Lenguaje interpretado o de script

Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados).

En Python, como en Java y muchos otros lenguajes, el código fuente se traduce a un pseudo código máquina intermedio llamado **bytecode** la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

Tipado Dinamico

La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

Fuertemente Tipado

No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o string) no podremos tratarla como un número (sumar la cadena “9” y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

Multiplataforma

El intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, FreeBSD, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

Orientado a Objetos y Programacion Funcional

La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos. Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.

¿Por qué Python?

Python es un lenguaje que todo el mundo debería conocer. Su sintaxis simple,

clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.

La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar.

Ejecucion de Python

Existen dos formas de ejecutar código Python. Podemos escribir líneas de código en el intérprete y obtener una respuesta del intérprete para cada línea (sesión interactiva) o bien podemos escribir el código de un programa en un archivo de texto y ejecutarlo.

Si utilizas Linux (u otro Unix) para conseguir este comportamiento, es decir, para que el sistema operativo abra el archivo .py con el intérprete adecuado, es necesario añadir una nueva línea al principio del archivo:

```
#!/usr/bin/python
print “Hola Mundo”
raw_input()
```

A esta línea se le conoce en el mundo Unix como **shebang, hashbang o sharpbang**. El par de caracteres #! indica al sistema operativo que dicho script se debe ejecutar utilizando el intérprete especificado a continuación.

```
chmod +x hola.py # darle autorizacion al file
./hola.py # corre el script
python hola.py # otra manera de correr el script
```

TIPOS BASICOS

Escalares

1. numeros enteros (int)
2. numeros flotantes (float)
3. numero complejos

Secuenciales

1. cadenas (strings)
2. listas (lists)
3. tuplas (tuples)
4. diccionarios (dictionaries)

Booleanos

1. True (verdadero)
2. False (falso)

Escalares

Numeros Enteros: Los números enteros son aquellos números positivos o negativos que no tienen decimales (además del cero).

1. ***Integer (int)***
2. ***Long***

Numeros Reales: Los números reales son los que tienen decimales. En Python se expresan mediante el tipo float .

Numeros Complejos: Los números complejos son aquellos que tienen parte imaginaria.
complejo = 2.1 + 7.8j

1.- Octal

El literal que se asigna a la variable también se puede expresar como un **octal**, anteponiendo un cero:
# 027 octal = 23 en base 10
entero = 027

2.- Hexadecimal

bien en hexadecimal, anteponiendo un 0x:
# 0×17 hexadecimal = 23 en base 10
entero = 0×17

Operadores

Veamos ahora qué podemos hacer con nuestros números usando los operadores por defecto. Para operaciones más complejas podemos recurrir al módulo math .

Operadores aritméticos

Operador	Suma	Ejemplo	
+	Suma	r = 3 + 2	# r es 5
-	Resta	r = 4 – 7	# r es -3
-	Negacion	r = -7	# r es -7

*	Multiplicacion	r = 6 * 2	# r es 12
**	Exponente	r = 2 ** 6	# r es 64
/	Division	r = 3.5 / 2	# r es 1.75
//	Division entera	r = 3.5 // 2	# r es 1
%	Modulo	r = 7 % 2	# r es 1

Operadores a nivel de bit

estos son operadores que actúan sobre las representaciones en binario de los operandos.

Por ejemplo, si veis una operación como 3 & 2 , lo que estais viendo es un and bit a bit entre los números binarios 11 y 10 (las representaciones en binario de 3 y 2). El operador and (&), del inglés “y”, devuelve 1 si el primer bit operando es 1 y el segundo bit operando es 1. Se devuelve 0 en caso contrario.

Operador	Descripcion	Ejemplo	
&	and	r = 3 & 2	# r es 2
|	or	r = 3 | 3	# r es 3
^	xor	r = 3 ^ 2	# r es 1
~	not	r = 3 ~ 2	# r es -4
<<	Desplazamiento izquierda	r = 3 << 1	# r es 6
>>	Desplazamiento derecha	r = 3 >> 1	# r es 1

Cadenas

Las cadenas no son más que texto encerrado entre comillas simples (‘cadena’) o dobles (“cadena”). Dentro de las comillas se pueden añadir caracteres especiales escapándolos con \ , como \n , el carácter de nueva línea, o \t , el de tabulación.

También es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter \n , así como las comillas sin tener que escaparlas.

```
triple = """primera linea
          esto se vera en otra linea"""
```

Las cadenas también admiten operadores como **+** , que funciona realizando una **concatenación** de las cadenas utilizadas como operandos y ***** , en la que se repite la cadena tantas veces como lo indique el número utilizado como segundo operando.

```
a = "uno"
b = "dos"
c = a + b       # c es "unodos"
d = a * 3       # d es "unounouno"
```

Booleanos

Como decíamos al comienzo del capítulo una variable de tipo booleano sólo puede tener dos valores: **True** (cierto) y **False** (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante.

Operadores lógicos o condicionales

Operador	Descripcion	Ejemplo
and	**Se cumple a y b ?**	**r = True and False # r es False**
or	**Se cumple a o b ?**	**r = True or False # r es True**
not	**No a**	**r = not True # r es False**

Operadores relacionales (comparaciones entre valores)

Operador	Descripcion	Ejemplo
= =	**Son iguales a y b ?**	**r = 5 == 3 # r es False**
!=	**Son distintos a y b ?**	**r = 5 != 3 # r es True**
<	**Es a menor que b ?**	**r = 5 < 3 # r es False**
>	**Es a mayor que b ?**	**r = 5 > 3 # r es True**
<=	**Es a menor o igual que b ?**	**r = 5 <= 5 # r es True**
>=	**Es a mayor o igual que b**	**r = 5 >=3 # r es True**

	?	

Colecciones

Anteriormente vimos algunos tipos básicos, como los números, las cadenas de texto y los booleanos. Ahora veremos algunos tipos de **colecciones de datos: listas, tuplas y diccionarios.**

Listas

La **lista** es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por **arrays, o vectores.** Las listas pueden contener cualquier tipo de dato: **números, cadenas, booleanos, ... y también listas.**

Crear una lista es tan sencillo como indicar entre corchetes, y separados por comas, los valores que queremos incluir en la lista:
lista = [22, True, "una lista", [1, 2]]

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes. Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1:
lista = [11, False]
mi_var = lista[0] # salida: mi_var vale 11

Si queremos acceder a un elemento de una lista incluida dentro de otra lista tendremos que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:
lista = ["una lista", [1, 2]]
mi_var = lista[1][0] # salida: mi_var vale 1

También podemos utilizar este operador para modificar un elemento de la lista si lo colocamos en la parte izquierda de una asignación:.
lista= [22, True]
lista[0] = 99 # modifica <lista>, la cual sera: [99, True]

Una curiosidad sobre el operador [] de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda;

es decir, con [-1] accederíamos al último elemento de la lista, con [-2] al penúltimo, con [-3] , al antepenúltimo, y así sucesivamente.

Otra cosa inusual es lo que en Python se conoce como **slicing o particionado**, y que consiste en ampliar este mecanismo para permitir seleccionar porciones de la lista. Si en lugar de un número escribimos dos números inicio y fin separados por dos puntos **(inicio:fin)** Python interpretará que queremos una lista que vaya desde la posición inicio a la posición fin , sin incluir este último. Si escribimos tres números **(inicio:fin:salto)** en lugar de dos, el tercero se utiliza para determinar cada cuantas posiciones añadir un elemento a la lista.

```
lista = [99, True, "una lista", [1, 2]]
mi_var = lista[0:2] #  mi_var vale [99, True]
mi_var = lista[0:4:2] # mi_var vale [99, "una lista"]
```

Los números negativos también se pueden utilizar en un **slicing**, con el mismo comportamiento que se comentó anteriormente.

Hay que mencionar así mismo que no es necesario indicar el principio y el final del **slicing**, sino que, si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista, respectivamente.

```
lista = [99, True, "una lista"]
mi_var = lista[1:]  #   mi_var vale [True, "una lista"]
mi_var = lista[:2]  #  mi_var vale [99, True]
mi_var = lista[:]  #  mi_var vale [99, True, "una lista"]
mi_var = lista[::2]  #   mi_var vale [99, "una lista"]
```

También podemos utilizar este mecanismo para modificar la lista.

```
lista= [99, True, "una lista", [1, 2]]
lista[0:2] = [0, 1]  #  lista vale [0, 1, "una lista", [1, 2]]
```

pudiendo incluso modificar el tamaño de la lista si la lista de la parte derecha de la asignación tiene un tamaño menor o mayor que el de la selección de la parte izquierda de la asignación:.

```
lista[0:2] = [False]  #  lista vale [False, "una lista", [1, 2]]
```

Tuplas

Todo lo que hemos explicado sobre las listas se aplica también a las tuplas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

```
tupla = (1, 2, True, "python")
```

type(tupla) # salida: type “tuple”

Además hay que tener en cuenta que es necesario añadir una coma para tuplas de un solo elemento, para diferenciarlo de un elemento entre paréntesis.
Tupla = (1,)
type(tupla) # salida: type “tuple”

Para referirnos a elementos de una **tupla**, como en una **lista**, se usa el **operador []**
tupla = (1, 2, True, “python”)
mi_var = tupla[0] # mi_var es 1
mi_var = tupla[0:2] # mi_var es (1, 2)

Podemos utilizar el **operador []** debido a que las **tuplas**, al igual que las **listas**, forman parte de un tipo de **objetos llamados secuencias**. Permitirme un pequeño inciso para indicaros que las **cadenas de texto** también son **secuencias**, por lo que no os extrañará que podamos hacer cosas como estas:
cadena = “hola mundo”
print cadena[0] # salida: h
print cadena[5:] # salida: mundo
print cadena[::3] # salida: hauo

Volviendo al tema de las **tuplas**, su diferencia con las **listas** estriba en que las **tuplas** no poseen estos mecanismos de **modificación** a través de funciones tan útiles de los que hablábamos al final de la anterior sección. Además son inmutables, es decir, **sus valores no se pueden modificar una vez creada**; y tienen un tamaño fijo.

Diccionarios

Los **diccionarios**, también llamados **matrices asociativas**, deben su nombre a que son colecciones que relacionan una clave y un valor.
diccionario = {“Love Actually “: “Richard Curtis”, “Kill Bill”: “Tarantino”,
“Amélie”: “Jean-Pierre Jeunet”}

El primer valor se trata de la **clave** y el segundo del **valor** asociado a la **clave**. Como **clave** podemos utilizar cualquier **valor inmutable**: podríamos usar **números, cadenas, booleanos, tuplas**, ... pero no **listas o diccionarios**, dado que son **mutables.** 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.

La diferencia principal entre los **diccionarios** y las **listas** o las **tuplas** es que a los **valores** almacenados en un **diccionario** se les accede no por su **índice**, porque de hecho no tienen orden, sino por su **clave**, utilizando de nuevo el operador [] .
diccionario["Love Actually "] # devuelve "Richard Curtis"

Al igual que en **listas** y **tuplas** también se puede utilizar este operador para reasignar valores.
diccionario["Kill Bill"] = "Quentin Tarantino"

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los **diccionarios** no son **secuencias**, si no **mappings** (mapeados, asociaciones).

CONTROL DE FLUJO

Sentencias condicionales

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición.

Aquí es donde cobran su importancia el tipo **booleano** y los **operadores lógicos y relacionales** que aprendimos en el capítulo sobre los tipos básicos de Python.

If
La forma más simple de un estamento condicional es un if (del inglés si) seguido de la condición a evaluar, dos puntos (:) y en la siguiente línea e indentado, el código a ejecutar en caso de que se cumpla dicha condición.

```
fav = "mundogeek.net"
if fav == "mundogeek.net":
        print "Tienes buen gusto!"
        print "Gracias"
```

if ... else
Vamos a ver ahora un condicional algo más complicado. ¿Qué haríamos si

quisiéramos que se ejecutaran unas ciertas órdenes en el caso de que la condición no se cumpliera? Sin duda podríamos añadir otro **if** que tuviera como condición la negación del primero:

```
if fav == “mundogeek.net”:
        print “Tienes buen gusto!”
        print “Gracias”
if fav != “mundogeek.net”:
        print “Vaya, que lástima”
```

pero el condicional tiene una segunda construcción mucho más útil.

```
if fav == “mundogeek.net”:
        print “Tienes buen gusto!”
        print “Gracias”
else:
        print “Vaya, que lástima”
```

if ... elif ... elif ... else

Todavía queda una construcción más que ver, que es la que hace uso del <**elif**>

```
numero = int(raw_input('Ingresa un numero'))
if numero < 0:
        print “Negativo”
elif numero > 0:
        print “Positivo”
else:
        print “Cero”
```

A if C else B

También existe una construcción similar al operador **if – else**

```
num = int(raw_input('Introduce un numero: '))
var = 'par' if (num % 2 == 0) else 'impar'
print var
```

Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

While

El bucle **while** (mientras) ejecuta un fragmento de código mientras se cumpla

una condición.

```
edad = 0
while edad < 18:
        edad = edad + 1
        print 'Felicidades, tienes ' + str(edad)
```

Bucle infinito

Sin embargo hay situaciones en las que un bucle infinito es útil. Por ejemplo, veamos un pequeño programa que repite todo lo que el usuario diga hasta que escriba adios.

```
while True:
        entrada = raw_input('> (adios para salir ')
        if entrada == 'adios':
                break
        else:
                print entrada
```

bucle infinito programa alternativo

```
salir = False
while not salir:
        entrada = raw_input(> (adios para salir): ')
        if entrada == 'adios':
                salir = True
        else:
                print entrada
```

Otra palabra clave que nos podemos encontrar dentro de los bucles es **continue** (continuar).

```
edad = 0
while edad < 18:
        edad = edad + 1
        if edad % 2 == 0:
                continue
        print 'Felicidades, tienes ' + str(edad)
```

for .. in

En Python **for** se utiliza como una forma genérica de iterar sobre una secuencia. Y como tal intenta facilitar su uso para este fin.

```
secuencia = [“uno”, “dos”, “tres”]
for elemento in secuencia:
        print elemento
```

FUNCIONES

Una **función** es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar **procedimientos**. En Python no existen los **procedimientos**, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor **None** (nada), equivalente al null de Java.

En Python las funciones se declaran de la siguiente forma:

```
def mi_funcion(param1, param2):
    print param1
    print param2

# activa la funcion
mi_funcion('hola', 2)
```

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de **docstring** (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):
    '''Esta funcion imprime los dos valores pasados
    como parametros'''
    print param1
    print param2
```

También es posible, no obstante, definir funciones con un número variable de argumentos, o bien asignar valores por defecto a los parámetros para el caso de que no se indique ningún valor para ese parámetro al llamar a la función.

Los valores por defecto para los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto.

```
def imprimir(texto, veces = 1):
    print veces * texto

# activa la funcion
imprimir('Tigre')
imprimir('Oso', 2)
```

Para definir funciones con un número variable de argumentos colocamos un último parámetro para la función cuyo nombre debe precederse de un signo *

```
def varios(param1, param2, *otros):
for val in otros:
        print val

# activa la funcion
varios(1, 2)
varios(1, 2, 3)
varios(1, 2, 3, 4)
```

También se puede preceder el nombre del último parámetro con **, en cuyo caso en lugar de una **tupla** se utilizaría un **diccionario**. Las **claves** de este **diccionario** serían los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros.

En el siguiente ejemplo se utiliza la función items de los diccionarios, que devuelve una lista con sus elementos, para imprimir los parámetros que contiene el diccionario.

```
def varios(param1, param2, **otros):
        for i in otros.items():
        print i

# activa funcion
varios(1, 2, tercero = 3)
```

Veamos por último cómo devolver valores, para lo que se utiliza la palabra clave return.

```
def sumar(x, y):
        return x + y

# activa funcion
print sumar(3, 2)
```

También podríamos pasar varios valores que retornar a return.

Sin embargo esto no quiere decir que las funciones Python puedan devolver varios valores, lo que ocurre en realidad es que Python crea una tupla al vuelo cuyos elementos son los valores a retornar, y esta única variable es la que se devuelve.

```
def f(x, y):
        return x * 2, y * 2
```

 # activa funcion
a, b = f(1, 2)

ORIENTACION A OBJETOS

OBJETOS

Un objeto es “una cosa”. Y, si una cosa es un sustantivo, entonces un objeto es un sustantivo.

Cuando pensamos en “objetos”, todos los sustantivos son objetos.

Describir un objeto, es simplemente mencionar sus cualidades. Las cualidades son adjetivos. Podemos decir que un adjetivo es una cualidad del sustantivo.

OBJETO (sustantivo)	ATRIBUTO (locución adjetiva)	CUALIDAD DEL ATRIBUTO (adjetivo)
	(es de) COLOR	VERDE
El OBJETO: VEHICULO	(es de) TAMAÑO	GRANDE
	(es de) ASPECTO	FEO

Cuando en el documento...	En la programación se denomina...	Y con respecto a la programación orientada a objetos es...
Hablamos de “objeto”	OBJETO	Un elemento
Hablamos de “atributos” (o cualidades)	PROPIEDADES	Un elemento
Hablamos de “acciones” que puede realizar el objeto	METODOS	Un elemento
Hablamos de “atributos-objeto”	COMPOSICION	Una tecnica
Vemos que los objetos relacionados entre sí, tienen nombres de atributos iguales (por ejemplo: color y tamaño) y sin embargo, pueden tener valores diferentes	POLIMORFISMO	Una caracteristica
Hablamos de objetos que son sub-tipos (o ampliación) de otros	HERENCIA	Una caracteristica

Paradigma: teoría cuyo núcleo central [...] suministra la base y modelo para resolver problemas [...] Definición de la Real Academia Española, vigésima tercera edición

La Programación Orientada a Objetos (**POO u OOP** según sus siglas en inglés) es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se modelan a través de **clases** y **objetos**, y en el que nuestro programa consiste en una serie de interacciones entre estos **objetos**.

Cómo tal, nos enseña un método -probado y estudiado- el cual se basa en las interacciones de objetos (todo lo descrito en el título anterior, "Pensar en objetos") para resolver las necesidades de un sistema informático. Básicamente, este **paradigma** se compone de **6 elementos y 7 características** que veremos a continuación.

Los **elementos** de la POO, pueden entenderse como los **materiales** que necesitamos para diseñar y programar un sistema, mientras que las **características**, podrían asumirse como las **herramientas** de las cuáles disponemos para construir el sistema con esos **materiales.**

Elementos:

- **Clases:** Las clases son los modelos sobre los cuáles se construirán nuestros objetos
- **Propiedades:** Las propiedades son las características intrínsecas del objeto.
- **Metodos:** Los **métodos** son **funciones** solo que técnicamente se denominan **métodos**, y representan acciones propias que puede realizar el objeto (y no otro)
- **Objeto:** Las **clases** por sí mismas, no son más que modelos que nos servirán para crear **objetos** en concreto. Podemos decir que una **clase** es el razonamiento abstracto de un **objeto**, mientras que el **objeto**, es su materialización. A la acción de crear **objetos**, se la denomina **instanciar una clase** y dicha instancia, consiste en asignar la **clase**, como valor a una variable

Caracteristica:

- **Herencia:** característica principal de la POO . Como comentamos

anteriormente, algunos **objetos** comparten las mismas **propiedades** y **métodos** que otro **objeto**, y además agregan nuevas **propiedades y métodos**. A esto se lo denomina **herencia**: *una clase que hereda de otra*. Vale aclarar, que en Python, cuando una clase no hereda de ninguna otra, debe hacerse heredar de **object**, que es la **clase principal de Python**, que define un **objeto**.

Clases y Objetos

Para entender este paradigma primero tenemos que comprender qué es una **clase** y qué es un **objeto**. Un **objeto** es una entidad que agrupa un **estado y una funcionalidad relacionadas**. El **estado del objeto** se define a través de variables llamadas **atributos**, mientras que la **funcionalidad** se modela a través de **funciones** a las que se les conoce con el nombre de **métodos del objeto.**

Un ejemplo de **objeto** podría ser un **automovil**, en el que tendríamos **atributos** como la marca, el número de puertas o el tipo de carburante y **métodos** como arrancar y parar. O bien cualquier otra combinación de **atributos y métodos** según lo que fuera relevante para nuestro programa.

Una **clase**, por otro lado, no es más que una **plantilla genérica** a partir de la cuál **instanciar los objetos**; **plantilla** que es la que define qué **atributos y métodos** tendrán los **objetos de esa clase.**

Volviendo a nuestro ejemplo: en el mundo real existe un **conjunto de objetos** a los que llamamos **automoviles** y que tienen un conjunto de **atributos comunes** y un **comportamiento común**, esto es a lo que llamamos **clase**. Sin embargo, mi **automovil** no es igual que el **automovil** de mi vecino, y aunque pertenecen a la misma **clase de objetos, son objetos distintos**.

```
class Coche:
    '''Abstraccion de los objetos coche.'''
    def __init__(self, gasolina):
        self.gasolina = gasolina
        print “Tenemos”, gasolina, “litros”

    def arrancar(self):
        if self.gasolina > 0:
            print “Arranca”
        else:
            print “No arranca”
```

```
    def conducir(self):
        if self.gasolina > 0:
            self.gasolina -= 1
            print "Quedan", self.gasolina, "litros"
        else:
            print "No se mueve"

mi_coche = Coche(3)

mi_coche.gasolina  #  salida: 3
mi_coche.arrancar()  #  salida: Arranca
mi_coche.conducir()  #  salida: Quedan 2 litros
mi_coche.conducir()  #  salida: Quedan 1 litros
mi_coche.conducir()  #  salida: Quedan 0 litros
mi_coche.conducir()  #  salida: No se mueve
mi_coche.arrancar()  #  salida: No arranca
mi_coche.gasolina  # salida: 0
```

Como último apunte recordar que en Python, como ya se comentó en repetidas ocasiones anteriormente, todo son **objetos**. Las **cadenas**, por ejemplo, tienen métodos como **upper()** , que devuelve el texto en mayúsculas o **count(sub)** , que devuelve el número de veces que se encontró la cadena sub en el texto.

Herencia

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el **encapsulamiento**, la **herencia** y el **polimorfismo**.

En un lenguaje orientado a objetos cuando hacemos que una **clase (subclase) herede** de otra **clase (superclase)** estamos haciendo que la **subclase** contenga todos los **atributos y métodos** que tenía la **superclase**. No obstante al acto de **heredar** de una **clase** también se le llama a menudo **extender una clase**.

```
class Instrumento:
    def __init__(self, precio):
        self.precio = precio

    def tocar(self):
        print 'Estamos tocando musica'
```

```
    def romper(self):
        print 'Eso lo pagas tu'
        print "Son", self.precio, '$$$'

class Percusion(Instrumento):
    pass

class Cuerdas(Instrumento):
    pass

## creando objeto
guitarra = Cuerdas(2000)
violin = Cuerdas(3000)
bateria = Percusion(1000)

## output
guitarra.tocar()
violin.romper()
```

Herencia múltiple
En Python, a diferencia de otros lenguajes como Java o C#, se permite la **herencia múltiple**, es decir, una **clase** puede heredar de varias **clases** a la vez. Por ejemplo, podríamos tener una **clase Cocodrilo** que heredara de la **clase Terrestre** , con métodos como caminar() y atributos como velocidad_caminar y de la **clase Acuatico** , con métodos como nadar() y atributos como velocidad_nadar . Basta con enumerar las clases de las que se hereda separándolas por comas:

```
class Cocodrilo(Terrestre, Acuatico):
    pass
```

En el caso de que alguna de las **clases padre** tuvieran **métodos** con el mismo nombre y número de parámetros las **clases** sobreescribirían la implementación de los **métodos de las clases más a su derecha en la definición.**

Polimorfismo
La palabra **polimorfismo**, del griego **poly morphos** (varias formas), se refiere a la habilidad de **objetos** de distintas **clases** de responder al mismo mensaje. Esto se puede conseguir a través de la herencia: un **objeto** de una **clase derivada** es al mismo tiempo un **objeto** de la **clase padre**, de forma que allí donde se requiere un **objeto** de la **clase padre** también se puede utilizar uno de la **clase hija**.

Encapsulación
La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos **guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada**, en caso contrario es **pública**. Los métodos cuyo nombre **comienza y termina con *dos guiones bajos son métodos especiales* que Python** llama automáticamente bajo ciertas circunstancias,

```
class Ejemplo:
        def  publico(self):
                print 'Publico'
        def  __privado(self):
                print 'Privado'

ej = Ejemplo()
ej.publico()
ej.__privado()
## acceder a el mediante una pequena trampa
ej._Ejemplo__privado()
```

En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro **objeto**, pero que este se produzca de forma controlada. Para esto podemos escribir **métodos** cuyo único cometido sea este, **métodos** que normalmente, por convención, tienen nombres como **getVariable** y **setVariable** ; de ahí que se conozcan también con el nombre de **getters y setters**.

```
class Fecha(object):
   def __init__(self):
      self.__dia = 1
   def getDia(self):
      return self.__dia
   def setDia(self, dia):
      if dia > 0 and dia < 31:
         self.__dia = dia
         print 'Total de dias: ', self.__dia
      else:
         print 'Error'
```

```
    dia = property(getDia, setDia)

## input
cantidad_dias = int(raw_input('Cantidad de dias: '))

## crea objeto <mi_fecha>
mi_fecha = Fecha()

## output
mi_fecha.setDia(cantidad_dias)
```

Clases de 'nuevo-estilo'
En el ejemplo anterior os habrá llamado la atención el hecho de que la clase **Fecha** derive de **object** . La razón de esto es que para poder usar propiedades la **clase** tiene que ser de **nuevo-estilo**, clases enriquecidas introducidas en Python 2.2 que serán el estándar en Python 3.0 pero que aún conviven con las **clases clásicas** por razones de retrocompatibilidad. Además de las propiedades las **clases de nuevo estilo** añaden otras funcionalidades como **descriptores** o **métodos estáticos**.

Para que una **clase sea de nuevo estilo** es necesario, por ahora, que extienda una **clase de nuevo-estilo**. En el caso de que no sea necesario heredar el comportamiento o el estado de ninguna clase, como en nuestro ejemplo anterior, se puede heredar de **object** , **que es un objeto vacio que sirve como base para todas las clases de nuevo estilo**.

La diferencia principal entre las clases antiguas y las de nuevo estilo consiste en que a la hora de crear una nueva clase anteriormente no se definía realmente un nuevo tipo, sino que todos los objetos creados a partir de clases, fueran estas las clases que fueran, eran de tipo instance.

Métodos especiales
A continuación se listan algunos especialmente útiles.

__init__(self, args)
Método llamado después de crear el objeto para realizar tareas de inicialización.

__new__(cls, args)
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que __init__ y que se encarga de construir y devolver el objeto en sí. Es equivalente

a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es self , sino la propia clase: cls .

__del__(self)
Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

__str__(self)
Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos print para mostrar nuestro objeto o cuando usamos la función str(obj) para crear una cadena a partir de nuestro objeto.

__cmp__(self, otro)
Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores < , <= , > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

__len__(self)
Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función len(obj) sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

Existen bastantes más métodos especiales, que permite entre otras cosas utilizar el mecanismo de slicing sobre nuestro objeto, utilizar los operadores aritméticos o usar la sintaxis de diccionarios, pero un estudio exhaustivo de todos los métodos queda fuera del propósito del capítulo.

Revisitando Objetos

Diccionarios

diccionario.get(k[, d])
Busca el valor de la **clave k** en el diccionario. Es equivalente a utilizar D[k] pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis D[k] , de no existir la clave se lanzaría una excepción.

diccionario.has_key(k)
Comprueba si el diccionario tiene la clave k . Es equivalente a la sintaxis k in D .

diccionario.items()
Devuelve una lista de tuplas con pares clave-valor.

diccionario.keys()
Devuelve una lista de las claves del diccionario.

diccionario.pop(k[, d])
Borra la clave k del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve d si se especificó el parámetro o bien se lanza una excepción.

diccionario.values()
Devuelve una lista de los valores del diccionario.

Cadenas

cadena.count(sub[, start[, end]])
Devuelve el número de veces que se encuentra sub en la cadena. Los parámetros opcionales start y end definen una subcadena en la que buscar.

cadena.find(sub[, start[, end]])
Devuelve la posición en la que se encontró por primera vez sub en la cadena o -1 si no se encontró.

cadena.join(sequence)
Devuelve una cadena resultante de concatenar las cadenas de la secuencia **sequence** separadas por la cadena sobre la que se llama el método.

cadena.partition(sep)
Busca el separador **sep** en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.

cadena.replace(old, new[, count])
Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena **old** por la cadena **new** . Si se especifica el parámetro **count** , este indica el número máximo de ocurrencias a reemplazar.

cadena.split([sep [,maxsplit]])
Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividirlas por el delimitador **sep** . En el caso de que no se especifique **sep**, se usan espacios. Si se especifica **maxsplit** , este indica el número máximo de particiones a realizar.

Listas

lista.append(object)
Añade un objeto al final de la lista.

lista.count(value)
Devuelve el número de veces que se encontró value en la lista.

lista.extend(iterable)
Añade los elementos del iterable a la lista.

lista.index(value[, start[, stop]])
Devuelve la posición en la que se encontró la primera ocurrencia de **value** . Si se especifican, **start y stop** definen las posiciones de inicio y fin de una sublista en la que buscar.

lista.insert(index, object)
Inserta el **objeto object** en la posición **index**.

lista.pop([index])
Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

lista.remove(value)
Eliminar la primera ocurrencia de value en la lista.

lista.reverse()
Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

lista.sort(cmp=None, key=None, reverse=False)
Ordena la lista. Si se especifica **cmp** , este debe ser una función que tome como parámetro dos valores x e y de la lista y devuelva -1 si x es menor que y , 0 si son iguales y 1 si x es mayor que y.

El parámetro reverse es un booleano que indica si se debe ordenar la lista de

forma inversa, lo que sería equivalente a llamar primero a **lista.sort() y después a lista.reverse()**.

Por último, si se especifica, el parámetro **key** debe ser una función que tome un elemento de la **lista** y devuelva una **clave** a utilizar a la hora de comparar, en lugar del elemento en si.

Las definiciones de **clase** pueden aparecer en cualquier lugar de un programa, pero normalmente estan al principio (tras las sentencias **import**). Las sintacticas de la definicion de clases son las mismas que para cualesquiera otras sentencias compuestas.

Esta definicion crea una nueva **clase** llamada **Punto**. La sentencia **pass** no tiene efectos; solo es necesaria porque una sentencia compuesta debe tener algo en su cuerpo.

Al crear la **clase Punto** hemos creado un nuevo **tipo**, que tambien se llama **Punto**. Los miembros de este **tipo** se llaman **instancias del tipo u objetos**. La creacion de una nueva **instancia** se llama **instanciacion**. Para **instanciar** un **objeto Punto** ejecutamos una funcion que se llama (lo ha adivinado) Punto:
blanco = Punto()

A la **variable blanco** se le asigna una referencia a un nuevo **objeto Punto**. A una funcion como **Punto** que crea un **objeto** nuevo se le llama **constructor**.

Atributos

Podemos agregar **nuevos datos** a una **instancia** utilizando la notacion de **punto**:
blanco.x = 3.0
blanco.y = 4.0

Esta sintaxis es similar a la sintaxis para seleccionar una variable de un modulo, como **math.pi** o **string.uppercase**. En este caso, sin embargo, estamos seleccionando un dato de una instancia. Estos ıtems con nombre se llaman **atributos**.

Output
print blanco.x
abcisa = blanco.y
print abcisa
print '('+ str(blanco.x) + ', '+ str(blanco.y) + ')'

Instancias como parametro

Puede usted pasar una **instancia** como **parametro** de la forma habitual. Por ejemplo:

```
def imprimePunto(p):
print  '('.+ str(p.x) + ', ''+ str(p.y) + ')'
```

imprimePunto acepta un punto como argumento y lo muestra en formato est´andar. Si llama a imprimePunto(blanco), el resultado es (3.0, 4.0).

Mismidad

El significado de la palabra **mismo** parece totalmente claro hasta que uno se para un poco a pensarlo, y entonces se da cuenta de que hay algo mas de lo que suponıa.

Por ejemplo, si dice **Pepe y yo tenemos la misma moto**, lo que quiere decir es que su moto y la de usted son de la misma marca y modelo, pero que son dos motos distintas. Si dice **Pepe y yo tenemos la misma madre**, quiere decir que su madre y la de usted son la misma persona. Ası que la idea de **identidad** es diferente segun el contexto.

Cuando habla de **objetos**, hay una ambig uedad parecida. Por ejemplo, si dos Puntos son el mismo, ¿significa que contienen los mismos datos (coordenadas) o que son de verdad el mismo **objeto**?

Para averiguar si dos referencias se refieren al mismo objeto, utilice el operador **==**.

```
p1 = Punto()
p1.x = 3
p1.y = 4
p2 = Punto()
p2.x = 3
p2.y = 4
print p1 == p2 ## salida 0
```

Aunque p1 y p2 contienen las mismas coordenadas, no son el mismo objeto.

```
p2 = p1
print p1 == p2 ## salida 1
```

Si asignamos p1 a p2, las dos variables son alias del mismo **objeto**. Este tipo de igualdad se llama **igualdad superficial** porque solo compara las referencias, pero no el contenido de los objetos.

Para comparar los contenidos de los objetos (**igualdad profunda**) podemos escribir una **funcion** llamada **mismoPunto**:

```
def mismoPunto(p1, p2) :
return (p1.x == p2.x) and (p1.y == p2.y)
p1 = Punto()
p1.x = 3
p1.y = 4
p2 = Punto()
p2.x = 3
p2.y = 4
print mismoPunto(p1, p2) ## salida: 1
```

Rectangulos

Digamos que queremos una clase que represente un rectángulo. La pregunta es, ¿que información tenemos que proporcionar para definir un rectángulo? Para simplificar las cosas, supongamos que el rectángulo esta orientado vertical u horizontalmente, nunca en diagonal. Tenemos varias posibilidades: podemos señalar el centro del rectángulo (dos coordenadas) y su tamaño (anchura y altura); o podemos señalar una de las esquinas y el tamaño; o podemos señalar dos esquinas opuestas. Un modo convencional es señalar la esquina inferior izquierda del rectángulo y el tamaño.

De nuevo, definiremos una nueva clase:

```
class Rectangulo(object):
    pass
class Punto(object):
    pass
def encuentraCentro(caja):
    p = Punto()
    p.x = caja.esquina.x + caja.anchura/2.0
    p.y = caja.esquina.y + caja.altura/2.0
    return p
def mueveRectangulo(caja, dx, dy):
    caja.esquina.x = caja.esquina.x + dx
    caja.esquina.y = caja.esquina.y + dy
def imprimePunto(p):
    print '(' + str(p.x) + ', ' + str(p.y) + ')'
def agrandaRect(caja, danchura, daltura):
    caja.anchura = caja.anchura + danchura
    caja.altura = caja.altura + daltura

## instanciacion del objeto <caja> y <caja.esquina>
caja = Rectangulo()
```

```
caja.anchura = 100.0
caja.altura = 200.0
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;

caja_1 = Rectangulo()
caja_1.anchura = 100.0
caja_1.altura = 200.0
caja_1.esquina = Punto()
caja_1.esquina.x = 0.0;
caja_1.esquina.y = 0.0;

## output
print 'Ancho  del rectangulo: ', caja.anchura
print 'Altura del rectangulo: ', caja.altura
print
centro = encuentraCentro(caja)
print 'Coordenadas del centro del rectangulo caja:   ',
imprimePunto(centro)
agrandaRect(caja_1, 50, 100)
centro_1 = encuentraCentro(caja_1)
print 'Coordenadas del centro del rectangulo caja_1: ',
imprimePunto(centro_1)
mueveRectangulo(caja, 40, 40)
print 'Caja con <x> movida: ', caja.esquina.x
print 'Caja con <y> movida: ', caja.esquina.y
centro = encuentraCentro(caja)
print 'Coordenadas del centro del rectangulo caja:   ',
imprimePunto(centro)
```

Version anterior modificada con deepcopy

```
## modificaciones sobre object_12.py
## se modifica: agrandaRectangulo y mueveRectangulo
## utilizamos: copy y deepcopy del modulo copy

class Rectangulo(object):
    pass
class Punto(object):
    pass
```

```
def encuentraCentro(caja):
    p = Punto()
    p.x = caja.esquina.x + caja.anchura/2.0
    p.y = caja.esquina.y + caja.altura/2.0
    return p
def mueveRectangulo(caja, dx, dy):
    import copy
    nuevaCaja = copy.deepcopy(caja)
    nuevaCaja.esquina.x = nuevaCaja.esquina.x + dx
    nuevaCaja.esquina.y = nuevaCaja.esquina.y + dy
    return nuevaCaja
def imprimePunto(p):
    print '(' + str(p.x) + ', ' + str(p.y) + ')'
def agrandaRectangulo(caja, danchura, daltura):
    import copy
    nuevaCaja = copy.deepcopy(caja)
    nuevaCaja.anchura = nuevaCaja.anchura + danchura
    nuevaCaja.altura = nuevaCaja.altura + daltura
    return nuevaCaja

## instanciacion del objeto <caja> y <caja.esquina>
caja = Rectangulo()
caja.anchura = 200.0
caja.altura = 300.0
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;

caja_1 = Rectangulo()
caja_1.anchura = 300.0
caja_1.altura = 400.0
caja_1.esquina = Punto()
caja_1.esquina.x = 5.0;
caja_1.esquina.y = 5.0;

## output
print 'Ancho caja:     ', caja.anchura
print 'Altura caja:    ', caja.altura
print 'Abcisa caja:    ', caja.esquina.x
print 'Ordenada caja:  ', caja.esquina.y
centro = encuentraCentro(caja)
print 'Centro de caja:',
```

```
imprimePunto(centro)
print
print 'Ancho caja_1:     ', caja_1.anchura
print 'Altura caja_1:    ', caja_1.altura
print 'Abcisa caja_1:    ', caja_1.esquina.x
print 'Ordenada caja_1:  ', caja_1.esquina.y
centro_1 = encuentraCentro(caja_1)
print 'Centro de caja_1:',
imprimePunto(centro_1)
print
caja_2 = agrandaRectangulo(caja_1, 500, 700)
print 'Ancho caja_2:    ', caja_2.anchura
print 'Altura caja_2:   ', caja_2.altura
print 'Abcisa caja_2:   ',caja_2.esquina.x
print 'Ordenada caja_2: ',caja_2.esquina.y
centro_2 = encuentraCentro(caja_2)
print 'Centro caja_2:  ',
imprimePunto(centro_2)
print
caja_3 = mueveRectangulo(caja, 40, 40)
print 'Caja con <x> movida:    ', caja_3.esquina.x
print 'Caja con <y> movida:    ', caja_3.esquina.y
centro_3 = encuentraCentro(caja_3)
print 'Centro de caja movida: ',
imprimePunto(centro_3)
```

Clases y funciones

Como otro ejemplo de un tipo definido por el usuario, definiremos una **clase** llamada **Hora** que registra la hora del día. La definición de la clase es como sigue:

```
class Hora(object):
        pass
def imprimeHora(hora, min, seg):
        print hora, ':', min, ':', seg
def comparaHorarios(t1, t2):
   if t1 < t2:
      return False
   else:
      return True
```

Podemos crear un nuevo **objeto Hora** y asignar: horas, minutos y segundos

```
hora_newyork = Hora()
```

```
hora_newyork.horas = 11
hora_newyork.minutos = 59
hora_newyork.segundos = 30
```
Tambien podemos crear otro **objeto Hora**
```
hora_barcelona = Hora()
```
Output
```
print 'Horario de Barcelona: ',
hora_barcelona = imprimeHora(16, 45, 37)
print 'Horario New York:  ',  hora_newyork.horas, ':',
print hora_newyork.minutos, ':',
print hora_newyork.segundos
barcelona_newyork = comparaHorarios(hora_barcelona, hora_newyork)
print barcelona_newyork
```

PROGRAMACION FUNCIONAL

Funciones puras

función pura: Una función que no modifica ninguno de los objetos que recibe como parámetros. La mayoría de las funciones puras son rentables.

Estilo funcional de programación: Un estilo de programación en el que la mayoría de las funciones son puras.

La función crea un nuevo **objeto Hora**, inicializa sus atributos y devuelve una referencia al **nuevo objeto**. A esto se le llama **función pura** porque no modifica ninguno de los objetos que se le pasan y no tiene efectos laterales, como mostrar un valor o tomar una entrada del usuario.

Aquí tiene un ejemplo de cómo usar esta **función**. Crearemos dos objetos **Hora: horaActual**, que contiene la hora actual, y **horaPan**, que contiene la cantidad de tiempo que necesita un panadero para hacer pan. Luego usaremos **sumaHora** para averiguar cuándo estará hecho el pan.
```
class Hora(object):
        pass

def sumaHora(t1, t2):
        suma = Hora()
        suma.horas = t1.horas + t2.horas
        suma.minutos = t1.minutos + t2.minutos
        suma.segundos = t1.segundos + t2.segundos
        if suma.segundos >= 60:
```

```
            suma.segundos = suma.segundos - 60
            suma.minutos = suma.minutos + 1
    if suma.minutos >= 60:
            suma.minutos = suma.minutos - 60
            suma.horas = suma.horas
    return suma

def imprimeHora(hora):
    print str(hora.horas) + ":" +
    str(hora.minutos) + ":" +
    str(hora.segundos)

##  creamos dos objetos: horaActual y horaPan
horaActual = Hora()
horaActual.horas = 9
horaActual.minutos = 14
horaActual.segundos = 30
horaPan = Hora()
horaPan.horas = 3
horaPan.minutos = 35
horaPan.segundos = 0

## output
horaHecho = sumaHora(horaActual, horaPan)
imprimeHora(horaHecho)
```

Modificadores

Hay veces en las que es útil que una **función modifique** uno o más de los **objetos** que recibe como parámetros. Normalmente, el llamante conserva una referencia a los objetos que pasa, así que cualquier cambio que la **función** haga será visible para el llamante. Las **funciones** que trabajan así se llaman **modificadores**. Incremento, que añade un número dado de segundos a un **objeto Hora**, se escribiría de forma natural como un **modificador**. Un esbozo rápido de la función podría ser éste:

```
def incremento(hora, segundos):
    if hora.segundos >= 60:
            hora.segundos = hora.segundos - 60
            hora.minutos = hora.minutos + 1
    if hora.minutos >= 60:
            hora.minutos = hora.minutos - 60
            hora.horas = hora.horas + 1
```

Una solución es sustituir las sentencias **if** por sentencias **while**:

```
def incremento(hora, segundos):
    while hora.segundos >= 60:
        hora.segundos = hora.segundos - 60
        hora.minutos = hora.minutos + 1
    while hora.minutos >= 60:
        hora.minutos = hora.minutos - 60
        hora.horas = hora.horas + 1
```

Todo lo que se pueda hacer con **modificadores** puede hacerse también con **funciones puras**. En realidad, algunos lenguajes de programación sólo permiten funciones puras. Hay ciertas evidencias de que los **programas** que usan **funciones puras son más rápidos de desarrollar y menos propensos a los errores** que los programas que usan **modificadores**. Sin embargo, a veces los modificadores son útiles, y en algunos casos los **programas funcionales son menos eficientes**.

En general, recomendamos que escriba **funciones puras siempre que sea razonable hacerlo así** y recurra a los **modificadores** sólo si hay una ventaja convincente. Este enfoque podría llamarse e**stilo funcional de programación.**

Desarrollo de prototipos frente a planificación
En este capítulo mostramos una aproximación al desarrollo de programas a la que llamamos **desarrollo de prototipos**. En cada caso, escribimos un esbozo basto (o prototipo) que realizaba el cálculo básico y luego lo probamos sobre unos cuantos casos, corrigiendo los fallos tal como los encontrábamos.
Aunque este enfoque puede ser efectivo, puede conducirnos a código que es innecesariamente complicado, ya que trata con muchos casos especiales, y poco fiable, porque es difícil saber si encontró todos los errores.
Una alternativa es el **desarrollo planificado**, en el que una comprensión del problema en profundidad puede hacer la programación mucho más fácil. En este caso, el enfoque es que un **objeto Hora** es en realidad ¡un número de tres dígitos en base 60! El componente segundo es la ***columna de unidades***, el componente minuto es la ***columna de las sesententas*** y el componente hora es la ***columna de las tresmilseiscentenas***.

Cuando escribimos sumaHora e incremento, en realidad estábamos haciendo una suma en base 60, que es por lo que debíamos acarrear de una columna a la siguiente. Esta observación sugiere otro enfoque para el problema. Podemos convertir un **objeto Hora** en un simple número y sacar provecho del hecho de que la máquina sabe la aritmética necesaria. La siguiente función convierte un **objeto Hora en un entero**:

```
def convierteASegundos(t):
    minutos = t.horas * 60 + t.minutos
    segundos = minutos * 60 + t.segundos
    return segundos
```

Ahora, sólo necesitamos una forma de convertir un **entero en un objeto Hora**:

```
def haceHora(segundos):
    hora = Hora()
    hora.horas = segundos/3600
    segundos = segundos - hora.horas * 3600
    hora.minutos = segundos/60
    segundos = segundos - hora.minutos * 60
    hora.segundos = segundos
    return hora
```

puede usar estas funciones para reescribir sumaHora:

```
def sumaHora(t1, t2):
    segundos = convierteASegundos(t1) + convierteASegundos(t2)
    return haceHora(segundos)
```

Generalización

De algún modo, convertir de base 60 a base 10 y de vuelta es más difícil que simplemente manejarse con las horas. La conversión de base es más abstracta; nuestra intuición para tratar con las horas es mejor.

Pero si tenemos la comprensión para tratar las horas como números en base 60, y hacer la inversión de escribir las funciones de conversión (**convierteASegundos y haceHora**), obtenemos un programa que es más corto, más fácil de leer y depurar y más fiable.

También es más fácil añadir funcionalidades más tarde. Por ejemplo, imagine restar dos Horas para hallar el intervalo entre ellas. La aproximación ingenua sería implementar la resta con acarreo. Con el uso de las funciones de conversión será más fácil y con mayor probabilidad, correcto. Irónicamente, a veces hacer un poblema más complejo (o más general) lo hace más fácil (porque hay menos casos especiales y menos oportunidades de error).

Algoritmos

Cuando escribe una solución general para una clase de problemas, en contraste con una solución específica a un problema concreto, ha escrito un algoritmo.

Mencionamos esta palabra antes pero no la definimos con precisión. No es fácil de definir, así que probaremos un par de enfoques.

Primero, piense en algo que **no es un algoritmo**. Cuando usted aprendió a multiplicar números de una cifra, probablemente memorizó la tabla de multiplicar. En efecto, memorizó 100 soluciones específicas. Ese tipo de conocimiento **no es algorítmico**.

Pero si usted era **haragán** probablemente hizo trampa aprendiendo algunos trucos. Por ejemplo, para encontrar el producto de **n por 9**, puede escribir **n – 1** como el primer dígito y **10 – n** como el segundo dígito. Este truco es una solución general para multiplicar cualquier número de una cifra por 9. **¡Eso es un algoritmo!**

De forma similar, las técnicas que aprendió para la suma y la resta con acarreo y la división larga son todas algoritmos. Una de las características de los algoritmos es que no requieren inteligencia para llevarse a cabo. Son procesos mecánicos en los que cada paso sigue al anterior de acuerdo a un conjunto simple de reglas.

En nuestra opinión, es un poco vergonzoso que los humanos pasen tanto tiempo en la escuela aprendiendo a ejecutar algoritmos que, de forma bastante similar, no exigen inteligencia. Por otra parte, el proceso de diseñar algoritmos es interesante, un desafío intelectual y una parte primordial de lo que llamamos programar.

Algunas de las cosas que la gente hace naturalmente, sin dificultad ni pensamiento consciente, son las más difíciles de expresar algorítmicamente. Entender el lenguaje natural es un buen ejemplo. Todos lo hacemos, pero hasta el momento nadie ha sido capaz de explicar cómo lo hacemos, al menos no en la forma de un algoritmo.

Glosario

función pura: *Una función que no modifica ninguno de los objetos que recibe como parámetros. La mayoría de las funciones puras son rentables.*

modificador: *Una función que modifica uno o más de los objetos que recibe como parámetros. La mayoría de los modificadores no entregan resultado.*

estilo funcional de programación: *Un estilo de programación en el que la mayoría de las funciones son puras.*

desarrollo de prototipos: *Una forma de desarrollar programas empezando con un prototipo y probándolo y mejorándolo gradualmente.*

desarrollo planificado: *Una forma de desarrollar programas que implica una comprensión de alto nivel del problema y más planificación que desarrollo incremental o desarrollo de prototipos.*

algoritmo: *Un conjunto de instrucciones para solucionar una clase de problemas por medio de un proceso mecánico sin intervención de inteligencia.*

Clases y métodos