
...     print locals()
>>> func1(1)
{'x': 1}
```

Además, hay varias formas de definir los parámetros de una función y pasarle argumentos.

```
>>> def foo(x, y=0):
...     return x - y
>>> foo(3, 1)
2
>>> foo(3)
3
>>> foo()
Traceback (most recent call last):
  ...
TypeError: foo() takes at least 1 argument (0 given)
>>> foo(y=1, x=3)
2
```

Como se puede ver y recordar del capítulo 4, existen dos formas de pasar los argumentos, por la posición y como par clave=valor.

7.1.4. Funciones anidadas

Este un concepto que no se ha tratado antes en este curso. En Python se pueden crear funciones dentro de funciones y todas las reglas sobre ámbito de las variables funcionan.

```
>>> def outer():
...     x = 1
...     def inner():
...         print x
...     inner() # 2
...
>>> outer()
1
```

En la función *inner()* se imprime la variable x y como no existe en el ámbito local, se busca en el ámbito superior, la función *outer()*.

7.1.5. Las funciones son objetos

Como ya se ha estudiado antes, todo en Python es un objeto, incluyendo las funciones y clases, entre otras. Una variable puede apuntar a una función como apuntaría a una lista o un diccionario.

```
>>> def add(x, y):
...     return x + y
>>> def sub(x, y):
...     return x - y
>>> def apply(func, x, y): # 1
...     return func(x, y) # 2
>>> apply(add, 2, 1) # 3
3
>>> apply(sub, 2, 1)
1
```

No solo se pueden pasar funciones como argumentos, sino devolver funciones desde una función:

```
>>> def outer():
...     def inner():
...         print "Inside inner"
```

```
...     return inner
...
>>> foo = outer()
>>> foo
<function inner at 0x...>
>>> foo()
Inside inner
```

Al igual que con las variables, las funciones se pueden asignar y devolver.

7.1.6. Closures

Si se modifica el ejemplo anterior se obtiene:

```
>>> def outer():
...     x = 1
...     def inner():
...         print x
...     return inner
>>> foo = outer()
>>> foo.func_closure
(<cell at 0xb695ff44: int object at 0x8d842e0>,)
```

A primera vista parece que la variable x es local a la función *outer()* por lo que al llamarse la función *inner()* al momento de hacer el retorno de *outer()* debería desaparecer pero la variable se mantiene.

Python soporta una funcionalidad llamada **function closures** que implica que funciones internas definidas en un ámbito no global recuerdan como eran sus namespaces al momento de su **definición**.

Se puede hacer lo mismo no con una variable interna sino con un parámetro:

```
>>> def outer(x):
...     def inner():
...         print x # 1
...     return inner
>>> print1 = outer(1)
>>> print2 = outer(2)
>>> print1()
```

```
1
>>> print2()
2
```

Está técnica por si sola es poderosa, desde una función *outer()* se pueden construir funciones *inner* y devolversas, pasándoles los parámetros que se deseen.

7.1.7. Decoradores

Un decorador es un "callable" que recibe una función como argumento y devuelve otra función.

```
>>> def outer(some_func):
...     def inner():
...         print "before some_func"
...         ret = some_func()
...         return ret + 1
...     return inner
>>> def foo():
...     return 1
>>> decorated = outer(foo)
>>> decorated()
before some_func
2
```

A la función *outer* se le pasa un argumento *some_func*. Dentro de *outer* hay una función *inner* la cual imprime una cadena antes de ejecutar *some_func* y luego devolver su respuesta más 1, modificando así la salida de la función. Luego se pasa *foo* a *outer* y el resultado se asigna a *decorated*.

La variable *decorated* es la versión "decorada" de *foo* - es *foo* más algo adicional. Es más, si el decorador fuera realmente útil, reemplazariamos *foo* con su versión decorada:

```
>>> foo = outer(foo)
>>> foo()
before some_func
2
>>>
```

Ahora todas las llamadas a *foo* no obtendrían el original, sino la versión decorada.

Para mostrar con más detalle se utilizará un ejemplo con objetos coordenada. Estos objetos serán

pares x e y y se harán algunas operaciones con los objetos.

```
>>> class Coordinate(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __repr__(self):
...         return "Coord: " + str(self.__dict__)
>>> def add(a, b):
...     return Coordinate(a.x + b.x, a.y + b.y)
>>> def sub(a, b):
...     return Coordinate(a.x - b.x, a.y - b.y)
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> add(one, two)
Coord: {'y': 400, 'x': 400}
```

Si además de sumar y restar se desea chequear que no pasen ciertos límites, por ejemplo mantener solo coordenadas positivas. Hasta ahora las operaciones son:

```
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> three = Coordinate(-100, -100)
>>> sub(one, two)
Coord: {'y': 0, 'x': -200}
>>> add(one, three)
Coord: {'y': 100, 'x': 0}
```

Si se desea implementar límites se puede crear un decorador:

```
>>> def wrapper(func):
...     def checker(a, b):
...         if a.x < 0 or a.y < 0:
...             a = Coordinate(a.x if a.x > 0 else 0, a.y if a.y > 0 else 0)
...         if b.x < 0 or b.y < 0:
...             b = Coordinate(b.x if b.x > 0 else 0, b.y if b.y > 0 else 0)
```

```
...         ret = func(a, b)
...         if ret.x < 0 or ret.y < 0:
...             ret = Coordinate(ret.x if ret.x > 0 else 0,
                                 ret.y if ret.y > 0 else 0)
...         return ret
...     return checker
>>> add = wrapper(add)
>>> sub = wrapper(sub)
>>> sub(one, two)
Coord: {'y': 0, 'x': 0}
>>> add(one, three)
Coord: {'y': 200, 'x': 100}
```

Con el decorador las funciones se utilizan igual que antes solo que ahora devuelven sus valores en un límite. Si se desea, se puede hacer más obvia la aplicación del decorador mediante el símbolo @.

En vez de hacer:

```
>>> add = wrapper(add)
```

Se puede "decorar" mediante el símbolo @ y el nombre del decorador:

```
>>> @wrapper
... def add(a, b):
...     return Coordinate(a.x + b.x, a.y + b.y)
```

Esta sintáxis no es diferente de reemplazar *add* con la función retornada por *wrapper*, solo se hace más explícito.

7.1.8. *args y *kwargs

Los decoradores también se pueden utilizar con funciones que reciban una indeterminada cantidad de parámetros, así el decorador se puede aplicar sobre mayor cantidad de funciones.

```
>>> def one(*args):
...     print args # 1
>>> one()
()
```

```
>>> one(1, 2, 3)
(1, 2, 3)
>>> def two(x, y, *args): # 2
...     print x, y, args
>>> two('a', 'b', 'c')
a b ('c',)
```

En caso que se pase una lista o tupla, sus elementos pueden ser utilizados como argumentos si es que se coloca un * delante.

```
>>> def add(x, y):
...     return x + y
>>> lst = [1,2]
>>> add(lst[0], lst[1]) # 1
3
>>> add(*lst) # 2
3
```

También se pueden utulizar diccionarios para pasar pares clave=valor como argumentos:

```
>>> def foo(**kwargs):
...     print kwargs
>>> foo()
{}
>>> foo(x=1, y=2)
{'y': 2, 'x': 1}
```

Y también:

```
>>> dct = {'x': 1, 'y': 2}
>>> def bar(x, y):
...     return x + y
>>> bar(**dct)
3
```

7.1.9. Decoradores más genericos

Utilizando los argumentos de número variable se pueden escribir decoradores más genéricos, como un decorador que escriba las variables pasadas a una función antes de mostrar su resultado:

```
>>> def logger(func):
...     def inner(*args, **kwargs):
...         print "Arguments were: %s, %s" % (args, kwargs)
...         return func(*args, **kwargs)
...     return inner
```

La función inner recibe una cantidad indeterminada de a y se los pasa a la función **func** pasada como argumentos. Esto permite decorar cualquier función, sin importar los parámetros que reciba:

```
>>> @logger
... def foo1(x, y=1):
...     return x * y
>>> @logger
... def foo2():
...     return 2
>>> foo1(5, 4)
Arguments were: (5, 4), {}
20
>>> foo1(1)
Arguments were: (1,), {}
1
>>> foo2()
Arguments were: (), {}
2
```

7.2. Ejercicios

1. Escriba un decorador que imprima un mensaje antes y después de ejecutar una función.

2. Crea un decorador que imprima los argumentos de la función y el retorno de cada llamada a la función. Debe soportar una cantidad indeterminada de argumentos posicionales y por nombre. Su uso debe ser como sigue:

```
>>> @logged
... def func(*args):
...     return 3 + len(args)
>>> func(4, 4, 4)
you called func(4, 4, 4)
it returned 6
6
```

3. Cree una clase FuncionesAritméticas que tenga una lista de números como atributo y funciones que implementen las cuatro operaciones sobre los elementos de dicha lista. Agreguele un decorador a cada función de manera tal que antes de mostrar el resultado imprima los elementos presentes en la lista.

4. Agregue un decorador a una función que imprima un mensaje de "Hola" al llamar a una función. Luego agregue otro decorador que diga "¿Cómo estás?" luego del mensaje del anterior decorador.

Capítulo 8

Persistencia de la data

8.1. Gestión de archivos de texto plano

En todo sistema operativo hay unidades de almacenamiento con nombre, también llamados archivos. Mediante la función *open()* se abre un archivo y se accede a este mediante las funciones que Python provee para el manejo de archivos.

Operación	**Significado**
output = open('/home/pedro/spam', 'w')	Crear archivo de escritura.
input = open('data', 'r')	Crear archivo de lectura ('r' significa read)
input = open('data')	Igual que la anterios ('r' es el default)
aString = input.read()	Leer un archivo en una sola cadena
aString = input.read(N)	Leer hasta N caracteres(o bytes) en una cadena
aString = input.readline()	Leer la siguiente línea en una cadena
aList = input.readlines()	Leer un archivo entero en una lista de cadenas.
output.write(aString)	Escribe una cadena de caracteres en un archivo
output.writelines(aList)	Escribe las cadenas de una lista en un archivo
output.close()	Cierre del archivo
output.flush()	Grabar el buffer en el disco sin cerrar
anyFile.seek(N)	Cambiar la posición en el archivo a N.
for line in open('data'): use line	Leer línea por línea.
codecs.open('f.txt', encoding='utf8')	Python 2.X Unicode archivos de texto (cadenas unicode).
open('f.bin', 'rb')	Python 2.X bytes files (cadenas str).

Para abrir un archivo se utiliza:

```
archivo = open(nombre\_archivo, modo)
archivo.metodo()
```

El primer parámetro debe incluir tanto el nombre como la ruta a la ubicación del archivo, si no se incluye ruta, Pytho interpreta que está ubicado en el mismo directorio. El parámetro modo es la forma en que se abre el archivo y puede ser "r" para lectura, "w" para escritura y "a" para adicionar al final del texto. Agregando un + en el modo se abre el archivo para lectura y escritura.

Una vez que se obtiene un objeto archivo, se pueden llamar sus métodos para leer o escribir del archivo.

```
>>> myfile = open('myfile.txt', 'w')
>>> myfile.write('hello text file\n')
>>> myfile.write('goodbye text file\n')
>>> myfile.close()
>>> myfile = open('myfile.txt')
>>> myfile.readline()
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()
''
```

Si se quiere leer con una sola línea el contenido de un archivo:

```
>>> open('myfile.txt').read()
'hello text file\ngoodbye text file\n'
>>> print(open('myfile.txt').read())
hello text file
goodbye text file
```

O si se desea iterar línea por línea:

```
>>> for line in open('myfile.txt'):
...     print(line)
...
hello text file
```

```
goodbye text file
```

8.1.1. Guardar Objetos

Para guardar objetos de diversos tipos en archivos de texto, deben ser convertidos en cadenas y luego guardados.

```
>>> X, Y, Z = 43, 44, 45
>>> S = 'Spam'
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w' )
>>> F.write(S + '\n')
>>> F.write('%s,%s,%s\n' % (X, Y, Z))
>>> F.write(str(L) + '@' + str(D) + '\n')
>>> F.close()
```

Luego de crear un archivo, se puede abrir y leer el contenido en una cadena.

```
>>> chars = open('datafile.txt').read()
>>> chars
"Spam\n43,44,45\n[1, 2, 3]@{'a': 1, 'b': 2}\n"
>>> print chars
Spam
43,44,45
[1, 2, 3]@{'a': 1, 'b': 2}
```

Para reconvertir la data guardada nuevamente en objetos Python se requiere hacerlo de forma manual.

```
>>> F = open('datafile.txt')
>>> line = F.readline()
>>> line
'Spam\n'
```

```
>>> line.rstrip()
'Spam'
>>>
```

Para la primera línea se utiliza el método rstrip para eliminar el caracter de nueva línea. En caso hubieran habido espacios en blanco también los hubiera eliminado.

Luego se cogen la siguiente línea y se extraen los números de ella.

```
>>> line = F.readline()
>>> line
'43,44,45\n'
>>> parts = line.split(',')
>>> parts
['43', '44', '45\n']
```

Se tiene la lista con los números pero aún son cadenas, por lo que se necesita convertirlos a números:

```
>>> numbers = [int(P) for P in parts]
>>> numbers
[43, 44, 45]
```

Para convertir la lista y el diccionario de la tercera línea del archivo se utilizará **eval** una función que trata a una cadena de texto que ingresa como código ejecutable.

```
>>> line = F.readline()
>>> line
"[1, 2, 3]@{'a': 1, 'b': 2}\n"
>>> parts = line.split('@')
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
>>> objects = [eval(P) for P in parts]
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

8.1.2. La librería Pickle

Usar **eval** es algo arriesgado, es una función que va a ejecutar cualquier código que se le pase, sin importar lo que este haga. Si se desea guardar objetos Python en un archivo, la librería Pickle es lo ideal.

```
>>> import pickle
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'w')
>>> pickle.dump(D, F)
>>> F.close()
>>>
>>> F = open('datafile.pkl', 'r')
>>> E = pickle.load(F)
>>> E
{'a': 1, 'b': 2}
```

El módulo **pickle** lleva a cabo lo que se conoce como serialización de objetos, convertir objetos a y desde cadenas de bytes. Se puede intentar leer el diccionario de la forma habitual pero no habría mucho por ver.

```
>>> open('datafile.pkl', 'r').read()
"(dp0\nS'a'\np1\nI1\nsS'b'\np2\nI2\ns."
```

8.1.3. JSON

Pickle utiliza un formato propio de Python y optimizado para perfomance durante años. JSON es un formato relativamente nuevo y muy utilizado para intercambiar información. Se puede también guardar objetos en formato JSON, en especial diccionarios y listas, cuya sintaxis se parece a la de JSON. Por ejemplo un diccionario Python con estructuras de datos anidadas es muy similar a la data de JSON.

```
>>> name = dict(first='Bob', last='Smith')
>>> rec = dict(name=name, job=['dev', 'mgr'], age=40.5)
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}}
```

Para convertirlo a JSON y de JSON de vuelta a Python:

```
>>> import json
>>> json.dumps(rec)
'{"age": 40.5, "job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}}'
>>> s = json.dumps(rec)
>>> s
'{"age": 40.5, "job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}}'
>>> O = json.loads(s)
>>> O
{u'age': 40.5, u'job': [u'dev', u'mgr'],
u'name': {u'last': u'Smith', u'first': u'Bob'}}
>>> O == rec
True
```

Se puede guardar la data en JSON en archivos:

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
    "age": 40.5,
    "job": [
        "dev",
        "mgr"
    ],
    "name": {
        "last": "Smith",
        "first": "Bob"
    }
}
>>> P = json.load(open('testjson.txt'))
>>> P
{u'age': 40.5, u'job': [u'dev', u'mgr'],
u'name': {u'last': u'Smith', u'first': u'Bob'}}
```

En Python también se puede utilizar el formato csv, así como xml, pero por razones de espacio no vamos a verlo en el presente curso.

8.2. MySQL

La persistencia de data utilizando bases de datos relacionales en Python tienen amplio soporte. Se puede acceder a las bases de datos más conocidas como PostgreSQL, Oracle, MySQL, SQLite entre otras a través de un API consistente, con pocas variantes entre la forma de acceder a una u otra base de datos.

En Python el acceso a las bases de datos está basado en tres conceptos clave:

- *Objetos de conexión*: Representa una conexión a la base de datos, son la interfaz a las operaciones, proveen detalles de implementación y generan objetos cursor.
- *Objetos cursor*: Representa una sentencia SQL enviada como cadena y puede ser utilizada para acceder y moverse por los resultados de la sentencia.
- *Resultados de las consultas*: Son retornados a Python como secuencias anidadas (listas de tuplas por ejemplo), representando tablas o registros. Dentro de los registros, los valores de cada columna son objetos de Python como cadenas, enteros, flotantes, etc.

Para conectarse a un servidor MySQL se sigue el siguiente código:

```
>>> from MySQLdb import Connect
>>> conn = Connect(host='localhost', user='root', passwd='qwerty')
```

Después de la conexión hay una variedad de operaciones que se pueden llevar a cabo como:

```
conn.close()
conn.commit()
conn.rollback()
```

Una de las cosas más útiles que s epueden hacer con una conexión es crear un objeto cursor:

```
cursobj = connobj.cursor()
```

Los objetos cursor tienen algunos métodos (como *callproc* para llamar a un procedimiento almacenado) pero el más importante es **execute**:

```
cursobj.execute(sqlstring [, parameters])
```

El método **execute** puede servir para ejecutar diversos tipos de sentencias SQL como CREATE, UPDATE, INSERT, SELECT, etc.

Después de ejecutar el método **execute** el atributo **rowcount** del cursor da el número de filas modificadas o consultadas y el atributo **description** da los nombres de las columnas y sus tipos luegos de una consulta. Para completar la operación en consultas se llama a uno de los métodos **fetch**.

```
tuple = cursobj.fetchone()
listoftuple = cursobj.fetchmany([size])
listoftuple = cursobj.fetchall()
```

8.2.1. Un breve tutorial

El código utilizado en el ejemplo presentado es muy similar, a excepción de la conexión, al código utilizado para acceder a otros gestores de bases de datos. En general solo es necesario cambiar la conexión y el resto del código puede seguir igual al cambiar de gestor.

Para conectarse:

```
>>> from MySQLdb import Connect
>>> conn = Connect(host='localhost', user='root', passwd='qwerty')
```

Se empieza importando la librería que conecta a MySQL y estableciendo una conexión.

Se crea un cursor para enviar sentencias SQL al servidor de bases de datos y se envía una para crear la primera tabla:

```
>>> curs = conn.cursor()
>>> use_bd = 'USE test'
>>> curs.execute(use_bd)
>>> tblcmd = 'create table people (name char(30), job char(10), pay int(4))'
>>> curs.execute(tblcmd)
```

Se crea un cursor, se escoge que BD usar y luego se crea una tabla con tres columnas.

Para agregar registros se utiliza una sentencia INSERT:

```
>>> curs.execute('insert into people values (%s, %s, %s)', ('Bob', 'dev', 5000))
1L
>>> curs.rowcount
1L
```

La sentencia inserta agrega un registro a la tabla people y rowcount devuelve cuantos registros se han producido o afectado por la última sentencia.

Para insertar múltiples registros con una sola sentencia se utiliza el método *executemany* y una secuencia de registros a insertar (p. ej una lista de listas). Este método funciona como se si se llamara

a execute por cada lista ingresada.

```
>>> curs.executemany('insert into people values (%s, %s, %s)',
...           [ ('Sue', 'mus', '70000'),
...             ('Ann', 'mus', '60000')])
2L
```

Se puede hacer lo mismo mediante un for:

```
>>> rows = [['Tom', 'mgr', 100000],
...         ['Kim', 'adm', 30000],
...         ['pat', 'dev', 90000]]
>>>
>>> for row in rows:
...     curs.execute('insert into people values (%s , %s, %s)', row)
...
1L
1L
1L
>>> conn.commit()
```

Para que todos los cambios hechos se guarden en la BD, se utiliza el método commit del objeto conexión. En caso no se llame, cuando se cierre la conexión (mediante *conn.close()* o cuando se termine el programa) se perderán todos los cambios enviados.

Ejecutar consultas

Luego de agregar 6 registros en la tabal people, se pueden ejecutar consultas para verlos.

```
>>> curs.execute('select * from people')
6L
>>> curs.fetchall()
(('Bob', 'dev', 5000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 60000L),
('Tom', 'mgr', 100000L), ('Kim', 'adm', 30000L), ('pat', 'dev', 90000L))
```

Se puede también usar un for:

```
>>> curs.execute('select * from people')
6L
>>> for row in curs.fetchall():
...     print row
...
('Bob', 'dev', 5000L)
('Sue', 'mus', 70000L)
('Ann', 'mus', 60000L)
('Tom', 'mgr', 100000L)
('Kim', 'adm', 30000L)
('pat', 'dev', 90000L)
```

Debido a que son tuplas se puede hacer lo siguiente:

```
>>> curs.execute('select * from people')
6L
>>> for (name, job, pay) in curs.fetchall():
...     print(name, ':', pay)
...
('Bob', ':', 5000L)
('Sue', ':', 70000L)
('Ann', ':', 60000L)
('Tom', ':', 100000L)
('Kim', ':', 30000L)
('pat', ':', 90000L)
```

Se puede utilizar *fetchone* para retornar registro por registro:

```
>>> curs.execute('select * from people')
6L
>>> while True:
...     row = curs.fetchone()
...     if not row: break
...     print row
```

```
...
('Bob', 'dev', 5000L)
('Sue', 'mus', 70000L)
('Ann', 'mus', 60000L)
('Tom', 'mgr', 100000L)
('Kim', 'adm', 30000L)
('pat', 'dev', 90000L)
```

Cuando se hace *fetchall* se agota la tabla resultante, al igual que cuando se llega al final mediante *fetchone*, por lo que se debe utilizar *execute* de nuevo para tener otro set de registros con el que trabajar.

Además de INSERT y SELECT se pueden ejecutar UPDATE

```
>>> curs.execute('update people set pay=%s where pay <= %s', [65000, 60000])
3L
>>> curs.execute('select * from people')
6L
>>> curs.fetchall()
(('Bob', 'dev', 65000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L),
 ('Tom', 'mgr', 100000L), ('Kim', 'adm', 65000L), ('pat', 'dev', 90000L))
```

Y DELETE:

```
>>> curs.execute('delete from people where name = %s', ['Bob'])
1L
>>> curs.execute('delete from people where pay >= %s',(90000,))
2L
>>> curs.execute('select * from people')
3L
>>> curs.fetchall()
(('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L), ('Kim', 'adm', 65000L))
>>> conn.commit()
```

Script de ejemplo

Este Script fue obtenido de [2].

```
from MySQLdb import Connect

conn = Connect(host='localhost', user='root', passwd='qwerty')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass  # Did not exist

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')
curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev',
50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev',
60000))
curs.execute('insert people values (%s, %s, %s)', ('Ann', 'mgr',
40000))
curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Bob',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]

while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))
```

```
conn.commit()       # Save inserted records
```

8.3. Ejercicios

1. Crear un programa que al ser ejecutado cree un archivo de texto que contenga el himno nacional.
2. Crear un programa que lea el archivo del ejercicio anterior y lo imprima en pantalla.
3. Escriba un programa que lea entradas del teclado y las escriba en un archivo, luego de recibir algo debe preguntar "¿Desea ingresar otro texto (S/N)?", si se ingresa S se sigue ingresando textos y si la respuesta es N se muestra el archivo en pantalla y termina el programa.
4. Escriba un programa que guarde dos listas y dos diccionarios en diferentes líneas de un archivo.
5. Recupere las listas y diccionarios del ejercicio anterior.
6. Escriba un programa que cree una tabla Persona con los campos nombre, apellido paterno, apellido materno y DNI.
7. Ingrese tres registros en la tabla anterior.
8. Escriba un programa que cree uan tabla Persona como la de un ejercicio anterior pero le agregue un FK a una tabla Profesión. Llene la tabla Profesión con cuatro registros y la tabla Persona con 6 registros.

Fuentes de Información

[1] Duque, Raúl González. *Python para todos.*

[2] Lutz, Mark. *Programming Python.* 4 edición, 2011.

[3] Lutz, Mark. *Learning Python.* O'Reilly, 2012.

[4] Marzal, Andrés y Gracia, Isabel. *Introducción a la programación con Python.* 2003.

[5] Stallings, William. *Organización y Arquitectura de Computadoras.* 7 edición, 2005.

[6] Ziadé, Tarek. *Expert Python Programming.* 2008.