Manual de Python

Alfredo Sánchez Alberca

Febrero 2020

Índice general

1 Introducción a Python

1.1 ¿Qué es Python?

Python es un lenguaje de programación de alto nivel multiparadigma que permite:

- Programación imperativa
- Programación funcional
- Programación orientada a objetos

Fue creado por Guido van Rossum en 1990 aunque actualmente es desarrollado y mantenido por la Python Software Foundation

1.2 Principales ventajas de Python

- Es de código abierto (certificado por la OSI).
- Es interpretable y compilable.
- Es fácil de aprender gracias a que su sintaxis es bastante legible para los humanos.
- Es un lenguaje maduro (29 años).
- Es fácilmente extensible e integrable en otros lenguajes (C, Java).
- Esta mantenido por una gran comunidad de desarrolladores y hay multitud de recursos para su aprendizaje.

1.3 Tipos de ejecución

1.3.1 Interpretado en la consola de Python

Se ejecuta cada instrucción que introduce el usuario de manera interactiva.

```
> python
>>> name = "UNO"
>>> print("Hola ", name)
Hola UNO
```

1.3.2 Interpretado en archivo

Se leen y se ejecutan una a una todas las instrucciones del archivo.

```
# Archivo hola.py
name = "UNO"
print("Hola ", name)
```

```
> python hola.py
Hola UNO
```

2 Tipos de datos simples

2.1 Tipos de datos primitivos simples

- **Números** (numbers): Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números.
  Ejemplo. 0, -1, 3.1415.
- **Cadenas** (strings): Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles. **Ejemplo**. 'Hola', "Adiós".
- **Booleanos** (boolean): Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

2.2 Tipos de datos primitivos compuestos (contenedores)

- **Listas** (lists): Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. Se representan con corchetes y los elementos se separan por comas. **Ejemplo**. [1, "dos", [3, 4], True].
- **Tuplas** (tuples). Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. A diferencia de las listas son inmutables, es decir, que no cambian durante la ejecución. Se representan mediante paréntesis y los elementos se separan por comas. **Ejemplo**. (1, 'dos', 3)
- **Diccionarios** (dictionaries): Colecciones de objetos con una clave asociada. Se representan con llaves, los pares separados por comas y cada par contiene una clave y un objeto asociado separados por dos puntos.
  Ejemplo. {'pi':3.1416, 'e':2.718}.

2.3 Clase de un dato (`type()`)

La clase a la que pertenece un dato se obtiene con el comando `type()`

```
>>> type(1)
<class 'int'>
>>> type("Hola")
<class 'str'>
>>> type([1, "dos", [3, 4], True])
<class 'list'>
>>>type({'pi':3.1416, 'e':2.718})
<class 'dict'>
>>>type((1, 'dos', 3))
<class 'tuple'>
```

2.4 Números (clases `int` y `float`)

Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números. Pueden ser enteros (`int`) o reales (`float`).

```
>>> type(1)
<class 'int'>
>>> type(-2)
<class 'int'>
>>> type(2.3)
<class 'float'>
```

2.4.1 Operadores aritméticos

- Operadores aritméticos: `+` (suma), `-` (resta), `*` (producto), `/` (cociente), `//` (cociente división entera), `%` (resto división entera), `**` (potencia).

Orden de prioridad de evaluación:

---

1. Funciones predefinidas
2. Potencias
3. Productos y cocientes
4. Sumas y restas

Se puede saltar el orden de evaluación utilizando paréntesis `( )`.

```
>>> 2+3
5
>>> 5*-2
-10
>>> 5/2
2.5
>>> 5//2
2
>>> (2+3)**2
25
```

2.4.2 Operadores lógicos con números

Devuelven un valor lógico o booleano.

- Operadores lógicos: `==` (igual que), `>` (mayor que), `<` (menor que), `>=` (mayor o igual que), `<=` (menor o igual que), `!=` (distinto de).

```
>>> 3==3
True
>>> 3.1<=3
False
```

```
>>> -1!=1
True
```

2.5 Cadenas (clase `str`)

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas ' o dobles ".

```
'Python'

"123"

'True'

# Cadena vacía
''

# Cadena con un espacio en blanco
' '

# Cambio de línea
'\n'

# Tabulador
'\t'
```

2.5.1 Acceso a los elementos de una cadena

Cada carácter tiene asociado un índice que permite acceder a él.

Cadena	P	y	t	h	o	n
Índice positivo	0	1	2	3	4	5
Índice negativo	-6	-5	-4	-3	-2	-1

- `c[i]` devuelve el carácter de la cadena `c` con el índice `i`.

El índice del primer carácter de la cadena es 0.

También se pueden utilizar índices negativos para recorrer la cadena del final al principio.

El índice del último carácter de la cadena es -1.

```
>>> 'Python'[0]
'P'
>>> 'Python'[1]
'y'

>>> 'Python'[-1]
'n'

>>> 'Python'[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
IndexError: string index out of range
```

2.5.2 Subcadenas

- `c[i:j:k]` : Devuelve la subcadena de `c` desde el carácter con el índice `i` hasta el carácter anterior al índice `j`, tomando caracteres cada `k`.

```
>>> 'Python'[1:4]
'yth'
>>> 'Python'[1:1]
''
>>> 'Python'[2:]
'thon'
>>> 'Python'[:-2]
'Pyth'
>>> 'Python'[:]
'Python'
>>> 'Python'[0:6:2]
'Pto'
```

2.5.3 Operaciones con cadenas

- `c1 + c2`: Devuelve la cadena resultado de concatenar las cadenas `c1` y `c2`.
- `c * n`: Devuelve la cadena resultado de concatenar `n` copias de la cadena `c`.
- `c1 in c2`: Devuelve `True` si `c1` es una cadena concenida en `c2` y `False` en caso contrario.
- `c1 not in c2`: Devuelve `True` si `c1` es una cadena no contenida en `c2` y `False` en caso contrario.

```
>>> 'Me gusta ' + 'Python'
'Me gusta Python'
>>> 'Python' * 3
'PythonPythonPython'
>>> 'y' in 'Python'
True

>>> 'tho'in  'Python'
True

>>> 'to'not  in 'Python'
True
```

2.5.4 Operaciones de comparación de cadenas

- `c1 == c2` : Devuelve `True` si la cadena `c1` es igual que la cadena `c2` y `False` en caso contrario.
- `c1 > c2` : Devuelve `True` si la cadena `c1` sucede a la cadena `c2` y `False` en caso contrario.
- `c1 < c2` : Devuelve `True` si la cadena `c1` antecede a la cadena `c2` y `False` en caso contrario.
- `c1 >= c2` : Devuelve `True` si la cadena `c1` sucede o es igual a la cadena `c2` y `False` en caso contrario.

- `c1 <= c2` : Devuelve `True` si la cadena `c1` antecede o es igual a la cadena `c2` y `False` en caso contrario.
- `c1 != c2` : Devuelve `True` si la cadena `c1` es distinta de la cadena `c2` y `False` en caso contrario.

Utilizan el orden establecido en el código ASCII.

```
>>>'Python'  == 'python'
False

>>>          < 'python'
'Python'
True

>>> 'a' > 'Z'
True
>>> 'A' >= 'Z'
False
>>> '' < 'Python'
True
```

2.5.5 Funciones de cadenas

- `len(c)` : Devuelve el número de caracteres de la cadena `c`.
- `min(c)` : Devuelve el carácter menor de la cadena `c`.
- `max(c)` : Devuelve el carácter mayor de la cadena `c`.
- `c.upper()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en mayúsculas.
- `c.lower()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en minúsculas.
- `c.title()` : Devuelve la cadena con los mismos caracteres que la cadena `c` con el primer carácter en mayúsculas y el resto en minúsculas.
- `c.split(delimitador)` : Devuelve la lista formada por las subcadenas que resultan de partir la cadena `c` usando como delimitador la cadena `delimitador`. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

```
>>> len('Python')
6
>>> min('Python')
'P'

>>> max('Python')
'y'

>>> 'Python'.upper()
'PYTHON'
>>> 'A,B,C'.split(',')
['A', 'B', 'C']
>>> 'I love Python'.split()
['I', 'love', 'Python']
```

2.5.6 Cadenas formateadas (`format()`)

- `c.format(valores)`: Devuelve la cadena `c` tras sustituir los valores de la secuencia `valores` en los marcadores de posición de `c`. Los marcadores de posición se indican mediante llaves `{}` en la cadena `c`, y el reemplazo de los valores se puede realizar por posición, indicando en número de orden del valor dentro de las llaves, o por nombre, indicando el nombre del valor, siempre y cuando los valores se pasen con el formato `nombre = valor`.

```
>>> 'Un {} vale {} {}'.format('€', 1.12, '$')
'Un € vale 1.12 $'
>>> 'Un {2} vale {1} {0}'.format('€', 1.12, '$')
'Un $ vale 1.12 €'
>>> 'Un {moneda1} vale {cambio} {moneda2}'.format(moneda1 = '€',
    cambio = 1.12, moneda2 = '$')
'Un € vale 1.12 $'
```

Los marcadores de posición, a parte de indicar la posición de los valores de reemplazo, pueden indicar también el formato de estos. Para ello se utiliza la siguiente sintaxis:

- `{:n}` : Alinea el valor a la izquierda rellenando con espacios por la derecha hasta los `n` caracteres.
- `{:>n}` : Alinea el valor a la derecha rellenando con espacios por la izquierda hasta los `n` caracteres.
- `{:^n}` : Alinea el valor en el centro rellenando con espacios por la izquierda y por la derecha hasta los `n` caracteres.
- `{:nd}` : Formatea el valor como un número entero con `n` caracteres rellenando con espacios blancos por la izquierda.
- `{:n.mf}` : Formatea el valor como un número real con un tamaño de `n` caracteres (incluído el separador de decimales) y `m` cifras decimales, rellenando con espacios blancos por la izquierda.

```
>>> 'Hoy es {:^10},        {:10} y pasado {:>10}'.format('lunes', '
   mañana martes',
   'miércoles')
'Hoy es   lunes   , mañana martes      y pasado miércoles'
>>> 'Cantidad {:5d}'.format(12)'
'Cantidad    12'
>>> 'Pi vale {:8.4f}'.format(3.141592)
'Pi vale   3.1416'
```

2.6 Datos lógicos o booleanos (clase `bool`)

Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

`False` tiene asociado el valor 0 y `True` tiene asociado el valor 1.

2.6.1 Operaciones con valores lógicos

- Operadores lógicos: == (igual que), > (mayor), < (menor), >= (mayor o igual que), <= (menor o igual que), != (distinto de).
- `not b` (negación) : Devuelve `True` si el dato booleano `b` es `False` , y `False` en caso contrario.
- `b1 and b2` : Devuelve `True` si los datos booleanos `b1` y `b2` son `True`, y `False` en caso contrario.
- `b1 or b2` : Devuelve `True` si alguno de los datos booleanos `b1` o `b2` son `True`, y `False` en caso contrario.

2.6.2 Tabla de verdad

x	y	not x	x and y	x or y
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

```
>>> not True
False
>>> False or True
True
>>> True and False
False
>>> True and True
True
```

2.7 Conversión de datos primitivos simples

Las siguientes funciones convierten un dato de un tipo en otro, siempre y cuando la conversión sea posible.

- `int()` convierte a entero.

  Ejemplo. `int('12')12`

  `int(True)1` `int('c')` Error

- `float()` convierte a real.

  Ejemplo. `float('3.14')3.14`

  `float(True)1.0`

  `float('III')` Error

- `str()` convierte a cadena.

  Ejemplo. `str(3.14)'3.14'`

  `str(True)'True'`

- `bool()` convierte a lógico.

  Ejemplo. `bool('0')False`

  `bool('3.14')True`

  `bool('')False`

  `bool('Hola')True`

2.8 Variables

Una variable es un identificador ligado a algún valor. Reglas para nombrarlas:

- Comienzan siempre por una letra, seguida de otras letras o números.
- No se pueden utilizarse palabras reservadas del lenguaje.

A diferencia de otros lenguajes no tienen asociado un tipo y no es necesario declararlas antes de usarlas (tipado dinámico).

Para asignar un valor a una variable se utiliza el operador = y para borrar una variable se utiliza la instrucción `del`.

```
1  lenguaje = 'Python'
2  x = 3.14
3  y = 3 + 2
4  # Asignación múltiple (asignará a a1 el valor 1 y a a2 el valor 2)
5  a1, a2 = 1, 2
6  # Intercambio de valores
7  a, b = b, a
8  # Incremento (equivale a x = x + 2)
9  x += 2
10 # Decremento (equivale a x = x - 1)
11 x -= 1
12 # Valor no definido
```

```
13 x = None
14 del x
```

2.9 Entrada por terminal (`input()`)

Para asignar a una variable un valor introducido por el usuario en la consola se utiliza la instrucción

`input(mensaje)`: Muestra la cadena `mensaje` por la terminal y devuelve una cadena con la entrada del usuario.

El valor devuelto siempre es una cadena, incluso si el usuario introduce un dato numérico.

```
1 >>> language = input('¿Cuál es tu lenguaje favorito? ')
2 ¿Cuál es tu lenguaje favorito? Python
3 >>> language
4 'Python'
5 >>> age = input('¿Cuál es tu edad? ')
6 ¿Cuál es tu edad? 20
7 >>> age
8 '20'
```

2.9.1 Salida por terminal (`print()`)

Para mostrar un dato por la terminal se utiliza la instrucción

`print(dato1, ..., sep='', end='\n', file=sys.stdout)`

donde

- `dato1, ...` son los datos a imprimir y pueden indicarse tantos como se quieran separados por comas.
- `sep` establece el separador entre los datos, que por defecto es un espacio en blanco `''`.
- `end` indica la cadena final de la impresión, que por defecto es un cambio de línea `\n`.
- `file` indica la dirección del flujo de salida, que por defecto es la salida estándar `sys.stdout`.

```
1 >>> print('Hola')
2 Hola
3 >>> name = 'UNO'
```

```
>>> print('Hola', name)
Hola UNO
>>> print('pi es', 3.1415)
pi es 3.1415
>>> print('Hola', name, sep='')
HolaUNO
>>> print('Hola', name, end='!\n')
Hola UNO!
```

3 Estructuras de control

3.1 Condicionales (`if`)

```
if condición1:
bloque código
elif condición2:
  bloque código
...
else :
  bloque código
```

Evalúa la expresión lógica `condición1` y ejecuta el primer bloque de código si es `True`; si no, evalúa la siguientes condiciones hasta llegar a la primera que es `True` y ejecuta el bloque de código asociado. Si ninguna condición es `True` ejecuta el bloque de código después de **`else`**:.

Pueden aparecer varios bloques `elif` pero solo uno **`else`** al final.

Los bloques de código deben estar indentados por 4 espacios.

La instrucción condicional permite evaluar el estado del programa y tomar decisiones sobre qué código ejecutar en función del mismo.

```
>>> edad = 14
>>> if edad <= 18 :
...     print('Menor')
... elif edad > 65:
...     print('Jubilado')
... else:
...     print('Activo')
...
Menor
>>> age = 20
```

```
>>> if edad <= 18 :
...     print('Menor')
... elif edad > 65:
```

```
...     print('Jubilado')
... else:
...     print('Activo')
...
Activo
```

3.2 Bucles condicionales (`while`)

```
whilecondición:
  bloque código
```

Repite la ejecución del bloque de código mientras la expresión lógica `condición` sea cierta. Se puede interrumpir en cualquier momento la ejecución del bloque de código con la instrucción **`break`**.

El bloque de código debe estar indentado por 4 espacios.

```
>>> # Pregunta al usuario por un número hasta que introduce 0.
>>> num = None
>>> while num != 0:
...     num = int(input('Introduce un número: '))
...

Introduce un número: 2
Introduce un número: 1
Introduce un número: 0
>>>
```

Alternativa:

```
>>> # Pregunta al usuario por un número hasta que introduce 0.
>>> while True:
...     num = int(input('Introduce un número: '))
...     if num == 0:
...         break
...

Introduce un número: 2
Introduce un número: 1
Introduce un número: 0
>>>
```

3.3 Bucles iterativos (`for`)

```
for i in secuencia:
bloque código
```

Repite la ejecución del bloque de código para cada elemento de la secuencia `secuencia`, asignado dicho elemento a `i` en cada repetición.

Se puede interrumpir en cualquier momento la ejecución del bloque de código con la instrucción **`break`** o saltar la ejecución para un determinado elemento de la secuencia con la instrucción **`continue`**.

El bloque de código debe estar indentado por 4 espacios.

Se utiliza fundamentalmente para recorrer colecciones de objetos como cadenas, listas, tuplas o diccionarios.

A menudo se usan con la instrucción `range`:

- `range(fin)` : Genera una secuencia de números enteros desde 0 hasta `fin-1`.
- `range(inicio, fin, salto)` : Genera una secuencia de números enteros desde `inicio` hasta `fin-1` con un incremento de `salto`.

```
>>> palabra = 'Python'
>>> for letra in palabra:
...     print(letra)
...
P
y
t
h
o
n
>>> for i in range(1, 10, 2):
...     print(i, end=", ")
...
1, 3, 5, 7, 9, >>>
```

4 Tipos de datos estructurados

4.1 Listas

Una **lista** es una secuencia ordenada de objetos de distinto tipo.

Se construyen poniendo los elementos entre corchetes `[]` separados por comas. Se caracterizan por:

- Tienen orden.
- Pueden contener elementos de distintos tipos.
  - Son mutables, es decir, pueden alterarse durante la ejecución de un programa.

```
# Lista vacía
>>> type([])
<class 'list'>
# Lista con elementos de distintos tipos
>>> [1, "dos", True]
# Listas anidadas
>>> [1, [2, 3], 4]
```

4.1.1 Creación de listas mediante la función `list()`

Otra forma de crear listas es mediante la función `list()`.

- `list(c)` : Crea una lista 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.

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

4.1.2 Acceso a los elementos de una lista

Se utilizan los mismos operadores de acceso que para cadenas de caracteres.

- `l[i]` : Devuelve el elemento de la lista `l` con el índice `i`.

El índice del primer elemento de la lista es 0.

```
>>> a = ['P', 'y', 't', 'h', 'o', 'n']
>>> a[0]
'P'
```

```
>>> a[5]
'n'
>>> a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> a[-1]
'n'
```

4.1.3 Sublistas

- `l[i:j:k]` : Devuelve la sublista desde el elemento de `l` con el índice `i` hasta el elemento anterior al índice `j`, tomando elementos cada `k`.

```
>>> a = ['P', 'y', 't', 'h', 'o', 'n']
>>> a[1:4]
['y', 't', 'h']
>>> a[1:1]
[]
>>> a[:-3]
['y', 't', 'h']
>>> a[:]
['P', 'y', 't', 'h', 'o', 'n']
>>> a[0:6:2]
['P', 't',
'o']
```

4.1.4 Operaciones que no modifican una lista

- `len(l)`: Devuelve el número de elementos de la lista `l`.
- `min(l)`: Devuelve el mínimo elemento de la lista `l` siempre que los datos sean comparables.
- `max(l)`: Devuelve el máximo elemento de la lista `l` siempre que los datos sean comparables.
- `sum(l)`: Devuelve la suma de los elementos de la lista `l`, siempre que los datos se puedan sumar.
- `dato in l`: Devuelve `True` si el dato `dato` pertenece a la lista `l` y `False` en caso contrario.
- `l.index(dato)`: Devuelve la posición que ocupa en la lista `l` el primer elemento con valor `dato`.
- `l.count(dato)`: Devuelve el número de veces que el valor `dato` está contenido en la lista `l`.

- `all(l)` : Devuelve `True` si todos los elementos de la lista `l` son `True` y `False` en caso contrario.
- `any(l)` : Devuelve `True` si algún elemento de la lista `l` es `True` y `False` en caso contrario.

```
>>> a = [1, 2, 2, 3]
>>> len(a)
4
>>> min(a)
1
>>> max(a)
3
>>> sum(a)
8
>>> 3 in a
True
>>> a.index(2)
1
>>> a.count(2)
2
>>> all(a)
True
>>> any([0, False, 3<2])
False
```

4.1.5 Operaciones que modifican una lista

- `l1 + l2`: Crea una nueva lista concatenando los elementos de la listas `l1` y `l2`.
- `l.append(dato)`: Añade `dato` al final de la lista `l`.
- `l.extend(sequencia)`: Añade los datos de `sequencia` al final de la lista `l`.
- `l.insert(índice, dato)`: Inserta `dato` en la posición `índice` de la lista `l` y desplaza los elementos una posición a partir de la posición `índice`.
- `l.remove(dato)`: Elimina el primer elemento con valor `dato` en la lista `l` y desplaza los que están por detrás de él una posición hacia delante.
- `l.pop([índice])`: Devuelve el dato en la posición `índice` y lo elimina de la lista `l`, desplazando los elementos por detrás de él una posición hacia delante.
- `l.sort()`: Ordena los elementos de la lista `l` de acuerdo al orden predefinido, siempre que los elementos sean comparables.
- `l.reverse()`: invierte el orden de los elementos de la lista `l`.

```
>>> a = [1, 3]
>>> b = [2 , 4, 6]
>>> a.append(5)
>>> a
```

```
[1, 3, 5]
>>> a.remove(3)
>>> a
[1, 5]
>>> a.insert(1, 3)
>>> a
[1, 3, 5]
>>> b.pop()
6
>>> c = a + b
>>> c
[1, 3, 5, 2, 4]
>>> c.sort()
>>> c
[1, 2, 3, 4, 5]
>>> c.reverse()
>>> c
[5, 4, 3, 2, 1]
```

4.1.6 Copia de listas

Existen dos formas de copiar listas:

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

```
>>> a = [1, 2, 3]
>>> # copia por referencia
>>> b = a
>>> b
[1, 2, 3]
```

```
>>> b.remove(2)
>>> b
[1, 3]
>>> a
[1, 3]
```

```
>>> a = [1, 2, 3]
>>> # copia por referencia
>>> b = list(a)
>>> b
[1, 2, 3]
>>> b.remove(2)
>>> b
[1, 3]
>>> a
[1, 2, 3]
```

4.2 Tuplas

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

Se construyen poniendo los elementos entre paréntesis `()` 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))
```

4.2.1 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)
```

4.2.2 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
```

4.3 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':'Jorge', 'despacho': 218, 'email':'jorge@uno.com.ar'}
# Diccionarios anidados
{'nombre_completo':{'nombre': 'Juan', 'Apellidos': 'Pérez' }}
```

4.3.1 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':'Jorge', 'despacho': 218, 'email':'jorge@uno.com.ar'}
>>> a['nombre']
'Jorge'
>>> a['despacho'] = 210
>>> a
{'nombre':'Jorge', 'despacho': 218, 'email':'jorge@uno.com.ar'}
>>> a.get('email')
'jorge@uno.com.ar'
>>> a.get('universidad', 'UNO')
'UNO'
```

4.3.2 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':'Jorge', 'despacho': 218, 'email':'jorge@uno.com.ar'}
>>> len(a)
3
>>> min(a)
'despacho'
>>> 'email' in a
True
>>> a.keys()
dict_keys(['nombre', 'despacho', 'email'])
>>> a.values()
dict_values(['Jorge', 218, 'jorge@uno.com.ar'])
>>> a.items()
dict_items([('nombre', 'Jorge'), ('despacho', 218), ('email', '
    jorge@uno.com.ar')])
```

4.3.3 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':'Jorge', 'despacho': 218, 'email':'jorge@uno.com.ar'}
>>> a['universidad'] = 'UNO'
>>> a
{'nombre': 'Jorge', 'despacho ': 218, 'email': ' jorge@uno.com.ar', '
    universidad': 'UNO'}
>>> a.pop('despacho')

218
```

```
>>> a
{'nombre': 'Jorge', 'email':  'jorge@uno.com.ar','universidad': 'UNO'}
>>> a.popitem()
('universidad', 'UNO')
>>> a
{'nombre': 'Jorge', 'email':  'jorge@uno.com.ar'}
>>> del a['email']
>>> a
{'nombre': 'Jorge'}
>>> a.clear()
>>> a
{}
```

4.3.4 Copia de diccionarios

Existen dos formas de copiar diccionarios:

- **Copia por referencia** `d1 = d2`: Asocia a 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'}
```

5 Funciones

5.1 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!
```

5.1.1 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!
```

5.1.2 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.
- **Argumentosnominales**: 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!
```

5.1.3 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
```

5.2 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!
```

5.3 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.
- `**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 menu(*platos):
...     print('Hoy tenemos: ', end='')
...     for plato in platos:
...         print(plato, end=', ')
...     return
...
>>> menu('pasta', 'pizza', 'ensalada')
Hoy tenemos: pasta, pizza, ensalada,
```

5.4 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
```

5.5 Ámbito de los parámetros y variables de una función

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
```

5.6 Paso de argumentos por referencia

En Python el paso de argumentos a una función es siempre por referencia, es decir, se pasa una referencia al objeto del argumento, de manera que cualquier cambio que se haga dentro de la función mediante el parámetro asociado afectará al objeto original, siempre y cuando este sea mutable.

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

5.7 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    un triángulo.
    de

    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.
```

5.8 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
```

5.8.1 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
```

5.8.2 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
```

5.9 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
```

5.9.1 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
```

5.9.2 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)
```

5.9.3 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]
```

5.9.4 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}
```

5.9.5 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
```

5.10 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.

5.10.1 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']
```

5.10.2 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}
```

6 Archivos

6.1 Archivos

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 archivos.

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

Las operaciones más habituales con archivos son:

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

6.1.1 Creación y escritura de archivos

Para crear un archivo nuevo se utiliza la instrucción

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

Si el archivo indicado por la ruta ya existe en el sistema, se reemplazará por el nuevo. Una vez creado el archivo, para escribir datos en él se utiliza el método `archivo.write(c)`: Escribe la cadena `c` en el archivo referenciado por `archivo`.

```
1 >>> f = open('bienvenida.txt', 'w')
2 ... f.write('¡Bienvenido a Python!')
```

6.1.2 Añadir datos a un archivo

Si en lugar de crear un archivo nuevo queremos añadir datos a un archivo existente se debe utilizar la instrucción

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

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

```
1 >>> f = open('bienvenida.txt', 'a')
2 ... f.write('\n¡Hasta pronto!')
```

6.1.3 Abrir un archivo

Para abrir un archivo en modo lectura se utiliza la instrucción

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

Una vez abierto el archivo, se puede leer todo el contenido del archivo o se puede leer línea a línea.

6.1.4 Leer datos de un archivo

`archivo.read()`: Devuelve todos los datos contenidos en `archivo` como una cadena de caracteres.

`archivo.readlines()`: Devuelve una lista de cadenas de caracteres donde cada cadena es una linea del archivo referenciado por `archivo`.

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

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

6.1.5 Cerrar un archivo

Para cerrar un archivo se utiliza el método `archivo.close()`: Cierra el archivo referenciado por el objeto `archivo`.

Cuando se termina de trabajar con un archivo 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 archivo, Python intentará cerrarlo cuando estime que ya no se va a usar más.

```
>>> f = open('bienvenida.txt'):
... print(f.read())
... f.close() # Cierre del archivo
...
¡Bienvenido a Python!
¡Hasta pronto!
```

6.1.6 Renombrado y borrado de un archivo

Para renombrar o borrar un archivo se utilizan funciones del módulo `os`.

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

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

Antes de borrar o renombrar 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 archivo en la ruta `ruta` y `False` en caso contrario.

6.1.7 Renombrado y borrado de un archivo o directorio

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

6.1.8 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 archivos 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.

7 Excepciones

7.1 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
```

7.1.1 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 archivo 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.

7.1.2 Control de excepciones

`try-except-else` Paraevitarlainterrucióndelaejecucióndelprogramacuandoseproduce 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.

7.1.3 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('archivo.txt') # El archivo no existe
... except FileNotFoundError:
...     print('¡El archivo no existe!')
... else:
...     print(f.read())
¡El archivo no existe!
```

8 Programación Orientada a Objetos

8.1 Objetos

Python también permite *la programación orientada a objetos*, que es un paradigma de programación en la que los datos y las operaciones que pueden realizarse con esos datos se agrupan en unidades lógicas llamadas **objetos**.

Los objetos suelen representar conceptos del dominio del programa, como un estudiante, un coche, un teléfono, etc. Los datos que describen las características del objeto se llaman **atributos** y son la parte estática del objeto, mientras que las operaciones que puede realizar el objeto se llaman **métodos** y son la parte dinámica del objeto.

La programación orientada a objetos permite simplificar la estructura y la lógica de los grandes programas en los que intervienen muchos objetos que interactúan entre si. **Ejemplo**. Una tarjeta de crédito puede representarse como un objeto:

- Atributos: Número de la tarjeta, titular, balance, fecha de caducidad, pin, entidad emisora, estado (activa o no), etc.
- Métodos: Activar, pagar, renovar, anular.

8.1.1 Acceso a los atributos y métodos de un objeto

- `dir(objeto)`: Devuelve una lista con los nombres de los atributos y métodos del objeto `objeto`.

Para ver si un objeto tiene un determinado atributo o método se utiliza la siguiente función:

- `hasattr(objeto, elemento)`: Devuelve `True` si `elemento` es un atributo o un método del objeto `objeto` y `False` en caso contrario.

Paraaccederalosatributosymétodosdeunobjetoseponeelnombredelobjetoseguidodel*operador punto* y el nombre del atributo o el método. • `objeto.atributo`: Accede al atributo `atributo` del objeto `objeto`.

- `objeto.método(parámetros)`: Ejecuta el método `método` del objeto `objeto` con los parámetros que se le pasen

En Python los tipos de datos primitivos son también objetos que tienen asociados atributos y métodos.

Ejemplo. Las cadenas tienen un método `upper` que convierte la cadena en mayúsculas. Para aplicar este método a la cadena `c` se utiliza la instrucción `c.upper()`.

```
>>> c = 'Python'

>>> print(c.upper())     # Llamada al método upper del objeto c (cadena)
PYTHON
```

Ejemplo. Las listas tienen un método `append` que convierte añade un elemento al final de la lista.

Para aplicar este método a la lista `l` se utiliza la instrucción `l.append(<elemento>)`.

```
>>> l = [1, 2, 3]

>>> l.append(4)          # Llamada al método append del objeto l
                         (lista)
>>> print(l)

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

8.2 Clases (`class`)

Los objetos con los mismos atributos y métodos se agrupan **clases**. Las clases definen los atributos y los métodos, y por tanto, la semántica o comportamiento que tienen los objetos que pertenecen a esa clase. Se puede pensar en una clase como en un *molde* a partir del cuál se pueden crear objetos.

Para declarar una clase se utiliza la palabra clave `class` seguida del nombre de la clase y dos puntos, de acuerdo a la siguiente sintaxis:

```
class <nombre-clase>:

  <atributos>

  <métodos>
```

Los atributos se definen igual que las variables mientras que los métodos se definen igual que las funciones. Tanto unos como otros tienen que estar indentados por 4 espacios en el cuerpo de la clase.

Ejemplo El siguiente código define la clase `Saludo` sin atributos ni métodos. La palabra reservada `pass` indica que la clase está vacía.

```
>>> class Saludo:
...     pass        # Clase vacía sin atributos ni métodos.
>>> print(Saludo)
<class '__main__.Saludo'>
```

Es una buena práctica comenzar el nombre de una clase con mayúsculas.

8.2.1 Clases primitivas

En Python existen clases predefinidas para los tipos de datos primitivos:

- `int`: Clase de los números enteros.
- `float`: Clase de los números reales.
- `str`: Clase de las cadenas de caracteres.
- `list`: Clase de las listas.
- `tuple`: Clase de las tuplas.
- `dict`: Clase de los diccionarios.

```
>>> type(1)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type('Python')
<class 'str'>
>>> type([1,2,3])
<class 'list'>
>>> type((1,2,3))
<class 'tuple'>
>>> type({1:'A', 2:'B'})
<class 'dict'>
```

8.2.2 Instanciación de clases

Para crear un objeto de una determinada clase se utiliza el nombre de la clase seguida de los parámetros necesarios para crear el objeto entre paréntesis.

- `clase(parámetros)`: Crea un objeto de la clase `clase` inicializado con los `parámetros` dados.

Cuando se crea un objeto de una clase se dice que el objeto es una *instancia* de la clase.

```
>>> class Saludo:
...     pass        # Clase vacía sin atributos ni métodos.
>>> s = Saludo()  # Creación del objeto mediante instanciación de la clase.
>>> s
<__main__.Saludo object at fcfc7756be0>       #Dirección de memoria 0x7 donde se crea el objeto
>>> type(s)
<class '__main__.Saludo'>     # Clase del objeto
```

8.2.3 Definición de métodos

Los métodos de una clase son las funciones que definen el comportamiento de los objetos de esa clase.

Se definen como las funciones con la palabra reservada `def`. La única diferencia es que su primer parámetro es especial y se denomina `self`. Este parámetro hace siempre referencia al objeto desde donde se llama el método, de manera que para acceder a los atributos o métodos de una clase en su propia definición se puede utilizar la sintaxis `self.atributo` o `self.método`.

```
>>> class Saludo:
...     mensaje = "Bienvenido "            # Definición de un atributo
...     def saludar(self, nombre):         # Definición de un método
...         print(self.mensaje + nombre)
...         return
...
>>> s = Saludo()
>>> s.saludar('Alf')
Bienvenido Alf
```

La razón por la que existe el parámetro `self` es porque Python traduce la llamada a un método de un objeto `objeto.método(parámetros)` en la llamada `clase.método(objeto, parámetros)`, es decir, se llama al método definido en la clase del objeto, pasando como primer argumento el propio objeto, que se asocia al parámetro `self`.

8.2.4 El método `__init__`

En la definición de una clase suele haber un método llamado `__init__` que se conoce como *inicializador*. Este método es un método especial que se llama cada vez que se instancia una clase y sirve para inicializar el objeto que se crea. Este método crea los atributos que deben tener todos los objetos de la clase y por tanto contiene los parámetros necesarios para su creación, pero no devuelve nada.
Se invoca cada vez que se instancia un objeto de esa clase.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):    # Inicializador
            self.id = id      # atributo id         Creación del
... self.saldo = cantidad # atributo saldo         Creación del
...         return
...     def mostrar_saldo(self):
...          print('El saldo es', self.saldo, '€')
...         return
```

```
>>> t = Tarjeta('1111111111', 1000) # Creación de un objeto con argumentos
>>> t.muestra_saldo()
El saldo es 1000 €
```

8.2.5 Atributos de instancia vs atributos de clase

Los atributos que se crean dentro del método `__init__` se conocen como atributos del objeto, mientras que los que se crean fuera de él se conocen como atributos de la clase. Mientras que los primeros son propios de cada objeto y por tanto pueden tomar valores distintos, los valores de los atributos de la clase son los mismos para cualquier objeto de la clase.

En general, no deben usarse atributos de clase, excepto para almacenar valores constantes.

```
>>> class Circulo:
...     pi = 3.14159                    # Atributo de clase
...     def __init__(self, radio):
...         self.radio = radio          # Atributo de instancia
...     def area(self):
...         return Circulo.pi * self.radio ** 2
...
>>> c1 = Circulo(2)
>>> c2 = Circulo(3)
>>> print(c1.area())
12.56636
>>> print(c2.area())
28.27431
>>> print(c1.pi)
3.14159
>>> print(c2.pi)
3.14159
```

8.2.6 El método `__str__`

Otro método especial es el método llamado `__str__` que se invoca cada vez que se llama a las funciones `print` o `str`. Devuelve siempre una cadena que se suele utilizar para dar una descripción informal del objeto. Si no se define en la clase, cada vez que se llama a estas funciones con un objeto de la clase, se muestra por defecto la posición de memoria del objeto.

```
>>> class Tarjeta:
...     def __init__(self, numero, cantidad = 0):
...         self.numero = numero
...         self.saldo = cantidad
...         return
...     def __str__(self):
...return 'Tarjeta número {} con saldo {:.2f€}'.format(self. numero, str(self.saldo))
>>> t = tarjeta('0123456789', 1000)
>>> print(t)
```

```
Tarjeta número 0123456789 con saldo €1000.00
```

8.3 Herencia

Una de las características más potentes de la programación orientada a objetos es la **herencia**, que permite definir una especialización de una clase añadiendo nuevos atributos o métodos. La nueva clase se conoce como *clase hija* y hereda los atributos y métodos de la clase original que se conoce como *clase madre*.

Para crear un clase a partir de otra existente se utiliza la misma sintaxis que para definir una clase, pero poniendo detrás del nombre de la clase entre paréntesis los nombres de las clases madre de las que hereda.

Ejemplo. A partir de la clase `Tarjeta` definida antes podemos crear mediante herencia otra clase `Tarjeta_Descuento` para representar las tarjetas de crédito que aplican un descuento sobre las compras.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
... def mostrar_saldo(self): # Método de la clase Tarjeta que hereda
    la clase Tarjeta_descuento
...         print('El saldo es', self.saldo, '€.')
...         return
...
>>> class Tarjeta_descuento(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def mostrar_descuento(self):   # Método exclusivo de la clase
    Tarjeta_descuento
...         print('Descuento de', self.descuento, '% en los pagos.')
...         return
...
>>> t = Tarjeta_descuento('0123456789', 2, 1000)
>>> t.mostrar_saldo()
El saldo es 1000 €.
>>> t.mostrar_descuento()
Descuento de 2 % en los pagos.
```

La principal ventaja de la herencia es que evita la repetición de código y por tanto los programas son más fáciles de mantener.

En el ejemplo de la tarjeta de crédito, el método `mostrar_saldo` solo se define en la clase madre. De esta manera, cualquier cambio que se haga en el cuerpo del método en la clase madre, automáticamente

se propaga a las clases hijas. Sin la herencia, este método tendría que replicarse en cada una de las clases hijas y cada vez que se hiciese un cambio en él, habría que replicarlo también en las clases hijas.

8.3.1 Jerarquía de clases

A partir de una clase derivada mediante herencia se pueden crear nuevas clases hijas aplicando de nuevo la herencia. Ello da lugar a una jerarquía de clases que puede representarse como un árbol donde cada clase hija se representa como una rama que sale de la clase madre.

Debido a la herencia, cualquier objeto creado a partir de una clase es una instancia de la clase, pero también lo es de las clases que son ancestros de esa clase en la jerarquía de clases. El siguiente comando permite averiguar si un objeto es instancia de una clase:

- `isinstance(objeto, clase)`: Devuelve `True` si el objeto `objeto` es una instancia de la clase `clase` y `False` en caso contrario.

```
1  # Asumiendo la definición de las clases Tarjeta y Tarjeta_descuento
       anteriores.
2  >>> t1 = Tarjeta('1111111111', 0)
3  >>> t2 = t = Tarjeta_descuento('2222222222', 2, 1000)
4  >>>                 Tarjeta)
   isinstance(t1,
5  True

6  >>>                 Tarjeta_descuento)
   isinstance(t1,
7  False

8  >>>                 Tarjeta_descuento)
   isinstance(t2,
9  True

10 >>>                 Tarjeta)
   isinstance(t2,
11 True
```

8.3.2 Sobrecarga y polimorfismo

Los objetos de una clase hija heredan los atributos y métodos de la clase madre y, por tanto, a priori tienen tienen el mismo comportamiento que los objetos de la clase madre. Pero la clase hija puede definir nuevos atributos o métodos o reescribir los métodos de la clase madre de manera que sus objetos presenten un comportamiento distinto. Esto último se conoce como **sobrecarga**.

De este modo, aunque un objeto de la clase hija y otro de la clase madre pueden tener un mismo método, al invocar ese método sobre el objeto de la clase hija, el comportamiento puede ser distinto a cuando se

invoca ese mismo método sobre el objeto de la clase madre. Esto se conoce como **polimorfismo** y es otra de las características de la programación orientada a objetos.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
...     def mostrar_saldo(self):
...         print('El saldo es {:.2f€}.'.format(self.saldo))
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad
...         return
>>> class Tarjeta_Oro(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad * (1 - self.descuento / 1...00)
>>> t1 = Tarjeta('1111111111', 1000)
>>> t2 = Tarjeta_Oro('2222222222', 1, 1000)
>>> t1.pagar(100)
>>> t1.mostrar_saldo()
El saldo es €900.00.
>>> t2.pagar(100)
>>> t2.mostrar_saldo()
El saldo es €901.00.
```

8.4 Principios de la programación orientada a objetos

La programación orientada a objetos se basa en los siguientes principios:

- **Encapsulación**: Agrupar datos (atributos) y procedimientos (métodos) en unidades lógicas (objetos) y evitar maninupar los atributos accediendo directamente a ellos, usando, en su lugar, métodos para acceder a ellos.
- **Abstracción**: Ocultar al usuario de la clase los detalles de implementación de los métodos. Es decir, el usuario necesita saber *qué* hace un método y con qué parámetros tiene que invocarlo (*interfaz*), pero no necesita saber *cómo* lo hace.
- **Herencia**: Evitar la duplicación de código en clases con comportamientos similares, definiendo los métodos comunes en una clase madre y los métodos particulares en clases hijas.
- **Polimorfismo**: Redefinir los métodos de la clase madre en las clases hijas cuando se requiera un comportamiento distinto. Así, un mismo método puede realizar operaciones distintas dependiendo del objeto sobre el que se aplique.

Resolver un problema siguiendo el paradigma de la programación orientada a objetos requiere un cambio de mentalidad con respecto a como se resuelve utilizando el paradigma de la programación procedimental.

La programación orientada a objetos es más un proceso de modelado, donde se identifican las entidades que intervienen en el problema y su comportamiento, y se definen clases que modelizan esas entidades. Por ejemplo, las entidades que intervienen en el pago con una tarjeta de crédito serían la tarjeta, el terminal de venta, la cuenta corriente vinculada a la tarjeta, el banco, etc. Cada una de ellas daría lugar a una clase.

Después se crean objetos con los datos concretos del problema y se hace que los objetos interactúen entre sí, a través de sus métodos, para resolver el problema. Cada objeto es responsable de una subtarea y colaboran entre ellos para resolver la tarea principal. Por ejemplo, la terminal de venta accede a los datos de la tarjeta y da la orden al banco para que haga un cargo en la cuenta vinculada a la tarjeta.

De esta forma se pueden abordar problemas muy complejos descomponiéndolos en pequeñas tareas que son más fáciles de resolver que el problema principal (*¡divide y vencerás!*).

9 Módulos

9.1 Módulos

El código de un programa en Python puede reutilizarse en otro importándolo. Cualquier archivo 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.

9.1.1 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.

9.1.2 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
```

9.1.3 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 archivos.
- 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.

9.1.4 Otras librerías imprescindibles

Estas librerías no vienen en la distribución estándar de Python y necesitan instalarse. También puede optarse por la distribución Anaconda que incorpora la mayoría de estas librerías.

- 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.

10 Apéndice: Depuración de código

10.1 Depuración de programas

La depuración es una técnica que permite *trazar* un programa, es decir, seguir el flujo de ejecución de un programa paso a paso, ejecutando una instrucción en cada paso, y observar el estado de sus variables.

Cuandounprogramatieneciertacomplejidad,ladepuraciónesimprescindibleparedetectarposibles errores.

Python dispone del módulo `pyd` para depurar programas, pero es mucho más cómodo utilizar algún entorno de desarrollo que incorpore la depuración, como por ejemplo Visual Studio Code.

10.1.1 Comandos de depuración

- **Establecer punto de parada**: Detiene la ejecución del programa en una línea concreta de código.
- **Continuar la ejecución**: Continúa la ejecución del programa hasta el siguiente punto de parada o hasta que finalice.
- **Próximo paso**: Ejecuta la siguiente línea de código y para la ejecución.
- **Próximo paso con entrada en función**: Ejecuta la siguiente línea de código. Si se trata de una llamada a una función entonces ejecuta la primera instrucción de la función y para la ejecución.
- **Próximo paso con salida de función**: Ejecuta lo que queda de la función actual y para la ejecución.
- **Terminar la depuración**: Termina la depuración.

10.1.2 Depuración en Visual Studio Code

Antes de iniciar la depuración de un programa en VSCode hay que establecer algún punto de parada. Para ello basta con hacer click en le margen izquierdo de la ventana con del código a la altura de la línea donde se quiere parar la ejecución del programa.

Punto de parada en Visual Studio Code>

Para iniciar la depuración de un programa en VSCode hay que hacer clic sobre el botón Visual Studio Code debbuger o pulsar la combinación de teclas (Ctr+Shift+D).

La primera vez que depuremos un programa tendremos que crear un archivo de configuración del depurador (`launch.json`). Para ello hay que hacer clic en el botón `Run and Debug`. VSCode mostrará los distintos archivos de configuración disponibles y debe seleccionarse el más adecuado para el tipo de programa a depurar. Para programas simples se debe seleccionar `Python file`.

La depuración comenzará iniciando la ejecución del programa desde el inicio hasta el primer punto de parada que encuentre.

Una vez iniciado el proceso de depuración, se puede avanzar en la ejecución del programa haciendo uso de la barra de depuración que contiene botones con los principales comandos de depuración.

Barra de depuración de Visual Studio Code

Durante la ejecución del programa, se puede ver el contenido de las variables del programa en la ventana del estado de las variables.

El usuario también puede introducir expresiones y ver cómo se evalúan durante la ejecución del programa en la ventana de vista de expresiones.

Ventana de estado de variables de Visual Studio Code

11 Bibliografía

11.1 Referencias

11.1.1 Webs

- Python Sitio web de Python.
- Repl.it Entorno de desarrollo web para varios lenguajes, incluido Python.
- Python tutor Sitio web que permite visualizar la ejecución el código Python.

11.1.2 Libros y manuales

- Tutorial de Python Tutorial rápido de python.
- Python para todos Libro de introducción a Python con muchos ejemplos. Es de licencia libre.
- Python para principiantes Libro de introducción Python que abarca orientación a objetos. Es de licencia libre.
- Python crash course Libro de introducción a Python gratuito.
- Think python 2e. Libro de introducción a Python que abarca también algoritmos, estructuras de datos y gráficos. Es de licencia libre.
- Learning Python Libro de introducción a Python con enfoque de programación orientada a objetos.

11.1.3 Vídeos

- Curso “Python para todos”.