Programación funcional

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/10/09 a las 15:09:00

Índice

1. Introducción

1.1. Concepto

La **programación funcional** es un paradigma de programación declarativa basado en el uso de **definiciones**, **expresiones** y **funciones matemáticas**.

Tiene su origen teórico en el **cálculo lambda**, un sistema matemático creado en 1930 por Alonzo Church.

Los lenguajes funcionales se pueden considerar *azúcar sintáctico* (es decir, una forma equivalente pero sintácticamente más sencilla) del cálculo lambda.

En programación funcional, una función define un cálculo a realizar a partir de unos datos de entrada, con la propiedad de que el resultado de la función sólo puede depender de esos datos de entrada.

Eso significa que una función no puede tener estado interno ni su resultado puede depender del estado del programa.

Además, una función no puede producir ningún efecto observable fuera de ella (los llamados **efectos laterales**), salvo calcular y devolver su resultado.

Esto quiere decir que en programación funcional no existen los efectos laterales, o se dan de forma muy localizada en partes muy concretas e imprescindibles del programa.

Por todo lo expuesto anteriormente, se dice que las funciones en programación funcional son **funciones puras**, es decir, funciones que lo único que hacen es calcular su resultado (sin ningún otro efecto) y en las que ese resultado sólo depende de los datos de entrada.

Además, los valores nunca cambian porque no tienen estado interno que se pueda alterar con el tiempo.

Como consecuencia de todo lo anterior, en programación funcional es posible sustituir cualquier expresión por su valor, propiedad que se denomina **transparencia referencial**.

En programación funcional, las funciones también son valores, por lo que se consideran a éstas como **ciudadanas de primera clase**.

Un programa funcional está formado únicamente por definiciones de valores y por expresiones que hacen uso de los valores definidos.

Por tanto, en programación funcional, ejecutar un programa equivale a evaluar una expresión.

Para describir el proceso llevado a cabo por el programa no es necesario bajar al nivel de la máquina, sino que basta con interpretarlo como un **sistema de evaluación de expresiones**.

Esa evaluación de expresiones se lleva a cabo mediante **reescrituras** que usan las definiciones para tratar de alcanzar la forma normal de la expresión.

1.2. Transparencia referencial

En programación funcional, **el valor de una expresión depende, exclusivamente, de los valores de las subexpresiones que la forman**.

Dichas subexpresiones, además, pueden ser sustituidas libremente por otras que tengan el mismo valor.

A esta propiedad se la denomina **transparencia referencial**.

Formalmente, se puede definir así:

Transparencia referencial:

Si $p = q$, entonces $f(p) = f(q)$.

En la práctica, eso significa que la evaluación de una expresión no puede provocar **efectos laterales** y que su valor no puede depender del momento en el que se evalúe la expresión (**la expresión siempre va a valer lo mismo**).

En consecuencia, un requisito para conseguir la transparencia referencial es que el valor de una expresión no dependa de cuándo se evalúe.

Es decir: **una expresión en programación funcional siempre debe tener el mismo valor**.

Por tanto, en programación funcional no se permite que la misma expresión, evaluada en dos momentos diferentes, dé como resultado dos valores diferentes.

Asimismo, el valor de una expresión tampoco debe depender del orden en el que se evalúen sus subexpresiones.

1.2.1. Efectos laterales

Los **efectos laterales** son aquellos que provocan un cambio de estado irremediable en el sistema, que además son observables fuera del contexto donde se producen y que puede dar lugar a que una misma expresión tenga valores diferentes según el momento en el que se evalúe.

Por ejemplo, las **instrucciones de E/S** (entrada y salida) provocan efectos laterales, ya que:

- Al **leer un dato** de la entrada (ya sea el teclado, un archivo del disco, una base de datos...) estamos afectando al estado del dispositivo de entrada, y además no se sabe de antemano qué valor se va a recibir, ya que éste proviene del exterior y no lo controlamos.
- Al **escribir un dato** en la salida (ya sea la pantalla, un archivo, una base de datos...) estamos realizando un cambio que afecta irremediablemente al estado del dispositivo de salida.

En posteriores temas veremos que existe un paradigma (el **paradigma imperativo**) que se basa principalmente en provocar efectos laterales.

Uno de los requisitos para alcanzar la transparencia referencial es que no existan efectos laterales.

Por tanto, en programación funcional no están permitidos los efectos laterales.

Eso significa que:

- Al evaluar una expresión no se pueden provocar efectos laterales.

  Si esto ocurriera, no podríamos sustituir una expresión por su valor.

- El valor de una expresión no puede depender de un efecto lateral ni verse afectado por la existencia de efectos laterales.

  Si esto ocurriera, la expresión podría tener valores distintos en momentos distintos.

En cualquiera de los dos casos, se rompería la transparencia referencial.

1.3. Modelo de ejecución

Cuando escribimos programas (y algoritmos) nos interesa abstraernos del funcionamiento detallado de la máquina que va a ejecutar esos programas.

Nos interesa buscar una *metáfora*, un símil de lo que significa ejecutar el programa.

De la misma forma que un arquitecto crea modelos de los edificios que se pretenden construir, los programadores podemos usar modelos que *simulan* en esencia el comportamiento de nuestros programas.

Esos modelos se denominan **modelos computacionales** o **modelos de ejecución**.

Los modelos de ejecución nos permiten **razonar** sobre los programas sin tener que ejecutarlos.

Definición:

> **Modelo de ejecución:**
>
> Es una herramienta conceptual que permite a los programadores razonar sobre el funcionamiento de un programa sin tener que ejecutarlo directamente en el ordenador.

Podemos definir diferentes modelos de ejecución dependiendo, principalmente, de:

- El paradigma de programación utilizado (ésto sobre todo).
- El lenguaje de programación con el que escribamos el programa.
- Los aspectos que queramos estudiar de nuestro programa.

1.3.1. Modelo de sustitución

En programación funcional, **un programa es una expresión** y lo que hacemos al ejecutarlo es **evaluar dicha expresión**, usando para ello las definiciones de operadores y funciones predefinidas por el lenguaje, así como las definidas por el programador en el código fuente del programa.

Recordemos que la **evaluación de una expresión**, en esencia, es el proceso de **sustituir**, dentro de ella, unas *subexpresiones* por otras que, de alguna manera bien definida, estén *más cerca* del valor a calcular, y así hasta calcular el valor de la expresión al completo.

Por ello, la ejecución de un programa funcional se puede modelar como un **sistema de reescritura** al que llamaremos **modelo de sustitución**.

El modelo de sustitución es un buen modelo de ejecución para la programación funcional gracias a que se cumple la *transparencia referencial*.

La ventaja del modelo de sustitución es que no necesitamos recurrir a pensar que debajo de todo esto hay un ordenador con una determinada arquitectura *hardware*, que almacena los datos en celdas de la memoria principal, que ejecuta ciclos de instrucción en la CPU, que las instrucciones modifican los datos de la memoria...

Todo resulta mucho más fácil que eso, ya que **todo se reduce a evaluar expresiones**, reescribiendo unas subexpresiones por otras, sin importar aspectos secundarios como la tecnología, el momento en el que se evalúan, el orden en el que se evalúan, etc.

Y la evaluación de expresiones no requiere pensar que hay un ordenador que lleva a cabo el proceso de evaluación.

Esto se debe a que la programación funcional se basa en el **cálculo lambda**, que es un modelo teórico matemático.

Ya estudiamos que evaluar una expresión consiste en encontrar su forma normal.

En programación funcional:

- Los intérpretes alcanzan este objetivo a través de múltiples pasos de **reducción** de las expresiones para obtener otra equivalente más simple.
- Toda expresión posee un valor definido, y ese valor **no depende del orden ni el momento** en el que se evalúe.
- El significado de una expresión es su valor, y **no puede ocurrir ningún otro efecto**, ya sea oculto o no, en ninguna operación que se utilice para calcularlo.

2. Definiciones

2.1. Definiciones

Introduciremos ahora en nuestro lenguaje una nueva instrucción (técnicamente es una **sentencia**) con la que vamos a poder hacer **definiciones**.

A esa sentencia la llamaremos **definición**, y expresa el hecho de que **un *nombre* representa un *valor***.

Las definiciones tienen la siguiente sintaxis:

⟨*definición*⟩ ::= **`identificador =`** ⟨*expresión*⟩

Por ejemplo:

```
x = 25
```

A partir de ese momento, el identificador `x` representa el valor `25`.

Y si `x` vale `25`, la expresión `2 + x * 3` vale `77`.

2.2. Identificadores y ligaduras (*binding*)

Los **identificadores** son los nombres o símbolos que representan a los elementos del lenguaje.

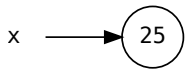
Cuando hacemos una definición, lo que hacemos es asociar un identificador con un valor.

Esa asociación se denomina **ligadura** (o ***binding***).

Por esa razón, también se dice que **una definición crea una ligadura**.

También decimos que el identificador está **ligado** (***bound***).

Lo representaremos gráficamente así:

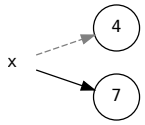


En Python (a diferencia de lo que ocurre en un **lenguaje funcional *puro***) las ligaduras empiezan a existir en el momento en que se ejecuta su definición, no antes.

2.2.1. Ligaduras irrompibles

En un **lenguaje funcional *puro***, un identificador ya ligado no se puede ligar a otro valor. Por ejemplo, lo siguiente daría un error en un lenguaje funcional puro:

```
x = 4  # ligamos el identificador x al valor 4
x = 7  # intentamos ligar x al valor 7, pero ya está ligado al valor 4
```



En consecuencia, **las ligaduras** entre nombres y valores **no se pueden romper**, de forma que un nombre, una vez ligado a un valor, **no se puede volver a ligar a otro valor distinto** durante la ejecución del programa (efecto que se conoce como *rebinding*).

En la práctica, eso significa que el nombre representa un **dato *constante***.

Que las ligaduras sean irrompibles son un requisito necesario para alcanzar la **transparencia referencial**.

Si hago:

```
x = "Hola"
```

en un lenguaje funcional puro, luego no puedo hacer:

```
x = "Hala"
```

porque eso sería hacer un *rebinding* y provocaría que, de nuevo, la expresión `x` tuviera distintos valores según el momento, lo que va en contra de la transparencia referencial.

Python no es un lenguaje funcional puro, por lo que sí se permite volver a ligar el mismo identificador a otro valor distinto.

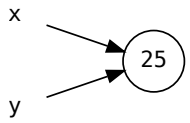
- Al hacer esto, se estaría perdiendo el valor anterior.
- Así que, por ahora, el *rebinding* está prohibido para nosotros (**no lo hagamos**).

Lo que sí se puede hacer es:

```
x = 25
y = x
```

En este caso estamos ligando a `y` el mismo valor que tiene `x`, algo perfectamente válido en un lenguaje funcional.

Lo que hace el intérprete en este caso no es crear dos valores `25` en memoria (sería un gasto inútil de memoria), sino que `x` e `y` *comparten* el único valor `25` que existe:

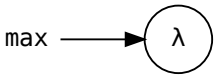


Por tanto:

```
>>> x = 25
>>> y = x
>>> y
25
```

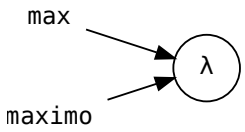
El **nombre** de una **función** es un identificador que está ligado a la función correspondiente (que en programación funcional es un valor como cualquier otro).

Por ejemplo, `max` es un identificador ligado a la función que devuelve el máximo de dos números (que representaremos aquí como **λ**):



Así que ese valor se puede ligar a otro identificador y, de esta forma, ambos identificadores compartirían el mismo valor (y, por tanto, representarían a la misma función). Por ejemplo:

```
>>> maximo = max
>>> maximo(3, 4)
4
```



2.2.2. Inmutabilidad

Para alcanzar la transparencia referencial, también es necesario que **los objetos** de datos que manipula el programa (es decir, los valores) **no tengan un estado interno** que pueda cambiar durante la ejecución del programa.

Eso se consigue haciendo que los valores sean **inmutables**.

Por ejemplo, en Python las cadenas son inmutables porque, una vez creadas, no se pueden cambiar los caracteres que la forman.

Si hago `x = "Hola"`, luego no puedo cambiar el interior de esa cadena (por ejemplo, cambiando la `'o'` por una `'a'`), porque entonces la expresión `x` tendría distintos valores dependiendo del momento en el que se evalúe, lo que va en contra de la transparencia referencial.

Eso significa que en programación funcional tampoco estaría permitido hacer cosas como esta:

```
import math
math.constante = 405
```

ya que entonces estaríamos **creando una nueva ligadura dentro del objeto** que representa al módulo `math`, lo que en la práctica supone que estamos **cambiando el estado interno de ese objeto** y, por tanto, estaría **dejando de ser inmutable**.

Recordemos que **los módulos son objetos** y, como tales, **son valores** como cualquier otro.

En cambio, sí sería correcto hacer algo así:

```
import math
constante = 405
```

ya que ahí no se está cambiando el estado interno de ningún valor.

2.2.3. Reglas léxicas

Cuando hacemos una definición debemos tener en cuenta ciertas cuestiones relativas al identificador:

- ¿Cuál es la **longitud máxima** de un identificador?
- ¿Qué **caracteres** se pueden usar?
- ¿Se distinguen **mayúsculas** de **minúsculas**?
- ¿Coincide con una palabra clave o reservada?
  - **Palabra clave**: palabra que forma parte de la sintaxis del lenguaje.

* **Palabra reservada**: palabra que no puede emplearse como identificador.

En Python, los identificadores pueden ser combinaciones de letras minúsculas y mayúsculas (y distingue entre ellas), dígitos y subrayados (_), no pueden empezar por un dígito, no pueden coincidir con una palabra reservada y pueden tener cualquier longitud.

2.2.4. Tipo de un identificador

Cuando un identificador está ligado a un valor, a efectos prácticos el identificador actúa como si fuera el valor.

Como cada valor tiene un tipo de dato asociado, también podemos hablar del **tipo de un identificador**.

El **tipo de un identificador** es el tipo del dato con el que está ligado.

Si un identificador no está ligado, no tiene sentido preguntarse qué tipo tiene.

2.2.5. Anotaciones de tipo en definiciones

Las **anotaciones de tipo** (en inglés, *type hints*) en Python son una característica **opcional** que permite indicar qué tipo de datos se espera que tenga un identificador.

En otros lenguajes (sobre todo, de tipado estático) se denominan *declaraciones de tipo*.

Sirven principalmente como documentación y para herramientas de análisis estático (como mypy, Pyright o Pyre) que comprueban tipos antes de ejecutar el código, pero no son comprobadas en tiempo de ejecución por Python.

Si incorporamos las anotaciones de tipo en las sentencias de definición, la sintaxis de las mismas queda así:

⟨*definición*⟩ ::= **identificador** [**:** ⟨*tipo*⟩] **=** ⟨*expresión*⟩

siendo ⟨*tipo*⟩ una expresión que represente un tipo del lenguaje.

Por ejemplo:

```
edad: int = 50
nombre: str = "Ana"
precio: float = 19.95
activo: bool = True
```

Hay que tener en cuenta que el intérprete no comprueba en ningún momento que el tipo anotado a un identificador se corresponda realmente con el tipo del valor ligado a dicho identificador.

Eso significa que lo siguiente, aunque incorrecto, se ejecutaría sin ningún error por parte del intérprete:

```
edad: str = 50
```

El uso de anotaciones de tipo, en cambio, mejora la legibilidad del código y permite la comprobación con herramientas externas, las cuales sí podrían detectar la línea anterior como incorrecta mediante un análisis estático (sin ejecutar) del código.

2.2.6. Otras formas de crear ligaduras

Las definiciones no son las únicas construcciones del lenguaje que crean ligaduras.

Por ejemplo, los módulos se ligan con sus nombres al hacer `import` o `from ... import`.

Otras formas de crear ligaduras que aún no hemos visto incluyen las siguientes:

- Los parámetros se ligan a sus argumentos en las llamadas a funciones.
- Las clases se ligan con sus nombres en las definiciones de clase.
- Las funciones se ligan con sus nombres en las definiciones de funciones imperativas.
- Los bucles `for` o los gestores de contexto crean ligaduras.
- Las expresiones generadoras, así como las listas por comprensión, los conjuntos por comprensión y los diccionarios por comprensión.

2.3. Espacios de nombres

Ciertas estructuras o construcciones sintácticas del programa definen **espacios de nombres**.

Un **espacio de nombres** (del inglés, *namespace*) es una correspondencia entre nombres y valores; es decir, es un **conjunto de ligaduras**.

Su función es, por tanto, **almacenar ligaduras**.

En un espacio de nombres, **un identificador sólo puede tener como máximo una ligadura**. En cambio, el mismo identificador puede estar ligado a diferentes valores en diferentes espacios de nombres.

Como un programa puede tener varios espacios de nombres, es posible tener varias ligaduras diferentes con el mismo nombre en distintas partes del programa.

Decimos que una ligadura es **local** al espacio de nombres donde se almacena, o que **su almacenamiento es local** a ese espacio de nombres.

Los espacios de nombres pueden estar ***anidados***, es decir, que puede haber espacios de nombres dentro de otros espacios de nombres.

Durante la ejecución de un programa se pueden ir creando en la memoria ciertas estructuras de datos que representan *espacios de nombres*.

En Python, estas estructuras pueden ser:

- Los **marcos** que se crean al ejecutar *scripts* y al invocar funciones (o métodos) definidas por el programador.
- Los **objetos**.
- Las **clases** (que también son objetos).
- Los **módulos** (que también son objetos).

En resumen: los **marcos** y los **objetos** son los únicos espacios de nombres que existen en Python.

Un espacio de nombres muy importante en Python es el que incluye las **definiciones predefinidas del lenguaje** (funciones como `max` o `sum`, tipos como `str` o `int`, etc.)

Ese espacio de nombres se denomina `__builtins__` y viene implementado en forma de *módulo* que se importa automáticamente en cada sesión interactiva o cada vez que se arranca un programa Python.

Pero sabemos que también podemos usar directamente las definiciones que contiene, por lo que el efecto es como si Python ejecutara las siguientes dos sentencias nada más entrar en el intérprete:

```
import __builtins__
from __builtins__ import *
```

Esto no es exactamente así en realidad, pero por ahora haremos como si así fuera, por simplicidad.

2.4. Marcos (*frames*)

Un **marco** (del inglés, *frame*) es una estructura que se crea en memoria para **representar la ejecución o *activación* de un *script* de Python o una función o método definido por el programador** en Python o Java.

Los marcos son **espacios de nombres** y, entre otras cosas, almacenan las ligaduras que se definen dentro de ese *script*, función o método.

Los marcos son conceptos ***dinámicos***:

- Se crean y se destruyen a medida que la ejecución del programa pasa por ciertas partes del mismo.
- Van almacenando nuevas ligaduras conforme se van ejecutando nuevas instrucciones que crean las ligaduras.

Por ahora, el único marco que existe en nuestros programas es el llamado ***marco global***, también llamado **espacio de nombres global**.

El marco global se crea en el momento en que **se empieza a ejecutar el programa** y existe durante toda la ejecución del mismo (sólo se destruye al finalizar la ejecución del programa).

Si se está trabajando en una sesión con el intérprete interactivo, el marco global **se crea justo al empezar la sesión** y existe durante toda la sesión (sólo se destruye al salir de la misma).

Por tanto, las definiciones que se ejecutan directamente en una sesión interactiva con el intérprete, crean ligaduras que se almacenan en el marco global.

Por ejemplo, si iniciamos una sesión con el intérprete interactivo y justo a continuación tecleamos lo siguiente:

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 25
```

```
6 >>> x
7 25
```

- Aquí estamos trabajando con el *marco global* (el único marco que existe hasta ahora para nosotros).
- En la línea 1, el identificador x aún no está ligado, por lo que su uso genera un error (el marco global no contiene hasta ahora ninguna ligadura para x).
- En la línea 6, en cambio, el identificador puede usarse sin error ya que ha sido ligado previamente en la línea 5 (el marco global ahora contiene una ligadura para x con el valor 25).

Los marcos son espacios de nombres dinámicos, que se van creando y destruyendo durante la ejecución del programa.

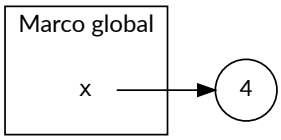
Igualmente, las ligaduras que contiene también se van creando y destruyendo a medida que se van ejecutando las instrucciones que forman el programa.

Si tenemos la siguiente sesión interactiva:

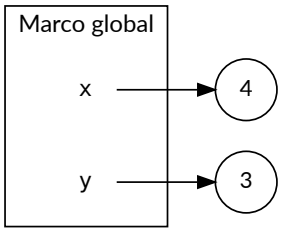
```
1 >>> 2 + 3
2 5
3 >>> x = 4
4 >>> y = 3
5 >>> z = y
```

Según hasta donde hayamos ejecutado, el marco global contendrá lo siguiente:

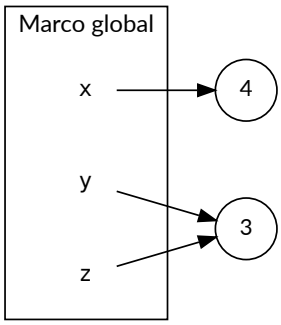
Marco global en las líneas 1–2



Marco global en la línea 3



Marco global en la línea 4



Marco global en la línea 5

Hemos visto que una ligadura es una asociación entre un identificador y un valor.

También hemos visto que los espacios de nombres almacenan ligaduras, y que un marco es un espacio de nombres.

Por tanto, **los marcos almacenan ligaduras, pero *NO* almacenan los valores** a los que están asociados los identificadores de esas ligaduras.

Por eso hemos dibujado a los valores fuera de los marcos en los diagramas anteriores.

Los valores se almacenan en una zona de la memoria del intérprete conocida como el **montículo**.

En cambio, los marcos se almacenan en otra zona de la memoria conocida como la **pila de control**, la cual estudiaremos mejor más adelante.

2.5. Evaluación de expresiones con identificadores

Podemos usar un identificador ligado dentro de una expresión, siempre que la expresión sea válida según las reglas del lenguaje.

El identificador representa a su valor ligado y se evalúa a dicho valor en cualquier expresión donde aparezca ese identificador:

```
>>> x = 25
>>> 2 + x * 3
```

```
77
```

Intentar usar en una expresión un identificador no ligado provoca un error de tipo `NameError` (*nombre no definido*):

```
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

2.5.1. Resolución de identificadores

Durante la evaluación de una expresión, para cada uno de los diferentes identificadores que aparecen en ella, habrá que comprobar si ese identificador está ligado y a qué valor.

- Si no está ligado, es un error de *nombre no definido*.
- En caso contrario, tendrá que determinar a qué valor está ligado para poder sustituir, en la expresión, cada aparición del identificador por su valor.

Para ello, habrá que buscar una ligadura con ese identificador en uno o varios espacios de nombres.

El proceso de localizar (si es que existe) la ligadura adecuada que liga a un identificador con su valor, se denomina **resolución del identificador**.

En general, al proceso de determinar con qué valores están ligados los identificadores de un programa se le denomina **resolución de identificadores** o **resolución de nombres**.

Los espacios de nombres donde se buscan las ligaduras para ese identificador dependerán del *contexto* en el que se está intentando resolver dicho identificador, que básicamente depende de si queremos resolver un atributo de un objeto, o no.

Por ejemplo, en el siguiente código:

```
>>> x = 4
>>> x
4
```

se busca una ligadura para `x` en el marco global.

En cambio, si se intenta acceder a un atributo de un objeto:

```
>>> import math
>>> math.pi
3.141592653589793
```

se busca una ligadura para `pi` en el espacio de nombres asociado al módulo `math`.

La resolución de identificadores es un proceso que usa mecanismos distintos dependiendo de si el lenguaje es interpretado o compilado:

- Si es un **lenguaje interpretado** (como Python): el intérprete usa un concepto llamado **entorno** en tiempo de ejecución para localizar la ligadura.

  En tal caso, hablamos de **resolución de nombres *dinámica***.

- Si es un **lenguaje compilado** (como Java): el compilador determina, en tiempo de compilación, si una ligadura es accesible haciendo uso del concepto de **ámbito** y, en caso afirmativo, deduce en qué espacio de nombres está la ligadura.

  En tal caso, hablamos de **resolución de nombres *estática***.

En general, la resolución de identificadores puede ser una tarea complicada ya que puede involucrar muchos conceptos como espacios de nombres, ámbitos, entornos, reglas de visibilidad, sombreado, sobrecargas... muchos de los cuales aún no hemos estudiado.

3. *Scripts*

3.1. *Scripts*

Cuando tenemos muchas definiciones o muy largas, resulta tedioso tener que introducirlas una y otra vez cada vez que abrimos una nueva sesión con el intérprete interactivo.

Lo más cómodo es teclearlas todas una sola vez dentro un archivo que luego cargaremos desde dentro del intérprete.

Ese archivo se llama ***script*** y, por ahora, contendrá una lista de las definiciones que nos interese usar en nuestras sesiones interactivas con el intérprete.

Al cargar el *script*, se ejecutarán sus instrucciones una tras otra casi de la misma forma que si las estuviéramos tecleando nosotros directamente en nuestra sesión con el intérprete, en el mismo orden.

Llegado el momento, los *scripts* contendrán el código fuente de nuestros programas y los ejecutaremos desde el intérprete por lotes.

Los nombres de archivo de los *scripts* en Python llevan extensión `.py`.

Para cargar un *script* en nuestra sesión tenemos dos opciones:

1. Usar la orden **`from`** dentro de la sesión actual.

   Por ejemplo, para cargar un *script* llamado `definiciones.py`, usaremos:

   ```
   >>> from definiciones import *
   ```

   Observar que en el **`from`** se pone el nombre del script pero **sin la extensión `.py`**.

2. Iniciar una nueva sesión con el intérprete interactivo indicándole que cargue el *script* mediante la opción `-i` en la línea de órdenes del sistema operativo:

   ```
   $ python -i definiciones.py
   >>>
   ```

   En este caso **sí** se pone el nombre completo del script, **con la extensión `.py`**.

A partir de ese momento, en el intérprete interactivo podremos usar las definiciones que se hayan cargado desde del *script*.

Por ejemplo, si el *script* `definiciones.py` tiene el siguiente contenido:

```
x = 25
j = 14
```

Al cargar el *script* (usando cualquiera de las dos opciones que hemos visto anteriormente) se ejecutarán sus instrucciones (las dos definiciones) y, en consecuencia, se crearán en el marco global las ligaduras `x` → `25` y `j` → `14`:

```
>>> x                                  # antes de cargar el script, da error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> from definiciones import *         # aquí cargamos el script, y ahora
>>> x                                  # la x sí está ligada a un valor
25
```

Una limitación importante que hay que tener en cuenta es que **el *script* sólo puede usar definiciones que se hayan creado en el mismo *script*** (exceptuando las definiciones predefinidas del lenguaje).

Por ejemplo, si tenemos el siguiente *script* llamado `prueba.py`:

```
j = w + 1
```

donde se intenta ligar a `j` el valor ligado a `w` más uno, lo siguiente no funcionará:

```
>>> w = 3
>>> from prueba import *
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ricardo/python/prueba.py", line 1, in <module>
    j = w + 1
NameError: name 'w' is not defined
```

Aunque `w` esté ligado a un valor al cargar el *script*, éste no podrá acceder a esa ligadura y da error de «*nombre no definido*».

Cuando estudiemos la programación modular entenderemos por qué.

4. Documentación interna

4.1. Concepto

El uso apropiado de una clara disposición del texto, la inclusión de comentarios apropiados y una buena elección de los identificadores, se conoce como **documentación interna** o **autodocumentación**.

La autodocumentación no es ningún lujo superfluo; por el contrario, se considera preciso adquirir desde el principio el hábito de desarrollar programas claros y bien autodocumentados.

4.2. Comentarios

Los comentarios en Python empiezan con el carácter # y se extienden hasta el final de la línea.

Los comentarios pueden aparecer al comienzo de la línea o a continuación de un espacio en blanco o una porción de código.

Los comentarios no pueden ir dentro de un literal de tipo cadena.

Un carácter # dentro de un literal cadena es sólo un carácter más.

```
# este es el primer comentario
spam = 1  # y este es el segundo comentario
          # ... y este es el tercero
texto = "# Esto no es un comentario porque va entre comillas."
```

Cuando un comentario ocupa varias líneas, se puede usar el «truco» de poner una cadena con triples comillas:

```
x = 1
"""
Esta es una cadena
que ocupa varias líneas
y que actúa como comentario.
"""
y = 2
```

Python evaluará la cadena pero, al no usarse dentro de ninguna expresión ni ligarse a ningún identificador, simplemenata la ignorará (como un comentario).

Los comentarios:

- Deben expresar información que no sea evidente por la simple lectura del código fuente.
- No deben decir lo mismo que el código, porque entonces será un comentario *redundante* que, si no tenemos cuidado, puede acabar diciendo algo incompatible con el código si no nos preocupamos de actualizar el comentario cuando cambie el código (que lo hará).
- Deben ser los justos: ni más ni menos que los necesarios.

4.3. Identificadores significativos

Se recomienda usar identificadores descriptivos.

Es mejor usar:

```
ancho = 640
alto = 400
superficie = ancho * alto
```

que

```
x = 640
y = 400
z = x * y
```

aunque ambos programas sean equivalentes en cuanto al efecto que producen y el resultado que generan.

Si el identificador representa varias palabras, se puede usar el carácter de guión bajo (_) para separarlas y formar un único identificador:

```
altura_triangulo = 34.2
```

4.4. Estándares de codificación

Un estándar de codificación (o estándar de programación) es un conjunto de normas, reglas y recomendaciones que definen cómo debe escribirse el código fuente en un lenguaje de programación dentro de un proyecto, equipo o empresa.

Los objetivos de un estándar de codificación son:

- Mejorar la legibilidad y comprensión del código por parte de cualquier miembro del equipo.
- Facilitar el mantenimiento y la corrección de errores.
- Asegurar consistencia en el estilo y estructura del código.
- Reducir la probabilidad de errores por prácticas incorrectas o confusas.
- Permitir una colaboración eficiente en proyectos donde varias personas contribuyen.

Qué suele incluir un estándar de codificación:

- Nombres de variables, funciones y clases (por ejemplo, usar `snake_case`, `camelCase` o `PascalCase`).
- Formato y estilo: sangrías, espacios, líneas en blanco.
- Estructura de archivos y carpetas en el proyecto.
- Convenciones de comentarios y documentación.
- Buenas prácticas específicas del lenguaje (por ejemplo, en Python, seguir la norma PEP 8).
- Reglas sobre uso de excepciones, manejo de errores y *logging*.
- Restricciones o recomendaciones de diseño de software (por ejemplo, evitar funciones demasiado largas o complejas).

4.4.1. PEP 8

En Python, el estándar de codificación más utilizado es PEP 8, que define:

- Sangría de 4 espacios.
- Uso de `snake_case` para nombres de funciones y variables.
- Uso de `PascalCase` para nombres de clases.
- Líneas de máximo 79 caracteres.
- Espacios alrededor de operadores.

4.4.2. `pylint`

`pylint` es una herramienta que comprueba determinado tipo de errores en el código fuente de un programa Python.

Trata de asegurar que el programa se ajusta a un estándar de codificación.

Localiza determinados patrones que están mal vistos o que pueden mejorarse fácilmente.

Sugiere cambios en el código, recomienda refactorizaciones y ofrece detalles sobre la complejidad del código.

Se puede instalar o manualmente haciendo:

```
$ pip install pylint
```

Desde la consola, `pylint` se ejecuta directamente sobre un archivo `.py`:

```
$ pylint prueba.py
************* Module prueba
prueba.py:1:0: C0114: Missing module docstring (missing-module-docstring)

-----------------------------------------------------------------------
Your code has been rated at 5.00/10 (previous run: 10.00/10, -5.00)
```

Se puede deshabilitar la comprobación de determinados *defectos* usando la opción `--disable`:

```
$ pylint --disable=missing-docstring prueba.py

-----------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 5.00/10, +5.00)
```

Con la opción `--lists-msgs` se pueden consulta la lista de todos los comprobadores predefinidos.

Resumen

Resumen

Resumiendo, los conceptos fundamentales sobre los que se asienta la programación funcional son:

- Casi todas las instrucciones son expresiones, no sentencias.
- Las definiciones son las únicas sentencias de un programa funcional.
- Transparencia referencial.
- Ausencia de efectos laterales.
- Funciones puras.
- El valor de una expresión no depende de nada ajeno a la misma, sólo de las subexpresiones que la forman.
- Ligaduras irrompibles.
- Inmutabilidad.
- Las funciones también son valores.
- Ejecutar un programa es evaluar una expresión.
- No importa el orden en el que se ejecuten las instrucciones.
- No importa el orden en el que se evalúen las subexpresiones de una expresión.

Y otros dos conceptos fundamentales que aún no hemos estudiado pero que veremos en breve:

- Abstracciones lambda.
- Funciones de orden superior.

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.

Blanco, Javier, Silvina Smith, and Damián Barsotti. 2009. *Cálculo de Programas*. Córdoba, Argentina: Universidad Nacional de Córdoba.

Van-Roy, Peter, and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Cambridge, Mass: MIT Press.