Programación imperativa

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Modelo de ejecución **2**
1.1. Máquina de estados . . . 2
1.2. Secuencia de instrucciones . . . 2

2. Asignación destructiva **3**
2.1. Variables . . . 3
2.2. Estado . . . 3
2.3. Sentencia de asignación . . . 3
2.4. Evaluación de expresiones con variables . . . 4
2.5. Constantes . . . 4

3. Mutabilidad **5**
3.1. Tipos mutables e inmutables . . . 5
3.1.1. Inmutables . . . 5
3.1.2. Mutables . . . 7
3.2. Alias de variables . . . 10
3.2.1. `id` . . . 11
3.2.2. `is` . . . 12

4. Cambios de estado ocultos **12**
4.1. Funciones puras . . . 12
4.2. Funciones impuras . . . 12
4.3. Efectos laterales . . . 13
4.4. Transparencia referencial . . . 13
4.5. Entrada y salida por consola . . . 14
4.5.1. `print` . . . 14
4.5.2. `input` . . . 15

5. Saltos **16**
5.1. Incondicionales . . . 16
5.2. Condicionales . . . 17

Bibliografía **17**

1. Modelo de ejecución

1.1. Máquina de estados

- La **programación imperativa** es un paradigma de programación basado en el concepto de **sentencia**.
- Un programa imperativo está formado por una sucesión de sentencias que se ejecutan **en orden**.
- Una sentencia es una instrucción del programa que lleva a cabo una de estas acciones:
  - **Cambiar el estado interno** del programa, normalmente mediante la llamada **sentencia de asignación**.
  - Cambiar el **flujo de control** del programa, haciendo que la ejecución se bifurque (*salte*) a otra parte del mismo.
- El modelo de ejecución de un programa imperativo es el de una **máquina de estados**, es decir, una máquina que va pasando por diferentes estados a medida que el programa va ejecutándose.
- En programación imperativa, el concepto de **tiempo** cobra mucha importancia: el programa actuará de una forma un otra según el estado en el que se encuentre, es decir, según el momento en el que estemos observando al programa.

  Eso significa que, ante los mismos datos de entrada, una función puede devolver **valores distintos en momentos distintos**.
- En programación funcional, en cambio, el comportamiento de una función no depende del momento en el que se ejecute, ya que siempre devolverá los mismos resultados ante los mismos datos de entrada.
- Eso significa que, para modelar el comportamiento de un programa imperativo, ya **no nos vale el modelo de sustitución**.

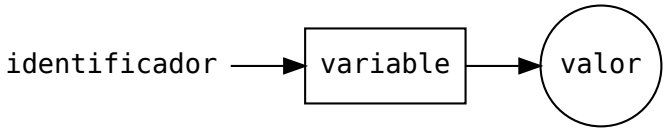
1.2. Secuencia de instrucciones

- Un programa imperativo es una **secuencia de instrucciones**, y ejecutar un programa es provocar los **cambios de estado** que dictan las instrucciones en el **orden** definido por el programa.
- Las instrucciones del programa van provocando **transiciones** entre estados, haciendo que la máquina pase de un estado al siguiente.
- Para modelar el comportamiento de un programa imperativo tendremos que saber en qué estado se encuentra el programa, para lo cual tendremos que seguirle la pista desde su estado inicial al estado actual.
- Eso básicamente se logra «ejecutando» mentalmente el programa instrucción por instrucción y llevando la cuenta de los valores ligados a sus identificadores.

2. Asignación destructiva

2.1. Variables

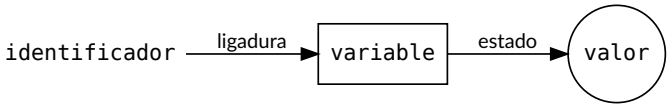
- Una **variable** es un lugar en la **memoria** donde se puede **almacenar un valor**.
- El valor de una variable **puede cambiar** durante la ejecución del programa.
- A partir de ahora, un identificador no se liga directamente con un valor, sino que tendremos:
  - Una **ligadura** entre un identificador y una **variable**.
  - **La variable almacena el valor**.



2.2. Estado

- La **ligadura** es la asociación que se establece entre un identificador y una variable.
- El **estado de una variable** es el valor que tiene una variable en un momento dado.

  Por tanto, el estado es la asociación que se establece entre una variable y un valor.



- Tanto las ligaduras como los estados pueden cambiar durante la ejecución de un programa imperativo.
- El **estado de un programa** es el conjunto de los estados de todas sus variables.

2.3. Sentencia de asignación

- La forma más básica de cambiar el estado de una variable es usando la **sentencia de asignación**.
- Es la misma instrucción que hemos estado usando hasta ahora para ligar valores a identificadores, pero ahora, en el paradigma imperativo, tiene otro significado:

```
x = 4
```

  Significa que el identificador `x` está ligado a una variable cuyo valor pasa a ser `4`.
- La asignación es **destructiva** porque al cambiar un valor a una variable se destruye su valor anterior. Por ejemplo, si ahora hacemos:

```
x = 9
```

El valor de la variable a la que está ligada el identificador x pasa ahora a ser 9, perdiéndose el valor 4 anterior.

- Por abuso del lenguaje, se suele decir:

  «se asigna el valor 9 a la variable x»

  en lugar de:

  «se asigna el valor 9 a la variable ligada al identificador x»

- Aunque esto simplifica las cosas a la hora de hablar, hay que tener cuidado, porque llegará el momento en el que podamos tener varios identificadores distintos ligados a la misma variable.
- Cada nueva asignación provoca un cambio de estado en el programa.
- En el ejemplo anterior, el programa pasa de estar en un estado en el que la variable x vale 4 a otro en el que la variable vale 9.
- Al final, un programa imperativo se puede reducir a una **secuencia de asignaciones** realizadas en el orden dictado por el programa.
- Este modelo de funcionamiento está estrechamente ligado a la arquitectura de un ordenador: hay una memoria formada por celdas que contienen datos que pueden cambiar a lo largo del tiempo según dicten las instrucciones del programa que controla al ordenador.

2.4. Evaluación de expresiones con variables

- Al evaluar expresiones, las variables actúan de modo similar a las ligaduras de la programación funcional, con la única diferencia de que su valor puede cambiar a lo largo del tiempo, por lo que deberemos *seguirle la pista* a las asignaciones que sufra dicha variable.
- Todo lo visto hasta ahora sobre marcos, ámbitos, sombreado de variables, entornos, etc. se aplica igualmente a las variables.

2.5. Constantes

- En programación funcional no existen las variables y un identificador sólo puede ligarse a un valor (un identificador ligado no puede re-ligarse a otro valor distinto).
  - En la práctica, eso significa que un identificador ligado actúa como un valor constante que no puede cambiar durante la ejecución del programa.
  - El valor de esa constante es el valor al que está ligado el identificador.
- En programación imperativa, los identificadores se ligan a variables, que son las que realmente contienen los valores.
- Una **constante** en programación imperativa sería el equivalente a una variable cuyo valor no puede cambiar durante la ejecución del programa.
- Muchos lenguajes de programación permiten definir constantes, pero **Python no es uno de ellos**.

- En Python, una constante **es una variable más**, pero **es responsabilidad del programador** no cambiar su valor durante todo el programa.
- Python no hace ninguna comprobación ni muestra mensajes de error si se cambia el valor de una constante.
- En Python, por **convenio**, los identificadores ligados a un valor constante se escriben con todas las letras en **mayúscula**:

```
PI = 3.1415926
```

  El nombre en mayúsculas nos recuerda que `PI` es una constante.

- Aunque nada nos impide cambiar su valor (cosa que debemos evitar):

```
PI = 99
```

3. Mutabilidad

3.1. Tipos mutables e inmutables

- En Python existen tipos cuyos valores son *inmutables* y otros que son *mutables*.
- Un valor **inmutable** es aquel cuyo estado interno no puede cambiar durante la ejecución del programa.

  Los tipos inmutables en Python son los números (`int` y `float`), los booleanos (`bool`), las cadenas (`str`), las tuplas (`tuple`), los rangos (`range`) y los conjuntos congelados (`frozenset`).

- Un valor **mutable** es aquel cuyo estado interno (su **contenido**) puede cambiar durante la ejecución del programa.

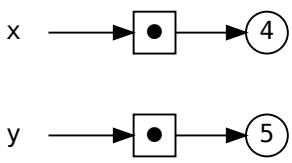
  El principal tipo mutable en Python es la lista (`list`), pero también están los conjuntos (`set`) y los diccionarios (`dict`).

3.1.1. Inmutables

- Un valor de un tipo inmutable no puede cambiar su contenido.

Por ejemplo, si tenemos:

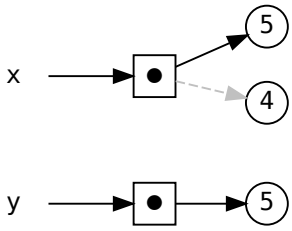
```
x = 4
y = 5
```



y hacemos:

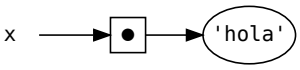
```
x = 5
```

quedaría:



- Lo que hace la asignación `x = 5` no es cambiar el contenido del valor `5`, sino hacer que la variable `x` contenga otro valor distinto (el valor `4` no se modifica en ningún momento).
- Con las cadenas sería exactamente igual.
- Si tenemos:

  ```
  x = 'hola'
  ```

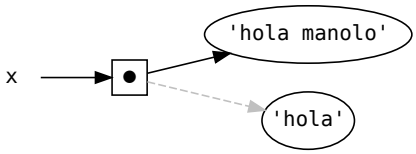
  

  y luego hacemos:

  ```
  x = 'hola manolo'
  ```

  se crea una nueva cadena y se la asignamos a la variable `x`.

  Es decir: la cadena `'hola'` original **no se cambia**, sino que desaparece y queda **sustituida por una nueva**.

- Aunque las cadenas son datos inmutables, también son datos compuestos y podemos acceder individualmente a sus elementos componentes y operar con ellos aunque no podamos cambiarlos.
- Para ello podemos usar las operaciones comunes a toda secuencia de elementos (una cadena también es una **secuencia de caracteres**):

Operación	Resultado
x `in` *s*	`True` si *x* está en *s*
x `not in` *s*	Lo contrario
s`[`*i*`]`	(*Indexación*) El *i*-ésimo elemento de *s*, empezando por 0
s`[`*i*`:`*j*`]`	(*Slicing*) Rodaja de *s* desde *i* hasta *j*

Operación	Resultado
`s[i:j:k]`	Rodaja de *s* desde *i* hasta *j* con paso *k*
`s.index(x)`	Índice de la primera aparición de *x* en *s*
`s.count(x)`	Número de veces que aparece *x* en *s*

- El *operador* de **indexación** consiste en acceder al elemento situado en la posición indicada entre corchetes:

```
  +-----+-----+-----+-----+-----+-----+
s | 124 | 333 | 'a' | 3.2 |  9  |  53 |
  +-----+-----+-----+-----+-----+-----+
     0     1     2     3     4     5
    -6    -5    -4    -3    -2    -1
```

```
>>> s[2]
'a'
>>> s[-2]
9
```

- El **slicing** (*hacer rodajas*) es una operación que consiste en obtener una subsecuencia a partir de una secuencia, indicando los índices de los elementos inicial y final de la misma:

```
  +---+---+---+---+---+---+
s | P | y | t | h | o | n |
  +---+---+---+---+---+---+
  0   1   2   3   4   5   6
 -6  -5  -4  -3  -2  -1
```

```
>>> s[0:2]
'Py'
>>> s[-5:-4]
'y'
>>> s[-4:-5]
''
>>> s[0:4:2]
'Pt'
```

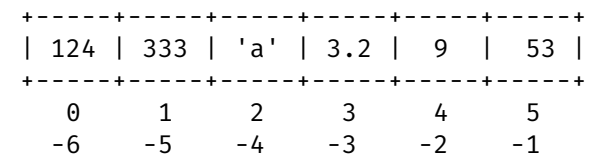
3.1.2. Mutables

- Los valores de tipos **mutables**, en cambio, pueden cambiar su estado interno durante la ejecución del programa.
- Hasta ahora, hemos visto un único tipo mutable: la **lista**.
- Una lista puede cambiar el valor de sus elementos, aumentar o disminuir de tamaño.
- Al cambiar el estado de una lista no se crea una nueva lista, sino que **se modifica la ya existente**:

```
>>> x = [24, 32, 15, 81]
>>> x[0] = 99
>>> x
```

```
[99, 32, 15, 81]
```

- Las listas, como toda secuencia mutable, se pueden modificar usando ciertas operaciones:
  - Los *operadores* de **indexación** y **slicing**:




```
>>> l = [124, 333, 'a', 3.2, 9, 53]
>>> l[3]
3.2
>>> l[3] = 99
>>> l
[124, 333, 'a', 99, 9, 53]
>>> l[0:2] = [40]
>>> l
[40, 'a', 99, 9, 53]
```

- *Métodos* propios de las listas, como `append`, `clear`, `insert`, `remove`, `reverse` o `sort`.

(s y t son listas, y x es un valor cualquiera)

Operación	Resultado
`s[i] = x`	El elemento *i*-ésimo de *s* se sustituye por *x*
`s[i:j] = t`	La rodaja de *s* desde *i* hasta *j* se sustituye por *t*
`s[i:j:k] = t`	Los elementos de `s[i:j:k]` se sustituyen por *t*
`del s[i:j]`	Elimina los elementos de `s[i:j]` ↔ `s[i:j] = []`
`del s[i:j:k]`	Elimina los elementos de `s[i:j:k]`
`s.append(x)`	Añade *x* al final de *s* ↔ `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s* ↔ `del s[:]`
`s.extend(t)` ó	Amplía *s* con el contenido de *t* ↔
`s += t`	`s[len(s):len(s)] = t`
`s.insert(i , x)`	Inserta *x* en *s* en el índice *i* ↔ `s[i:i] = [x]`
`s.pop([i = -1])`	Devuelve el elemento *i*-ésimo y lo elimina de *s*
`s.remove(x)`	Elimina el primer elemento de *s* que sea igual a *x*
`s.reverse()`	Invierte los elementos de *s*

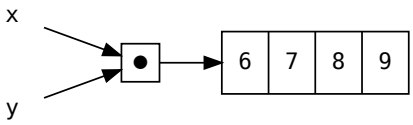
Partiendo de `x = [8, 10, 7, 9]`:

Ejemplo	Valor de `x` después
`x.append(14)`	`[8, 10, 7, 9, 14]`
`x.clear()`	`[]`
`x.insert(3, 66)`	`[8, 10, 7, 66, 9]`
`x.remove(7)`	`[8, 10, 9]`
`x.reverse()`	`[9, 7, 10, 8]`

3.2. Alias de variables

- Cuando una variable que contiene un valor se asigna a otra, se produce un fenómeno conocido como **alias de variables**, según el cual los dos identificadores se ligan a **la misma variable** (la comparten):

```
x = [6, 7, 8, 9]
y = x  # x se asigna a y
```

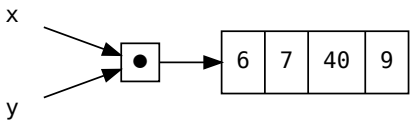


- Ahora, si el valor es **mutable** y cambiamos su **contenido** desde x, también cambiará y (y viceversa), pues ambas son **la misma variable**:

```
>>> y[2] = 40
>>> x
[6, 7, 40, 9]
```

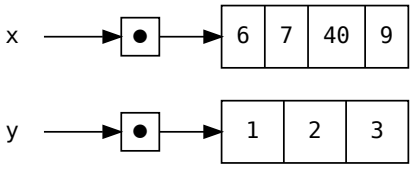
- No es lo mismo cambiar el **valor** que cambiar el **contenido** del valor.
- Cambiar el **contenido** es algo que sólo se puede hacer si el valor es **mutable** (por ejemplo, cambiando un elemento de una lista):

```
>>> x = [6, 7, 8, 9]
>>> y = x
>>> y[2] = 40
```



- Cambiar el **valor** es algo que **siempre** se puede hacer simplemente **asignando** a la variable **un nuevo valor**:

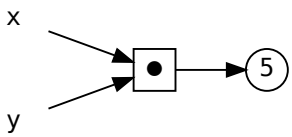
```
>>> y = [1, 2, 3]
```



- El intérprete puede crear alias de variables **implícitamente** para ahorrar memoria y sin que seamos conscientes de ello.
- No tiene mucha importancia práctica, aunque es interesante saberlo en ciertos casos.

- Por ejemplo, el intérprete de Python crea internamente una variable para cada número entero comprendido entre -5 y 256, por lo que todas las variables de nuestro programa que contengan el mismo valor dentro de ese intervalo compartirán el mismo espacio en memoria (serán alias):

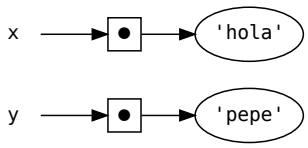
```
x = 5
y = 5
```



- También crea variables compartidas cuando contienen exactamente las mismas cadenas.

Si tenemos:

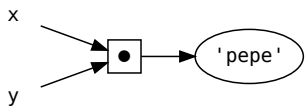
```
x = 'hola'
y = 'pepe'
```



y hacemos:

```
x = 'pepe'
```

quedaría:



- El intérprete aprovecharía la variable ya creada y no crearía una nueva, para ahorrar memoria.

3.2.1. `id`

- Para saber si dos variables son realmente **la misma variable**, se puede usar la función `id`.
- La **función `id`** devuelve un identificador único para cada dato en memoria.
- Por tanto, si dos variables tienen el mismo `id`, significa que son realmente la misma variable (están situadas en la misma celda de la memoria).

```
>>> x = 'hola'
>>> y = 'hola'
>>> id(x) == id(y)
```

```
True
```

```
>>> x = [1, 2, 3, 4]
>>> y = [1, 2, 3, 4]
>>> id(x) == id(y)
False
>>> y = x
>>> id(x) == id(y)
True
```

3.2.2. `is`

- Otra forma de comprobar si dos datos son realmente el mismo dato en memoria (es decir, si son **idénticos**) es usar el operador `is`, que comprueba la **identidad** de un dato:
- Su sintaxis es:

  <is> ::= *<valor1>* **`is`** *<valor2>*

- Es un operador relacional que devuelve `True` si *<valor1>* y *<valor2>* son **el mismo dato en memoria** (es decir, si se encuentran almacenados en la misma celda de la memoria y, por tanto, son **idénticos**) y `False` en caso contrario.
- Cuando se usa con variables, devuelve `True` si son la misma variable, y `False` en caso contrario.
- En la práctica, equivale a hacer `id(`*<valor1>*`) == id(`*<valor2>*`)`

4. Cambios de estado ocultos

4.1. Funciones puras

- Las **funciones puras** son aquellas que cumplen que:
  - su valor de retorno depende únicamente del valor de sus argumentos, y
  - calculan su valor de retorno sin provocar cambios de estado observables en el exterior de la función.
- Una llamada a una función pura se puede sustituir libremente por su valor de retorno sin afectar al resto del programa (es lo que se conoce como **transparencia referencial**).
- Las funciones *puras* son las únicas que existen en programación funcional.

4.2. Funciones impuras

- Por contraste, una función se considera **impura**:
  - si su valor de retorno o su comportamiento dependen de algo más que de sus argumentos, o

  - si provoca cambios de estado observables en el exterior de la función.

    En éste último caso decimos que la función provoca **efectos laterales**.

- Toda función que provoca efectos laterales es impura, pero no todas las funciones impuras provocan efectos laterales (puede ser impura porque su comportamiento se vea afectado por los efectos laterales provocados por otras partes del programa).

4.3. Efectos laterales

- Un **efecto lateral** es cualquier cambio de estado llevado a cabo en una parte del programa (generalmente, una función) que pueda afectar a otra parte del mismo de una manera poco evidente o impredecible.
- Una función puede provocar efectos laterales, o bien verse afectada por efectos laterales provocados por otras partes del programa.
- Los casos típicos de efectos laterales en una función son:
  - Cambiar el valor de una variable global.
  - Cambiar el estado de un argumento mutable.
  - Realizar una operación de entrada/salida.

  En cualquiera de estos casos, tendríamos una función **impura**.

4.4. Transparencia referencial

- En un lenguaje imperativo **se pierde la transparencia referencial**, ya que ahora el valor de una función puede depender no sólo de los valores de sus argumentos, sino también además de los valores de las variables libres que ahora pueden cambiar durante la ejecución del programa:

```
>>> suma = lambda x, y: x + y + z
>>> z = 2
>>> suma(3, 4)
9
>>> z = 20
>>> suma(3, 4)
27
```

- Por tanto, cambiar el valor de una variable global es considerado un **efecto lateral**, ya que puede alterar el comportamiento de otras partes del programa de formas a menudo impredecibles.
- Si el efecto lateral lo produce la propia función también estamos perdiendo transparencia referencial, pues en tal caso no podemos sustituir libremente la llamada a la función por su valor de retorno, ya que ahora la función **hace algo más** que calcular dicho valor y que es observable fuera de la función.
- Por ejemplo, una función que imprime algo por la pantalla o escribe algo en un archivo del disco tampoco es una función pura y, por tanto, en ella no se cumple la transparencia referencial.
- Lo mismo pasa con las funciones que modifican algún argumento mutable. Por ejemplo:

```
>>> ultimo = lambda x: x.pop()
>>> lista = [1, 2, 3, 4]
>>> ultimo(lista)
4
>>> ultimo(lista)
3
>>> lista
[1, 2]
```

- Los efectos laterales hacen que sea muy difícil razonar sobre el funcionamiento del programa, puesto que las funciones impuras no pueden verse como simples correspondencias entre los datos de entrada y el resultado de salida, sino que además hay que tener en cuenta los **efectos ocultos** que producen en otras partes del programa.
- Por ello, se debe **evitar**, siempre que sea posible, escribir funciones impuras (o sea, que provoquen efectos laterales).
- Ahora bien: muchas veces, la función que se desea escribir tiene efectos laterales porque esos son, precisamente, los efectos deseados.
  - Por ejemplo, una función que actualice los salarios de los empleados en una base de datos, a partir del salario base y los complementos.
- En ese caso, es importante **documentar** adecuadamente la función para que, quien desee usarla, sepa perfectamente qué efectos produce más allá de devolver un resultado.

4.5. Entrada y salida por consola

- Nuestro programa puede comunicarse con el exterior realizando **operaciones de entrada/salida**.
- Interpretamos la palabra *exterior* en un sentido amplio; por ejemplo:
  - El teclado
  - La pantalla
  - Un archivo del disco duro
  - Otro ordenador de la red
- La entrada/salida por consola se refiere a las operaciones de lectura de datos por el teclado y escritura por la pantalla.
- Las operaciones de entrada/salida se consideran **efectos laterales** porque producen cambios en el exterior o pueden provocar que el resultado de una función dependa de los datos leídos.

4.5.1. `print`

- La función `print` imprime (*escribe*) por la salida (normalmente la pantalla) el valor de una o varias expresiones.
- Su sintaxis es:

```
<print> ::= print(<expresión>(, <expresión>)*
                  [, sep=<expresión>][, end=<expresión>])
```

- El `sep` es el *separador* y su valor por defecto es `' '` (un espacio).
- El `end` es el *terminador* y su valor por defecto es `'\n'` (el carácter de nueva línea).
- Las expresiones se convierten en cadenas antes de imprimirse.
- Por ejemplo:

```
>>> print('hola', 'pepe', 23)
hola pepe 23
```

4.5.1.1. El valor `None`

- Es importante resaltar que la función `print` **no devuelve** el valor de las expresiones, sino que las **imprime** (provoca el efecto lateral de cambiar la pantalla haciendo que aparezcan nuevos caracteres).
- La función `print` como tal no devuelve ningún valor, pero como en Python todas las funciones devuelven *algún* valor, en realidad lo que ocurre es que **devuelve un valor `None`**.
- `None` es un valor especial que significa **ningún valor** y se utiliza principalmente para casos en los que no tiene sentido que una función devuelva un valor determinado, como es el caso de `print`.
- Pertenece a un tipo de datos especial llamado `NoneType` cuyo único valor posible es `None`, y para comprobar si un valor es `None` se usa `<valor> is None`.
- Podemos comprobar que, efectivamente, `print` devuelve `None`:

```
>>> print('hola', 'pepe', 23) is None
hola pepe 23  # ésto es lo que imprime print
True          # ésto es el resultado de comprobar si el valor de print es None
```

4.5.2. `input`

- La función `input` *lee* datos desde la entrada (normalmente el teclado) y devuelve el valor del dato introducido.
- Siempre devuelve una **cadena**.
- Su sintaxis es:

```
<input> ::= input([<prompt>])
```

- Por ejemplo:

```
>>> nombre = input('Introduce tu nombre: ')
Introduce tu nombre: Ramón
>>> print('Hola,', nombre)
Hola, Ramón
```

- Provoca el *efecto lateral* de alterar el estado de la consola imprimiendo el *prompt* y esperando a que desde el exterior se introduzca el dato solicitado (que en cada ejecución podrá tener un valor distinto).

5. Saltos

5.1. Incondicionales

- Un **salto incondicional** es una ruptura abrupta del flujo de control del programa hacia otro punto del mismo.
- Se llama *incondicional* porque no depende de ninguna condición, es decir, se lleva a cabo **siempre** que se alcanza el punto del salto.
- Históricamente, a esa instrucción que realiza saltos incondicionales se la ha llamado **instrucción *GOTO***.
- El uso de instrucciones *GOTO* es considerado, en general, una mala práctica de programación ya que favorece la creación del llamado **código *espagueti***: programas con una estructura de control tan complicada que resultan casi imposibles de mantener.
- En cambio, usados controladamente y de manera local, puede ayudar a escribir soluciones sencillas y claras.
- Python no incluye la instrucción *GOTO* pero se puede simular instalando un paquete llamado `goto-statement`:

```
$ pip3 install --user goto-statement
```

- Sintaxis:

```
<goto> ::= goto <etiqueta>
<label> ::= label <etiqueta>
<etiqueta> ::= .<identificador>
```

- Un ejemplo de uso:

```
from goto import with_goto

CODIGO="""
print('Esto se hace')
goto .fin
print('Esto se salta')
label .fin
print('Aquí se acaba')
"""

exec(with_goto(compile(CODIGO, '', 'exec')))
```

5.2. Condicionales

- Un **salto condicional** es un salto que se lleva a cabo sólo si se cumple una determinada condición.
- En Python, usando el paquete `with_goto`, podríamos implementarlo de la siguiente forma:

```
<salto_condicional> ::= if <condición>: goto <etiqueta>
```

- Ejemplo de uso:

```
from goto import with_goto

CODIGO="""
primero = 2
ultimo = 25

i = primero

label .inicio
if i == ultimo: goto .fin

print(i, end=' ')
i += 1
goto .inicio

label .fin
"""

exec(with_goto(compile(CODIGO, '', 'exec')))
```

Bibliografía

Aguilar, Luis Joyanes. 2008. *Fundamentos de Programación*. Aravaca: McGraw-Hill Interamericana de España.

Pareja Flores, Cristóbal, Manuel Ojeda Aciego, Ángel Andeyro Quesada, and Carlos Rossi Jiménez. 1997. *Desarrollo de Algoritmos Y Técnicas de Programación En Pascal*. Madrid: Ra-Ma.