**Introducción a la programación**

*Se suele decir que una persona no entiende algo de verdad hasta que puede explicárselo a otro. En realidad, no lo entiende de verdad hasta que puede explicárselo a un computador.* — [Donald Knuth](http://es.wikipedia.org/wiki/Donald_Knuth).

Si tuvieramos que resumir el propósito de la programación en una frase, ésta debería ser:

que el computador haga el trabajo por nosotros.

Los computadores son buenos para hacer tareas rutinarias. Idealmente, cualquier problema tedioso y repetitivo debería ser resuelto por un computador, y los seres humanos sólo deberíamos encargarnos de los problemas realmente interesantes: los que requieren creatividad, pensamiento crítico y subjetividad.

La **programación** es el proceso de transformar un método para resolver problemas en uno que pueda ser entendido por el computador.

**Algoritmos**

*La informática se trata de computadores tanto como la astronomía se trata de telescopios*. — [Edsger Dijkstra](http://es.wikipedia.org/wiki/Edsger_Dijkstra).

Al diseñar un programa, el desafío principal es crear y describir un procedimiento que esté completamente bien definido, que no tenga ambigüedades, y que efectivamente resuelva el problema.

Así es como la programación no es tanto sobre computadores, sino sobre resolver problemas de manera estructurada. El objeto de estudio de la programación no son los programas, sino los algoritmos.

Un **algoritmo** es un procedimiento bien definido para resolver un problema.

Todo el mundo conoce y utiliza algoritmos a diario, incluso sin darse cuenta:

* Una receta de cocina es un algoritmo; si bien podríamos cuestionar que algunos pasos son ambiguos (¿cuánto es «una pizca de sal»? ¿qué significa «agregar a gusto»?), en general las instrucciones están lo suficientemente bien definidas para que uno las pueda seguir sin problemas.

La entrada de una receta son los ingredientes y algunos datos como: ¿para cuántas personas se cocinará? El proceso es la serie de pasos para manipular los ingredientes. La salida es el plato terminado.

En principio, si una receta está suficientemente bien explicada, podría permitir preparar un plato a alguien que no sepa nada de cocina.

* El [método para multiplicar](http://es.wikipedia.org/wiki/Algoritmo_de_multiplicaci%C3%B3n) números a mano que aprendimos en el colegio es un algoritmo. Dado cualquier par de números enteros, si seguimos paso a paso el procedimiento siempre obtendremos el producto:

La entrada del algoritmo de multiplicación son los dos factores. El proceso es la secuencia de pasos en que los dígitos van siendo multiplicados las reservas van siendo sumadas, y los productos intermedios son finalmente sumados. La salida del algoritmo es el producto obtenido.

Un algoritmo debe poder ser usado mecánicamente, sin necesidad de usar inteligencia, intuición ni habilidad.

A lo largo de esta asignatura, haremos un recorrido por los conceptos elementales de la programación, con énfasis en el aspecto práctico de la disciplina.

Al final del semestre, usted tendrá la capacidad de identificar problemas que pueden ser resueltos por el computador, y de diseñar y escribir programas sencillos. Además, entenderá qué es lo que ocurre dentro del computador los programas que usted usa.

*Los computadores son inútiles: sólo pueden darte respuestas*. — [Pablo Picasso](http://es.wikipedia.org/wiki/Pablo_Picasso).

Componentes de un algoritmo

Conceptualmente, un algoritmo tiene tres componentes:

* la **entrada**: son los datos sobre los que el algoritmo opera;
* el **proceso**: son los pasos que hay que seguir, utilizando la entrada;
* la **salida**: es el resultado que entrega el algoritmo.

El proceso es una secuencia de **sentencias**, que debe ser realizada en orden. El proceso también puede tener **ciclos** (grupos de sentencias que son ejecutadas varias veces) y **condicionales** (grupos de sentencias que sólo son ejecutadas bajo ciertas condiciones).

**Cómo describir un algoritmo**

Consideremos un ejemplo sencillo: un algoritmo para resolver ecuaciones cuadráticas.

Una [ecuación cuadrática](http://es.wikipedia.org/wiki/Ecuaci%C3%B3n_de_segundo_grado) es una ecuación de la forma ax2+bx+c=0ax2+bx+c=0, donde aa, bb y cc son datos dados, con a≠0a≠0, y xx es la incógnita cuyo valor que se desea determinar.

Por ejemplo, 2x2−5x+2=02x2−5x+2=0 es una ecuación cuadrática con a=2a=2, b=−5b=−5 y c=2c=2. Sus soluciones son x1=1/2x1=1/2 y x2=2x2=2, como se puede comprobar fácilmente al reemplazar estos valores en la ecuación. El problema es cómo obtener estos valores en primer lugar.

El problema computacional de resolver una ecuación cuadrática puede ser planteado así:

Dados aa, bb y cc, entontrar los valores reales de xx que satisfacen ax2+bx+c=0ax2+bx+c=0.

La entrada del algoritmo, pues, son los valores aa, bb y cc, y la salida son las raíces reales xx (que pueden ser cero, una o dos) de la ecuación. En un programa computacional, los valores de aa, bb y cc deberían ser ingresados usando el teclado, y las soluciones xx deberían ser mostradas a continuación en la pantalla.

Al estudiar álgebra aprendemos un algoritmo para resolver este problema. Es lo suficientemente detallado para que pueda usarlo cualquier persona, incluso sin saber qué es una ecuación cuadrática, o para que lo pueda hacer un computador. A continuación veremos algunas maneras de describir el procedimiento.

***Lenguaje natural***

Durante el proceso mental de diseñar un algoritmo, es común pensar y describir los pasos en la misma manera en que hablamos a diario. Por ejemplo:

Teniendo los valores de aa, bb y cc, calcular el discriminante D=b2−4acD=b2−4ac. Si es discriminante es negativo, entonces la ecuación no tiene soluciones reales. Si es discriminante es igual a cero, entonces la ecuación tiene una única solución real, que es x=−b/2ax=−b/2a. Si el discriminante es positivo, entonces la ecuación tiene dos soluciones reales, que son x1=(−b−D−−√)/2ax1=(−b−D)/2a y x2=(−b+D−−√)/2ax2=(−b+D)/2a.

Esta manera de expresar un algoritmo no es ideal, ya que el lenguaje natural es:

* impreciso: puede tener ambigüedades;
* no universal: personas distintas describirán el proceso de maneras distintas; y
* no estructurado: la descripción no está expresada en función de componentes simples.

Aún así, es posible identificar los pasos del algoritmo. Por ejemplo, hay que evaluar la expresión b2−4acb2−4ac, y ponerle el nombre DD a su resultado. Esto se llama **asignación**, y es un tipo de instrucción que aparece en casi todos los algoritmos. Después de eso, el algoritmo puede usar el nombre DD para referirse al valor calculado.

***Diagrama de flujo***

Un **diagrama de flujo** es una representación gráfica de un algoritmo. Los pasos son representados por varios tipos de bloques, y el flujo de ejecución es indicado por flechas que conectan los bloques:

El inicio y el final del algoritmo son representados con bloques circulares. El algoritmo siempre debe ser capaz llegar desde uno hasta el otro, sin importar por qué camino lo hace. Un algoritmo no puede «quedarse pegado» en la mitad.

La entrada y la salida de datos son representadas con romboides, que en la figura de arriba están pintados de verde.

Los diamantes representan condiciones en las que el algoritmo sigue uno de dos caminos. que están etiquetados con *sí* o *no*, dependiendo si la condición es verdadera o falsa.

También puede haber ciclos, representados por flechas que regresan a bloques anteriores. En este ejemplo, no hay ciclos.

Otras sentencias van dentro de rectángulos, que en la figura están pintados de azul. En este ejemplo, las sentencias son asignaciones, representadas en la forma nombre = valor.

Los diagramas de flujo no son usados en la práctica para programar, pero son útiles para ilustrar cómo funcionan algoritmos sencillos.

***Pseudocódigo***

El **pseudocódigo** es una descripción estructurada de un algoritmo basada en ciertas convenciones notacionales. Si bien es muy parecido al código que finalmente se escribirá en el computador, el pseudocódigo está pensado para ser leído por humanos.

Una manera de escribir el algoritmo para la ecuación cuadrática en pseudocódigo es la siguiente:

leer a

leer b

leer c

discriminante = b² - 4ac

si discriminante < 0:

escribir 'La ecuación no tiene soluciones reales'

o si no, si discriminante = 0:

x = -b / 2a

escribir 'La solución única es', x

o si no:

x1 = (-b - √discriminante) / 2a

x2 = (-b + √discriminante) / 2a

escribir 'Las dos soluciones reales son:'

escribir x1

escribir x2

Las líneas que comienzan con leer y escribir denotan, respectivamente, la entrada y la salida del programa. Los diferentes casos son representados usando sentencias si y o si no. Las asignaciones siguen la misma notación que en el caso de los diagramas de flujo.

La notación de pseudocódigo es bien liberal. Uno puede mezclar notación de matemáticas con frases en español, siempre que quede absolutamente claro para el lector qué representa cada una de las líneas del algoritmo.

***Código***

El producto final de la programación siempre debe ser código que pueda ser ejecutado en el computador. Esto requiere describir los algoritmos en un **lenguaje de programación**. Los lenguajes de programación definen un conjunto limitado de conceptos básicos, en función de los cuales uno puede expresar cualquier algoritmo.

En esta asignatura, usaremos el lenguaje de programación [Python](http://python.org/) para escribir nuestros programas.

El código en Python para resolver la ecuación cuadrática es el siguiente:

a = float(raw\_input('Ingrese a: '))

b = float(raw\_input('Ingrese b: '))

c = float(raw\_input('Ingrese c: '))

discriminante = b \*\* 2 - 4 \* a \* c

**if** discriminante < 0:

**print** 'La ecuacion no tiene soluciones reales'

**elif** discriminante == 0:

x = -b / (2 \* a)

**print** 'La solucion unica es x =', x

**else**:

x1 = (-b - (discriminante \*\* 0.5)) / (2 \* a)

x2 = (-b + (discriminante \*\* 0.5)) / (2 \* a)

**print** 'Las dos soluciones reales son:'

**print** 'x1 =', x1

**print** 'x2 =', x2

raw\_input()

A partir de ahora, usted aprenderá a entender, escribir y ejecutar códigos como éste.

**Desarrollo de programas**

Un **programa** es un archivo de texto que contiene código para ser ejecutado por el computador.

En el caso del lenguaje Python, el programa es ejecutado por un **intérprete**. El intérprete es un programa que ejecuta programas.

Los programas escritos en Python deben estar contenidos en un archivo que tenga la extensión .py. En Windows, el programa puede ser ejecutado haciendo doble clic sobre el ícono del archivo.

Para probar cómo hacerlo, descargue el programa [cuadratica.py](http://progra.usm.cl/apunte/_static/programas/cuadratica.py) que sirve para resolver ecuaciones cuadráticas.

**Edición de programas**

Un programa es un [archivo de texto](http://es.wikipedia.org/wiki/Archivo_de_texto). Por lo tanto, puede ser creado y editado usando cualquier [editor de texto](http://es.wikipedia.org/wiki/Editor_de_texto), como el Bloc de Notas.

Lo que no se puede usar es un procesador de texto, como Microsoft Word.

Haga la prueba: abra el programa cuadratica.py con el Bloc de Notas (u otro editor) y verá su contenido.

Otros editores de texto (mucho mejores que el Bloc de Notas) que usted puede instalar son:

* en Windows: [Notepad++](http://notepad-plus-plus.org/), [Textpad](http://www.textpad.com/);
* en Mac: [TextWrangler](http://www.barebones.com/products/textwrangler/), [TextMate](http://macromates.com/);
* en Linux: [Gedit](http://projects.gnome.org/gedit/), [Kate](http://kate-editor.org/).

**Instalación del intérprete de Python**

Una cosa es editar el programa, y otra es ejecutarlo. Para poder ejecutar un programa en Python hay que instalar el **intérprete**.

En la [página de descargas de Python](http://www.python.org/download/) está la lista de instaladores. Debe descargar el indicado para su computador y su sistema operativo.

La versión que debe instalar es la **2.7.3**, no la 3.2.3.

No use los instaladores que dicen x86-64 a no ser que esté seguro que su computador tiene una arquitectura de 64 bits (lo más probable es que no sea así).

**Ejecución de un programa**

Una vez escrito el programa e instalado el intérprete, es posible ejecutar los programas. Para hacerlo, haga doble clic en el ícono del programa.

**Uso de la consola**

La ejecución de programas no es la única manera de ejecutar el intérprete. Si uno ejecuta Python sin pasarle ningún programa, se abre la **consola** (o **intérprete interactivo**).

La consola permite ingresar un programa línea por línea. Además, sirve para evaluar expresiones y ver su resultado inmediatamente. Esto permite usarla como si fuera una calculadora.

La consola interactiva siempre muestra el símbolo >>>, para indicar que ahí se puede ingresar código. En todos los libros sobre Python, y a lo largo de este apunte, cada vez que aparezca un ejemplo en el que aparezca este símbolo, significa que debe ejecutarse en la consola, y no en un programa. Por ejemplo:

**>>>** a = 5

**>>>** a > 10

False

**>>>** a \*\* 2

25

En este ejemplo, al ingresar las expresiones a > 10 y a \*\* 2, el intérprete interactivo entrega los resultados False y 25.

No hay ningún motivo para tipear el símbolo >>> ni en un programa ni en un certamen, pues no es parte de la sintaxis del lenguaje.

**Entornos de desarollo**

En general, usar un simple editor de texto para escribir programas no es la manera más eficiente de trabajar.

Los **entornos de desarrollo** (también llamados *IDE*, por sus siglas en inglés) son aplicaciones que hacen más fácil la tarea de escribir programas.

Python viene con su propio entorno de desarrollo llamado **IDLE**. IDLE viene con una consola y un editor de texto.

Además, hay otros buenos entornos de desarrollo más avanzados para Python:

* [PyScripter](http://code.google.com/p/pyscripter/downloads/list),
* [WingIDE 101](http://www.wingware.com/downloads/wingide-101/3.2.12-1/binaries)

Usted puede probar éstos y usar el que más le acomode durante el semestre.

El siguiente video muestra cómo usar IDLE para desarrollar un programa y para usar la consola interactiva:

Si desea trabajar con PyScripter en vez de IDLE, puede ver [este otro video](http://www.youtube.com/watch?v=bzF5rDtQLS4) con una demostración de cómo usarlo.

**Tipos de datos**

Un **tipo de datos** es la propiedad de un valor que determina su dominio (qué valores puede tomar), qué operaciones se le pueden aplicar y cómo es representado internamente por el computador.

Todos los valores que aparecen en un programa tienen un tipo.

A continuación revisaremos los tipos de datos elementales de Python. Además de éstos, existen muchos otros, y más adelante aprenderemos a crear nuestros propios tipos de datos.

**Números enteros**

El tipo **int** (del inglés *integer*, que significa «entero») permite representar números enteros.

Los valores que puede tomar un int son todos los números enteros: ... -3, -2, -1, 0, 1, 2, 3, ...

Los números enteros literales se escriben con un signo opcional seguido por una secuencia de dígitos:

1570

+4591

-12

**Números reales**

El tipo **float** permite representar números reales.

El nombre float viene del término [punto flotante](http://es.wikipedia.org/wiki/Punto_flotante), que es la manera en que el computador representa internamente los números reales.

Hay que tener mucho cuidado, porque los números reales no se pueden representar de manera exacta en un computador. Por ejemplo, el número decimal 0.7 es representado internamente por el computador mediante la aproximación 0.69999999999999996. Todas las operaciones entre valores float son aproximaciones. Esto puede conducir a resultados algo sorpresivos:

**>>>** 1/7 + 1/7 + 1/7 + 1/7 + 1/7 + 1/7 + 1/7

0.9999999999999998

Los números reales literales se escriben separando la parte entera de la decimal con un punto. Las partes entera y decimal pueden ser omitidas si alguna de ellas es cero:

**>>>** 881.9843000

881.9843

**>>>** -3.14159

-3.14159

**>>>** 1024.

1024.0

**>>>** .22

0.22

Otra representación es la notación científica, en la que se escribe un factor y una potencia de diez separados por una letra e. Por ejemplo:

**>>>** -2.45E4

-24500.0

**>>>** 7e-2

0.07

**>>>** 6.02e23

6.02e+23

**>>>** 9.1094E-31

9.1094e-31

Los dos últimos valores del ejemplo son iguales, respectivamente, a 6.02×10236.02×1023 (la [constante de Avogadro](http://es.wikipedia.org/wiki/Constante_de_Avogadro)) y 9.1094×10−319.1094×10−31 (la [masa del electrón](http://es.wikipedia.org/wiki/Electr%C3%B3n)).

**Números complejos**

El tipo **complex** permite representar números complejos.

Los números complejos tienen una parte real y una imaginaria. La parte imaginaria es denotada agregando una j inmediatamente después de su valor:

3 + 9j

-1.4 + 2.7j

**Valores lógicos**

Los valores lógicos True y False (verdadero y falso) son de tipo **bool**, que representa valores lógicos.

El nombre bool viene del matemático [George Boole](http://es.wikipedia.org/wiki/George_Boole), quien creó un sistema algebraico para la lógica binaria. Por lo mismo, a True y False también se les llama **valores booleanos**. El nombre no es muy intuitivo, pero es el que se usa en informática, así que hay que conocerlo.

**Texto**

A los valores que representan texto se les llama **strings**, y tienen el tipo **str**.

Los strings literales pueden ser representados con texto entre comillas simples o comillas dobles:

"ejemplo 1"

'ejemplo 2'

La ventaja de tener dos tipos de comillas es que se puede usar uno de ellos cuando el otro aparece como parte del texto:

"Let's go!"

'Ella dijo "hola"'

Es importante entender que los strings no son lo mismo que los valores que en él pueden estar representados:

**>>>** 5 == '5'

False

**>>>** True == 'True'

False

Los strings que difieren en mayúsculas y minúsculas, o en espacios también son distintos:

**>>>** 'mesa' == 'Mesa'

False

**>>>** ' mesa' == 'mesa '

False

**Nulo**

Existe un valor llamado **None** (en inglés, «ninguno») que es utilizado para representar casos en que ningún valor es válido, o para indicar que una variable todavía no tiene un valor que tenga sentido.

El valor None tiene su propio tipo, llamado NoneType, que es diferente al de todos los demás valores.

**Programas simples**

Un programa es una secuencia de **sentencias**. Una sentencia representa una instrucción bien definida que es ejecutada por el computador. En Python, cada línea del código representa una sentencia.

Hay que distinguir entre:

* **sentencias simples**: son una única instrucción; y
* **sentencias de control**: contienen varias otras sentencias, que a su vez pueden ser simples o de control.

Las sentencias simples son ejecutadas secuencialmente, una después de la otra.

Todas las sentencias siguen ciertas reglas acerca de cómo deben ser escritas. Si no son seguidas, el programa está incorrecto y no se ejecutará. A este conjunto de reglas se le denomina **sintaxis**.

A continuación veremos algunas sentencias simples, con las que se pueden escribir algunos programas sencillos. Más adelante introduciremos las sentencias de control.

Como ejemplo, consideremos el siguiente programa, que pide al usuario ingresar una temperatura en grados Fahrenheit y entrega como resultado el equivalente en grados Celsius:

[Descargue](http://progra.usm.cl/apunte/_static/programas/temperatura.py) el programa y ejecútelo para convencerse de que funciona correctamente.

**Expresiones y variables**

Una **expresión** es una combinación de valores y operaciones que son evaluados durante la ejecución del algoritmo para obtener un resultado.

Por ejemplo, 2 + 3 es una expresión aritmética que, al ser evaluada, siempre entrega el valor 5 como resultado. En esta expresión, 2 y 3 son **valores literales** y + es el operador de adición.

En el programa de conversión de temperaturas aparece la expresión (f - 32.0) \* (5.0 / 9.0), cuyo resultado depende de cuál es el valor de f al momento de la evaluación. A diferencia de los valores literales, ff es una **variable** que tiene un valor específico que puede ser distinto cada vez que la expresión es evaluada.

En esta expresión, \* es el operador de multiplicación y / el de división.

Una expresión puede ser usada como una sentencia de un programa por sí sola, pero la mayoría de las veces esto no tiene ningún efecto. El programa evaluará la expresión, pero no hará nada con el resultado obtenido.

**Asignaciones**

Una **asignación** es una sentencia que asocia un nombre al resultado de una expresión. El nombre asociado al valor se llama **variable**.

La sintaxis de una asignación es la siguiente:

variable = expresión

Por ejemplo, el programa de conversión de temperaturas tiene la siguiente asignación:

c = (f - 32.0) \* (5.0 / 9.0)

Cuando aparece una asignación en un programa, es interpretada de la siguiente manera:

* primero la expresión a la derecha del signo = es evaluada, utilizando los valores que tienen en ese momento las variables que aparecen en ella;
* una vez obtenido el resultado, el valor de la variable a la izquierda del signo = es reemplazado por ese resultado.

Bajo esta interpretación, es perfectamente posible una asignación como ésta:

i = i + 1

Primero la expresión i + 1 es evaluada, entregando como resultado el sucesor del valor actual de i. A continuación, la variable i toma el nuevo valor. Por ejemplo, si i tiene el valor 15, después de la asignación tendrá el valor 16.

Esto no significa que 15=1615=16. Una asignación no es una igualdad matemática ni una ecuación.

Por ejemplo, las siguientes asignaciones son correctas, suponiendo que las variables que aparecen en ellas ya fueron asignadas previamente:

nombre = 'Perico Los Palotes'

discriminante = b \*\* 2 - 4 \* a \* c

pi = 3.14159

r = 5.0

perimetro = 2 \* pi \* r

sucesor = n + 1

a = a

es\_menor = x < 4

x0 = x1 + x2

r = 2 \* abs(x - x0)

nombre = raw\_input('Ingrese su nombre')

Las siguientes no son asignaciones válidas, pues no respetan la sintaxis nombre = expresión (tarea: identifique los errores):

n + 1 = 5

7 = a

2\_pi\_r = 2 \* pi \* r

area del circulo = pi \* r \*\* 2

x \*\* 2 = x \* x

**Entrada**

La **entrada** es la parte del programa en que el usuario ingresa datos.

La manera más simple de ingresar datos es hacerlo a través del teclado. La función raw\_input(mensaje) pide al usuario ingresar un valor, que puede ser asignado a una variable para ser usado por el programa. El mensaje es lo que se mostrará al usuario antes de que él ingrese el valor.

El valor ingresado por el usuario siempre es interpretado como texto, por lo que es de tipo str. Si es necesario usarlo como si fuera de otro tipo, hay que convertirlo explícitamente.

Por ejemplo, en el programa de conversión de temperaturas, la entrada es realizada por la sentencia:

f = float(raw\_input('Ingrese temperatura en Fahrenheit: '))

Cuando el programa llega a esta línea, el mensaje Ingrese temperatura en Fahrenheit: es mostrado al usuario, que entonces debe ingresar un valor, que es convertido a un número real y asociado al nombre f.

Desde esa línea en adelante, la variable f puede ser usada en el programa para referirse al valor ingresado.

**Salida**

La **salida** es la parte del programa en que los resultados son entregados al usuario.

La manera más simple de entregar la salida es mostrando texto en la pantalla. En Python, la salida del programa es realizada por la sentencia **print** (*imprimir* en inglés).

Si se desea imprimir un texto tal cual, la sintaxis es la siguente:

**print** valor\_a\_imprimir

Si los valores a imprimir son varios, deben ser puestos separados por comas. Por ejemplo, el programa de conversión de temperaturas tiene la siguiente sentencia de salida:

**print** 'El equivalente en Celsius es:', c

En este caso, se está imprimiendo el mensaje El equivalente en Celsius es: y a continuación, en la misma línea, el valor de la variable c.

Las comillas sólo sirven para representar un string en el código, y no forman parte del string. Al imprimir el string usando print las comillas no aparecen:

**>>>** 'Hola'

'Hola'

**>>> print** 'Hola'

Hola

**Comentarios**

Un **comentario** es una sección del código que es ignorada por el intérprete. Un comentario puede ser utilizado por el programador para dejar un mensaje en el código que puede ser útil para alguien que tenga que leerlo en el futuro.

En Python, cualquier texto que aparezca a la derecha de un signo # es un comentario:

**>>>** 2 + 3 *# Esto es una suma*

5

**>>>** *# Esto es ignorado*

>>>

La excepción son los signos # que aparecen en un string:

**>>>** "123 # 456" *# 789*

'123 # 456'

**Evitar que se cierre el programa**

La ejecución de programas en Windows presenta un inconveniente práctico: cuando el programa termina, la ventana de ejecución se cierra inmediatamente, por lo que no es posible alcanzar a leer la salida del programa.

Por ejemplo, al ejecutar el programa temperatura.py tal como está arriba, el usuario verá el mensaje Ingrese temperatura... y a continuación ingresará el valor. Una vez que el programa entrega como resultado el equivalente en grados Celcius, no quedan más sentencias para ejecutar, por lo que el programa se cierra.

Existen otras maneras de ejecutar programas con las que este problema no ocurre. Por ejemplo, al ejecutar un programa desde una IDE, generalmente la salida aparece en una ventana que no se cierra.

Una solución para evitar que la ventana se cierre es agregar un raw\_input() al final del código. De este modo, el programa quedará esperando que el usuario ingrese cualquier cosa (un enter basta) antes de cerrarse.

Los programas presentados en este apunte no tendrán el raw\_input() al final, pero usted puede agregarlo por su cuenta si así lo desea. En controles y certámenes, no será necesario hacerlo.

**Expresiones**

Una **expresión** es una combinación de valores y operaciones que, al ser evaluados, entregan un valor.

Algunos elementos que pueden formar parte de una expresión son: valores **literales** (como 2, "hola" o 5.7), **variables**, **operadores** y **llamadas a funciones**.

Por ejemplo, la expresión 4 \* 3 - 2 entrega el valor 10 al ser evaluada por el intérprete:

**>>>** 4 \* 3 - 2

10

El valor de la siguiente expresión depende del valor que tiene la variable n en el momento de la evaluación:

**>>>** n / 7 + 5

Una expresión está compuesta de otras expresiones, que son evaluadas recursivamente hasta llegar a sus componentes más simples, que son los literales y las variables.

Hasta aquí va la primera parte.

**Operadores**

Un **operador** es un símbolo en una expresión que representa una operación aplicada a los valores sobre los que actúa.

Los valores sobre los que actúa un operador se llaman **operandos**. Un **operador binario** es el que tiene dos operandos, mientras que un **operador unario** es el que tiene sólo uno.

Por ejemplo, en la expresión 2.0 + x el operador + es un operador binario que en este contexto representa la operación de adición. Sus operandos son 2.0 y x.

Las operaciones más comunes se pueden clasificar en: aritméticas, relacionales, lógicas y de texto.

***Operadores aritméticos***

Las **operaciones aritméticas** son las que operan sobre valores numéricos y entregan otro valor numérico como resultado. Los valores numéricos son los que tienen tipo entero, real o complejo.

Las siguientes son algunas operaciones aritméticas básicas, junto con el operador que las representa en Python:

* la **suma** +;
* la **resta** -;
* la **multiplicación** \*;
* la **división** /;
* el **módulo** % (resto de la división);
* la **potencia** \*\* («elevado a»).

En general, si los operandos son de tipo entero, el resultado también será de tipo entero. Pero basta que uno de los operandos sea real para que el resultado también lo sea:

**>>>** 8 - 5

3

**>>>** 8 - 5.0

3.0

**>>>** 8.0 - 5

3.0

**>>>** 8.0 - 5.0

3.0

Esta regla suele causar confusión en el caso de la división. Al dividir números enteros, el resultado siempre es entero, y es igual al resultado real **truncado**, es decir, sin su parte decimal:

**>>>** 5 / 2

2

**>>>** 5 / -2

-3

Si uno de los operandos es complejo, el resultado también será complejo:

**>>>** 3 + 4

7

**>>>** 3 + (4+0j)

(7+0j)

El operador de módulo entrega el resto de la división entre sus operandos:

**>>>** 7 % 3

1

Un uso bastante común del operador de módulo es usarlo para determinar si un número es divisible por otro:

**>>>** 17 % 5 *# 17 no es divisible por 5*

2

**>>>** 20 % 5 *# 20 si es divisible por 5*

0

Una relación entre / y % que siempre se cumple para los números enteros es:

(a / b) \* b + (a % b) == a

Hay dos operadores aritméticos unarios:

* el **positivo** +, y
* el **negativo** -.

El positivo entrega el mismo valor que su operando, y el negativo también pero con el signo cambiado:

**>>>** n = -4

**>>>** +n

-4

**>>>** -n

4

***Operaciones relacionales***

Las **operaciones relacionales** sirven para comparar valores. Sus operandos son cualquier cosa que pueda ser comparada, y sus resultados siempre son valores lógicos.

Algunas operaciones relacionales son:

* el **igual a** == (no confundir con el = de las asignaciones);
* el **distinto a** !=;
* el **mayor que** >;
* el **mayor o igual que** >=;
* el **menor que** <;
* el **menor o igual que** <=;

Algunos ejemplos en la consola interactiva:

**>>>** a = 5

**>>>** b = 9

**>>>** c = 14

**>>>** a < b

True

**>>>** a + b != c

False

**>>>** 2.0 == 2

True

**>>>** 'amarillo' < 'negro'

True

Los operadores relacionales pueden ser encadenados, como se usa en matemáticas, de la siguiente manera:

**>>>** x = 4

**>>>** 0 < x <= 10

True

**>>>** 5 <= x <= 20

False

La expresión 0 < x <= 10 es equivalente a (0 < x) and (x <= 10)

***Operaciones lógicas***

Los **operadores lógicos** son los que tienen operandos y resultado de tipo lógico.

En Python, hay tres operaciones lógicas:

* la conjunción lógica **and** (en español: «y»),
* la disyunción lógica **or** (en español: «o»), y
* la negación lógica **not** (en español: «no»).

Los operadores and y or son binarios, mientras que not es unario:

**>>>** True **and** False

False

**>>> not** True

False

La siguiente tabla muestra todos los resultados posibles de las operaciones lógicas. Las primeras dos columnas representan los valores de los operandos, y las siguientes tres, los resultados de las operaciones.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P | q | p and q | p or q | not p |
| True | True | True | True | False |
| True | False | False | True |  |
| False | True | False | True | True |
| False | False | False | False |  |

***Operaciones de texto***

Los operadores + y \* tienen otras interpretaciones cuando sus operandos son strings.

+ es el operador de **concatenación** de strings: pega dos strings uno después del otro:

**>>>** 'perro' + 'gato'

'perrogato'

La concatenación no es una suma. Ni siquiera es una operación conmutativa.

\* es el operador de **repetición** de strings. Recibe un operando string y otro entero, y entrega como resultado el string repetido tantas veces como indica el entero:

**>>>** 'waka' \* 2

'wakawaka'

Más adelante veremos muchas más operaciones para trabajar sobre texto. Por ahora utilizaremos las más elementales. Otras operaciones que pueden serle útiles por el momento son:

* obtener el ii-ésimo caracter de un string (partiendo desde cero) usando los corchetes:
* **>>>** nombre = 'Perico'
* **>>>** nombre[0]
* 'P'
* **>>>** nombre[1]
* 'e'
* **>>>** nombre[2]
* 'r'
* comprarar strings alfabéticamente con los operadores relacionales (lamentablemente no funciona con acentos y eñes):
* **>>>** 'a' < 'abad' < 'abeja'
* True
* **>>>** 'zapato' <= 'alpargata'
* False
* obtener el largo de un string con la función len:
* **>>>** len('papalelepipedo')
* 14
* **>>>** len("")
* 0
* verificar si un string está dentro de otro con el operador in:
* **>>>** 'pollo' **in** 'repollos'
* True
* **>>>** 'pollo' **in** 'gallinero'
* False

**Precedencia**

La **precedencia de operadores** es un conjunto de reglas que especifica en qué orden deben ser evaluadas las operaciones de una expresión.

La precedencia está dada por la siguiente lista, en que los operadores han sido listados en orden de menor a mayor precedencia:

* or
* and
* not
* <, <=, >, >=, !=, ==
* +, - (suma y resta)
* \*, /, %
* +, - (positivo y negativo)
* \*\*

Esto significa, por ejemplo, que las multiplicaciones se evalúan antes que las sumas, y que las comparaciones se evalúan antes que las operaciones lógicas:

**>>>** 2 + 3 \* 4

14

**>>>** 1 < 2 **and** 3 < 4

True

Operaciones dentro de un mismo nivel son evaluadas en el orden en que aparecen en la expresión, de izquierda a derecha:

**>>>** 15 \* 12 % 7 *# es igual a (15 \* 12) % 7*

5

La única excepción a la regla anterior son las potencias, que son evaluadas de derecha a izquierda:

**>>>** 2 \*\* 3 \*\* 2 *# es igual a 2 \*\* (3 \*\* 2)*

512

Para forzar un orden de evaluación distinto a la regla de precedencia, debe usarse paréntesis:

**>>>** (2 + 3) \* 4

20

**>>>** 15 \* (12 % 7)

75

**>>>** (2 \*\* 3) \*\* 2

64

Otra manera de forzar el orden es ir guardando los resultados intermedios en variables:

**>>>** n = 12 % 7

**>>>** 15 \* n

75

Como ejemplo, consideremos la siguiente expresión:

15 + 59 \* 75 / 9 < 2 \*\* 3 \*\* 2 **and** (15 + 59) \* 75 % n == 1

y supongamos que la variable n tiene el valor 2. Aquí podemos ver cómo la expresión es evaluada hasta llegar al resultado final, que es False:

15 + 59 \* 75 / 9 < 2 \*\* 3 \*\* 2 and (15 + 59) \* 75 % n == 1

# ↓

15 + 59 \* 75 / 9 < 2 \*\* 9 and (15 + 59) \* 75 % n == 1

# ↓

15 + 59 \* 75 / 9 < 512 and (15 + 59) \* 75 % n == 1

# ↓

15 + 4425 / 9 < 512 and (15 + 59) \* 75 % n == 1

# ↓

15 + 491 < 512 and (15 + 59) \* 75 % n == 1

# ↓

15 + 491 < 512 and 74 \* 75 % n == 1

# ↓

15 + 491 < 512 and 5550 % n == 1

# ↓

15 + 491 < 512 and 5550 % 2 == 1

# ↓

15 + 491 < 512 and 0 == 1

# ↓

506 < 512 and 0 == 1

# ↓

True and 0 == 1

# ↓

True and False

# ↓

False

La operación entre paréntesis (15 + 59) debe ser evaluada antes de la multiplicación por 75, ya que es necesario conocer su resultado para poder calcular el producto. El momento preciso en que ello ocurre no es importante.

Lo mismo ocurre con la evaluación de la variable n: sólo importa que sea evaluada antes de ser usada por el operador de módulo.

En el ejemplo, ambos casos fueron evaluados inmediatamente antes de que su valor sea necesario.

Las reglas completas de precedencia, incluyendo otros operadores que aún no hemos visto, pueden ser consultados en [la sección sobre expresiones](http://docs.python.org/reference/expressions.html) de la documentación oficial de Python.

***¿Cómo aprenderse las reglas de precedencia?***

La respuesta es: mejor no aprendérselas. Las reglas de precedencia son muchas y no siempre son intuitivas,

Un programa queda mucho más fácil de entender si uno explícitamente indica el orden de evaluación usando paréntesis o guardando en variables los resultados intermedios del cálculo.

Un buen programador siempre se preocupa de que su código sea fácil de entender por otras personas, ¡e incluso por él mismo en unas semanas más adelante!

**Llamadas a función**

Los operadores forman un conjunto bastante reducido de operaciones. Más comúnmente, las operaciones más generales son representadas como **funciones**.

Al igual que en matemáticas, las funciones tienen un nombre, y reciben **parámetros** (o **argumentos**) que van entre paréntesis después del nombre. La operación de usar la función para obtener un resultado se llama **llamar la función**.

Ya conocemos la función raw\_input(), que entrega como resultado el texto ingresado por el usuario mediante el teclado.

La función abs entrega el valor absoluto de su argumento:

**>>>** abs(4 - 5)

1

**>>>** abs(5 - 4)

1

La función len recibe un string y entrega su largo. (más adelante veremos otros usos de la función len):

**>>>** len('hola mundo')

10

**>>>** len('hola' \* 10)

40

Los nombres de los tipos también sirven como funciones, que entregan el equivalente de su parámetro en el tipo correspondiente:

**>>>** int(3.8)

3

**>>>** float('1.5')

1.5

**>>>** str(5 + 6)

'11'

**>>>** int('5' + '6')

56

Las funciones min y max entregan el mínimo y el máximo de sus argumentos:

**>>>** min(6, 1, 8)

1

**>>>** min(6.0, 1.0, 8.0)

1.0

**>>>** max(6, 1, 4, 8)

8

La función round redondea un número real al entero más cercano:

**>>>** round(4.4)

4.0

**>>>** round(4.6)

5.0

Algunas funciones matemáticas como la exponencial, el logaritmo y las trigonométricas pueden ser usadas, pero deben ser importadas primero usando la sentencia import, que veremos en detalle más adelante:

**>>> from** **math** **import** exp

**>>>** exp(2)

7.3890560989306504

**>>> from** **math** **import** sin, cos

**>>>** cos(3.14)

-0.9999987317275395

**>>>** sin(3.14)

0.0015926529164868282

La lista completa de funciones matemáticas que pueden ser importadas está en la [descripción del módulo math](http://docs.python.org/library/math.html) en la documentación de Python.

Más adelante también aprenderemos a crear nuestras propias funciones. Por ahora, sólo necesitamos saber cómo llamarlas.

Por supuesto, siempre es necesario que los argumentos de una llamada tengan el tipo apropiado:

**>>>** round('perro')

Traceback (most recent call last):

File "<console>", line 1, in <module>

TypeError: a float is required

**>>>** len(8)

Traceback (most recent call last):

File "<console>", line 1, in <module>

TypeError: object of type 'int' has no len()

**Errores y excepciones**

No siempre los programas que escribiremos están correctos. Existen muchos tipos de errores que pueden estar presentes en un programa.

No todos los errores pueden ser detectados por el computador. Por ejemplo, el siguiente programa tiene un error lógico bastante evidente:

n = int(raw\_input('Escriba un numero: '))

doble = 3 \* n

**print** 'El doble de n es', doble

El computador no se dará cuenta del error, pues todas las instrucciones del programa son correctas. El programa simplemente entregará siempre la respuesta equivocada.

Existen otros errores que sí pueden ser detectados. Cuando un error es detectado *durante* la ejecución del programa ocurre una **excepción**.

El intérprete anuncia una excepción deteniendo el programa y mostrando un mensaje con la descripción del error. Por ejemplo, podemos crear el siguiente programa y llamarlo division.py:

n = 8

m = 0

**print** n / m

**print** 'Listo'

Al ejecutarlo, el intérprete lanzará una excepción, pues la división por cero es una operación inválida:

Traceback (most recent call last):

File "division.py", line 3, in <module>

print n / m

ZeroDivisionError: division by zero

La segunda línea del mensaje indica cómo se llama el archivo donde está el error y en qué línea del archivo está. En este ejemplo, el error está en la línea 3 de division.py. La última línea muestra el nombre de la excepción (en este caso es ZeroDivisionError) y un mensaje explicando cuál es el error.

Los errores y excepciones presentados aquí son los más básicos y comunes.

**Error de sintaxis**

Un **error de sintaxis** ocurre cuando el programa no cumple las reglas del lenguaje. Cuando ocurre este error, significa que el programa está mal escrito. El nombre del error es SyntaxError.

Los errores de sintaxis siempre ocurren *antes* de que el programa sea ejecutado. Es decir, un programa mal escrito no logra ejecutar ninguna instrucción. Por lo mismo, el error de sintaxis no es una excepción.

A continuación veremos algunos ejemplos de errores de sintaxis

**>>>** 2 \* (3 + 4))

File "<stdin>", line 1

2 \* (3 + 4))

^

SyntaxError: invalid syntax

**>>>** n + 2 = 7

File "<stdin>", line 1

SyntaxError: can't assign to operator

**>>>** True = 1000

File "<stdin>", line 1

SyntaxError: assignment to keyword

**Error de nombre**

Un **error de nombre** ocurre al usar una variable que no ha sido creada con anterioridad.

El nombre de la excepción es NameError:

**>>>** x = 20

**>>>** 5 \* x

100

**>>>** 5 \* y

Traceback (most recent call last):

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

NameError: name 'y' is not defined

Para solucionar este error, es necesario asignar un valor a la variable antes de usarla.

**Error de tipo**

En general, todas las operaciones en un programa pueden ser aplicadas sobre valores de tipos bien específicos. Un **error de tipo** ocurre al aplicar una operación sobre operandos de tipo incorrecto.

El nombre de la excepción es TypeError.

Por ejemplo, no se puede multiplicar dos strings:

**>>>** 'seis' \* 'ocho'

Traceback (most recent call last):

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

TypeError: can't multiply sequence by non-int of type 'str'

Tampoco se puede obtener el largo de un número:

**>>>** len(68)

Traceback (most recent call last):

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

TypeError: object of type 'int' has no len()

Cuando ocurre un error de tipo, generalmente el programa está mal diseñado. Hay que revisarlo, idealmente hacer un ruteo para entender el error, y finalmente corregirlo.

**Error de valor**

El **error de valor** ocurre cuando los operandos son del tipo correcto, pero la operación no tiene sentido para ese valor.

El nombre de la excepción es ValueError.

Por ejemplo, la función int puede convertir un string a un entero, pero el string debe ser la representación de un número entero. Cualquier otro valor lanza un error de valor:

**>>>** int('41')

41

**>>>** int('perro')

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: 'perro'

**>>>** int('cuarenta y uno')

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: 'cuarenta y uno'

Para corregir el error, hay que preocuparse de siempre usar valores adecuados.

**Error de división por cero**

El **error de division por cero** ocurre al intentar dividir por cero.

El nombre de la excepción es ZeroDivisionError:

**>>>** 1/0

Traceback (most recent call last):

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

ZeroDivisionError: division by zero

**Error de desborde**

El **error de desborde** ocurre cuando el resultado de una operación es tan grande que el computador no puede representarlo internamente.

El nombre de la excepción es OverflowError:

**>>>** 20.0 \*\* 20.0 \*\* 20.0

Traceback (most recent call last):

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

OverflowError: (34, 'Numerical result out of range')

Para los interesados en saber más sobre excepciones, pueden revisar [la sección sobre excepciones](http://docs.python.org/library/exceptions.html) en la documentación oficial de Python.

**Sentencias de control**

Un programa es una sucesión de **sentencias** que son ejecutadas secuencialmente.

Por ejemplo, el siguiente programa tiene cuatro sentencias:

n = int(raw\_input('Ingrese n: '))

m = int(raw\_input('Ingrese m: '))

suma = n + m

**print** 'La suma de n y m es:', suma

Las primeras tres son asignaciones, y la última es una llamada a función. Al ejecutar el programa, cada una de estas sentencias es ejecutada, una después de la otra, una sola vez.

Además de las sentencias simples, que son ejecutadas en secuencia, existen las **sentencias de control** que permiten modificar el flujo del programa introduciendo ciclos y condicionales.

Un **condicional** es un conjunto de sentencias que pueden o no ejecutarse, dependiendo del resultado de una condición.

Un **ciclo** es un conjunto de sentencias que son ejecutadas varias veces, hasta que una condición de término es satisfecha.

Tanto los condicionales como los ciclos contienen a otras sentencias. Para indicar esta relación se utiliza la **indentación**: las sentencias contenidas no se escriben en la misma columna que la sentencia de control, sino un poco más a la derecha:

n = int(raw\_input())

m = int(raw\_input())

**if** m < n:

t = m

m = n

n = t

**print** m, n

En este ejemplo, las tres asignaciones están contenidas dentro de la sentencia de control if. El print m, n no está indentado, por lo que no es parte de la sentencia if.

Este programa tiene cuatro sentencias, de las cuales la tercera es una sentencia de control, que contiene a otras tres sentencias.

Para indentar, utilizaremos siempre cuatro espacios.

**Condicional if**

La sentencia **if** («si») ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, no se hace nada:

La sintaxis es la siguiente:

if condición:

sentencias

Por ejemplo, el siguente programa felicita a alguien que aprobó la asignatura:

nota = int(raw\_input('Ingrese su nota: '))

**if** nota >= 55:

**print** 'Felicitaciones'

Ejecute este programa, probando varias veces con valores diferentes.

**Condicional if-else**

La sentencia **if-else** («si-o-si-no») decide qué instrucciones ejecutar dependiendo si una condición es verdadera o falsa:

La sintaxis es la siguiente:

if condición:

qué hacer cuando la condición es verdadera

else

qué hacer cuando la condición es falsa

Por ejemplo, el siguiente programa indica a alguien si es mayor de edad:

edad = int(raw\_input('Cual es su edad? '))

**if** edad < 18:

**print** 'Usted es menor de edad'

**else**:

**print** 'Usted es adulto'

El siguiente programa realiza acciones distintas dependiendo de si el número de entrada es par o impar:

n = int(raw\_input('Ingrese un numero: '))

**if** n % 2 == 0:

**print** 'El numero es par'

**print** 'La mitad del numero es', n / 2

**else**:

**print** 'El numero es impar'

**print** 'El sucesor del numero es', n + 1

**print** 'Listo'

La última sentencia no está indentada, por lo que no es parte del condicional, y será ejecutada siempre.

**Condicional if-elif-else**

La sentencia **if-elif-else** depende de dos o más condiciones, que son evaluadas en orden. La primera que es verdadera determina qué instrucciones serán ejecutadas:

La sintaxis es la siguiente:

if condición1:

qué hacer si condición1 es verdadera

elif condición2:

qué hacer si condición2 es verdadera

...

else:

qué hacer cuando ninguna de las

condiciones anteriores es verdadera

El último else es opcional.

Por ejemplo, la tasa de impuesto a pagar por una persona según su sueldo puede estar dada por la siguiente tabla:

|  |  |
| --- | --- |
| **Sueldo** | **tasa de impuesto** |
| menos de 1000 | 0% |
| 1000 ≤ sueldo < 2000 | 5% |
| 2000 ≤ sueldo < 4000 | 10% |
| 4000 o más | 12% |

Entonces, el programa que calcula el impuesto a pagar es el siguiente:

sueldo = int(raw\_input('Ingrese su sueldo: '))

**if** sueldo < 1000:

tasa = 0.00

**elif** sueldo < 2000:

tasa = 0.05

**elif** sueldo < 4000:

tasa = 0.10

**else**:

tasa = 0.12

**print** 'Usted debe pagar', tasa \* sueldo, 'de impuesto'

Siempre sólo una de las alternativas será ejecutada. Apenas una de las condiciones es verdadera, el resto de ellas no siguen siendo evaluadas.

Otra manera de escribir el mismo programa usando sólo sentencias if es

la siguiente:

sueldo = int(raw\_input('Ingrese su sueldo: '))

**if** sueldo < 1000:

tasa = 0.00

**if** 1000 <= sueldo < 2000:

tasa = 0.05

**if** 2000 <= sueldo < 4000:

tasa = 0.10

**if** 4000 < sueldo:

tasa = 0.12

**print** 'Usted debe pagar', tasa \* sueldo, 'de impuesto'

Esta manera es menos clara, porque no es evidente a primera vista que sólo una de las condiciones será verdadera.

**Ciclo while**

El ciclo **while** («mientras») ejecuta una secuencia de instrucciones mientras una condición sea verdadera:

Cada una de las veces que el cuerpo del ciclo es ejecutado se llama **iteración**.

La condición es evaluada antes de cada iteración. Si la condición es inicialmente falsa, el ciclo no se ejecutará ninguna vez.

La sintaxis es la siguiente:

while condición:

sentencias

Por ejemplo, el siguiente programa multiplica dos números enteros sin usar el operador \*:

m = int(raw\_input())

n = int(raw\_input())

p = 0

**while** m > 0:

m = m - 1

p = p + n

**print** 'El producto de m y n es', p

Para ver cómo funciona este programa, hagamos un ruteo con la entrada m = 4 y n = 7:

|  |  |  |
| --- | --- | --- |
| **p** | **m** | **N** |
|  | 4 |  |
|  |  | 7 |
| 0 |  |  |
|  | 3 |  |
| 7 |  |  |
|  | 2 |  |
| 14 |  |  |
|  | 1 |  |
| 21 |  |  |
|  | 0 |  |
| 28 |  |  |

En cada iteración, el valor de m decrece en 1. Cuando llega a 0, la condición del while deja de ser verdadera por lo que el ciclo termina. De este modo, se consigue que el resultado sea sumar m veces el valor de n.

Note que el ciclo no termina apenas el valor de m pasa a ser cero. La condición es evaluada una vez que la iteración completa ha terminado.

En general, el ciclo while se utiliza cuando no es posible saber de antemano cuántas veces será ejecutado el ciclo, pero sí qué es lo que tiene que ocurrir para que se termine.

**Ciclo for con rango**

El ciclo **for con rango** ejecuta una secuencia de sentencias una cantidad fija de veces.

Para llevar la cuenta, utiliza una **variable de control** que toma valores distintos en cada iteración.

Una de las sintaxis para usar un for con rango es la siguiente:

for variable in range(fin):

qué hacer para cada valor de la variable de control

En la primera iteración, la variable de control toma el valor 0. Al final de cada iteración, el valor de la variable aumenta automáticamente. El ciclo termina justo antes que la variable tome el valor fin.

Por ejemplo, el siguiente programa muestra los cubos de los números del 0 al 20:

**for** i **in** range(21):

**print** i, i \*\* 3

Un **rango** es una sucesión de números enteros equiespaciados. Incluyendo la presentada más arriba, hay tres maneras de definir un rango:

range(final)

range(inicial, final)

range(inicial, final, incremento)

El valor inicial siempre es parte del rango. El valor final nunca es parte del rango. El incremento indica la diferencia entre dos valores consecutivos del rango.

Si el valor inicial es omitido, se supone que es 0. Si el incremento es omitido, se supone que es 1.

Con algunos ejemplos quedará más claro:

|  |  |
| --- | --- |
| range(9) | 0, 1, 2, 3, 4, 5, 6, 7, 8 |
| range(3, 13) | 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 |
| range(3, 13, 2) | 3, 5, 7, 9, 11 |
| range(11, 4) | ningún valor |
| range(11, 4, -1) | 11, 10, 9, 8, 7, 6, 5 |

Usando un incremento negativo, es posible hacer ciclos que van hacia atrás:

**for** i **in** range(10, 0, -1):

**print** i

**print** 'Feliz anno nuevo!'

En general, el ciclo for con rango se usa cuando el número de iteraciones es conocido antes de entrar al ciclo.

**Salir de un ciclo**

Además de las condiciones de término propias de los ciclos while y for, siempre es posible salir de un ciclo en medio de una iteración usando la sentencia **break**. Lo lógico es que sea usada dentro de un if, para evitar que el ciclo termine prematuramente en la primera iteración:

Por ejemplo, en el programa para determinar si un número es primo o no, la búsqueda de divisores puede ser terminada prematuramente apenas se encuentra el primero de ellos:

es\_primo = True

**for** d **in** range(2, n):

**if** n % d == 0:

es\_primo = False

**break**

**Saltar a la siguiente iteración**

La sentencia **continue** se usa para saltar a la iteración siguiente sin llegar al final de la que está en curso.

Por ejemplo, el siguiente programa muestra el seno, el coseno y la tangente de los números del 1 al 30, pero omitiendo los que terminan en 7:

**from** **math** **import** sin, cos, tan

**for** i **in** range(1, 31):

**if** i % 10 == 7:

**continue**

**print** i, sin(i), cos(i), tan(i)

**Patrones comunes**

Como hemos visto hasta ahora, los programas son una combinación de asignaciones, condicionales y ciclos, organizados de tal manera que describan el algoritmo que queremos ejecutar.

Existen algunas tareas muy comunes y que casi siempre se resuelven de la misma manera. Por lo tanto, es conveniente conocerlas.

En programación, se llama **patrón** a una solución que es aplicable a un problema que ocurre a menudo. A continuación veremos algunos patrones comunes que ocurren en programación.

**Sumar y multiplicar cosas**

La suma y la multiplicación son operaciones binarias: operan sobre dos valores.

Para sumar y multiplicar más valores, generalmente dentro de un ciclo que los vaya generando, hay que usar una variable para ir guardando el resultado parcial de la operación. Esta variable se llama **acumulador**.

En el caso de la suma, el acumulador debe partir con el valor cero. Para la multiplicación, con el valor uno. En general, el acumulador debe ser inicializado con el [elemento neutro](http://es.wikipedia.org/wiki/Elemento_neutro) de la operación que será aplicada.

Por ejemplo, el siguiente programa entrega el producto de los mil primeros números naturales:

producto = 1

**for** i **in** range(1, 1001):

producto = producto \* i

**print** producto

El siguiente programa entrega la suma de los cubos de los números naturales cuyo cuadrado es menor que mil:

i = 1

suma = 0

**while** i \*\* 2 < 1000:

valor = i \*\* 3

i = i + 1

suma = suma + valor

**print** suma

En todos los casos, el patrón a seguir es algo como esto:

acumulador = valor\_inicial

ciclo:

valor = ...

...

acumulador = acumulador operación valor

El cómo adaptar esta plantilla a cada situación de modo que entregue el resultado correcto es responsabilidad del programador.

**Contar cosas**

Para contar cuántas veces ocurre algo, hay que usar un acumulador, al que se le suele llamar **contador**.

Tal como en el caso de la suma, debe ser inicializado en cero, y cada vez que aparezca lo que queremos contar, hay que incrementarlo en uno.

Por ejemplo, el siguiente programa cuenta cuántos de los números naturales menores que mil tienen un cubo terminado en siete:

c = 0

**for** i **in** range(1000):

ultimo\_digito = (i \*\* 3) % 10

**if** ultimo\_digito == 7:

c = c + 1

**print** c

**Encontrar el mínimo y el máximo**

Para encontrar el máximo de una secuencia de valores, hay que usar un acumulador para recordar cuál es el mayor valor visto hasta el momento. En cada iteración, hay que examinar cuál es el valor actual, y si es mayor que el máximo, actualizar el acumulador.

El acumulador debe ser inicializado con un valor que sea menor a todos los valores que vayan a ser examinados.

Por ejemplo, el siguiente programa pide al usuario que ingrese diez números enteros positivos, e indica cuál es el mayor de ellos:

**print** 'Ingrese diez numeros positivos'

mayor = -1

**for** i **in** range(10):

n = int(raw\_input())

**if** n > mayor:

mayor = n

**print** 'El mayor es', mayor

Otra manera de hacerlo es reemplazando esta parte:

**if** n > mayor:

mayor = n

por ésta:

mayor = max(mayor, n)

En este caso, como todos los números ingresados son positivos, inicializamos el acumulador en -1, que es menor que todos los valores posibles, por lo que el que sea el mayor eventualmente lo reemplazará.

¿Qué hacer cuando no exista un valor inicial que sea menor a todas las entradas posibles? Una solución es poner un número «muy negativo», y rezar para que el usuario no ingrese uno menor que él. Esta no es la mejor solución, ya que no cubre todos los casos posibles:

mayor = -999999999

**for** i **in** range(10):

n = int(raw\_input())

mayor = max(mayor, n)

Una opción más robusta es usar el primero de los valores por examinar:

mayor = int(raw\_input()) *# preguntar el primer valor*

**for** i **in** range(9): *# preguntar los nueve siguientes*

n = int(raw\_input())

mayor = max(mayor, n)

La otra buena solución es usar explícitamente el valor −∞−∞, que en Python puede representarse usando el tipo float de la siguiente manera:

mayor = -float('inf') *# asi se dice "infinito" en Python*

**for** i **in** range(10):

n = int(raw\_input())

mayor = max(mayor, n)

Si sabemos de antemano que todos los números por revisar son positivos, podemos simplemente inicializar el acumulador en -1.

Por supuesto, para obtener el menor valor se hace de la misma manera, pero inicializando el acumulador con un número muy grande, y actualizándolo al encontrar un valor menor.

**Generar pares**

Para generar pares de cosas en un programa, es necesario usar dos ciclos anidados (es decir, uno dentro del otro).

Ambos ciclos, el exterior y el interior, van asignando valores a sus variables de control, y ambas son accesibles desde dentro del doble ciclo.

Por ejemplo, todas las casillas de un tablero de ajedrez pueden ser identificadas mediante un par (fila, columna). Para recorrer todas las casillas del tablero, se puede hacer de la siguiente manera:

**for** i **in** range(1, 9):

**for** i **in** range(1, 9):

**print** 'Casilla', i, j

Cuando los pares son desordenados (es decir, el par (a,b)(a,b) es el mismo que el par (b,a)(b,a)), el ciclo interior no debe partir desde cero, sino desde el valor que tiene la variable de control del ciclo interior.

Por ejemplo, el siguiente programa muestra todas las piezas de un juego de dominó:

**for** i **in** range(7):

**for** j **in** range(i, 7):

**print** i, j

Además, otros tipos de restricciones pueden ser necesarias. Por ejemplo, en un campeonato de fútbol, todos los equipos deben jugar entre ellos dos veces, una como local y una como visita. Por supuesto, no pueden jugar consigo mismos, por lo que es necesario excluir los pares compuestos por dos valores iguales. El siguiente programa muestra todos los partidos que se deben jugar en un campeonato con 6 equipos, suponiendo que los equipos están numerados del 0 al 5:

**for** i **in** range(6):

**for** j **in** range(6):

**if** i != j:

**print** i, j

Otra manera de escribir el mismo código es:

**for** i **in** range(6):

**for** j **in** range(6):

**if** i == j:

**continue**

**print** i, j

**Funciones**

Supongamos que necesitamos escribir un programa que calcule el [número combinatorio](http://es.wikipedia.org/wiki/N%C3%BAmero_combinatorio) C(m,n)C(m,n), definido como:

C(m,n)=m!(m−n)!n!,C(m,n)=m!(m−n)!n!,

donde n!n! (el [factorial](http://es.wikipedia.org/wiki/Factorial) de nn) es el producto de los números enteros desde 1 hasta nn:

n!=1⋅2⋅⋯⋅(n−1)⋅n=∏i=1nin!=1⋅2⋅⋯⋅(n−1)⋅n=∏i=1ni

El código para calcular el factorial de un número entero nn es sencillo:

f = 1

**for** i **in** range(1, n + 1):

f \*= i

Sin embargo, para calcular el número combinatorio, hay que hacer lo mismo tres veces:

comb = 1

*# multiplicar por m!*

f = 1

**for** i **in** range(1, m + 1):

f = f \* i

comb = comb \* f

*# dividir por (m - n)!*

f = 1

**for** i **in** range(1, m - n + 1):

f = f \* i

comb = comb / f

*# dividir por n!*

f = 1

**for** i **in** range(1, n + 1):

f = f \* i

comb = comb / f

La única diferencia entre los tres cálculos de factoriales es el valor de término de cada ciclo for (m, m - n y n, respectivamente).

Escribir el mismo código varias veces es tedioso y propenso a errores. Además, el código resultante es mucho más dificil de entender, pues no es evidente a simple vista qué es lo que hace.

Lo ideal sería que existiera una función llamada factorial que hiciera el trabajo sucio, y que pudiéramos usar de la siguiente manera:

factorial(m) / (factorial(m - n) \* factorial(n))

Ya vimos anteriormente que Python ofrece «de fábrica» algunas funciones, como int, min y abs. Ahora veremos cómo crear nuestras propias funciones.

**Funciones**

En programación, una **función** es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:

* los **parámetros**, que son los valores que recibe la función como entrada;
* el **código de la función**, que son las operaciones que hace la función; y
* el **resultado** (o **valor de retorno**), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

En el ejemplo del factorial, el parámetro es el entero al que queremos calcularle el factorial, el código es el ciclo que hace las multiplicaciones, y el resultado es el valor calculado.

***Definición de funciones***

Las funciones en Python son creadas mediante la sentencia def:

def nombre(parámetros):

# código de la función

Los parámetros son variables en las que quedan almacenados los valores de entrada.

La función contiene código igual al de cualquier programa. La diferencia es que, al terminar, debe entregar su resultado usando la sentencia return.

Por ejemplo, la función para calcular el factorial puede ser definida de la siguiente manera:

**def** factorial(n):

f = 1

**for** i **in** range(1, n + 1):

f \*= i

**return** f

En este ejemplo, el resultado que entrega una llamada a la función es el valor que tiene la variable f al llegar a la última línea de la función.

Una vez creada, la función puede ser usada como cualquier otra, todas las veces que sea necesario:

**>>>** factorial(0)

1

**>>>** factorial(12) + factorial(10)

482630400

**>>>** factorial(factorial(3))

720

**>>>** n = 3

**>>>** factorial(n \*\* 2)

362880

Las variables que son creadas dentro de la función (incluyendo los parámetros y el resultado) se llaman **variables locales**, y sólo son visibles dentro de la función, no desde el resto del programa.

Por otra parte, las variables creadas fuera de alguna función se llaman **variables globales**, y son visibles desde cualquier parte del programa. Sin embargo, su valor no puede ser modificado, ya que una asignación crearía una variable local del mismo nombre.

En el ejemplo, las variables locales son n, f e i. Una vez que la llamada a la función termina, estas variables dejan de existir:

**>>>** factorial(5)

120

**>>>** f

Traceback (most recent call last):

File "<console>", line 1, in <module>

NameError: name 'f' is not defined

Después de definir la función factorial, podemos crear otra función llamada comb para calcular números combinatorios:

**def** comb(m, n):

fact\_m = factorial(m)

fact\_n = factorial(n)

fact\_m\_n = factorial(m - n)

c = fact\_m / (fact\_n \* fact\_m\_n)

**return** c

Esta función llama a factorial tres veces, y luego usa los resultados para calcular su resultado. La misma función puede ser escrita también de forma más sucinta:

**def** comb(m, n):

**return** factorial(m) / (factorial(n) \* factorial(m - n))

El programa completo es el siguiente:

(Puede descargarlo [aquí](http://progra.usm.cl/apunte/_static/programas/combinatorios.py)).

Note que, gracias al uso de las funciones, la parte principal del programa ahora tiene sólo cuatro líneas, y es mucho más fácil de entender.

**Múltiples valores de retorno**

En Python, una función puede retornar más de un valor.

Por ejemplo, la siguiente función recibe una cantidad de segundos, y retorna el equivalente en horas, minutos y segundos:

**def** convertir\_segundos(segundos):

horas = segundos / (60 \* 60)

minutos = (segundos / 60) % 60

segundos = segundos % 60

**return** horas, minutos, segundos

Al llamar la función, se puede asignar un nombre a cada uno de los valores retornados:

**>>>** h, m, s = convertir\_segundos(9814)

**>>>** h

2

**>>>** m

43

**>>>** s

34

Técnicamente, la función está retornando una **tupla** de valores, un tipo de datos que veremos más adelante:

**>>>** convertir\_segundos(9814)

(2, 43, 34)

**Funciones que no retornan nada**

Una función puede realizar acciones sin entregar necesariamente un resultado.

Por ejemplo, si un programa necesita imprimir cierta información muchas veces, conviene encapsular esta acción en una función que haga los print

**def** imprimir\_datos(nombre, apellido, rol, dia, mes, anno):

**print** 'Nombre completo:', nombre, apellido

**print** 'Rol:', rol

**print** 'Fecha de nacimiento:', dia, '/', mes, '/', anno

imprimir\_datos('Perico', 'Los Palotes', '201101001-1', 3, 1, 1993)

imprimir\_datos('Yayita', 'Vinagre', '201101002-2', 10, 9, 1992)

imprimir\_datos('Fulano', 'De Tal', '201101003-3', 14, 5, 1990)

En este caso, cada llamada a la función imprimir\_datos muestra los datos en la pantalla, pero no entrega un resultado. Este tipo de funciones son conocidas en programación como **procedimientos** o **subrutinas**, pero en Python son funciones como cualquier otra.

Técnicamente, todas las funciones retornan valores. En el caso de las funciones que no tienen una sentencia return, el valor de retorno siempre es None. Pero como la llamada a la función no aparece en una asignación, el valor se pierde, y no tiene ningún efecto en el programa.

**Módulos**

Un **módulo** (o **biblioteca**) es una colección de definiciones de variables, funciones y tipos (entre otras cosas) que pueden ser importadas para ser usadas desde cualquier programa.

Ya hemos visto algunos ejemplos de cómo usar módulos, particularmente el módulo matemático, del que podemos importar funciones como la exponencial y el coseno, y las constantes π y *e*:

**>>> from** **math** **import** exp, cos

**>>> from** **math** **import** pi, e

**>>> print** cos(pi / 3)

0.5

Las ventajas de usar módulos son:

* las funciones y variables deben ser definidas sólo una vez, y luego pueden ser utilizadas en muchos programas sin necesidad de reescribir el código;
* permiten que un programa pueda ser organizado en varias secciones lógicas, puestas cada una en un archivo separado;
* hacen más fácil compartir componentes con otros programadores.

Python viene «de fábrica» con muchos módulos listos para ser usados. Además, es posible descargar de internet e instalar módulos prácticamente para hacer cualquier cosa. Por último, aprenderemos a crear nuestros propios módulos.

**Módulos presentes en Python**

Éstos son algunos de los módulos estándares de Python, que pueden ser usado desde cualquier programa.

El módulo [math](http://docs.python.org/library/math.html) contiene funciones y constantes matemáticas:

**>>> from** **math** **import** floor, radians

**>>>** floor(-5.9)

-6.0

**>>>** radians(180)

3.1415926535897931

El módulo [random](http://docs.python.org/library/random.html) contiene funciones para producir números aleatorios (es decir, al azar):

**>>> from** **random** **import** choice, randrange,

**>>>** choice(['cara', 'sello'])

'cara'

**>>>** choice(['cara', 'sello'])

'sello'

**>>>** choice(['cara', 'sello'])

'sello'

**>>>** randrange(10)

7

**>>>** randrange(10)

2

**>>>** randrange(10)

5

**>>>** r = range(5)

**>>>** r

[0, 1, 2, 3, 4]

**>>>** shuffle(r)

**>>>** r

[4, 2, 0, 3, 1]

El módulo [datetime](http://docs.python.org/library/datetime.html) provee tipos de datos para manipular fechas y horas:

**>>> from** **datetime** **import** date

**>>>** hoy = date(2011, 5, 31)

**>>>** fin\_del\_mundo = date(2012, 12, 21)

**>>>** (fin\_del\_mundo - hoy).days

570

El módulo [fractions](http://docs.python.org/library/fractions.html) provee un tipo de datos para representar números racionales:

**>>> from** **fractions** **import** Fraction

**>>>** a = Fraction(5, 12)

**>>>** b = Fraction(9, 7)

**>>>** a + b

Fraction(143, 84)

El módulo [turtle](http://docs.python.org/library/turtle.html) permite manejar una tortuga (¡haga la prueba!):

**>>> from** **turtle** **import** Turtle

**>>>** t = Turtle()

**>>>** t.forward(10)

**>>>** t.left(45)

**>>>** t.forward(20)

**>>>** t.left(45)

**>>>** t.forward(30)

**>>> for** i **in** range(10):

**...** t.right(30)

**...** t.forward(10 \* i)

**...**

>>>

La lista completa de módulos de Python puede ser encontrada en la [documentación de la biblioteca estándar](http://docs.python.org/library/index.html).

**Importación de nombres**

La sentencia import importa objetos desde un módulo para poder ser usados en el programa actual.

Una manera de usar import es importar sólo los nombres específicos que uno desea utilizar en el programa:

**from** **math** **import** sin, cos

**print** sin(10)

**print** cos(20)

En este caso, las funciones sin y cos no fueron creadas por nosotros, sino importadas del módulo de matemáticas, donde están definidas.

La otra manera de usar import es importando el módulo completo, y accediendo a sus objetos mediante un punto:

**import** **math**

**print** math.sin(10)

**print** math.cos(10)

Las dos formas son equivalentes. Como siempre, hay que optar por la que hace que el programa sea más fácil de entender.

**Creación de módulos**

Un módulo sencillo es simplemente un archivo con código en Python. El nombre del archivo indica cuál es el nombre del módulo.

Por ejemplo, podemos crear un archivo llamado pares.py que tenga funciones relacionadas con los números pares:

**def** es\_par(n):

**return** n % 2 == 0

**def** es\_impar(n):

**return** **not** es\_par(n)

**def** pares\_hasta(n):

**return** range(0, n, 2)

En este caso, el nombre del módulo es pares. Para poder usar estas funciones desde otro programa, el archivo pares.py debe estar en la misma carpeta que el programa.

Por ejemplo, el programa mostrar\_pares.py puede ser escrito así:

**from** **pares** **import** pares\_hasta

n = int(raw\_input('Ingrese un entero: '))

**print** 'Los numeros pares hasta', n, 'son:'

**for** i **in** pares\_hasta(n):

**print** i

Y el programa ver\_si\_es\_par.py puede ser escrito así:

**import** **pares**

n = int(raw\_input('Ingrese un entero: '))

**if** pares.es\_par(n):

**print** n, 'es par'

**else**:

**print** n, 'no es par'

Como se puede ver, ambos programas pueden usar los objetos definidos en el módulo simplemente importándolos.

**Usar módulos como programas**

Un archivo con extensión .py puede ser un módulo o un programa. Si es un módulo, contiene definiciones que pueden ser importadas desde un programa o desde otro módulo. Si es un programa, contiene código para ser ejecutado.

A veces, un programa también contiene definiciones (por ejemplo, funciones y variables) que también pueden ser útiles desde otro programa. Sin embargo, no pueden ser importadas, ya que al usar la sentencia import el programa completo sería ejecutado. Lo que ocurriría en este caso es que, al ejecutar el segundo programa, también se ejecutaría el primero.

Existe un truco para evitar este problema: siempre que hay código siendo ejecutado, existe una variable llamada \_\_name\_\_. Cuando se trata de un programa, el valor de esta variable es '\_\_main\_\_', mientras que en un módulo, es el nombre del módulo.

Por lo tanto, se puede usar el valor de esta variable para marcar la parte del programa que debe ser ejecutada al ejecutar el archivo, pero no al importarlo.

Por ejemplo, el siguiente programa convierte unidades de medidas de longitud:

Este programa es útil por sí solo, pero además sus cuatro funciones y las constantes km\_por\_milla y cm\_por\_pulgada podrían ser útiles para ser usadas en otro programa.

Al poner el cuerpo del programa dentro del if \_\_name\_\_ == '\_\_main\_\_', el archivo puede ser usado como un módulo. Si no hiciéramos esto, cada vez que otro programa importe una función se ejecutaría el programa completo.

Haga la prueba: [descargue el programa](http://progra.usm.cl/apunte/_static/programas/conversion_unidades.py) y ejecútelo. Luego, escriba otro programa que importe alguna de las funciones. A continuación, haga lo mismo, pero eliminando el if.

**Listas**

Una **lista** es una colección ordenada de valores. Una lista puede contener cualquier cosa.

En Python, el tipo de datos que representa a las listas se llama list.

**Cómo crear listas**

Las dos maneras principales de crear una lista son:

* usar una lista literal, con los valores entre corchetes:
* **>>>** primos = [2, 3, 5, 7, 11]
* **>>>** primos
* [2, 3, 5, 7, 11]
* **>>>** []
* []
* **>>>** [1.0 + 2.0, 3.0 + 4.0 + 5.0]
* [3.0, 12.0]
* **>>>** ['hola ' + 'mundo', 24 \* 7, True **or** False]
* ['hola mundo', 168, True]
* usar la función list aplicada sobre un iterable:
* **>>>** list('hola')
* ['h', 'o', 'l', 'a']
* **>>>** list(range(10))
* [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
* **>>>** list()
* []

**Operaciones sobre listas**

len(l) entrega el largo de la lista; es decir, cuántos elementos tiene:

**>>>** colores = ['azul', 'rojo', 'verde', 'amarillo']

**>>>** len(colores)

4

**>>>** len([True, True, True])

3

**>>>** len([])

0

l[i] entrega el i-ésimo valor de la lista. El valor i se llama **índice** del valor. Al igual que para los strings, los índices parten de cero:

**>>>** colores = ['azul', 'rojo', 'verde', 'amarillo']

**>>>** colores[0]

'azul'

**>>>** colores[3]

'amarillo'

Además, es posible modificar el valor del i-ésimo elemento:

**>>>** colores[1] = 'negro'

**>>>** colores

['azul', 'negro', 'verde', 'amarillo']

Si el índice i indica un elemento que no está en la lista, ocurre un **error de índice**:

**>>>** colores[4]

Traceback (most recent call last):

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

IndexError: list index out of range

Si el índice es negativo, los elementos se cuentan desde el final hacia atrás:

**>>>** colores[-1]

'amarillo'

**>>>** colores[-4]

'azul'

l.append(x) agrega el elemento x al final de la lista:

**>>>** primos = [2, 3, 5, 7, 11]

**>>>** primos.append(13)

**>>>** primos.append(17)

**>>>** primos

[2, 3, 5, 7, 11, 13, 17]

Un comentario al margen: append es un **método**. Los métodos son funciones que están dentro de un objeto. Cada lista tiene su propia función append. Es importante tener esta distinción clara, ya que hay operaciones que están implementadas como funciones y otras como métodos.

sum(x) entrega la suma de los valores de la lista:

**>>>** sum([1, 2, 1, -1, -2])

1

**>>>** sum([])

0

l1 + l2 concatena las listas l1 y l2:

**>>>** list('perro') + [2, 3, 4]

['p', 'e', 'r', 'r', 'o', 2, 3, 4]

l \* n repite n veces la lista l:

**>>>** [3.14, 6.28, 9.42] \* 2

[3.14, 6.28, 9.42, 3.14, 6.28, 9.42]

**>>>** [3.14, 6.28, 9.42] \* 0

[]

Para saber si un elemento x está en la lista l, se usa x in l. La versión negativa de in es not in:

**>>>** r = range(0, 20, 2)

**>>>** r

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

**>>>** 12 **in** r

True

**>>>** 15 **in** r

False

**>>>** 15 **not** **in** r

True

* l[i:j] es el operador de rebanado, que entrega una nueva lista que tiene desde el i-ésimo hasta justo antes del j-ésimo elemento de la lista l:

**>>>** x = [1.5, 3.3, 8.4, 3.1, 2.9]

**>>>** x[2:4]

[8.4, 3.1]

l.count(x) cuenta cuántas veces está el elemento x en la lista:

**>>>** letras = list('paralelepipedo')

**>>>** letras.count('p')

3

l.index(x) entrega cuál es el índice del valor x:

**>>>** colores = ['azul', 'rojo', 'verde', 'amarillo']

**>>>** colores.index('verde')

2

**>>>** colores.index('fucsia')

Traceback (most recent call last):

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

ValueError: 'fucsia' is not in list

l.remove(x) elimina el elemento x de la lista:

**>>>** l = [7, 0, 3, 9, 8, 2, 4]

**>>>** l.remove(2)

**>>>** l

[7, 0, 3, 9, 8, 4]

del l[i] elimina el i-ésimo elemento de la lista:

**>>>** l = [7, 0, 3, 9, 8, 2, 4]

**>>> del** l[2]

**>>>** l

[7, 0, 9, 8, 2, 4]

l.reverse() invierte la lista:

**>>>** l = [7, 0, 3, 9, 8, 2, 4]

**>>>** l.reverse()

**>>>** l

[4, 2, 8, 9, 3, 0, 7]

l.sort() ordena la lista:

**>>>** l = [7, 0, 3, 9, 8, 2, 4]

**>>>** l.sort()

**>>>** l

[0, 2, 3, 4, 7, 8, 9]

Para todas estas operaciones, siempre hay que tener muy claro si la lista es modificada o no. Por ejemplo, el rebanado no modifica la lista, sino que crea una nueva:

**>>>** ramos = ['Progra', 'Mate', 'Fisica', 'Ed.Fisica']

**>>>** ramos[:2]

['Progra', 'Mate']

**>>>** len(ramos) *# la lista sigue teniendo cuatro elementos*

4

**Iteración sobre una lista**

Una lista es un objeto **iterable**. Esto significa que sus valores se pueden recorrer usando un ciclo for:

valores = [6, 1, 7, 8, 9]

**for** i **in** valores:

**print** i \*\* 2

En cada iteración del for, la variable i toma uno de los valores de la lista, por lo que este programa imprime los siguientes valores:

36

1

49

64

81

**Tuplas**

Una **tupla** es una secuencia de valores agrupados.

Una tupla sirve para agrupar, como si fueran un único valor, varios valores que, por su naturaleza, deben ir juntos.

El tipo de datos que representa a las tuplas se llama tuple. El tipo tuple es inmutable: una tupla no puede ser modificada una vez que ha sido creada.

Una tupla puede ser creada poniendo los valores separados por comas y entre paréntesis. Por ejemplo, podemos crear una tupla que tenga el nombre y el apellido de una persona:

**>>>** persona = ('Perico', 'Los Palotes')

**>>>** persona

('Perico', 'Los Palotes')

**Desempaquetado de tuplas**

Los valores individuales de una tupla pueden ser recuperados asignando la tupla a las variables respectivas. Esto se llama **desempaquetar la tupla** (en inglés: *unpack*):

**>>>** nombre, apellido = persona

**>>>** nombre

'Perico'

Si se intenta desempaquetar una cantidad incorrecta de valores, ocurre un error de valor:

**>>>** a, b, c = persona

Traceback (most recent call last):

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

ValueError: need more than 2 values to unpack

Además, también es posible extraer los valores usando su índice, al igual que con las listas:

**>>>** persona[1]

'Los Palotes'

A diferencia de las listas, los elementos no se pueden modificar:

**>>>** persona[1] = 'Smith'

Traceback (most recent call last):

File "<console>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

**Comparación de tuplas**

Dos tuplas son iguales cuando tienen el mismo tamaño y cada uno de sus elementos correspondientes tienen el mismo valor:

**>>>** (1, 2) == (3 / 2, 1 + 1)

True

**>>>** (6, 1) == (6, 2)

False

**>>>** (6, 1) == (6, 1, 0)

False

Para determinar si una tupla es menor que otra, se utiliza lo que se denomina **orden lexicográfico**. Si los elementos en la primera posición de ambas tuplas son distintos, ellos determinan el ordenamiento de las tuplas:

**>>>** (1, 4, 7) < (2, 0, 0, 1)

True

**>>>** (1, 9, 10) < (0, 5)

False

La primera comparación es True porque 1 < 2. La segunda comparación es False porque 1 > 0. No importa el valor que tengan los siguientes valores, o si una tupla tiene más elementos que la otra.

Si los elementos en la primera posición son iguales, entonces se usa el valor siguiente para hacer la comparación:

**>>>** (6, 1, 8) < (6, 2, 8)

True

**>>>** (6, 1, 8) < (6, 0)

False

La primera comparación es True porque 6 == 6 y 1 < 2. La segunda comparación es False porque 6 == 6 y 1 > 0.

Si los elementos respectivos siguen siendo iguales, entonces se sigue probando con los siguientes uno por uno, hasta encontrar dos distintos. Si a una tupla se le acaban los elementos para comparar antes que a la otra, entonces es considerada menor que la otra:

**>>>** (1, 2) < (1, 2, 4)

True

**>>>** (1, 3) < (1, 2, 4)

False

La primera compación es True porque 1 == 1, 2 == 2, y ahí se acaban los elementos de la primera tupla. La segunda comparación es False porque 1 == 1 y 3 < 2; en este caso sí se alcanza a determinar el resultado antes que se acaben los elementos de la primera tupla.

Este método de comparación es el mismo que se utiliza para poner palabras en orden alfabético (por ejemplo, en guías telefónicas y diccionarios):

**>>>** 'auto' < 'auxilio'

True

**>>>** 'auto' < 'autos'

True

**>>>** 'mes' < 'mesa' < 'mesadas' < 'mesas' < 'meses' < 'mi'

True

**Usos típicos de las tuplas**

Las tuplas se usan siempre que es necesario agrupar valores. Generalmente, conceptos del mundo real son representados como tuplas que agrupan información sobre ellos. Por ejemplo, un partido de fútbol se puede representar como una tupla de los equipos que lo juegan:

partido1 = ('Milan', 'Bayern')

Para representar puntos en el plano, se puede usar tuplas de dos elementos (x, y). Por ejemplo, podemos crear una función distancia que recibe dos puntos y entrega la distancia entre ellos:

**def** distancia(p1, p2):

x1, y1 = p1

x2, y2 = p2

dx = x2 - x1

dy = y2 - y1

**return** (dx \*\* 2 + dy \*\* 2) \*\* 0.5

Al llamar a la función, se le debe pasar dos tuplas:

**>>>** a = (2, 3)

**>>>** b = (7, 15)

**>>>** distancia(a, b)

13.0

Las fechas generalmente se representan como tuplas agrupando el año, el mes y el día. La ventaja de hacerlo en este orden (el año primero) es que las operaciones relacionales permiten saber en qué orden ocurrieron las fechas:

**>>>** hoy = (2011, 4, 19)

**>>>** ayer = (2011, 4, 18)

**>>>** navidad = (2011, 12, 25)

**>>>** anno\_nuevo = (2012, 1, 1)

**>>>** hoy < ayer

False

**>>>** hoy < navidad < anno\_nuevo

True

Una tupla puede contener otras tuplas. Por ejemplo, una persona puede ser descrita por su nombre, su rut y su fecha de nacimiento:

persona = ('Perico Los Palotes', '12345678-9', (1980, 5, 14))

En este caso, los datos se pueden desempaquetar así:

**>>>** nombre, rut, (a, m, d) = persona

**>>>** m

5

A veces a uno le interesa sólo uno de los valores de la tupla. Para evitar crear variables innecesarias, se suele asignar estos valores a la variable \_. Por ejemplo, si sólo nos interesa el mes en que nació la persona, podemos obtenerlo así:

**>>>** \_, \_, (\_, mes, \_) = persona

**>>>** mes

5

Una tabla de datos generalmente se representa como una lista de tuplas. Por ejemplo, la información de los alumnos que están tomando un ramo puede ser representada así:

alumnos = [

('Perico', 'Los Palotes', '201199001-5', 'Civil'),

('Fulano', 'De Tal', '201199002-6', 'Electrica'),

('Fulano', 'De Tal', '201199003-7', 'Mecanica'),

]

En este caso, se puede desempaquetar los valores automáticamente al recorrer la lista en un ciclo for:

**for** nombre, apellido, rol, carrera **in** alumnos:

**print** nombre, 'estudia', carrera

O, ya que el apellido y el rol no son usados:

**for** nombre, \_, \_, carrera **in** alumnos:

**print** nombre, 'estudia', carrera

Es posible crear tuplas de largo uno dejando una coma a continuación del único valor:

**>>>** t = (12,)

**>>>** len(t)

1

En otros lenguajes, las tuplas reciben el nombre de **registros**. Este nombre es común, por lo que conviene conocerlo.

**Iteración sobre tuplas**

Al igual que las listas, las tuplas son iterables:

**for** valor **in** (6, 1):

**print** valor \*\* 2

Además, se puede convertir una tupla en una lista usando la función list, y una lista en una tupla usando la función tuple:

**>>>** a = (1, 2, 3)

**>>>** b = [4, 5, 6]

**>>>** list(a)

[1, 2, 3]

**>>>** tuple(b)

(4, 5, 6)

**Diccionarios**

Un **diccionario** es un tipo de datos que sirve para asociar pares de objetos.

Un diccionario puede ser visto como una colección de **llaves**, cada una de las cuales tiene asociada un **valor**. Las llaves no están ordenadas y no hay llaves repetidas. La única manera de acceder a un valor es a través de su llave.

**Cómo crear diccionarios**

Los diccionarios literales se crean usando llaves ({ y }). La llave y el valor van separados por dos puntos:

**>>>** telefonos = {'Pepito': 5552437, 'Jaimito': 5551428, 'Yayita': 5550012}

En este ejemplo, las llaves son 'Pepito', 'Jaimito' y 'Yayita', y los valores asociados a ellas son, respectivamente, 5552437, 5551428 y 5550012.

Un diccionario vacío puede ser creado usando {} o con la función dict():

**>>>** d = {}

**>>>** d = dict()

**Cómo usar un diccionario**

El valor asociado a la llave k en el diccionario d se puede obtener mediante d[k]:

**>>>** telefonos['Pepito']

5552437

**>>>** telefonos['Jaimito']

5551428

A diferencia de los índices de las listas, las llaves de los diccionarios no necesitan ser números enteros.

Si la llave no está presente en el diccionario, ocurre un **error de llave** (KeyError):

**>>>** telefonos['Fulanito']

Traceback (most recent call last):

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

KeyError: 'Fulanito'

Se puede agregar una llave nueva simplemente asignándole un valor:

**>>>** telefonos['Susanita'] = 4448139

**>>>** telefonos

{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 5551428, 'Yayita': 5550012}

Note que el orden en que quedan las llaves en el diccionario no es necesariamente el mismo orden en que fueron agregadas.

Si se asigna un valor a una llave que ya estaba en el diccionario, el valor anterior se sobreescribe. Recuerde que un diccionario no puede tener llaves repetidas:

**>>>** telefonos

{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 5551428, 'Yayita': 5550012}

**>>>** telefonos['Jaimito'] = 4448139

**>>>** telefonos

{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 4448139, 'Yayita': 5550012}

Los valores sí pueden estar repetidos. En el ejemplo anterior, Jaimito y Susanita tienen el mismo número.

Para borrar una llave, se puede usar la sentencia del:

**>>> del** telefonos['Yayita']

**>>>** telefonos

{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 4448139}

Los diccionarios son iterables. Al iterar sobre un diccionario en un ciclo for, se obtiene las llaves:

**>>> for** k **in** telefonos:

**...**  **print** k

**...**

Pepito

Susanita

Jaimito

Para iterar sobre las llaves, se usa d.values():

**>>> for** v **in** telefonos.values():

**...**  **print** v

**...**

5552437

4448139

4448139

Para iterar sobre las llaves y los valores simultáneamente, se usa el método d.items():

**>>> for** k, v **in** telefonos.items():

**...**  **print** 'El telefono de', k, 'es', v

**...**

El telefono de Pepito es 5552437

El telefono de Susanita es 4448139

El telefono de Jaimito es 4448139

También es posible crear listas de llaves o valores:

**>>>** list(telefonos)

['Pepito', 'Susanita', 'Jaimito']

**>>>** list(telefonos.values())

[5552437, 4448139, 4448139]

len(d) entrega cuántos pares llave-valor hay en el diccionario:

**>>>** numeros = {15: 'quince', 24: 'veinticuatro'}

**>>>** len(numeros)

2

**>>>** len({})

0

k in d permite saber si la llave k está en el diccionario d:

**>>>** patas = {'gato': 4, 'humano': 2, 'pulpo': 8, 'perro': 4, 'ciempies': 100}

**>>>** 'perro' **in** patas

True

**>>>** 'gusano' **in** patas

False

Para saber si una llave *no* está en el diccionario, se usa el operador not in:

**>>>** 'gusano' **not** **in** patas

True

**Restricciones sobre las llaves**

No se puede usar cualquier objeto como llave de un diccionario. Las llaves deben ser de un tipo de datos inmutable. Por ejemplo, no se puede usar listas:

**>>>** d = {[1, 2, 3]: 'hola'}

Traceback (most recent call last):

File "<console>", line 1, in <module>

TypeError: unhashable type: 'list'

Típicamente, se usa números, tuplas y strings como llaves de los diccionarios.

**Conjuntos**

Un **conjunto** es una colección desordenada de valores no repetidos.

Los conjuntos de Python son análogos a los conjuntos matemáticos. El tipo de datos que representa a los conjuntos se llama set.

El tipo set es mutable: una vez que se ha creado un conjunto, puede ser modificado.

**Cómo crear conjuntos**

Las dos maneras principales de crear un conjunto son:

* usar un conjunto literal, entre llaves:
* **>>>** colores = {'azul', 'rojo', 'blanco', 'blanco'}
* **>>>** colores
* {'rojo', 'azul', 'blanco'}

Note que el conjunto no incluye elementos repetidos, y que los elementos no quedan en el mismo orden en que fueron agregados.

* usar la función set aplicada sobre un iterable:
* **>>>** set('abracadabra')
* {'a', 'r', 'b', 'c', 'd'}
* **>>>** set(range(50, 2000, 400))
* {1250, 50, 1650, 850, 450}
* **>>>** set([(1, 2, 3), (4, 5), (6, 7, 8, 9)])
* {(4, 5), (6, 7, 8, 9), (1, 2, 3)}

El conjunto vacío debe ser creado usando set(), ya que {} representa el diccionario vacío.

Los elementos de un conjunto deben ser inmutables. Por ejemplo, no es posible crear un conjunto de listas, pero sí un conjunto de tuplas:

**>>>** s = {[2, 4], [6, 1]}

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

**>>>** s = {(2, 4), (6, 1)}

>>>

Como un conjunto no es ordenado, no tiene sentido intentar obtener un elemento usando un índice:

**>>>** s = {'a', 'b', 'c'}

**>>>** s[0]

Traceback (most recent call last):

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

TypeError: 'set' object does not support indexing

Sin embargo, sí es posible iterar sobre un conjunto usando un ciclo for:

**>>> for** i **in** {'a', 'b', 'c'}:

**...**  **print** i

**...**

a

c

b

**Operaciones sobre conjuntos**

len(s) entrega el número de elementos del conjunto s:

**>>>** len({'azul', 'verde', 'rojo'})

3

**>>>** len(set('abracadabra'))

5

**>>>** len(set())

0

x in s permite saber si el elemento x está en el conjunto s:

**>>>** 3 **in** {2, 3, 4}

True

**>>>** 5 **in** {2, 3, 4}

False

x not in s permite saber si x no está en s:

**>>>** 10 **not** **in** {2, 3, 4}

True

s.add(x) agrega el elemento x al conjunto s:

**>>>** s = {6, 1, 5, 4, 3}

**>>>** s.add(-37)

**>>>** s

{1, 3, 4, 5, 6, -37}

**>>>** s.add(4)

**>>>** s

{1, 3, 4, 5, 6, -37}

s.remove(x) elimina el elemento x del conjunto s:

**>>>** s = {6, 1, 5, 4, 3}

**>>>** s.remove(1)

**>>>** s

{3, 4, 5, 6}

Si el elemento x no está en el conjunto, ocurre un **error de llave**:

**>>>** s.remove(10)

Traceback (most recent call last):

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

KeyError: 10

& y | son, respectivamente, los operadores de intersección y unión:

**>>>** a = {1, 2, 3, 4}

**>>>** b = {2, 4, 6, 8}

**>>>** a & b

{2, 4}

**>>>** a | b

{1, 2, 3, 4, 6, 8}

s - t entrega la diferencia entre s y t; es decir, los elementos de s que no están en t:

**>>>** a - b

{1, 3}

s ^ t entrega la diferencia simétrica entre s y t; es decir, los elementos que están en s o en t, pero no en ambos:

**>>>** a ^ b

{1, 3, 6, 8}

El operador < aplicado sobre conjuntos significa «es subconjunto de»:

**>>>** {1, 2} < {1, 2, 3}

True

**>>>** {1, 4} < {1, 2, 3}

False

s <= t también indica si s es subconjunto de t. La distinción ocurre cuando los conjuntos son iguales:

**>>>** {1, 2, 3} < {1, 2, 3}

False

**>>>** {1, 2, 3} <= {1, 2, 3}

True

**Procesamiento de texto**

Hasta ahora, hemos visto cómo los tipos de datos básicos (strings, enteros, reales, booleanos) y las estructuras de datos permiten representar y manipular información compleja y abstracta en un programa.

Sin embargo, en muchos casos la información no suele estar disponible ya organizada en estructuras de datos convenientes de usar, sino en documentos de texto.

Por ejemplo, las páginas webs son archivos de puro texto, que describen la estructura de un documento en un lenguaje llamado HTML. Usted puede ver el texto de una página web buscando una instrucción «Ver código fuente» (o algo parecido) en el navegador. A partir de este texto, el navegador extrae la información necesaria para reconstruir la página que finalmente usted ve.

Un texto siempre es un string, que puede ser tan largo y complejo como se desee. El procesamiento de texto consiste en manipular strings, ya sea para extraer información del string, para convertir un texto en otro, o para codificar información en un string.

En Python, el tipo str provee muchos métodos convenientes para hacer procesamiento de texto, además de las operaciones más simples que ya aprendimos (como s + t, s[i] y s in t).

**Saltos de línea**

Un string puede contener caracteres de **salto de línea**, que tienen el efecto equivalente al de presionar la tecla Enter. El caracter de salto de línea se representa con \n:

**>>>** a = 'piano**\n**violin**\n**oboe'

**>>> print** a

piano

violin

oboe

Los saltos de línea sólo son visibles al imprimir el string mediante la sentencia print. Si uno quiere ver el valor del string en la consola, el salto de línea aparecerá representado como \n:

**>>>** a

'piano\nviolin\noboe'

**>>> print** a

piano

violin

oboe

Aunque dentro del string se representa como una secuencia de dos símbolos, el salto de línea es un único caracter:

**>>>** len('a**\n**b')

3

En general, hay varios caracteres especiales que se representan comenzando con una barra invertida (\) seguida de una letra. Experimente, y determine qué significan los caracteres especiales \t y \b:

**print** 'abcde**\t**efg**\t**hi**\t**jklm'

**print** 'abcde**\b**efg**\b**hi**\b**jklm'

Para incluir una barra invertida dentro de un string, hay que hacerlo con \\:

**>>> print** 'c:**\\**>'

c:\>

**>>> print** '[**\\**](file://\)o/ o**\n** | /|**\\\n**/ **\\** / **\\**'

\o/ o

| /|\

/ \ / \

**Reemplazar secciones del string**

El método s.replace(antes, despues) busca en el string s todas las apariciones del texto antes y las reemplaza por despues:

**>>>** 'La mar astaba sarana'.replace('a', 'e')

'Le mer estebe serene'

**>>>** 'La mar astaba sarana'.replace('a', 'i')

'Li mir istibi sirini'

**>>>** 'La mar astaba sarana'.replace('a', 'o')

'Lo mor ostobo sorono'

Hay que tener siempre muy claro que esta operación no modifica el string, sino que retorna uno nuevo:

**>>>** orden = 'Quiero arroz con pollo'

**>>>** orden.replace('arroz', 'pure').replace('pollo', 'huevo')

'Quiero pure con huevo'

**>>>** orden

'Quiero arroz con pollo'

**Separar y juntar strings**

s.split() separa el strings en varios strings, usando los espacios en blanco como separador. El valor retornado es una lista de strings:

**>>>** oracion = 'El veloz murcielago hindu comia feliz cardillo y kiwi'

**>>>** oracion.split()

['El', 'veloz', 'murcielago', 'hindu', 'comia', 'feliz', 'cardillo', 'y', 'kiwi']

Además, es posible pasar un parámetro al método split que indica cuál será el separador a usar (en vez de los espacios en blanco):

**>>>** s = 'Ana lavaba las sabanas'

**>>>** s.split()

['Ana', 'lavaba', 'las', 'sabanas']

**>>>** s.split('a')

['An', ' l', 'v', 'b', ' l', 's s', 'b', 'n', 's']

**>>>** s.split('l')

['Ana ', 'avaba ', 'as sabanas']

**>>>** s.split('aba')

['Ana lav', ' las s', 'nas']

Esto es muy útil para pedir al usuario que ingrese datos en un programa de una manera más conveniente, y no uno por uno. Por ejemplo, antes hacíamos programas que funcionaban así:

Ingrese a: **2.3**

Ingrese b: **1.9**

Ingrese c: **2.3**

El triangulo es isoceles.

Ahora podemos hacerlos así:

Ingrese lados del triangulo: **2.3 1.9 2.3**

El triangulo es isoceles.

En este caso, el código del programa podría ser:

entrada = raw\_input('Ingrese lados del triangulo: ')

lados = entrada.split()

a = int(lados[0])

b = int(lados[1])

c = int(lados[2])

**print** 'El triangulo es', tipo\_triangulo(a, b, c)

O usando la función map, más simplemente:

entrada = raw\_input('Ingrese lados del triangulo: ')

a, b, c = map(int, entrada.split())

**print** 'El triangulo es', tipo\_triangulo(a, b, c)

s.join(lista\_de\_strings) une todos los strings de la lista, usando al string s como «pegamento»:

**>>>** valores = map(str, range(10))

**>>>** pegamento = ' '

**>>>** pegamento.join(valores)

'0 1 2 3 4 5 6 7 8 9'

**>>>** ''.join(valores)

'0123456789'

**>>>** ','.join(valores)

'0,1,2,3,4,5,6,7,8,9'

**>>>** ' --> '.join(valores)

'0 --> 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> 9'

**Mayúsculas y minúsculas**

s.isupper() y s.islower() indican si el string está, respectivamente, en mayúsculas o minúsculas:

**>>>** s = 'hola'

**>>>** t = 'Hola'

**>>>** u = 'HOLA'

**>>>** s.isupper(), s.islower()

(False, True)

**>>>** t.isupper(), t.islower()

(False, False)

**>>>** u.isupper(), u.islower()

(True, False)

s.upper() y s.lower() entregan el string s convertido, respectivamente, a mayúsculas y minúsculas:

**>>>** t

'Hola'

**>>>** t.upper()

'HOLA'

**>>>** t.lower()

'hola'

s.swapcase() cambia las minúsculas a mayúsculas, respectivamente, a mayúsculas y minúsculas:

**>>>** t.swapcase()

'hOLA'

Lamentablemente, ninguno de estos métodos funcionan con acentos y eñes:

**>>> print** 'ñandú'.upper()

ñANDú

**Revisar contenidos del string**

s.startswith(t) y s.endswith(t) indican si el string s comienza y termina, respectivamente, con el string t:

**>>>** objeto = 'paraguas'

**>>>** objeto.startswith('para')

True

**>>>** objeto.endswith('aguas')

True

**>>>** objeto.endswith('x')

False

**>>>** objeto.endswith('guaguas')

False

Nuestro conocido operador in indica si un string está contenido dentro de otro:

**>>>** 'pollo' **in** 'repollos'

True

**>>>** 'pollo' **in** 'gallinero'

False

**Alineación de strings**

Los métodos s.ljust(n), s.rjust(n) y s.center(n) rellenan el string con espacios para que su largo sea igual a n, de modo que el contenido quede alineado, respectivamente, a la izquierda, a la derecha y al centro:

**>>>** contenido.ljust(20)

'hola '

**>>>** contenido.center(20)

' hola '

**>>>** contenido.rjust(20)

' hola'

Estos métodos son útiles para imprimir tablas bien alineadas:

datos = [

('Pepito', (1991, 12, 5), 'Osorno', '\*\*\*'),

('Yayita', (1990, 1, 31), 'Arica', '\*'),

('Fulanito', (1992, 10, 29), 'Porvenir', '\*\*\*\*'),

]

**for** n, (a, m, d), c, e **in** datos:

**print** n.ljust(10),

**print** str(a).rjust(4), str(m).rjust(2), str(d).rjust(2),

**print** c.ljust(10), e.center(5)

Este programa imprime lo siguiente:

Pepito 1991 12 5 Osorno \*\*\*

Yayita 1990 1 31 Arica \*

Fulanito 1992 10 29 Porvenir \*\*\*\*

**Interpolación de strings**

El método format permite usar un string como una plantilla que se puede completar con distintos valores dependiendo de la situación.

Las posiciones en que se deben rellenar los valores se indican dentro del string usando un número entre paréntesis de llaves:

**>>>** s = 'Soy {0} y vivo en {1}'

Estas posiciones se llaman *campos*. En el ejemplo, el string s tiene dos campos, numerados del cero al uno.

Para llenar los campos, hay que llamar al método format pasándole los valores como parámetros:

**>>>** s.format('Perico', 'Valparaiso')

'Soy Perico y vivo en Valparaiso'

**>>>** s.format('Erika', 'Berlin')

'Soy Erika y vivo en Berlin'

**>>>** s.format('Wang Dawei', 'Beijing')

'Soy Wang Dawei y vivo en Beijing'

El número indica en qué posición va el parámetro que está asociado al campo:

**>>>** '{1}{0}{2}{0}'.format('a', 'v', 'c')

'vaca'

**>>>** '{0} y {1}'.format('carne', 'huevos')

'carne y huevos'

**>>>** '{1} y {0}'.format('carne', 'huevos')

'huevos y carne'

Otra opción es referirse a los campos con un nombre. En este caso, hay que llamar al método format diciendo explícitamente el nombre del parámetro para asociarlo al valor:

**>>>** s = '{nombre} estudia en la {universidad}'

**>>>** s.format(nombre='Perico', universidad='UTFSM')

'Perico estudia en la UTFSM'

**>>>** s.format(nombre='Fulana', universidad='PUCV')

'Fulana estudia en la PUCV'

**>>>** s.format(universidad='UPLA', nombre='Yayita')

'Yayita estudia en la UPLA'

**Archivos**

Todos los datos que un programa utiliza durante su ejecución se encuentran en sus variables, que están almacenadas en la [memoria RAM](http://es.wikipedia.org/wiki/Memoria_RAM) del computador.

La memoria RAM es un medio de almacenamiento **volátil**: cuando el programa termina, o cuando el computador se apaga, todos los datos se pierden para siempre.

Para que un programa pueda guardar datos de manera permanente, es necesario utilizar un medio de almacenamiento **persistente**, de los cuales el más importante es el [disco duro](http://es.wikipedia.org/wiki/Disco_duro).

Los datos en el disco duro están organizados en [archivos](http://es.wikipedia.org/wiki/Archivo_(inform%C3%A1tica)). Un **archivo** es una secuencia de datos almacenados en un medio persistente que están disponibles para ser utilizados por un programa. Todos los archivos tienen un nombre y una ubicación dentro del sistema de archivos del sistema operativo.

Los datos en un archivo siguen estando presentes después de que termina el programa que lo ha creado. Un programa puede guardar sus datos en archivos para usarlos en una ejecución futura, e incluso puede leer datos desde archivos creados por otros programas.

Un programa no puede manipular los datos de un archivo directamente. Para usar un archivo, un programa siempre abrir el archivo y asignarlo a una variable, que llamaremos el **archivo lógico**. Todas las operaciones sobre un archivo se realizan a través del archivo lógico.

Dependiendo del contenido, hay muchos tipos de archivos. Nosotros nos preocuparemos sólo de los **archivos de texto**, que son los que contienen texto, y pueden ser abiertos y modificados usando un editor de texto como el Bloc de Notas. Los archivos de texto generalmente tienen un nombre terminado en .txt.

**Lectura de archivos**

Para leer datos de un archivo, hay que abrirlo de la siguiente manera:

archivo = open(nombre)

nombre es un string que tiene el nombre del archivo. archivo es el archivo lógico a través del que se manipulará el archivo.

Si el archivo no existe, ocurrirá un **error de entrada y salida** (IOError).

Es importante recordar que la variable archivo es una representación abstracta del archivo, y no los contenidos del mismo.

La manera más simple de leer el contenido es hacerlo línea por línea. Para esto, basta con poner el archivo lógico en un ciclo for:

for linea in archivo:

# hacer algo

Una vez que los datos han sido leídos del archivo, hay que cerrarlo:

archivo.close()

Por ejemplo, supongamos que tenemos el archivo himno.txt que tiene el siguiente contenido:

Puro Chile

es tu cielo azulado

puras brisas

te cruzan también.

El archivo tiene cuatro líneas. Cada línea termina con un salto de línea (\n), que indica que a continuación comienza una línea nueva.

El siguiente programa imprime la primera letra de cada línea del himno:

archivo = open('himno.txt')

**for** linea **in** archivo:

**print** linea[0]

archivo.close()

El ciclo for es ejecutado cuatro veces, una por cada línea del archivo. La salida del programa es:

P

e

p

t

Otro ejemplo: el siguiente programa imprime cuántos símbolos hay en cada línea:

archivo = open('himno.txt')

**for** linea **in** archivo:

**print** len(linea)

archivo.close()

La salida es:

11

20

13

19

Note que el salto de línea (el “enter”) es considerado en la cuenta:

+---+---+---+---+---+---+---+---+---+---+---+

| P | u | r | o | | C | h | i | l | e | \n| = 11 símbolos

+---+---+---+---+---+---+---+---+---+---+---+

Para obtener el string sin el salto de línea, se puede usar el método strip, que elimina todos los símbolos de espaciado al principio y al final del string:

**>>>** s = ' Hola**\n**'

**>>>** s.strip()

'Hola'

Si modificamos el programa para eliminar el salto de línea:

archivo = open('himno.txt')

**for** linea **in** archivo:

**print** len(linea.strip())

archivo.close()

entonces la salida es:

10

19

12

18

Lo importante es comprender que los archivos son leídos línea por línea usando el ciclo for.

**Escritura en archivos**

Los ejemplos anteriores suponen que el archivo por leer existe, y está listo para ser abierto y leído. Ahora veremos cómo crear los archivos y cómo escribir datos en ellos, para que otro programa después pueda abrirlos y leerlos.

Uno puede crear un archivo vacío abriéndolo de la siguiente manera:

archivo = open(nombre, 'w')

El segundo parámetro de la función open indica el uso que se le dará al archivo. 'w' significa «escribir» (*write* en inglés).

Si el archivo señalado no existe, entonces será creado. Si ya existe, entonces será sobreescrito. Hay que tener cuidado entonces, pues esta operación elimina los datos del archivo que existía previamente.

Una vez abierto el archivo, uno puede escribir datos en él usando el método write:

a = open('prueba.txt', 'w')

a.write('Hola ')

a.write('mundo.')

a.close()

Una vez ejecutado este programa, el archivo prueba.txt será creado (o sobreescrito, si ya existía). Al abrirlo en el Bloc de Notas, veremos este contenido:

Hola mundo.

Para escribir varias líneas en el archivo, es necesario agregar explícitamente los saltos de línea en cada string que sea escrito. Por ejemplo, para crear el archivo himno.txt que usamos más arriba, podemos hacerlo así:

a = open('himno.txt', 'w')

a.write('Puro Chile**\n**')

a.write('es tu cielo azulado**\n**')

a.write('puras brisas**\n**')

a.write('te cruzan también.**\n**')

a.close()

Además del modo 'w' (*write*), también existe el modo 'a' (*append*), que permite escribir datos al final de un archivo existente. Por ejemplo, el siguiente programa abre el archivo prueba.txt que creamos más arriba, y agrega más texto al final de él:

a = open('prueba.txt', 'a')

a.write('**\n**')

a.write('Chao ')

a.write('pescao.')

a.close()

Si abrimos el archivo prueba.txt en el Bloc de Notas, veremos esto:

Hola mundo.

Chao pescao.

De haber abierto el archivo en modo 'w' en vez de 'a', el contenido anterior (la frase Hola mundo) se habría borrado.

**Archivos de valores con separadores**

Una manera usual de almacenar datos con estructura de tabla en un archivo es la siguiente: cada línea del archivo representa una fila de la tabla, y los datos de una fila se ponen separados por algún símbolo especial.

Por ejemplo, supongamos que queremos guardar en un archivo los datos de esta tabla:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nombre** | **Apellido** | **Nota 1** | **Nota 2** | **Nota 3** | **Nota 4** |
| Perico | Los Palotes | 90 | 75 | 38 | 65 |
| Yayita | Vinagre | 39 | 49 | 58 | 55 |
| Fulana | De Tal | 96 | 100 | 36 | 71 |

Si usamos el símbolo : como separador, el archivo, que llamaremos alumnos.txt, debería quedar así:

Perico:Los Palotes:90:75:38:65

Yayita:Vinagre:39:49:58:55

Fulanita:De Tal:96:100:36:71

El formato de estos archivos se suele llamar [CSV](http://en.wikipedia.org/wiki/CSV_(file_format)), que en inglés son las siglas de *comma-separated values* (significa «valores separados por comas», aunque técnicamente el separador puede ser cualquier símbolo). A pesar del nombre especial que reciben, los archivos CSV son archivos de texto como cualquier otro, y se pueden tratar como tales.

Los archivos de valores con separadores son muy fáciles de leer y escribir, y por esto son muy usados. Como ejemplo práctico, si usted desea hacer un programa que analice los datos de una hoja de cálculo Excel, puede guardar el archivo con el formato CSV directamente en el Excel, y luego abrirlo desde su programa escrito en Python.

Para leer los datos de un archivo de valores con separadores, debe hacerlo línea por línea, eliminar el salto de línea usando el método strip y luego extraer los valores de la línea usando el método split. Por ejemplo, al leer la primera línea del archivo de más arriba obtendremos el siguiente string:

'Perico:Los Palotes:90:75:38:65**\n**'

Para separar los seis valores, lo podemos hacer así:

**>>>** linea.strip().split(':')

['Perico', 'Los Palotes', '90', '75', '38', '65']

Como se trata de un archivo de texto, todos los valores son strings. Una manera de convertir los valores a sus tipos apropiados es hacerlo uno por uno:

valores = linea.strip().split(':')

nombre = valores[0]

apellido = valores[1]

nota1 = int(valores[2])

nota2 = int(valores[3])

nota3 = int(valores[4])

nota4 = int(valores[5])

Una manera más breve es usar las rebanadas y la función map:

valores = linea.strip().split(':')

nombre, apellido = valores[0:2]

nota1, nota2, nota3, nota4 = map(int, valores[2:6])

O podríamos dejar las notas en una lista, en vez de usar cuatro variables diferentes:

notas = map(int, valores[2:6])

Por ejemplo, un programa para imprimir el promedio de todos los alumnos se puede escribir así:

archivo\_alumnos = open('alumnos.txt')

**for** linea **in** archivo\_alumnos:

valores = linea.strip().split(':')

nombre, apellido = valores[0:2]

notas = map(int, valores[2:6])

promedio = sum(notas) / 4.0

**print** '{0} obtuvo promedio {1}'.format(nombre, promedio)

archivo\_alumnos.close()

Para escribir los datos en un archivo, hay que hacer el proceso inverso: convertir todos los datos al tipo string, pegarlos en un único string, agregar el salto de línea al final y escribir la línea en el archivo.

Si los datos de la línea ya están en una lista o una tupla, podemos convertirlos a string usando la función map y pegarlos usando el método join:

alumno = ('Perico', 'Los Palotes', 90, 75, 38, 65)

linea = ':'.join(map(str, alumno)) + '**\n**'

archivo.write(linea)

Otra manera es armar el string parte por parte:

linea = '{0}:{1}:{2}:{3}:{4}:{5}**\n**'.format(nombre, apellido,

nota1, nota2, nota3, nota4)

archivo.write(linea)

Como siempre, usted debe preferir la manera que le parezca más simple de entender.

**Arreglos**

Las estructuras de datos que hemos visto hasta ahora (listas, tuplas, diccionarios, conjuntos) permiten manipular datos de manera muy flexible. Combinándolas y anidándolas, es posible organizar información de manera estructurada para representar sistemas del mundo real.

En muchas aplicaciones de Ingeniería, por otra parte, más importante que la organización de los datos es la capacidad de hacer muchas operaciones a la vez sobre grandes conjuntos de datos numéricos de manera eficiente. Algunos ejemplos de problemas que requieren manipular grandes secuencias de números son: la predicción del clima, la construcción de edificios, y el análisis de indicadores financieros entre muchos otros.

La estructura de datos que sirve para almacenar estas grandes secuencias de números (generalmente de tipo float) es el **arreglo**.

Los arreglos tienen algunas similitudes con las listas:

* los elementos tienen un orden y se pueden acceder mediante su posición,
* los elementos se pueden recorrer usando un ciclo for.

Sin embargo, también tienen algunas restricciones:

* todos los elementos del arreglo deben tener el mismo tipo,
* en general, el tamaño del arreglo es fijo (no van creciendo dinámicamente como las listas),
* se ocupan principalmente para almacenar datos numéricos.

A la vez, los arreglos tienen muchas ventajas por sobre las listas, que iremos descubriendo a medida que avancemos en la materia.

Los arreglos son los equivalentes en programación de las **matrices** y **vectores** de las matemáticas. Precisamente, una gran motivación para usar arreglos es que hay mucha teoría detrás de ellos que puede ser usada en el diseño de algoritmos para resolver problemas verdaderamente interesantes.

**Crear arreglos**

El módulo que provee las estructuras de datos y las funciones para trabajar con arreglos se llama **NumPy**, y no viene incluído con Python, por lo que hay que instalarlo por separado.

Descargue el instalador apropiado para su versión de Python desde la [página de descargas de NumPy](http://tinyurl.com/bajar-numpy). Para ver qué versión de Python tiene instalada, vea la primera línea que aparece al abrir una consola.

Para usar las funciones provistas por NumPy, debemos importarlas al principio del programa:

**from** **numpy** **import** array

Como estaremos usando frecuentemente muchas funciones de este módulo, conviene importarlas todas de una vez usando la siguiente sentencia:

**from** **numpy** **import** \*

(Si no recuerda cómo usar el import, puede repasar la materia sobre [módulos](http://progra.usm.cl/apunte/materia/modulos.html)).

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

**>>>** a = array([6, 1, 3, 9, 8])

**>>>** a

array([6, 1, 3, 9, 8])

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

**>>>** b = array([6.0, 1, 3, 9, 8])

**>>>** b

array([ 6., 1., 3., 9., 8.])

Otra opción es convertir el arreglo a otro tipo usando el método astype:

**>>>** a

array([6, 1, 3, 9, 8])

**>>>** a.astype(float)

array([ 6., 1., 3., 9., 8.])

**>>>** a.astype(complex)

array([ 6.+0.j, 1.+0.j, 3.+0.j, 9.+0.j, 8.+0.j])

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

* zeros(n) crea un arreglo de n ceros;
* ones(n) crea un arreglo de n unos;
* arange(a, b, c) crea un arreglo de forma similar a la función range, con las diferencias que a, b y c pueden ser reales, y que el resultado es un arreglo y no una lista;
* linspace(a, b, n) crea un arreglo de n valores equiespaciados entre a y b.

**>>>** zeros(6)

array([ 0., 0., 0., 0., 0., 0.])

**>>>** ones(5)

array([ 1., 1., 1., 1., 1.])

**>>>** arange(3.0, 9.0)

array([ 3., 4., 5., 6., 7., 8.])

**>>>** linspace(1, 2, 5)

array([ 1. , 1.25, 1.5 , 1.75, 2. ])

**Operaciones con arreglos**

Las limitaciones que tienen los arreglos respecto de las listas son compensadas por la cantidad de operaciones convenientes que permiten realizar sobre ellos.

Las operaciones aritméticas entre arreglos se aplican elemento a elemento:

**>>>** a = array([55, 21, 19, 11, 9])

**>>>** b = array([12, -9, 0, 22, -9])

# sumar los dos arreglos elemento a elemento

**>>>** a + b

array([67, 12, 19, 33, 0])

# multiplicar elemento a elemento

**>>>** a \* b

array([ 660, -189, 0, 242, -81])

# restar elemento a elemento

**>>>** a - b

array([ 43, 30, 19, -11, 18])

Las operaciones entre un arreglo y un valor simple funcionan aplicando la operación a todos los elementos del arreglo, usando el valor simple como operando todas las veces:

**>>>** a

array([55, 21, 19, 11, 9])

# multiplicar por 0.1 todos los elementos

**>>>** 0.1 \* a

array([ 5.5, 2.1, 1.9, 1.1, 0.9])

# restar 9.0 a todos los elementos

**>>>** a - 9.0

array([ 46., 12., 10., 2., 0.])

Note que si quisiéramos hacer estas operaciones usando listas, necesitaríamos usar un ciclo para hacer las operaciones elemento a elemento.

Las operaciones relacionales también se aplican elemento a elemento, y retornan un arreglo de valores booleanos:

**>>>** a = array([5.1, 2.4, 3.8, 3.9])

**>>>** b = array([4.2, 8.7, 3.9, 0.3])

**>>>** c = array([5, 2, 4, 4]) + array([1, 4, -2, -1]) / 10.0

**>>>** a < b

array([False, True, True, False], dtype=bool)

**>>>** a == c

array([ True, True, True, True], dtype=bool)

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones any y all. any retorna True si al menos uno de los elementos es verdadero, mientras que all retorna True sólo si todos lo son (en inglés, *any* signfica «alguno», y *all* significa «todos»):

**>>>** any(a < b)

True

**>>>** any(a == b)

False

**>>>** all(a == c)

True

**Funciones sobre arreglos**

NumPy provee muchas funciones matemáticas que también operan elemento a elemento. Por ejemplo, podemos obtener el seno de 9 valores equiespaciados entre 0 y *π*/2 con una sola llamada a la función sin:

**>>> from** **numpy** **import** linspace, pi, sin

**>>>** x = linspace(0, pi/2, 9)

**>>>** x

array([ 0. , 0.19634954, 0.39269908,

0.58904862, 0.78539816, 0.9817477 ,

1.17809725, 1.37444679, 1.57079633])

**>>>** sin(x)

array([ 0. , 0.19509032, 0.38268343,

0.55557023, 0.70710678, 0.83146961,

0.92387953, 0.98078528, 1. ])

Como puede ver, los valores obtenidos crecen desde 0 hasta 1, que es justamente como se comporta la función seno en el intervalo [0, *π*/2].

Aquí también se hace evidente otra de las ventajas de los arreglos: al mostrarlos en la consola o al imprimirlos, los valores aparecen perfectamente alineados. Con las listas, esto no ocurre:

**>>>** list(sin(x))

[0.0, 0.19509032201612825, 0.38268343236508978, 0.5555702330

1960218, 0.70710678118654746, 0.83146961230254524, 0.9238795

3251128674, 0.98078528040323043, 1.0]

**Arreglos aleatorios**

El módulo NumPy contiene a su vez otros módulos que proveen funcionalidad adicional a los arreglos y funciones básicos.

El módulo numpy.random provee funciones para crear **números aleatorios** (es decir, generados al azar), de las cuales la más usada es la función random, que entrega un arreglo de números al azar distribuidos uniformemente entre 0 y 1:

**>>> from** **numpy.random** **import** random

**>>>** random(3)

array([ 0.53077263, 0.22039319, 0.81268786])

**>>>** random(3)

array([ 0.07405763, 0.04083838, 0.72962968])

**>>>** random(3)

array([ 0.51886706, 0.46220545, 0.95818726])

**Obtener elementos de un arreglo**

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice —1:

**>>>** a = array([6.2, -2.3, 3.4, 4.7, 9.8])

**>>>** a[0]

6.2

**>>>** a[1]

-2.3

**>>>** a[-2]

4.7

**>>>** a[3]

4.7

Una seccion del arreglo puede ser obtenida usando el operador de rebanado a[i:j]. Los índices i y j indican el rango de valores que serán entregados:

**>>>** a

array([ 6.2, -2.3, 3.4, 4.7, 9.8])

**>>>** a[1:4]

array([-2.3, 3.4, 4.7])

**>>>** a[2:-2]

array([ 3.4])

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

**>>>** a[:2]

array([ 6.2, -2.3])

**>>>** a[2:]

array([ 3.4, 4.7, 9.8])

Un tercer índice puede indicar cada cuántos elementos serán incluídos en el resultado:

**>>>** a = linspace(0, 1, 9)

**>>>** a

array([ 0. , 0.125, 0.25 , 0.375, 0.5 , 0.625, 0.75 , 0.875, 1. ])

**>>>** a[1:7:2]

array([ 0.125, 0.375, 0.625])

**>>>** a[::3]

array([ 0. , 0.375, 0.75 ])

**>>>** a[-2::-2]

array([ 0.875, 0.625, 0.375, 0.125])

**>>>** a[::-1]

array([ 1. , 0.875, 0.75 , 0.625, 0.5 , 0.375, 0.25 , 0.125, 0. ])

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

**>>>** b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])

**>>>** b[2:5]

array([ 10.99, -2.29, 3.86])

**>>>** b[:5]

array([ 17.41, 2.19, 10.99, -2.29, 3.86])

**>>>** b[1:1]

array([], dtype=float64)

**>>>** b[1:5:2]

array([ 2.19, -2.29])

**Algunos métodos convenientes**

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos min y max, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

**>>>** a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.2])

**>>>** a.min()

-2.5

**>>>** a.max()

8.4000000000000004

Los métodos argmin y argmax entregan respectivamente la posición del mínimo y del máximo:

**>>>** a.argmin()

4

**>>>** a.argmax()

2

Los métodos sum y prod entregan respectivamente la suma y el producto de los elementos:

**>>>** a.sum()

24.041592653589795

**>>>** a.prod()

-11393.086289208301

**Arreglos bidimensionales**

Los **arreglos bidimensionales** son tablas de valores. Cada elemento de un arreglo bidimensional está simultáneamente en una fila y en una columna.

En matemáticas, a los arreglos bidimensionales se les llama [matrices](http://es.wikipedia.org/wiki/Matriz_(matem%C3%A1tica)), y son muy utilizados en problemas de Ingeniería.

En un arreglo bidimensional, cada elemento tiene una posición que se identifica mediante dos índices: el de su fila y el de su columna.

**Crear arreglos bidimensionales**

Los arreglos bidimensionales también son provistos por NumPy, por lo que debemos comenzar importando las funciones de este módulo:

**from** **numpy** **import** \*

Al igual que los arreglos de una dimensión, los arreglos bidimensionales también pueden ser creados usando la función array, pero pasando como argumentos una lista con las filas de la matriz:

a = array([[5.1, 7.4, 3.2, 9.9],

[1.9, 6.8, 4.1, 2.3],

[2.9, 6.4, 4.3, 1.4]])

Todas las filas deben ser del mismo largo, o si no ocurre un error de valor:

**>>>** array([[1], [2, 3]])

Traceback (most recent call last):

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

ValueError: setting an array element with a sequence.

Los arreglos tienen un atributo llamado shape, que es una tupla con los tamaños de cada dimensión. En el ejemplo, a es un arreglo de dos dimensiones que tiene tres filas y cuatro columnas:

**>>>** a.shape

(3, 4)

Los arreglos también tienen otro atributo llamado size que indica cuántos elementos tiene el arreglo:

**>>>** a.size

12

Por supuesto, el valor de a.size siempre es el producto de los elementos de a.shape.

Hay que tener cuidado con la función len, ya que no retorna el tamaño del arreglo, sino su cantidad de filas:

**>>>** len(a)

3

Las funciones zeros y ones también sirven para crear arreglos bidimensionales. En vez de pasarles como argumento un entero, hay que entregarles una tupla con las cantidades de filas y columnas que tendrá la matriz:

**>>>** zeros((3, 2))

array([[ 0., 0.],

[ 0., 0.],

[ 0., 0.]])

**>>>** ones((2, 5))

array([[ 1., 1., 1., 1., 1.],

[ 1., 1., 1., 1., 1.]])

Lo mismo se cumple para muchas otras funciones que crean arreglos; por ejemplom la función random:

**>>> from** **numpy.random** **import** random

**>>>** random((5, 2))

array([[ 0.80177393, 0.46951148],

[ 0.37728842, 0.72704627],

[ 0.56237317, 0.3491332 ],

[ 0.35710483, 0.44033758],

[ 0.04107107, 0.47408363]])

**Operaciones con arreglos bidimensionales**

Al igual que los arreglos de una dimensión, las operaciones sobre las matrices se aplican término a término:

**>>>** a = array([[5, 1, 4],

**...**  [0, 3, 2]])

**>>>** b = array([[2, 3, -1],

**...**  [1, 0, 1]])

**>>>** a + 2

array([[7, 3, 6],

[2, 5, 4]])

**>>>** a \*\* b

array([[25, 1, 0],

[ 0, 1, 2]])

Cuando dos matrices aparecen en una operación, ambas deben tener exactamente la misma forma:

**>>>** a = array([[5, 1, 4],

**...**  [0, 3, 2]])

**>>>** b = array([[ 2, 3],

**...**  [-1, 1],

**...**  [ 0, 1]])

**>>>** a + b

Traceback (most recent call last):

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

ValueError: shape mismatch: objects cannot be broadcast to a single shape

**Obtener elementos de un arreglo bidimensional**

Para obtener un elemento de un arreglo, debe indicarse los índices de su fila i y su columna j mediante la sintaxis a[i, j]:

**>>>** a = array([[ 3.21, 5.33, 4.67, 6.41],

[ 9.54, 0.30, 2.14, 6.57],

[ 5.62, 0.54, 0.71, 2.56],

[ 8.19, 2.12, 6.28, 8.76],

[ 8.72, 1.47, 0.77, 8.78]])

**>>>** a[1, 2]

2.14

**>>>** a[4, 3]

8.78

**>>>** a[-1, -1]

8.78

**>>>** a[0, -1]

6.41

También se puede obtener secciones rectangulares del arreglo usando el operador de rebanado con los índices:

**>>>** a[2:3, 1:4]

array([[ 0.54, 0.71, 2.56]])

**>>>** a[1:4, 0:4]

array([[ 9.54, 0.3 , 2.14, 6.57],

[ 5.62, 0.54, 0.71, 2.56],

[ 8.19, 2.12, 6.28, 8.76]])

**>>>** a[1:3, 2]

array([ 2.14, 0.71])

**>>>** a[0:4:2, 3:0:-1]

array([[ 6.41, 4.67, 5.33],

[ 2.56, 0.71, 0.54]])

**>>>** a[::4, ::3]

array([[ 3.21, 6.41],

[ 8.72, 8.78]])

Para obtener una fila completa, hay que indicar el índice de la fila, y poner : en el de las columnas (significa «desde el principio hasta el final»). Lo mismo para las columnas:

**>>>** a[2, :]

array([ 5.62, 0.54, 0.71, 2.56])

**>>>** a[:, 3]

array([ 6.41, 6.57, 2.56, 8.76, 8.78])

Note que el número de dimensiones es igual a la cantidad de rebanados que hay en los índices:

**>>>** a[2, 3] *# valor escalar (arreglo de cero dimensiones)*

2.56

**>>>** a[2:3, 3] *# arreglo de una dimensión de 1 elemento*

array([ 2.56])

**>>>** a[2:3, 3:4] *# arreglo de dos dimensiones de 1 x 1*

array([[ 2.56]])

**Otras operaciones**

La **trasposicion** consiste en cambiar las filas por las columnas y viceversa. Para trasponer un arreglo, se usa el método transpose:

**>>>** a

array([[ 3.21, 5.33, 4.67, 6.41],

[ 9.54, 0.3 , 2.14, 6.57],

[ 5.62, 0.54, 0.71, 2.56]])

**>>>** a.transpose()

array([[ 3.21, 9.54, 5.62],

[ 5.33, 0.3 , 0.54],

[ 4.67, 2.14, 0.71],

[ 6.41, 6.57, 2.56]])

El método reshape entrega un arreglo que tiene los mismos elementos pero otra forma. El parámetro de reshape es una tupla indicando la nueva forma del arreglo:

**>>>** a = arange(12)

**>>>** a

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

**>>>** a.reshape((4, 3))

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 9, 10, 11]])

**>>>** a.reshape((2, 6))

array([[ 0, 1, 2, 3, 4, 5],

[ 6, 7, 8, 9, 10, 11]])

La función diag aplicada a un arreglo bidimensional entrega la diagonal principal de la matriz (es decir, todos los elementos de la forma a[i, i]):

**>>>** a

array([[ 3.21, 5.33, 4.67, 6.41],

[ 9.54, 0.3 , 2.14, 6.57],

[ 5.62, 0.54, 0.71, 2.56]])

**>>>** diag(a)

array([ 3.21, 0.3 , 0.71])

Además, diag recibe un segundo parámetro opcional para indicar otra diagonal que se desee obtener. Las diagonales sobre la principal son positivas, y las que están bajo son negativas:

**>>>** diag(a, 2)

array([ 4.67, 6.57])

**>>>** diag(a, -1)

array([ 9.54, 0.54])

La misma función diag también cumple el rol inverso: al recibir un arreglo de una dimensión, retorna un arreglo bidimensional que tiene los elementos del parámetro en la diagonal:

**>>>** diag(arange(5))

array([[0, 0, 0, 0, 0],

[0, 1, 0, 0, 0],

[0, 0, 2, 0, 0],

[0, 0, 0, 3, 0],

[0, 0, 0, 0, 4]])

**Reducciones por fila y por columna**

Algunas operaciones pueden aplicarse tanto al arreglo completo como a todas las filas o a todas las columnas.

Por ejemplo, a.sum() entrega la suma de todos los elementos del arreglo. Además, se le puede pasar un parámetro para hacer que la operación se haga por filas o por columnas:

**>>>** a = array([[ 4.3, 2.9, 9.1, 0.1, 2. ],

**...**  [ 8. , 4.5, 6.4, 6. , 4.3],

**...**  [ 7.8, 3.1, 3.4, 7.8, 8.4],

**...**  [ 1.2, 1.5, 9. , 6.3, 6.8],

**...**  [ 7.6, 9.2, 3.3, 0.9, 8.6],

**...**  [ 5.3, 6.7, 4.6, 5.3, 1.2],

**...**  [ 4.6, 9.1, 1.5, 3. , 0.6]])

**>>>** a.sum()

174.4

**>>>** a.sum(0)

array([ 38.8, 37. , 37.3, 29.4, 31.9])

**>>>** a.sum(1)

array([ 18.4, 29.2, 30.5, 24.8, 29.6, 23.1, 18.8])

El parámetro indica a lo largo de qué dimensión se hará la suma. El 0 significa «sumar a lo largo de las filas». Pero hay que tener cuidado, ¡por que lo que se obtiene son las sumas de las columnas! Del mismo modo, 1 significa «a lo largo de las columnas, y lo que se obtiene es el arreglo con las sumas de cada fila.

Las operaciones a.min() y a.max() funcionan del mismo modo:

**>>>** a.min()

0.1

**>>>** a.min(0)

array([ 1.2, 1.5, 1.5, 0.1, 0.6])

**>>>** a.min(1)

array([ 0.1, 4.3, 3.1, 1.2, 0.9, 1.2, 0.6])

a.argmin() y a.argmax() también:

**>>>** a.argmin(0)

array([3, 3, 6, 0, 6])

**>>>** a.argmin(1)

array([3, 4, 1, 0, 3, 4, 4])

**Productos entre arreglos**

Recordemos que **vector** es sinónimo de arreglo de una dimensión, y **matriz** es sinónimo de arreglo de dos dimensiones.

**Producto interno (vector-vector)**

El **producto interno** entre dos vectores es la suma de los productos entre elementos correspondientes:

El producto interno entre dos vectores se obtiene usando la función dot provista por NumPy:

**>>>** a = array([-2.8 , -0.88, 2.76, 1.3 , 4.43])

**>>>** b = array([ 0.25, -1.58, 1.32, -0.34, -4.22])

**>>>** dot(a, b)

-14.803

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

**>>>** precios = array([200, 100, 500, 400, 400, 150])

**>>>** cantidades = array([1, 0, 0, 2, 1, 0])

**>>>** total\_a\_pagar = dot(precios, cantidades)

**>>>** total\_a\_pagar

1400

También se usa para calcular promedios ponderados:

**>>>** notas = array([45, 98, 32])

**>>>** ponderaciones = array([30, 30, 40]) / 100.

**>>>** nota\_final = dot(notas, ponderaciones)

**>>>** nota\_final

55.7

**Producto matriz-vector**

El **producto matriz-vector** es el vector de los productos internos El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función dot entre las filas de la matriz y el vector:

El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función dot:

**>>>** a = array([[-0.6, 4.8, -1.2],

[-2. , -3.6, -2.1],

[ 1.7, 4.9, 0. ]])

**>>>** x = array([-0.6, -2. , 1.7])

**>>>** dot(a, x)

array([-11.28, 4.83, -10.82])

**Producto matriz-matriz**

El **producto matriz-matriz** es la matriz de los productos internos entre las filas de la primera matriz y las columnas de la segunda:

Esta operación también es obtenida usando la función dot:

**>>>** a = array([[ 2, 8],

[-3, 7],

[-8, -5]])

**>>>** b array([[-3, -5, -6, -3],

[-9, -2, 3, -3]])

**>>>** dot(a, b)

array([[-78, -26, 12, -30],

[-54, 1, 39, -12],

[ 69, 50, 33, 39]])

La multiplicación de matrices puede ser vista como varios productos matriz-vector (usando como vectores todas las filas de la segunda matriz), calculados de una sola vez.

En resumen, al usar la función dot, la estructura del resultado depende de cuáles son los parámetros pasados:

dot(vector, vector) → número

dot(matriz, vector) → vector

dot(matriz, matriz) → matriz

**Resolución de sistemas lineales**

Repasemos el producto matriz-vector:

Esta operación tiene dos operandos: una matriz y un vector. El resultado es un vector. A los operandos los denominaremos respectivamente A y x, y al resultado, b.

Un problema recurrente en Ingeniería consiste en obtener cuál es el vector x cuando A y b son dados:

La ecuación matricial Ax=bAx=b es una manera abreviada de expresar un [sistema de ecuaciones lineales](http://es.wikipedia.org/wiki/Sistema_de_ecuaciones_lineales). Por ejemplo, la ecuación del diagrama es equivalente al siguiente sistema de tres ecuaciones que tiene las tres incógnitas ww, yy y zz:

36w+51y+13z52w+34y+74z7y+1.1z=3=45=3336w+51y+13z=352w+34y+74z=457y+1.1z=33

En matemáticas, este sistema se representa matricialmente así:

⎡⎣⎢36525134713741.1⎤⎦⎥⎡⎣⎢wyz⎤⎦⎥=⎡⎣⎢34533⎤⎦⎥[36511352347471.1][wyz]=[34533]

La teoría detrás de la resolución de problemas de este tipo usted la aprenderá en sus ramos de matemáticas. Sin embargo, como este tipo de problemas aparece a menudo en la práctica, aprenderemos cómo obtener rápidamente la solución usando Python.

Dentro de los varios módulos incluídos en NumPy (por ejemplo, ya vimos numpy.random), está el módulo numpy.linalg, que provee algunas funciones que implementan algoritmos de álgebra lineal, que es la rama de las matemáticas que estudia los problemas de este tipo. En este módulo está la función solve, que entrega la solución x de un sistema a partir de la matriz A y el vector b:

**>>>** a = array([[ 36. , 51. , 13. ],

**...**  [ 52. , 34. , 74. ],

**...**  [ 0. , 7. , 1.1]])

**>>>** b = array([ 3., 45., 33.])

**>>>** x = solve(a, b)

**>>>** x

array([-7.10829222, 4.13213834, 3.70457422])

Podemos ver que el vector x en efecto satisface la ecuación Ax = b:

**>>>** dot(a, x)

array([ 3., 45., 33.])

**>>>** b

array([ 3., 45., 33.])

Sin embargo, es importante tener en cuenta que los valores de tipo real casi nunca están representados de manera exacta en el computador, y que el resultado de un algoritmo que involucra muchas operaciones puede sufrir de algunos errores de redondeo. Por esto mismo, puede ocurrir que aunque los resultados se vean iguales en la consola, los datos obtenidos son sólo aproximaciones y no exactamente los mismos valores:

**>>>** (dot(a, x) == b).all()

False

**Interfaces gráficas**

Todo programa necesita contar con algún mecanismo para recibir la entrada y entregar la salida. Ya hemos visto dos maneras de hacer entrada:

* entrada por teclado (raw\_input), y
* entrada por archivo (for linea in archivo: ...);

y dos maneras de hacer salida:

* salida por consola (print), y
* salida por archivo (archivo.write(...)).

La mayoría de los programas que ocupamos a diario no funcionan así, sino que tienen una **interfaz gráfica**, compuesta por ventanas, menúes, botones y otros elementos, a través de los cuales podemos interactuar con el programa.

Los programas con interfaces gráficas son fundamentalmente diferentes a los programas con interfaces de texto. Los programas que hemos escrito hasta ahora se ejecutan completamente de principio a fin, deteniéndose sólo cuando debemos ingresar datos.

Los programas gráficos, por otra parte, realizan acciones sólo cuando ocurren ciertos eventos provocados por el usuario (como hacer clic en un botón, o escribir algo en una casilla de texto), y el resto del tiempo se quedan esperando que algo ocurra, sin hacer nada. El programa no tiene control sobre cuándo debe hacer algo. Esto requiere que los programas sean estructurados de una manera especial, que iremos aprendiendo de a poco.

Python incluye un módulo llamado Tkinter que provee todas las funciones necesarias, que deben ser importadas al principio del programa:

**from** **Tkinter** **import** \*

**Creación de la ventana**

El siguiente programa es la interfaz gráfica más simple que se puede crear:

**from** **Tkinter** **import** \*

w = Tk()

w.mainloop()

Haga la prueba: copie este programa en el editor de texto, guárdelo y ejecútelo. Debería aparecer una ventana vacía:



La sentencia w = Tk() crea la ventana principal del programa, y la asigna a la variable w. Toda interfaz gráfica debe tener una ventana principal en la que se irán agregando cosas. Esta línea va al principio del programa.

La sentencia w.mainloop() indica a la interfaz que debe quedarse esperando a que el usuario haga algo. Esta línea siempre debe ir al final del programa.

Al ejecutarlo, puede darse cuenta que el programa no termina. Esto ocurre porque la llamada al método mainloop() se «queda pegada» esperando que algo ocurra. Esto se llama un **ciclo de eventos**, y es simplemente un ciclo infinito que está continuamente esperando que algo ocurra.

Todos los programas con interfaz gráfica deben seguir esta estructura: la creación de la ventana al principio del programa y la llamada al ciclo de eventos al final del programa.

**Creación de widgets**

Un **widget** es cualquier cosa que uno puede poner en una ventana. Por ahora, veremos tres tipos de widgets sencillos, que son suficientes para crear una interfaz gráfica funcional:

* las **etiquetas** (Label) sirven para mostrar datos,
* los **botones** (Button) sirven para hacer que algo ocurra en el programa, y
* los **campos de entrada** (Entry) sirven para ingresar datos al programa.

En un programa en ejecución, estos widgets se ven así:



El Entry es análogo al raw\_input de los programas de consola: sirve para que el programa reciba la entrada. El Label es análogo al print: sirve para que el programa entregue la salida.

Un botón puede ser visto como un «llamador de funciones»: cada vez que un botón es presionado, se hace una llamada a la función asociada a ese botón. Los botones no tienen un análogo, pues los programas de consola se ejecutan de principio a fin inmediatamente, y por esto no necesitan que las llamadas a las funciones sean gatilladas por el usuario.

Para agregar un widget a un programa, hay que ocupar las funciones con los nombres de los widgets (Label, Button y Entry). Estas funciones reciben como primer parámetro obligatorio la ventana que contendrá el widget. Además, tienen parámetros opcionales que deben ser pasados usando la sintaxis de asignación de parámetros por nombre. Por ejemplo, el parámetro text sirve para indicar cuál es el texto que aparecerá en un botón o en una etiqueta.

Por ejemplo, la siguiente sentencia crea un botón con el texto Saludar, contenido en la ventana w:

b = Button(w, text='Saludar')

Si bien esto crea el botón y lo asigna a la variable b, el botón no es agregado a la ventana w inmediatamente: lo que hicimos fue simplemente decirle al botón cuál es su contenedor, para que lo tenga en cuenta al momento de ser agregado. Para que esto ocurra, debemos llamar al método pack, que es una manera de decirle al widget «empaquétate dentro de tu contenedor»:

b.pack()

Como referencia, el programa que crea la ventana de la imagen es el siguiente (¡pruébelo!):

**from** **Tkinter** **import** \*

w = Tk()

l = Label(w, text='Etiqueta')

l.pack()

b = Button(w, text='Boton')

b.pack()

e = Entry(w)

e.pack()

w.mainloop()

Los widgets van siendo apilados verticalmente, desde arriba hacia abajo, en el mismo orden en que van siendo apilados. Ya veremos cómo empaquetarlos en otras direcciones.

**Controladores**

Al crear un botón de la siguiente manera:

b = Button(w, text='Saludar')

no hay ninguna acción asociada a él. Al hacer clic en el botón, nada ocurrirá.

Para que ocurra algo al hacer clic en el botón, hay que asociarle una acción. Un **controlador** es una función que será ejecutada al hacer clic en un botón.

Los controladores deben ser funciones que no reciben ningún parámetro.

Por ejemplo, supongamos que queremos que el programa imprima el mensaje Hola en la consola cada vez que se haga clic en el botón que dice «Saludar». Primero, hay que crear el controlador:

**def** saludar():

**print** 'Hola'

Para asociar el controlador al botón, hay que pasarlo a través del parámetro command (en inglés: «orden») al momento de crear el botón:

b = Button(w, text='Saludar', command=saludar)

Esta línea significa: crear el botón b, contenido en la ventana w, que tenga el texto 'Saludar' y que al hacer clic en él se ejecute la función saludar.

El siguiente ejemplo es un programa completo que tiene dos botones: uno para saludar y otro para salir del programa. El controlador del segundo botón es la función exit, que ya viene con Python:

**from** **Tkinter** **import** \*

**def** saludar():

**print** 'Hola'

w = Tk()

l = Label(w, text='Hola progra')

l.pack()

b1 = Button(w, text='Saludar', command=saludar)

b1.pack()

b2 = Button(w, text='Salir', command=exit)

b2.pack()

w.mainloop()

El programa se ve así:



Ejecute el programa, y pruebe lo que ocurre al hacer clic en ambos botones.

**Modelos**

Mediante el uso de controladores, ya podemos hacer interfaces que hagan algo, pero que siguen teniendo una limitación: las interfaces sólo reaccionan a eventos que ocurren, pero no tienen memoria para recordar información.

Un **modelo** es un dato almacenado que está asociado a la interfaz. Usando modelos, se puede lograr que la interfaz vaya cambiando su estado interno a medida que ocurren eventos.

En general, a la hora de crear un programa con interfaz gráfica, debemos crear un modelo para cada dato que deba ser recordado durante el programa.

Tkinter rovee varios tipos de modelos, pero para simplificar podemos limitarnos a usar sólo modelos de tipo string. Un modelo puede ser creado de la siguiente manera:

m = StringVar()

Aquí, el modelo m es capaz de recordar un string

Para modificar el valor del modelo m, se debe usar el método set, que recibe el valor como único parámetro:

m.set('hola')

Para obtener el valor del modelo m, se debe usar el método get, que no recibe ningún parámetro:

s = m.get()

En este ejemplo, la variable s toma el valor 'hola'.

Como los modelos creados por StringVar almacenan datos de tipo string, hay que tener cuidado de hacer las conversiones apropiadas si se desea usar datos numéricos:

a = StringVar()

b = StringVar()

a.set(5) *# es convertido a string*

b.set(8) *# es convertido a string*

**print** a.get() + b.get() *# imprime 58*

**print** int(a.get()) + int(b.get()) *# imprime 13*

Usted podría preguntarse cuál es la razón para usar modelos en vez de usar las variables propias de Python, —es decir, las que son creadas mediante asignaciones— para almacenar los datos. Los modelos tienen la ventaja que es posible asociarlos a elementos de la interfaz que responden automáticamente cuando el valor del modelo cambia.

Por ejemplo, podemos asociar una etiqueta a un modelo. La etiqueta siempre mostrará en la interfaz el valor que tiene el modelo, incluso cuando éste cambie.

Para asociar un modelo a una etiqueta, hay que usar el parámetro textvariable:

x = StringVar()

l = Label(w, textvariable=x)

l.pack()

Cada vez que cambie el valor del modelo x, el texto de la etiqueta será actualizado inmediatamente.

También podemos asociar un campo de entrada a un modelo. El valor asociado al modelo siempre será el texto que está ingresado en el campo.

Para asociar un modelo a un campo de texto, también se usa el parámetro textvariable:

x = StringVar()

e = Entry(w, textvariable=x)

e.pack()

Cuando se obtenga el valor del modelo mediante la llamada x.get(), el valor retornado será lo que el usuario haya ingresado en el campo hasta ese momento.

**Resumen**

Para diseñar un programa que tiene una interfaz gráfica, hay tres elementos importantes que hay que tener en consideración.

* Los elementos que componen la interfaz. A esto se le suele denominar la **vista** del programa.
* Los **modelos** que mantienen el estado de la interfaz en todo momento.
* Los **controladores** que reaccionan a eventos del usuario.

Los controladores pueden interactuar con los modelos mediante sus métodos get y set. Los cambios en los modelos pueden verse reflejados en la vista.