



Universidad Complutense de Madrid

---

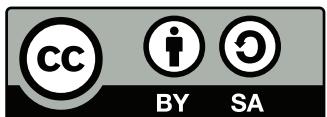
Laboratorio de Computación Científica

Laboratory for Scientific Computing

Introducción a la programación en Python ✯ Introduction to  
Python Programming

Juan Jiménez  
Héctor García de Marina  
Lía García

3 de septiembre de 2024



El contenido de estos apuntes está bajo licencia Creative Commons Atribución-ShareAlike 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>  
©Juan Jiménez

# Índice general

<b>1. Introducción a la programación en Python</b>	
<b>Introduction to Python Programming</b>	<b>9</b>
1.1. Un entorno de programación para Python . . . . .	10
1.1. A python's development environment . . . . .	10
1.1.1. Anaconda. Una distribuci- ón de herramientas de computación de software abierto . . . . .	10
1.1.1. Anaconda. A free software computing tools distribution . . . . .	10
1.1.2. Spyder . . . . .	14
1.1.2. Spyder . . . . .	14
1.1.3. Variables . . . . .	17
1.1.3. Variables . . . . .	17
1.2. Operaciones aritméticas, relacionales y lógicas. . . . .	26
1.2. Arithmetical, relational and logical operations . . . . .	26
1.2.1. Operaciones aritméticas . . . . .	26
1.2.1. Arithmetic operations . . . . .	26
1.2.2. Precedencia de los operadores aritméticos . . . . .	28
1.2.2. Arithmetic operator precedence . . . . .	28
1.2.3. Operaciones Relacionales y lógicas. . . . .	30
1.2.3. Relational and logical operations. . . . .	30
1.3. Scripts en Python . . . . .	36
1.3. Scripts in Python . . . . .	36
1.3.1. El editor de textos de Spyder. . . . .	37
1.3.1. The Spyder's text editor . . . . .	37
1.3.2. Ejecución de scripts . . . . .	40
1.3.2. Running a script . . . . .	40
1.4. Funciones en Python . . . . .	41
1.4. Functions in Python . . . . .	41
1.5. Los <i>Namespace</i> en Python y el ámbito de las variables . . . . .	46
1.5. Namespace and scope in Python . . . . .	46
1.6. Depuración . . . . .	52
1.6. Debugging . . . . .	52
1.7. Control de flujo . . . . .	57
1.7. Flow control . . . . .	57
1.7.1. Flujo condicional. . . . .	57
1.7.1. Conditional flow. . . . .	57
1.7.2. Bucles . . . . .	65
1.7.2. Loops . . . . .	65
1.7.3. Funciones recursivas. . . . .	75

1.7.3. recursive functions . . . . .	75
1.7.4. Algoritmos y diagramas de flujo. . . . .	77
1.7.4. Algorithms and flow charts. . . . .	77
1.8. Exportar e importar datos en Python . . . . .	82
1.7.5. Reading and writing data in Pyhton . . . . .	82
1.9. Ejercicios . . . . .	90

# Índice de figuras

1.1. Ventana de Anaconda Navigator. Se ha señalado en rojo el icono correspondiente a Spyder.	11
1.2. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.	12
1.3. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.	12
1.4. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.	14
1.4. Anaconda-Navgator window. The Environments button has been encircled in red	14
1.5. Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.	15
1.5. Anaconda-Navgator window. The Environments button has been encircled in red	15
1.6. Spyder: Entorno de desarrollo integrado para Python	16
1.6. Spyder: An integer development environment for Python	16
1.7. Tipos de datos en Python.	20
1.7. Data types in Python	20
1.8. Editor de textos para Python incluido en Spyder	38
1.8. Text editor for Python included in Spyder	38
1.9. Relación de inclusión entre los Namesapces de Python	50
1.9. Inclusion relationship among Python Namespaces	50
1.10. Un ejemplo de error de sintaxis (falta cerrar un paréntesis	53
1.10. A syntax error example (a closing bracket has been omitted	53
1.11. Breakpoint activo señalado con una flecha azul, y menú desplegable <i>debug</i>	55
1.11. Active Breackpoint pointed by a blue arrow and <i>debug</i> drop-down menu	55
1.12. Parada de programa en un breakpoint y herramientas de depuración	56
1.12. program stop on a breakpoint and debugging tools	56
1.13. Esquema general de la estructura de flujo condicional <code>if</code> los términos escritos entre paréntesis son opcionales.	62
1.13. General outline of an <code>if</code> conditional flow structure. Terms enclosed in parenthesis are optional	62
1.14. Esquema general de la estructura de un bucle <code>for</code> los términos escritos entre paréntesis son opcionales.	66
1.14. <code>for</code> structure General outline. Terms enclosed in parentheses are optional.	66
1.15. Esquema general de la estructura de un bucle <code>while</code> los términos escritos entre paréntesis son opcionales.	73
1.15. General structure for a while loop. Terms enclosed in parentheses are optional.	73
1.16. Símbolos empleados en diagramas de flujo	78
1.16. Symbols used in flowcharts	78
1.17. Diagrama de flujo para el problema de los números primos	80
1.17. Flow chart for the prime numbers problem	80



# Índice de cuadros

1.1.	Algunos métodos de las Listas en Python . . . . .	23
1.1.	Some methods of Python's List . . . . .	23
1.2.	Operadores aritméticos definidos en Python . . . . .	27
1.2.	Arithmetic operators defined in python . . . . .	27
1.3.	Operadores relacionales definidos en Python . . . . .	31
1.3.	Relational operators defined in Python . . . . .	31
1.4.	Operadores lógicos entre valores y variables . . . . .	32
1.4.	Logical operators on values and variables . . . . .	32



# Capítulo/Chapter 1

# Introducción a la programación en Python

## Introduction to Python Programming

### UN PARACAIDISTA SE DESNUCA

Por darle una sorpresa y a escondidas, su hacendosa, tierna e inocente esposa le cambió a última hora el paracaídas de reglamento por otro de punto, amorosamente tejido por ella, de malla ancha.

---

Julio Ceron. Diario ABC,  
2.03.1986. p.55

Este capítulo presenta una introducción general a la programación. Para su desarrollo, vamos a emplear uno de los lenguajes de programación que más aceptación ha tenido en los últimos años: Python. Este lenguaje fue desarrollado por Guido van Rossum a finales de los años 80 del siglo pasado. Es un lenguaje de alto nivel de propósito general.

El porqué emplear python está relacionado con la existencia de un gran número de *módulos* especialmente diseñados para el cálculo científico. Además, al tratarse de un lenguaje interpretado, es posible centrarse en el estudio de la programación en sí, sin tener que preocuparse de la compilación. En resumen, emplea-

This chapter presents a general introduction to programming. We will use Python, one of the currently most used programming languages, along their pages. Guido van Rossum developed Python in the late 1980s. It is a high-level, general-purpose programming language.

Why Python? Well, it has many useful *modules* that allow us to develop code for scientific computing efficiently and reliably. Moreover, being an interpreted language facilitates focusing on programming skills without bothering with the compiling process. We will use Python to learn basic general programming methods and solve numerical problems.

remos Python tanto para aprender a programar como para resolver problemas de cálculo científico.

Este capítulo no pretende ser exhaustivo, –cosa que por otro lado resulta imposible en el caso de Python–, sino tan solo dar una breve introducción a su uso. Afortunadamente, Python cuenta con una muy buena documentación, accesible a través de la Red.

## 1.1. Un entorno de programación para Python

En primer lugar antes de empezar a describir el lenguaje de programación, vamos a introducir las herramientas que usaremos. Es frecuente que los lenguajes de programación cuenten con lo que se conoce como un entorno de desarrollo integrado o, abreviadamente IDE (acrónimo tomado de su nombre en inglés: *integrated development environment*). Un IDE suministra un habitualmente un entorno gráfico y un conjunto de herramientas tales como ayuda en línea, un depurador o un editor que facilitan la construcción y el depurado del código. Nosotros vamos a emplear un IDE específicamente diseñada para cálculo científico, no es el único, ni es necesariamente el mejor. Pero es adecuado para empezar a trabajar con Python. Se trata de *Spyder*. Veamos en primer lugar como conseguirlo y como instalarlo en un ordenador personal.

### 1.1.1. Anaconda. Una distribución de herramientas de computación de software abierto

Entre los distintos medios en los que podemos instalar Python y algunas de sus herramientas de desarrollo, hemos seleccionado Anaconda porque es software libre, fácil de instalar e incluye Spyder. La Manera más sencilla de instalarlo es descargar el instalador de Anaconda de su página Web:

<https://www.anaconda.com/download>

This chapter is by no means exhaustive. It would be impossible in the case of a language like Python. It is just an introduction to its use. Fortunately, Python has excellent online documentation and plenty of examples on the Web.

## 1.1. A python's development environment

Before describing the programming language, we will introduce the tools we should use. It is common for programming languages to provide an *integrated development environment* (IDE). An IDE usually includes a graphic environment and tools, such as online help, a debugger, or a text editor, that simplify code development and debugging. We are going to use an IDE specifically designed for scientific computing. It's not the only one available; neither is perhaps the best one, but it is suitable to begin working with Python. We refer to *Spyder*. First, Let's see how to get it and install it on a personal computer.

### 1.1.1. Anaconda. A free software computing tools distribution

Among the many ways you may follow to install Python and some of its development tools, We have selected Anaconda because it is free software, easy to install, and includes Spyder. The easy way to install Anaconda is by downloading it from its Web page:

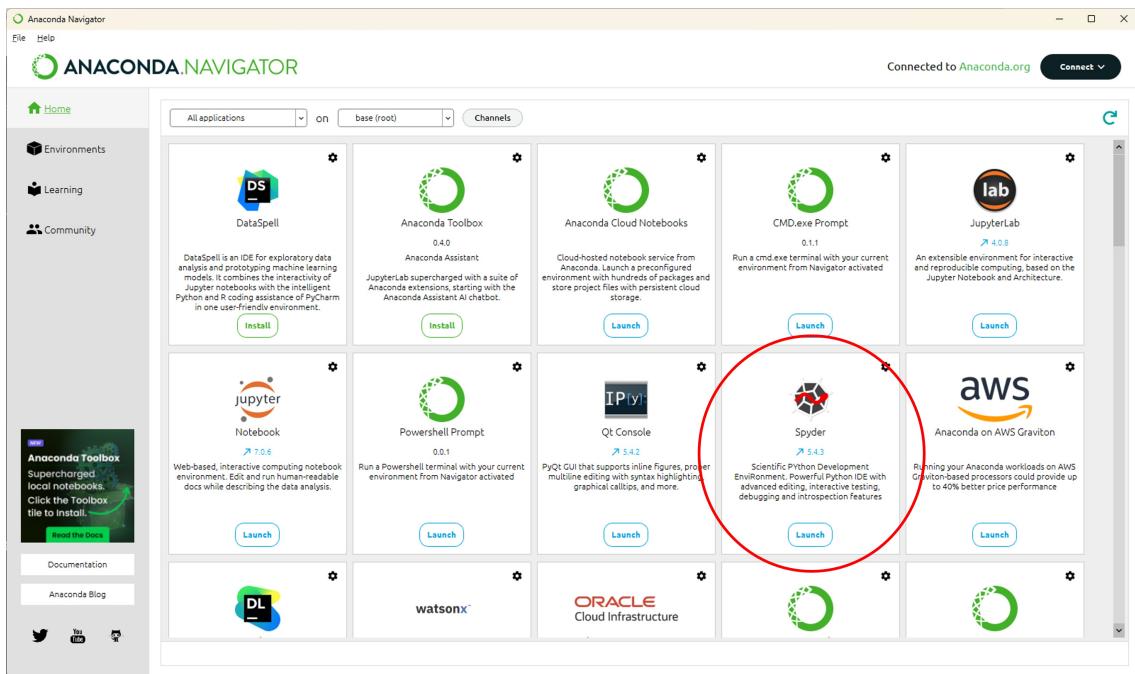


Figura 1.1: Ventana de Anaconda Navigator. Se ha señalado en rojo el ícono correspondiente a Spyder.

Hay instaladores de Anaconda disponibles para Windows, Linux y Mac. El proceso de instalación es en todos los casos bastante sencillo. La página web contiene además documentación en la que se explica detalladamente el funcionamiento de Anaconda.

Una vez instalado, Anaconda proporciona una aplicación (Navegador) desde la que manejarlo: *anaconda-navigator*. Si ejecutamos Anaconda-Navigator, obtendremos una ventana como la que se muestra en la figura 1.1.

El navegador de Anaconda, muestra un conjunto de iconos. Cada uno de ellos corresponde a una aplicación distinta. Muchas de ellas están relacionadas con IDEs especialmente diseñados para usar Python, otras están relacionadas con otros programas orientados al tratamiento de datos, las representaciones gráficas o la estadística. En algunos de los iconos aparece la palabra *launch*. Indica que la aplicación se encuentra instalada en nuestro ordenador,. Si pulsamos con el ratón sobre el botón *launch*, la aplicación se ejecuta. En otros casos, aparece la palabra *install*. Se trata de aplicaciones

Anaconda installers are available for Windows, Linux, and Mac. The installation process is straightforward. The Anaconda webpage offers complete information on installation and performance.

Once installed, it supplies an application *anaconda-navigator* to manage Anaconda. When we start Anaconda-Navigator, we get a window like that shown in figure 1.1.

Anaconda-Navigator shows a set of icons. Anyone of them is intended to launch a different application. Many of them are IDEs specifically designed to use Python. Others relate to other programs devoted to data processing, graphics, or statistics. The word *launch* is written in some of these icons. This means that the application has already been installed on our computer. Pressing the left-hand button of the mouse over the word *launch* opens the application. In other cases, the word *install* is written over the icon. In these cases, the application is not installed on our computer, but we can install it by pressing the left-hand button of the mouse over the word

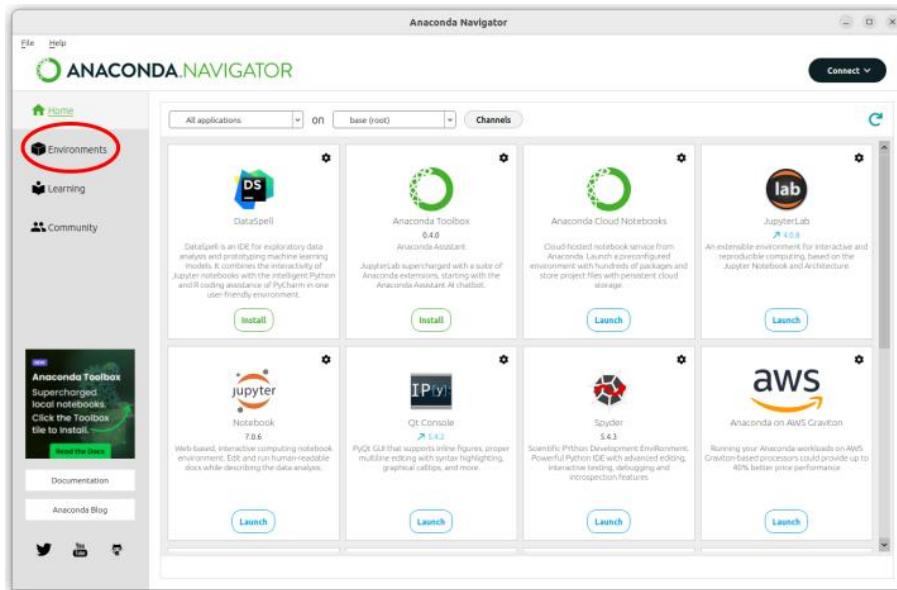


Figura 1.2: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.

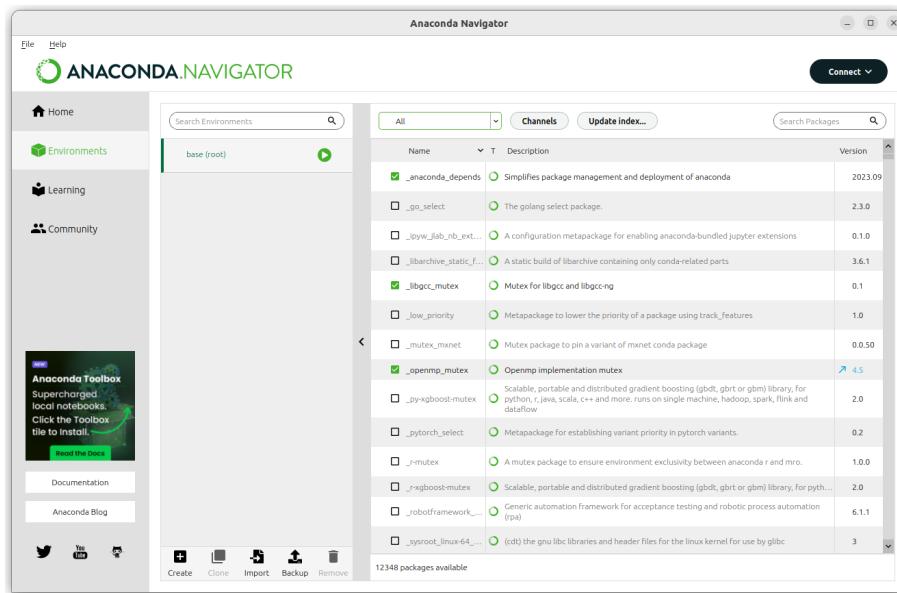


Figura 1.3: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.

que no están instaladas en nuestro ordenador, pero que pueden ser instaladas si pulsamos en botón *install*.

En nuestro caso, vamos a emplear Spyder, que aparece entre las aplicaciones ya instaladas, y que aparece rodeada en la figura 1.1 con un círculo rojo.

Antes de entrar en la descripción de spyder, vamos a emplear Anaconda navigator para instalar un plug-in, que nos permitirá manejar desde spyder, otro de los entornos de programación de Python más empleados actualmente: *Jupyter Notebook*. Jupiter Notebook, es en realidad un IDE para programar en Python independiente de Spyder. Es muy útil porque permite combinar código en python con texto y gráficos. Más adelante explicaremos como usarlo.

**Instalación de spyder-notebook.** para instalar el plug-in de Spyder que permite manejar notas de jupyter-notes, pulsamos en la ventana de Anaconda-Navigator el botón *Environments*, (enmarcado en rojo en la figura 1.2). La ventana de Anaconda-Navigator nos muestra ahora los entornos de trabajo de Anaconda, (figura 1.3). Por defecto, estaremos en el entorno base (root). Este será además el único entorno disponible si acabamos de hacer una instalación de Anaconda nueva en el ordenador. Si seleccionamos *All* en el menu desplegable situado en la parte superior izquierda del panel de la derecha, obtenemos una relación de todos los paquetes disponibles en anaconda. Aquellos que ya están instalados aparecen marcados a la izquierda con un cuadro verde, mientras que para aquellos disponibles pero no instalados el cuadro aparece en blanco.

En la parte superior derecha de éste panel hay un buscador. Si introducimos en el la palabra spyder, El panel nos mostrará los paquetes relacionados con Spyder (figura 1.4). Seleccionamos el paquete spyder notebook y en la parte inferior derecha del panel nos apareceran dos nuevos botones. Si seleccionamos el botón *apply*, Anaconda comenzará el proceso de instalación del paquete. Lo primero que hace es abrir un pop-up y comprobar las dependencias, una vez terminada la compro-

*install*.

We are going to use Spyder, which has already been installed. In figure 1.1, the Spyder icon has been enclosed in a red circle.

Before starting with a Spyder description, we will use Anaconda to install a plug-in, allowing us to use another of Phyton's most currently used IDEs: *Jupyter Notes*. Jupyter Notes is an IDE for Python code development. It is independent of Spyder. It is a handy tool because it allows us to combine Python code with graphics and text. We will describe Jupyter Notes later on.

**Spyder-notebook installation.** To install the plugin that allows using Jupyter Notes from Spyder, we press the mouse left button on the *Environments* icon (framed in red in figure 1.2). Then, the Anaconda-Navigator window shows us Anaconda work environments (figure 1.3). If we have just installed Anaconda, the only environment already available should be the base (root) environment. We will focus on the right-side panel. The upper left corner has a drop-down menu; we select the option *all*. Then, the panel shows us all the software packages available in the Anaconda default channel. Those packages installed in the computer are ticked with a green square label before the package name; packages available but not installed are ticked with a white square.

There is a browser located in the upper right corner of the panel. We will write the word *spyder* or *spyder notebook*. The panel now shows those software packages related to Spyder (figure 1.4). We select the package spyder notebook by clicking the white square before the package name. A new pair of push buttons appear on the panel's lower side. Clicking on *apply* triggers the Spyder notebook installation process. First, Anaconda launches a pop-up window and checks Spyder Notebook software dependencies. The process may take some time, so be patient until it finishes the checking. Once the checking process is over, we will press the *apply* button on the pop-up window and let Anaconda install the software.

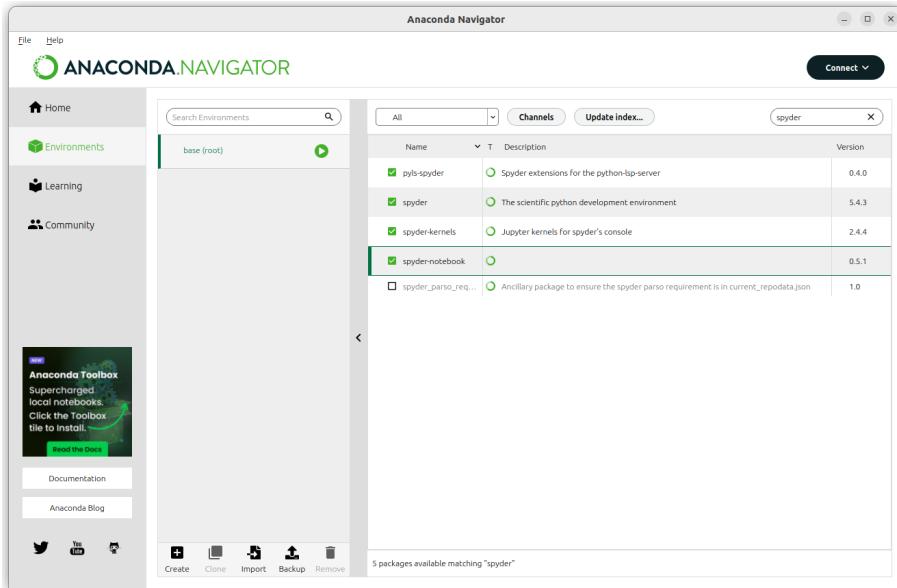


Figura 1.4: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.  
Figure 1.4: Anaconda-Navigator window. The Environments button has been encircled in red

bación, pulsamos el botón apply del pop-up y dejamos que Anaconda instale spyder notebook.

### 1.1.2. Spyder

Vamos a centrarnos en esta sección en describir Spyder. Como ya dijimos, Spyder es un entorno de programación integrado, especialmente pensado para trabajar con Python en computación científica. Para lanzar el entorno basta con que pulsemos el botón *launch* del ícono de Spyder, en la ventana de Anaconda-navigator (figura: 1.1). Al hacerlo, se nos abrirá una nueva ventana como la que se muestra en la figura 1.6.

En la ventana de Spyder podemos distinguir tres paneles distintos<sup>1</sup>. Además, en la parte superior de las ventanas tenemos una barra con menús desplegables y otra con botones. Vamos a describir brevemente algunas de las características principales del entorno a la vez que nos vamos introduciendo en las características fundamentales de la programación

<sup>1</sup>La configuración que Spyder que se describe aquí, es la configuración por defecto. El usuario puede cambiarla a su gusto modificando los valores por defecto.

### 1.1.2. Spyder

Let's focus on the Spyder description. As we said before, Spyder is an Integer Development Environment specially designed to use Python for scientific computing. To start Spyder, we push the *launch* button on the Spyder Icon, located in the Anaconda-Navigator window (figure 1.1). Then, a new window opens, as shown in figure 1.6.

This (Spyder) window has three different panels<sup>1</sup>. Besides, in the upper part of the window, there is a toolbar and a second bar that contains drop-down menus. We are going to give a brief description of the Spyder environment and, at the same time, some of Python programming's basic features. Anyway, we will not be exhaustive. An exhaustive description is far beyond the reach of these notes. The best way to learn to deal with Spyder is by

<sup>1</sup>We describe here the default Spyder configuration. The user may change this default configuration according to their preferences.

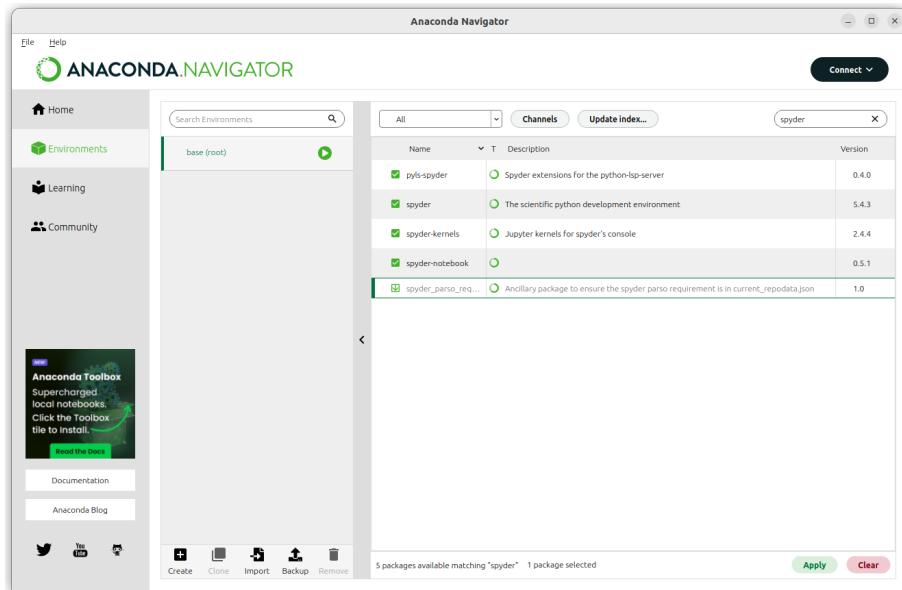


Figura 1.5: Ventana de Anaconda-Navigator, se ha señalado en rojo el botón Environments.  
Figure 1.5: Anaconda-Navgator window. The Environments button has been encircled in red

en Python. En cualquier caso, no vamos a dar una descripción exhaustiva. La mejor manera de aprender a manejar Spyder es usarlo y consultar la abundante documentación disponible.

**El terminal mejorado de Python** De los tres paneles mostrados en la figura 1.6, vamos a describir primero el situado abajo a la derecha. Este panel es un terminal, se conoce con el nombre de Ipython (Interactive python) e incluye muchas mejoras sobre el terminal estándar de python. Ipython nos muestra el símbolo `In [1]:`, que recibe el nombre de *prompt*, y a continuacion una barra vertical | parpadeante. El terminal permite al usuario interactuar directamente con Python; es decir, Python puede recibir instrucciones directamente a través del terminal, ejecutar las instrucciones, y devolver y/o guardar en memoria los resultados obtenidos. Veamos un ejemplo. Si escribimos en el terminal:

`In [1]: 2+2`

y pulsamos la tecla *intro*, Python calcula la suma pedida, muestra el resultado y nos des-

using it. Moreover, there is a vast amount of available documentation.

**The Python’s Enhance Terminal** Returning to the three panels shown in figure 1.6, we will focus first on the left-down one. This panel is a Python console. The console displays a symbol `In [1]:` known as the *prompt* followed by a flicking vertical bar. The console allows the user to interact with Python straight; i.e., Python can get direct instructions through the terminal, run the instructions, and show and/or save in memory the achieved results. Let’s see an example. If we write in the console:

`In [1]: 2+2`

and press the intro key; Python calculates the sum, shows the result using an output (*echo*) mark, `Out [1]:`, gives back a new clean prompt, and stops waiting for a new command:

`Out[1]: 4`

`In [2]:`

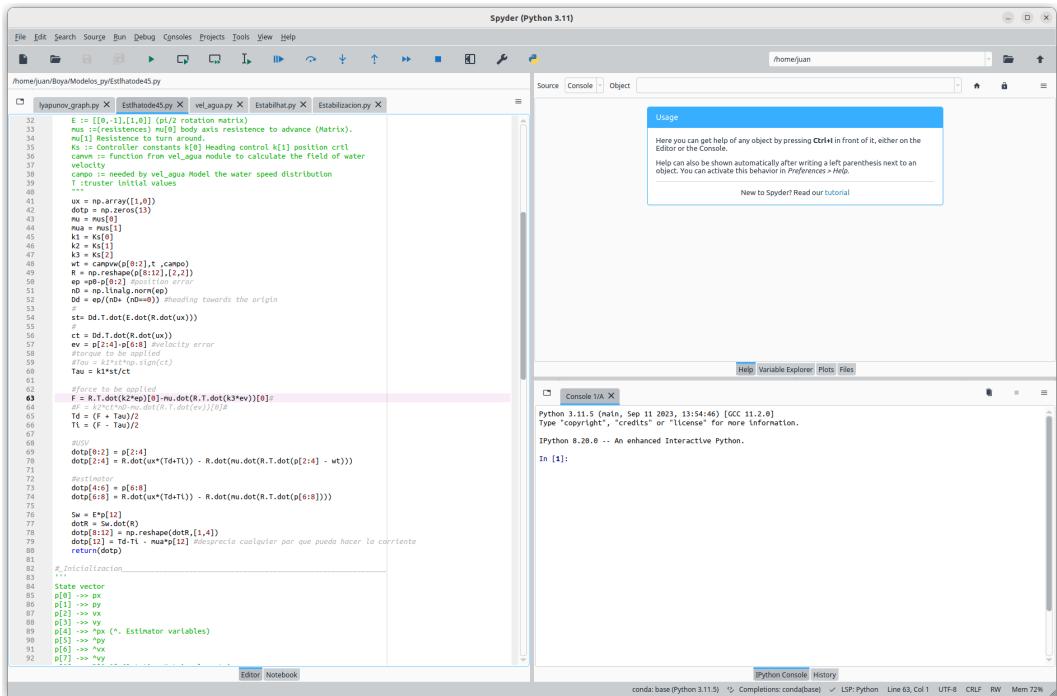


Figura 1.6: Spyder: Entorno de desarrollo integrado para Python  
Figure 1.6: Spyder: An integer development environment for Python

vuelve un nuevo prompt, esperando una nueva orden:

```
Out[1]: 4
In [2]:
```

Es interesante notar que el resultado nos lo ha mostrado empleando una marca de salida (*echo*) **Out[1]:**. En programación, cuando una orden nos muestra por pantalla el resultado de la operación realizada, se dice que el ordenador ha hecho *echo*. Como veremos más adelante esto no es lo más habitual, en la mayoría de los casos, el ordenador ejecuta la orden recibida y cuando termina nos devuelve el prompt **In [2]:** para indicarnos que ha terminado y está listo para recibir una nueva instrucción. De este modo, podemos emplear Python de modo análogo a como empleamos una calculadora. De hecho, la sintaxis es prácticamente la misma.

In programming slang, it is usual to call the answer the computer shows on the screen an *echo*. As we shall see later on, it is not usual for the computer to show us the result of the operations. Most of the time, it just executes the commands, and when it finishes, it shows us the prompt **In[2]:** again to tell us that it is ready for a new command. In this way, we can use Python as a calculator. The syntax is, indeed, quite similar.

### 1.1.3. Variables

El uso de Python como si fuera una calculadora no tiene demasiado interés. Una vez ejecutada la orden, tras pulsar la tecla *intro* el ordenador no guarda ninguna información sobre la operación realizada. Si queremos realizar un cálculo complejo descomponiéndolo en operaciones más sencillas, deberemos anotar los resultados parciales y volver a copiarlos en el terminal si queremos emplearlos de nuevo. Por supuesto, hace ya mucho tiempo que alguien encontró una buena solución, para este y otros problemas similares: el uso de variables.

Podemos ver una variable como una región de la memoria del computador, donde un programa guarda una determina información: números, letras, etc. Una característica fundamental de una variable es su nombre, ya que permite identificarla. Como nombre para una variable se puede escoger cualquier combinación de letras y números, empezando siempre con una letra, en el caso de Python<sup>2</sup>. Se puede además emplear el signo “\_”. Python distingue entre mayúsculas y minúsculas, por lo que si elegimos como nombres de variable Pino, PINO y PiNo, Python las considerará como variables distintas.

El método más elemental de crear o emplear una variable es asignarle la información para la que se creó. Para hacerlo, se emplea el símbolo de asignación , que coincide con el signo = empleado en matemáticas. Como veremos más adelante la asignación en programación y la igualdad en matemáticas no representan exactamente lo mismo. La manera de asignar directamente información a una variable es escribir el nombre de la variable, a continuación el signo de asignación y, por último, la información asignada, `variable_1 = 18`. Si escribimos dicha expresión en la terminal de Python y pulsamos la tecla *intro*:

```
In [2]: variable_1 = 18
In [3]:
```

---

<sup>2</sup>Como se verá más adelante, Python tiene un conjunto de nombres de instrucciones y comandos ya definidos. Se debe evitar emplear dichos nombres, ya que de hacerlo se puede perder acceso al comando de Python que representan

### 1.1.3. Variables

Nevertheless, handling Python as a calculator is hardly ever enjoyable. Once a command has been executed after pressing the *intro* key, the computer keeps no memory of the operation. If we wish to make a complex computation, splitting it into simpler operations, we should take note of the partial results and copy them again on the console to be used again. Of course, a long time ago, somebody found a good solution for this and other similar problems: using variables.

We can consider a variable as a computer memory region where a program has allocated specific information: numbers, characters, etc. A variable fundamental characteristic is its name because it permits one to identify it univocally. . We can take as a variable name whatever combination of lowercase and uppercase letters and numbers, provided that the first character is always a letter. When using Python<sup>2</sup>, We may also use the symbol “\_”. For Python, uppercase and lowercase letters are different symbols. Thus, if we choose variable names, such as Oack, OACK, and OaCk, they represent different variables for Phyton.

The most straightforward method to create or use a variable is to assign the information we want to contain. To do this, we use the assignment symbol , which coincides with the mathematical symbol =. As will be seen later, programming assignment and mathematical equality are not the same concept. To assign some piece of information to a variable, we write the variable name, then the assignment symbol, and, eventually, the assigned information `variable_1 = 18`. If we write this expression on the console and press the *intro* key, we get:

```
In [2]: variable_1 = 18
In [3]:
```

In this case, the computer doesn't echo the result. Anyway, the variable has been saved in

---

<sup>2</sup>As we shall see later on, Python has a set of command names and keywords already defined. We should avoid using these names or keywords as names for our variables. Otherwise, we could lose access to the corresponding Python command or keyword.

En este caso, Python no hace eco, no nos muestra por pantalla ningún resultado. Sin embargo, la variable ha quedado guardada en la memoria del ordenador. Podemos pedirle a Python que nos la muestre,

```
In [3]: variable_1
Out[3]: 18
```

```
In [4]: print(variable_1)
18
```

In [5]:

En el primer caso, hemos escrito directamente el nombre de la variable en el prompt de IPython. En el segundo, hemos hecho uso de una función de Python `print()`. Mas adelante veremos con detalle las funciones en Python. Por el momento, es suficiente con decir que una función es un objeto de programación que toma una variable de entrada y opera sobre dicha variable y nos devuelve un resultado. La función `print()` toma como variable de entrada, una variable cualquiera y nos imprime en el terminal de Ipython su contenido.

Podemos asignar también a una variable el resultado de una operación aritmética,

```
In [5]: variable_2 = 2 * 5
```

Para saber qué variables tiene guardadas el ordenador en memoria, podemos emplear algunos de los comandos especiales de la consola de Ipython,

In [5]: %whos	Type	Data/Info
Variable		
variable_1	int	18
variable_2	int	10

```
In [6]: %who
variable_1
variable_2
```

En el primer caso, hemos empleado el comando de Ipython `whos`. Delante del comando hemos puesto el carácter %, que sirve para indicar al ordenador que se trata de un comando de la Consola y no de Python. El ordena-

the computer's memory. We may ask Python to show it,

```
In [3]: variable_1
Out[3]: 18
```

```
In [4]: print(variable_1)
18
```

In [5]:

In the first case, we have written the name of our variable straightforwardly after the Ipython prompt. In the second case, we use the Python function `print()`. Later on, we will see the concept of function in Python in more detail. Meanwhile, it is enough to say that a function is a programming object that takes an input variable, operates it and returns a result. In our case, the function `print()` takes a variable whatsoever and prints in the Ipython console the content of the variable.

We can also assign to a variable the result of an arithmetic operation,

```
In [5]: variable_2 = 2 * 5
```

To know which variables are saved in the computer memory, we can use some special Ipython console commands:

In [6]: %who

variable\_1

In the first case, we used the Ipython command `whos`. Notice that we have written the symbol % just before the command. This tells the computer we are introducing a console (Ipython) command, not an ordinary Python

dor nos muestra todas las variables que hemos definido, el tipo de variable y el dato que contienen. En el segundo caso hemos empleado el comando `who`, que nos da simplemente una lista de las variables contenidas en memoria.

Acabamos de mencionar, el concepto de tipo de una variable. En algunos lenguajes, es preciso indicar al ordenador qué tipo de información se guardará en una determinada variable, antes de poder emplearlas. Esto permite manejar la memoria del computador de una manera más eficiente, asignando zonas adecuadas a cada variable, en función del tamaño de la información que guardarán. A este proceso, se le conoce con el nombre de *declaración* de variables. En Python no es necesario declarar las variables antes de emplearlas. El tipo de dato se asigna directamente cuando la variable se crea, de acuerdo con la información asignada. Para saber el tipo de dato que contiene una variable se le aplica la función `type()`. Vamos a ver los tipos de datos estándar o *built-in* de Python.

**Numeric.** Se trata de datos que corresponden a valores o cantidades numéricas. Dentro de los datos numéricos, Python define tres tipos distintos:

*Integer.* Permite definir números enteros positivos y negativos. El tipo se representa mediante la abreviatura `int`. Una característica específica de los enteros en Python es que no tienen limitación de tamaño.

```
In [92]: a = 35
In [93]: type(a)
Out[93]: int
```

*Float.* Permite definir números en coma flotante, es decir, una representación aproximada de un número real. (ver el capítulo ??). Se pueden introducir separando la parte entera de la decimal mediante un punto y también en notación científica. El tipo se representa mediante la abreviatura `float`.

Es interesante notar que, `a = 3` Creará una variable entera mientras que `a=3.` creará una variable real.

command. The computer displays the variables we have defined until this point, the variable Type, and the data contained. In the second case, we have used the command `who`, which shows us the bare list of the variables saved in the computer memory.

We just mentioned the *type* of a variable. In some programming languages, it is necessary to explicitly say the kind of information a variable will store before using it. The sort of information stored defines the type of the variable. This helps to manage the computer memory more efficiently, assigning memory zones according to the variable size. This process is known as variable *declaration*. In Python, it is not necessary to declare variables. Python sets the type to a variable when it is created. To know the data type of a variable, we use the function `type()`. Let's see standard or built-in types in Python.

**Numeric.** Data which represent numerical quantities. Inside the numeric data, Python defines three different types:

*Integer.* It allows for representing whole numbers, positives, and negatives. The type is represented by the abbreviation `int`. A specific feature of integers in Python is that they have no size limitation.

*Float.* Floating point numbers, i.e., approximated representations of real numbers. (See chapter ??). We use a point to split the integer and decimal parts of the number. It is also possible to write a floating point number using scientific notation. This type is represented by the word `float`.

Notice that `a = 3` creates an integer variable, but `a=3.` creates a real variable.

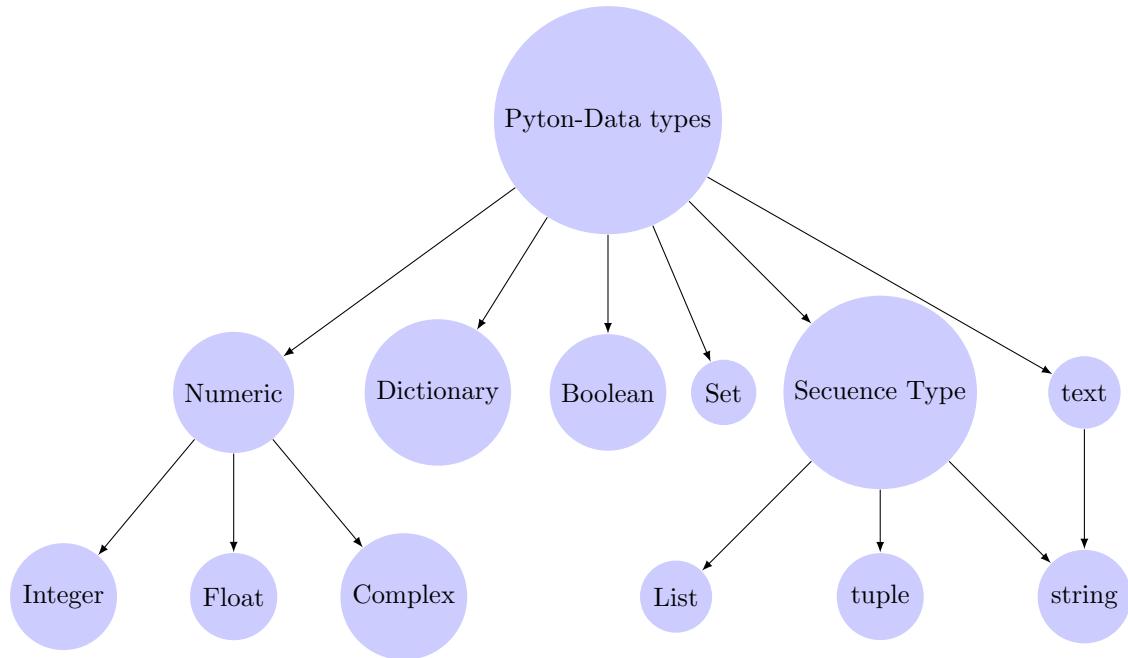


Figura 1.7: Tipos de datos en Python.

Figure 1.7: Data types in Python

```

In [95]: b = -3.5
In [96]: type(b)
In [94]: type(a)
Out[94]: intOut[96]: float
In [98]: c = 3e -4
In [99]: type(c)
Out[99]: float
  
```

*Complex.* Por último, es posible tambíer manejar en Python números complejos. Para ello hay que definirlos como la suma de su parte real más su parte imaginaria que va siempre seguida del simbolo *j*. Aunque introduzcamos las partes reales e imaginarias de un número complejo como enteros, Python siempre los considera números en coma flotante.

Finally, it is possible in Python to deal with complex numbers. To define them, we write the real part of the number, the addition symbol, and the imaginary part, followed by the symbol *j*. Notice that Python considers a complex number's real and imaginary parts as floating point numbers even if you write them as integers.

```

In [100]: d = 2+3j
In [101]: type(d)
Out[101]: complex
In [102]: d.real #parte real
#del número d
Out[102]: 2.0
In [103]: d.imag #parte imaginaria del número d
  
```

**Boolean.** Como veremos más adelante, es muy frecuente en programación necesitar saber si una condición se cumple (es verdadera) o no (es falsa) esto lleva a que en muchos lenguajes de programación, existe un tipo de variable que solo toma dos valores: `True` (verdadero) o `False` (falso).

```
In [1]: D = True
In [2]: type(D)
Out[2]: bool
```

**Sequence Type** Todos los tipos incluidos en Sequence Type, así como los tipos dictionary y set, son en realidad estructuras de datos. Podemos verlos como contenedores, que nos permiten almacenar y manipular varios (muchos) datos distintos empleando una sola variable.

*String.* Es el formato propio para crear variables que contengan texto. Se construyen a partir de caracteres UNICODE, encerrados entre comillas,

```
In [3]: tex1 = 'a'
In [4]: type(tex1)
Out[4]: str
In [5]: tex2 ='perro'
In [6]: type(tex2)
Out[6]: str
```

Una propiedad común a todos los tipos sequence es que son indexables. La indexación es una propiedad muy importante que vamos a emplear habitualmente en programación. La variable `tex2` del ejemplo anterior contiene en realidad una cadena de caracteres; las letras que componen la palabra *perro*. Python nos permite extraer individualmente dichos caracteres empleando para ello un índice correspondiente a la posición que ocupan en la cadena. Para ello, escribimos el nombre de la variable seguido del índice encerrado entre corchetes. Hay que tener en cuenta que al primer elemento de una secuencia le corresponde el índice cero.

As we will see later, it is widespread in programming to check if a condition fulfills (it is true) or not (it is false). This leads to defining a type of variable that can take just two possible values: `True` or `False`.

**Sequence Type** Every kind of variable included in Sequence Type, and also the types dictionary and set, are Data structures. We can see them as containers that allow us to deal with several (many) data using a single variable.

*String* This is the proper type to generate variables that contain text. They are built using UNICODE characters enclosed in brackets.

A common property of all types belonging to Sequence Type is that they are indexable. . The indexation is a fundamental property that we frequently use in programming. The variable `tex2` in the previous example is an array of characters, the letters that compound the work *perro* (Dog in Spanish HA, HA). Using an index, Python, allows us to extract such characters individually. The index represents the position the character we want to extract takes in the word. To extract a character from a variable of type String, we write the variable's name followed by the character's index enclosed in square brackets. It is important to note that Python begins to count the elements of a sequence in zero.

```
In [7]: letra_1 = tex[0]
In [8]: print(letra_1)
p
In [9]: letra_4 = tex[3]
In [10]: print(letra_4)
r
```

Python permite también indexar una secuencia, empezando por el final. En esta caso, al último elemento le corresponde el índice -1, al penúltimo -2, etc,

```
In [40]: tex[-1]
Out[40]: 'o'
In [41]: tex[-3]
Out[41]: 'r'
```

*Listas.* Las listas en python son colecciones ordenadas de datos. Los elementos de una lista pueden ser datos de cualquier tipo y no tienen por qué ser homogeneos. Para crear una lista basta escribir los elementos que la componen entre corchetes y separados por comas,

```
In [11]: L = ['carta', 23, 2.5, 1-2j]
In [12]: L
Out[12]: ['carta', 23, 2.5, (1-2j)]
In [13]: type(L)
Out[13]: list
```

Hemos construido una lista en la que el primer elemento es una cadena, el segundo un entero, el tercero un float y el cuarto un número complejo. Debido a su versatilidad, las listas son un tipo de variable muy empleado en Python. Pdemos copiar elementos de una lista a otra variable empleando índices, y tambien podemos asignar a un elemento de una lista un valor nuevo, por supuesto, perderemos el valor antiguo,

```
In [14]: L3 = L[3]
In [15]: print(L3)
(1-2j)
In [16]: L[2] = 0.
In [17]: print(L)
['carta', 0.0, 2.5, (1-2j)]
```

Dado que una lista no es más que una colección ordenada de variables, podemos anidar listas dentro de listas,

Python also permits indexing a sequence beginning by the end. In this case, the sequence's last element takes the index -1, the next-to-last takes -2, and so on.

Lists are ordered collections of data. The elements of a list can be data of whatever type, and they do not need to be homogeneous. To create a list, write the elements that built it up, enclosed in square brackets and separated by commas.

We had built a list in which the first element is a string, the second an integer, the third a real –float type– number, and the fourth is a complex number. Due to their versatility, the Lists are a very useful type in Python. We can copy elements of a list in another variable using indexes and assign a new value to an element of a list. Of course, in this case, we will lose the old value.

As far as a list is nothing more than an ordered collection of variables, we can nest a list inside another list,

append()	añade un elemento al final de la lista adds a element to the end of a list	L.append(-3)
copy()	Crea una copia de la lista Returns a copy of a list	L2 = L.copy()
clear()	Borra todos los elementos de una lista clears all elements from a list	L.clear
count()	Cuenta el número de elementos de una lista Counts the elements of a list	L.count()
insert()	Inserta un nuevo elemento en una posición específica Inserts an element at a specific position in a list	L.insert(i,-4)
pop()	Extrae el ultimo elemento de una lista. Si se añade un índice extrae el elemnto indicado extract the last element of a list. Addind and index, extract the specific element	a =L.pop() a=L.pop(i)

Tabla 1.1: Algunos métodos de las Listas en Python

Table 1.1: Some methods of Python's List

```
In [14]: LL = [1, 'p', ,L,-0.7]
In [15]: print(LL)
[1, 'p', ['carta', 23, 2.5, (1-2j)], -0.7]
```

Hemos incluido la lista L como el tercer elemento de la nueva lista LL. Podemos extraer un elemento de la lista interior, empleando dos índices; el primero para referirnos a la posición de la lista L dentro de la lista LL, y el segundo para indicar la posición del elemento deseado dentro de la lista L. Así por ejemplo, si queremos obtener la palabra 'carta',

```
In [16]: ctr = LL[2][0]
In [17]: print(ctr)
carta
```

Podemos cambiar datos de una lista, usando el simbolo de asignación,

```
In [18]: LL[3] = 'J'
In [19]: print(LL)
[1, 'p', ['carta', 23, 2.5, (1-2j)], 'J']
```

Una característica importante de las listas, son sus métodos. Los métodos son funciones especiales que nos permiten manipular las listas de una manera directa. La tabla 1.1 muestra algunos de los más usuales. La forma de emplearlos es escribir el nombre de la lista, seguido de un punto el nombre del método y, entre paréntesis los parámetros que toma. Si el método no toma ningún parámetro se deja el

We have included the list L as the third element of a new list LL. We can extract an element from the inner list using two indexes. The first one indicates the position of the list L inside the list LL, and the second one indicates the position of the wanted element inside the list L. For instance, if we want to get the word 'carta',

We can change list data using the assignment symbol,

An essential feature of Lists is their methods. The methods are special functions that allow us to manipulate the lists in a straight way. Table 1.1 shows some of the most used list methods. To use a method, we write the list name followed by a point and name of the method and, enclosed in brackets, the parameters the method takes. If the method doesn't use parameters, we write an empty bracket

paréntesis vacío, pero nunca se omite. Veamos algunos ejemplos:

```
In [12]: L = [1, 'a', -0.5, 'en', 2.5, 'a']
In [13]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [14]: L.append(-2)
In [15]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a', -2]
In [16]: L.pop()
Out[16]: -2
In [17]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [18]: L.insert(3, 'insrt')
In [19]: print(L)
[1, 'a', -0.5, 'insrt', 'en', 2.5, 'a']
In [20]: L.pop(3)
Out[20]: 'insrt'
In [21]: print(L)
[1, 'a', -0.5, 'en', 2.5, 'a']
In [22]: L.count('a')
Out[22]: 2
In [23]: L.count('s')
Out[23]: 0
```

*Tuplas* Las tuplas, son similares a las listas. La principal diferencia es que son inmutable. es decir, una vez que se han creado no se puede modificar el valores de los elementos que contiene. Para crearlas se encierran los elementos que constituyen la tupla entre paréntesis y separados con comas. En el ejemplo que sigue se observa como al intentar modificar un elemento de una tupla una vez creada, Python nos devuelve un aviso de error.

```
In [41]: T = (0, 'int', 2.4, [1, 4, 3.5])
In [42]: T
Out[42]: (0, 'int', 2.4, [1, 4, 3.5])
In [43]: T = (0, 'int', 2.4, [1, 4, 3.5])
In [44]: print(T)
(0, 'int', 2.4, [1, 4, 3.5])
In [45]: type(T)
Out[45]: tuple
In [46]: T[3]
Out[46]: [1, 4, 3.5]
In [47]: T[4] = 12
Traceback (most recent call last):
Cell In[47], line 1
      T[4] = 12
TypeError: 'tuple' object does not support item assignment
```

after the method name but never leave it off.  
Let's see some examples:

**Dictionary.** Los diccionarios son estructuras en las que los datos aparecen asociados a claves. La forma más sencilla de crearlos es mediante pares clave:valor separados por comas y encerrados entre llaves. El símbolo ':' asocia cada elemento con su clave. Un diccionario puede contener dentro cualquier tipo de variable, incluidas listas u otros diccionarios. Para acceder a un elemento guardado en un diccionario se escribe el nombre del diccionario seguido de la clave del elemento deseado escrita entre corchetes.

```
In [63]: D = {'nombre': 'Pepe', 'día': 27, 'mes': 'Febrero', 'datos': [1, 3.5, 6, 0.0]}
In [64]: D
Out[64]: {'nombre': 'Pepe', 'día': 27, 'mes': 'Febrero', 'datos': [1, 3.5, 6, 0.0]}
In [65]: type(D)
Out[65]: dict
```

Para añadir nuevos elementos a un diccionario, se emplea el nombre del diccionario seguido de la clave que tendrá el elemento, escrita entre corchetes y se usa el símbolo de asignación para añadir el valor del elemento. Al igual que sucedía en el caso de las listas, es posibles anidar diccionarios dentro de otros diccionarios. Para acceder al diccionario interno, empleamos su clave. Para acceder a un elemento del diccionario interno empleamos la clave del diccionario interno seguida de la clave del elemento. Para borrar un elemento de un diccionario empleamos la orden de python `del()`. Igual que las listas, los diccionarios cuentan con un buen número de métodos propios. Los interesados pueden consultarlos en las páginas de referencia de Python.

A dictionary is a structure in which data are associated with keys. The simplest way to create them is by using a pair key: data separated by commas and enclosed in curly braces. The symbol ':' associates each element with its key. A dictionary can contain whatever type of variable, including lists and other dictionaries. To get access to data stored in a dictionary, we write the name of the dictionary followed by the key of the element we want to access, enclosed in square brackets.

To add new elements, we write the name of the dictionary followed by the key we want to assign to the element, enclosed in square brackets, and we use the assignment symbol to add the element value. As in the case of lists, it is possible to nest one dictionary into another. To access the inner dictionary, we use its key. To access data inside the inner dictionary, we use the inner dictionary key followed by the data key. (see In[78]: in the example below). To eliminate an element from a dictionary, we use the Python command `del()`. Similar to the Lists, there are quite a few dictionary methods. Readers interested should be addressed to Python reference pages.

```
In [75]: D['nuevo'] = {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}
In [76]: D
Out[76]:
{'nombre': 'Pepe',
 'día': 12,
 'mes': 'Febrero',
 'datos': [1, 3.5, 6, 0.0],
 'nuevo': {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}}
In [77]: D['día']
Out[77]: 12
In [78]: D['nuevo']['num.']}
```

```

Out[78]: 18
In [81]: del(D['día'])
In [82]: D
Out[82]:
{'nombre': 'Pepe',
 'mes': 'Febrero',
 'datos': [1, 3.5, 6, 0.0],
 'nuevo': {'calle': 'Atocha', 'num.': 18, 'Piso': '3D'}}

```

**Set.** Los conjuntos son colecciones datos no repetidos e inmutables. Para crear un conjunto en Python se escriben sus elementos separados por comas y encerrados entre llaves. Si hay elementos repetidos en la definición, el conjunto creado solo los contendrá una vez. No vamos a verlos en más detalle.

```

In [85]: C= {'L', 'M', 'X', 'J', 'V'}
In [86]: C
Out[86]: {'J', 'L', 'M', 'V', 'X'}
In [87]: C= {'L', 'M', 'X', 'J', 'V', 'M'}
In [88]: C
Out[88]: {'J', 'L', 'M', 'V', 'X'}
In [89]: C[1] = 23
Traceback (most recent call last):
Cell In[89], line 1
      C[1] = 23
TypeError: 'set' object does not support item assignment

```

## 1.2. Operaciones aritméticas, relacionales y lógicas.

### 1.2.1. Operaciones aritméticas

Una vez que sabemos como crear variables en Python, vamos a ver como podemos realizar operaciones aritméticas elementales con ellas. La sintaxis es muy sencilla, y podemos sintetizarla de la siguiente manera:

```

resultado = operando1operador1
           operando2operador2operando3 ...
           operadorn-1operandon

```

Es decir basta concatenar los operadores con los operandos y definir una variable en la que guardar el resultado. Por ejemplo,

**Set.** Sets are immutable non-repeated data collections. To make a set in Python we write its elements separated by commas and enclosed in curly braces. If there are repeated elements in its definition, the set created will contain a single instance of the repeated element.

## 1.2. Arithmetical, relational and logical operations

### 1.2.1. Arithmetic operations

Once we know how to create variables in Python, we will see how to perform basic arithmetic operations. The syntax is pretty simple, and we can synthesize it as follows:

```

result = operand1operator1
         operand2operator2operand3 ...
         operatorn-1operandn

```

That is, it is enough to concatenate operands and operators and define a variable that gets the result. For instance,

```
In [1]: a = 3
In [2]: b = 4
In [3]: c = 12.4
In [4]: d = a + b - c
In [5]: print(d)
-5.4
```

En este ejemplo los operandos son las variables `a`, `b`, `c`, los operadores empleados son el símbolo `+` que representa la operación suma y el símbolo `-` que representa la resta. `d` es la variable en la que se guarda el resultado, en este caso, de la suma de las dos primeras variables y su diferencia con la tercera.

Los operadores aritméticos disponibles en Python cubren las operaciones aritméticas habituales. La tabla 1.2 contiene los operadores definidos en python.

In this example, the operands are the variables `a`, `b`, `c`. the operator are the symbol `+` which represents the addition operation and the symbol `-` which represents the subtraction. `d` is the variable that holds the results. In this case, the addition of the two first variables and the result with the third one.

Python available arithmetics operators cover the usual arithmetic operations. Table 1.2 shows the arithmetic operators defined in Python.

Tabla 1.2: Operadores aritméticos definidos en Python  
Table 1.2: Arithmetic operators defined in python

Operación Operation	Símbolo Symbol	Uso Use	notas notes
Suma Addition	<code>+</code>	<code>r=a+b</code>	
Diferencia Subtraction	<code>-</code>	<code>r=a-b</code>	
Producto Product	<code>*</code>	<code>r=a*b</code>	
División division	<code>/</code>	<code>d=a/b</code>	
División entera Integer division	<code>//</code>	<code>d=a//b</code>	Calcula la división entera. Divide y redondea el cociente hacia cero. Calculate the division and round the quotient towards zero
Resto de la división entera Modullus	<code>%</code>	<code>d=a % b</code>	Calcula el resto de la división entera Calculate the remainder after integer division
Potenciación Power	<code>**</code>	<code>y=a ** b</code>	Potencia. Eleva <code>a</code> al exponente <code>b</code> : $a^b$ Raise <code>a</code> to the power of <code>b</code>

```
In [92]: a = 5
In [93]: b = 3
In [94]: print(a+b)
8
In [95]: print(a-b)
2
In [96]: print(a*b)
15
In [97]: a = 5
In [98]: b = 3
In [99]: suma = a+b
In [100]: print(suma)
8
In [101]: dif = a-b
In [102]: print(dif)
2
In [103]: prod = a*b
In [104]: print(prod)
15
In [105]: div = a/b
In [106]: print(div)
1.6666666666666667
In [107]: div_ent = a//b
In [108]: print(div_ent)
1
In [109]: rem = a%b
In [110]: print(rem)
2
In [111]: pot = a**b
In [112]: print(pot)
125
```

### 1.2.2. Precedencia de los operadores aritméticos

Los ejemplos anteriores muestran el uso básico de los siete operadores aritméticos definidos en Python.

Combinando operadores aritméticos, es posible elaborar expresiones complejas. Por ejemplo,

```
In[1]: R=5*3-6/3+2**3+2-4
```

La pregunta que surge inmediatamente es en qué orden realiza Python las operaciones indicadas. Para evitar ambigüedades, Python —como todos los lenguajes de programación— establece un orden de precedencia, que permite saber exactamente en qué orden se realizan

### 1.2.2. Arithmetic operator precedence

The previous examples show a basic use of the seven arithmetic operators defined in Python.

By combining arithmetic operators, it is possible to build up complex expressions. For instance,

```
In[1]: R=5*3-6/3+2**3+2-4
```

A question arises: in which order does Python carry out the operations involved in this expression? To avoid ambiguities, Python—as any other programming language—establishes a precedence order that allows knowing exactly in which order the operations will be carried out. Python precedence order is:

las operaciones. En Python el orden de precedencia es:

1. En primer lugar se calculan las potencias.
2. A continuación los productos y las divisiones, que tienen el mismo grado de precedencia.
3. Por último, se realizan las sumas y las restas.

Por tanto, en el ejemplo que acabamos de mostrar, Python calcularía primero,

$2^{**}3=8$

a continuación el producto y la división

$5*3=15$

$6/3=2$

Por último sumaría todos los resultados intermedios, y guardaría el resultado en la variable R

$15-2+8-4=17$

R=17

**Uso de paréntesis para alterar el orden de precedencia.** Cuando necesitamos escribir una expresión complicada, en muchos casos es necesario alterar el orden de precedencia. Para hacerlo, se emplean paréntesis. Sus reglas de uso son básicamente dos:

- La expresiones entre paréntesis tienen precedencia sobre cualquier otra operación.
- Cuando se emplean paréntesis anidados (unos dentro de otros) los resultados siempre se calculan del paréntesis más interno hacia fuera.

Por ejemplo,

```
In[1]: y=2+4/2
In[2] : print(y)
4
In[3]: y=(2+4)/2
In[4]: print(y)
3
```

1. First, it calculates the powers.
2. Then, products and divisions that share the same precedence degree.
3. Eventually, additions and subtractions are carried out.

Thus, in the example we have just shown, Python would calculate first,

$2^{**}3=8$

then, the product and the division

$5*3=15$

$6/3=2$

eventually, it would sum up all intermedia results and save the final result in the variable R.

$15-2+8-4=17$

R=17

**Using parentheses to modify the precedence order.** When evaluating a complex expression, we need to modify the order of precedence in many cases. To do it, we use parentheses. The rules of use are mainly two:

- Expressions enclosed in parenthesis have precedence over whatever other operation.
- When using nested paratheses (parenthesis enclosed in other parentheses). The results are always obtained from the inner parenthesis to the outer one.

For instance,

```
In[1]: y=2+4/2
In[2] : print(y)
4
In[3]: y=(2+4)/2
In[4]: print(y)
3
```

In the first operation, the precedence order makes Python divide 4 between 2 and then add 2 to the result. In the second case, the Parenthesis has precedence; Python first adds 2 and 4 and then divides the result between 2.

En la primera operación, el orden de precedencia de los operadores hace que Python divida primero 4 entre 2 y a continuación le sume 2. En el segundo caso, el paréntesis tiene precedencia; Python suma primero 2 y 4 y a continuación divide el resultado entre 2.

El uso correcto de los paréntesis para alterar la precedencia de los operadores, permite expresar cualquier operación matemática que deseemos. Por ejemplo calcular la hipotenusa de un triángulo rectángulo a partir de valor de sus catetos,

$$h = (c_1^2 + c_2^2)^{\frac{1}{2}}$$

Que en Python podría expresarse como,

```
In[1]: h=(c1^2+c2^2)**(1/2)
```

O la expresión general para obtener las raíces de una ecuación de segundo grado,

$$x = \frac{-b \pm (b^2 - 4 \cdot a \cdot c)^{\frac{1}{2}}}{2 \cdot a}$$

en este caso es preciso dividir el cálculo en dos expresiones, una para la raíz positiva,

```
In[2]: x=(-b+(b^2-a*c)**(1/2))/(2*a)
```

y otra para la raíz negativa

```
In[3]: x=(-b-(b^2-a*c)**(1/2))/(2*a)
```

Es necesario ser cuidadosos a la hora de construir expresiones que incluyen un cierto número de operaciones. Así, en el ejemplo que acabamos de ver, el paréntesis final `2*a` es necesario; si se omite, Python multiplicará por `a` el resultado de todo lo anterior, en lugar de dividirlo.

### 1.2.3. Operaciones Relacionales y lógicas.

Aunque son distintas, las operaciones relacionales y las lógicas estas estrechamente relacionadas entre sí. Al igual que en el caso de las operaciones aritméticas, en las operaciones relativas y lógicas existen operandos – variables sobre las que se efectúa la operación – y operadores, que indican cuál es la operación

Using parentheses to alter the operator's precedence allows building whatever mathematical expression we wish. For example, to calculate the hypotenuse of a rectangular triangle using the values of its catheti,

$$h = (c_1^2 + c_2^2)^{\frac{1}{2}},$$

we may express this in Python as,

```
In[1]: h=(c1^2+c2^2)**(1/2)
```

Or, as another example, the generic solution of a quadratic equation

$$x = \frac{-b \pm (b^2 - 4 \cdot a \cdot c)^{\frac{1}{2}}}{2 \cdot a},$$

in this case, it is necessary to split the result into two expressions: one for the positive root,

```
In[2]: x=(-b+(b^2-a*c)**(1/2))/(2*a)
```

and the second one for the negative root.

```
In[3]: x=(-b-(b^2-a*c)**(1/2))/(2*a)
```

Caution is needed when building expressions that contain a large number of operations. Take the example just shown: if we forget the last parenthesis `2*a`, Python would multiply by `a` the result of the remaining operation instead of dividing it by `a`.

### 1.2.3. Relational and logical operations.

Although they are not equal, relational and logical operations are strongly related. As we have seen in the case of arithmetic operations, relational and logic operations are built up using operands—variables on which we perform the operations—and operators that indicate which operations we carry out on the

que se efectúa sobre los operandos. La diferencia fundamental es que tanto en el caso de las operaciones relacionales como lógicas el resultado solo puede ser 1 (**True**) o 0 (**False**).

**Operadores relacionales.** La tabla 1.3 muestra los operadores relacionales disponibles en el entorno de Python. Su resultado es siempre la verdad o falsedad de la relación indicada.

Tabla 1.3: Operadores relacionales definidos en Python

Table 1.3: Relational operators defined in Python

operación operation	símbolo symbol	ejemplo example	notas notes
menor que minor than	<	r=a<b	El resultado es <b>True</b> si $a$ es menor que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is minor than $b$ . Otherwise the result is <b>False</b>
mayor que greater than	>	r=a>b	El resultado es <b>True</b> si $a$ es mayor que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is greater than $b$ . Otherwise the result is <b>False</b>
mayor o igual que greater than or equal to	>=	r=a>=b	El resultado es <b>True</b> si $a$ es mayor o igual que $b$ . En otro caso <b>False</b> . The result is <b>True</b> if $a$ is minor than $b$ or equal to $b$ . Otherwise the result is <b>False</b>
menor o igual que Less than or equal to	<=	r=a<=b	El resultado es <b>True</b> si $a$ es menor o igual que $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is greater than $b$ or equal to $b$ . Otherwise the result is <b>False</b>
igual a equal to	==	a==b	El resultado es <b>True</b> si $a$ es igual a $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is equal to $b$ . Otherwise the result is <b>False</b>
Distinto de not equal to	!=	a!=b	El resultado es <b>True</b> si $a$ es distinto de $b$ . En otro caso el resultado es <b>False</b> . The result is <b>True</b> if $a$ is not equal to $b$ . Otherwise the result is <b>False</b>

Es importante señalar que el operador relacional que permite comparar si dos variables son iguales es **==** (doble igual), no confundirlo con el igual simple **=** empleado como sabemos como símbolo de asignación.

**Operadores Lógicos** En Python se distinguen tres conjuntos de operadores lógicos según el tipo de variable sobre la que actúen. Aquí

operand. The main difference with arithmetic operators is that both relational and logic operations would only cast 1 (**True**) or 0 (**False**) as an operation result.

**Relational Operators.** Table 1.3 shows the relational operators available in Python. Their result is always the truth or falsehood of the relationship the operator represents.

It is important to note that the symbol double-equal **==** compares whether two variables are equal. Please do not mistake it for the assignation symbol **=**.

**Logical operator** Python distinguishes three sets of logical operators. We will only present one of them: Logical operators to compare variables.

vamos a ver solo uno de ellos: los operadores lógicos entre variables.

La tabla 1.4 muestra los operadores lógicos entre variables. El resultado, es siempre un (1) **True** o un (0) **False**.

Table 1.4 shows the logical operator defined in Python for variables. The result is always (1) **True** or (0) **False**.

Tabla 1.4: Operadores lógicos entre valores y variables  
Table 1.4: Logical operators on values and variables

operación operation	símbolo symbol	ejemplo example	notas notes
and	and	r=a and b	Operación lógica <i>and</i> entre las variables a y b Logical operation <i>and</i> between variables a and b
or	or	r=a or b	Operación lógica <i>or</i> entre las variables a y b Logical operation <i>or</i> between the variables a and b
negación not	not	r= not a	complemento de a (si a es <b>True</b> entonces <b>not</b> a es <b>False</b> ) a complement (if a is <b>True</b> then <b>not</b> a is <b>False</b> )

En cuanto a su funcionamiento, son los operadores típicos del álgebra de Bool. Así el operador **and** sigue la tabla de verdad propia de la operación *and*, el resultado solo es verdadero (1) si sus operandos son verdaderos (1)<sup>3</sup>,

<sup>3</sup>En realidad, Python considerará verdadero cualquier operando distinto de 0

A logical operator works following the standard of Bool's Algebra. So, the **and** operator follows the true table of the *and* operation in Bool's Algebra, the result is true (1) only if both its operands are true,<sup>3</sup>

<sup>3</sup>In fact, Python takes any variable a true, whenever it is not zero.

Tabla de verdad de la operación **and**  
Truth table for **and** operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	1
1	0	0
0	1	0
0	0	0

el operador **or**, responde a la tabla de verdad del operación booleana *or*, el resultado es verdadero si cualquiera de sus operandos es verdadero o si ambos lo son.

Veamos a continuación como los operadores de lógicos de Python satisfacen las tablas de verdad,

The operator **or** follows the true table of the boolean operation *or*, and the result is true if any one of their operands is true or both of them are true.

Let's see how the logical operator in Python fullfil the truth tables,

Tabla de verdad de la operación `or`Truth table for `or` operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	1
1	0	1
0	1	1
0	0	0

In [1]: `a = 1`In [2]: `b = 0`In [3]: `c = 1`In [4]: `d = 0`In [5]: `a and b`

Out[5]: 0

In [6]: `a and c`

Out[6]: 1

In [7]: `b and d`

Out[7]: 0

In [8]: `a and d`

Out[8]: 0

In [9]: `a or b`

Out[9]: 1

In [10]: `a or c`

Out[10]: 1

In [11]: `a or d`

Out[11]: 1

In [13]: `b or d`

Out[13]: 0

In [14]: `not a`Out[14]: `False`In [15]: `not b`Out[15]: `True`

Es importante destacar que para Python, las operaciones lógicas devuelven valores = 0, si el resultado es cierto ó  $\neq 0$  si es falso, mientras que las operaciones relaciones y el complemento (la negación) devuelven valores lógicos booleanos `True` o `False`.

It is important to remark that logical operations in Python cast a value = 0 if the result is true or  $\neq 0$  if the result is false, while relational operations and the complement (negation) cast boolean values `True` or `false`. That is

cos *True* o *False*. En realidad, esto no supone ningún problema, ya que el lenguaje maneja estos valores lógicos igual que sus equivalentes enteros 1 y 0. Otro aspecto interesante tiene que ver con el hecho de que Python considere como verdadera, cualquier variable que tome un valor distinto de cero. Esto no vas a permitir comentar algunos aspectos del modo en que se llevan a cabo las operaciones lógicas. observa los siguientes ejemplos,

```
In [25]: vdo = 7.5
In [26]: fls = 0
In [27]: vdo2 = 1
In [28]: vdo3 = 18
In [29]: vdo and fls
Out[29]: 0
In [30]: vdo or fls
Out[30]: 7.5
In [31]: vdo and vdo2
Out[31]: 1
In [32]: vdo2 and vdo
Out[32]: 7.5
In [33]: vdo or vdo2
Out[33]: 7.5
In [34]: vdo2 or vod
Out[34]: 1
```

En las líneas In[25] a In[26] hemos creado cuatro variables numérica. Para Python todas ellas son verdaderas excepto `fls`, que vale cero y por tanto es falsa. Si aplicamos el operador `and` a las variables `vdo` y `fls`, el resultado es cero, puesto que una de las variables es cero. Sin embargo, en la línea In[30] observamos que el resultado de la operación `or` realizada es 7.5, que es valor de la primera variable. Este resultado está relacionado con el modo en el que Python realizar las operaciones lógicas. Para una operación `or` es suficiente que uno de los dos operando sea distinto de cero (cierto). El programa comprueba que el primer operando cumpla con la condición, pero una vez que ha comprobado que la primera variable es distinta de cero, ya no necesita comprobar más, sea cual sea el valor de la segunda el resultado será cierto. así que, se límita a devolvernos el valor de la primera variable, distinto de cero y por tanto cierto.

Por contraste, en las líneas In[31] e In[32]

not a problem at all because Python manages these boolean values as their integer counterpart 1 and 0. The fact that Python considers true any value not equal to zero will help us to show how it carries out logical operations, get a lookout to the next examples,

We defined four numeric variables in lines In[25] to In[26]. All of them are true for Python except for `fls`, which is zero and, therefore, false. If we apply the operator `and` to variables `vdo` and `fls`, the result is zero because one of the variables is zero. However, in line In[30], we see that the result of the `or` operation carried out is 7.5, Which is the value of the first variable, i.e., `vdo`. This result is related to Python's way of performing logic operations. For an `or` operation to yield true, it is enough that one of the operands be not equal to zero (true). The program begins trying the first operand, and once it checks that it is true, it is not necessary to test the value of the second operand because the result of the operation will be true, no matter which value the second operand takes. So, once Python is sure the first operand is not equal to zero, it casts its value as the *true* result of the operation.

By contrast, in lines In[31] and In[32], we

se le pide a python que evalue la operación *and* aplicada a las variables, *vdo* y *vdo2*. En este caso, Python tiene necesariamente que comprobar que las dos variables cumplen la condición, puesto que aunque la primera variable sea distinta de cero, el resultado solo es cierto si también la segunda variable es distinta de cero.

Es interesante notar cómo el resultado de la operación cambia al cambiar el orden de los operandos. En los dos casos (In[31] e In[32]) el resultado de la operación es cierto, pero en cada caso devuelve como valor distinto de cero, el valor de la segunda variable, que es la última cuya validez ha comprobado Python. Mira el resto de los ejemplos, y comprueba que siempre se cumple el criterio descrito.

Por último, indicar que los operadores lógicos pueden combinarse entre sí con operadores relacionales y con operadores aritméticos. El orden de precedencia es el siguiente:

1. Paréntesis ()
2. Operadores aritméticos en su orden de precedencia
3. Operadores relacionales, todos tienen el mismo orden de precedencia por lo que se evalúan de izquierda a derecha
4. **and**
5. **or**

Es aconsejable el uso de paréntesis cuando se encadenan varias operaciones lógicas para asegurar su uso correcto y facilitar la lectura de las sentencias.

Por ejemplo,

```
In [3]: a = 12.5
In [4]: b = 3.6
In [5]: c = -3.2
In [6]: d = 0.

In [15]: a>b or d>c
Out[15]: True

In [16]: (a>b) or (d>c)
Out[16]: True
```

ask Python to evaluate the result of the *and* operation applied to *vdo* and *vdo2*. Now, Python has to check that both variables fulfil the condition. Although the first variable is not equal to zero, the result will only be true if the second variable is not zero.

It's worth noting that the result of this operation can change depending on the order of the operators. In both cases (In[31] and [32]), the operation yields true, but Python assigns the value of the second operand as the result because it is the last variable that Python has checked.

Lastly, we can combine logical operators with relational and arithmetic operators. The precedence order is as follows,

1. Parenthesis ()
2. Aritmétical operators in their order of precedence
3. Relational operator, all have the same precedence order; thus, they are evaluated from left to right.
4. **and**
5. **or**

Using parenthesis when several logical operations are linked is good practice to ensure their correct use and to ease the code reading. for instance,

En ambos casos, primero se ejecutan los operadores relacionales, es decir se comprueba si  $a > b$  y se comprueba si  $c > d$ , por último se aplica el operador lógico `or` a los resultados. Pero en el segundo caso es más fácil ver lo que se pretende calcular, gracias al uso de paréntesis.

Veamos otro ejemplo en el que el uso de paréntesis es necesario para obtener el resultado correcto. Se trata de la función lógica *or exclusivo* o *XOR*. Se trata de una operación que compara dos variables lógicas de modo que el resultado solo es verdadero si uno de los operandos es verdadero pero el otro es falso. La tabla de verdad de la operación *XOR* toma la forma,

In both cases, Python first computes the relational operators, i.e., it checks if  $a > b$  and  $c > d$ . Then, it applies the logical operator `or` to the previous results. But, in the second case, it is easier to understand the operation's objective, thanks to the parentheses.

Let's see another example where the parentheses are mandatory: the logical operation *exclusive or* o *XOR*. It is an operation that compares two logical variables and yields true only if one of the operands is true and the other is false. the *XOR* operation true table is as follows,

Tabla de verdad de la operación `xor`  
Truth table for `xor` operation

operando 1 operand 1	operando 2 operand 2	resultado result
1	1	0
1	0	1
0	1	1
0	0	0

Esta operación no está definida como tal en Python pero se puede implementar usando las operaciones `and`, `or` y `not`:

$$\text{XOR} \equiv (\text{a or b}) \text{ and not}(\text{a and b}).$$

Dejamos como un ejercicio comprobar que esta combinación de operaciones cumple con la tabla de verdad de la operación *XOR* y que pasaría que retiramos los paréntesis.

This operation is not defined in Python but can be easily implemented using the logical operations `and`, `or` y `not`:

$$\text{XOR} \equiv (\text{a or b}) \text{ and not}(\text{a and b}).$$

We left it as an exercise to check that such a combination of logical operations fulfils the *XOR* true table and what happens if we remove the parenthesis.

### 1.3. Scripts en python

Hasta ahora, hemos manejado siempre Python desde la línea de comandos. Es decir, hemos introducido las instrucciones de Python en la ventana de comandos. Este modo de emplear el programa es poco eficiente, ya que exige volver a introducir todos los comandos de nuevo cada vez que queremos repetir un cálculo.

Python puede emplear ficheros de texto en los que introducimos un conjunto de comandos, los guardamos, y volvemos a emplearlos

### 1.3. Scripts in Python

So far, we have always dealt with Python using the command line. That is, we have constantly introduced instructions using the command line. This method is rather inefficient because we need to rewrite the commands again anytime we want to repeat the same operations.

With Python, you can effortlessly save a set of instructions in a text file and reuse them anytime you need to carry out the same calculation. This is the usual way to work in

siempre que queramos. Esta es la forma habitual de trabajar no solo de Python, sino de otros muchos entornos de programación. Un fichero que contiene código de Python recibe el nombre genérico de *Script*. Un Script de Python no es más que un fichero de texto que contiene líneas formadas por comandos válidos de Python. Lo habitual es que cada línea contenga un comando. El fichero se guarda con un nombre y la extensión .py. Por ejemplo: `miprograma.py`. El nombre del fichero, puede contener números y letras, y algunos caracteres especiales como la barra baja (\_). En general es no es aconsejable emplear nombres que incluyan espacios en blanco u otros caracteres especiales, ya que es posible que den problemas al intentar ejecutarlos.

### 1.3.1. El editor de textos de Spyder.

Podemos emplear un editor de textos cualquiera, que genere texto en ASCII, como por ejemplo el 'block de notas', para escribir nuestros programas. Sin embargo, si trabajamos en el entorno de Spyder, lo ideal es emplear su propio editor de textos.

El editor de textos de Spyder ocupa el panel izquierdo de la ventana del IDE, (figura 1.6). En la figura 1.8 se muestra una vista más detallada de este panel.

La figura nos muestra un panel con varias pestañas. Cada una corresponde a un archivo de texto que contiene un programa escrito en Python. En la pestaña se indica el nombre del archivo. El archivo visible en la figura muestra el nombre `untitled0.py`. Se trata de un archivo nuevo que acabamos de abrir para escribir un nuevo programa. Para crear un nuevo Script en Spyder basta pulsar el icono situado a la izquierda en la parte superior del panel del editor de texto o bien desplegar el menú *File* situado justo encima y seleccionar la opción *New file*.

Si nos fijamos en la figura 1.8, vemos que el archivo numera en la parte derecha las líneas de código. Además aparecen ya unas líneas escritas,

Las líneas que aparecen escritas por defecto en el archivo de texto, no son comandos

Python and many other programming environments. A Python code file is usually called a *Script*. Thus, a Python Script is a text file that contains lines of valid Python commands. The file is saved using a name and the extension .py. So a valid name would be, for example, `myprogram.py`. The file name may contain numbers, letters, and special characters, such as the underline (\_). Using names that include blanks or other special characters is not advisable as a general criterion because we may have problems running them.

### 1.3.1. The Spyder's text editor

We can use whatever text editor we want to write our programs, provided it generates text in ASCII format. For instance, we can use the classical 'notepad'. Nevertheless, If we are working with Spyder, we should use its text editor.

The Spyder's text editor is located in the left panel of the IDE window (figure 1.6. Figure 1.8 shows a more detailed view.

The figure shows a panel with several tabs. Each one belongs to a different text file containing a Python program. The file name is written in the tab. The file shown in the figure has the name `untitled0.py`. It is a new file we have just opened to write a new program. To create a new Spyder script, it is enough to click the left icon on top of the editor panel or unfold the File menu located just above and select the option *New file*.

Looking at figure 1.8, we see how the code lines are numbered on the right side of the file how the code lines are numbered. Besides, some code lines are included.

These lines, written by default in the text file, are not Python commands; they are comments, i.e. lines that supply information on the file content to the reader. Usually, these lines contain the program's purpose, the type of variables it uses, the date of creation, the author, etc. Python accepts two kinds of

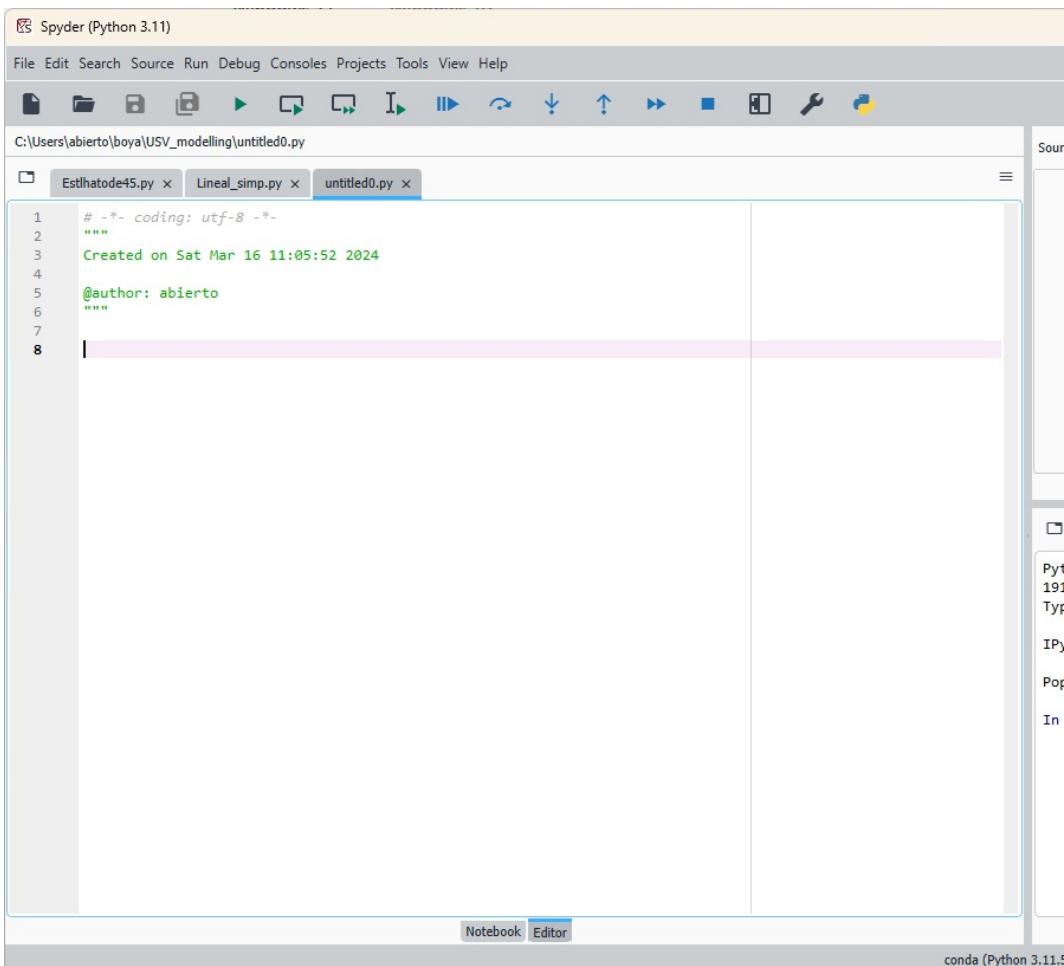


Figura 1.8: Editor de textos para Python incluido en Spyder  
Figure 1.8: Text editor for Python included in Spyder

de Python. Se trata de comentarios. Es decir, líneas en la que damos información a quien lee el fichero sobre su contenido, autor, fecha de creación etc. Python admite dos tipos básicos de comentarios:

- Líneas individuales cuyo primer carácter es una almohadilla `#`. Python interpreta que toda la línea es un comentario y se la salta.
- Grupos de líneas encerrados entre grupos de tres comillas `"""`. Se emplean para comentarios más largos, en los que se dan explicaciones más completas de lo

comments:

- Single comment lines. They start with a hash `#` symbol. Python considers the whole line a comment and skips it.
- A sequence of lines enclosed into groups of three quotation marks `"""`. They are used for lengthier comments, which contain longer descriptions of the code performance.

que hace el código.

---

```
untitled0.py
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar 16 20:05:44 2024
4
5 @author: abierto
6 """
```

---

En nuestro caso, Spyder nos ha escrito un comentario de línea indicando la codificación con la que se guarda el texto del archivo (*utf8*). Después crea un área para un comentario del segundo tipo, (líneas 2 a 6 del código). Nos escribe la fecha en que se ha creado el archivo y el autor. Evidentemente estos valores pueden modificarse. Este espacio para comentarios creado por Python, es un buen sitio para que nosotros añadamos información sobre para qué sirve el fichero de código, el tipo e variables que define, etc.

Para seguir con el mismo ejemplo, vamos a escribir algunas líneas de código y vamos a guardar el fichero con un nombre nuevo. El fichero se guarda, bien pulsando el tercer ícono por la izquierda de los que se muestran encima del panel del editor de texto, o desplegando el menú File y pulsando la opción Save as. En nuestro caso, lo vamos a guardar con el nombre *ejemplo.py*.

In our case, Spyder has written a line comment showing the text codification (*utf8*) used to store the file's information. Below, Spyder creates a comment area –the second type of comment– using the code lines 2 to 6. There, it writes the file creation date and the name of the file author. Of course, this data can be changed by the user. This comment area created by Python is an excellent place to add information about the file features, the type of variables defined, etc.

We will continue with the example, adding some code lines and then save the file with a new name. To save a file, click the third icon from the left in the icon panel above the editor or unfold the File menu and select Save as. We will save our file with the name *ejemplo.py*.

---

```
ejemplo.py
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Mar 16 18:48:46 2024
4 Este Script es un simple ejemplo para mostrar el uso básico
5 de los Scripts de Python
6 This Script is a basic example just to show how to use Python's
7 Scripts
8 @author: abierto
9 """
10
11 #Primero creamos unas variables y mostramos su valor
12 #first we create some variables
13 a = [1,2,3,3,4,0,-8]
14 print(a)
15 b = 12
16 print(b)
17 c = [1,0,1,0]
```

```

18 print(c)
19 #Ahora realizamos algunas operaciones basicas con las variables
20 #Now we perform some basic operation with the variables created
21 a[3] = a[3] + 4.2
22 p = a.pop()
23 d = p+b-c[0]
24 #por ultimo mostramos los resultados de nuestras operaciones
25 #and eventually we show the results
26 print(a)
27 print(p)
28 print(d)

```

---

Si nos fijamos en los cambios introducidos, hemos añadido una líneas al comentario largo, explicando qué contiene el fichero. Además, hemos introducido comentarios de línea, explicando qué hace el código en cada paso. Es fácil ver que el programa crea dos listas **a** y **c** y una variable entera **b**. Además las imprime en el terminal para que veamos el valor que tomas. a continuación, suma 4.5 al valor del cuarto elemento de la lista **a**, extrae el último valor de dicha lista y lo guarda en una variable nueva **p**. En la línea 23 crea una nueva variable **d** sumando **p** y **b** y restando al resultado el último elemento de la lista **c**. Por último nos muestra el valor de las variables **a**, **p** y **d**.

### 1.3.2. Ejecución de scripts

Una vez que hemos terminado de escribir nuestro código, necesitamos ejecutarlo, para obtener los resultados del programa. Dentro del entorno de Spyder, la ejecución de un script se lleva a cabo pulsando el ícono señalado con un triángulo verde, situado en la parte superior del panel del script o bien seleccionando en el menú desplegable *Run* la opción *Run*.

Los resultados de la ejecución aparecen en la consola de Ipython,

```
In [4]: runfile('C:/Users/abierto/Documents/borrar/ejemplo.py',
              wdir='C:/Users/abierto/Documents/borrar')
[1, 2, 3.3, 4, 0, -8]
12
[1, 0, 1, 0]
[1, 2, 3.3, 8.2, 0]
-8
3
```

Focusing on the changes, we added some lines to the long comment explaining the file's contents. Besides, we have introduced some line comments now and then, describing what the code does in each step. It is easy to check that we have created two Lists **a** and **c** and an integer variable **b**. Moreover, python prints these new variables on the console to show their values. Then, it adds 4,5 to the current value of the fourth element of the list **a**, pops out the last values of this list and saves it in a new variable **p**. On line 23, a new variable, **d** is created, adding **p** and **b** and subtracting from the result the last element of the list **c**. Eventually, Python shows the values of variables **a**, **p** and **d**.

### 1.3.2. Running a script

Once we have finished our code edition, we need to run it to obtain the results. Using the Spyder environment, we run a script by clicking the green triangle icon located on top of the editor panel or unfolding the Run menu and selecting the option Run.

The results are shown on the Ipython console,

En realidad, al pulsar el icono de ejecución, lo que hacemos es ejecutar el comando `runfile()`, indicando entre paréntesis el fichero que queremos que se ejecute. Además, la consola de Ipython muestra los resultados de los comandos `print` que hemos definido en nuestro programa. A efectos prácticos, ejecutar el script es equivalente a escribir sus líneas de código una a una en la consola. De hecho, Python ha guardado en memoria las variables que hemos creado, de modo que podemos usarlas si introducimos nuevos comandos en la consola. Por ejemplo, podemos añadir un nuevo elemento al final de la lista `c`,

```
In [7]: c.append(31.2)
In [8]: print(c)
[1, 0, 1, 0, 31.2]
```

In [9]:

## 1.4. Funciones en Python

en la sección anterior hemos visto como emplear scripts para guardar nuestro código y poder reutilizarlo siempre que queramos. Aunque esto supone una mejora respecto a reescribir nuestros comandos de nuevo en la consola, todavía da lugar a código poco flexible y reutilizable. Vamos a introducir en esta sección una estructura de programación que va hacer de nuestro código una herramienta mucho más potente. Se trata de las funciones.

En Python podemos definir funciones directamente en la consola, pero esto no es, en general, una buena idea; cuando cerramos Spyder perdemos el trabajo hecho. Vamos por tanto a utilizar scripts para definir nuestras funciones.

La estructura general de una función en python toma la siguiente forma

```
def nombre_de_la_funcion(e1,e2,...):
    código de la función
    return, r1, r2, ...
```

La primera línea de la definición de una función recibe el nombre de cabecera. La cabecera de la función empieza siempre con la

When we click the icon Run, we execute the command `runfile()`, including, enclosing in parentheses, the name of the file we want to run. Then, Python shows the results of the `print` command that we have included in our program. From the point of view of the results, there is no difference between running the script or writing their lines one by one in the Ipython console. In fact, Python has saved the variables we created running the script in the computer memory, and we can use them if we introduce new commands in the console. For instance, we can add a new element at the end of list `c`,

```
In [7]: c.append(31.2)
In [8]: print(c)
[1, 0, 1, 0, 31.2]
```

In [9]:

## 1.4. Functions in Python

In the previous section, we learned about using scripts to save our code and reuse it later. While this is an improvement over repeatedly rewriting the same commands in the console, it still results in code that is not very flexible and reusable. In this section, we will introduce functions, a programming structure that enables our code to become a more powerful tool.

In Python, it is possible to define functions using the console. However, this is not advisable because all work will be lost once Spyder is exited. takes in Python the following general structure,

```
def funcion_name(e1,e2,...):
    function code
    return, r1, r2, ...
```

$$ax^2 + bx + c = 0$$

The first line of a function definition is known as the function header. It always starts with the keyword `def`. Following this, we write the name of the function. Valid function names follow the same rules as file names. After the function name, we enclose input variables

palabra clave `def`. A continuación se indica el nombre que tendrá la función. Los nombre válidos deben empezar por letras y en general no es buena idea emplear caractéres especiales. A continuación, entre paréntesis y separadas por comas se incluyen las variables de entrada, es decir, aquella variables que la función va a emplear para realizar sus cálculos. Si no necesita tomar variables de entrada, se deja es paréntesis vacío, pero no se omite. Por último se añade el símbolo ':' para indicar que se ha terminado de definir la cabecera de la función.

Deabajo de la cabecera, viene las líneas de código de la función, que constituyen el cuerpo de la misma. Una característica importante de Python es que todas las líneas de código pertenecientes a una función deben estar indentadas, es decir, deben empezar dejando un número igual de espacios en blanco al principio de todas las lineas. Una de las ventajas de emplear el editor de textos de Spyder es que, si definimos una línea de cabecera correctamente, y pulsamos la tecla intro, directamente nos indenta las líneas siguientes. La última linea duna función suele emplearse para devolver variables que contienen los resultados que desamos optener de las operaciones realizadas por la función. Empieza con la palabra clave `return` y va seguida de una lista de variables separadas por comas. Por supuesto dichas variables se tienen que haber definido antes en las líneas de código de la función. A continuación vamos a ver un ejemplo de una función construida para obtener las soluciones de una ecuación de segundo grado,

$$ax^2 + bx + c = 0$$

Cuyas soluciones toman la forma general,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

in parentheses, separated by commas. These input variables are the variables that the function will use to perform its operations. When the function does not need input variables, we leave the parentheses empty but never omit them. Eventually, we write the symbol ':' to end the function header.

Beneath the header file comes the function body, that is, the code lines of the function. One important feature of Python language is that any line belonging to the body function should be indented. The function body lines should begin, leaving the same number of blank spaces from the beginning of the line. One advantage of using the Spyder text editor is that it automatically acknowledges the file header lines and indents the following lines. The last function line is usually used to recover variables that contain the function operations' relevant results. It starts with the keyword `return`, followed by a list of variables separated by commas. These variables should be defined in the function body.

Let's look at an example function for solving quadratic equations.

$$ax^2 + bx + c = 0$$

it has the following general solutions,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

---

eq-seg-grado.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Mar 18 18:33:43 2024
4 Este script contiene una función para obtener la solución de una
5 ecuación de segundo grado.
6 This script defines a function to solve a quadratic equation

```

```

7  #author: abierto
8  """
9  def soleq2(a,b,c):
10    """
11      Esta función calcula las soluciones de la ecuación general,
12       $ax^2 + bx + c = 0$ 
13      Variables de entrada:
14          coeficientes a, b y c
15      Variables de salida:
16          xp Solucion para la raíz cuadrada positiva
17          xm solucion para la raiz cuadrada negativa
18
19      this function calculates the solutions of the general equation
20       $a*x^2 + b*x + c = 0$ 
21      input variables:
22          a,b and c coeficients
23      output variables;
24          xp solution using positive square root
25          xm solutions using negative square root
26      """
27      xp = (-b+(b**2-4*a*c)**(1/2))/2/a
28      xm = (-b-(b**2-4*a*c)**(1/2))/2/a
29      return xp, xm
30
31  #aquí añadimos un ejemplo de uso que no es parte de la función
32  #here we add a 'how to use' example that's not part of the function
33
34  solplus, solminus = soleq2(2, 3, 1)
35
36  #comprobacion
37  #checking
38  print('raíz positiva/positive root:', 2*solplus**2+3*solplus+1)
39  print('raiz negativa/negative root:', 2*solminus**2+3*solminus+1)

```

---

Para definir nuestra función hemos abierto un nuevo script en el editor de texto de Spyder y lo hemos guardado con el nombre `eq_seg_grado.py`. Evidentemente, se puede guardar con cualquier nombre, aunque es buena costumbre tratar de buscar uno que nos de alguna pista de qué contiene el script.

Tras, incluir un breve comentario sobre el contenido del fichero, hemos definido la cabecera de la función en la línea 9. Es interesante fijarse en la definición: el nombre de la función es `soleq2`, y ha continuación, entre paréntesis hemos definido como variables de entrada los tres coeficientes de la ecuación que queremos resolver. Cerramos la cabecera, añadiendo el símbolo 'dos puntos' final. Inmediantamente debajo, de la cabecera, hemos incluido

We have opened a new script in the Spyder text editor to define our function. We have saved the Script with the name `eq_seg_grado.py`. Obviously, you can save the file with whatever valid name you like, but it is a good idea to find a name that gives us some hints in the script content (`quad_eq.p` should be a nice name for this script).

After briefly commenting on the file's content, we define the function header on line number 9. It is interesting to look carefully at this definition: After writing the keyword `def`, we have given a name to the function `soleq2`, and, next, we have defined the input variables, enclosed in parentheses and separated by commas. These input variables will be the coefficients of the quadratic equation

un comentario amplio, describiendo qué hace la función cuáles son sus variables de entrada, y qué variables, con los resultados, nos va a devolver.

La línea 27 contiene las operaciones matemáticas para obtener la solución de la ecuación correspondiente a la solución con raíz cuadrada positiva y guarda el resultado en la variable `xp`, mientras que la línea 28 contiene la solución correspondiente a la raíz cuadrada negativa y guarda el resultado en la variable `xm`. ¡Compruébalo! Por último la línea 29 devuelve las variables `xp, xm` con los resultados calculados.

Aquí termina la definición de la función. Por eso, las líneas siguientes de código, que no forman parte de la función, no están indentadas. Las hemos añadido simplemente para mostrar cómo utilizar la función `soleq2` y comprobar que los resultados son correctos.

En la línea 34 se muestra cómo emplear (llamar) a la función que acabamos de crear. En primer lugar definimos dos variables en los que guardar los resultados `solplus` y `solminus`. Python asociará, por orden, estas variables separadas por comas, a las variables definidas tras la palabra clave `return`. Es decir, copiará el contenido de la variable `xp` en la variable `solplus` y el de la variable `xm` en la variable `solminus`. Evidentemente, podemos elegir cualquier nombre de variable válido para guardar los resultados. A continuación escribimos el símbolo de asignación `=`, seguido del nombre de la función. Por último, damos valores a las variables de entrada `a, b, c`.

Las líneas 38 y 39 comprueban que las dos raíces obtenidas satisfacen la ecuación de segundo grado,  $2x^2 + 3x + 1 = 0$ . Hemos añadido al comando `print`, un texto entre comillas para que nos indique qué resultado de las comprobaciones corresponde a cada solución obtenida.

Si ejecutamos el script que hemos creado, pulsando el icono con el triángulo verde, Obtenemos en la consola de Ipython el siguiente resultado,

```
In [1]: runfile('C:/Users/abrierto/boya/USV_modelling/eq_seg_grado.py',
wdir='C:/Users/abrierto/boya/USV_modelling')
raíz positiva/positive root: 0.0
```

we wish to solve. We close the function header, adding the final colon. Just beneath the header, we have added a rather lengthy comment explaining the purpose of the function and describing its input and output variables.

We have written the mathematical operations for the quadratic equation's positive and negative squared roots in lines 27 and 28, respectively. Check it! In the first case, we have saved the result in variable `xp` and, in the second case, we have saved the result for the negative squared root in variable `xm`. Lastly, line number 29 returns variables `xp, xm` with the operation results.

The function definition finishes here. The remaining code lines do not belong to the function, so they are not indented. We have added them to show how to use the function `soleq2` and check if the results are correct.

In line 34, we show how to use (call) the already created function. First, we define two variables, `solplus` and `solminus` for saving the function results. These variables, separated by commas, are orderly associated with those returned by the function,i.e. the variables included in the function code after the keyword `return`. Thus, Python will copy the content of variable `xp` in variable `solplus` and the content of variable `xm` in variable `solminus`. It is unnecessary to say that we can give whatever valid name we want to the variables we use to save the function results. After the variables above, we write the assignation symbol followed by the function name. lastly, we write values for input variables, `a, b, c`.

Lines 38 and 39 check that both obtained solutions fulfil the quadratic equation  $2x^2 + 3x + 1 = 0$ . We have added a quoted text to the `print` command that helps to determine which result is up to each solution.

If we click on the green triangle icon and run the script we have written, we get the following result in the Ipython console,

```
raiz negativa/negative root: 0.0
```

In [2]:

Es importante recordar que un script no es más que una secuencia de instrucciones que se ejecutan en el terminal. Por tanto, si seguimos la secuencia del script del ejemplo, lo primero que hemos hecho es *crear* la función que hemos definido y guardarla en la memoria del ordenador. Por eso, podemos ahora si queremos seguir utilizándola directamente en el terminal,

```
In [12]: coef = [3,-2,2]
In [13]: sp, sm = soleq2(coef[0],coef[1],coef[2])
In [14]: print(sp)
(0.333333333333334+0.7453559924999299j)
In [15]: print(sm)
(0.333333333333333-0.7453559924999299j)
In [16]: print(solplus)
-0.5
In [17]: print(solminus)
-1.0
```

En este caso, hemos agrupado los coeficientes de la ecuación de segundo grado que queremos resolver en una lista `coef`. Despues, llamamos a la función. El procedimiento es el mismo de antes, definimos unas variables de salida para guardar los resultados `sp`, `sn` escribimos el signo de asignación, el nombre de la función y damos valores a las variables de entrada. Tal y como las hemos escrito, la función identificará el coeficiente `a` con el primer elemento de la lista `coef[0]`, el segundo con coeficiente con el segundo elemento de la lista, etc.

A continuación empleamos el comando `print`, para observar los valores de las variables de salida. En esta caso, la ecuación no tiene soluciones reales, y el resultado son dos números complejos conjugados.

Las dos últimas líneas In[16] e In[17] las hemos empleado para imprimir el valor de las variables `solplus` y `solminus`. Esta variables se crearon cuando se ejecutó el script, pero Python las conserva todavía en memoria.

It is important to note that a script is just a sequence of instructions that run in the console. Thus, if we track the sequence of the example script, we first *create* the function defined in the script and save it in the computer memory. Hence, as long as we don't quit Spyder, we can keep on using it directly in the Ipython terminal,

Notice that we have collected the quadratic equation coefficients in a Python list in this example. Then, we call the function. The procedure is the same as before; we define output variables to save the results `sp`, `sn`, write the assignation symbol and the function name and give values to the input variables. These last have been written so that the function should identify the quadratic equation `a` coefficient with the first element of the list, the second coefficient with the second element of the list, etc.

Next, we use the command `print` to watch the output values of the output variables. In this case, the equation does not have real solutions, and the results are two complex conjugated numbers.

The last two lines, In[16] and In[17], are used to show the value of `solplus` and `solminus`. these variables were created when we ran the script, but Python still holds them in the computer memory.

## 1.5. Los *NameSpace* en Python y el ámbito de las variables

Acabamos de ver que python conserva en la memoria las variables creadas para emplearlas posteriormente. Es importante entender cómo python lleva a cabo la gestión de las variables creadas en memoria. Esto nos lleva al concepto de *namespace*. Por *namespace* entendemos un espacio de memoria, reservado por Python donde los programas y funciones puedes acceder para buscar variables, funciones y, en general, objetos de programación.

En Python se definen cuatro *namespaces* diferentes. Su definición es dinámica, es decir, tienen una duración en tiempo limitada: se crean cuando son necesarios y se destruyen una vez que una vez que dejan de ser necesarios. Habitualmente varios *namespaces* coexisten a la vez. Veamos cuales son:

**Built-in Namespace.** Contiene todas las variables y funciones propias de Python. Dichas variables son accesibles siempre que tenemos abierta una consola de Python. Podemos listar sus nombres escribiendo en la consola de Python el comando `dir(__builtins__)`. Por ejemplo, la función `print`, pertenece a este *namespace*. Podemos acceder a las funciones y variables contenidas en Built-in, desde cualquier función o programa de Python.

**Global namespace.** Podríamos decir que este *namespace* contiene todas las variables y funciones que hemos definido en un script. Sin embargo, para entender mejor a qué nos estamos refiriendo es preciso introducir el conceptos de importación. Volvamos por un momento al ejemplo que vimos más arriba en el que creamos un script que contenía una función `soleq2` que calculaba las soluciones de una ecuación de segundo grado. Hemos visto que si ejecutamos el script, en Spyder la función quedaba en la memoria del ordenador y podíamos emplearla para resolver cualquier ecuación de segundo grado directamente en la consola de Ipython.

## 1.5. Namespace and Scope in Python

[Variable! Namespace] We have seen that Python holds the created variables in the computer memory so that they can be used later. Understanding how Python manages the variables saved in the computer memory is essential. This takes us to the idea of Python *namespaces*. A *namespace* could be considered a memory space reserved by Python, where programs and functions can access to find variables, functions and, in general, programming objects.

Python defines four different *namespaces*. They are dynamically defined, meaning they last for a limited time. They are created when necessary and destroyed when they are no longer needed. Usually, several *namespaces* co-exist. They are:

**Built-in Namespace.** This namespace stores all Python Built-in variables and functions. We can access these variables whenever we have a Python terminal opened. We list their names by writing the command `dir(__builtins__)` on the Ipython terminal. For instance, the function `print` belongs to this *namespace*. We can access every function and variable stored in the Built-in namespace from whatever function or program we write in Python.

**Global Namespace.** We could say that this *namespace* contains all functions and variables defined in a script. Nevertheless, we will introduce the importation concept to understand better what we mean by this. Let's return to the example above, where we wrote a script with a function `soleq2`, allowing us to get a quadratic equation's roots. We have seen that if we run the script in Spyder, the function `soleq2` is stored in the computer memory, and we can use it straightforwardly on the Ipython terminal to solve any quadratic function.

But what if we want to use the function `soleq2` in another script? Do we need to write down the equation again in the new script? The answer is not. Python supplies a method

Pero, ¿Qué pasa si queremos emplear dicha función en otro script? ¿Debemos volver a escribirla de nuevo? La respuesta es no. Para emplear la función `soleq2`, podemos *importar* nuestro primer script en el segundo mediante el comando `import`. Veamos un ejemplo,

---

Example\_import.py

---

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Mar 28 17:25:30 2024
4 Esto es un ejemplo para ver el comando import
5 This is an example on the use of the import command
6 @author: abierto
7 """
8 #importamos el script entero esto no siempre es una buena idea
9 #we import the whole script although not always it is a good idea
10 import eq_seg_grado
11 e = 2
12 f = -4
13 g = 1
14 solplus, solminus = eq_seg_grado.soleq2(e, f, g)
15
16 print('Example_import:', solplus)
17 print('Example_import:', solminus)
18
19 print('eq_seg_grado:', eq_seg_grado.solplus)
20 print('eqseg_grado:', eq_seg_grado.solminus)
```

---

La línea 10 de ejemplo `Example_import.py`, usa el comando `import` para *importar* el contenido del script `eq_seg_grado.py`. Importante: se escribe el nombre del archivo sin la extensión `.py`. Cuando se ejecuta esta línea de código, se ejecuta el script `eq_seg_grado.py` y se crea un namespace global que contiene todas las funciones y variables creadas por dicho script.

En las líneas 11 – 13 definimos tres variables, para usarlas como coeficientes de una ecuación de segundo grado y en la línea 14 llamamos a la función `soleq2` para resolver dicha ecuación. Esta función está contenida en el namespace global de `eq_seg_grado.py`, para acceder a ella escribimos:

`eq_se_grado.soleq2`

Es decir: primero indicamos el namespace y, separado por un punto, el nombre de la función. Esta es la forma en que podemos ac-

to access the functions and variables of one script from another. The method is known as *importation*; we import one script into another using the command `import`. The following script shows an example,

Line number 10 on `Example_import.py` example, uses the command `import` to *import* the content of script `eq_seg_grado.py`. Attention: we write the name of the file without the extension `.py`. When program line number 10 is executed, the script `eq_seg_grado.py` is thoroughly executed, and a global namespace is created to save the variables and functions created by this script.

In lines 11 – 13, we define three variables to be used as coefficients for a quadratic equation. Later, in line 14, we call function `soleq2` to solve the equation. But `soleq2` belongs to the global namespace created when the script `eq_seg_grado` was imported. So, to access it, we write:

`eq_se_grado.soleq2`

We indicate the namespace first and then the function name, separated by a point. This is how we access the variables and functions

ceder a cualquiera de las funciones o variables definidas en un script a través de otro script.

¿por qué definir las variables y funciones en namespaces separados? Una razón es que permite repetir nombres de variables y funciones en distintas partes de nuestro código sin que accidentalmente unos borren a otros. En el ejemplo anterior, en la línea 14 hemos guardo los resultados de las soluciones de la ecuación de segundo grado en las variables `solplus` y `solminus`. Pero estos nombres de variable ya existían. Hemos dicho que al importar `eq_seg_grado`, este script se ejecuta. Si revisamos sú código, vemos que en la línea 34 `solplus, solminus = soleq2(2, 3, 1)`, Se llama a la función `soleq2`, y se emplean exáctamente los mismos nombres de variable para guardar los resultados. Aunque ambos resultados se guardan con los mismos nombres, pertenecen a namespaces distintos. Si ejecutamos el script `Example_import.py` obtenemos en el terminal de Ipython los siguiente resultados,

En términos generales, cuando se importa un script, lo que se crea es una estructura de programación, que en Python se conoce con el nombre de módulo. El módulo toma el nombre del script que lo genera, exceptuando la extensión. Para acceder a las funciones contenidas en el módulo, basta anteponer al nombre de la variable o función el nombre del módulo separado por un punto.

Si descontamos las funciones y variables definidas en el Built-in, todo el resto de código escrito en Python, está pensado para crear módulos. Lo habitual es que estos módulos se agrupen en paquetes de software que engloban funciones y objetos orientados a tareas específicas. A lo largo de estos apuntes, veremos algunos de ellos como `numpy`, orientado al cálculo científico o `matplotlib`, dedicado a la realización de gráficos.

defined in one script from another script.

Why is it convenient to define variables and functions in separated namespaces? One reason is that it allows repeating the same variable and function names in different locations of our code, avoiding them clashing accidentally. In the last example, in line 14, we saved the quadratic equation solutions in variables `solplus` y `solminus`. But these variable names have already been used. We said we were running it when we imported `eq_seg_grado`. If we review the code of this script we can see that in line 34

`solplus, solminus = soleq2(2, 3, 1)`, the program calls `soleq2` and the results are saved in variables with identical names, `solplus` y `solminus`. Although we are saving the solutions in both cases with the same names, they belong to different namespaces. After running the script `Example_import.py`, we achieve the following result,

Generally, a programming structure is created whenever we import a script. In Python, these program structures are called modules. A script and the module generated when the script is imported share the same name (except for the extension). To access the functions and variables held in the module, it is enough to put the name of the module before the name of the function or variable separated with a point.

Except for the functions and variables defined in the Built-in namespace, the remaining code written in Python is intended to create modules. Usually, the modules are grouped into software packages which enclose functions and programming objects devoted to specific tasks. In these lecture notes, we will meet some of them, like `numpy`, oriented to scientific computing or `matplotlib`, devoted to graphical representation.

```
In [1]: runfile('C:/Users/abierto/Documents/borrar/Example_import.py',
              wdir='C:/Users/abierto/Documents/borrar')
raíz positiva/positive root: 0.0
raiz negativa/negative root: 0.0
Example_import: 1.7071067811865475
Example_import: 0.2928932188134524
eq_seg_grado: -0.5
eqseg_grado: -1.0
```

In [2]:

Al importar `eq_seg_grado`, se ejecuta dicho script. Se calcula por tanto la solución de la ecuación de segundo grado:  $x^2+2x+3 = 0$  y se muestran en el terminal las comprobaciones de los resultados. Una vez terminada la importación, todas las variables y funciones creadas por `eq_seg_grado`, están disponibles en su namespace global. despues, observamos en el terminal los resultados de las dos ecuaciones de segundo grado resueltas. Las dos primeras, corresponden a la ecuación  $2x^2+3x+1 = 0$ , y las dos últimas a la ecuación  $x^2+2x+3 = 0$ . Para imprimir estas últimas, hemos empleado `eq_seg_grado.solplus`  
`eq_seg_grado.solminus`,

para indicar que nos referimos a las variables guardadas en el namespace de `eq_seg_grado`.

**Local Namespace.** Este *namespace* es propio de las funciones. Cuando se ejecuta una función en Python, se crea un namespace específico (local) para dicha función en la que se guardan las variables que la función crea y utiliza durante su ejecución. Una vez que la función termina, el namespace local asociado se destruye, y las variables dejan de estar accesibles.

Si nos fijamos en el código de la función `soleq2`, vemos que la función crea unas variables `a,b,c` en su cabecera. Dentro de la función se emplean dichas variables para resolver una ecuación de segundo grado, y se guardan las soluciones en las variables `xp,xm`. Estas cinco variables pertenecen al namespace local de la función `soleq2` y solo el código de dicha función puede acceder a ellas.

Cuando llamamos a la función `soleq2` tanto si lo hacemos desde un módulo como desde la línea de comandos, empleamos el signo de asignación para copiar las variables de salida y damos valores a las variables de entrada. Por ejemplo si nos fijamos en la línea 14 del módulo `Example_import.py`,

```
solplus, solminus = eq_seg_grado.soleq2(e, f, g)
```

As we have already said, this script is run when we import `eq_seg_grado`. Thus, the program calculates first the solution of the quadratic function:  $x^2 + 2x + 3 = 0$  and shows the checking of the solutions on the Ipython terminal. Once the importation is over, all created functions and variables are available in the script's global namespace. Then, we see the results of the two quadratic equations solved on the Ipython terminal. First, the solutions of  $2x^2 + 3x + 1 = 0$  and then the solutions of  $x^2 + 2x + 3 = 0$ . To print these last solutions, we use

```
eq_seg_grado.solplus  
eq_seg_grado.solminus,
```

To indicate that, we want to print the variables held in the global namespace `eq_seg_grado`.

**Local Namespace.** This *namespace* is used by Python functions. Whenever a function is executed, a specific namespace (local) is created for the function. Every variable created by the function is saved in its local namespace, and it is available for the function to use during execution. Once the function ends, its associated local namespace is destroyed, and the variables are no longer accessible.

Focus on function `soleq2` code, we can see how the function creates variables `a,b,c` in its header. In the function body, these variables are used as the coefficients of the quadratic function to be solved. The solutions are saved in variables `xp,xm`. These five variables belong to the local namespace of the function `soleq2`, and only this function code can access them.

When we call function `soleq2`, whether from another module or the command line, we use the assignation symbol to copy the output variables and give values to the input variables. For instance, recall the line number 14 of the `Example_import.py` module,

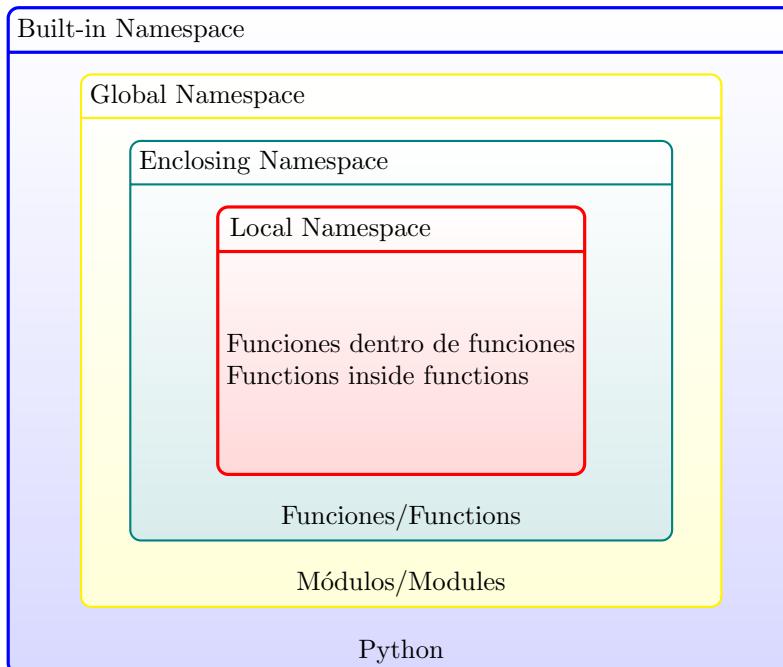


Figura 1.9: Relación de inclusión entre los Namespaces de Python  
 Figure 1.9: Inclusion relationship among Python Namespaces

El contenido de las variables `e,f,g`, que pertenecen al namespace global del módulo `Example_import.py` en qué se llama a la función, se copian en las variables `a,b,c` que pertenecen al namespace local de la función. Cuando se ejecuta el comando `return` el valor de las variables `xp,xm` que pertenecen al namespace local de la función, se copian en las variables

`soplus, solminus`

que pertenecen al namespace global del módulo `Example_import.py`.

**Enclosing namespace.** En ocasiones, llamamos a una función desde dentro de otra función. Por ejemplo nuestra función `soleq2`, llama a la función `print`, que pertenece al namespace Built-in the Python. Cuando se ejecuta la función `soleq2`, ésta crea su propio namespace, y cuando dentro de ésta se ejecuta la función `print` esta última también crea su propio namespace local, dentro del namespace de la función `soleq2`. Se dice entonces que el namespace local de `soleq2` *encierra* al

When the function is called, variables `e,f,g`, which belong to the global namespace of the modules `Example_import.py` are copied in variables `a,b,c`, which belong to the function local namespace. When the function at the end executes the command `return`, the values of variables `xp,xm`, which belong to the local namespace of the function, are copied in variables

`soplus, solminus`

, which belong to the global namespace of module `Example_import.py`.

**Enclosing namespace.** Sometimes, we call a function from inside another function. So, our function `soleq2` calls to función `print`, which belongs to the Python Built-in namespace. When function `soleq2` is executed, it creates its own namespace and when inside it function, `print` is executed this last, also creates its own local namespace inside the namespace of `soleq2`. The local namespace of `soleq2` *encloses* the namespace of `print`. In general, when a fun-

namespace de `print`. En general, cuando una función llama a otra, El namespace de la primera se considera el Enclosing namespace de la segunda. El porqué de esta definición se entiende mejor explicando el concepto de ámbito de una variable.

**Ámbito.** Acabamos de describir el concepto de *namespace* como una región de memoria, asociada a un módulo o a una función. La existencia de distintos namespace, hace que no todas las variables y/o funciones puedan ser manipuladas desde cualquier lado. El ámbito de una variable o función define las funciones, módulos, etc desde lo que es posible acceder y manipular dicha variable o función. Los ámbitos, están fuertemente relacionados con los namespaces y en python se cumple una regla general conocida como la regla "LEGB"(Local-;Enclosing-;Global-;Built-in). Nos da, el orden de precedencia de los namespaces. Así, una función tiene acceso a todas sus variables y funciones locales (Local namespace) a todas las de la función que la ha llamado, si es el caso (Enclosing namespace), a todas las del módulo que ha llamado a la función (Global namespace) a todas las del Built-in namespace. También se cumple lo contrario: desde la línea de comando solo se tiene acceso a las variables que se han creado en la propia línea de comando. Desde un módulo se tiene acceso a las variables y funciones built-in y pero no a las de las funciones contenidas en el módulo. Desde una función se tiene acceso al contenido Built-in y al del módulo y, en su caso, la función, que contiene la función, pero no al de las funciones, contenidas o llamadas en dicha función. La figura 1.9, muestra un esquema con el anidamiento de los namespaces en Python.

**Más sobre el comando `import`.** El comando `import`, es quizás de los más empleados en Python. Más arriba hemos descrito su uso para importar un módulo entero en un script. Sin embargo, es posible emplearlo de otras muchas maneras:

```

1 import eq_sec_grado
2 import eq_sec_grado as sec
3 from eq_sec_grado import *
```

ction calls another, the namespace of the first function is considered the Enclosing namespace of the second function. The reason for this name may be better understood by explaining the concept of variable scope.

**Scope** We have described the concept of *namespace* as a memory area linked to a Python module or function. The namespaces make sure that not all variables or functions can be handled from everywhere. The scope of a variable or a function defines which functions, modules, etc., can manipulate such variable or function. The scopes are tightly related to namespaces. There is the rule known as the "LEGBrule (Local-;Enclosing-;Global-;Built-in)" that establishes the namespace precedence order. So, a function has access to all its local variables and function (Local Namespace), to all ones belonging to the function that calls it, if any (Enclosing namespace), to all ones belonging to the module that calls the function (global namespace) to all builtin namespace variables and functions. It is also true that if we follow the opposite direction from the command line, we can only access built-in variables and variables created from the command line itself. From a module, we have only access to variables and functions built-in or created from the module but not to variables belonging to functions contained in the module. From a function, access is granted to built-in variables, those belonging to the module that contains the function, those belonging to the function that includes the function, if any, and those belonging to the function itself. However, a function has no direct access to variables belonging to functions contained in or called from the function. Figure 1.9 shows a schematic view of Python namespace nesting.

**More on `import` command.** The `import` command is widely used in Python programming. We have already described how to use it to import a whole module. Nevertheless, it is possible to use it in very many different ways,

```
4 from eq_sec_grado import soleq2
5 from eq_sec_grado import soleq2 as cuqui
```

En todos los casos, el módulo desde el que se realiza la importación es el mismo, pero el modo de hacerlo es lo que cambia. Así en la línea 1 del ejemplo importamos el módulo entero, la forma de acceder a las variables y funciones del módulo, es a través de su namespace global: `eq_sec_grado.soleq2`. En la línea 2, también estamos importando el módulo completo, pero ahora, usamos un 'alias', `sec`, para el nombre del namespace asociado. Esto quiere decir que para acceder a las variables y funciones del módulo debemos ahora emplear el alias en lugar del nombre del módulo: `sec.soleq2`. En la línea 3 no importamos propiamente el módulo. Lo que hacemos es importar todas las funciones y variables contenidas el el módulo. De este modo no tenemos ya que anteponerles el nombre del módulo para usarlas. Es muy importante ser muy cuidadoso cuando se hace una importación así. Si en el script que ha importado el módulo hay funciones y/o variables que cuyo nombre coincide con alguno de los importados, tendremos un conflicto de nombres y solo prevalecerá el último creado. En el número 4 solo hemos importado la función `soleq2`. Podemos emplearla en el módulo de destino sin necesidad de hacer referencia a su módulo de origen. Por último, en número 5 del ejemplo hemos importado tan solo la función `soleq2`, pero además le hemos dado un alias. Esto quiere decir que en el modulo de destino deberemos emplearla usando el alias en lugar del nombre original de la función. Es decir, en lugar de utilizar `x1,x2 = soleq2(1,-3,2)` deberemos usar `x1,x2=cuqui(1,-3,2)`.

## 1.6. Depuración

Siempre que escribimos un programa, es preciso comprobar su funcionamiento y, en muchos casos, corregir los errores cometido. El proceso de corrección de código desde su versión original hasta la versión definitiva se conoce con el nombre de depuración de código. Podemos distinguir dos tipos de errores:

The module from which the import takes place is the same in every case, but not how the import is carried out. So, in the line number 1 of the example, we import the module as a whole. The method to access the variables and function of the module is by using its global namespace `eq_sec_grado.soleq2`. In line 2, we are also importing the whole module, but now, we use an 'alias', `sec` instead of the module name. This means that to access variables or functions belonging to the module, we use the alias: `sec.soleq2`. In line 3, we are not importing the module as it is. We are just importing all variables and functions included in the module. In this case, we no longer need to put before the module name to access them. But we should be careful using this importing method. If the script which has imported the module has variables or functions whose names coincide with those of the variables or functions imported, the names will clash. Only the last created ones will be accessible. In name 4, we only imported function `soleq2` from the module. We can use it in the destination script without putting it before the origin module name. lastly, in line 5 we have imported only function `soleq2` but this time we have assigned it an alias. This means we have to use the alias instead of the proper function name when we use it in the destination module. That is: we have to write `x1,x2=cuqui(1,-3,2)`, instead of `x1,x2 = soleq2(1,-3,2)`.

## 1.6. Debugging

Whenever we write a program, we must check its performance; often, we must fix the errors. The process that takes us from the original program version to the final one is known as code debugging. We could differentiate two main groups of errors:

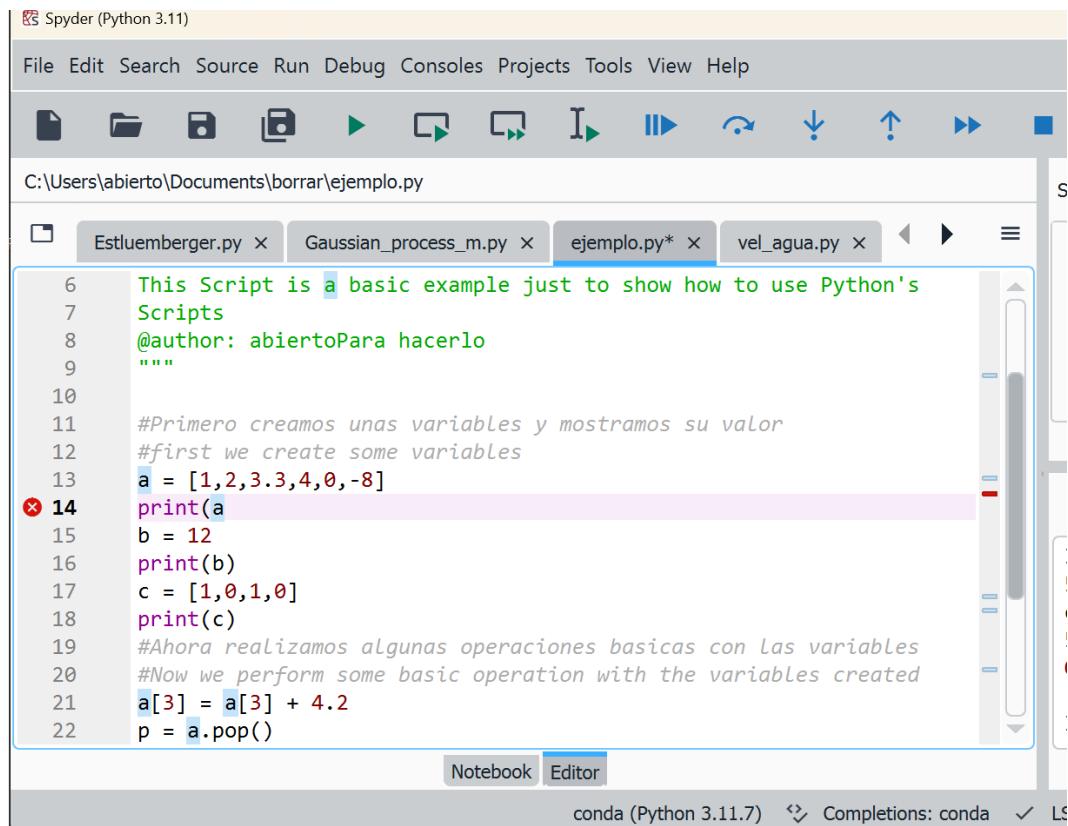


Figura 1.10: Un ejemplo de error de sintaxis (falta cerrar un paréntesis)  
 Figure 1.10: A syntax error example (a closing bracket has been omitted)

**Errores de sintaxis.** Normalmente son errores de escritura. Hemos escrito mal el nombre de una función o un comando o bien no hemos escrito correctamente el código siguiendo las reglas del lenguaje. El editor de Spyder advierte directamente de estos errores. Además, si se intenta ejecutar el código, en la consola de Ipython aparecerá un mensaje de error.

Como ejemplo veamos los errores del script contenido en el editor de textos de la figura 1.10. Si nos fijamos, el editor a dibujado delante de la línea 14 un círculo rojo con un aspa en el centro. Es la manera que tiene de indicarnos que en esas líneas hay errores de sintaxis. Si pasamos el ratón por encima del círculo rojo, obtendremos un mensaje con información sobre el error cometido.

Hay un segundo tipo de errores de sintaxis, que están relacionados con las propiedades de

**Syntax errors.** Broadly speaking, these are *spelling* mistakes. We have written something wrong, like a variable or a function name. Also, we make a syntax error if we do not follow the language rules. The Spyder editor warns us in either case. If we ignore the editor signals and run the program, we will get an error message in the Ipython console.

For example, figure 1.10 shows a script with several syntax errors. The editor has drawn a bold red circle with a cross in the centre before the lines containing an error. If we put the mouse atop the red circles, we get a message with some information on the error made.

There is a second type of syntax error that the editor does not detect. It is related to the properties of variables and functions. An error message will be cast on the Ipython console if we try to run the program. For instance, if we

las funciones y variables que empleamos. Estos errores no son detectados directamente por el editor, pero arrojan tambien un mensaje de error en la consola de Ipython cuando los cometemos. por ejemplo, si escribimos las dos siguientes líneas de código,

```
In [88]: a = 3
In [89]: a.pop()
Traceback (most recent call last):
Cell In[89], line 1
    a.pop()
AttributeError:
'int' object has no attribute 'pop'
In [90]:
```

Obtenemos un error en que se nos dice que no podemos extraer elementos de una variable entera, como es el caso de la variable `a=3`, creada. Efectivamente, solo podemos extraer elementos de una lista.

**Errores de codificación.** Este segundo tipo de errores son mucho más difíciles de detectar. El código se ejecuta sin problemas, pero los resultados no son los esperados. Ante esta situación, no queda más remedio que ir revisando el código, paso a paso para detectar donde está el error. Normalmente, este proceso se lleva a cabo con la ayuda de programas específicos llamados depuradores.

El editor de texto de Spyder nos permite ejecutar un programa paso a paso, ver los valores que van tomando las variables etc, mediante el depurador que lleva incorporado. Para ello, se definen en el editor de Spyder *breakpoints*, esto es: líneas en las cuales Python detendrá la ejecución de un programa, entrará en modo de depuración y esperará instrucciones del usuario. La figura 1.11 muestra el código del ejemplo de la ecuación de segundo grado, en el que se ha definido un *breakpoint*, pulsando con el ratón sobre el espacio situado a continuación del número de la línea en que se desea introducir el *breakpoint*. Spyder indica que el *breakpoint* está activo, dibujado un círculo rojo. Alternativamente, también es posible establecer o remover *breakpoints* empleando el botón *Set/Clear Breakpoint* de la pestaña *Debug*, situada encima del editor de

write the two following code lines,

We get an error message telling us that we cannot extract an element from an integer variable, as is the case of the variable `a=3`. For sure, you can only 'pop' an element from a list.

**Codding errors.** Errors of this kind are usually more challenging to find out than Syntax errors. The code runs correctly, but we do not get the expected results. The only way to find the bug is to review the code line by line. Fortunately, there are specific programs that can carry out this debugging task. They are called, well, yes, debuggers.

Thanks to its built-in debugger, the Spyder text editor allows us to run a program line by line, watching the values the variables are taking, etc. To use the debugger, we need to define in the Spyder editor *breakpoints*, i.e., lines in which Python will stop the program, get into debugging mode, and wait for the user's instructions. Figure 1.11 shows the code of the quadratic equation example, in which we have defined a *breakpoint*, pushing the mouse left button on top of the space after the number of the line at which we want to define a *breakpoint*. Spyder marks the active *breakpoint*, drawing a bold red circle in the space after the line number. It is also possible to set or remove *breakpoints*, using the *Set/Clear Breakpoint* button located in the tab *Debug* on top of the text editor.

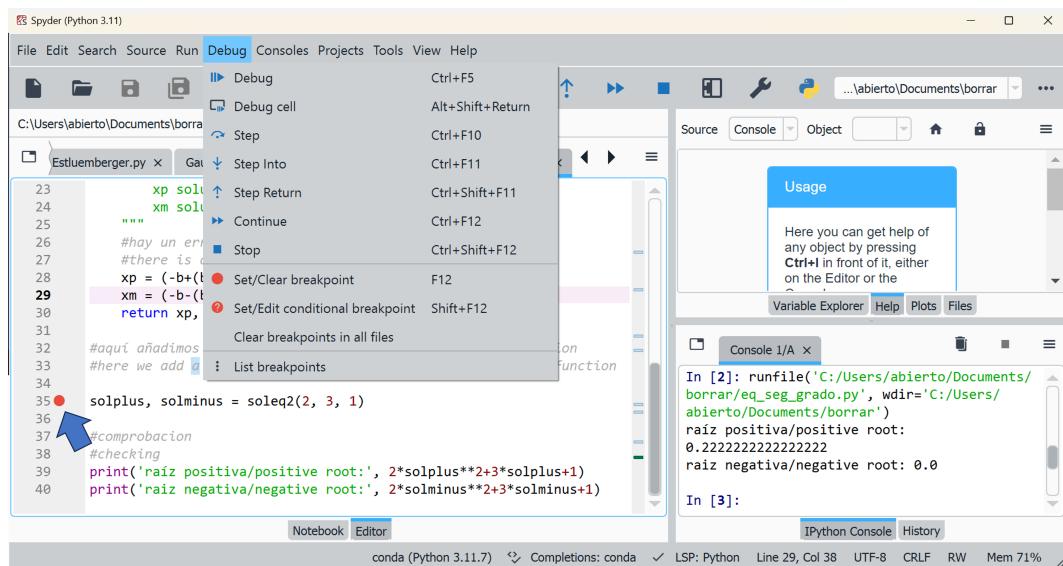


Figura 1.11: Breakpoint activo señalado con una flecha azul, y menú desplegable *debug*  
Figure 1.11: Active Breakpoint pointed by a blue arrow and *debug* drop-down menu

textos.

Si una vez señalado el *breakpoint*, entramos en el modo de depuración, Pulsando el primero de los botones azules en la parte superior del editor de textos (ver figura 1.12), Spyder nos indica que ha detenido la ejecución del programa en la línea marcada por el *breakpoint* (línea 35 en el ejemplo). Además, en el terminal de Ipython nos indica también que ha entrado en el modo de depuración,

Suppose after activating the *breakpoint*, we start the debugging mode by pressing the first of the blue buttons located on top of the text editor (see figure 1.12). In that case, Spyder shows us that it has stopped the program run at the line marked by the *breakpoint* red dot (line 35 in the example). Besides, the Ipython console also shows that it has come into debugging mode.

```
IPdb [1]: !continue
> c:\users\abierto\documents\borrar\eq_seg_grado.py(35)<module>()
 33 #here, we add a 'how to use' example that's not part of the function
 34
2--> 35 solplus, solminus = soleq2(2, 3, 1)
 36
 37 #comprobacion
```

IPdb [2]:

A partir de aquí Spyder pone a nuestra disposición las herramientas de depuración, la figura 1.12 muestra la línea en que se ha parado la ejecución del programa, (señalada con una flecha verde), y algunas de estas herramientas. Se trata del mismo módulo de ejem-

Now, we can access Spyder debugger's tools. Figure 1.12 shows the line where the program execution has been stopped (marked with a green arrow) and some of these tools. It is the same modules used in previous examples, but we have added a small error in line 28 this

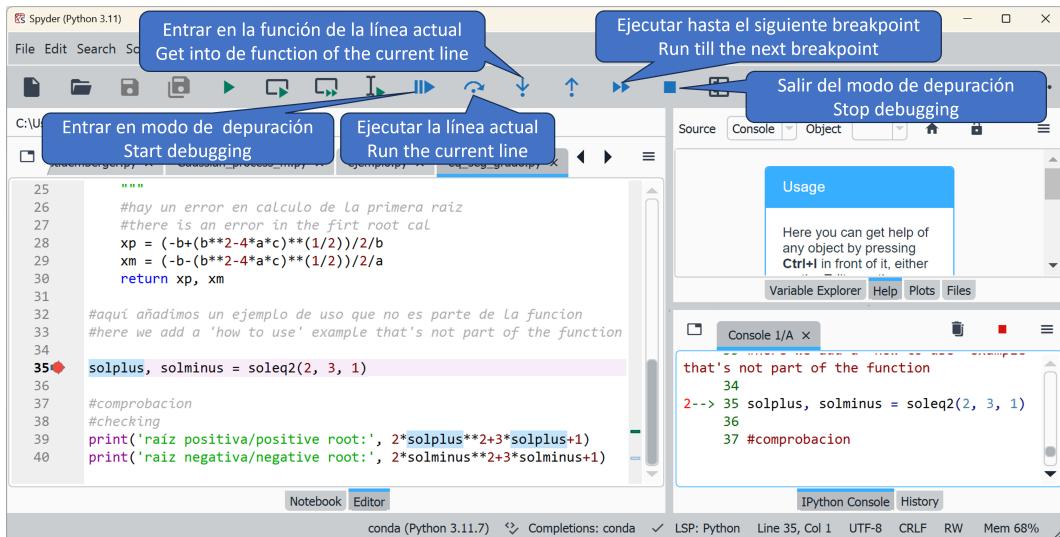


Figura 1.12: Parada de programa en un breakpoint y herramientas de depuración  
Figure 1.12: program stop on a breakpoint and debugging tools

los anteriores, en el que hemos introducido un pequeño error en la línea 28. Básicamente, Spyder nos da la posibilidad de ejecutar el código línea a línea, de entrar e ir línea a línea en las funciones que llama nuestro programa, o de continuar la ejecución hasta el final del programa o hasta el siguiente *breakpoint*.

Si pulsamos el botón “ejecutar línea actual” Spyder ejecutará la línea de programa señalada con la flecha verde y se parará en la línea siguiente. En cada paso, podemos ver el valor que toman las variables, pidiendo su valor directamente en el terminal de Ipython. En el ejemplo de la figura, el breakpoint está situado en la línea 35 que corresponde a la llamada a la función `soleq`. Si volvemos a pulsar el “ejecutar la línea actual”, el depurador ejecutará la función y saltará directamente a la linea 39. Si queremos que recorra paso a paso el código correspondiente a la función, empleamos el el botón “entrar en la función de la línea actual”. El uso del resto de las opciones del depurador el bastante evidente (ver figura 1.12). Si entramos en la función, ejecutamos su código línea a línea y vamos comprobando el resultado que deberían dar, comprobaremos que la línea 28 no da un resultado correcto y estaremos en condiciones de repasar su código

time. Spyder allows us to run the program line by line, get into the function our program calls to and cover it line by line, or continue the program execution till the next breakpoint or to the program end.

If we select the button “Run the current line,” the program will execute the code line marked with the green arrow and stop in the following line. In each step, we can check the values the variables so far defined take by asking for them at the Ipython terminal. In the example shown in the figure, the breakpoint is located in line 35, where the program calls to `soleq` function. If we push the button “Run the current line” again, the debugger will execute the function and jump to line 39. If we want to cover line by line the function `soleq`, then we may push the button “Get into the function of the current line”. The performance of the remaining debugger tools is easy to understand, looking at figure 1.12. If we get into the function `soleq` and execute its code line by line, checking their results, we will realise that there is a bug in line 28 because the result cast by this line is wrong. So we would be ready to review its code and discover the error. Can you find it?

y descubrir el error. ¿Cuál es?.

## 1.7. Control de Flujo

En la sección anterior, se introdujo el modo de escribir programas en Python mediante el uso de *scripts* y funciones. En todos los casos vistos, la ejecución del programa empezaba por la primera línea del programa, y continuaba, por orden, línea tras línea hasta alcanzar el final del programa. Se trata de programas en los que el *flujo* es lineal, porque los resultados de cada línea de programa se van obteniendo regularmente uno detrás de otro.

Hay ocasiones en las que, por diferentes razones que expondremos a continuación, puede interesarnos alterar el orden en que se ejecutan las sentencias de un programa, bien repitiendo una parte de los cálculos un determinado número de veces o bien ejecutando unas partes de código u otras en función de que se satisfagan unas determinadas condiciones.

El control del orden en que se ejecutan las sentencias de un programa es lo que se conoce con el nombre de *control de flujo*. Veremos dos tipos principales de control de flujo: El flujo condicional y los bucles.

### 1.7.1. Flujo condicional.

Empezaremos con un ejemplo sencillo de cómo y para qué condicionar el flujo de un programa. Supongamos que queremos construir un programa que reciba como variable de entrada un número cualquiera y nos muestre un mensaje por pantalla si el número es par.

Para ello, podríamos hacer uso del operador, *resto de la división entera*, %. Si el resto de la división entre dos del número suministrado a la función es cero, se trata de un número par; si no, es un número impar. Podríamos hacer uso de operadores relacionales, en particular de == para comprobar si el resto de la división entre dos es cero. Por último necesitaríamos algún mecanismo que permitiera al programa escribir un mensaje solo cuando el número introducido sea par.

## 1.7. Flow control

In the previous section, we introduced how to write Python programs using scripts and functions. In all the examples shown, the programs ran from the first line and went on, in an orderly manner, line after line, until the program ended. These programs follow a linear *flow* because the results of each line are obtained sequentially, one after another.

Many times, in situations we will describe later, we would like to modify the order in which the program sentences are executed. Sometimes, we need to repeat a block of code several times, while other times, we want a chunk of code to be executed only when a predefined condition is fulfilled.

The control of the order a program follows when executing its sentences is known as *flow control*. We will describe two main kinds of flow control: Conditional flow and loops.

### 1.7.1. Conditional flow.

We begin with a simple example of how and why conditioning the flow. Suppose we want to make a program that gets a number as input and shows us a message on the screen if the number is even.

We can use the operator *modulus*, %. If we divide the number by two and the remainder is zero, the number is even; otherwise, it is odd. We may use relational operators, specifically the == operator, to check if the division by two remainder is zero. Eventually, we need some method which allows the program to write a message only if the input number is even.

**if-elif-else.** The Python structure **if** supplies the method we need. But, let's see first the code of the example we are talking about,

**if - elif - else.** El mecanismo que necesitamos nos lo suministra la estructura `if` de Python. Veamos en primer lugar el código del ejemplo del que venimos hablando,

---

numero\_par.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Fri Apr 19 15:31:19 2024
5 Script para crear una función que comprueba si un número es par
6 Script to create a function tha checks if a number is even
7 @author: juan
8 """
9
10 def par(n):
11     """
12     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
13     indicando que el número es par.
14     This funtion takes an integer number n as input. If n is even then
15     it shows a message indicating the number is even.
16     """
17     if n%2 == 0:
18         print(str(n), ' es un numero par')
19         print(str(n), 'is an even number')
20     return()

```

---

El script genera una función `par(a)`. En la línea 17 aparece definida la entrada a una bloque `if`. Empieza con la palabra clave `if` que va siempre seguida de una expresión que da un resultado lógico: verdadero (1) o falso (0). Esta expresión puede ser cualquier combinación válida de expresiones relacionales o lógicas.

En nuestro caso comparamos el resto de la división del número introducido entre dos con cero. La entrada del bloque constituye una condición, y termina siempre con el símbolo “dos puntos” (:). El cuerpo del bloque mininlinepythonif lo constituyan las líneas posteriores a la línea de entrada, que deben estar sangradas. Estas líneas solo se ejecutan si se cumple la condición de la entrada del bloque `if`. En nuestro caso, el cuerpo lo constituyen las líneas 18 y 19. Por último, la línea 20 corresponde al comando `return()` que está alineado con las líneas de código de la función, no con las de bloque mininlinepythonif. Por tanto, en nuestro ejemplo, si el número introducido no es par, la función se saltará todas

The script generates a function `par(a)`. In line 17, we find the definition of an `if` opening. It starts with the keyword `if` followed by an expression that casts a logic result: true (1) or false (0). This expression could be any valid combination of relational or logical expressions.

In our example, we are comparing the remainder of the division of the input number (`n`) by two with zero. The `if` block opening line always makes up a condition and always ends with a colon (:) symbol. The body of the `if` block comprises the code lines that follow the opening line, which must be indented. The program executes these lines only if the condition defined in the block opening line is met. In our case, the `if` block body comprised lines 18 and 19. Eventually, in line 20, the command `return()` closes the function. Notice how this line is not indented like the lines of the `if` body. It is instead aligned with the lines of the function body because it is not part of the `if` block. Thus, if we introduce an odd number

las líneas del cuerpo del bloque `if` e irá directamente a la línea 20.

En las líneas de código siguientes se muestra un ejemplo del funcionamiento de la función `par`. Para probarla hemos importado directamente la función en el terminal de Ipython, y luego la hemos llamado dándole como variable de entrada primero el número 3 y luego el número 8. En el primer caso no se cumple la condición impuesta, ya que el número no es par, por tanto la función se salta todo el bloque condicional, y ejecutamente directamente el comando `return()`. Como no hemos pedido que devuelva nada, la función no devuelve nada. En el segundo caso, el número es par y por tanto si se cumple la condición necesaria para ejecutar el bloque `if`. Como resultado, la función imprime en el terminal de Ipython el mensaje indicado. Una vez que acaba la ejecución del bloque condicional, ejecuta el comando `return()` igual que en el caso anterior.

```
In [2]: from numero_par import par
In [3]: par(3)
Out[3]: ()
In [4]: par(8)
8  es un numero par
8  is an even number
Out[4]: ()
```

Acabamos de ver la estructura condicional `if` más sencilla posible. Podríamos complicarla un poco pidiendo que también nos saque un mensaje por pantalla cuando el número sea impar. Esto supone incluir en nuestro programa una disyuntiva; si es par el programa debe hacer una cosa y si no, debe hacer otra. Para incluir este tipo de disyuntivas en una estructura `if`, se emplea la palabra clave `else`. Veamos nuestro ejemplo modificado,

into the function, it will skip the lines of the `if` block body and will go straight to line 20.

Below are some code lines that demonstrate the function `par` performance. To test it, we imported the function to the IPython terminal and called it twice - first with the input variable of 3 and then with the input variable of 8. When we passed 3, the condition was not met because the number was odd. Therefore, the function skipped the conditional block and directly executed the `return` command. Since we didn't ask the function to return anything, it returned an empty bracket. On the other hand, when we passed 8, the number was even, and it fulfilled the imposed condition to execute the `if` block. As a result, the function printed the stated message in the IPython terminal. Once the `if` block has been completed, the program executes the command `return`, as in the even number case.

We have seen the simplest conditional `if` structure. We could make it a bit more complex, also asking for a message when the input number is odd. This means we want to include a disjunctive in our program; if the number is even, the program should do one thing; if the number is odd, it should do another one. To include this kind of disjunctive in an `if` structure, we use the keyword `else`. We can see below our example after being modified,

---

numero\_par\_mod.py

---

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Fri Apr 19 15:31:19 2024
5 Script para crear una función que comprueba si un número es par
6 Script to create a function tha checks if a number is even
7 @author: juan
8 """
9
```

```

10 def par(n):
11     """
12     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
13     indicando que el número es par.
14     This function takes an integer number n as input. If n is even then
15     it shows a message indicating the number is even.
16     """
17     if n%2 == 0:
18         print(str(n), ' es un numero par')
19         print(str(n), 'is an even number')
20     return()
21
22 def par2(n):
23     """
24     Esta función toma como entrada un número n y muestra un mensaje
25     indicando si el número es par o impar
26     This function takes an integer number n as input and shows a message
27     stating whether the number is even or odd
28     """
29     if n%2 == 0:
30         print(str(n), ' es un numero par')
31         print(str(n), 'is an even number')
32     else:
33         print(str(n), ' es un numero impar')
34         print(str(n), 'is an odd number')
35     return()

```

---

Hemos añadido a nuestro script una segunda función a la que hemos llamado `par2`. El código es idéntico al de la función `par`, hasta que llegamos a la palabra clave `else`, que marca la disyuntiva; si el número es par, el programa ejecuta las líneas de código entre el `if` y el `else`, si el número no es par ejecutará las líneas situadas debajo del `else` y que aparecen indentadas. En cualquiera de los dos casos, ejecutará la línea 35 que no pertenece al bloque condicional.

Las siguientes líneas de código muestran los resultados de aplicar la función `par2` a los números 3 y 8.

We have added a second function called `par2` to our script. The code is identical to the `par` function until we reach the `else` keyword. At that point, the disjunctive is set. If the input number is even, the program will execute the code lines between the `if` and `else`. Otherwise, it will execute the indented lines beneath the `else`. In both cases, the program will execute line 35, which does not belong to the conditional block.

The following code lines show the result of applying the function `par2` to the numbers 3 and 8.

In [11]: `from numero_par_mod import par2`

```

In [12]: par2(3)
3  es un numero impar
3  is an odd number
Out[12]: ()

```

```
In [13]: par2(8)
8 es un numero par
8 is an even number
Out[13]: ()
```

La estructura `if` admite todavía ampliar el número de posibilidades de elección mediante la palabra clave `elif`. Al igual que con `if`, `elif` va seguido de una expresión lógica que establece una condición, si se cumple se ejecutará el código de las líneas siguientes, si no se cumple, el programa saltará a la siguiente línea que contenga una palabra clave: otro `elif`, un `else` o directamente a la siguiente línea no indentada, saliendo así del bloque condicional. Para ver cómo funciona, vamos a modificar nuestro ejemplo anterior, para que, si el número introducido no es divisible por dos, compruebe si es divisible por tres,

The `if` structure allows to expand the number of possible elections even more, using the keyword `elif`. As in the case of the keyword `if`, `elif` is followed by a logical expression which establishes a condition. The following indented code lines will be executed if the condition is fulfilled. Otherwise, the program will jump to the next line with a keyword: another `elif`, an `else` or straightforwardly to the next non-indented line, exiting the conditional block. We are going to see how it works, modifying our last example. Now we want the program to check when the input number is odd, whether it is divisible by three,

---

numero\_div2\_div3.py

---

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 19 15:31:19 2024
5  Script para crear una función que comprueba si un número es par y si no lo es
6  comprueba si es divisible por tres
7  Script to create a function tha checks if a number is even
8  @author: juan
9  """
10
11
12 def partthree(n):
13     """
14     Esta función toma como entrada un número n y muestra un mensaje
15     indicando si el número es par si no comprueba si es divisible por 3
16     This function takes an integer number n as input and checks if the
17     number is even. If not it checks if the number is divisible by 3
18     """
19     if n%2 == 0:
20         print(str(n), ' es un numero par')
21         print(str(n), 'is an even number')
22     elif n%3 ==0:
23         print(str(n), ' es un numero divisible por tres')
24         print(str(n), 'is a number divisible by three')
25     else:
26         print(str(n), ' es un numero impar pero no divisible por tres')
27         print(str(n), 'is an odd number but it not divisible by three')
28
29 return()
```

---

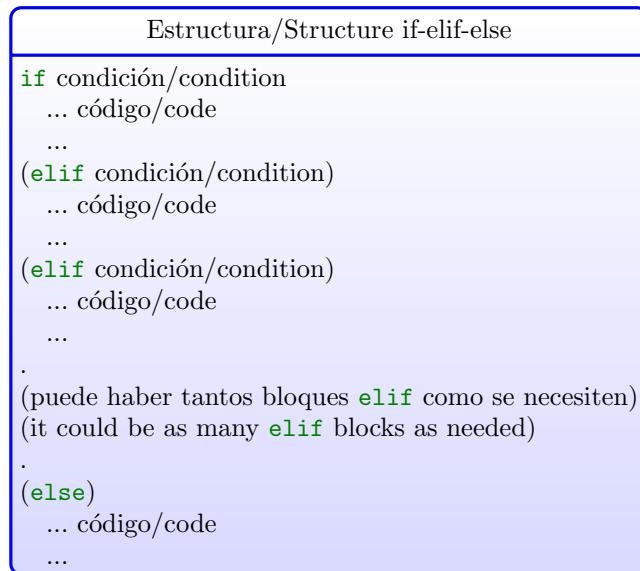


Figura 1.13: Esquema general de la estructura de flujo condicional `if` los términos escritos entre paréntesis son opcionales.

Figure 1.13: General outline of an `if` conditional flow structure. Terms enclosed in parenthesis are optional

Si llamamos ahora a la función dando como valor de entrada un número par, Ejecutará el código situado debajo del `if` y antes del `elif` y se saltará todo lo demás hasta llegar al final del bloque condicional. Si el número introducido no es par, pero es divisible por tres, se saltará el código situado por debajo del `elif`, ejecutará el código contenido debajo del `elif` hasta el `else` y saltará el resto del código hasta llegar al final del bloque condicional. Por último, si el número no es par ni divisible por tres, solo ejecutará el código situado por debajo del `else`.

Un aspecto que debemos resaltar, es que el programa ejecutará el código correspondiente a la primera condición que se cumpla, y se saltará el resto hasta llegar al final del bloque condicional. Así por ejemplo, si en nuestro ejemplo introducimos el número 6, el programa nos mostrará el mensaje “el número es par”, puesto que ésta es la primera condición que se cumple, pero nunca nos mostrará el mensaje “el número es divisible por 3”. Porque una vez comprobada y cumplida la primera condición (ser par) el programa sal-

If we introduce now an even number in the function `parthree`, it executes the code beneath the `if` and before the `elif`. It skips all the remaining code till the end of the conditional block. If the introduced number is odd and divisible by three, the function skips the code beneath the `if` and executes the code between the `elif` and the `else`. It skips the remaining code till the end of the conditional block. Lastly, if the number is odd and is not divisible by three, it only executes the code beneath the `else`.

Note that the program will execute the code belonging to the first condition met and skip the remaining code till the end of the conditional block. For instance, if we introduce the number 6 in our example function, the program will show the message “6 is an even number” because this is the first condition it meets. But it never will show us the message ‘6 is a number divisible by three because, once the program checks that the first condition is fulfilled (be even), the program jumps to the end of the conditional structure without further checking. Figure 1.13 shows a comple-

ta directamente al final de la estructura, sin comprobar nada más. La figura 1.13 muestra el esquema completo de una estructura `if`. Los términos entre paréntesis pueden estar o no presentes en una implementación concreta. Las siguientes líneas de comandos muestran la importación y el uso de la función `partthree` empleando distintos número como entrada,

```
In [18]: from numero_div2_div3 import partthree
In [19]: partthree(4)
4   es un numero par
4   is an even number
Out[19]: ()
In [20]: partthree(6)
6   es un numero par
6   is an even number
Out[20]: ()
In [21]: partthree(9)
9   es un numero divisible por tres
9   is a number divisible by three
Out[21]: ()
In [22]: partthree(7)
7   es un numero impar pero no divisible por tres
7   is an odd number but it not divisible by three
Out[22]: ()
```

**Estructuras if anidadas.** En el ejemplo anterior, hemos visto cómo, si el número introducido en la función era par y además divisible por tres, el programa nunca nos informaría de esta segunda propiedad, debido al carácter excluyente de la estructura `if`. Una manera de resolver este problema, es mediante el uso de estructuras `if` anidadas. La idea es muy sencilla, se construye una estructura `if` para comprobar una determinada condición, si esta se cumple, dentro de su código se construye otra estructura `if` para comprobar una segunda condición, y así sucesivamente, todas las veces que sea necesario. Si modificamos nuestro ejemplo anterior, incluyendo un `if` anidado,

te outline of the `if` structure. Terms enclosed in parentheses are optional and could or may not be included in a specific program implementation. The following commands show the import and use of function `partthree`, using different numbers as inputs.

**Nested if structures** In the previous example, we saw that the program could not detect if the input number was both even and divisible by three due to the exclusive nature of the `if` statement. We can solve this problem using nested `if` structures. It is a straightforward idea. We build an `if` statement to check a specific condition. If the condition is fulfilled, inside the `if` statement, we build another `if` statement to check a second condition, and so on as many times as needed. Notice that in a nested `if` structure, the inner condition is checked only when the outer one is met. We can modify our previous example, using now a nested `if`,

---

numero.parytres\_nested.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 19 15:31:19 2024
5  Script para crear una función que comprueba si un número es par
```

```

6  Si lo es, tiene un if anidado para probar si además es divisible entre tres
7  si no lo es comprueba si es divisible por 3.
8
9  Script to create a function tha checks if a number is even and if it is so,
10 checks if it is divisible by three to. else, the program checks if the number
11 can be divided by 3
12 ©author: juan
13 """
14
15 def partthreenested(n):
16     """
17     Esta función toma como entrada un número n. Si n es par devuelve un mensaje
18     indicando que el número es par.
19     Si no, comp
20     This funtion takes an integer number n as input. If n is even then
21     it shows a message indicating the number is even.
22     """
23     if n%2 == 0:
24
25         if n%3 == 0:
26             print(str(n),' es un numero par y divisible entre 3')
27             print(str(n),'is an even & divisible by 3 number')
28         else:
29             print(str(n),' es un numero par')
30             print(str(n),'is an even number')
31     elif n%3 == 0:
32         print(str(n),' es un numero divisible entre 3')
33         print(str(n),'is an number divisible by 3')
34     else:
35         print(str(n),' es un numero no es divisible entre 2 ni entre 3')
36         print(str(n),'is not divisible by 2 neither by 3 ')
37
38     return()
39

```

---

la modificación realizada está en las líneas 23-25. Se trata de una nueva estructura `if` definida dentro de la que comienza en la línea 20 comprobando si el número es par. El programa solo llegará a comprobar este nuevo `if`, si efectivamente se a cumplido la condición de que el número sea par. Si nos fijamos en la estructura del código, vemos que tiene tres niveles de indentación: El primero corresponde al código perteneciente a la función `partthreenested`, el segundo corresponde al código la estructura `if` más externa y el tercero al código del `if` anidado. Si añadieramos un nuevo `if` dentro de éste último, deberíamos indentar su código un nivel más.

We can see that we have added code lines 23 – 24. This added code defines a new `if` statement defined inside the `if` structure that starts at line 20. The program will only check this new condition if the number is even and it has met the condition imposed in line 20. If we focus on the code structure, we can see that it has three indentation levels: The first one belongs to the code of function `partthreenested`, the second one to the outer `if` structure and the last one to the nested `if`. If we would add a new `if` inside the last one, we should indent the code one level more.

```
In [41]: from numero_parytres_nested import partreenested as piubello

In [42]: piubello(12)
12 es un numero par y divisible entre 3
12 is an even & divisible by 3 number
Out[42]: ()

In [43]: piubello(5)
5 es un numero no es divisible entre 2 ni entre 3
5 is not divisible by 2 nor by 3
Out[43]: ()

In [44]: piubello(4)
4 es un numero par
4 is an even number
Out[44]: ()

In [45]: piubello(9)
9 es un numero divisible entre 3
9 is an number divisible by 3
Out[45]: ()
```

### 1.7.2. Bucles

En ocasiones, es preciso repetir una operación un número determinado de veces o hasta que se cumple una cierta condición. Los lenguajes de alto nivel poseen estructuras específicas, para repetir la ejecución de un trozo de programa las veces que sea necesario. Cada repetición recibe el nombre de iteración. Estas estructuras reciben el nombre genérico de bucles. Vamos a ver dos tipos: los bucles `for` y los bucles `while`.

**Bucles for.** Un bucle `for` repite las sentencias contenidas en el bucle un determinado número de veces, es decir realiza un número fijo de iteraciones. La estructura general de un bucle `for` se muestra en la figura 1.14.

El bucle empieza con la palabra clave `for`, seguida de una variable a la que hemos dado el nombre genérico de índice. Después viene la palabra clave `in`, una lista de valores y el símbolo `:`. La variable índice irá tomando sucesivamente los valores de los elementos contenidos en la [lista de valores]. El código contenido en el bucle, desde la línea siguiente al `for`, debe estar indentado, igual que vimos

### 1.7.2. Loops

Sometimes, it is necessary to repeat an operation a number of times or till a particular condition is fulfilled. High-level program languages have specific structures to repeat a chunk of code as many times as needed. Each repetition is named an iteration. These structures receive the generic name of loops. We will see two types of loops: `for` loops and `while` loops.

**For loops** A `for` loop repeats the sentences inside the loop a predefined number of times. That is, it performs a fixed number of iterations. A general outline of the `for` loop structure is depicted in figure 1.14.

The loop starts with the keyword `for` followed by a variable we have called index. After, we write the keyword `in`, a list of values and a colon `:`. The index (variable) will take all values contained in the [list of values] one after another. The code contained in the `for` loop, from the line just beneath the `for`, should be written using indented lines, in the same way that we seen for the conditional structures. Inside the loop, this code will run as many ti-

Estructura de un bucle for / For loop structure

```
for índice/index in [Lista de valores/Values list]
    ... código /...code
    (condición: / condition: break)
        ... código / ...code
    (condición: / condition: continue)
        ... código / ... code
```

Figura 1.14: Esquema general de la estructura de un bucle `for` los términos escritos entre paréntesis son opcionales.

Figure 1.14: `for` structure General outline. Terms enclosed in parentheses are optional.

en el caso de las estructuras condicionales `if`. Se ejecutará tantas veces como valores tenga la lista de valores. Antes de hablar de las sentencias `break` y `continue`, veamos alguno ejemplos.

mes as items the list of values has. Let's have a look at some examples before describing the `break` and `continue` sentences.

---

— numero\_par.py —

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Mon Apr 29 15:08:34 2024
5 Este modulo contiene algunos ejemplos del uso de los bucles for
6 This module contains some examples of for loops use
7 @author: juan
8 """
9
10 def sumalist(A):
11     """
12     Input A should be a list of numbers no mater how long
13     the function shows the values on the screen, sum them and save the result
14     in the returned variable y
15
16     La entrada A debe ser una lista de números the cualquier longitud.
17     La función muestra los valore de la lista, los suma y devuelve el resultado
18     en la variable y
19     """
20     y = 0 #definimos la variable en la que guardaremos el resultado de la suma
21         #define the variable to save the result of the sum
22     for i in A:
23         print(i)
24         y = y + i
25     return y
26
27 def sumaelementos(A,B):
28     """
29     Inputs A and B shoudd be two list of numbers with the same length
30     The funtion calculates the sum of the elements of A and B that share the
```

```

31     same location inside their respective list i.e. The result is save
32     in list S in the same location: S[i] = A[i]+B[i]
33 """
34
35     if len(A) != len(B):
36         print('Las listas que me das no son igual de largas')
37         print('Sling your hook! The lists lengths are different. ')
38         return([]) #we return an empty list
39     else:
40         S = []
41         index = range(len(A)) #index to cover the elements of A&B
42         for j in index:
43             S.append(A[j] + B[j]) #adding the elements in the same location
44
45     return(S)
46
47 def firsttime(A,b):
48 """
49     esta función busca la primera vez que un número b aparece en una lista
50     A y devuelve su posición dentro de la lista.
51     Si no aparece muestra un mensaje por pantalla
52     This program find the first time a number b appears in a list A
53 """
54     p = 0
55     for i in A:
56         if i==b:
57             break
58         p = p + 1 #p += 1
59     if p == len(A):
60         print('el número', b, 'no está en la lista')
61         print('number', b, 'is not included in the list')
62     else:
63         print('el número', b, 'ocupa en la lista la posición', p )
64         print('number', b, 'is located at position', p, 'in the list' )
65     return(p)
66
67 def buscapar(A):
68 """
69     Esta función busca los numeros pares de una lista de numeros y
70     crea una nueva lista solo con los pares contenidos en la lista inicial
71 """
72     B = [] #and empty list to save the even numbers
73     for i in A:
74         if i%2 !=0:
75             continue
76         B.append(i)
77     return B
78
79
80 def listadelistas(A,b):
81 """
82     Esta función busca las veces que aparece un número en una lista

```

```

83     bidimensional (una lista de listas de números)
84     Devuleve las veces que aparece el número y los indices de las posisiciones
85     en que aparece.
86     """
87     ind = [] #an empty list to save the list indexes where the number is.
88     counter = 0 #number of times the number is repeated in the list
89     for i in range(len(A)):
90         for j in range(len(A[i])):
91             if A[i][j] == b:
92                 ind.append((i,j))
93                 counter = counter+1
94
95     return(ind,counter)
96
97 def pairs(A,B):
98     """
99     A and B are two list of numbers of the same lenght. The function build a list
100    of pairs, taken one element from and one element from b with the condition that
101    they should be different. It doesn't the mater the order i.e (1,2) and (2,1)
102    are different a valid'
103
104     """
105    pares = [] #an empty list
106    for i in A:
107        for j in B:
108            if i != j:
109                pares.append((i,j))
110
111    return(pares)

```

---

La función `sumalist`, toma como variable de entrada una lista de números de cualquier longitud y devuelve como salida la suma de todos los números contenidos en la lista. Además, imprime en el terminal de Ipython cada uno de los números. El bucle `for` comienza en la línea 22. La primera vez que el código llega a dicha línea asigna a la variable `i` el primer elemento de la lista `A`. En la línea 23 muestra por pantalla el valor de dicho elemento. En la línea 24 suma su contenido a la variable `y` y vuelve de nuevo a la línea 22 y asigna a la variable `i` el valor del segundo elemento de `A`. El proceso se repite hasta que se han recorrido todos los elementos de la lista. Es entonces cuando el programa termina el bucle y ejecuta la linea 25, devolviendo el valor de la variable `y`, en la que se ha acumulado la suma de los elementos de la lista.

La función `sumaelementos`, toma como variables de entrada dos listas de igual longitud

The function `sumalist` takes a whatever length list of numbers as an input and returns the sum of the number in the list as an output. Besides, it writes each number of the list in the Ipython console. The `for` starts in line 22. The first time the program arrives at this line, it assigns to variable `i` the first element of the list `A`. In line 23, the program shows the value of this element on the screen. In line 24, it adds this value to variable `y`, and the program goes back to line 22 and assigns to variable `i` the second value of the list `A`. This process repeats until the elements of the list `A` have been exhausted. Then, the program finishes the `for` loop and runs line 25 returning the value of variable `y`, which will have accumulated the sum of list `A` elements.

The function `sumaelementos` takes two lists of equal length as input variables and returns a list with the sums of the elements located at the same position on the input lists. A con-

y devuelve como variable de salida una lista cuyos elementos contienen las sumas de los elementos de las listas de entrada que ocupan la misma posición. El código está protegido por una estructura condicional, de modo que si las lista de entrada no tienen la misma longitud el programa muestra un mensaje por pantalla y devuelve una lista vacía. Si las listas tienen la misma longitud, en la línea 41 se crea la variable `index` empleando el comando `range`. Por tanto, `index` sera una lista que contendrá los valores enteros desde 0 hasta la *longitud de las listas de entrada* – 1 . En la línea 42 se inicia el bucle `for` sobre dicha lista `i in index`. En la línea 43 se suman los elementos de las listas de entrada que ocupan la posición indicada por el valor de `j` y se añade el resultado a la lista `S`. Cuando el bucle ha recorrido y sumado todos los elementos de las listas de entrada, el bucle termina y se devuelve el valor de `S`. Esta manera de crear una lista de índices para recorrer los elementos de una estructura de datos se usa con mucha frecuencia en programación.

Volvamos, antes de seguir con los ejemplos, a la estructura de la figura (1.14). En ella aparecen dos palabras clave opcionales que permiten interrumpir o alterar la ejecución de un bucle `for`. Ambas, deben siempre incluirse dentro de una estructura condicional. La palabra clave `break`, interrumpe la ejecución del bucle `for` que la contiene y sigue adelante con la ejecución del resto de las líneas de código que haya por debajo del bucle. La palabra clave `continue` salta todas las líneas de código que tenga por debajo, dentro del bucle y empieza una nueva iteración. Para ver su funcionamiento volvamos al programa de ejemplo anterior.

La función `firsttime` busca la primera vez que un número `b` aparece en una lista `A`, devuelve la posición que ocupa en la lista y termina la ejecución. El bucle `for` comienza en la línea 55. La estructura `if`, contenida en el bucle `for`, comprueba si el elemento correspondiente de la lista coincide con el valor buscado. Si es así, `p`, ejecuta la instrucción `break` y termina la ejecución del bucle. Si no coincide con el número, incrementa el valor de la variable `p` en 1, y vuelve al principio del bu-

ditalional structure protects the code so that if both input lists are not the same length, the program casts a message on the screen and returns an empty list. If both lists have the same size, in line 41, the program creates the variable `index` using the command `range`. Therefore, `index` will be a list of numbers from 0 to *length of the input lists* – 1. In line 42, the `for` loop stars, using this list `i in index`. In line 43, the elements of the input lists, located at the position indicated by the value of `j`, are summed, and the result is appended to the list `S`. When the loop has covered all the elements of the input lists, the loop is over, and the program returns the value of the list `S`. It is interesting to note the method for covering the elements of a data structure using the `range` command because, in programming, it is a very commonly used method.

Before proceeding with the code examples, we will return to figure 1.14. There are two keywords included in the general `for` structure layout that allows to interrupt or alter the execution of a `for` block. Both should always be included in a conditional statement. The keyword `break` interrupts the execution of the `for` loop that encloses it, and the program jumps to the following line after the `for` block. The keyword `continue` skips every code line beneath it, belonging to the `for` block and starts a new iteration. We will see how they work, resuming the study of the example code included above.

The function `firsttime` searches for the first time that a number `b` appears in a list `A`, returns the position the number is located at in the list and finishes the program execution. The `for` loop starts at line 55. The `if` structure, enclosed in the `for` loop, checks if the element `i` of the list meets the wanted number. if so, the program executes the `break` command, and the program returns the position `p` of the number in the list. If the element `i` of the list `A` does not match the wanted number, the program increments the value of variable `p` in 1 and goes back to the beginning of the `for` loop and will try the next element of the list. If the wanted number is in the list, the final value of `p` will be less than the list length, and the Ipython console will display a mes-

cle. Si el número está en la lista, el valor de `p` será menor que la longitud de la lista y por, tanto se mostrará en el terminal de Ipython un mensaje indicando que el número está en la lista y qué posición ocupa. Si el número no está en la lista, el bucle `for` no se interrumpe y el valor de `p` coincidirá con la longitud de la lista, mostrando entonces en el terminal de Ipython un mensaje indicando que el número no está en la lista<sup>4</sup>.

La función `buscapar` busca los números pares de una lista y crea una nueva lista con los números pares contenidos en la lista original. En la línea 71 se comprueba si el número es impar y, caso de serlo, la instrucción `continue` hace que el programa no ejecute la línea 73, evitando que los números impares se incluyan en la lista `B`, que solo contendrá números pares al final de la ejecución del programa.

**bucles for anidados.** Los bucles `for` pueden anidarse unos dentro de otros de modo análogo a como se hace con las estructuras `if`. Las funciones `listadelistas` y `pairs` muestran ejemplos de su uso.

`listadelistas` toma una lista de dos dimensiones —una lista `A` formada por listas de números— y busca cuantas veces aparece un número en ella. Para ello emplea dos bucles `for` anidados. En la línea 86, empieza el primero de los bucles, creando un índice `i` que recorrerá la lista. En la línea 87 comienza un segundo bucle creando un segundo índice `j`. Es segundo índice recorrerá cada una de las lista de números incluidas en la lista `A`. Este bucle `for` interior contiene una estructura condicional que comprueba si el elemento correspondiente, al elemento que ocupa la posición `j` dentro de la lista de números `i`, coincide con el valor deseado. Si coincide, los índices de número se guardan como una dupla en la lista `ind` y se incrementa en una unidad el contador `counter`, que lleva la cuenta de las veces que ha aparecido el número. Veamos un ejemplo de cómo funciona,

---

<sup>4</sup>Hay maneras mejores de hacer esto mismo en Python. El ejemplo solo pretende mostrar el uso del comando `break`

sage indicating that the number is in the list and the position the number has in the list. If the program can not find the number in the list, the `for` loop is not interrupted, and the Ipython console will display a message indicating that the number is not in the list<sup>4</sup>.

The function `buscapar` searches for the even numbers included in a list and builds a new list with the even numbers contained in the original list. The program checks in line 71 if the number is odd. If this is the case, the `continue` command makes the program skip line 73, avoiding including the odd numbers in list `B`, which will only have even numbers when the program finishes.

**Nested for loops.** The `for` loops can be nested one into another, likewise to how we nested `if` structures. Functions `listadelistas` and `pairs` show a couple of examples of how to do it.

The function `listadelistas` takes a bidimensional list – a list `A` in which each element is, in turn, a list of numbers– as input and searches how many times a number also supplied to the function as an input, is enclosed in the list. It performs this task using two nested `for` loops. The first (outer) loop starts at line 86, defining an index `i`, which will cover the list. A second (inner) `for` loop starts at line 87, creating a second index `j`. This second index will cover each list of numbers included in list `A`. The inner `for` loop has a conditional structure inside that checks if the number that takes position `j` inside the number list `i` meets the wanted value. If so, the index `(i,j)` are saved as a tuple in list `ind` and the variable `counter`, which counts the times the wanted number has been found in the list, is increased in one unit. The following code presents an example of the function performance,

---

<sup>4</sup>there are better ways to do this. This example is intended only to show the performance of the `break` statement.

```
In [12]: LoL =[[1],[1,2,5],[2,-1,5,3,5],[4,3,2,0,5],[2,1]]  
In [13]: from ejemplo_for import listadelistas  
In [14]: indice, contador = listadelistas(LoL,5)  
In [15]: indice  
Out[15]: [(1, 2), (2, 2), (2, 4), (3, 4)]  
In [16]: contador  
Out[16]: 4
```

La función nos muestra las veces que aparece el número 4 en la lista LoL. Dejamos al sufrido lector que trate de averiguar como funciona la función `pairs`.

**List comprehension.** Python cuenta con excelentes recursos para realizar operaciones de modo iterativo sobre diversos tipos de estructuras de datos. Entre ellos, vamos a ver una conocida con el nombre de *List comprehension*. La idea es crear una lista, en la que los elementos se obtienen como el resultado de iteraciones en otra u otras listas. Su estructura general toma la forma:

The function shows us the times that number 4 is included in the list LoL. We leave you to try to discover how the function `pairs` works. (if you have followed the reading till now, you should find it out with your hands tied behind your backs).

**List comprehension.** Python has excellent resources for iterating over different data structures. We will not present you many of them, but just one very useful and widely used by Python programmers: *List comprehension*. The idea is to build a list whose elements result from iterations on several other lists. Its general structure is,

---

```
newlist = [expression for item in iterable if condition == True]
```

La nueva lista se forma a partir de una operación (expresión) que se realiza sobre cada elemento (item) de otra estructura de datos (iterable). Además se puede imponer una condición sobre cada elemento para que se lleva a cabo la operación. El siguiente script de Python, contiene un par de ejemplos,

The new list is built from an operation (expression) carried out on any element (item) of another (iterable) data structure. Moreover, it is possible to add a condition for the operation to be carried out. The following Python script contains a couple of examples.

---

#### List\_comprehension\_example.py

---

```
1 #!/usr/bin/env python3  
2 # -*- coding: utf-8 -*-  
3 """  
4 Created on Mon May 13 08:59:21 2024  
5 Un par de funciones usando list comprehension  
6 A pair of function using list comprehension  
7 @author: juan  
8 """  
9  
10  
11 def lc_demo(A):  
12     """  
13     Esta función busca los numeros pares en una lista A y los guarda en
```

```

14     una nueva lista B
15     This function finds the odd number in list A, a created a new list
16     B with them
17     """
18     B = [i for i in A if i%2 != 0]
19     return B
20
21 def pair_cmh(A,B):
22     """
23     A and B are two list of numbers of the same lenght. The function build a list
24     of pairs, taken one element from and one element from b with the condition that
25     they should be different. It doesn't the mater the order i.e (1,2) and (2,1)
26     are different a valid'. But, but we will use a list comprehension
27
28     """
29     pares = [(i,j) for i in A for j in B if i != j]
30     return(pares)

```

---

La función `lc_demo`, Crea una lista con los números impares contendidos en otra lista. Es fácil ver la estructura de la *list comprehension*, creada en la línea 18: la lista B, está formada por los elementos `i` incluidos en la lista A, que cumplan `i%2 != 0`, es decir, que no sean divisibles por dos y, por tanto, impares.

La segunda función, `pair_cmh`, es un poco más compleja, pero deja ver las posibilidades del método. La función crea una lista de tuplas, para ello emplea dos estructuras `for`, tal y como están escritas, es como si fueran dos `for` anidados. Así, la función compara cada elemento `i` de la primera lista con todos los elementos `j` de la segunda y, si son distintos `i != j`, guarda los dos números como una tupla `(i,j)` en la lista `pares`.

**Bucle while.** Este bucle tiene la misma finalidad que un bucle `for`, repetir un trozo de código un determinado número de veces. Lo que cambia, es el mecanismo que determina cuantas iteraciones realizará el bucle. En el caso de un bucle `while` las iteraciones se repiten un número indefinido de veces mientras se cumpla una determinada condición impuesta al principio del bucle. La figura 1.15 muestra la estructura general de un bucle `while`. Como sucede en todas las estructuras de programación que hemos visto en Python, es preciso que el código correspondiente al bucle `while` esté indentado. El siguiente script muestra algunos

The function `lc_demo` makes a list with the odd numbers contained in another list. It is easy to understand the structure of the *comprehension list* created in line 18: List B is built up by the elements `i`, included in list A which fulfil `i%2 != 0`, that is, those elements that are not divisible by two and, thus, odd numbers.

The second function, `pair_cmh`, is slightly more complex, but it allows us to appreciate the possibilities of the method. The function creates a list of tuples. To do so, it uses a pair of `for` structures, written in a way that is equivalent to two nested `for` statements. Hence, the function compares each element, `i`, from the first list with any element, `j`, from the second one. If the elements are different `i != j`, it saves the elements as a tuple `(a,b)` in the list `pares`.

**While loop** This kind of loop has the same purpose as the `for` loop, to repeat a chunk of code several times. The main difference is the mechanism determining the number of iterations the loop will take. For a `while` loop, the iterations are repeated an undefined number of times while a condition imposed at the beginning of the loop is fulfilled. Figure 1.15 shows an outline of the `while` loop structure. As in every Python programming structure we have been so far, the code of the `while` loop should be indented. The following script

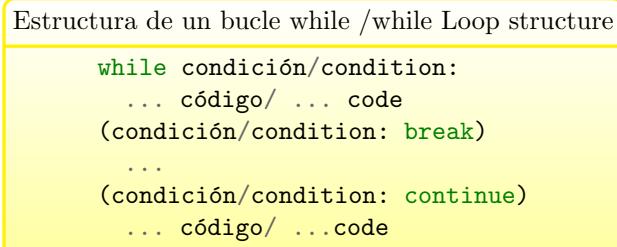


Figura 1.15: Esquema general de la estructura de un bucle while los términos escritos entre paréntesis son opcionales.

Figure 1.15: General structure for a while loop. Terms enclosed in parentheses are optional.

ejemplos de uso,

shows some examples of how to use it.

---

examples\_while.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon May 13 10:46:35 2024
5  Un par de ejemplos de uso de bucles while
6  A pair of loop while examples
7  @author: juan
8  """
9
10 def pmax(a,M):
11     """
12     This funtion optain the minimum exponent n such that a**n >= M
13     esta funci-on optiene el minimo esponte n tal que a**n >= M
14     """
15     n = 0
16     while a**n < M:
17         n = n+1
18     return n
19
20 def L_suma(A,B):
21     """
22     This function take tho list of list of numbers A and B and adds the numbers
23     located at the same position in both list. (both lists should have the
24     same dimensions)
25     Esta función toma dos listas A y B y crea una nueva lista con la suma de los
26     números que ocupan la misma posición (ambas listas deben tener las mismas
27     dimensiones)
28     """
29
30     i = 0 #indice de para recorrer las listas de números
31             #numbers to cover the list of number list
32     C = []
33     n_lista = len(A)
34     while i < n_lista:
35         n_num = len(A[i])

```

```

36     j = 0 #indice para recorrer los numeros de cada lista de numeros
37         #index to cover the number of each number list
38     C.append([])
39     while j < n_num:
40         C[i].append(A[i][j]+B[i][j])
41         j = j+1
42         i = i+1
43     return C

```

---

La función `pmax`, toma como entradas dos números y calcula a qué exponente es necesario elevar el primero de los números para que el resultado sea mayor que el segundo. El código del bucle `while` comienza en la línea 16 comprobando si el número elevado al valor de `n` es menor que el valor de la segunda variable de entrada `M`. Si efectivamente se cumple esta condición, se ejecuta la línea 17 que lo único que hace es incrementar en una unidad el valor de `n`. A continuación el programa vuelve de nuevo a la línea 16 y comprueba si se cumple la condición para el nuevo valor de `n`. Este proceso continúa hasta que la condición deja de cumplirse. En ese momento, el programa sale del bucle `while` y ejecuta la instrucción `return`, devolviendo el último valor calculado de `n`.

Un aspecto muy importante del bucle `while` es que al programarlo hay que asegurarse de que dentro del bucle existe la posibilidad de cambiar la condición de entrada. Si no, el programa no podrá terminar nunca el bucle<sup>5</sup>. Las sentencias `break` y `continue` son idénticas a las descritas en el caso de los bucles `for`, por lo que no insistiremos más sobre el asunto.

**Bucles while anidados.** Del mismo modo que se anidan los bucles `for`, es posible anidar bucles `while`. En el código de ejemplo anterior se ha definido una función, `L_suma`, Que emplea dos bucles `while` anidados para sumar los valores de dos listas de listas de números que ocupan la misma posición. Las listas deben tener las mismas dimensiones, mismo número de listas de números y mismo número de valores en cada lista de número. Lo primero

<sup>5</sup>Cuando se produce esta situación por un error en el diseño del programa, el bucle se puede parar pulsando a la vez las teclas `ctrl+C`

The function `pmax` takes two numbers as input and calculates the exponent you have to raise the first number to obtain a result greater than the second one. The `while` loop code starts at line 16, checking whether the first input variable raises to `n` is less than the second input variable. When this condition is fulfilled, the program executes line 17 that increases in one unit the value of `n`. Then, the program returns to line 16 and checks if the condition is met for a new value of `n`. This process goes on until the condition is no longer met. At this moment, the program exits the `while` loop and executes the `return` instruction, returning the last value of `n` the program calculated.

A critical point to consider when programming a `while` is ensuring that the input to the loop condition will change inside the loop. Otherwise, the program will get into an infinite loop and never stop<sup>5</sup>. The commands `break` and `continue` are identical to those described for the `for` loops. For this reason, we will not insist on them any more.

**Nested while loops.** In a similar way as with the `for` loops, it is possible to nest `while` loops. We have defined the function `L_suma`, in the above code example, which uses two nested `while` loops to add the values of two lists located at the same position in each list. The lists should have the same dimension, the same number of lists of numbers and the same number of numbers in each list of numbers. We first define an index to cover the lists of numbers contained in the input lists `A` and `B`. Then, a new empty list `C` is created. An outer

<sup>5</sup>If you get into this situation for an error in the program design, please don't panic: press the board keys `Control + C` simultaneously and breathe normally.

que se hace en definir un índice, para ir recorriendo cada una de las listas de números contenidas en las listas de entrada A y B. Despues se crea una nueva lista C, vacía. En la línea 34 se empieza el bucle `while` exterior. Este bucle se ejecutará mientras le índice `i` sea menor que el número de listas de números contenidas en las listas de entrada. Una vez que se entra en dicho bucle, lo primero que se hace es comprobar cuantos numeros hay en la lista `i` y se añade una nueva lista (de números) vacía a la lista `C`. El programa entra en el `while` interior, que va recorriendo los valores de las listas de números `i` contenidas en las listas de entrada, los suma y los añade a la lista `C`. Cada vez que termina con una de las listas de números, `j = n_num`, El programa sale del bucle interior, incrementa en una unidad el valor de `i`, cuenta cuantos números hay en la nueva lista y añade una nueva lista vacía a `C` y vuelve a entrar en el bucle interior. El proceso continúa hasta que se agotan las listas de números, `i = n_lista`; cuando se cumple esta condición, el programa sale del bucle `while` exterior y devuelve el valor de `C`.

### 1.7.3. Funciones recursivas.

Una función recursiva es una función que se llama a sí misma. Hemos esperado hasta aquí para hablar de ellas porque, de alguna manera, se comportan como un bucle y necesitan una condición de salida para dejar de llamarse a sí mismas y terminar su ejecución. En general, son delicadas de manejar y tienden a consumir mucha memoria ya que cada vez que la función se llama a sí misma necesita crear un nuevo espacio de memoria independiente. Veamos un ejemplo de función recurrente que permite obtener el término enésimo de la sucesión de Fibonacci.

$$f_0 = 0, f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5, f_6 = 8, \dots f_i = f_{i-1} + f_{i-2} \dots$$

La sucesión empieza con los términos 0 y 1 y a partir de ahí cada término es la suma de los dos anteriores. Podemos convertir directamente esta definición en código,

`while` loop starts at line 34. This loop will run while the index `i` is less than the number of list of numbers contained in the input lists. Once the program gets into this outer loop, the program checks first of all how many numbers the list `i` contains and appends a new empty list (of numbers) to list `C`. The program then gets into the inner `while` loop, which covers the values of the lists of numbers `i` contained in the input lists, adds these values between then and appends the result to the list `C`. Any time a list of numbers is completed, `j = n_num`, the program jumps off the inner loop, increases in one unit the value of `i`, checks how many numbers are in the new list of numbers, adds a new empty list to `C` and get again into the inner loop. The process goes on till the lists of numbers are exhausted, `i = n_lista`; when this condition is met, the program gets out the outer `while` loop and returns the value of `C`.

### 1.7.3. recursive functions.

A recursive function is a function that calls itself. We have waited till now to speak of them because, in some ways, they behave like a loop and need an output condition to stop their execution. Speaking in general, they are difficult to deal with and tend to be very memory-consuming because any time a function calls itself creates a new memory independent memory area. We are going to see an example of a recursive function which allows for obtaining the Fibonacci series  $n$  term.

The series starts with the terms 0 and 1, and later on, each term is the sum of the previous two. We can straightforwardly cast this definition into code,

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Mon May 13 12:38:14 2024
5
6  @author: juan
7  """
8
9  def fibonacci(n):
10    """
11    Parameters
12    -----
13    n : TYPE integer
14        DESCRIPTION. n is the fibonacci series term whose value we want to
15        calculate
16        n: es el término de la serie de fibonacci cuyo valor queremos calcular
17
18    Returns
19    -----
20    s : TYPE integer
21        DESCRIPTION the value of fibonacci series term n
22
23    """
24    if n < 2:
25        #the value of the term is n itself
26        s = n
27    else:
28        s = fibonacci(n-1)+fibonacci(n-2)
29    return(s)
30
31
32 def fibo_series(n):
33    """
34    Parameters
35    -----
36    n : TYPE integer
37        DESCRIPTION. Last number of the series whe want to build
38
39    Returns
40    -----
41    Fs: List with the n first fibonacci series terms.
42
43    """
44    Fs = [fibonacci(i) for i in range(n)]
45    return(Fs)

```

---

Si  $n$  es menor que dos, la función `fibonacci` da como valor el término correspondiente (1 o 0). Si  $n$  es mayor que dos, vuelve a llamarse a sí misma con entrada  $n-1$  y  $n-2$ , para calcular

if  $n$  is less than or equal to two, the function `fibonacci` returns the corresponding term (1 or 0). If  $n$  is greater than two, the function calls itself with inputs  $n-1$  and  $n-2$ , to cal-

el valor enésimo de la sucesión a partir de la suma de los dos anteriores, la función se irá llamando a sí misma hasta llegar a  $n < 2$ . A partir de ahí irá devolviendo los valores obtenidos en cada llamada hasta obtener el enésimo término.

La segunda función, `fibo_series`, emplea una *list comprehension* para crear una lista con los  $n$  primeros términos de la serie de Fibonacci<sup>6</sup>.

#### 1.7.4. Algoritmos y diagramas de flujo.

Un algoritmo es, en la definición de la Real Academia:

Un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

La palabra algoritmo ha llegado hasta nosotros transcrita del nombre del matemático árabe *Al-Juarismi* (circa 780-850 dc). En programación los algoritmos son importantes, porque suponen un paso previo a la creación de un programa.

Habitualmente, partimos de un problema para el que tenemos un enunciado concreto. Por ejemplo: obtener los  $n$  primeros números primos.

El siguiente paso, sería pensar y definir un algoritmo que permita resolver nuestro problema, es importante caer en la cuenta de que un mismo problema puede resolverse, en muchos casos, por distintos caminos. Por tanto es posible diseñar distintos algoritmos para resolver un mismo problema. Un posible algoritmo para el problema de los números primos sería el siguiente,

- Considerar 2 como el primer números primo. (un número primo es aquel que solo es divisible por si mismo o por uno.)

---

<sup>6</sup>si se emplea un valor de  $n$  muy grande, es posible que se agote el número máximo de llamadas recursivas a una función admitido por Python, en cuyo caso el programa dará un error. Además se puede modificar directamente la función `fibonacci` para obtener los  $n$  primeros términos de la serie con menos esfuerzo

culeate the value of the nth term of the series by adding the values of the previous two. The function will be calling itself till it arrives at  $n \leq 2$ . From then on, it will be returning the values of the terms obtained in each call till it gets the nth one.

the second function `fibo_series`, uses a *list comprehension* to create a list with the  $n$  first terms of the Fibonacci series<sup>6</sup>.

#### 1.7.4. Algorithms and flow charts.

According to the Oxford Dictionary, an algorithm is defined as:

##### 2. Mathematics and Computing.

A procedure or set of rules used in calculation and problem-solving; (in later use spec.) a precisely defined set of mathematical or logical operations for the performance of a particular task.

The word algorithm is a transliteration of the Arab mathematician *Al-Juarismi* name (circa 780-850 ac). Algorithms are fundamental in programming due to they are the previous step to program creation.

Usually, we depart from a problem we have a precise statement for. For instance: Obtain the first  $n$  prime numbers.

The next step would be to define an algorithm that allows us to solve the problem. It is important to realise that the same problem can, in many cases, be solved by different methods. Therefore, it is possible to design different algorithms to solve the same problem. An algorithm for the odd numbers problem may be as follows,

- Take number 2 as the first prime number. (A prime number is such that is only divisible by itself and by one)
- Cover all odd numbers from 3 till the number  $n$  of prime numbers required is completed.

---

<sup>6</sup>If the number  $n$  is too large, it is possible that the maximum number of recursive function calls allowed by Python be reached. In this case, the program casts an error. Besides, we could modify the function `fibonacci` to obtain the list with the first  $n$  terms of the series, with less effort.

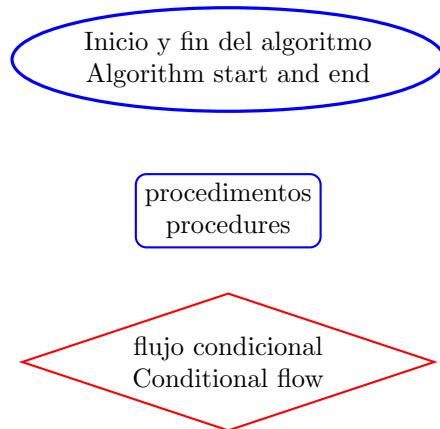


Figura 1.16: Símbolos empleados en diagramas de flujo

Figure 1.16: Symbols used in flowcharts

- Recorrer todos los números impares desde 3 hasta que se complete el número  $n$  de números primos solicitados.
- Para cada número, probar a dividirlo por todos los primos obtenidos hasta ese momento. Si no es divisible por ninguno, el número es primo y se guarda, si es divisible por alguno de ellos, se interrumpe el proceso y se prueba con el siguiente.

En ocasiones, facilita la comprensión de un algoritmo representarlo gráficamente mediante un diagrama de flujo. Los diagramas de flujo emplean símbolos bien definidos para representar los distintos pasos de un algoritmo y flechas para indicar la relación entre ellos; la relación en la que la información *fluye* de un paso del algoritmo a otro.

No hay una norma rígida para realizar un diagrama de flujo, el grado de detalle con que se describe el algoritmo va en función de las necesidades del programador, o de los destinatarios a quienes va dirigido el diagrama. La idea fundamental es que facilite la comprensión del algoritmo. Se utilizan diversos símbolos para indicar, procedimientos, condiciones, almacenamiento de resultados, etc. La figura 1.16 muestra los tres símbolos más empleados.

Para indicar el inicio y el fin de un algoritmo se emplea como símbolo una elipse. Para indicar un procedimiento concreto, como por

- For each odd number, try to divide it between all prime numbers discovered so far. If the number is not divisible by any of them, then the number is prime. save it and try the next odd number. Otherwise, i.e. if it is divisible for one of them, break the process and try the next odd number.

Sometimes understanding an algorithm is easier if we represent it graphically using a flow chart. Flowcharts use well-defined symbols to represent the steps of an algorithm and arrows to indicate the relationship among the steps; how the information *flows* from one step to another.

There is no rigid form to designing a flow chart; the degree of detail description used in the algorithm depends on the programmer's needs or the needs of the people the flow chart addresses. The basic idea is to use the chart to ease the algorithm's understanding. There are several symbols to indicate procedures, conditions, data storage, etc. Figure 1.16 shows the three most common chart symbols.

We use an ellipse to indicate the beginning and end of an algorithm. To indicate a procedure, for instance, to perform a computation,

ejemplo realizar un cálculo, asignar un valor a una variable, etc, se emplea como símbolo un rectángulo. Por último se emplea un rombo como símbolo, para representar una condición.

Los símbolos se relacionan mediante flechas que indican el sentido en que se ejecuta el algoritmo. los rombos suelen tener dos flechas de salida marcadas con las palabras "si" y "no", para indicar por donde sigue el flujo de información dependiendo de si la condición representada se cumple o no.

Por último un bucle se representa habitualmente mediante una flecha que devuelve el flujo a un símbolo ya recorrido anteriormente.

La figura 1.17 muestra un posible diagrama de flujo para el problema de los números primos. Como puede observarse, contiene más información que la versión que hemos dado del algoritmo descrito con palabras.

Las líneas que marcan los flujos de información nos indican que será necesario implementar un bucle exterior hasta que se complete el número  $n$  de primos solicitados y un bucle interior que deberá comprobar si cada nuevo número impar que probamos, es divisible por los números primos encontrados hasta ese momento.

Hay una tercera condición que debe interrumpir la comprobación para el primer número primo que resulte ser divisor del número que se está comprobando.

Es fácil extraer del diagrama de flujo las estructuras de programación que necesitaremos para elaborar un código que nos permita resolver el problema planteado. Por ejemplo, parece lógico implementar el bucle exterior empleando un bucle **while**, implementar el bucle interior con un **for**, que de tantas iteraciones como primos se han encontrado hasta ese momento, Emplear un **break** para interrumpir la comprobación, etc.

Por supuesto es posible realizar un diagrama de flujo más detallado, en el que incluso se incluya explícitamente parte del código que se va a utilizar. Por ejemplo se podría indicar que se empleará el comando `%` para comprobar si un número es divisible entre otro. Sin embargo, hay que tener cuidado para evitar que un exceso de detalle dificulte entender la lógica del algoritmo contenida en el diagrama.

to assign a value to a variable, etc., we use a rectangle. finally, we use a rhombus as a symbol to indicate a condition.

The symbols are related by arrows to establish the direction of the program flow. Rhombus usually have two exit arrows marked with the words 'yes' and 'not' to indicate where the program flows depending on whether the condition is met or not.

Finally, a loop is usually represented by an arrow which comes back to a previously covered symbol.

Figure 1.17 shows a possible flow chart for the prime numbers problems. As you can see, it has more information than the text version of the algorithm we gave above.

The arrows describing the information flow show that we must implement an outer loop until the required number  $n$  of prime numbers is fulfilled. We will also need an inner loop if any odd number we test is divisible by any of the prime numbers found so far.

There is a third condition which should break the checking when we find the first odd number being a divisor for the odd number the program is checking.

From the chart flow, it is easy to obtain the programming structures needed to build a code that allows us to solve the prime numbers problem. So, it seems reasonable to implement the outer loop using a **while** loop, a **for** loop –that computes as many iterations as prime numbers have been found so far– for the inner loop, a **break** to interrupt the checking process, etc.

Of course, it is possible to build a more detailed flow chart, even including explicitly part of the code to be included in the program. For instance, we may indicate that we will use the command `%` to check if a number is divisible by another. Nevertheless, it is essential to note that you should be careful to avoid an excess of details that make it difficult to understand the logic of the algorithm described by the flow chart.

Eventually, we have to cast the algorithm into code, generating a computer program that allows us to solve the problem. We need to

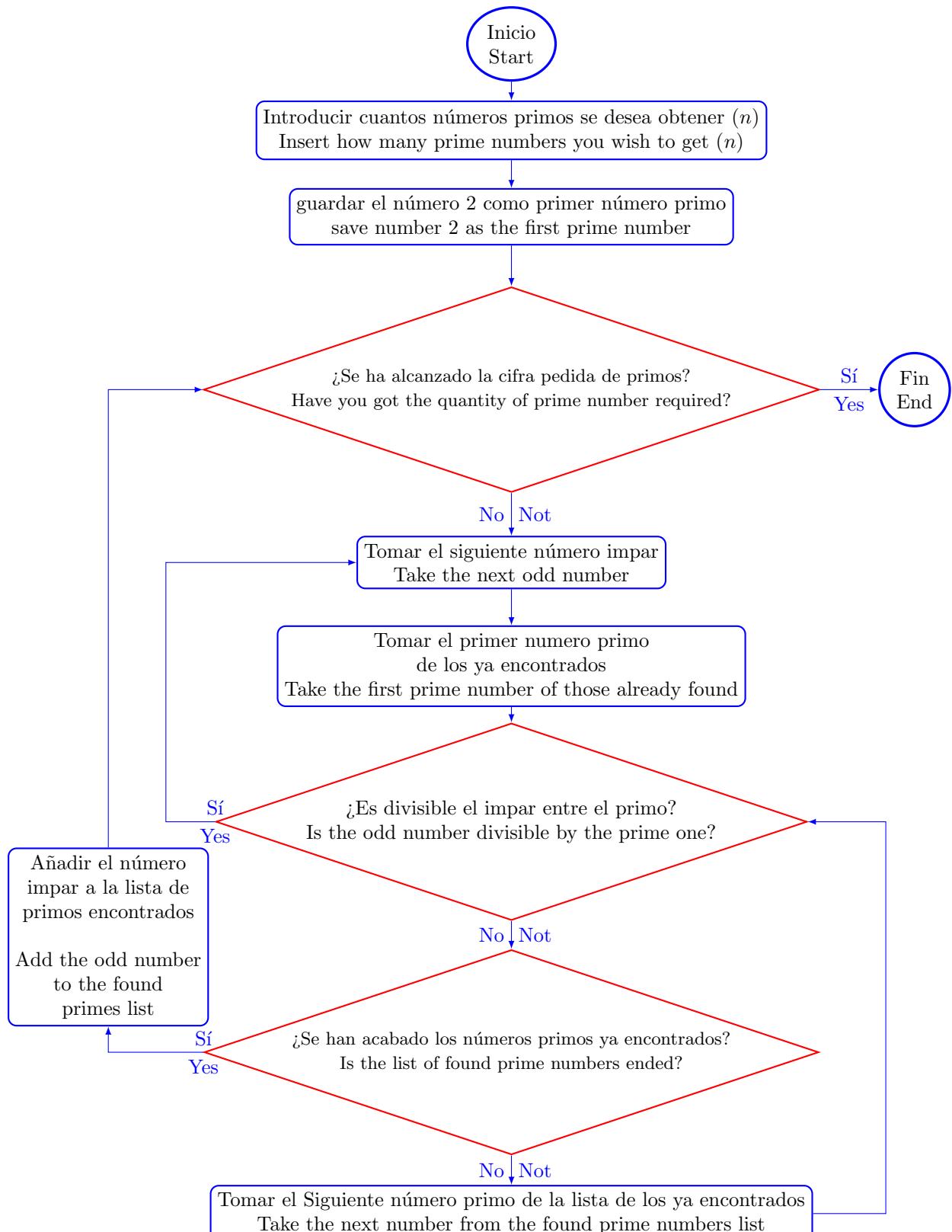


Figura 1.17: Diagrama de flujo para el problema de los números primos

Figure 1.17: Flow chart for the prime numbers problem

Por último el algoritmo se codifica dando lugar a un programa de ordenador que permite resolver el problema. Para ello, hay que identificar las instrucciones del algoritmo con estructuras de programación validas: bucles, condicionales, etc.

El siguiente código muestra dos funciones para calcular los  $n$  primeros números primos. La primera emplea una variable de aviso para detectar cuando un número impar no es divisible por todos los primos anteriores a él. Trata de comprender como funciona

La segunda emplea una estructura `for` especial definida en Python, de la que no hemos hablado anteriormente. Se trata de la estructura `for - else`. Si nos fijamos en las líneas de código 37 a 41, vemos una estructura `for` en la que se comprueba si el numero impar correspondiente es divisible por los primos ya obtenidos. En la linea 41 aparece una instrucción `else`; dicha instrucción esta asociada al bucle `for` y solo se ejecuta cuando el bucle `for` llega al final. En nuestro caso, solo se ejecutará si el bucle `for` no es interrumpido por la instrucción `break` contenida en él. Es decir solo se ejecuta si el numero impar que estamos probando no es divisible por ninguno de los primos anteriores. Si se da esta situación, el bucle `for` llega has el final y además quiere decir que el número impar en cuestión es el además el siguiente número primo. El programa entra entoces en el bloque `else` y ejecuta la única sentencia que contiene, guardando el número impar en la lista de primos. Ni que decir tiene que, como siempre, el código correspondiente al bloque `else`, debe estar indentado.

identify the algorithm instruction with valid programming structures: loops, conditionals, etc.

The following code shows two functions for calculating the  $n$  first prime numbers.

The first function uses a flag variable to detect when an even number is not divisible for any prime number less than it. Try to understand how it works.

The second one uses a Python special `for` structure, which we have not spoken of so far. It is the `for - else` structure. If we focus on code lines 37 to 41, we shall see a `for` structure in which we check if the odd number we are checking is divisible by any of the previously found prime numbers. In line 41, there is an `else` instruction. It is linked to the `for` loop and the program will execute this instruction only if the for loop reaches its end. In our case, the program reaches this `else` command only if the `break` instruction inside does not interrupt the `for` loop. In other words, this happens only if the odd number we are checking is not divisible by any of the previously found primes and, therefore, it is the next prime number. Then the program gets into the `else` block and executes the single line it contains, saving the odd number in the list of prime numbers. It is needless to say that the code within the "else" statement should be properly indented.

---

primos.py

---

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Tue May 14 11:22:29 2024
5  función para obtener los primeros n numeros primos
6  En la primera empleamos un flag.
7  En la segunda empleamos la estructura for else
8  function to get the first n prime numbers
9  @author: juan
10 """
11
12 def numpr(n):

```

```

13     primos = [2] #introducimos 2 como el primer primo
14             #we take 2 as the first prime number
15     i   = 0
16     impar = 3 #tomamos 3 como el rpier impar para probar
17             #we take 3 as the first odd number to test
18     while i<n:
19         pr = 1 #flag activation
20         for i in primos:
21             if impar%i == 0:
22                 pr = 0 #flag deactivation
23                 break
24         if pr ==1:
25             primos.append(impar)
26
27         impar = impar + 2
28     return(primos)
29
30 def numprels(n):
31     primos = [2] #introducimos 2 como el primer primo
32             #we take 2 as the first prime number
33     i   = 0
34     impar = 3 #tomamos 3 como el primer impar para probar
35             #we take 3 as the first odd number to test
36     while i<n:
37         for i in primos:
38             if impar%i == 0:
39                 break
40         else:
41             primos.append(impar)
42
43         impar = impar + 2
44     return(primos)
45
46
47
48

```

---

## 1.8. Exportar e importar datos en Python

Almacenar datos es importante para trabajar en varias sesiones y compartir resultados. Dado que cuando Python se cierra, todas las variables de la memoria se pierden, los datos deben almacenarse de alguna otra forma. A veces también es necesario leer datos que están almacenados en un archivo.

### 1.7.5. Reading and writing data in Python

Storing data is important for working in multiple sessions and sharing results. Since when Python shuts down, all variables in memory are lost, data must be stored in some other way. Sometimes it is also necessary to read data that is stored in a file.

**Text files** A text file, many times saved with the *.txt* extension, is a file that only contains plain text. To work with text files we need to

**Archivos de texto** Un archivo de texto, que muchas veces se guarda con extensión `.txt`, es un archivo que sólo contiene texto sin formato. Para trabajar con ficheros de texto, necesitamos utilizar la función `open` que devuelve un objeto de tipo fichero. Se suele utilizar con dos argumentos:

```
f=open(filename, mode)
```

`f` es el objeto tipo archivo devuelto por la función, `filename` es un string con el nombre y ruta del archivo de texto que se quiere abrir y `mode` es otro string que indica de qué manera va a usarse el archivo. Algunos de los modos más comunes son:

- `'r'` es el modo por defecto. Indica que el archivo se abre sólo para leerse.
- `'w'` abre el archivo para escribir en él. Si el archivo no existe lo crea.
- `'a'` es el modo añadir. Añade texto al final del archivo. Si el archivo no existe lo crea al igual que el modo `'w'`.
- `'r+'` abre un archivo (pero no lo crea) para leer y escribir en él.
- `'w+'` abre un archivo para leer y escribir, lo crea si es que no existe. Borra lo que tuviera escrito anteriormente en él.
- `'a+'` abre un archivo para leer y escribir, lo crea si es que no existe. Añade los datos al final del archivo.

Los objetos de tipo archivo tienen métodos que nos permiten acceder y modificar su contenido. Algunos de esos métodos son:

- `write` que permite escribir datos en el archivo
- `read` lee todos los datos que contiene el archivo y los devuelve
- `readline` lee una línea (hasta el carácter fin de línea) del archivo.
- `readlines` lee el archivo completo pero devuelve una lista donde cada línea es uno de los elementos

use the `open` function, that returns a file object. It is usually used with two arguments:

`f` is the file object returned by the function, `filename` is a string with the name and path of the text file to be opened and `mode` is another string indicating how the file is to be used. Some of the most common modes are:

- `'r'` is the default mode. It indicates that the file is opened for reading only.
- `'w'` opens the file for writing to. If the file does not exist, it creates it.
- `'a'` is the append mode. Adds text to the end of the file. If the file does not exist, it creates it as in `'w'` mode.
- `'r+'` opens a file (but does not create it) for reading and writing to it.
- `'w+'` opens a file for reading and writing, creates it if it does not exist. Discards what was previously written to it.
- `'a+'` opens a file for reading and writing, creates it if it does not exist. Adds data to the end of the file.

File type objects have methods that allow us to access and modify their content. Some of these methods are:

- `write` which allows writing data to the file
- `read` reads all the data contained in the file and returns it
- `readline` reads a line (up to the end-of-line character) from the file
- `readlines` reads the whole file but returns a list where each line is one of the elements

En el siguiente ejemplo se puede ver el código para usar algunos de los métodos anteriores. Se abre un archivo y se escriben 5 líneas en él. A continuación se lee dicho archivo. Se vuelve a abrir el archivo pero en modo añadir `a`, escribiendo una línea al final del archivo. Se lee pero usando el método `readlines`, de manera que lo que se obtiene es una lista donde cada elemento es una de las líneas. Finalmente se vuelve a abrir el archivo y se lee, en la variable `line` cada línea, se para el bucle cuando el método `readline` no devuelve nada.

```
f=open('textfile.txt','w')
for i in range(5):
    f.write(f"Line {i}\n")
f.close()

f=open('textfile.txt','r')
print(f.read())
f.close()

f=open('textfile.txt','a')
f.write("Another line")
f.close()

f=open('textfile.txt','r')
print(f.readlines())
f.close()

f=open('textfile.txt','r')
while True:
    line=f.readline()
    if not line:
        break
    print(line)

f.close()

Line 0
Line 1
Line 2
Line 3
Line 4

['Line 0\n', 'Line 1\n', 'Line 2\n', 'Line 3\n', 'Line 4\n', 'Another line']

Line 0

Line 1
```

In the following example you can see the code to use some of the above methods. You open a file and write 5 lines to it. Then read the file. The file is reopened but in append mode, writing a line to the end of the file. It is read but using the `readlines` method, so that what you get is a list where each element is one of the lines. Finally the file is reopened and read, in the variable `line` each line, and the loop is stopped when the `readline` method does not return anything.

Line 2

Line 3

Line 4

Another line

**Trabajar con números y arrays** En muchos de los casos vamos a necesitar guardar y leer números o matrices. Podríamos utilizar los métodos anteriores para guardar los números o matrices en un archivo y leerlos de nuevo. Sin embargo, una manera mejor de hacerlo es usar el paquete numpy para guardar y leer directamente un array.

Dentro de numpy tenemos los métodos **savetxt** y **loadtxt** para guardar arrays en archivos de texto o cargarlos desde archivos de texto.

Para usar **savetxt** tenemos que proporcionar los siguientes argumentos: un string con el nombre del archivo y la variable que contiene el array que vamos a guardar. Además, de manera opcional se pueden proporcionar más. Por ejemplo el formato, bajo la clave de **fmt** (en el ejemplo el formato es un número (%), con tres cifras decimales (.3) de tipo float (f). El tipo de delimitador entre elementos, una cabecera etc. Se puede consultar la documentación para verlos todos.

El método **loadtxt** nos permite leer un array de un archivo y guardarlo directamente en una variable de tipo array de numpy. Para ello simplemente proporcionaremos el archivo donde está almacenado el array y el método nos devolverá el array. En el ejemplo puede verse cómo se guarda una matriz  $2 \times 3$  en un archivo de texto, con una cabecera, cómo se lee usando las funciones **open** y **read** y cómo se lee la matriz y se almacena en un array de numpy usando el método **savetxt**.

```
import numpy as np

arr=np.array([[0.3,4.3,3.5],[-1.23,0.032,-3.23]])
np.savetxt('matrix01.txt', arr,fmt='%.3f', delimiter=' ', header='Col1 Col2 Col3')
f=open('matrix01.txt','r')
```

**Working with numbers and arrays** In many cases we will need to save and read numbers or arrays. We could use the above methods to save the numbers or arrays to a file and read them back. However, a better way to do this is to use the numpy package to directly save and read an array.

Within numpy we have the methods **savetxt** and **loadtxt** to save arrays to text files or load them from text files.

To use **savetxt** we have to provide the following arguments: a string with the name of the file and the variable containing the array to be saved. In addition, you can optionally provide more. For example the format, under the key of **fmt** (in the example the format is a number (%), with three decimal digits (.3) of type float (f)). The type of delimiter between elements, a header etc. You can consult the documentation to see them all.

The **loadtxt** method allows us to read an array from a file and store it directly into a numpy array variable. To do this we simply provide the file where the array is stored and the method will return the array. In the example you can see how a  $2 \times 3$  array is stored in a text file, with a header, how it is read using the **open** and **read** functions and how the array is read and stored in a numpy array using the **savetxt** method.

```

print(f.read())
f.close()
#Now we read it with loadtxt

m1= np.loadtxt('matrix01.txt')
print(m1)

# Col1 Col2 Col3
0.300 4.300 3.500
-1.230 0.032 -3.230

[[ 0.3    4.3    3.5   ]
 [-1.23   0.032 -3.23 ]]

```

**Datos en formato CSV** Hay muchos datos científicos que se almacenan en el formato de archivo de valores separados por comas (CSV). Se trata de un archivo de texto que utiliza una coma para separar los valores. Es un formato muy útil que permite almacenar grandes tablas de datos (números y texto) en texto plano. Cada línea (fila) de los datos es un registro de datos, y cada registro consta de uno o varios campos, separados por comas. También se puede abrir con Microsoft Excel, y visualizar las filas y columnas.

Python tiene su propio módulo csv que puede manejar la lectura y escritura del archivo csv. No vamos a introducir este módulo csv aquí. En su lugar, usaremos el paquete numpy para manejar el archivo csv ya que muchas veces leeremos el archivo csv directamente a un array numpy.

Usando el mismo método `savetxt` de numpy del epígrafe anterior podemos indicar que el carácter delimitador que queremos poner para separar los datos es una coma. A la hora de leer los datos con `loadtxt` tendremos también que indicar que se ha usado la coma para separar unas columnas de otras, como se ve en el siguiente ejemplo.

```

import numpy as np

# Create a random 20x4 matrix
m2=np.random.random([20,4])

```

**Data in CSV format** There is a lot of scientific data that is stored in the Comma Separated Values (CSV) file format. This is a text file that uses a comma to separate values. It is a very useful format that allows large tables of data (numbers and text) to be stored in plain text. Each line (row) of data is a data record, and each record consists of one or more fields, separated by commas. It can also be opened with Microsoft Excel, and rows and columns can be displayed.

Python has its own csv module that can handle reading and writing the csv file. We will not introduce this csv module here. Instead, we will use the numpy package to handle the csv file since we will often read the csv file directly to a numpy array.

Using the same `savetxt` method of numpy from the previous section, we can indicate that the delimiter character we want to use to separate the data is a comma. When reading the data with `loadtxt` we will also have to indicate that the comma has been used to separate some columns from others, as shown in the following example.

```
#Save it as a csv file
np.savetxt('randomMat.csv',m2, fmt='%.2f', delimiter=',')

#Read it back with loadtxt
m3=np.loadtxt('randomMat.csv',delimiter=',')

print(m3)

[[0.08 0.91 0.36 0.19]
 [0.61 0.22 0.7 0.66]
 [0.77 0.57 0.77 0.4 ]
 [0.58 0.32 0.89 0.78]
 [0.63 0.09 0.97 0.78]
 [0.35 0.7 0.5 0.96]
 [0.65 0.78 0.04 0.04]
 [0.96 0.31 0.45 0.26]
 [0.75 0.51 0.09 0.87]
 [0.93 0.13 0.8 0.61]
 [0.67 0.99 0.83 0.76]
 [0.21 0.72 0.56 0.46]
 [0.81 0.82 0.67 0.88]
 [0.12 0.01 0.02 0.55]
 [0.88 0.08 0.95 0.7 ]
 [0.74 0.84 0.56 0.3 ]
 [0.77 0.73 0.67 0.89]
 [0.44 0.23 0.67 0.18]
 [0.61 0.4 0.06 0.1 ]
 [0.55 0.35 0.89 0.11]]
```



**Archivos JSON** JSON son las siglas de JavaScript Object Notation y es un formato muy utilizado de archivo de datos, independiente del lenguaje de programación. Un archivo JSON normalmente termina con la extensión ".json". Exploraremos brevemente cómo manejar archivos JSON en Python.

El texto en JSON se guarda mediante cadenas entrecomilladas que contienen valores en pares clave-valor dentro de llaves {}. En realidad es muy similar al diccionario que vimos en Python. Por ejemplo:

{

**JSON files** JSON stands for JavaScript Object Notation and is a widely used data file format, independent of programming language. A JSON file usually ends with the extension ".json". We will briefly explore how to handle JSON files in Python.

Text in JSON is stored as quoted strings containing values in key-value pairs inside {} braces. It's actually very similar to the dictionary we saw in Python. For example:

```

"id": "Frodo",
"tipo" : "Hobbit",
"estado":{
    "posicion": [2.345,-5.985],
    "velocidad lineal": 0.15,
    "velocidad angular": 0,
    "baterias": 7.32
}
}

```

Aunque podríamos trabajar directamente con el archivo de texto y todas las funciones anteriores, hay librerías de python que nos ofrecen métodos para guardar, leer y manipular datos en formato JSON de manera sencilla. Una de ellas es la librería `json`. Importando esta librería tenemos el método `dump` para guardar un diccionario de python (uni en el ejemplo) en un archivo json. A este proceso se le llama serializar. A `dump` hay que proporcionarle el objeto python de tipo archivo (por eso abrimos un archivo con `open` en modo escritura 'w').

Para leer un archivo de datos de tipo JSON y guardar el contenido en un diccionario de python podemos usar el método `load` de la librería `json`. Al igual que a `dump` tendremos que proporcionarle el objeto archivo (ahora lo abriremos en modo lectura 'r') y nos devolverá un objeto de tipo diccionario.

Although we could work directly with the text file and all the above functions, there are Python libraries that provide methods to store, read and manipulate data in JSON format in a simple way. One of them is the `json` library. Importing this library we have the `dump` method to save a Python dictionary (uni in the example) in a json file. This process is called serialize. The `dump` must be provided with the python object of type file (that's why we open a file with `open` in write mode 'w').

To read a JSON data file and save the content in a Python dictionary, we can use the `load` method of the `json` library. Like `dump` we will have to provide it with the file object (now we will open it in read mode 'r') and it will return a dictionary object.

```

import json as js

robot= {
    "id": "Frodo",
    "tipo" : "Hobbit",
    "estado":{
        "posicion": [2.345,-5.985],
        "velocidad lineal": 0.15,
        "velocidad angular": 0,
        "baterias": 7.32
    }
}

js.dump(robot,open('jsondat.json','w'))

mi_robot=js.load(open('jsondat.json','r'))

```

```
print(mi_robot)  
{'id': 'Frodo', 'tipo': 'Hobbit', 'estado': {'posicion': [2.345, -5.985], 'velocidad lineal': 0,
```

Python es un lenguaje de programación muy usado para trabajar con datos. Cuenta con librerías muy potentes y apropiadas para el manejo de grandes cantidades de datos como por ejemplo pandas.

Python is a widely used programming language for working with data. It has very powerful libraries suitable for handling large amounts of data such as pandas.



## 1.9. Ejercicios

1. Tipos de variables en Matlab; matrices y vectores. Escribe, por orden, las siguientes expresiones en la *línea de comandos* de Matlab e interpreta los resultados que obtienes. En los casos en que Matlab devuelva un mensaje de error, trata de averiguar la razón. **Nota:** Es importante hacerlos por orden ya que algunos operaciones se apoyan en los resultados de operaciones anteriores.

```

1 a=[[1,2,3],[4,5,6]]
2 a=[1,2,3],4,5
3 a=[[1 2 3] 1 2 3]
4 a=[[1,2,3],[5,6,8],[3,2,6]]
5 b=[1,2,3]
6 c=[a b]
7 c=[a,b]
8 d=[b,a]
* function/function type()
9 type(a)
10 type(a[0])
11 type(a[0][1])
12 a[2][2] = 'b'
13 type(a[2][2])
* cambiando listas copiadas/changing
  copied lists
14 c[1] = 'ojo'
15 print('c=',c)
16 print('a=',a)
17 print('b=',b)
18 print('d=',d)
19 e = a.copy()
20 a[2][2] = 'cambia tb e'
21 print('c=',c)
22 print('a=',a)
23 print('b=',b)
24 print('d=',d)
25 print('e=',e)
26 e[2][2]= 'pero no al revés'
27 print('c=',c)
28 print('a=',a)

29 print('b=',b)
30 print('d=',d)
31 print('e=',e)
32 %who
33 %whos
34 del(d)
35 %reset
* Operadores/operators
36 a = -6
37 b = 5
38 c = 3.5
39 e = 0.9
40 a*e
41 g=a*e
42 g=e*a
43 a*b
44 r=a-b
45 f=(a+b)*c
46 f= a+b*c
47 p=a**2
48 r=a**2-a*a
* Encadenando operaciones/linking
  operations
49 w=a*b**-1
50 w=a/b
51 w=a//b
52 w=a*b**-1-a/b
53 w=a**-1*b
54 w=(a*b)**-1
55 w=a**-1*b-1/a*b
56 w=a**-1*b-a
* Relacionales y lógicos / relational
  and logic
57 y0 = p<2
58 y1 = p<=2

```

<pre> 59 y2 = p&gt;2 60 y3 = p&gt;=2 61 y4 = p==2 62 y5 = p!=36 63 r1 = a&lt;5 and b==5 </pre>	<pre> 64 r2 = a&lt;b or b&gt;=5 65 r3 = a&lt;b or b&gt;5 66 r4 = (a&lt;b or b&gt;=5) and not(a&lt;b and b&gt;=5) </pre>
--	---

2. Obtén empleando la *línea de comandos* de Ipython el resultado de las siguientes expresiones:

$$\frac{\frac{5 + 3x^2}{6y} - \frac{1 + 4\sqrt{2x}}{6}y}{\frac{-x + \sqrt{x^2 - 4xy}}{2x}}$$

Emplea para  $x$  e  $y$  los valores que prefieras.

3. Abre un fichero nuevo en el editor de textos de Spyder. Escribe las siguiente líneas:

```

A = [35,23]
B = [12,-12.34]
x = [A[0]*B[0],A[1]*B[1]]
y = [A[0]+B[0],A[1]+B[1]]

```

Guarda el fichero con el *primero.py*. Emplea el comando `%reset` en el terminal de IPython para borrar cualquier variable anterior. Ejecuta el fichero y comprueba que existen las variables `A`, `B`, `x`, `y`.

Vuelve ejecutar el comando `%reset`. Emplea el comando `import primero`. ¿Se han creado las variables? ¿qué ha pasado ahora?

Vuelve a borrar las variables del namespace de Python e importa de nuevo tu programa con `import primero as kiki`. ¿Dónde estan ahora las variables `A`, `B`, `x`, `y`?

Crea una variable `x = 'nueva'` Comprueba que las variables `x` y `kiki.x` son distintas.

Importa la variable `x` del modulo `primero` directamente usando la expresion:

`from primero import x`

¿Cuánto vale ahora la variable `x`?

4. Crea en el terminal de IPython la siguiente función, ponle el nombre que quieras

---

```

1 def mifun(A):
2     E = A[0] + A[1]**2
3     D = A[0] - A[1]**-1
4     return(E,D)

```

---

Crea variables adecuadas, ejecuta la función que has creado, guardando el resultado en una sola variable `resultado`.

Ejecuta las siguientes líneas de comando y trata de explicar lo que hacen

```

uno = 2
dos = 4
resultado = mifun([uno,dos])
print(resultado)
me, da = mifun(resultado)
la,risa = mifun(mifun([uno,dos]))
print(me,da,la,risa)

```

5. Escribe el siguiente código en un fichero. Ejecútalo y analiza los resultados,

---

```

1      from random import random
2      dato = random()
3      if dato == 0.5:
4          dato = 0
5      elif dato < 0.5:
6          dato = -dato*10
7      else:
8          dato = dato*10
9      print(dato)

```

---

Modifica el programa,

- a) Para que muestre los resultados de los cálculos por pantalla
  - b) para que, genere una variable (distinta de 'dato') con valor  $-1$  si se cumple que 'dato' es menor o igual que  $0.3$ , con valor  $-0.5$  si 'dato' es mayor que  $0.3$  y menor que  $0.5$ , con valor  $0$  si 'dato' es mayor o igual que  $0.5$  y menor o igual que  $0.7$  y con valor  $1$  en cualquier otro caso.
  - c) Convierte el *script*, en una función, que tome 'dato' como variable de entrada, y devuelva el valor de la variable obtenida, según la condición que se cumpla. (Lógicamente se debe eliminar o comentar la línea `dato=random()`).
6. Crea un scrip de python con el siguiente código y trata de adivinar qué hace antes de ejecutarlo. Utiliza el *help* de Spyder o el interrogante `abs?` en el terminal, para averiguar lo que hace la función `abs`.

---

```

1      n = 5
2      x = []
3      for i in range(n):
4          x.append([])
5          for j in range(n):
6              if i == j:
7                  x[i].append(1)
8              elif abs(i-j) == 1:
9                  x[i].append(-1)
10             else:
11                 x[i].append(0)

```

---

Introduce un punto de ruptura (*breakpoint*) en el código, por ejemplo en la línea 12. Ejecuta paso a paso el programa para ver como va generando el resultado.

7. Crea un script con el siguiente código,

---

```

1  n = 5
2  x = []
3  i = 0
4
5      while i<n:
6          j=0
7          x.append([])
8          while j<n:
9              if i==j:
10                  x[i].append(1)
11              elif abs(i-j)==1:
12                  x[i].append(-1)
13              else:
14                  x[i].append(0)
15              j = j + 1
16      i = i+1

```

---

Repite el estudio del programa anterior

8. Modifica el *script* anterior de modo que quede el siguiente código,

---

```

1  n = 5
2  x = []
3  i = 0
4      while 1:
5          j=0
6          x.append([])
7          while 1:
8              if i==j:
9                  x[i].append(1)
10             elif abs(i-j)==1:
11                 x[i].append(-1)
12             else:
13                 x[i].append(0)
14             j = j + 1
15             if j==n:
16                 break
17         i = i+1
18         if i==n:
19             break

```

---

Repite el estudio del programa anterior. ¿Qué sucede si comentamos la(s) línea(s) que contiene la sentencia `break`? Reconvierte los *scripts* anteriores en funciones que admitan como variable de entrada el tamaño de la lista que se desea generar y devuelvan como variable de salida la lista creada.

9. Escribe —empleando bucles y sin hacer uso del operador (`**`) — una función `potencia` que reciba como argumentos de entrada un número real  $x \in \mathbb{R}$  y un entero  $n \in \mathbb{Z}$  y devuelva el

valor de la  $n$ -ésima potencia de  $x$  ( $x^n$ ). El segundo argumento de entrada puede ser opcional. Si se omite debe tomarse como  $n = 2$  (devolviendo el cuadrado de  $x$ ).

- Crea una función que determine la suma de los inversos de los números naturales impares menores de 1000. Emplea bucles para el cálculo.

$$S_{imp} = \sum_{n=1}^{500} \frac{1}{2n-1} = \frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{999}$$

Modifica tu programa, de modo que admita como variable de entrada un número  $m \in \mathbb{N}$  y calcule el valor de la suma de los inversos de los impares menores o iguales que  $m$ .

- Los números de Fibonacci,  $F(n)$  son una serie de números naturales. Los dos primeros valores de la serie son:  $F(1) = 1$  y  $F(2) = 2$ . Los demás términos se calculan como la suma de los dos anteriores:  $F(n) = F(n - 1) + F(n - 2)$ . Escribe un script que genere los primeros  $m$  números de la serie de Fibonacci, almacenándolos en una lista  $F$ . Emplea para el cálculo la estructura `while`.
- Escribe una función que admita como variable de entrada una lista y devuelva como variable de salida otra lista de igual tamaño con los elementos colocados en orden inverso a los de la lista original.  
¿Sabrías hacerlo con una única línea de código? Pista: Usa *List comprehension*
- Modifica el programa del ejercicio 11 para que guarde la lista con los números de Fibonacci creada en un archivo de texto.
  - Modifícalo ahora para que los guarde en un archivo *CSV*.

# Índice alfabético

- = Símbolo de asignación, 17
- anaconda, 10
- Anidación
  - If anidado, 63
  - While anidado, 74
  - For anidado, 70
- Bucles, 65
  - Bucle for, 65
  - Bucle while, 72
- Control de flujo, 57
- Depurador, 52
- Editor de textos, 37
- Error
  - de sintáxis, 53
- Errores
  - de codificación, 54
- Flow
  - Loops, 65
- Flujo, 57
  - Bucle for, 65
  - Bucle while, 72
  - Diagrama de Flujo, 77
  - Bucles, 65
  - Condicional, 57
- Funciones
  - Recursivas, 75
- funciones, 41
- import, 51
- indexación, 21
- Ipython, 15
- List comprehension, 71
- Loops, 65
- Namespace, 46
- Nesting
  - Nested for, 70
- Operaciones, 26
  - Aritméticas, 26
- Operadores
  - Relaciones y lógicos, 30
  - Precedencia, 28
- Operations
  - Arithmétics, 26
- Python, 9
  - entorno de programación, 10
- Scripts, 36
- Spyder, 14
- Símbolo de asignación, 17
- Variable
  - nombre, 17
  - tipo, 19
  - Ambito, 46
- Variables, 17
- Ámbito de una variable o función, 51



# Alphabetic Index

= assignment symbol, 17  
[, 46  
Anaconda, 10  
Assignment Symbol, 17  
Debugger, 52  
Error  
    syntax error., 53  
Flow  
    while loop, 72  
flow, 57  
    for loop, 65  
    Conditional, 57  
flow control, 57  
functions, 41  
  
import, 51  
indexation, 21  
  
List comprehension, 71  
Loops  
    while loop., 72  
  
Nesting  
    Nested if, 63  
    Nested While, 74  
  
Operations, 26  
    Relational and Logic , 30  
Operators  
    Precedence, 28  
  
Python, 9  
    IDE, 10  
  
Scripts, 36  
Spyder, 14  
  
Text editor, 37  
  
Variable and function scope, 51  
Variable  
    name, 17  
    Scope, 46  
    Type, 19  
Variables, 17