Python es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un simple pero efectivo sistema de programación orientado a objetos. La elegante sintaxis de Python y su tipado dinámico, junto a su naturaleza interpretada lo convierten en un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en muchas áreas, para la mayoría de plataformas.

El intérprete de Python y la extensa librería estándar se encuentran disponibles libremente en código fuente y de forma binaria para la mayoría de las plataformas desde la Web de Python, https://www.python.org/ y se pueden distribuir libremente. El mismo sitio también contiene distribuciones y referencias a muchos módulos libres de Python de terceros, programas, herramientas y documentación adicional.

El intérprete de Python es fácilmente extensible con funciones y tipos de datos implementados en C o C++ (u otros lenguajes que permitan ser llamados desde C). Python también es apropiado como un lenguaje para extender aplicaciones modificables.

Python es fácil de utilizar y es un verdadero lenguaje de programación ofreciendo mucha más estructura y soporte para programas grandes que la que ofrecen scripts de shell o archivos por lotes. Por otro lado, Python también ofrece mayor comprobación de errores que C y siendo un *lenguaje de muy alto nivel* tiene tipos de datos de alto nivel incorporados como listas flexibles y diccionarios. Debido a sus tipos de datos más generales, Python es aplicable a más dominios que Awk o Perl, aunque hay muchas cosas que son tan sencillas en Python como en esos lenguajes.

Python te permite dividir tu programa en módulos que pueden reutilizarse en otros programas de Python. Tiene una gran colección de módulos estándar que puedes utilizar como la base de tus programas o como ejemplos para empezar a aprender Python. Algunos de estos módulos proporcionan cosas como entrada/salida de ficheros, llamadas a sistema, sockets e incluso interfaces a herramientas de interfaz gráfica como Tk.

Python es un lenguaje interpretado, lo cual puede ahorrarte mucho tiempo durante el desarrollo ya que no es necesario compilar ni enlazar. El intérprete puede usarse interactivamente, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones cuando se hace desarrollo de programas de abajo hacia arriba. Es también una calculadora de escritorio práctica

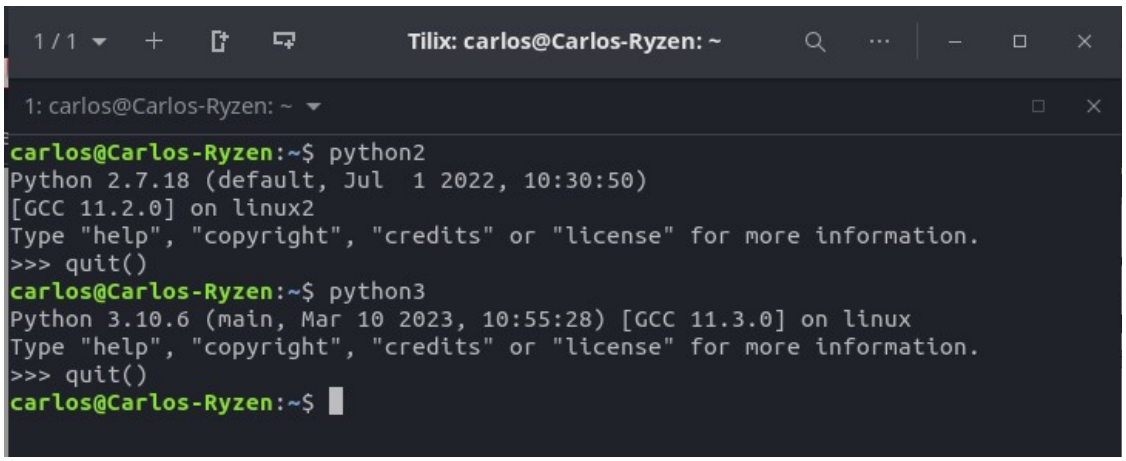
Usando el intérprete de Python

Para ser utilizado se debe abrir una Consola de Comandos (CMD en Windows) o bien una consola de Terminal (en GNU/Linux).

Al hacer, luego de instalado en Windows (Python esta preinstalado en GNU/Linux), este se podrá llamar en la terminal, escribiendo en la línea de comandos la instrucción `python`

Al existir al menos dos versiones muy difundidas (la 2.x y la 3.x), al ejecutar python, se abrirá la versión instalada por defecto.

Pero si desea abrir alguna versión en particular, se podrá indicar esta en la llamada.



Para salir del intérprete se puede escribir la llamada a la función `quit()`

También se puede salir del intérprete con estado de salida cero ingresando el carácter de fin de archivo (`Control-D` en Unix, `Control-Z` en Windows).

Las características para edición de líneas del intérprete incluyen edición interactiva, sustitución de historial y completado de código en sistemas que soportan la biblioteca GNU Readline.

Modo interactivo

Cuando se leen los comandos desde un terminal, se dice que el intérprete está en *modo interactivo*. En este modo, espera el siguiente comando con el *prompt primario*, generalmente tres signos de mayor que (`>>>`); para las líneas de continuación, aparece el *prompt secundario*, por defecto tres puntos (`...`). El intérprete imprime un mensaje de bienvenida que indica su número de versión y un aviso de copyright antes de imprimir el primer *prompt primario*:

```
python3.12
Python 3.12 (default, April 4 2022, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Codificación del código fuente

De forma predeterminada, los archivos fuente de Python se tratan como codificados en UTF-8. En esa codificación, los caracteres de la mayoría de los idiomas del mundo se pueden usar simultáneamente en literales, identificadores y comentarios, aunque la biblioteca estándar solo usa caracteres ASCII para los identificadores, una convención que debería seguir cualquier código que sea portable.Para mostrar todos estos caracteres correctamente, tu editor debe reconocer que el archivo es UTF-8, y debe usar una fuente que admita todos los caracteres del archivo.

Para declarar una codificación que no sea la predeterminada, se debe agregar una línea de comentario especial como la *primera* línea del archivo. La sintaxis es la siguiente:

```
# -*- coding: encoding -*-
```

donde *encoding* es uno de los `codecs` soportados por Python.

Por ejemplo, para declarar que se utilizará la codificación de Windows-1252, la primera línea del archivo de código fuente debe ser:

```
# -*- coding: cp1252 -*-
```

Una excepción a la regla de *primera línea* es cuando el código fuente comienza con una linea UNIX «shebang». En ese caso, la declaración de codificación debe agregarse como la segunda línea del archivo. Por ejemplo:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Usando Python como una calculadora

Números

El intérprete puede utilizarse como una simple calculadora; puedes introducir una expresión en él y este escribirá los valores. La sintaxis es sencilla: los operadores `+`, `-`, `*` y `/` funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis (`()`) pueden ser usados para agrupar. Por ejemplo:

```
>>>

2 + 2
4

50 - 5*6
20

(50 - 5*6) / 4
5.0

8 / 5  # division always returns a floating point number
1.6
```

Los números enteros (ej. `2`, `4`, `20`) tienen tipo `int`, los que tienen una parte fraccionaria (por ejemplo `5.0`, `1.6`) tiene el tipo `float`. Vamos a ver más acerca de los tipos numéricos más adelante en el tutorial.

La división (`/`) siempre retorna un número decimal de punto flotante. Para hacer floor division y obtener un número entero como resultado puede usarse el operador `//`; para calcular el resto puedes usar `%`:

```
>>>

17 / 3  # classic division returns a float
5.666666666666667
>>>

17 // 3  # floor division discards the fractional part
5

17 % 3  # the % operator returns the remainder of the division
2

5 * 3 + 2  # floored quotient * divisor + remainder
17
```

Con Python, es posible usar el operador `**` para calcular potencias

```
>>>

5 ** 2  # 5 al cuadrado
25

2 ** 7  # 2 a la septima
128
```

El signo igual (=) se usa para asignar un valor a una variable. Después, no se muestra ningún resultado antes del siguiente prompt interactivo:

```
>>>

width = 20

height = 5 * 9

width * height
900
```

Si una variable no está «definida» (no se le ha asignado un valor), al intentar usarla dará un error:

```
>>>

n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Cadenas de caracteres

Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples (`'...'`) o dobles (`"..."`) con el mismo resultado 2. `\` puede ser usado para escapar comillas:

```
>>>

'spam eggs'  # comilla simple
'spam eggs'

'doesn\'t'  # use \' para poder incorporar una comilla simple al texto...
"doesn't"

"doesn't"  # ...o bien use comillas dobles
"doesn't"
```

En el intérprete interactivo, la salida de caracteres está encerrada en comillas y los caracteres especiales se escapan con barras invertidas. Aunque esto a veces se vea diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra en comillas dobles si la cadena contiene una comilla simple y ninguna doble, de lo contrario es encerrada en comillas simples. La función `print()` produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales y escapados:

```
>>>

'"Isn\'t," they said.'
'"Isn\'t," they said.'

print('"Isn\'t," they said.')
"Isn't," they said.

s = 'First line.\nSecond line.'  # \n means newline

s  # without print(), \n is included in the output
'First line.\nSecond line.'

print(s)  # with print(), \n produces a new line
First line.
Second line.
```

Si no quieres que los caracteres precedidos por `\` se interpreten como caracteres especiales, puedes usar *cadenas sin formato* agregando una `r` antes de la primera comilla:

```
>>>

print('C:\algun\nombre')  # los caracteres \n indican una nueva línea!
C:\algun
ombre
```

```
print(r'C:\algun\nombre')  # atencion a la r antes de la comilla
C:\some\name
```

Las cadenas de texto literales pueden contener múltiples líneas. Una forma es usar triples comillas: `"""..."""` o `'''...'''`. Los fin de línea son incluidos automáticamente, pero es posible prevenir esto agregando una `\` al final de la línea. Por ejemplo:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

produce la siguiente salida (tener en cuenta que la línea inicial no está incluida):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Las cadenas se pueden concatenar (pegar juntas) con el operador `+` y se pueden repetir con `*`:

```
>>>

# 3 times 'un', followed by 'ium'

3 * 'un' + 'ium'
'unununium'
```

Dos o más *cadenas literales* (es decir, las encerradas entre comillas) una al lado de la otra se concatenan automáticamente.

```
>>>

'Py' 'thon'
'Python'
```

Esta característica es particularmente útil cuando quieres dividir cadenas largas:

```
>>>

text = ('Put several strings within parentheses '

        'to have them joined together.')

text
'Put several strings within parentheses to have them joined together.'
```

Las cadenas de texto se pueden *indexar* (subíndices), el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato diferente para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>>

word = 'Python'

word[0]  # character in position 0
'P'

word[5]  # character in position 5
'n'
```

Los índices también pueden ser números negativos, para empezar a contar desde la derecha:

```
>>>

word[-1]  # last character
'n'

word[-2]  # second-last character
'o'

word[-6]
'P'
```

Nótese que -0 es lo mismo que 0, los índice negativos comienzan desde -1.

Además de la indexación, también se admite el corte. Mientras que la indexación se usa para obtener caracteres individuales, el corte le permite obtener una subcadena:

```
word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'

word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Los índices de las rebanadas tienen valores por defecto útiles; el valor por defecto para el primer índice es cero, el valor por defecto para el segundo índice es la longitud de la cadena a rebanar.

```
>>>

word[:2]   # character from the beginning to position 2 (excluded)
'Py'

word[4:]   # characters from position 4 (included) to the end
'on'

word[-2:]  # characters from the second-last (included) to the end
```

```
'on'
```

Nótese cómo el inicio siempre se incluye y el final siempre se excluye. Esto asegura que `s[:i] + s[i:]` siempre sea igual a `s`:

```
>>>

word[:2] + word[2:]
'Python'

word[:4] + word[4:]
'Python'
```

Listas

Python tiene varios tipos de datos *compuestos*, utilizados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo.

```
>>>

squares = [1, 4, 9, 16, 25]

squares
[1, 4, 9, 16, 25]
```

Al igual que las cadenas (y todas las demás tipos integrados sequence), las listas se pueden indexar y segmentar:

```
>>>

squares[0]  # indexing returns the item
1

squares[-1]
25

squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Todas las operaciones de rebanado retornan una nueva lista que contiene los elementos pedidos. Esto significa que la siguiente rebanada retorna una shallow copy de la lista:

```
>>>

squares[:]
[1, 4, 9, 16, 25]
```

Las listas también admiten operaciones como concatenación:

```
>>>

squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A diferencia de las cadenas, que son immutable, las listas son de tipo mutable, es decir, es posible cambiar su contenido:

```
>>>

cubes = [1, 8, 27, 65, 125]  # something's wrong here

4 ** 3  # the cube of 4 is 64, not 65!
64

cubes[3] = 64  # replace the wrong value

cubes
[1, 8, 27, 64, 125]
```

También puede agregar nuevos elementos al final de la lista, utilizando el *método* `append()` (vamos a ver más sobre los métodos luego):

```
>>>

cubes.append(216)  # add the cube of 6

cubes.append(7 ** 3)  # and the cube of 7

cubes
[1, 8, 27, 64, 125, 216, 343]
```

También es posible asignar a una rebanada, y esto incluso puede cambiar la longitud de la lista o vaciarla totalmente:

```
>>>

letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']

# replace some values

letters[2:5] = ['C', 'D', 'E']

letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
# now remove them

letters[2:5] = []

letters
['a', 'b', 'f', 'g']

# clear the list by replacing all the elements with an empty list

letters[:] = []

letters
[]
```

La función predefinida `len()` también sirve para las listas

```
>>>

letters = ['a', 'b', 'c', 'd']

len(letters)
4
```

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
>>>

a = ['a', 'b', 'c']

n = [1, 2, 3]

x = [a, n]

x
[['a', 'b', 'c'], [1, 2, 3]]

x[0]
['a', 'b', 'c']

x[0][1]
'b'
```

Primeros pasos hacia la programación

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos más dos. Por ejemplo, podemos escribir una parte inicial de la serie de Fibonacci así:

```
# serie de Fibonacci:

# la suma de los dos elementos origina el próximo

a, b = 0, 1

while a < 10:

    print(a)

    a, b = b, a+b

0
1
1
2
3
5
8
```

Este ejemplo introduce varias características nuevas.

- La primera línea contiene una *asignación múltiple*: las variables `a` y `b` obtienen simultáneamente los nuevos valores 0 y 1. En la última línea esto se usa nuevamente, demostrando que las expresiones de la derecha son evaluadas primero antes de que se realice cualquiera de las asignaciones. Las expresiones del lado derecho se evalúan de izquierda a derecha.
- El bucle `while` se ejecuta mientras la condición (aquí: `a < 10`) sea verdadera. En Python, como en C, cualquier valor entero que no sea cero es verdadero; cero es falso. La condición también puede ser una cadena de texto o una lista, de hecho, cualquier secuencia; cualquier cosa con una longitud distinta de cero es verdadera, las secuencias vacías son falsas. La prueba utilizada en el ejemplo es una comparación simple. Los operadores de comparación estándar se escriben igual que en C: `<` (menor que), `>` (mayor que), `==` (igual a), `<=` (menor que o igual a), `>=` (mayor que o igual a) y `!=` (distinto a).
- El cuerpo del bucle está *indentado*: la indentación es la forma que usa Python para agrupar declaraciones. En el intérprete interactivo debes teclear un tabulador o espacio(s) para cada línea indentada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; todos los editores de texto modernos tienen la facilidad de agregar la indentación automáticamente. Cuando se ingresa una instrucción compuesta de forma interactiva, se debe finalizar con una línea en blanco para indicar que está completa (ya que

el analizador no puede adivinar cuando tecleaste la última línea). Nota que cada línea de un bloque básico debe estar sangrada de la misma forma.

- La función `print()` escribe el valor de los argumentos que se le dan. Difiere de simplemente escribir la expresión que se quiere mostrar (como hicimos antes en los ejemplos de la calculadora) en la forma en que maneja múltiples argumentos, cantidades de punto flotante, y cadenas. Las cadenas de texto son impresas sin comillas y un espacio en blanco se inserta entre los elementos, así puedes formatear cosas de una forma agradable:

```
>>>
i = 256*256

print('The value of i is', i)
The value of i is 65536
```

El parámetro nombrado *end* puede usarse para evitar el salto de linea al final de la salida, o terminar la salida con una cadena diferente:

```
>>>
a, b = 0, 1

while a < 1000:

    print(a, end=',')

    a, b = b, a+b

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Más herramientas para control de flujo

Además de la sentencia `while` que acabamos de introducir, Python soporta las sentencias de control de flujo que podemos encontrar en otros lenguajes, con algunos cambios.

La sentencia `if`

Tal vez el tipo más conocido de sentencia sea el `if`. Por ejemplo:

```
>>>
x = int(input("Please enter an integer: "))
Please enter an integer: 42

if x < 0:

    x = 0
```

```
    print('Negative changed to zero')

elif x == 0:

    print('Zero')

elif x == 1:

    print('Single')

else:

    print('More')

More
```

Puede haber cero o más bloques `elif`, y el bloque `else` es opcional. La palabra reservada `elif` es una abreviación de “else if”, y es útil para evitar un sangrado excesivo. Una secuencia `if` … `elif` … `elif` … sustituye las sentencias `switch` o `case` encontradas en otros lenguajes.

Si necesitas comparar un mismo valor con muchas constantes, o comprobar que tenga un tipo o atributos específicos puede que encuentres útil la sentencia `match`.

La sentencia `for`

La sentencia `for` en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia `for` de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo:

```
>>>

# Measure some strings:

words = ['gato', 'window', 'tio']

for w in words:

    print(w, len(w))

cat 4
window 6
tio 3
```

Código que modifica una colección mientras se itera sobre la misma colección puede ser complejo de hacer bien. Sin embargo, suele ser más directo iterar sobre una copia de la colección o crear una nueva colección:

```
# Crea una coleccion
users = {'Juan': 'activo', 'Éléonora': 'inactivo', '景太郎': 'activo'}

# Estrategia 1: Iterar sobre una copia
for user, status in users.copy().items():
    if status == 'inactivo':
        del users[user]

# Estrategia 2:Crea una nueva coleccion
active_users = {}
for user, status in users.items():
    if status == 'activo':
        active_users[user] = status
```

La función `range()`

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función integrada `range()`, la cual genera progresiones aritméticas:

```
>>>

for i in range(5):

    print(i)

0
1
2
3
4
```

El valor final dado nunca es parte de la secuencia; `range(10)` genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama “paso”):

```
>>>

list(range(5, 10))
[5, 6, 7, 8, 9]


list(range(0, 10, 3))
[0, 3, 6, 9]


list(range(-10, -100, -30))
[-10, -40, -70]
```

Para iterar sobre los índices de una secuencia, puedes combinar `range()` y `len()` así:

```
>>>

a = ['Mary', 'had', 'a', 'little', 'lamb']

for i in range(len(a)):

    print(i, a[i])

0 Mary
1 had
2 a
3 little
4 lamb
```

En la mayoría de los casos, sin embargo, conviene usar la función `enumerate()`

Las sentencias `break`, `continue`, y `else` en bucles

La sentencia `break`, como en C, termina el bucle `for` o `while` más anidado.

Las sentencias de bucle pueden tener una cláusula `else` que es ejecutada cuando el bucle termina, después de agotar el iterable (con `for`) o cuando la condición se hace falsa (con `while`), pero no cuando el bucle se termina con la sentencia `break`. Se puede ver el ejemplo en el siguiente bucle, que busca números primos:

```
>>>

for n in range(2, 10):

    for x in range(2, n):

        if n % x == 0:

            print(n, 'equals', x, '*', n//x)

            break

    else:

        # loop fell through without finding a factor

        print(n, 'is a prime number')

2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Sí, este es el código correcto. Fíjate bien: el `else` pertenece al ciclo `for`, no al `if`.)

Cuando se usa con un bucle, la cláusula `else` tiene más en común con el `else` de una sentencia `try` que con el de un `if`: en una sentencia `try` la cláusula `else` se ejecuta cuando no se genera ninguna excepción, y el `else` de un bucle se ejecuta cuando no hay ningún `break`

La declaración `continue`, también tomada de C, continua con la siguiente iteración del ciclo:

```
>>>
for num in range(2, 10):

    if num % 2 == 0:

        print("Encontré un número par", num)

        continue

    print("Encontré un número impar", num)

Encontré un número par 2
Encontré un número impar 3
Encontré un número par 4
Encontré un número impar 5
Encontré un número par 6
Encontré un número impar 7
Encontré un número par 8
Encontré un número impar 9
```

Tuplas

Una **tupla** es una secuencias ordenadas de objetos de distintos tipos.

Se construyen poniendo los elementos entre parentesis `( )` separados por comas.

Se caracterizan por:

- Tienen orden.
- Pueden contener elementos de distintos tipos.
- Son inmutables, es decir, no pueden alterarse durante la ejecución de un programa.

Se usan habitualmente para representar colecciones de datos una determinada estructura semántica, como por ejemplo un vector o una matriz.

```
# Tupla vacía
type(())
<class 'tuple'>
# Tupla con elementos de distintos tipos
(1, "dos", True)
# Vector
(1, 2, 3)
# Matriz
((1, 2, 3), (4, 5, 6))
```

Creación de tuplas mediante la función `tuple()`

Otra forma de crear tuplas es mediante la función `tuple()`.

- `tuple(c)` : Crea una tupla con los elementos de la secuencia o colección `c`.

Se pueden indicar los elementos separados por comas, mediante una cadena, o mediante una colección de elementos iterable.

```
>>> tuple()
()
>>> tuple(1, 2, 3)
(1, 2, 3)
>>> tuple("Python")
('P', 'y', 't', 'h', 'o', 'n')
>>> tuple([1, 2, 3])
(1, 2, 3)
```

Operaciones con tuplas

El acceso a los elementos de una tupla se realiza del mismo modo que en las listas. También se pueden obtener subtuplas de la misma manera que las sublistas.

Las operaciones de listas que no modifican la lista también son aplicables a las tuplas.

```
>>> a = (1, 2, 3)
>>> a[1]
```

```
2
>>> len(a)
3
>>> a.index(3)
2
>>> 0 in a
False
>>> b = ((1, 2, 3), (4, 5, 6))
>>> b[1]
(4, 5, 6)
>>> b[1][2]
6
```

Diccionarios

Un diccionario es una colección de pares formados por una *clave* y un *valor* asociado a la clave.

Se construyen poniendo los pares entre llaves `{ }` separados por comas, y separando la clave del valor con dos puntos `:`.

Se caracterizan por:

- No tienen orden.
- Pueden contener elementos de distintos tipos.
- Son mutables, es decir, pueden alterarse durante la ejecución de un programa.
- Las claves son únicas, es decir, no pueden repetirse en un mismo diccionario, y pueden ser de cualquier tipo de datos inmutable.

```
# Diccionario vacío
type({})
<class 'dict'>
# Diccionario con elementos de distintos tipos
{'nombre':'Alfredo', 'despacho': 218, 'email':'asalber@ceu.es'}
# Diccionarios anidados
{'nombre_completo':{'nombre': 'Alfredo', 'Apellidos': 'Sánchez Alberca'}}
```

Acceso a los elementos de un diccionario

- `d[clave]` devuelve el valor del diccionario `d` asociado a la clave `clave`. Si en el diccionario no existe esa clave devuelve un error.
- `d.get(clave, valor)` devuelve el valor del diccionario `d` asociado a la clave `clave`. Si en el diccionario no existe esa clave devuelve `valor`, y si no se especifica un valor por defecto devuelve `None`.

```
>>> a = {'nombre':'Alfredo', 'despacho': 218, 'email':'asalber@ceu.es'}
>>> a['nombre']
'Alfredo'
>>> a['despacho'] = 210
>>> a
{'nombre':'Alfredo', 'despacho': 218, 'email':'asalber@ceu.es'}
>>> a.get('email')
```

```
'asalber@ceu.es'
>>> a.get('universidad', 'CEU')
'CEU'
```

Operaciones que no modifican un diccionario

- `len(d)` : Devuelve el número de elementos del diccionario `d`.
- `min(d)` : Devuelve la mínima clave del diccionario `d` siempre que las claves sean comparables.
- `max(d)` : Devuelve la máxima clave del diccionario `d` siempre que las claves sean comparables.
- `sum(d)` : Devuelve la suma de las claves del diccionario `d`, siempre que las claves se puedan sumar.
- `clave in d` : Devuelve `True` si la clave `clave` pertenece al diccionario `d` y `False` en caso contrario.
- `d.keys()` : Devuelve un iterador sobre las claves de un diccionario.
- `d.values()` : Devuelve un iterador sobre los valores de un diccionario.
- `d.items()` : Devuelve un iterador sobre los pares clave-valor de un diccionario.

```
>>> a = {'nombre':'Alfredo', 'despacho': 218, 'email':'asalber@ceu.es'}
>>> len(a)
3
>>> min(a)
'despacho'
>>> 'email' in a
True
>>> a.keys()
dict_keys(['nombre', 'despacho', 'email'])
>>> a.values()
dict_values(['Alfredo', 218, 'asalber@ceu.es'])
>>> a.items()
dict_items([('nombre', 'Alfredo'), ('despacho', 218), ('email',
'asalber@ceu.es')])
```

Operaciones que modifican un diccionario

- `d[clave] = valor` : Añade al diccionario `d` el par formado por la clave `clave` y el valor `valor`.
- `d.update(d2)`. Añade los pares del diccionario `d2` al diccionario `d`.
- `d.pop(clave, alternativo)` : Devuelve del valor asociado a la clave `clave` del diccionario `d` y lo elimina del diccionario. Si la clave no está devuelve el valor `alternativo`.
- `d.popitem()` : Devuelve la tupla formada por la clave y el valor del último par añadido al diccionario `d` y lo elimina del diccionario.
- `del d[clave]` : Elimina del diccionario `d` el par con la clave `clave`.
- `d.clear()` : Elimina todos los pares del diccionario `d` de manera que se queda vacío.

```
>>> a = {'nombre':'Alfredo', 'despacho': 218, 'email':'asalber@ceu.es'}
>>> a['universidad'] = 'CEU'
>>> a
{'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es',
'universidad': 'CEU'}
>>> a.pop('despacho')
218
>>> a
{'nombre': 'Alfredo', 'email': 'asalber@ceu.es', 'universidad': 'CEU'}
>>> a.popitem()
('universidad', 'CEU')
>>> a
{'nombre': 'Alfredo', 'email': 'asalber@ceu.es'}
>>> del a['email']
>>> a
{'nombre': 'Alfredo'}
>>> a.clear()
>>> a
{}
```

Copia de diccionarios

Existen dos formas de copiar diccionarios:

- **Copia por referencia** `d1 = d2`: Asocia la la variable `d1` el mismo diccionario que tiene asociado la variable `d2`, es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de `l1` o `l2` afectará al mismo diccionario.
- **Copia por valor** `d1 = list(d2)`: Crea una copia del diccionario asociado a `d2` en una dirección de memoria diferente y se la asocia a `d1`. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de `l1` no afectará al diccionario de `l2` y viceversa.

```
>>> a = {1:'A', 2:'B', 3:'C'}
>>> # copia por referencia
>>> b = a
>>> b
{1:'A', 2:'B', 3:'C'}
>>> b.pop(2)
>>> b
{1:'A', 3:'C'}
>>> a
{1:'A', 3:'C'}

>>> a = {1:'A', 2:'B', 3:'C'}
>>> # copia por referencia
>>> b = dict(a)
>>> b
{1:'A', 2:'B', 3:'C'}
>>> b.pop(2)
>>> b
{1:'A', 3:'C'}
>>> a
{1:'A', 2:'B', 3:'C'}
```

Funciones (`def`)

Una función es un bloque de código que tiene asociado un nombre, de manera que cada vez que se quiera ejecutar el bloque de código basta con invocar el nombre de la función.

Para declarar una función se utiliza la siguiente sintaxis:

```
def <nombre-funcion> (<parámetros>):
    bloque código
    return <objeto>
```

```
>>> def bienvenida():
...     print('¡Bienvenido a Python!')
...     return
...
>>> type(bienvenida)
<class 'function'>
>>> bienvenida()
¡Bienvenido a Python!
```

Parámetros y argumentos de una función

Una función puede recibir valores cuando se invoca a través de unas variables conocidas como *parámetros* que se definen entre paréntesis en la declaración de la función. En el cuerpo de la función se pueden usar estos parámetros como si fuesen variables.

Los valores que se pasan a la función en una llamada o invocación concreta de ella se conocen como *argumentos* y se asocian a los parámetros de la declaración de la función.

```
>>> def bienvenida(nombre):
...     print('¡Bienvenido a Python', nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
```

Paso de argumentos a una función

Los argumentos se pueden pasar de dos formas:

- **Argumentos posicionales**: Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.
- **Argumentos nominales**: Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`.

```
>>> def bienvenida(nombre, apellido):
...     print('¡Bienvenido a Python', nombre, apellido + '!')
...     return
...
>>> bienvenida('Alfredo', 'Sánchez)
```

```
¡Bienvenido a Python Alfredo Sánchez!
>>> bienvenida(apellido = 'Sánchez', nombre = 'Alfredo')
¡Bienvenido a Python Alfredo Sánchez!
```

Retorno de una función

Una función puede devolver un objeto de cualquier tipo tras su invocación. Para ello el objeto a devolver debe escribirse detrás de la palabra reservada `return`. Si no se indica ningún objeto, la función no devolverá nada.

```
>>> def area_triangulo(base, altura):
...     return base * altura / 2
...
>>> area_triangulo(2, 3)
3
>>> area_triangulo(4, 5)
10
```

Una función puede devolver más de un objeto separándolos por comas tras la palabra reservada `return`. En tal caso, la función agrupará los objetos en una tupla y devolverá la tupla.

Argumentos por defecto

En la definición de una función se puede asignar a cada parámetro un argumento por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el argumento por defecto.

```
>>> def bienvenida(nombre, lenguaje = 'Python'):
...     print('¡Bienvenido a', lenguaje, nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
>>> bienvenida('Alf', 'Java')
¡Bienvenido a Java Alf!
```

Los parámetros con un argumento por defecto deben indicarse después de los parámetros sin argumentos por defectos. De lo contrario se produce un error.

Pasar un número indeterminado de argumentos

Por último, es posible pasar un número variable de argumentos a un parámetro. Esto se puede hacer de dos formas:

- `*parametro`: Se antepone un asterisco al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos separados por comas. Los argumentos se guardan en una lista que se asocia al parámetro.

```
>>> def menu(*platos):
```

```
...     print('Hoy tenemos: ', end='')
...     for plato in platos:
...         print(plato, end=', ')
...     return
...
>>> menu('pasta', 'pizza', 'ensalada')
Hoy tenemos: pasta, pizza, ensalada,
```

- `**parametro`: Se anteponen dos asteriscos al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos por pares `nombre = valor`, separados por comas. Los argumentos se guardan en un diccionario que se asocia al parámetro.

```
>>> def contacto(**info):
...     print('Datos del contacto)
...     for clave, valor in info.items():
...         print(clave, ":", valor)
...     return
...
>>> contacto(Nombre = "Alf", Email = "asalber@email.ar")
Datos del contacto
Nombre : Alf
Email : asalber@email.ar
>>> contacto(Nombre = "Alf", Email = "asalber@email.ar", Dirección = "Madrid")
Datos del contacto
Nombre : Alf
Email : asalber@email.ar
Dirección : Madrid
```

Ámbito de los parámetros y variables de una función

Los parámetros y las variables declaradas dentro de una función son de **ámbito local**, mientras que las definidas fuera de ella son de ámbito **ámbito global**.

Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función, es decir, cuando termina la ejecución de la función estas variables desaparecen y no son accesibles desde fuera de la función.

```
>>> def bienvenida(nombre):
...     lenguaje = 'Python'
...     print('¡Bienvenido a', lenguaje, nombre + '!')
...     return
...
>>> bienvenida('Alf')
¡Bienvenido a Python Alf!
>>> lenguaje
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lenguaje' is not defined
```

Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

```
>>> lenguaje = 'Java'
>>> def bienvenida():
...     lenguaje = 'Python'
...     print('¡Bienvenido a', lenguaje + '!')
...     return
...
>>> bienvenida()
¡Bienvenido a Python!
>>> print(lenguaje)
Java
```

Paso de argumentos por asignación

En Python los argumentos se pasan a una función por asignación, es decir, se asignan a los parámetros de la función como si fuesen variables locales. De este modo, cuando los argumentos son objetos mutables (listas, diccionarios, etc.) se pasa al parámetro una referencia al objeto, de manera que cualquier cambio que se haga dentro de la función mediante el parámetro asociado afectará al objeto original.

```
>>> primer_curso = ['Matemáticas', 'Física']
>>> def añade_asignatura(curso, asignatura):
...     curso.append(asignatura)
...     return
...
>>> añade_asignatura(primer_curso, 'Química')
>>> print(primer_curso)
['Matemáticas', 'Física', 'Química']
```

Las funciones son objetos

En Python las funciones son objetos como el resto de tipos de datos, de manera que es posible asignar una función a una variable y luego utilizar la variable para hacer la llamada a la función.

```
>>> def saludo(nombre):
...     print("Hola", nombre)
...     return
...
>>> bienvenida = saludo
>>> bienvenida("Alf")
Hola Alf
```

Esto permite, por tanto, pasar funciones como argumentos en la llamada a una función y que una función pueda devolver otras funciones.

```
>>> def impuesto(porcentaje):
...     def aplicar(base):
...             return base * porcentaje / 100
```

```
...     return aplicar
...
>>> iva = impuesto(21)
>>> iva(1000)
210.0
```

Funciones recursivas

Una función recursiva es una función que en su cuerpo contiene una llama a si misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función.

```
>>> def factorial(n):
...     if n == 0:
...         return 1
...     else:
...         return n * factorial(n-1)
...
>>> f(5)
120
```

Funciones recursivas múltiples

Una función recursiva puede invocarse a si misma tantas veces como quiera en su cuerpo.

```
>>> def fibonacci(n):
...     if n <= 1:
...         return n
...     else:
...         return fibonacci(n - 1) + fibonacci(n - 2)
...
>>> fibonacci(6)
8
```

Los riesgos de la recursión

Aunque la recursión permite resolver las tareas recursivas de forma más natural, hay que tener cuidado con ella porque suele consumir bastante memoria, ya que cada llamada a la función crea un nuevo ámbito local con las variables y los parámetros de la función.

En muchos casos es más eficiente resolver la tarea recursiva de forma iterativa usando bucles.

```
>>> def fibonacci(n):
...     a, b = 0, 1
...     for i in range(n):
...         a, b = b, a + b
...     return a
```

```
...
>>> fibonacci(6)
8
```

Documentación de funciones

Una práctica muy recomendable cuando se define una función es describir lo que la función hace en un comentario.

En Python esto se hace con un **docstring** que es un tipo de comentario especial se hace en la línea siguiente al encabezado de la función entre tres comillas simples `'''` o dobles `"""`.

Después se puede acceder a la documentación de la función con la función `help(<nombre-función>)`.

```
>>> def area_triangulo(base, altura):
... """Función que calcula el área de un triángulo.
...
... Parámetros:
...     - base: Un número real con la base del triángulo.
...     - altura: Un número real con la altura del triángulo.
... Salida:
...     Un número real con el área del triángulo de base y altura
especificadas.
... """
...     return base * altura / 2
...
>>> help(area_triangulo)
area_triangulo(base, altura)
    Función que calcula el área de un triángulo.

    Parámetros:
        - base: Un número real con la base del triángulo.
        - altura: Un número real con la altura del triángulo.
    Salida:
        Un número real con el área del triángulo de base y altura
especificadas.
```

Programación funcional

En Python las funciones son objetos de primera clase, es decir, que pueden pasarse como argumentos de una función, al igual que el resto de los tipos de datos.

```
>>> def aplica(funcion, argumento):
...     return funcion(argumento)
...
>>> def cuadrado(n):
...     return n*n
...
>>> def cubo(n):
...     return n**3
...
>>> aplica(cuadrado, 5)
25
>>> aplica(cubo, 5)
125
```

Funciones anónimas (`lambda`)

Existe un tipo especial de funciones que no tienen nombre asociado y se conocen como **funciones anónimas** o **funciones lambda**.

La sintaxis para definir una función anónima es

```
lambda <parámetros> : <expresión>
```

Estas funciones se suelen asociar a una variable o parámetro desde la que hacer la llamada.

```
>>> area = lambda base, altura : base * altura
>>> area(4, 5)
10
```

Aplicar una función a todos los elementos de una colección iterable (`map`)

`map(f, c)` : Devuelve una objeto iterable con los resultados de aplicar la función `f` a los elementos de la colección `c`. Si la función `f` requiere `n` argumentos entonces deben pasarse `n` colecciones con los argumentos. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
>>> def cuadrado(n):
...     return n * n
...
>>> list(map(cuadrado, [1, 2, 3])
[1, 4, 9]

>>> def rectangulo(a, b):
...     return a * b
...
>>> tuple(map(rectangulo, (1, 2, 3), (4, 5, 6)))
```

```
(4, 10, 18)
```

Filtrar los elementos de una colección iterable (filter)

`filter(f, c)` : Devuelve una objeto iterable con los elementos de la colección `c` que devuelven `True` al aplicarles la función `f`. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

`f` debe ser una función que recibe un argumento y devuelve un valor booleano.

```
>>> def par(n):
...     return n % 2 == 0
...
>>> list(filter(par, range(10))
[0, 2, 4, 6, 8]
```

Combinar los elementos de varias colecciones iterables (`zip`)

`zip(c1, c2, ...)` : Devuelve un objeto iterable cuyos elementos son tuplas formadas por los elementos que ocupan la misma posición en las colecciones `c1`, `c2`, etc. El número de elementos de las tuplas es el número de colecciones que se pasen. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
>>> asignaturas = ['Matemáticas', 'Física', 'Química', 'Economía']
>>> notas = [6.0, 3.5, 7.5, 8.0]
>>> list(zip(asignaturas, notas))
[('Matemáticas', 6.0), ('Física', 3.5), ('Química', 7.5), ('Economía', 8.0)]
>>> dict(zip(asignaturas, notas[:3]))
{'Matemáticas': 6.0, 'Física': 3.5, 'Química': 7.5}
```

Operar todos los elementos de una colección iterable (`reduce`)

`reduce(f, l)` : Aplicar la función `f` a los dos primeros elementos de la secuencia `l`. Con el valor obtenido vuelve a aplicar la función `f` a ese valor y el siguiente de la secuencia, y así hasta que no quedan más elementos en la lista. Devuelve el valor resultado de la última aplicación de la función `f`.

La función `reduce` está definida en el módulo `functools`.

```
>>> from functools import reduce
>>> def producto(n, m):
...     return n * m
...
>>> reduce(producto, range(1, 5))
24
```

Comprensión de colecciones

En muchas aplicaciones es habitual aplicar una función o realizar una operación con los elementos de una colección (lista, tupla o diccionario) y obtener una nueva colección de elementos transformados. Aunque esto se puede hacer recorriendo la secuencia con un bucle iterativo, y en programación funcional mediante la función `map`, Python incorpora un mecanismo muy potente que permite esto mismo de manera más simple.

Comprensión de listas

[*expresion* `for` *variable* `in` *lista* `if` *condicion*]

Esta instrucción genera la lista cuyos elementos son el resultado de evaluar la expresión *expresion*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
'Pablo':3}
>>> [nombre for (nombre, nota) in notas.items() if nota >= 5]
['Carmen', 'Juan', 'Mónica', 'María']
```

Comprensión de diccionarios

{*expresion-clave*`:`*expresion-valor* `for` *variables* `in` *lista* `if` *condicion*}

Esta instrucción genera el diccionario formado por los pares cuyas claves son el resultado de evaluar la expresión *expresion-clave* y cuyos valores son el resultado de evaluar la expresión *expresion-valor*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
>>> {palabra:len(palabra) for palabra in ['I', 'love', 'Python']}
{'I': 1, 'love': 4, 'Python': 6}
>>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
'Pablo':3}
>>> {nombre: nota +1 for (nombre, nota) in notas.items() if nota >= 5])
{'Carmen': 6, 'Juan': 9, 'Mónica': 10, 'María': 7}
```

Ficheros

Hasta ahora hemos visto como interactuar con un programa a través del teclado (entrada de datos) y la terminal (salida), pero en la mayor parte de las aplicaciones reales tendremos que leer y escribir datos en ficheros.

Al utilizar ficheros para guardar los datos estos perdurarán tras la ejecución del programa, pudiendo ser consultados o utilizados más tarde.

Las operaciones más habituales con ficheros son:

- Crear un fichero.
- Escribir datos en un fichero.
- Leer datos de un fichero.
- Borrar un fichero.

Creación y escritura de ficheros

Para crear un fichero nuevo se utiliza la siguiente función:

- `open(ruta, 'w')` : Crea el fichero con la ruta `ruta`, lo abre en modo escritura (el argumento 'w' significa *write*) y devuelve un objeto que lo referencia.

Si el fichero indicado por la ruta ya existe en el sistema, se reemplazará por el nuevo.

Una vez creado el fichero, para escribir datos en él se utiliza el siguiente método:

- `f.write(c)` : Escribe la cadena `c` en el fichero referenciado por `f` y devuelve el número de caracteres escritos.

```
>>> f = open('saludo.txt', 'w')
>>> f.write('¡Bienvenido a Python!')
21
```

Añadir datos a un fichero

Si en lugar de crear un fichero nuevo queremos añadir datos a un fichero existente se debe utilizar la siguiente función:

- `open(ruta, 'a')` : Abre el fichero con la ruta `ruta` en modo añadir (el argumento 'a' significa *append*) y devuelve un objeto que lo referencia.

Una vez abierto el fichero, se utiliza el método de escritura anterior y los datos se añaden al final del fichero.

```
>>> f = open('saludo.txt', 'a')
>>> f.write('\n¡Hasta pronto!')
15
```

Leer datos de un fichero

Para abrir un fichero en modo lectura se utiliza la siguiente función:

- `open(ruta, 'r')` : Abre el fichero con la ruta `ruta` en modo lectura (el argumento 'r' significa *read*) y devuelve un objeto que lo referencia.

Una vez abierto el fichero, se puede leer todo el contenido del fichero o se puede leer línea a línea. Para ello se utilizan las siguientes funciones:

- `f.read()` : Devuelve todos los datos contenidos en el fichero referenciado por `f` como una cadena de caracteres.
- `f.readlines()` : Devuelve una lista de cadenas de caracteres donde cada cadena es una linea del fichero referenciado por `f`.

```
>>> f = open('saludo.txt', 'r')
>>> print(f.read())
¡Bienvenido a Python!
¡Hasta pronto!

>>> f = open('saludo.txt', 'r')
>>> lineas = f.readlines()
>>> print(lineas)
['Bienvenido a Python!\n', '¡Hasta pronto!']
```

Cerrar un fichero

Para cerrar un fichero se utiliza el siguiente método:

`f.close()` : Cierra el fichero referenciado por el objeto `f`.

Cuando se termina de trabajar con un fichero conviene cerrarlo, sobre todo si se abre en modo escritura, ya que mientras está abierto en este modo no se puede abrir por otra aplicación. Si no se cierra explícitamente un fichero, Python intentará cerrarlo cuando estime que ya no se va a usar más.

```
>>> f = open('saludo.txt'):
>>> print(f.read())
¡Bienvenido a Python!
¡Hasta pronto!
>>> f.close()  # Cierre del fichero
>>> print(f.read())  # Produce un error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

La estructura `with open(...) as`

Para despreocuparnos del cierre de un fichero cuando ya no es necesario y no tener que cerrarlo explícitamente, se utiliza la siguiente estructura:

```
with open(ruta, modo) as f:
    bloque código
```

Esta estructura abre el fichero con la ruta `ruta` en el modo `modo` (`'w'` para escribir, `'a'` para añadir y `'r'` para leer) y devuelve una referencia al mismo en la variable `f`. El fichero permanece abierto mientras se ejecuta el bloque de código asociado y se cierra automáticamente cuando termina la ejecución del bloque.

```
>>> with open('saludo.txt', 'w') as f:
...     f.write("Hola de nuevo")
...
13
>>> with open('saludo.txt', 'r') as f:
...     print(f.read())
...
Hola de nuevo
>>> print(f.read())  # Produce un error al estar el fichero cerrado
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Renombrado y borrado de un fichero

Para renombra o borrar un fichero se utilizan funciones del módulo `os`.

`os.rename(ruta1, ruta2)` : Renombra y mueve el fichero de la ruta `ruta1` a la ruta `ruta2`.

`os.remove(ruta)` : Borra el fichero de la ruta `ruta`.

Antes de borrar o renombra un directorio conviene comprobar que existe para que no se produzca un error. Para ello se utiliza la función

`os.path.isfile(ruta)` : Devuelve `True` si existe un fichero en la ruta `ruta` y `False` en caso contrario.

Renombrado y borrado de un fichero o directorio

```
>>> import os
>>> f = 'saludo.txt'
>>> if os.path.isfile(f):
...     os.rename(f, 'bienvenida.txt') # renombrado
... else:
...     print('¡El fichero', f, 'no existe!')
...
>>> f = 'bienvenida.txt'
>>> if os.path.isfile(f):
...     os.remove(f) # borrado
... else:
...     print('¡El fichero', f, 'no existe!')
...
```

Creación, cambio y eliminación de directorios

Para trabajar con directorios también se utilizan funciones del módulo `os`.

`os.listdir(ruta)` : Devuelve una lista con los ficheros y directiorios contenidos en la ruta `ruta`.

`os.mkdir(ruta)` : Crea un nuevo directorio en la ruta `ruta`.

`os.chdir(ruta)` : Cambia el directorio actual al indicado por la ruta `ruta`.

`os.getcwd()` : Devuelve una cadena con la ruta del directorio actual.

`os.rmdir(ruta)` : Borra el directorio de la ruta `ruta`, siempre y cuando esté vacío.

Leer un fichero de internet

Para leer un fichero de internet hay que utilizar la función `urlopen` del módulo `urllib.request`.

`urlopen(url)` : Abre el fichero con la `url` especificada y devuelve un objeto del tipo fichero al que se puede acceder con los métodos de lectura de ficheros anteriores.

```
>>> from urllib import request
>>> f = request.urlopen('http://sisek.com.ar/Python-ej/archivo00.txt')
>>> datos = f.read()
>>> print(datos.decode('utf-8'))

===========================
Curso: programacion para NO programadores

Este es un archivo para el curso en el CSI

Pergamino

===========================
(fin del archivo)
```

Control de errores mediante excepciones

Python utiliza un objeto especial llamado **excepción** para controlar cualquier error que pueda ocurrir durante la ejecución de un programa.

Cuando ocurre un error durante la ejecución de un programa, Python crea una excepción. Si no se controla esta excepción la ejecución del programa se detiene y se muestra el error (*traceback*).

```
>>> print(1 / 0)  # Error al intentar dividir por 0.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Tipos de excepciones

Los principales excepciones definidas en Python son:

- `TypeError` : Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.
- `ZeroDivisionError` : Ocurre cuando se itenta dividir por cero.
- `OverflowError` : Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.
- `IndexError` : Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.
- `KeyError` : Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.
- `FileNotFoundError` : Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.
- `ImportError` : Ocurre cuando falla la importación de un módulo.

Consultar la documentaciónde Python para ver la lista de exepciones predefinidas.

Control de excepciones

`try - except - else`

Para evitar la interrución de la ejecución del programa cuando se produce un error, es posible controlar la exepción que se genera con la siguiente instrucción:

```
try:
    bloque código 1
except excepción:
    bloque código 2
else:
    bloque código 3
```

Esta instrucción ejecuta el primer bloque de código y si se produce un error que genera una excepción del tipo *excepción* entonces ejecuta el segundo bloque de código, mientras que si no se produce ningún error, se ejecuta el tercer bloque de código.

Control de excepciones

```
>>> def division(a, b):
...     try:
...         result = a / b
...     except ZeroDivisionError:
...         print('¡No se puede dividir por cero!')
...     else:
...         print(result)
...
>>> division(1, 0)
¡No se puede dividir por cero!
>>> division(1, 2)
```

```
0.5

>>> try:
...     f = open('fichero.txt')  # El fichero no existe
... except FileNotFoundError:
...     print('¡El fichero no existe!')
... else:
...     print(f.read())
¡El fichero no existe!
```

Módulos

El código de un programa en Python puede reutilizarse en otro importándolo. Cualquier fichero con código de Python reutilizable se conoce como *módulo* o *librería*.

Los módulos suelen contener funciones reutilizables, pero también pueden definir variables con datos simples o compuestos (listas, diccionarios, etc), o cualquier otro código válido en Python.

Python permite importar un módulo completo o sólo algunas partes de él. Cuando se importa un módulo completo, el intérprete de Python ejecuta todo el código que contiene el módulo, mientras que si solo se importan algunas partes del módulo, solo se ejecutarán esas partes.

Importación completa de módulos (`import`)

- `import M` : Ejecuta el código que contiene `M` y crea una referencia a él, de manera que pueden invocarse un objeto o función `f` definida en él mediante la sintaxis `M.f`.
- `import M as N` : Ejecuta el código que contiene `M` y crea una referencia a él con el nombre `N`, de manera que pueden invocarse un objeto o función `f` definida en él mediante la sintaxis `N.f`. Esta forma es similar a la anterior, pero se suele usar cuando el nombre del módulo es muy largo para utilizar un alias más corto.

Importación parcial de módulos (`from import`)

- `from M import f, g, ...` : Ejecuta el código que contiene `M` y crea referencias a los objetos `f, g, ...`, de manera que pueden ser invocados por su nombre. De esta manera para invocar cualquiera de estos objetos no hace falta precederlos por el nombre del módulo, basta con escribir su nombre.
- `from M import *` : Ejecuta el código que contiene `M` y crea referencias a todos los objetos públicos (aquellos que no empiezan por el carácter _) definidos en el módulo, de manera que pueden ser invocados por su nombre.

Cuando se importen módulos de esta manera hay que tener cuidado de que no haya coincidencias en los nombres de funciones, variables u otros objetos.

```
>>> import calendar
>>> print(calendar.month(2019, 4))
April 2019
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

>>> from math import *
>>> cos(pi)
-1.0
```

Módulos de la librería estándar más importantes

Python viene con una biblioteca de módulos predefinidos que no necesitan instalarse. Algunos de los más utilizados son:

- sys: Funciones y parámetros específicos del sistema operativo.
- os: Interfaz con el sistema operativo.
- os.path: Funciones de acceso a las rutas del sistema.
- io: Funciones para manejo de flujos de datos y ficheros.
- string: Funciones con cadenas de caracteres.
- datetime: Funciones para fechas y tiempos.
- math: Funciones y constantes matemáticas.
- statistics: Funciones estadísticas.
- random: Generación de números pseudo-aleatorios.

Otras librerías imprescindibles

Estas librerías no vienen en la distribución estándar de Python y necesitan instalarse.

- NumPy: Funciones matemáticas avanzadas y arrays.
- SciPy: Más funciones matemáticas para aplicaciones científicas.
- matplotlib: Análisis y representación gráfica de datos.
- Pandas: Funciones para el manejo y análisis de estructuras de datos.
- Request: Acceso a internet por http.

La librería Datetime

Para manejar fechas en Python se suele utilizar la librería `datetime` que incorpora los tipos de datos `date`, `time` y `datetime` para representar fechas y funciones para manejarlas. Algunas de las operaciones más habituales que permite son:

- Acceder a los distintos componentes de una fecha (año, mes, día, hora, minutos, segundos y microsegundos).
- Convertir cadenas con formato de fecha en los tipos `date`, `time` o `datetime`.
- Convertir fechas de los tipos `date`, `time` o `datetime` en cadenas formateadas de acuerdo a diferentes formatos de fechas.
- Hacer aritmética de fechas (sumar o restar fechas).
- Comparar fechas.

Los tipos de datos `date`, `time` y `datetime`

- `date(año, mes, dia)` : Devuelve un objeto de tipo `date` que representa la fecha con el `año`, `mes` y `dia` indicados.

- `time(hora, minutos, segundos, microsegundos)` : Devuelve un objeto de tipo `time` que representa un tiempo la `hora`, `minutos`, `segundos` y `microsegundos` indicados.
- `datetime(año, mes, dia, hora, minutos, segundos, microsegundos)` : Devuelve un objeto de tipo `datetime` que representa una fecha y hora con el `año`, `mes`, `dia`, `hora`, `minutos`, `segundos` y `microsegundos` indicados.

```
from datetime import date, time, datetime
>>> date(2020, 12, 25)
datetime.date(2020, 12, 25)
>>> time(13,30,5)
datetime.time(13, 30, 5)
>>> datetime(2020, 12, 25, 13, 30, 5)
datetime.datetime(2020, 12, 25, 13, 30, 5)
>>> print(datetime(2020, 12, 25, 13, 30, 5))
2020-12-25 13:30:05
```

_

Acceso a los componentes de una fecha

- `date.today()` : Devuelve un objeto del tipo `date` la fecha del sistema en el momento en el que se ejecuta.
- `datetime.now()`: Devuelve un objeto del tipo `datetime` con la fecha y la hora del sistema en el momento exacto en el que se ejecuta.
- `d.year` : Devuelve el año de la fecha `d`, puede ser del tipo `date` o `datetime`.
- `d.month` : Devuelve el mes de la fecha `d`, que puede ser del tipo `date` o `datetime`.
- `d.day` : Devuelve el día de la fecha `d`, que puede ser del tipo `date` o `datetime`.
- `d.weekday()` : Devuelve el día de la semana de la fecha `d`, que puede serpuede ser del tipo `date` o `datetime`.
- `t.hour` : Devuelve las horas del tiempo `t`, que puede ser del tipo `time` o `datetime`.
- `t.hour` : Devuelve los minutos del tiempo `t`, que puede ser del tipo `time` o `datetime`.
- `t.second` : Devuelve los segundos del tiempo `t`, que puede ser del tipo `time` o `datetime`.
- `t.microsecond` : Devuelve los microsegundos del tiempo `t`, que puede ser del tipo `time` o `datetime`.

```
>>> from datetime import date, time, datetime
>>> print(date.today())
2020-04-11
>>> dt = datetime.now()
>>> dt.year
2020
>>> dt.month
4
>>> dt.day
11
```

```
>>> dt.hour
22
>>> dt.minute
5
>>> dt.second
45
>>> dt.microsecond
1338
```

Conversión de fechas en cadenas con diferentes formatos

- `d.strftime(formato)` : Devuelve la cadena que resulta de transformar la fecha `d` con el formato indicado en la cadena `formato`. La cadena `formato` puede contener los siguientes marcadores de posición: `%Y` (año completo), `%y` (últimos dos dígitos del año), `%m` (mes en número), `%B` (mes en palabra), `%d` (día), `%A` (día de la semana), `%a` (día de la semana abrevidado), `%H` (hora en formato 24 horas), `%I` (hora en formato 12 horas), `%M` (minutos), `%S` (segundos), `%p` (AM o PM), `%C` (fecha y hora completas), `%x` (fecha completa), `%X` (hora completa).

```
>>> from datetime import date, time, datetime
>>> d = datetime.now()
>>> print(d.strftime('%d-%m-%Y'))
13-04-2020
>>> print(d.strftime('%A, %d %B, %y'))
Monday, 13 April, 20
>>> print(d.strftime('%H:%M:%S'))
20:55:53
>>> print(d.strftime('%H horas, %M minutos y %S segundos'))
20 horas, 55 minutos y 53 segundos
```

Conversión de cadenas en fechas

- `strptime(s, formato)` : Devuelve el objeto de tipo `date`, `time` o `datetime` que resulta de convertir la cadena `s` de acuerdo al formato indicado en la cadena `formato`. La cadena `formato` puede contener los siguientes marcadores de posición: `%Y` (año completo), `%y` (últimos dos dígitos del año), `%m` (mes en número), `%B` (mes en palabra), `%d` (día), `%A` (día de la semana), `%a` (día de la semana abrevidado), `%H` (hora en formato 24 horas), `%I` (hora en formato 12 horas), `%M` (minutos), `%S` (segundos), `%p` (AM o PM), `%C` (fecha y hora completas), `%x` (fecha completa), `%X` (hora completa).

```
>>> from datetime import date, time, datetime
>>> datetime.strptime('15/4/2020', '%d/%m/%Y')
datetime.datetime(2020, 4, 15, 0, 0)
>>> datetime.strptime('2020-4-15 20:50:30', '%Y-%m-%d %H:%M:%S')
datetime.datetime(2020, 4, 15, 20, 50, 30)
```

Aritmética de fechas

Para representar el tiempo transcurrido entre dos fechas se utiliza el tipo `timedelta`.

- `timedelta(dias, segundos, microsegundos)` : Devuelve un objeto del tipo `timedelta` que representa un intervalo de tiempo con los `dias`, `segundos` y `micorsegundos` indicados.
- `d1 - d2` : Devuelve un objeto del tipo `timedelta` que representa el tiempo transcurrido entre las fechas `d1` y `d2` del tipo `datetime`.
- `d + delta` : Devuelve la fecha del tipo `datetime` que resulta de sumar a la fecha `d` el intervalo de tiempo `delta`, donde `delta` es del tipo `timedelta`.

```
>>> from datetime import date, time, datetime, timedelta
>>> d1 = datetime(2020, 1, 1)
>>> d1 + timedelta(31, 3600)
datetime.datetime(2020, 2, 1, 1, 0)
>>> datetime.now() - d1
datetime.timedelta(days=132, seconds=1826, microseconds=895590)
```