

Reducción de datos CCD con IRAF y Python

Alexis Andrés

6 de septiembre de 2024

Índice general

1	Introducción a Python	5
Clase 1 Introducción a Python 6		
1	Nociones básicas	6
1.1	Python en diferentes sistemas operativos	6
1.1.1	Python en Windows	7
1.1.2	Python en Linux	7
1.1.3	Python en Mac OSX	8
1.2	Ejecutando Python	9
1.2.1	Creando un entorno virtual	10
1.2.2	El editor Geany	11
1.2.3	Jupyter Notebook	13
1.2.4	Visual Studio Code y PyCharm	15
1.3	Python como calculadora	15
1.3.1	Operaciones aritméticas	15
1.3.2	Operaciones matemáticas avanzadas	17
Clase 2 Variables y contenedores 19		
2	Introducción	19
2.1	Variables	19
2.1.1	Variables tipo int/float	20
2.1.2	Variables de tipo string	20
2.1.3	Comentarios	21
2.1.4	Variables tipo bool	22
2.2	Contenedores	23
2.2.1	Listas	23
2.2.2	Tuplas	26
2.2.3	Conjuntos	27

2.2.4	Diccionarios	28
Clase 3 Control de flujo y lógica		30
3	Controles de flujo	30
3.1	Condicionales	30
3.1.1	Declaraciones if	30
3.1.2	Declaraciones if-else	31
3.1.3	Declaraciones if-elif-else	32
3.1.4	Expresiones if-else	33
3.2	Bucles	34
3.2.1	Bucle while	34
3.2.2	Bucle for	36
3.2.3	Contenedores por comprensión	39
Clase 4 Funciones		41
4	Funciones en Python	41
4.1	Funciones predefinidas	41
4.2	Funciones definidas por el usuario	42
4.2.1	Funciones sin parámetros	43
4.2.2	Funciones con parámetros	44
4.2.3	Funciones que devuelven un valor	44
4.3	Más sobre funciones con parámetros	45
4.3.1	Argumentos con un valor predeterminado	46
4.3.2	Argumentos de palabras clave	47
4.4	Problemas	49
2 Introducción a la astronomía observacional		50
Clase 5 Telescopios ópticos		51
5	Telescopios	51
5.1	Naturaleza la luz	51
5.2	Principios básicos de óptica geométrica	53
5.3	Reflexión y refracción	53
5.4	Propiedades de telescopios	54
5.4.1	Recolección de luz	54

Clase 6 Fotometría	55
6 Conceptos de fotometría	55
6.1 Sistema de magnitudes	55
6.1.1 Flujo y luminosidad	55
6.1.2 Magnitudes	57
6.1.3 Magnitudes bolométricas y absolutas	59
6.1.4 Índices de color	60
6.1.5 Filtros de color	61
6.2 Radiación de cuerpo negro	62
Clase 7 Espectroscopía	64
7 Conceptos de espectroscopía	64
7.1 Líneas espectrales	64
7.1.1 Producción de líneas espectrales	65
7.1.2 Tipos de espectros	67
7.1.3 Clasificación espectral de estrellas	68
7.2 Velocidades radiales	68
7.2.1 El efecto Doppler	69
7.2.2 Sistemas binarios espectroscópicos	70
Clase 8 Detectores CCD y calibración	73
3 Python para datos CCD	74
Clase 9 Visualización de datos con Python	75
8 Numpy y Matplotlib	75
8.1 Arreglos numéricos con numpy	75
8.1.1 Arreglos de ceros y unos	76
8.1.2 Arreglos ordenados	77
8.1.3 Arreglos aleatorios	78
8.2 Visualización con Matplotlib	79
8.2.1 Gráfico de funciones	79
8.2.2 Gráfico de histogramas	80
8.2.3 Visualización de matrices	81
8.2.4 Gráfico de una función de dos variables	82

Clase 10 Reducción de datos con Python	85
9 El paquete Astropy	85
9.1 El módulo astropy.units	85
9.2 Trabajando con tablas	87
9.3 Un ejemplo un poco realista	90
Clase 11 Imágenes BIAS y DARK	93
10 El módulo ccdproc	93
10.1 Creando un archivo master bias	93
10.2 Creando un archivo master dark	98
Clase 12 Master flat y calibración final	100
11 Calibración de imágenes	100
11.1 Creando un archivo master flat	100
11.2 Calibrando las imágenes de objeto	105

Unidad 1

Introducción a Python

Clase 1 | Introducción a Python

1 Nociones básicas

Python es un lenguaje de programación interpretado de alto nivel cuyas características permiten su uso en diversas disciplinas. Los lenguajes de alto nivel en programación poseen una sintaxis que se asemeja al lenguaje humano, lo que facilita la claridad y legibilidad del código, haciendo su escritura y comprensión más accesibles. En contraste, los lenguajes de bajo nivel, como el *ensamblador*, utilizan una sintaxis más cercana al lenguaje máquina, lo que los hace más eficientes para la computadora pero más complejos de leer y escribir para los humanos.

Python, al ser un lenguaje interpretado, traduce y ejecuta el código línea por línea durante la ejecución, lo que puede resultar en tiempos de ejecución más largos en comparación con los lenguajes compilados, que se traducen en código máquina antes de ejecutarse. Sin embargo, esta diferencia en el tiempo de ejecución suele ser insignificante en muchos casos prácticos. Además, el tiempo requerido para desarrollar y mantener código en Python es considerablemente menor comparado con los lenguajes de bajo nivel, equilibrando así el tiempo total de desarrollo.

Adicionalmente, Python permite la integración de bibliotecas y códigos escritos en otros lenguajes de programación, aprovechando sus ventajas y capacidades para mejorar el rendimiento. Esta flexibilidad y facilidad de uso han contribuido a que Python se convierta en uno de los lenguajes de programación más populares en la actualidad, especialmente en los campos de ciencias e ingeniería.

En esta primera clase revisaremos algunos conceptos básicos de Python, cómo instalarlo en cualquier sistema operativo y además elegiremos un entorno de desarrollo integrado para escribir nuestros códigos Python.

1.1 Python en diferentes sistemas operativos

Python es un lenguaje de programación multiplataforma, esto significa que puede utilizarse en todos los sistemas operativos principales. Sin embargo, los métodos de instalación difieren

dependiendo de cada sistema operativo. El objetivo de esta sección es que puedas ejecutar el famoso programa «¡Hola Mundo!» usando Python en tu sistema operativo. Ten en cuenta que trabajaremos exclusivamente con Python 3.

1.1.1 Python en Windows

Por lo general, Windows no cuenta con una versión de Python por defecto, por lo que será necesario instalarlo además de un editor de texto. Para verificar si tu versión de Windows cuenta con Python, primero debes abrir el menú de inicio, escribir los caracteres **cmd** y hacer clic sobre la aplicación llamada *símbolo del sistema*. Esto abrirá una terminal de comandos en la que deberás escribir las palabras **python --version** (todo en minúsculas) y presiona la tecla Enter.

Si el resultado es algo similar a **3.x.x**, entonces Python ya está instalado en tu sistema. Sin embargo, si el resultado es un mensaje que dice que **python** no es un comando reconocido, entonces sigue las siguientes instrucciones para instalarlo.

- 1. Descarga el instalador de Python.** Abre tu navegador y dirígete a la página <https://www.python.org/downloads/>. Haz clic en el botón que dice «Download Python 3.x.x» (donde «3.x.x» es la versión más reciente).
- 2. Ejecuta el instalador.** Encuentra el archivo del instalador que acabas de descargar (normalmente estará en tu carpeta de descargas) y haz doble clic para ejecutarlo. En la primera ventana del instalador, asegúrate de marcar la casilla que dice «Add Python 3.x to PATH». Esto es importante para que puedas usar Python desde terminal de comandos sin problemas. Finalmente, haz clic en «Install Now» para comenzar la instalación.
- 3. Verifica la instalación.** Una vez que finalice la instalación, abre una nueva terminal de comandos y escribe las palabras **python --version**. Esto debería mostrar la versión de Python que has instalado.

1.1.2 Python en Linux

Cualquier sistema operativo Linux cuenta con una versión de Python instalada por defecto. Por lo tanto, solo necesitas verificar cuál versión tienes instalada en tu sistema. Para lograrlo,

abre una terminal de comandos con la combinación de teclas Ctrl+Alt+T y escribe las palabras **python --version** (todo en minúsculas) y presiona la tecla Enter. Si el resultado es un mensaje similar a **2.x.x** o si en cambio te aparece un mensaje que dice que el comando **python** no fue encontrado, entonces intenta escribiendo **python3 --version** y presiona Enter. El resultado debería ser algo como **3.x.x**.

1.1.3 Python en Mac OSX

La mayoría de los sistemas Mac OSX cuenta con al menos una versión de Python instalada por defecto. Esto significa que es posible que cuentes con dos versiones, para verificarlo, abre una terminal de comandos haciendo clic en **Aplicaciones > Utilidades > Terminal**. Ahora escribe las palabras **python --version** (todo en minúsculas) y presiona la tecla Enter.

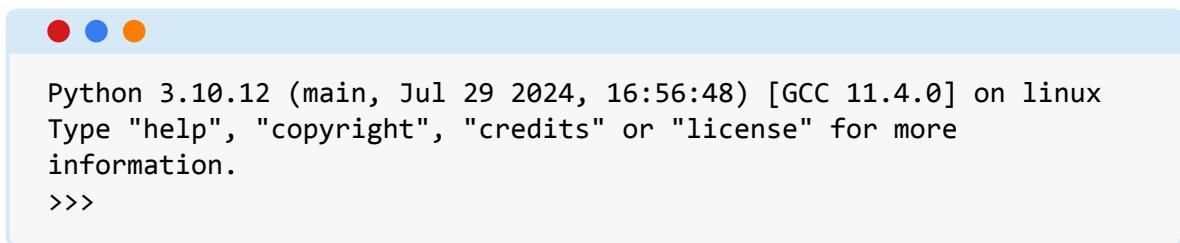
Al igual que con los sistemas Linux, es posible que el resultado sea un error o un mensaje con algo similar a **2.x.x**, lo que significa que la versión instalada es Python 2. Si cualquiera de las dos opciones anteriores es cierta, intenta escribiendo en la terminal **python3 --version** y presiona Enter. Si el resultado es algo similar a **3.x.x**, entonces cuentas con una versión de Python 3.

Si por algún motivo al escribir **python3 --version** obtienes un error, entonces necesitas instalar Python 3 en tu sistema. Para eso, sigue los siguientes pasos.

1. **Descarga el instalador.** Dirígete a la página <https://www.python.org/>. Coloca el cursor sobre el botón «Downloads» y selecciona «macOS». En la página de descargas deberías ver un botón llamado «**macOS 64-bit universal2 installer**» debajo del título «Stable Releases». Haz clic sobre ese botón para descargar el instalador con extensión **.pkg**
2. **Ejecuta el instalador.** Una vez que el archivo **.pkg** se haya descargado, haz doble clic en él para iniciar el proceso de instalación. Aparecerá una ventana del asistente de instalación. Sigue las instrucciones en pantalla. Generalmente, solo necesitas hacer clic en «Continuar» y luego en «Instalar». El instalador pedirá tu contraseña de administrador para proceder. Una vez que se complete el proceso, haz clic en «Cerrar» para terminar.
3. **Verifica la instalación.** Abre una nueva terminal de comandos y escribe las palabras **python3 --version**. El resultado debería ser la versión de Python que acabas de instalar.

1.2 Ejecutando Python

Para comenzar a usar Python, solo tienes que abrir una terminal de comandos, escribir la palabra **python3** y presionar la tecla Enter (Nota: este documento fue escrito en un sistema Linux, si en cambio estás usando Windows, para ejecutar Python solo debes escribir **python** en la terminal de comandos, sin agregar el número 3. Ten esto en cuenta en todo el documento a partir de este momento). El resultado debe ser algo similar a lo siguiente:



```
Python 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

Este mensaje nos muestra información sobre la versión de Python instalada, el sistema operativo en el que se instaló e información general. Inmediatamente después, aparece una línea con el símbolo **>>>**, el cual es conocido como el «Intérprete de Python» (en inglés: *Python prompt*). El intérprete de Python nos indica que podemos ingresar nuestros comandos desde ahí y serán ejecutados al presionar la tecla Enter.

Para nuestro primer ejemplo usaremos la función «**print**» definida por defecto en Python. Escribe el comando «**print("¡Hola mundo!")**» en tu sesión de Python y presionamos Enter. El resultado debería ser el siguiente:



```
>>> print("¡Hola mundo!")
¡Hola mundo!
>>>
```

Como puedes ver, la instrucción que se utilizó para mostrar el mensaje «¡Hola mundo!» es muy similar al lenguaje humano. Además, en la línea siguiente aparece de nuevo el intérprete de Python. Esto nos indica que Python está listo para seguir recibiendo instrucciones.

Para salir de Python y volver a una terminal de comandos normal, debes utilizar la función «**exit**» también definida por defecto. Específicamente, debes escribir el comando «**exit()**» y presionar Enter.

Esta es la forma más simple de ejecutar comandos de Python. Afortunadamente, no es la única. Para los propósitos del curso, necesitaremos de un Entorno de Desarrollo Integrado

(IDE, por sus siglas en inglés).

1.2.1 Creando un entorno virtual

Antes de continuar e independientemente de tu sistema operativo, es conveniente crear un entorno virtual de Python para evitar cualquier problema de compatibilidad con el sistema operativo y que las cosas funcionen correctamente. Primero debes abrir una nueva terminal de comandos e instalar el paquete `virtualenv`, ejecutando la instrucción:

```
pip install virtualenv
```

Cuando termine, escribe la siguiente instrucción para crear un entorno virtual llamado «omm» (puedes cambiar la palabra `omm` a cualquier otro nombre que prefieras, pero debe ser una única palabra, sin espacios):

```
python3 -m venv omm
```

El comando anterior creó un directorio llamado «omm» (o el nombre que hayas elegido) en el directorio en el que abriste la terminal de comandos. Por defecto, la terminal de comandos se abre en el directorio «`C:\Users\user`» en Windows, y en el directorio «`/home/user/`» en Linux/MacOSX (donde `user` es tu nombre de usuario en tu computadora, independientemente de tu sistema operativo).

Para activar el entorno virtual en Windows, debes ejecutar el siguiente comando:

```
C:\Users\user\omm\Scripts\activate
```

Ten en cuenta que debes cambiar la palabra `user` por tu nombre de usuario.

En cambio, para activarlo en Linux y Mac OSX, se hace con el comando:

```
source ~/omm/bin/activate
```

Como resultado, verás que a la izquierda de tu terminal de comandos aparece el nombre de tu entorno virtual encerrado en paréntesis, esto indica que está activado. Deberás ejecutar este comando para utilizar el entorno virtual de Python cada vez que abras una nueva terminal. Para desactivarlo, debes escribir el comando **deactivate** y ejecutarlo.

1.2.2 El editor Geany

Un programa de Python es simplemente un archivo de texto con extensión .py que puede escribirse en cualquier editor de texto. Existen muchos editores de texto y puedes usar el que prefieras. Recomiendo usar Geany porque es bastante simple, es ligero y fácil de instalar. Además permite ejecutar los programas directamente desde el editor, también utiliza el resaltado de sintaxis y permite usar una terminal integrada para ejecutar los códigos si así lo prefieres.

Geany en Windows: Para instalar Geany en Windows, dirígete a la página <https://www.geany.org/download/releases/> y descarga el instalador para Windows. Una vez que se haya descargado el archivo .exe, haz doble clic en él para iniciar el proceso de instalación. Sigue las instrucciones del asistente de instalación. Puedes dejar las opciones predeterminadas, a menos que deseas personalizar la instalación.

Configuración de Geany en Windows: Una vez instalado, abre Geany desde el menú de aplicaciones. En la ventana del editor, escribe la instrucción `print("¡Hola mundo!")` y utiliza la combinación de teclas **Ctrl+S** para guardar el archivo. Puedes guardar el archivo con el nombre `hello_world.py` en tu carpeta de trabajo. Asegúrate de colocar la extensión .py en el nombre de tu archivo al momento de guardarlo.

Para que Geany funcione correctamente con el entorno virtual, se debe hacer una configuración sencilla. Dentro de la ventana de Geany, dirígete a **Build -> Set Build Commands**. Ahora haz lo siguiente:

1. En el campo llamado «Compile», edita su contenido para que el comando tenga lo siguiente:

```
C:\Users\user\omm\Scripts\python -m py_compile "%f"
```

2. En el campo llamado «Execute», edita su contenido para que el comando tenga lo siguiente:

```
C:\Users\user\omm\Scripts\python "%f"
```

3. Haz clic en OK para guardar los cambios.

Usa las opciones de Build o los atajos de teclado (por ejemplo, F8 para compilar y F5 para ejecutar) para compilar y ejecutar el archivo Python. Verifica que el intérprete de Python utilizado sea el del entorno virtual y no el global.

El resultado debería ser un mensaje que dice «¡Hola mundo!», tal como cuando se ejecutó el comando desde una terminal.

Geany en Linux y Mac OSX: Para instalar Geany en Linux, es suficiente con que ejecutes los siguientes comandos en una terminal:

```
● ● ●  
sudo apt update  
sudo apt install geany
```

En Mac OSX, puedes usar Homebrew para instalar Geany desde una terminal con el comando:

```
● ● ●  
brew install --cask geany
```

La configuración de Geany en Linux y Mac OSX es la misma. Para hacerlo, sigue los pasos descritos en la sección «Configuración de Geany en Windows» con la única diferencia que el campo «Compile» debe contener lo siguiente: /home/user/omm/bin/python -m py_compile "%f" y el campo «Execute» debe tener: /home/user/omm/bin/python "%f".

Ten en cuenta que tanto en Windows como en Linux y Mac OSX, debes reemplazar la palabra «user» por tu nombre de usuario. Utilizaremos Geany principalmente para escribir y editar algunas funciones que utilizaremos. Ahora revisaremos algunos IDEs un poco más avanzados y que permiten hacer más cosas.

1.2.3 Jupyter Notebook

Jupyter Notebook es un IDE de Python basado en navegadores web, que además de ejecutar códigos de Python, permite incluir texto con formato, ecuaciones, imágenes y mucho más. Esencialmente, es un cuaderno donde es posible escribir código y tomar apuntes. Para instalarlo debes abrir una nueva terminal, activar el entorno virtual que creaste anteriormente y ejecutar la instrucción `pip install notebook`. En una terminal se vería así:

```
source ~/omm/bin/activate  
pip install notebook
```

Recuerda utilizar los comandos adecuados si usas Windows. Una vez que termine la instalación, puedes usar Jupyter Notebook escribiendo la instrucción `jupyter notebook` (separado y en minúsculas) en la misma terminal de comandos. Esto abrirá una nueva pestaña en tu navegador predeterminado y te mostrará un explorador de archivos con el contenido de la carpeta donde ejecutaste la instrucción anterior, como se muestra en la Figura 1.1

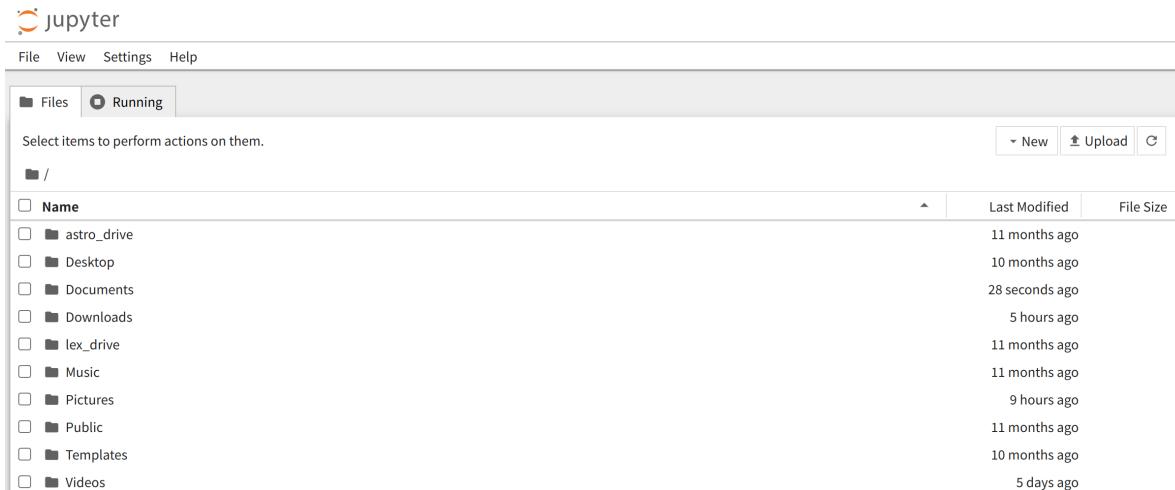


Figura 1.1: Interfaz gráfica de Jupyter Notebook

Desde la interfaz de Jupyter dirígete a tu carpeta de trabajo y crea un nuevo Notebook. Para lograrlo, haz click en el botón «New» en la esquina superior derecha y selecciona la opción «Notebook», luego elige el *kernel* llamado «Python 3 (ipykernel)» y esto abrirá una nueva pestaña en tu navegador.

En esa nueva pestaña, verás que aparecen los caracteres []: (encerrado con un círculo rojo en el panel izquierdo de la Figura 1.2) y a la derecha aparece una celda vacía donde

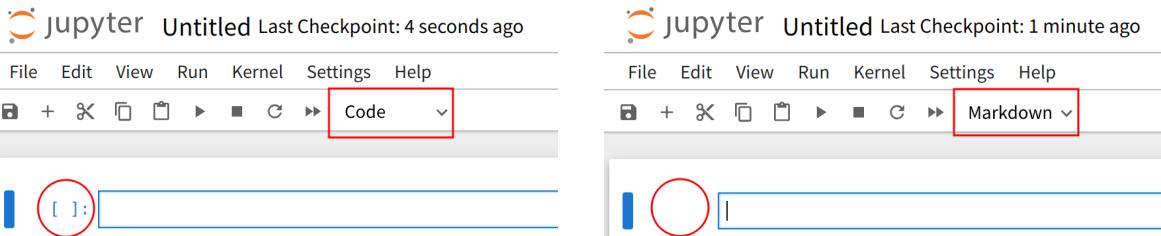


Figura 1.2: Opciones disponibles para escribir en Jupyter

puedes escribir los comandos de Python. Los caracteres []: en Jupyter Notebook son el equivalente a los caracteres >>> en la terminal.

En la parte superior del Notebook hay una barra de menú, que a su vez contiene un botón con la palabra «Code» (encerrada con un rectángulo rojo en el panel izquierdo de la Figura 1.2). Al darle clic se muestra un menú desplegable cuyas opciones son «Code» y «Markdown». Este último es un tipo de *lenguaje de marcas* bastante popular para escribir texto con formato. Si seleccionas «Markdown», los símbolos []: de la celda desaparecen (como se muestra en el panel derecho de la Figura 1.2) y ahora puedes escribir texto, ecuaciones, imágenes y más. Puedes revisar algunas nociones básicas sobre la escritura de texto Markdown en este enlace de [Github Docs](#).

Cuando vas a escribir código, siempre debes verificar que la celda está en modo «Code» o en su defecto, que a la izquierda de la celda aparezcan los símbolos []:, de lo contrario, tus comandos no se ejecutarán. Por ejemplo, da clic en la celda vacía y asegúrate que se encuentre en modo «Code» y escribe el comando `print("¡Hola mundo!")`. Para ejecutar el código de una celda, debes dar click a la celda y usar la combinación de teclas **Shift+Enter**. El resultado debería ser algo similar a lo siguiente:

```
[1]: print("¡Hola mundo!")
```

¡Hola mundo!

Para cerrar correctamente Jupyter Notebook, dirige tu cursor hacia la esquina superior izquierda del Notebook y selecciona **File -> Shutdown**. Una vez que lo hagas, podrás cerrar esa pestaña del navegador. Debes repetir este procedimiento para todas las pestañas del Notebook abiertas. Como último dato sobre Jupyter Notebook, debes saber que aunque necesite de un navegador web para funcionar, en realidad no necesita de una conexión a internet para ejecutar los comandos. El navegador web es únicamente para la interfaz gráfica.

1.2.4 Visual Studio Code y PyCharm

Visual Studio Code (VS Code) y PyCharm son otros IDEs muy populares que también trabajan con Notebooks de Python y también archivos de texto plano con extensión .py. Su interfaz es similar a la de Jupyter Notebook y funcionan de la misma manera. Puedes revisar las instrucciones de instalación y configuración de VS Code y PyCharm en los sitios web oficiales <https://code.visualstudio.com/> y <https://www.jetbrains.com/es-es/pycharm/download/>, respectivamente.

Como última opción, si no te es posible instalar Python en tu computadora, puedes utilizar la opción en línea llamada Google Colaboratory (o simplemente Colab), a la que puedes acceder desde tu cuenta de google drive. La ventaja de usar Colab es que no necesitas instalar nada en tu computadora y puedes usar todos los paquetes de Python que requieras. Sin embargo, sí necesitarás de una conexión a internet para poder utilizarlo.

PyCharm, VS Code, Colab y Jupyter Notebook son totalmente equivalentes entre sí, teniendo diferencias mínimas y por lo tanto puedes utilizar el IDE que sea de tu preferencia. Teniendo esto en cuenta, podemos iniciar con algunas nociones sobre la escritura de código Python usando Notebooks.

1.3 Python como calculadora

1.3.1 Operaciones aritméticas

Una de las formas elementales en las que Python puede ser utilizado es como una calculadora. En este sentido, la Tabla 1.1 muestra las operaciones aritméticas que pueden ejecutarse con Python. Quizá no estés muy familiarizado con las operaciones de división entera (representada con los caracteres //) y división modular (representada con el carácter %), pero las revisaremos con detalle más adelante.

Abre una nueva terminal de comandos, activa tu entorno virtual de Python e inicia una nueva sesión de Jupyter Notebook. Puedes abrir el mismo Notebook que creaste anteriormente o crear uno nuevo. Asegúrate que las celdas estén en modo «Code» e intenta ejecutar los siguientes comandos. El resultado de cada comando se mostrará al lado de los caracteres []: (en color naranja).

Tabla 1.1: Operadores aritméticos

Símbolo/Operador	Significado/Operación
+	Suma
-	Resta
*	Multiplicación
/	División
**	Exponenciación
//	División entera
%	División modular

Veamos cómo funcionan los operadores aritméticos con unos ejemplos sencillos. Comenzamos con una simple, para calcular el valor de la operación $5 + 9$:

[2]: `5 + 9`

[2]: 14

La siguiente expresión es equivalente a $40 - 5 \times 4$:

[3]: `40 - 5*4`

[3]: 20

Y también podemos agrupar términos utilizando los paréntesis. Por ejemplo, podemos calcular el valor de la expresión $(40 - 5 \times 4)/4$ usando la siguiente sintaxis:

[4]: `(40 - 5 * 4) / 4`

[4]: 5.0

Y para calcular el valor de $3^2 + 4^2$ se usa la sintaxis:

[5]: `4**2 + 3**2`

[5]: 25

Ahora revisemos con un poco de detalle el operador `//`, que en Python es el operador de división entera. Este operador divide dos números y redondea el resultado hacia abajo al número entero más cercano. Por ejemplo, sabemos que $7/3 = 2.333$. Si aplicamos el operador de división entera a esos mismos números se obtiene:

[6]: `7 // 3`

[6]: 2

En este caso, `7 // 3` es igual a 2 porque la parte fraccionaria (0.333...) se descarta. Ahora veamos cómo se comporta este operador con números negativos:

[7]: `-7 // 3`

[7]: -3

En este caso, `-7 // 3` es igual a `-3` porque el resultado se redondea hacia abajo (es decir, al entero más negativo).

Por otro lado, el operador `%` en Python es el operador de división modular (también conocido como resto o residuo). Este operador devuelve el resto de la división entre dos números. Es útil para determinar si un número es divisible por otro, o para obtener la parte sobrante después de dividir un número por otro. Veamos cómo funciona con ejemplos:

[8]: `7 % 3`

[8]: 1

En este caso, `7 % 3` es igual a 1 porque 7 dividido entre 3 es igual a 2 con un residuo de 1.

1.3.2 Operaciones matemáticas avanzadas

Para realizar operaciones matemáticas más avanzadas, necesitaremos utilizar un *módulo* de Python llamado `math`. Un módulo es un archivo que contiene definiciones y declaraciones de Python, como funciones, variables y clases. Los módulos permiten organizar y reutilizar código.

Para usar el módulo `math`, debemos importarlo en nuestro código. La forma más sencilla de hacerlo es usando la palabra clave «`import`». Dentro del módulo `math` están definidas muchas funciones matemáticas como la raíz cuadrada, logaritmos, funciones trigonométricas y mucho más. Para usarlas, debemos hacerlo con el operador de acceso, que en Python es un punto. La siguiente celda de código muestra cómo importar el módulo `math` y cómo acceder a la función raíz cuadrada, definida como `sqrt` dentro de dicho módulo:

```
[9]: import math  
print(math.sqrt(16))
```

4.0

También podemos hacer importaciones específicas de las funciones dentro del módulo. Para esto usamos la palabra «**from**». El siguiente ejemplo muestra cómo importar la función `sqrt` definida dentro del módulo `math`:

```
[10]: from math import sqrt  
print(sqrt(16))
```

4.0

En este último caso, no fue necesario escribir `math.sqrt` para calcular la raíz cuadrada, porque importamos directamente la función `sqrt`.

Adicionalmente, puedes importar módulos o funciones específicas y renombrarlas a tu gusto usando la palabra «**as**». Los siguientes dos ejemplos muestran cómo funciona esta sintaxis.

```
[11]: import math as m  
print(m.sqrt(16))
```

4.0

```
[12]: from math import sqrt as raiz_cuadrada  
print(raiz_cuadrada(16))
```

4.0

Como norma general, se recomienda hacer todas las importaciones al inicio del programa y utilizar alias claros y fácilmente identificables.

Clase 2 | Variables y contenedores

2 Introducción

En cualquier lenguaje de programación, las variables actúan como nombres simbólicos que almacenan datos, permitiéndonos utilizar la información de manera eficiente a lo largo de nuestros programas. Los contenedores, por otro lado, son estructuras de datos que permiten agrupar múltiples valores en una sola entidad, facilitando la organización y gestión de conjuntos de datos. A lo largo de esta lección, aprenderemos cómo declarar variables, entenderemos los tipos de datos básicos y profundizaremos en los contenedores más utilizados en Python, como listas, tuplas, conjuntos y diccionarios.

2.1 Variables

Las variables son parte fundamental en cualquier lenguaje de programación y constan simplemente de dos partes: el nombre de la variable y su valor. Ambas partes están separadas por el signo de igual (=). Es decir, se escribe el nombre de la variable a la izquierda del símbolo = y su valor se escribe a la derecha. Si se quisiera por ejemplo definir una variable llamada var_1 y asignarle el valor numérico 53, se haría de la siguiente manera:

```
[1]: var_1 = 53
```

Los nombres de las variables pueden contener letras minúsculas, mayúsculas, dígitos (0 - 9) y guiones bajos (_). Sin embargo, los nombres no pueden empezar con dígitos; deben iniciar necesariamente con una letra o un guion bajo. Por ejemplo, los nombres «_var», «_var1» o «_var_1» son válidos, pero «1_var» y «1var» son nombres no permitidos y devuelven un error, como puedes verificar con el siguiente ejemplo:

```
[2]: 1var = 53
```

```
Cell In[1], line 1
1var = 53
^
SyntaxError: invalid decimal literal
```

El mensaje de error nos indica que estamos colocando un valor numérico en una posición inválida al momento de definir la variable.

2.1.1 Variables tipo int/float

En Python se distinguen dos tipos principales de variables numéricas: las variables tipo **int**, que corresponden a los números enteros y las variables tipo **float**, que corresponden a números decimales o muy grandes escritos con notación científica. Por ejemplo, la variable definida anteriormente y que llamamos `var_1` es de tipo **int**. En cambio, los valores numéricos 1.5 y 4×10^3 son de tipo **float**. En realidad, la diferencia entre variables tipo **int** y **float** es más complicada que simplemente eso, pero no entraremos en detalles. En el siguiente ejemplo se definen dos variables tipo **float**.

```
[3]: var_2 = 1.5  
      var_3 = 4e3
```

Nota cómo la expresión 4×10^3 se sustituye por `4e3`, haciendo muy fácil escribir expresiones con valores numéricos grandes. Las variables **int** y **float** admiten las operaciones aritméticas vistas en la Clase 1. Al realizar una operación aritmética entre un **int** y un **float**, el resultado será un **float**.

2.1.2 Variables de tipo string

Otro tipo de variables son las de tipo *string* (o cadena de caracteres), que se caracterizan porque son un conjunto de palabras o caracteres formando un mensaje, tales como «¡Hola mundo!». En Python, estas variables se identifican como de tipo **str** y para definirlas se debe escribir el mensaje que queramos dentro de comillas simples (`'`) o comillas dobles (`"`). Esto se muestra en el siguiente ejemplo:

```
[4]: message_1 = 'Hola'  
      message_2 = "y buenos días"
```

Es posible aplicar las operaciones aritméticas de suma y multiplicación a las variables **str**. Al aplicar la suma a las variables `message_1` y `message_2` se forma una nueva oración. Intenta ejecutar las siguientes dos celdas de código:

```
[5]: message_4 = message_1 + message_2  
print(message_4)
```

Holay buenos días

```
[6]: message_5 = message_1 + ' ' + message_2  
print(message_5)
```

Hola y buenos días

La diferencia entre las variables `message_4` y `message_5` es que a `message_5` se le agregó explícitamente un espacio en blanco (' ') al momento de sumar las variables `message_1` y `message_2`. Las variables `str` no pueden sumarse con variables tipo `int` ni `float`.

Por otro lado, la multiplicación de una variable `str` y una de tipo `int` sí está permitida. Intenta ejecutar lo siguiente:

```
[7]: message_1 * 3
```

[7]: HolaHolaHola

En resumen, se multiplicó `message_1` por el número `3`, y el mensaje apareció tres veces. Las variables `str` no pueden ser multiplicadas por variables tipo `float`

2.1.3 Comentarios

Los comentarios son parte esencial de cualquier lenguaje de programación. Se trata de líneas de código que son ignorados por el intérprete/compilador y sirven para documentar el código. Mientras más líneas de código escribas, más difícil será recordar qué hace cada línea o bloque de código. Debido a esto se hacen necesarios los comentarios, y en Python hay dos formas de escribirlos. La primera es usando el símbolo de numeral: `#`. Cualquier cosa que escribamos después del símbolo será ignorada por Python:

```
[8]: pi = 3.14159265      #- Una variable float  
  
num = 5                  #- Una variable int  
  
message = "nueve"       #- Una variable str
```

En la primera línea del ejemplo anterior, se declaró una variable llamada `pi` y se le asignó un valor de `3.14159265` y seguidamente se escribió un comentario para aclarar el tipo de variable que se está definiendo. Esto se repitió para las variables `num` y `message`.

La otra forma de escribir comentarios consisten en colocarlos dentro de triples comillas simples (''' ''') o triples comillas dobles (""" """). Las triples comillas permiten que el texto entre ellas abarque más de una sola línea. En realidad, estas son un tipo especial de comentarios que se utilizan para documentar las funciones, métodos, clases o módulos y por lo tanto son llamadas *docstrings* (cadenas de documentación).

2.1.4 Variables tipo bool

Las variables tipo bool, o variables booleanas son aquellas que solo pueden tomar dos posibles valores: `True` (verdadero) o `False` (falso), a las cuales corresponden los valores numéricos de `0` y `1`, respectivamente. Este tipo de variables aparecen cuando se hacen comparaciones entre una o más variables. Para eso se utilizan operadores de comparación, de los cuales, los más comunes se muestran en la Tabla 2.1

Tabla 2.1: Operadores de comparación

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
==	Igual que
!=	Diferente de

Por ejemplo, sabemos que el número `3` es menor que el número `5`. Entonces la expresión `3 > 5` es falsa, como puedes comprobarlo:

```
[9]: 3 > 5
```

```
[9]: False
```

Estos operadores también pueden ser acompañados por las palabras clave definidas en Python: `and` y `or`. Intenta ejecutar los siguientes ejemplos:

```
[10]: 5 <= 2 and 4 > 2
```

```
[10]: False
```

Veamos con detalle qué es lo que está pasando en este ejemplo. El número `5` es mayor que `2`, por lo tanto, la expresión `5 <= 2` es falsa. Por otro lado, el número `4` es mayor que `2`, por lo tanto, la expresión `4 > 2` es verdadera. En otras palabras, la expresión es equivalente

a: **False and True**. El operador **and** devuelve un valor verdadero únicamente si todas las expresiones son verdaderas, de lo contrario devuelve un valor falso. Ya que en este caso una de las expresiones es falsa, entonces el resultado es **False**. Ahora revisa el siguiente ejemplo:

```
[11]: 5 <= 2 or 4 > 2
```

```
[11]: True
```

Estamos haciendo las mismas comparaciones que en el ejemplo anterior, pero ahora con el operador **or**, el cual devuelve un valor falso únicamente si todas las expresiones son falsas, de lo contrario devuelve un valor verdadero. En este caso, la expresión de la derecha es verdadera y por lo tanto, el resultado es un valor **True**.

Las variables booleanas se utilizan para iniciar, terminar o repetir algún proceso dependiendo si alguna condición se cumple o no. Esto lo veremos más adelante cuando trabajemos con controladores de flujo.

2.2 Contenedores

Los contenedores son un tipo de estructuras de datos que se utilizan para almacenar múltiples elementos o variables bajo un solo nombre. A diferencia de las variables, que pueden contener un único valor a la vez, los contenedores permiten mantener una colección de valores, los cuales a su vez pueden ser otros contenedores. Los contenedores son fundamentales para manejar datos de manera eficiente en Python, ya que permiten agrupar, iterar, acceder y manipular conjuntos de valores de manera ordenada y flexible. Los contenedores disponibles en python son las listas, tuplas, conjuntos y diccionarios.

2.2.1 Listas

Las listas son secuencias ordenadas de elementos o variables de cualquier tipo, que no tienen que estar relacionadas entre sí. Para definir una lista se colocan los elementos separados por comas y encerrados entre corchetes. El siguiente ejemplo muestra que una lista puede contener cualquier tipo de variable:

```
[12]: lista_1 = [1, 'Uno', True]
print(lista_1)
```

```
[1, 'Uno', True]
```

También es posible que una lista contenga a otras listas:

```
[13]: lista_2 = [lista_1, [0, 'Cero', False]]
```

```
[[1, 'Uno', True], [0, 'Cero', False]]
```

Además, el operador numérico de la suma está permitido en las listas. El resultado es una nueva lista:

```
[14]: lista_3 = lista_1 + lista_2
print(lista_3)
```

```
[1, 'Uno', True, [1, 'Uno', True], [0, 'Cero', False]]
```

Como se mencionó, las listas son una colección ordenada de elementos. Debido a esta característica, sus elementos pueden ser accedidos mediante índices, que corresponden a la posición del elemento en la lista. Sin embargo, se debe tener en cuenta que Python es un lenguaje de programación que comienza a contar desde el número cero. Es decir, al primer elemento le corresponde la posición (o índice) `0`, al segundo elemento la posición `1` y así sucesivamente. Por ejemplo, definamos una nueva lista con los nombres de algunos astrónomos:

```
[15]: astronomers = ['Carl Sagan', 'Stephen Hawking', 'Jocelyn Bell']
```

Para acceder a un elemento de la lista `astronomers`, se escribe el nombre de la lista y se acompaña del índice encerrado entre corchetes y sin espacios. Específicamente, para acceder a `'Carl Sagan'`, que es el primer elemento, lo hacemos de la siguiente manera:

```
[16]: astronomers[0] # Selecciona el primer elemento
```

```
[16]: 'Carl Sagan'
```

El último elemento de cualquier lista, independiente de cuántos elementos tenga, puede ser accedido mediante el índice `-1`, como puedes verificar:

```
[17]: astronomers[-1] # Selecciona el último elemento
```

```
[17]: 'Jocelyn Bell'
```

Una de las características de las listas es que sus elementos pueden ser modificados. En un contexto de programación, esto significa que las listas son *mutables*. Por ejemplo, para reemplazar el primer elemento de la lista por el nombre algún otro astrónomo, podemos hacerlo de la siguiente manera:

```
[18]: astronomers[0] = 'Johannes Kepler'  
print(astronomers)
```

```
['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell']
```

Puedes agregar elementos a una lista usando el método `list.append()` (el cual permite agregar un elemento a la vez) y el método `list.extend()` (el cual permite agregar varios elementos a la vez, en formato de lista). Ambos métodos agregan los elementos al final de la lista. Por ejemplo, para agregar nuevamente a Carl Sagan a la lista de astrónomos, podemos hacerlo de la siguiente manera:

```
[19]: astronomers.append('Carl Sagan')  
print(astronomers)
```

```
['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell', 'Carl Sagan']
```

Podemos en cambio, agregar dos elementos al mismo tiempo de la siguiente manera:

```
[20]: astronomers.extend(['Henrietta Leavitt', 'Vera Rubin'])  
print(astronomers)
```

```
['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell', 'Carl Sagan',  
'Henrietta Leavitt', 'Vera Rubin']
```

Algunos tipos de variables permiten ser convertidos a listas, como es el caso de las variables de tipo `str`. Para lograrlo, se utiliza la función `list()` predefinida en Python. Revisa el siguiente ejemplo:

```
[21]: letters = list(astronomers[2])  
print(letters)
```

```
['J', 'o', 'c', 'e', 'l', 'y', 'n', ' ', 'B', 'e', 'l', 'l']
```

Adicionalmente, las variables de tipo `str` también admiten acceder a sus elementos mediante índices:

```
[22]: name = 'Jocelyn'  
name[0]
```

```
[22]: 'J'
```

También puedes definir listas que sean una porción de otra lista. A esto se le conoce como «*rebanar*» o en inglés, «*slice*». Supongamos por ejemplo, que solo queremos seleccionar los primeros 3 elementos de la lista `astronomers`, entonces usamos la siguiente sintaxis:

```
[23]: new_astronomers = astronomers[0:3]  
print(new_astronomers)
```



```
['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell']
```

El ejemplo anterior está seleccionando los elementos de la lista que van desde el índice `0` (primer elemento) hasta el índice `3` (cuarto elemento). Por defecto en Python, el índice después de los dos puntos (`:`) no se incluye, por eso la lista `new_astronomers` solo tiene tres elementos y no cuatro.

2.2.2 Tuplas

Las tuplas representan la versión inmutable de las listas, es decir, pueden almacenar cualquier tipo de elementos pero sus valores no se pueden modificar. Para definirlas, basta con escribir los elementos separados por comas sin encerrarlos:

```
[24]: planets_1 = 'Mercurio', 'Venus', 'Tierra'  
planets_1
```

```
[24]: ('Mercurio', 'Venus', 'Tierra')
```

Alternativamente, las tuplas se definen al escribir los elementos separados por comas y encerrarlos entre paréntesis:

```
[25]: planets_2 = ('Marte', 'Júpiter', 'Saturno')  
planets_2
```

```
[25]: ('Marte', 'Júpiter', 'Saturno')
```

Al igual que en el caso de las listas, se puede acceder a los elementos de una tupla mediante índices y se puede convertir una variable de tipo `str` a una tupla usando la función `tuple()`. usando esa misma función, también puedes convertir una lista en una tupla.

2.2.3 Conjuntos

Los conjuntos (en inglés llamados «sets») son colecciones de datos no ordenados y además no permiten elementos duplicados. Esto significa que no es posible acceder a sus elementos mediante índices y que ninguno de sus elementos se repite. Los conjuntos se definen con sus elementos separados por comas y encerrados entre llaves:

```
[26]: my_set = {1,2,3,4,5,5,5}  
print(my_set)
```

```
{1, 2, 3, 4, 5}
```

También puedes definirlos con la función `set()`:

```
[27]: my_set = set([1,2,3,4,5,5,5])
```

```
{1, 2, 3, 4, 5}
```

Los conjuntos también son contenedores mutables. Para añadir elementos se utiliza la función `add()`:

```
[28]: my_set.add(6)  
print(my_set)
```

```
{1, 2, 3, 4, 5, 6}
```

Para eliminar un elemento se utilizan los métodos `remove()` o `discard()`, como puedes comprobar:

```
[29]: my_set.remove(3)  
print(my_set)
```

```
{1, 2, 4, 5, 6}
```

```
[30]: my_set.discard(2)  
print(my_set)
```

```
{1, 4, 5, 6}
```

La diferencia en ambos métodos es que `remove()` lanzará un error si el elemento no existe, mientras que `discard()` no hará nada si el elemento no está presente.

También puedes usar el operador `in` para verificar si un elemento está en el conjunto. Ten en cuenta que este operador también puede utilizarse para verificar si un elemento está en una lista o en una tupla. Veamos un ejemplo solo para los conjuntos:

```
[31]: print(2 in my_set)
```

```
False
```

Otras operaciones disponibles para los conjuntos son:

- **Unión:** combina todos los elementos de dos conjuntos, eliminando los duplicados, mediante el operador «|».
- **Intersección:** Devuelve los elementos comunes en ambos conjuntos, mediante el operador «&».
- **Diferencia:** Devuelve los elementos que están en el primer elemento pero no en el segundo, mediante el operador «-».
- **Diferencia simétrica:** Devuelve los elementos que están en un set o en otro, pero no en ambos, mediante el operador «^».

2.2.4 Diccionarios

Los diccionarios son estructuras de datos no ordenados y están compuestos de pares *clave-valor* (en inglés *key-values*), donde cada valor está asociado a una clave única. Para crear un diccionario se escriben los pares de clave-valor separados por comas y encerrados con llaves {}, o encerrados por la función `dict()`. Para entenderlo mejor, revisa el siguiente ejemplo:

```
[32]: stephen = {
    "first_name": "Stephen",
    "last_name": "Hawking",
    "birthday": (8, 1, 1942)
}
print(stephen)
```

```
{'first_name': 'Stephen', 'last_name': 'Hawking', 'birthday': (8, 1, 1942)}
```

```
[33]: albert = dict(
    first_name="Albert",
    last_name='Einstein',
    birthday= (14, 3, 1879)
)
print(albert)
```

```
{'first_name': 'Albert', 'last_name': 'Einstein', 'birthday': (14, 3, 1879)}
```

Las dos sintaxis anteriores son totalmente equivalentes para definir un diccionario. Para acceder a cualquier valor de un diccionario, se escribe el nombre del diccionario y en seguida, encerrado entre corchetes se escribe su clave correspondiente. Por ejemplo:

```
[34]: stephen['first_name']
```

```
[34]: 'Stephen'
```

Los diccionarios son contendores mutables, por lo tanto sus valores pueden cambiar, pero no sus claves. Puedes agregar un par clave-valor simplemente asignando un valor a una nueva clave:

```
[35]: stephen['nationality'] = "British"
print(stephen)
```

```
{'first_name': 'Stephen', 'last_name': 'Hawking', 'birthday': (8, 1, 1942), 'nationality': 'British'}
```

Puedes obtener todas las claves, valores, o pares clave-valor usando los métodos `keys()`, `values()` y `items()`, respectivamente:

```
[36]: albert.keys()
```

```
[36]: dict_keys(['first_name', 'last_name', 'birthday'])
```

```
[37]: albert.values()
```

```
[37]: dict_values(['Albert', 'Einstein', (14, 3, 1879)])
```

```
[38]: albert.items()
```

```
[38]: dict_items([('first_name', 'Albert'), ('last_name', 'Einstein'), ('birthday', (14, 3, 1879))])
```

Clase 3 | Control de flujo y lógica

3 Controles de flujo

Los controles de flujo en un lenguaje de programación permiten que ciertas partes de un código se ejecuten y otras no, dependiendo de si se cumplen o no algunas condiciones. Adicionalmente, también permiten ejecutar líneas de código una y otra vez mientras alguna condición siga siendo válida. Debido a esto, los principales controles de flujo y lógica son llamados *condicionales* y *bucles* (en inglés, *loops*). En esta clase revisaremos ambos conceptos.

3.1 Condicionales

Son el tipo de controladores de flujo más sencillos, su funcionamiento puede describirse de la siguiente manera: "Si la variable *x* es verdadera, entonces realiza una tarea; en caso contrario, realiza esta otra tarea". Para escribir este tipo de condiciones se utilizan las palabras en inglés «if» y «else», que significan «si» y «en caso contrario», respectivamente.

3.1.1 Declaraciones if

La forma más sencilla de usar los condicionales es cuando solo se quiere comprobar si alguna condición es verdadera y no hacer nada si la condición es falsa. En ese caso, se utiliza la palabra clave **if** una única vez. La sintaxis es la siguiente:

```
if <condition>:  
    <if-block>
```

Para este tipo de declaraciones, la condición, identificada como `<condition>` puede ser cualquier tipo de expresión de comparación o lógica que devuelva un valor **True** o **False**. Además, inmediatamente después de dicha condición se escriben dos puntos (:). El segmento `<if-block>` representa aquellos comandos o instrucciones que se ejecutarán únicamente si la condición es verdadera.

Algo muy importante es que los comandos del <**if**-block> deberán ser escritos abajo del «**if** <condition>:» con una sangría (en inglés «*indent*») de cuatro espacios. En Python, esta sangría es obligatoria siempre que se quiera separar bloques de código del resto de líneas de código. De no usar la sangría de cuatro espacios en los condicionales, el código generará un error. Como ejemplo, intenta ejecutar la siguiente celda de código:

```
[1]: #- Definiendo un número
number = 0
if number == 0.0:
    print(f"El número {number} es igual a 0.0")
```

```
Cell In[1], line 3
print(f"El número {number} es igual a 0.0")
^
```

```
IndentationError: expected an indented block after 'if' statement on
line 2
```

Este error es bastante simple. Python nos está diciendo que se esperaba un bloque con sangría después de la declaración **if**. Puedes corregirlo simplemente agregando la sangría de cuatro espacios:

```
[2]: #- Definiendo un número
number = 0
if number == 0.0:
    print(f"El número {number} es igual a 0.0")
```

```
[2]: El número 0 es igual a 0.0
```

La condición `number == 0.0` es verdadera y por lo tanto se mostró el mensaje. Si la condición hubiera sido falsa, no se habría mostrado ningún mensaje. Nota cómo se definió a la variable `number` como de tipo **int** y en la condición se comparó con una variable de tipo **float**, de modo que la condición sería equivalente a `0 == 0.0`. Esta condición resulta ser cierta porque el operador `==` únicamente compara si dos valores son equivalentes.

3.1.2 Declaraciones if-else

Cada bloque **if** puede ir seguido de un bloque opcional **else**, que se ejecutará únicamente si la primera condición es falsa. El bloque **else** no necesita de ninguna condición pero sí de los dos puntos (`:`) y la línea siguiente también debe tener sangría de cuatro espacios. La sintaxis para este tipo de declaraciones es la siguiente:

```
if <condition>:  
    <if-block>  
  
else:  
    <else-block>
```

Por ejemplo, podemos escribir un código que verifique si un número es par o impar, comprobando si el número es divisible por 2. Esto se puede hacer con una declaración **if-else** de la siguiente manera:

```
[3]: #- Definiendo un número  
number = 15  
  
#- Comprobando si es par o impar  
if number % 2 == 0:  
    result = "par"  
else:  
    result = "impar"  
  
# Muestra un mensaje con el resultado  
print(f"El número {number} es {result}.")
```

El número 15 es impar.

El número 15 no es divisible entre dos, por lo tanto la condición `number % 2 == 0` es falsa. Como resultado el bloque `result = "par"` no se ejecuta. En cambio, se ejecuta el bloque `result = "impar"` y se muestra el mensaje "El número 15 es impar".

3.1.3 Declaraciones if-elif-else

En muchas ocasiones será necesario comprobar más de dos posibles situaciones, y para esto se utiliza la sintaxis **if-elif-else**. La palabra **elif** es una abreviación de «*else if*» en inglés y en las declaraciones pueden haber cuantos bloques **elif** sean necesarios. El primer bloque que devuelva un valor **True** será ejecutado, y ninguno de los restantes será comprobado ni ejecutado. La sintaxis de este tipo de declaraciones es la siguiente:

```
if <condition>:  
    <if-block>  
  
elif <condition1>:  
    <elif-block1>
```

```

elif <condition2>:
    <if-block2>

else:
    <else-block>

```

El siguiente ejemplo muestra cómo usar las declaraciones **if-elif-else** para clasificar a una partícula fundamental de acuerdo a su masa:

```
[4]: # Masa de la partícula hipotética
mass = 5.0 # en GeV/c^2

# Clasificación basada en su masa
if mass < 0.1:
    classification = "Mesón Ligero"

elif 0.1 <= mass < 1.0:
    classification = "Mesón Pesado"

elif 1.0 <= mass < 10.0:
    classification = "Barión"

else:
    classification = "Partícula Exótica"

# Mostrar su clasificación
print(f"La partícula con masa {mass} GeV/c^2 es un {classification}.")
```

[4]: La partícula con masa 5.0 GeV/c² es un Barión.

Nuevamente, ten en cuenta que en cada bloque **if-elif-else** se necesita colocar una sangría obligatoria.

3.1.4 Expresiones if-else

Para finalizar con los condicionales, es posible aplicar la sintaxis **if-else** en una simple expresión de una línea (sin los bloques de sangría). Se necesita de tres elementos y por lo tanto se le conoce como un operador condicional ternario. La sintaxis es la siguiente:

```
x if <condition> else y
```

Si la condición es verdadera, entonces el resultado de la expresión anterior es x, si es

falsa, el resultado es `y`. Esta expresión resulta muy útil para asignar valores a una variable con base en alguna condición. Esto se verifica en el siguiente ejemplo.

```
[5]: #-Definiendo un número
number = -5

#- Comprobar si es positivo o negativo y guardar el resultado
result = "positivo" if number > 0 else "negativo"

#- Mostrar el resultado
print(f"El número {number} es {result}.")
```

[5]: El número -5 es negativo.

En este ejemplo, el operador ternario verifica si la variable `number` es mayor que cero. Si de hecho lo es, le asigna el valor "`positivo`" a la variable `result`, en caso contrario le asigna el valor "`negativo`".

3.2 Bucles

Como se ha visto, los condicionales permiten que los bloques de código con sangría se ejecuten una única vez dependiendo de alguna condición. Los bucles, en cambio, permiten que el mismo bloque se ejecute muchas veces. Los tipos de bucles disponibles en Python son el bucle `while`, el bucle `for` y algo que es conocido como «contenedores por comprensión».

3.2.1 Bucle while

Los bucles `while` se relacionan con los condicionales porque permiten que el bloque de código se siga ejecutando mientras alguna condición sea verdadera. Su sintaxis es muy similar a la de las declaraciones `if`:

```
while <condition>:
    <while-block>
```

Para este tipo de declaraciones también se deben aplicar los dos puntos luego de la condición y el bloque de instrucciones también debe tener sangría. La condición es evaluada al inicio de cada iteración y si es verdadera, el bloque se ejecuta. Si la condición es falsa, el bloque es ignorado y el programa continúa.

El ejemplo más simple usando un bucle while consiste en generar una cuenta regresiva comenzando en un número entero arbitrario:

```
[6]: #- Número inicial
count = 5

#- Cuenta regresiva
while count > 0:
    print(count)
    count -= 1

print("¡Fin!") # Se muestra cuando el bucle termina
```

```
5
4
3
2
1
¡Fin!
```

En este ejemplo particular, sucede lo siguiente:

- Se inicia con la variable count igual a 5
- El bucle while se ejecuta siempre y cuando count sea mayor que cero
- En cada iteración se imprime el valor actual de count y posteriormente su valor disminuye en una unidad
- Eventualmente, la variable count toma el valor de 0 y la condición se vuelve falsa
- Cuando el bucle termina, se imprime el mensaje "¡Fin!"

Es muy común cometer errores en los bucles **while**, el más frecuente es iniciar por accidente un bucle infinito. Esto sucede cuando la condición evaluada es siempre verdadera. Si en el ejemplo anterior no hubiésemos escrito la instrucción `count -= 1`, la variable count hubiera tenido siempre un valor igual a 5 y por lo tanto el bucle nunca habría finalizado. Al iniciar un bucle **while** debemos asegurarnos de agregar una línea de código que haga que la condición sea falsa en algún momento. Si por algún motivo inicias un bucle infinito, siempre puedes detenerlo con la combinación de teclas **Ctrl+C**.

También es posible terminar con el bucle **while** en cualquier momento, haciendo uso de la instrucción **break**. Por ejemplo, si quisieramos encontrar el primer número par de una lista, podríamos hacerlo con el bucle **while** de la siguiente manera:

```
[7]: #- Lista de números
numbers = [3, 7, 12, 5, 8, 10, 15]

#- Variable para guardar el número par
first_even = None

#- Bucle while
index = 0
while index < len(numbers):
    if numbers[index] % 2 == 0:
        first_even = numbers[index]
        break # El bucle termina al encontrar el primer número par
    index += 1

#- Imprime el valor encontrado
if first_even is not None:
    print(f"El primer número par encontrado fue {first_even}.")
else:
    print("No hay números pares en la lista.")
```

[7]: El primer número par encontrado fue 12.

3.2.2 Bucle for

Aunque los bucles `while` son bastante útiles para repetir declaraciones, en general resulta de mayor utilidad el iterar sobre un contenedor (como las listas, tuplas, conjuntos y diccionarios) o algún otro tipo de variable iterable (como las cadenas de caracteres). En estos casos, se toma un elemento del contenedor en cada iteración y el bucle termina cuando ya no hay más elementos. Esto se logra usando un bucle `for`. La sintaxis para los bucles `for` es diferente a la del bucle `while`, ya que no depende de ninguna condición y consta de lo siguiente:

```
for <variable> in <iterable>:
    <for-block>
```

De nuevo, necesitamos escribir los dos puntos para especificar cuándo inicia el bloque de código que se va a repetir, que a su vez debe tener sangría. En la sintaxis anterior, `<variable>` es el nombre de una variable que se asigna a un elemento cada vez que se ejecuta el bucle. El `<iterable>` es cualquier objeto que pueda devolver elementos. Todos los tipos de contenedores vistos en la Clase 2 son iterables. La cuenta regresiva puede reescribirse de la siguiente manera usando el bucle `for`:

```
[8]: for count in [5,4,3,2,1]:  
    print(count)  
  
    print("¡Fin!")
```

```
5  
4  
3  
2  
1  
¡Fin!
```

La instrucción **break** también se puede usar con los bucles **for**, y funcionan de la misma manera. Adicionalmente, también es posible utilizar la instrucción **continue**, que ignora el bloque de código únicamente en la iteración actual y luego continua con la siguiente. Por ejemplo, el siguiente ejemplo muestra cómo ignorar los números impares en una lista:

```
[9]: for num in [1, 2, 3, 4, 5, 6, 7, 8, 9]:  
    if num % 2 != 0:  
        continue # Ignorar y continuar con la siguiente iteración  
    print(num)  
  
print("¡Listo!")
```

```
2  
4  
6  
8  
¡Listo!
```

La instrucción **continue** también se puede utilizar en los bucles **while**.

Para usar un bucle **for** con una cadena de caracteres se procede de la misma forma que con las listas. El resultado es que se itera por cada una de las letras, por ejemplo:

```
[10]: for letter in "Hola":  
    print(letter)
```

```
H  
o  
l  
a
```

También es posible aplicar un bucle **for** en conjuntos y diccionarios, pero recuerda que este tipo de contenedores no está ordenado, así que el resultado podría no verse como esperarías. Para ilustrar esto se muestra el siguiente ejemplo:

```
[11]: # Iteración sobre un conjunto
for key in {'a', 'b', 'c', 'd', 'e'}:
    print(key)
```

b
c
f
d
a
e

El orden en que se muestra el resultado no es necesariamente al orden en el que se escribieron las letras "a", "b", "c", "d", "e" y "f". Como último ejemplo en esta sección, se muestra cómo puede usarse el ciclo `for` para iterar sobre los pares de clave-valor de un diccionario.

```
[12]: #- Definiendo el diccionario
d = {'a': 1, 'b': 2, 'c': 3}
```

```
[13]: #- Iterar sobre las claves
print("Keys:")
for key in d.keys():
    print(key)
print("=====")
```

Keys:
a
b
c
=====

```
[14]: #- Iterar sobre los valores
print("Values:")
for value in d.values():
    print(value)
print("=====")
```

Values:
1
2
3
=====

```
[15]: #Iterar sobre Los pares clave-valor
print("Items")
for key, value in d.items():
    print(f"Key: {key}, Value: {value}")
```

```
Items
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
```

3.2.3 Contenedores por comprensión

Como se ha visto, los ciclos `while` y `for` permiten hacer muchas cosas, pero para usarlos se necesita de al menos dos líneas de código: una para definir el bucle y la otra para especificar las acciones. Por ejemplo, supongamos que tenemos una lista con los nombres de los primeros cuatro planetas escritos en minúscula, y también tenemos una lista vacía en la que deseamos agregar esos mismos nombres pero en mayúscula:

```
[16]: #- Lista con planetas
planetas = ['mercurio', 'venus', 'tierra', 'marte']

#- Lista vacía
planetas_mayus = []
```

Para lograrlo podemos aplicar el método `list.upper()` a cada elemento de la lista `planetas` y luego anexarlos a la lista `planetas_mayus` en un bucle `for`:

```
[17]: for planet in planetas:
    planetas_mayus.append(planet.upper())

print(planetas_mayus)

['MERCURIO', 'VENUS', 'TIERRA', 'MARTE']
```

Lo anterior puede realizarse de manera equivalente en una sola línea gracias a la sintaxis de contenedores por comprensión, que puede aplicarse tanto a listas como a diccionarios. La sintaxis en cada caso es:

```
# Lista por comprensión
[<expr> for <loop-var> in <iterable>]

# Diccionario por comprensión
{<key-expr>: <value-expr> for <loop-var> in <iterable>}
```

Así, volviendo al ejemplo de los planetas, se puede crear la lista de la siguiente manera:

```
[18]: planetas_mayus = [planet.upper() for planet in planetas]
```

Para el caso de un diccionario, supongamos que tenemos una lista de números y queremos crear una nueva lista con el cuadrado de esos números, entonces:

```
[19]: #- Definiendo una Lista de números
numbers = [1, 10, 12.5, 65, 88]

#- Creando un diccionario por comprensión
results = {x: x**2 for x in numbers}

#- Mostrar el resultado
print(results)

{1: 1, 10: 100, 12.5: 156.25, 65: 4225, 88: 7744}
```

También es posible realizar filtros a los contenedores por comprensión, con ayuda de condicionales. La sintaxis es la siguiente:

```
[20]: # Lista por comprensión con filtro
[<expr> for <loop-var> in <iterable> if <condition>]

# Diccionario por comprensión con filtro
{<key-expr>: <value-expr> for <loop-var> in <iterable> if <condition>}
```

En este caso, si la condición resulta ser verdadera, entonces se agrega el elemento actual, en caso contrario se ignora y se evalúa el siguiente iterable. En este último ejemplo, se creará una lista con los planetas escritos en mayúscula si el nombre inicia con «m», y se calcularán los cuadrados de los números si el número es par:

```
[21]: #- Planetas que inician con M
new_list = [planet.upper() for planet in planetas if planet[0] == 'm']
print(new_list)

['MERCURIO', 'MARTE']

[22]: #- Cuadrado de números pares
results_even = {x: x**2 for x in numbers if x%2 ==0}
print(results_even)

{10: 100, 88: 7744}
```

Clase 4 | Funciones

4 Funciones en Python

Una función en Python es un bloque de declaraciones (comandos) que realizan una tarea específica. Es posible que dicha tarea necesite ejecutarse en repetidas ocasiones para diferentes valores de entrada en un mismo código. El propósito de las funciones es, que en lugar de escribir el mismo código múltiples veces para diferentes entradas, podamos realizar llamadas a funciones y así reutilizar el código contenido en ella una y otra vez. En esta clase aprenderás a escribir funciones y comprenderás por qué utilizarlas hace que los programas sean más fáciles de escribir, leer y corregir.

4.1 Funciones predefinidas

En Python existen funciones predefinidas, las cuales están disponibles para cualquier usuario. La más común y quizás la que más hemos utilizado a lo largo de este curso es la función `print()`. En este [enlace](#) puedes ver la lista completa de las funciones predefinidas en Python y recomiendo que revises qué es lo que hacen y cómo se utilizan.

Tomemos como ejemplo la función predefinida llamada `input()`, que se utiliza para pedir al usuario que ingrese cualquier información que se le pida:

```
[1]: input('Ingresa tu nombre: ')
```

Ingresa tu nombre: Python

```
[1]: 'Python'
```

En el ejemplo anterior, se ingresó la palabra «Python» y el resultado fue esa misma cadena de caracteres. Debes tener cuidado, porque la función `input()` siempre asume que lo que ingresas es una cadena de caracteres, incluso si lo que ingresas es un número:

```
[2]: number = input('Ingresa un número: ')
print(number)
print(type(number))
```

```
[2]: Ingresa un número: 5
5
<class 'str'>
```

En el ejemplo anterior, se ingresó el número 5, pero al utilizar la función `type()` vemos que se guardó como una variable de tipo `str` y no como de tipo `int`.

Para que el número sea almacenado como de tipo `int`, se debe usar explícitamente la función `int()` en conjunto con `input()` de la siguiente manera:

```
[3]: number = int( input('Ingresa un número: ') )
print(number)
print(type(number))
```

```
[3]: Ingresa un número: 5
5
<class 'int'>
```

Como puedes darte cuenta, las funciones son parte fundamental de cualquier código ya que permiten realizar tareas específicas. Ahora veremos cómo crear nuestras propias funciones para realizar cualquier tarea que se nos ocurra.

4.2 Funciones definidas por el usuario

La estructura y sintaxis para declarar una función en Python se muestra en la Figura 4.1.

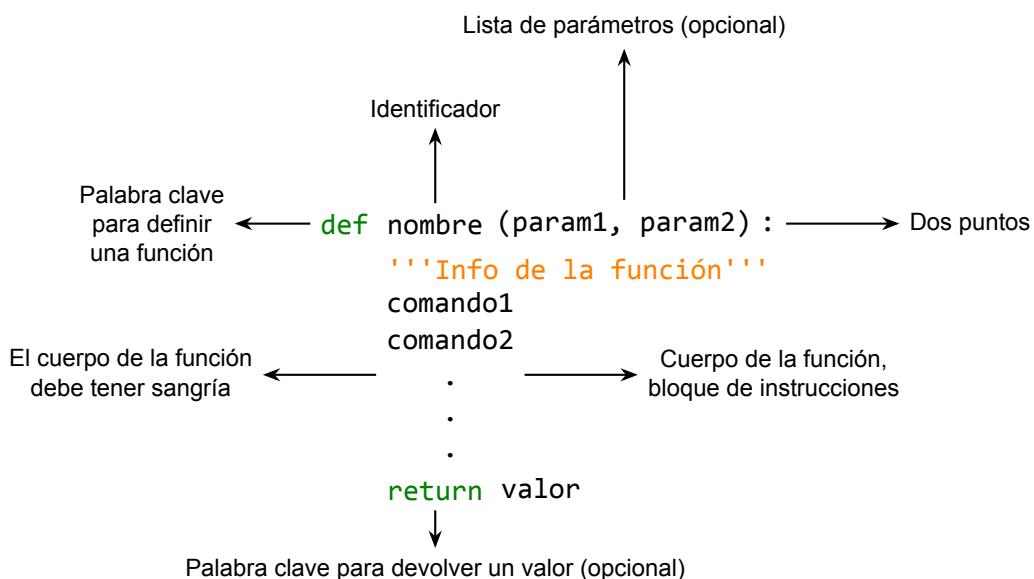


Figura 4.1: Estructura de una función en Python

Para declarar una función, primero se utiliza la palabra clave «**def**», que le informa a Python que se está definiendo una función. Luego de la palabra **def** y separado por un espacio, se debe escribir el nombre de la función, que sirve como *identificador* de la función. Inmediatamente después y sin espacios se colocan paréntesis y entre ellos se escriben los parámetros de la función separados por comas. Los parámetros son opcionales, como veremos más adelante. Al cerrar los paréntesis se escriben dos puntos. Esta primera línea constituye la «*definición de la función*».

Cualquier línea con sangría de cuatro espacios que se encuentre abajo de la definición de la función constituye el cuerpo de la función. El texto inmediatamente abajo de la definición, que en la figura dice **'''Info de la función'''**, constituye la documentación, o *docstring*, que describe qué es lo que hace la función. La documentación siempre se escribe entre comillas triples.

Las líneas debajo de la documentación son las líneas de código reales en el cuerpo de la función. En el ejemplo, estas líneas dicen `commando1, commando2, ...` y le indican a la función qué tarea debe realizar.

Si en el cuerpo de la función se calculó una variable llamada `valor`, dicha variable puede ser devuelta por la función en la última línea con ayuda de la palabra clave «**return**».

A continuación revisaremos diferentes tipos de funciones, comenzando por las más simples hasta llegar a declaraciones más estructuradas.

4.2.1 Funciones sin parámetros

El ejemplo más sencillo es el de una función que no requiere de parámetros de entrada y que imprime un saludo. Veamos cómo definirla siguiendo la sintaxis de la Figura 4.1:

```
[4]: def salu2():
    """Muestra un mensaje"""
    print("¡Hola!")
```

Para utilizarla, la llamamos mediante su nombre:

```
[5]: #- Invocando La función
salu2()
```

```
[5]: ¡Hola!
```

Como la función no necesita de ningún parámetro, fue suficiente con escribir salu2() y no escribir nada más. Ahora veamos un poco sobre funciones que sí requieren de parámetros de entrada.

4.2.2 Funciones con parámetros

La siguiente función requiere de un parámetro de entrada y muestra un mensaje personalizado cada vez que se invoca. Debes tener en cuenta que en este caso en particular, el parámetro de entrada tiene que ser una variable de tipo **str**.

```
[6]: def salu3(name):
    """Muestra un mensaje personalizado"""
    print(f"¡Hola, {name}!")
```

Para utilizarla, debemos llamarla mediante su nombre pero esta vez necesitamos agregar un *argumento*:

```
[7]: salu3("Señor Doctor Profesor Patricio")
```

```
[7]: ¡Hola, Señor Doctor Profesor Patricio!
```

Vale la pena hacer una aclaración: la variable «name» en la definición de la función salu3() es un parámetro, una pieza de información que la función necesita para hacer su trabajo. En cambio, el valor "Señor Profesor Patricio" en la expresión salu3("Señor Profesor Patricio") es un argumento. Es decir, un argumento es una pieza de información que se pasa de la llamada de una función hacia la función misma.

4.2.3 Funciones que devuelven un valor

Ahora veamos un ejemplo sencillo de una función que requiere de parámetros de entrada y que además devuelve un valor.

```
[8]: def suma(a, b):
    """Calcula la suma de dos números"""
    c = a + b
    return c

resultado = suma(3, 5)
print(resultado)
```

Este ejemplo muestra una nueva propiedad sobre las funciones. Hemos utilizado la palabra **return** para ser capaces de almacenar el resultado de la suma en una variable al momento de invocar la función. En realidad, la variable «c» definida dentro de la función no es necesaria. Hay una forma más simple de escribir una función que haga la misma tarea:

```
[9]: def sumar(a, b):
    """Calcula la suma de dos números"""
    return a + b
```

La función «suma» y la función «sumar» son totalmente equivalentes. Al momento de declarar funciones, intenta escribirlas de la manera más simple posible.

Veamos un ejemplo más. La siguiente función toma un número como parámetro de entrada y devuelve **True** si el número es par, o **False** si es impar.

```
[10]: def es_par(n):
    """Muestra si es par o no"""
    return n % 2 == 0
```

Comprobamos si los números **7** y **10** son pares o no:

```
[11]: es_par(7)
```

```
[11]: False
```

```
[12]: es_par(10)
```

```
[12]: True
```

4.3 Más sobre funciones con parámetros

Como habrás notado, es posible definir funciones con más de un parámetro. En la práctica, una función puede admitir una cantidad arbitraria de argumentos. Debido a esto, existen diferentes tipos de argumentos que pueden utilizarse en las funciones. A continuación revisaremos únicamente dos de ellos y cómo podemos combinarlos.

4.3.1 Argumentos con un valor predeterminado

Al momento de declarar una función es posible especificar un valor predeterminado para uno o más de sus argumentos. Esto resulta bastante útil, ya que permite invocar a la función con menos argumentos de los definidos. Por ejemplo:

```
[13]: def saludar(name, repeat=True, bye='¡Adiós!'):
    """Muestra un saludo y despedida personalizados"""

    print(f'Hola, {name}.')
    if repeat:
        print(f'Hola de nuevo, {name}.')
    print(bye)
```

Esta función se compone de un argumento obligatorio llamado «name» y dos argumentosopcionales, llamados «repeat» y «bye». El argumento name es obligatorio porque no tiene ningún valor asignado por defecto y sin él, la función devolverá un error.

De manera concreta, la función saludar() puede ser invocada de muchas formas. La primera, brindando solo el argumento obligatorio:

```
[14]: saludar('Marty McFly')
```

```
Hola, Marty McFly.  
Hola de nuevo, Marty McFly.  
¡Adiós!
```

Ya que se ingresó un solo valor, la función le asignó ese valor al primer parámetro definido en la función. En este caso, al parámetro name. A este tipo de asignaciones se les llama argumentos posicionales, porque se asignan según el orden en que se proporcionan.

La segunda forma de invocar la función es brindando uno de los argumentos opcionales:

```
[15]: saludar('Marty McFly', False)
```

```
Hola, Marty McFly.  
¡Adiós!
```

O como tercera opción, brindando todos los argumentos:

```
[16]: saludar('Marty McFly', False, 'Nos vemos')
```

```
Hola, Marty McFly.  
Nos vemos
```

4.3.2 Argumentos de palabras clave

Las funciones también pueden ser invocadas haciendo uso de los argumentos de palabras clave (en inglés llamados *keyword arguments*), los cuales tienen la forma «`kwarg=value`». En esencia, al invocar la función se especifica el nombre del argumento y se le asigna un valor. Para comprenderlo, revisemos de nuevo la función `saludar()`. Esta función puede ser invocada en cualquiera de las siguientes formas:

```
[17]: saludar('Dr. Brown') # 1 argumento posicional
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
¡Adiós!
```

```
[18]: saludar(name='Dr. Brown') # 1 argumento de palabra clave
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
¡Adiós!
```

```
[19]: saludar(name='Dr. Brown', repeat=False) # 2 argumentos palabra clave
```

```
Hola, Dr. Brown.  
¡Adiós!
```

```
[20]: saludar(repeat=False, name='Dr. Brown') # 2 argumentos palabra clave
```

```
Hola, Dr. Brown.  
¡Adiós!
```

```
[21]: saludar('Dr. Brown', False, 'Nos vemos') # 3 argumentos posicionales
```

```
Hola, Dr. Brown.  
Nos vemos
```

```
[22]: saludar('Dr. Brown', bye='Nos vemos') # 1 posicional, 1 palabra clave
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
Nos vemos
```

Pero todas las siguientes formas de invocarla son inválidas y resultan en un error:

```
[23]: saludar()
```

```
-----  
TypeError                                         Traceback (most recent call last)  
Cell In[25], line 1  
----> 1 saludar()  
  
TypeError: saludar() missing 1 required positional argument: 'name'
```

El mensaje de error en este caso es bastante claro: se invocó a la función sin brindarle el argumento obligatorio. Intenta lo siguiente:

```
[24]: saludar(name='Dr. Brown', False)
```

```
-----  
Cell In[26], line 1  
      saludar(name='Dr. Brown', False)  
          ^  
SyntaxError: positional argument follows keyword argument
```

El error ocurre porque se intenta usar un argumento posicional después de un argumento de palabra clave y eso no está permitido en Python. Nuevamente, intenta:

```
[25]: saludar(False, name='Dr. Brown')
```

```
-----  
TypeError                                         Traceback (most recent call last)  
Cell In[27], line 1  
----> 1 saludar(False, name='Dr. Brown')  
  
TypeError: saludar() got multiple values for argument 'name'
```

Este error se debe a que Python asigna el valor `False` al argumento `name` ya que ese valor se ingresa al principio como argumento posicional. Sin embargo, luego se ingresa el valor `'Dr. Brown'` al argumento `name` como argumento de palabra clave. De modo que se le está intentando asignar más de un valor al mismo argumento y eso resulta en un error. Para terminar, intenta ejecutar los siguiente:

```
[26]: saludar(action='Repetir')
```

```
-----  
TypeError                                         Traceback (most recent call last)  
Cell In[28], line 1  
----> 1 saludar(action='Repetir')  
  
TypeError: saludar() got an unexpected keyword argument 'action'
```

En este caso, el error se debe a que se intenta invocar a la función con un argumento que no está definido.

En conclusión, puedes observar lo siguiente al momento de invocar una función:

- Los argumentos de palabra clave deben ir después de los argumentos posicionales.
- Todos los argumentos de palabra clave deben coincidir con uno de los argumentos aceptados por la función (por ejemplo, `action` no es un argumento válido para la función `saludar`) y el orden en el que aparecen no es importante. Esto también incluye a los argumentos no opcionales (por ejemplo, `saludar(name='Dr. Brown')` es una forma válida de invocar la función).
- Ningún argumento puede recibir más de un solo valor.

4.4 Problemas

Los siguientes problemas están pensados para que te familiarices aún más con Python y las funciones. El objetivo es que realices las tareas que se te piden SIN utilizar las funciones predefinidas de Python.

1. Escribe una función que encuentre el mayor entre dos números.
2. Escribe una función que encuentre el mayor entre tres números.
3. Escribe una función que encuentre el mayor número en una lista.
4. Escribe una función que calcule la suma de los elementos en una lista.
5. Escribe una función que calcule el producto de los elementos en una lista.
6. Escribe una función que calcule el factorial de un número.
7. Escribe una función que invierta los caracteres de una variable de tipo `str`. Es decir, si se ingresa la palabra '`AMOR`', la función debe devolver la palabra '`ROMA`'.
8. Escribe una función que compruebe si una palabra/oración es un palíndromo.
9. Escribe una función que calcule la cantidad de caracteres que tiene una palabra.
10. Escribe una función que calcule la cantidad de caracteres que tienen todas las palabras en una lista.

Unidad 2

Introducción a la astronomía observacional

Clase 5 | Telescopios ópticos

Desde tiempos antiguos, el ser humano ha sido curioso y se ha permitido alzar la mirada al cielo, observando así diferentes cuerpos astronómicos, tales como el Sol, la Luna, los planetas del sistema solar e incluso las estrellas de nuestra Galaxia. Estos objetos nos brindan información crucial a través de la luz que emiten o reflejan, permitiéndonos estudiarlos desde la Tierra.

Para explorar las regiones más lejanas del universo, ha sido indispensable la construcción de telescopios; instrumentos que amplifican nuestra capacidad de observar y analizar regiones del espacio extremadamente distantes de nuestro planeta. Sin embargo, para aprovechar al máximo las observaciones astronómicas que estos telescopios nos proporcionan, es fundamental desarrollar una comprensión profunda de los principios de la luz, como su naturaleza, comportamiento y cómo interactúa con la materia.

5 Telescopios

5.1 Naturaleza la luz

En nuestra vida cotidiana, solemos referirnos a la luz como aquello que nuestros ojos pueden percibir. Sin embargo, gracias a la teoría del electromagnetismo y a las ecuaciones de Maxwell, entendemos que cuando partículas cargadas, como protones o electrones, se aceleran, generan ondas electromagnéticas. Estas ondas forman un vasto espectro conocido como el «espectro electromagnético», del cual la luz «óptica» o «visible» constituye solo una pequeña porción.

Las diferentes regiones del espectro electromagnético pueden ser descritas en términos de la longitud de onda, la frecuencia y la energía. La longitud de onda se mide en metros (m) y se representa con el símbolo λ (letra griega lambda), mientras que la frecuencia se mide en unidades en hertz ($1\text{Hz} \equiv \text{s}^{-1}$) y se representa con el símbolo ν (letra griega nu). Ambas cantidades se relacionan mediante:

$$c = \lambda\nu,$$

donde c es la velocidad de la luz en el espacio vacío y tiene un valor igual a $3 \times 10^8 \text{ m s}^{-1}$. La energía de la luz puede calcularse de manera simple con su frecuencia, gracias a la relación:

$$E = h\nu,$$

donde h es la constante de Planck y tiene un valor numérico de $6.62607015 \times 10^{-34} \text{ J Hz}^{-1}$. Esta ecuación nos indica que a menor frecuencia (o de manera equivalente, a mayor longitud de onda), menor será la energía de la luz y viceversa.

La región óptica del espectro electromagnético está compuesta de luz cuya longitud de onda corresponde a todos los colores que percibimos, que van desde el azul ($420 \times 10^{-9} \text{ m}$) hasta el rojo ($640 \times 10^{-9} \text{ m}$) y que pueden apreciarse en los arcoíris. En la Figura 5.1 se muestra la región visible dentro de todo el espectro electromagnético.

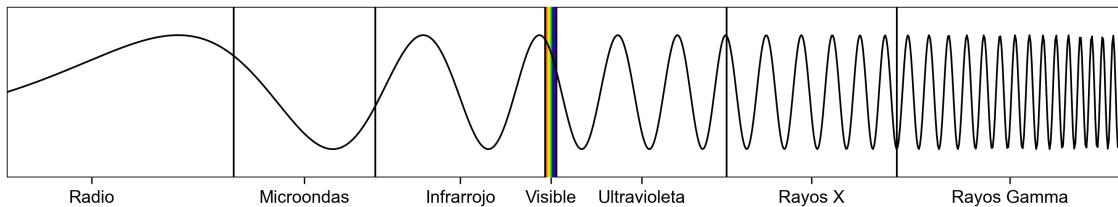


Figura 5.1: Espectro electromagnético

Las diferentes porciones del espectro electromagnético reciben nombres específicos. Por ejemplo, la luz con longitudes de onda un poco más largas que el color rojo y que no somos capaces de ver se conoce como infrarrojo. Las ondas de radio son las ondas electromagnéticas con las longitudes de onda más largas y, por lo tanto, las menos energéticas. La región entre las ondas de radio y el infrarrojo se llama microondas, con longitudes de onda entre micrómetros y centímetros.

En el otro extremo del espectro, encontramos ondas electromagnéticas con longitudes de onda más cortas. Específicamente, la luz con longitud de onda más corta que el color violeta se llama ultravioleta. La luz con longitudes de onda aún menores se conoce como rayos X, y la luz con la longitud de onda más corta de todas se llama rayos gamma, que también es el tipo de radiación electromagnética más energética del universo.

5.2 Principios básicos de óptica geométrica

5.3 Reflexión y refracción

La óptica geométrica es una rama de la óptica que estudia y describe la luz en términos de rayos, donde un rayo se define como una línea imaginaria que indica la dirección en la que se propaga la luz. Esta dirección sigue un camino recto, permitiendo que la luz viaje entre dos puntos en el menor tiempo posible. Con esta aproximación de rayos, es posible describir la **refracción** y **reflexión** de la luz, los cuales son dos principios fundamentales que nos ayudarán a comprender cómo funcionan los telescopios.

La refracción ocurre cuando la luz pasa a través de un material transparente, como el agua o el vidrio, mientras que la reflexión ocurre cuando la luz «rebota» sobre una superficie, como los espejos. Cuando la luz pasa de un material a otro (por ejemplo, del aire al agua), el camino que sigue la luz cambia de dirección, debido a que la velocidad de la luz es diferente en ambos medios. Para describir este cambio de dirección se utiliza la **Ley de Snell**:

$$n_1 \sin \theta_1 = n_2 \sin \theta_2,$$

donde n_1 es el índice de refracción del primer material, n_2 es el índice de refracción del segundo material, θ_1 es el ángulo incidente relativo a la normal de la superficie y θ_2 es el ángulo de refracción relativo a la normal.

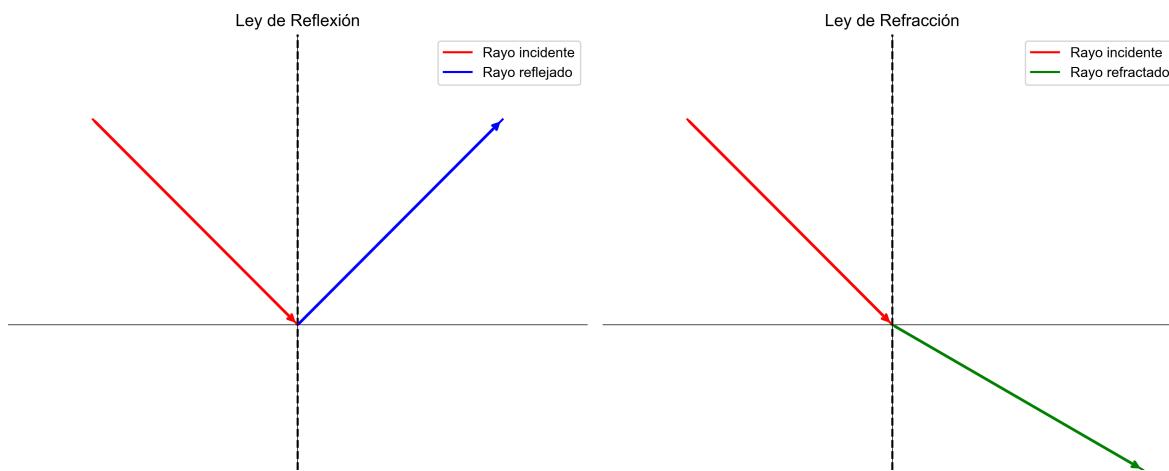


Figura 5.2: Leyes de refracción y reflexión

El índice de refracción de un material es simplemente el cociente entre la velocidad de la luz en el vacío y la velocidad de la luz en ese material: $n = c/v$. Como ejemplo, el índice de

refracción del vidrio es 1.520, el del diamante es 2.417 y el del agua es 1.333.

Para superficies reflectivas, ocurre un principio aún más simple: el ángulo de incidencia es igual al ángulo de reflexión, para todas las longitudes de onda de la luz. Esta es conocida como la **ley de reflexión**. La ley de refracción y ley de reflexión se ilustran en la Figura 5.2.

5.4 Propiedades de telescopios

5.4.1 Recolección de luz

El principio de recolección de luz de los telescopios es bastante simple: mientras más grande sea el espejo o el lente, más luz coleccionará. La cantidad de luz recibida por unidad de tiempo es directamente proporcional al área de recolección y ya que la mayoría de los telescopios son circulares, esto significa que la proporción sigue la relación πr^2 .

Clase 6 | Fotometría

La luz proveniente de los objetos astronómicos y que es recolectada por los telescopios, únicamente nos brinda información sobre la posición y movimiento de las fuentes en el cielo, su brillo aparente, líneas espectrales y la energía que emiten. A estas cantidades se les llama «observables». A pesar de esta limitante, el ser humano ha comprendido mucho sobre las estrellas y las galaxias con ayuda de algunos principios físicos. Gracias a ello, es posible extraer más información de los observables, como la distancia a la fuente, luminosidad, temperatura, composición química, tamaño, campos magnéticos, velocidades. Aprenderemos un poco sobre cómo los astrónomos determinan estos parámetros en las siguientes dos clases.

6 Conceptos de fotometría

Las fuentes astronómicas se clasifican en puntuales y extensas. En la mayoría de los instrumentos, las estrellas aparecen como fuentes puntuales, mientras que las fuentes extensas incluyen objetos como nebulosas, galaxias y el fondo cósmico de microondas. El tratamiento de la luz de estos dos tipos de fuentes varía, y es responsabilidad de la **fotometría**, una rama de la óptica dedicada al estudio y la medición precisa de la luz.

Exploraremos el tema de la fotometría, comenzando por el sistema que los astrónomos utilizan para describir el brillo aparente de las estrellas, el cual, aunque fundamental, puede resultar bastante confuso y contraintuitivo.

6.1 Sistema de magnitudes

6.1.1 Flujo y luminosidad

Lo único que los telescopios son capaces de medir es la cantidad de fotones que llegan a la superficie del detector en un tiempo dado. Este parámetro se define como el flujo y lo representaremos con la letra F , sus unidades son $\text{ph cm}^{-2} \text{ s}^{-1}$ (fotones/unidad de área/unidad de tiempo).

El flujo es una cantidad física que varía inversamente proporcional al cuadrado de la distancia. Esto significa que mientras más lejana sea la fuente, más pequeño será el flujo detectado. Esto se conoce como la ley del inverso del cuadrado y se ilustra de manera esquemática en la Figura 6.1, donde una fuente puntual está emitiendo luz en todas las direcciones (isotrópicamente). La cantidad de luz que pasa a través de una superficie de área A varía dependiendo de la distancia a la que se encuentra la superficie.

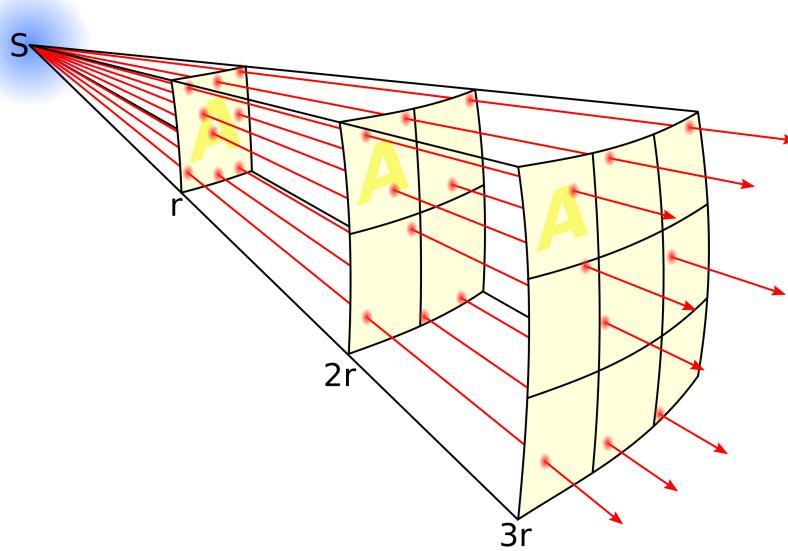


Figura 6.1: Esquema de la ley del inverso del cuadrado para flujos

Asumiendo que la luz de una fuente astronómica se distribuye de manera isotrópica, entonces a una distancia r , el área superficial está dada por $A_1 = 4\pi r^2$ y el flujo será $F_1 \propto \frac{1}{A_1} = \frac{1}{4\pi r^2}$. En cambio a una distancia igual a $2r$, el área estará dada por $A_2 = 4\pi(2r)^2 = 16\pi r^2 = 4A_1$ y el flujo será $F_2 \propto \frac{1}{4A_1} = \frac{1}{4}F_1$. Por lo tanto, si se duplica la distancia de la fuente, el flujo disminuye en un factor de 4.

La constante de proporcionalidad en $F \propto 1/(4\pi r^2)$ es una cantidad muy importante llamada luminosidad. La luminosidad se define como la energía total liberada por una fuente astronómica, se denota con la letra L y su valor depende de las propiedades de cada estrella, pero no de la distancia. En otras palabras, la relación matemática entre el flujo y la luminosidad es:

$$F = \frac{L}{4\pi r^2}$$

donde r es la distancia a la que se encuentra la fuente. Un parámetro muy importante es el valor del flujo en la superficie de la estrella. Si la estrella tiene un radio R , entonces el flujo

superficial, F_S está dado por:

$$F_S = \frac{L}{4\pi R^2}.$$

Podemos resolver esta ecuación para la luminosidad y se obtiene:

$$L = 4\pi R^2 F_S = A F_S,$$

esto nos permite estimar la luminosidad a partir de parámetros de la estrella como su radio R y su flujo superficial F_S . Con ayuda de la termodinámica es posible demostrar que

$$F_S = \sigma T^4,$$

donde $\sigma = 5.67 \times 10^{-5}$ erg cm⁻² s⁻¹ K⁻⁴ es la constante de Stefan-Boltzmann y T es la temperatura en la superficie de la estrella, medida en kelvins (K). Al combinar esta expresión con la luminosidad, se obtiene el resultado:

$$L = A \sigma T^4,$$

el cual es conocido como la ley de Stefan-Boltzmann.

6.1.2 Magnitudes

El brillo de una estrella puede describirse mediante **magnitudes**, donde magnitudes más grandes corresponden a estrellas más tenues. El concepto de magnitud se utiliza desde los tiempos de Hiparco, quien dividió las estrellas visibles en seis grupos de magnitudes igualmente separadas de acuerdo al ojo humano. Las estrellas más brillantes fueron clasificadas como magnitud 1, mientras que las menos brillantes, de magnitud 6. Esta clasificación poco intuitiva se basó originalmente en la apariencia de las estrellas cuando comenzaba a oscurecer durante el ocaso. Aquellas estrellas que aparecían primero (por lo tanto, las más brillantes) se clasificaron como de magnitud 1, las segundas fueron clasificadas como de magnitud 2 y así sucesivamente hasta llegar a las de magnitud 6 (las últimas en aparecer eran las más tenues). Debido a que la clasificación se basa en criterios del ojo humano, se conocen como «magnitudes aparentes».

Esta convención de magnitudes se sigue utilizando en la actualidad y se formula de la siguiente manera. La sensibilidad del ojo humano a la luz sigue una escala logarítmica,

entonces definimos a m_1 y m_2 como las magnitudes aparentes de estrellas cuyos flujos son F_1 y F_2 . Ya que estas cantidades deben seguir una escala logarítmica, se cumple la relación:

$$m_1 - m_2 = -k \log\left(\frac{F_1}{F_2}\right),$$

donde el signo negativo se utiliza para asignar los valores de magnitud más pequeños a las estrellas más brillantes. Diferentes estudios fotométricos indican que las estrellas de magnitud 6 son alrededor de 100 veces menos brillantes que las de magnitud 1. Matemáticamente se puede expresar como $F_1/F_2 = 100$ y además $m_1 - m_2 = 1 - 6 = -5$. Con esto se encuentra que $k = 2.5$ y entonces la relación para magnitudes es:

$$m_1 - m_2 = 2.5 \log_{10}\left(\frac{F_1}{F_2}\right); \quad \frac{F_1}{F_2} = 10^{\frac{2}{5}(m_1 - m_2)}.$$

Algo muy importante que hay que resaltar de este resultado es que no es posible determinar la magnitud de una estrella por sí misma, solo se puede comparar con otra estrella y determinar la diferencia entre sus magnitudes. Además, la ecuación no define un «punto cero» o magnitud de punto cero de manera explícita, de modo que una forma más general es

$$m = -2.5 \log(F) + C,$$

donde C es el «desfase» del punto cero.

Para que este sistema funcione, se debe definir una estrella de magnitud cero, llamada estrella estándar, con la cual sea posible comparar y obtener la magnitud de cualquier otra. De este modo, todos los astrónomos deben coincidir que cierta estrella tiene cierta magnitud. Debes tener en cuenta de que a pesar que se utilizan magnitudes para cuantificar el brillo de las estrellas, la cantidad medible u observable es en realidad el flujo.

Las estrellas estándar comenzaron a definirse gracias a los avances en la fotometría fotográfica en los años 1950. Actualmente, la estrella estándar es Vega, que es del tipo espectral A0 (hablaremos de esto más adelante). Cálculos más modernos implican que la magnitud de Vega es en realidad 0.03.

Una vez definidas las estrellas estándar, es posible obtener la magnitud de cualquier otra. Por ejemplo, la magnitud de Capella es de 0.1 y la de Sirio es de -1.6. Los objetos más brillantes en el cielo son por supuesto la Luna y el Sol, debido a su cercanía y tienen magnitudes de -12.5 y -26.76, respectivamente. Algunas de las estrellas más tenues que se

han observado gracias el Telescopio Espacial Hubble (HST, por sus siglas en inglés) tienen magnitudes de hasta 30.7.

6.1.3 Magnitudes bolométricas y absolutas

Todos los telescopios tienen un rango de longitudes de onda en los que son capaces de observar. En la práctica, las magnitudes se obtienen para una región específica del espectro electromagnético. Por ejemplo, cuando hablamos de magnitudes aparentes, nos referimos a magnitudes en el rango visible, pero también se trabaja en otras longitudes de onda. La versión **monocromática** (es decir, a una longitud de onda específica) de la relación entre magnitudes y flujos es:

$$m_\lambda = -2.5 \log(F_\lambda) + C_\lambda$$

El extremo opuesto a la magnitud monocromática es la **magnitud bolométrica**, que para su medición incluye la radiación en todas las longitudes de onda emitidas por la fuente. Estas magnitudes se indican con el subíndice «bol» y es posible aplicar una corrección bolométrica (BC, por sus siglas en inglés) a cualquier medición, simplemente calculando la diferencia entre la magnitud bolométrica y la magnitud en cualquier otra banda de longitudes de onda (como la visual):

$$BC_{\text{bol}} = m_{\text{bol}} - m_{\text{band}}$$

Otra cantidad de importancia es la **magnitud absoluta**, la cual se define como la magnitud aparente que tendría una estrella si estuviera ubicada a 10 pc. Supongamos que el flujo de una estrella a una distancia d es F . Si ahora colocamos esa misma estrella a una distancia d_0 , entonces su flujo será F_0 . Como se trata de la misma estrella, la luminosidad es la misma y entonces:

$$L = 4\pi d_0^2 F_0 = 4\pi d^2 F,$$

de aquí se obtiene que

$$\frac{F}{F_0} = \left(\frac{d_0}{d}\right)^2.$$

Sea m la magnitud de la estrella a una distancia d y M su magnitud absoluta, es decir, a una distancia $d_0 = 10\text{pc}$, entonces la relación entre m y M es la siguiente:

$$m - M = -2.5 \log\left(\frac{F}{F_0}\right) = -2.5 \log\left(\frac{d_0}{d}\right)^2 = 2.5 \log\left(\frac{d}{d_0}\right)^2 = 5 \log\left(\frac{d}{d_0}\right),$$

y sustituyendo $d_0 = 10\text{pc}$:

$$m - M = 5 \log\left(\frac{d}{10\text{pc}}\right) = 5 \log(d) - 5.$$

La cantidad $m - M$ es conocida como el **módulo de distancia**. Para objetos que se encuentran a más de unos cuantos pársecs, la luz proveniente de ellos se dispersa con el medio interestelar y esto provoca que se vean más tenues. Este efecto se llama **extinción interestelar** y debe tomarse en cuenta. Por lo tanto, una relación más útil es la del **módulo de distancia aparente**:

$$(m - M)_\lambda = 5 \log(d) - 5 + A_\lambda,$$

donde A_λ es la absorción en magnitudes a una longitud de onda λ .

Quizás la cantidad más importante de todas es la **magnitud bolométrica absoluta**, denotada por M_{bol} que puede usarse para obtener la luminosidad de una estrella en términos de la luminosidad solar:

$$M_{\text{bol}} = 4.74 - 2.5 \log\left(\frac{L}{L_\odot}\right),$$

donde L_\odot es la luminosidad solar y se mide en erg s^{-1} .

6.1.4 Índices de color

Algunos detectores de luz son más sensibles en un rango específico de longitudes de onda. En otras palabras, si un detector es más sensible a longitudes de onda cortas, algunas estrellas podrían verse más azules en el detector comparadas al ojo humano. Por lo tanto, cuando se establece una magnitud, se debe especificar cómo fue determinada.

Esta diferencia en la sensibilidad permite determinar magnitudes en diferentes bandas de luz o energía, lo que da lugar a cuantificar el color de las estrellas. El **índice de color** se define como la diferencia entre las magnitudes de una estrella en dos bandas distintas. Normalmente, la banda visible se identifica con la letra B , la banda ultravioleta con la letra U y la banda azul, con la letra V . Así, el índice de color $B - V$ de una estrella está dado por

$$B - V = m_B - m_V,$$

donde m_B y m_V son las magnitudes en la banda ultravioleta y visible, respectivamente. Una estrella con $B - V < 0$ se ve más azul que una estrella con $B - V > 0$.

6.1.5 Filtros de color

Es bastante común trabajar con filtros de color, los cuales son dispositivos que solo permiten pasar luz con una longitud de onda específica. Uno de los sistemas de filtros más utilizados es el *UBV*, el cual permite pasar luz únicamente en un rango estrecho de ultravioleta, azul y visible. La sensibilidad de este tipo de filtros se muestra en la Figura 6.2.

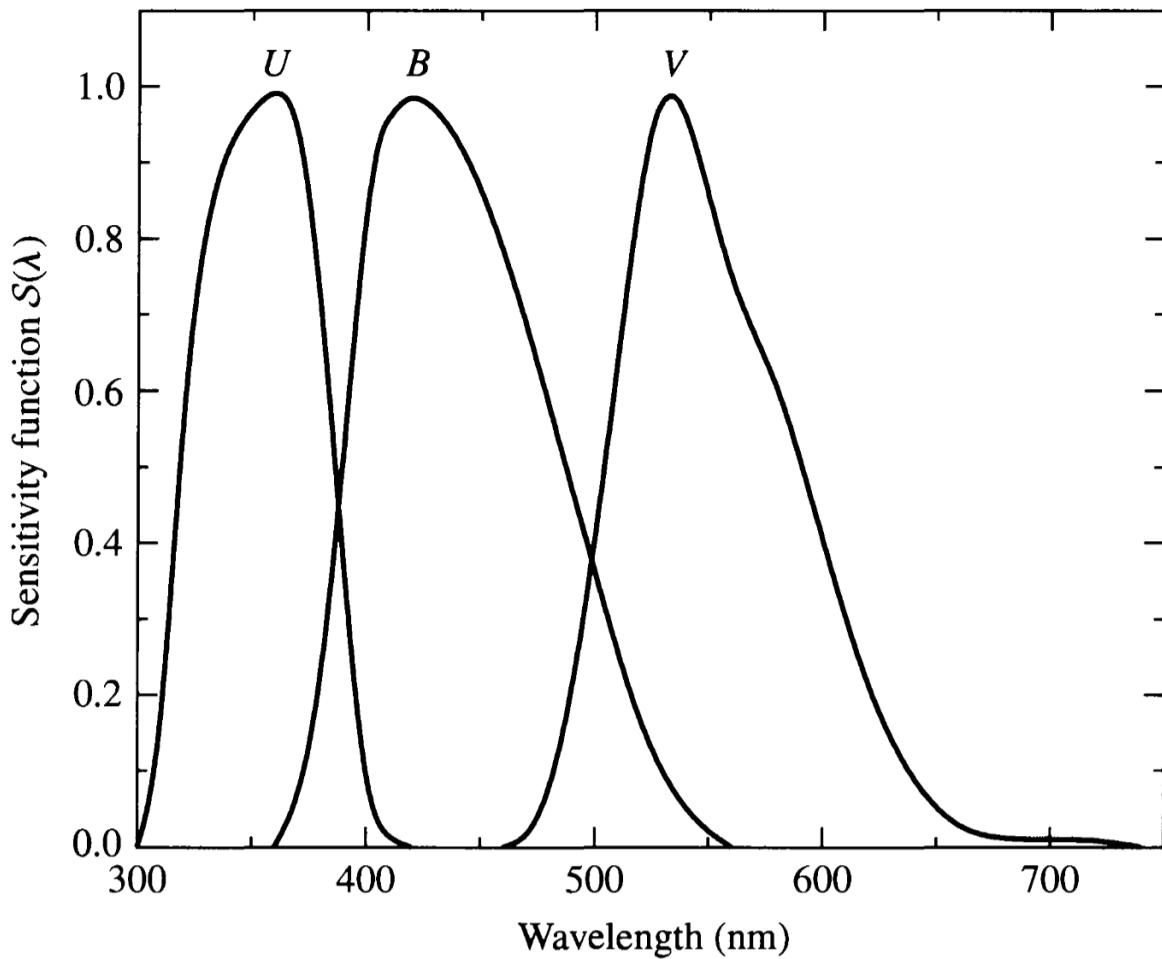


Figura 6.2: Curva de sensibilidad del sistema de filtros *UBV*.

Como se puede apreciar en la Figura 6.2, los filtros tienen una sensibilidad del 100 % en un intervalo muy estrecho de longitudes de onda:

- El filtro *U* está en la región ultravioleta, centrado en 365 nm y con un espesor efectivo de 68 nm.
- El filtro *B* está en la región azul, centrado en 440 nm y con un espesor efectivo de 98 nm
- El filtro *V* está en la región visible, centrada en 550 nm y con un espesor efectivo de

89 nm.

6.2 Radiación de cuerpo negro

En astronomía se utiliza el concepto de radiación electromagnética emitida por un objeto idealizado llamado **cuerpo negro**, que se define como un objeto que absorbe y emite luz de manera perfecta a todas las longitudes de onda. El flujo de un cuerpo negro depende solo de su temperatura superficial y está dado por las funciones de Planck:

$$B_\nu(T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{h\nu/(kT)} - 1}$$

$$B_\lambda(T) = \frac{2hc^2}{\lambda^5} \frac{1}{e^{hc/(\lambda kT)} - 1},$$

donde las unidades son $\text{erg s}^{-1} \text{cm}^{-2} \text{Hz}^{-1} \text{ster}^{-1}$ en la primera ecuación y $\text{erg s}^{-1} \text{cm}^{-2} \text{cm}^{-1} \text{ster}^{-1}$ en la segunda. Los esteradianes (ster) son unidades de ángulo sólido; el cielo completo tiene 4π esteradianes. Una gráfica de esta distribución se muestra en la Figura ??.

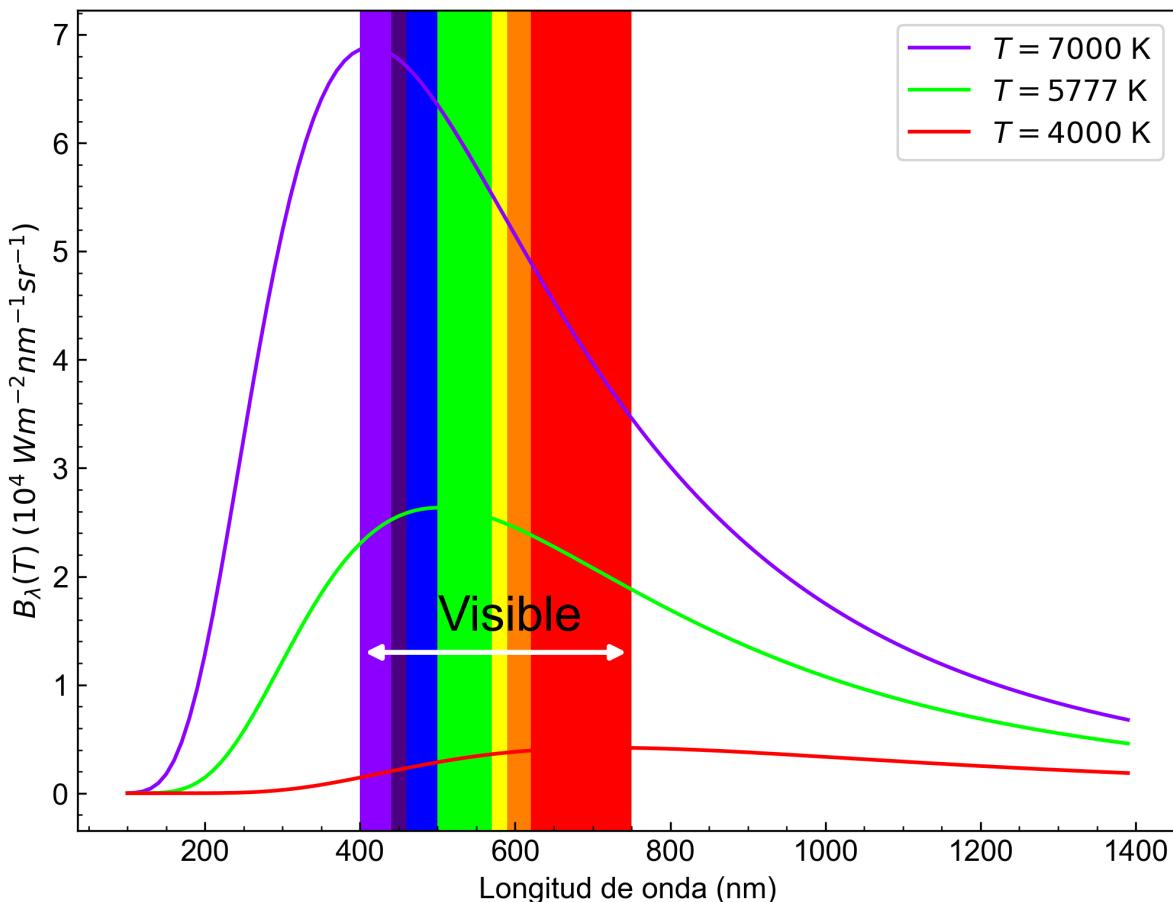


Figura 6.3: Distribución de energía de un cuerpo negro para diferentes temperaturas.

La ley de desplazamiento de Wien nos ayuda a obtener la longitud de onda a la que ocurre el máximo de cada distribución en función de la temperatura:

$$\lambda_{\text{máx}} = \frac{c}{T},$$

donde $c = 2900000 \text{ K nm}$ y la temperatura se mide en K.

Clase 7 | Espectroscopía

La espectroscopía es una de las técnicas fundamentales de la astronomía, ya que permite determinar la composición química de diferentes fuentes astronómicas, sus propiedades físicas, velocidades radiales y distancias. Además, la espectroscopía es la herramienta que se utiliza para determinar la masa de estrellas en sistemas binarios, calcular el contenido de materia oscura en galaxias y descubrir exoplanetas, entre otras cosas. En esta clase abordaremos algunos conceptos de espectroscopía y revisaremos cómo se utilizan para estudiar el cielo.

7 Conceptos de espectroscopía

7.1 Líneas espectrales

Comenzaremos definiendo un concepto muy importante en astronomía: un espectro. Llamamos **espectro** de una fuente a una representación gráfica que muestra cómo varía la intensidad de la luz de dicha fuente en función de su longitud de onda o su frecuencia. En este sentido, la espectroscopía como una técnica que se utiliza para estudiar los espectros de una fuente cuando su luz interactúa con la materia.

En 1666, Isaac Newton demostró que un prisma de vidrio puede dispersar la luz solar en un espectro. Estudios posteriores mostraron que sobre los colores del espectro solar aparecen algunas líneas oscuras. En 1815, Joseph von Fraunhofer obtuvo alrededor de 574 líneas sobre el espectro solar. Debido a esto, ahora son llamadas «líneas de Fraunhofer». En la Figura 7.1 se muestra un espectro solar con líneas de Fraunhofer.



Figura 7.1: Espectro solar con líneas de Fraunhofer. (Fuente: [BASe de données Solaire Sol.](#))

Para comprender qué son estas líneas oscuras, debemos revisar algunos conceptos sencillos de mecánica cuántica.

7.1.1 Producción de líneas espectrales

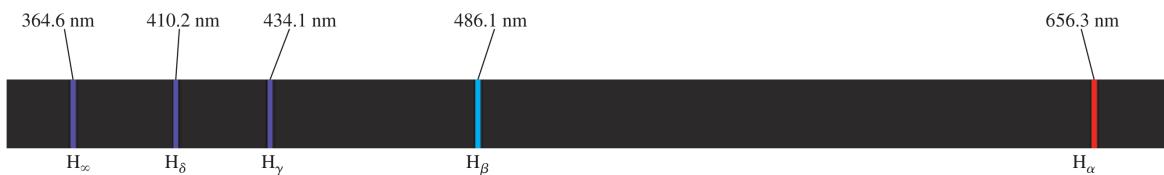
Las líneas espectrales aparecen en dos formas: en espectros de absorción, donde aparecen líneas oscuras en un fondo brillante (como en el caso del espectro solar) y en espectros de emisión, donde aparecen líneas brillantes sobre un fondo oscuro. Ambos tipos de líneas se deben a interacciones cuánticas entre electrones que orbitan a los átomos y fotones de luz.

Sabemos que los fotones tienen una energía específica que depende de su frecuencia. Esta energía está dada por $E = h\nu$. En el modelo semi-clásico del átomo, también llamado el átomo de Bohr, un electrón orbita al núcleo en un nivel de energía estable. Este nivel estable es llamado el nivel base, porque es el de menor energía. Si un fotón con una frecuencia específica interactúa con el electrón, éste puede ganar la energía suficiente para subir a un nivel de energía superior. A este proceso se le llama «excitación», y se dice que el átomo se encuentra en un nivel excitado. Si eso ocurre, entonces el fotón original es absorbido por el electrón de modo que ya no puede ser detectado por un observador. Posteriormente, cuando el electrón se encuentra en un nivel superior, ocurre el proceso de «desexcitación» y salta de nuevo hacia un nivel inferior, emitiendo en el proceso un fotón a una frecuencia específica. Sin embargo, este fotón es emitido en una dirección aleatoria, no en la misma dirección que el fotón incidente original.

Los espectros de absorción ocurren cuando el átomo absorbe fotones y pasa a un estado excitado. Las líneas oscuras son, por lo tanto, los fotones que fueron absorbidos. En cambio, los espectros de emisión ocurren cuando una fuente se encuentra en estado excitado y pasa a un estado de menor energía, liberando fotones. Las líneas brillantes son, por lo tanto, los fotones emitidos.

Cuando electrones en un átomo de hidrógeno en cualquier nivel $n > 2$ realizan una transición al nivel $n = 2$ se producen líneas de emisión en el rango visible, conocidas como la **serie de Balmer**. Por ejemplo, cuando la transición ocurre desde $n = 3$ hasta $n = 2$, la línea de emisión se conoce como «Balmer alfa» y se denota como H_α . Si la transición ocurre desde $n = 4$ hasta $n = 2$, la línea se llama «Balmer beta» y se denota como H_β y así sucesivamente para H_γ y H_δ .

En la Figura 7.2 se muestra un ejemplo de las líneas de Balmer. Las longitudes de onda a la que ocurren las líneas de emisión de H_α , H_β , H_γ y H_δ son 656.3 nm, 486.1 nm, 434.1 nm y 410.2 nm, respectivamente. Si estas líneas aparecen al observar una fuente, entonces podemos estar seguros que esa fuente contiene hidrógeno entre sus componentes químicos.



Otras series de líneas del hidrógeno son las que ocurren desde un nivel $n > 1$ hasta el nivel $n = 1$, que son conocidas como series de Lyman. De manera análoga, la línea debida a la transición de $n = 2$ hasta $n = 1$ es llamada «Lyman alfa» y se denota con Ly_α . La transición de $n = 3$ hasta $n = 1$ es «Lyman beta», denotada con Ly_β y así sucesivamente. Las líneas de Lyman ocurren en la región ultravioleta del espectro electromagnético. Las series de Paschen corresponden a transiciones desde un nivel $n > 3$ hasta $n = 3$ y ocurren en la región infrarroja del espectro. La Figura 7.3 muestra un diagrama de la formación de líneas espectrales en el átomo de hidrógeno.

Electron transitions for the Hydrogen atom

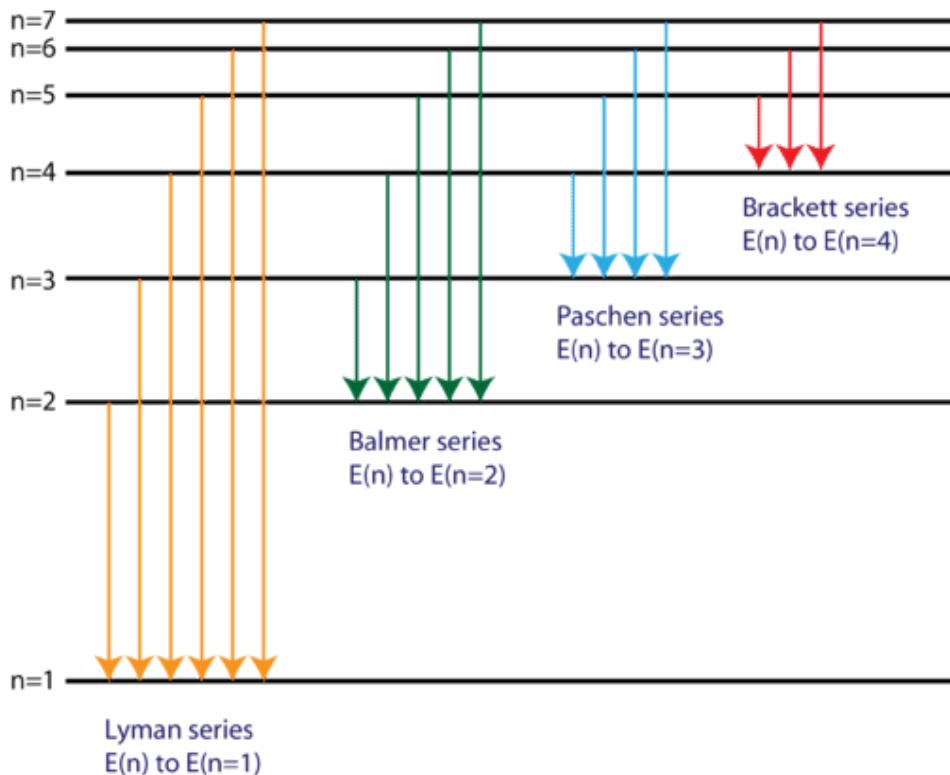


Figura 7.3: Transiciones de los electrones en modelo semicásico del átomo de hidrógeno. Fuente: CK-12 foundation.

7.1.2 Tipos de espectros

Diferentes objetos astronómicos producen diferentes tipos de espectros. Estudiar el espectro de un objeto es uno de las formas de identificar qué tipo de objeto es. Existen tres tipos generales de espectros conocidos: (1) espectros continuos, que muestran las componentes de todos los colores del arcoíris, (2) espectros de líneas oscuras sobre un fondo brillante y (3) espectros de líneas brillantes sobre un fondo oscuro.

En la Figura 7.4 se muestra un diagrama esquemático de cómo se producen los diferentes espectros. A continuación se explica detalladamente cada uno de ellos.

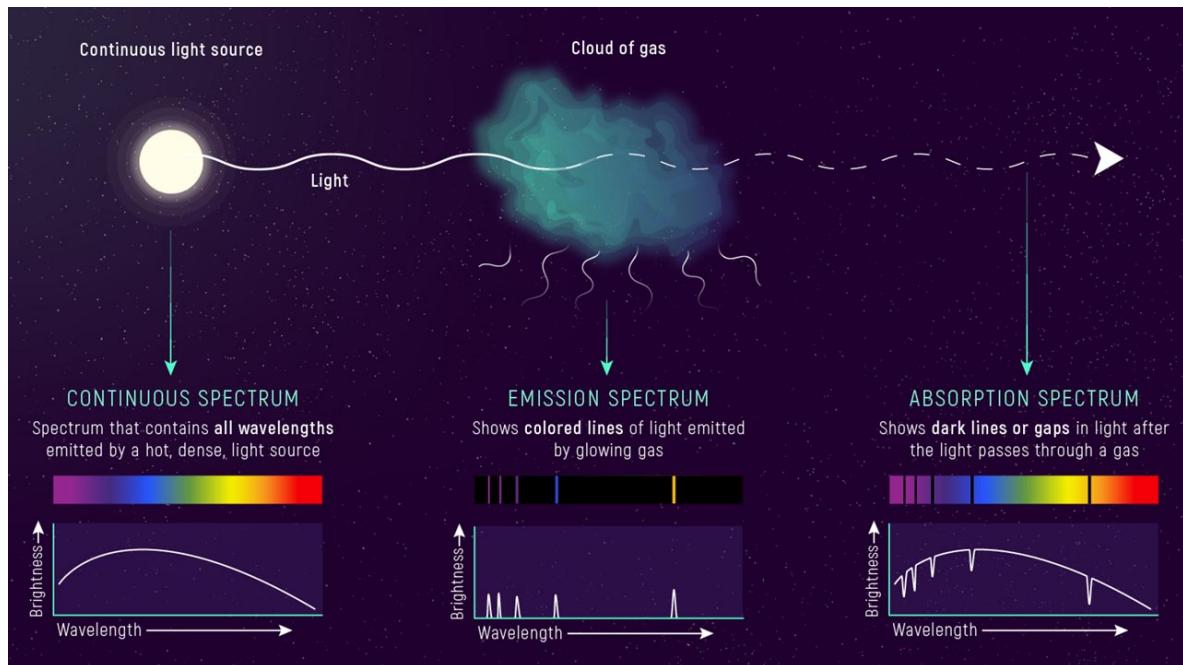


Figura 7.4: Tipos de espectros conocidos. Fuente: [Webb Space Telescope](#).

El **espectro continuo**: se forma a partir de objetos calientes y densos como el núcleo de las estrellas, que actúan como cuerpos negros. Si fuéramos capaces de ver la luz de estas fuentes de manera directa y sin intervención de ningún tipo de materia, entonces lo veríamos como la línea continua mostrada en el panel inferior izquierdo de la Figura 7.4.

El **espectro de absorción**: las estrellas están rodeadas por capas exteriores de gas que son menos densas que el núcleo. Estas capas son las atmósferas estelares. Los fotones emitidos en el núcleo abarcan todas las longitudes de onda, pero algunos con frecuencias específicas pueden ser absorbidos en la atmósfera. El efecto neto es que en la Tierra se detectan menos fotones de los generados en el núcleo y el espectro mostrará líneas de absorción que se muestran como una disminución en la intensidad, tal como aparece en el

panel inferior derecho de la Figura 7.4.

El **espectro de emisión**: ocurre cuando no se está viendo directamente a una estrella, sino a una nube de gas difusa. Los átomos de la nube pueden llevarse a estados excitados debido a su cercanía con estrellas jóvenes que emiten fotones lo suficientemente energéticos. Cuando sus átomos se desexciten, van a emitir fotones a longitudes de onda y frecuencias específicas. Debido a que la dirección en la que estos fotones son emitidos es aleatoria, un observador en la Tierra puede detectarlos y el espectro estará compuesto por líneas de emisión, como el mostrado en el panel central de la Figura 7.4.

7.1.3 Clasificaciónpectral de estrellas

Los astrónomos han diseñado una forma bastante sencilla para diferenciar y clasificar a las estrellas de acuerdo a su temperatura efectiva, también llamada temperatura superficial. La mayoría de las estrellas están dentro de las siguientes clases espectrales: **O, B, A, F, G, K, M**.

Las clases espectrales van desde la letra O, que son las estrellas más calientes, hasta la letra M, que son las estrellas más frías. Las letras no siguen ningún orden y pueden resultar confusas, ya que al igual que con el sistema de magnitudes, se asignaron debido a razones históricas antes que se comprendieran las propiedades físicas de las estrellas. Un método para recordar el orden de la clasificaciónpectral es utilizando el siguiente truco:

Oh, Be A Fine Girl, Kiss Me.

Adicionalmente, se le asigna un número entre 0 y 9 a cada letra de las clasesspectrales para refinar la clasificación, de modo que los números más pequeños indican temperaturas mayores. Por ejemplo, una estrella de tipo B2 es más caliente que una de tipo B6. El Sol es una estrella de tipo G5.

7.2 Velocidades radiales

Una de las tareas fundamentales de la astronomía es la medición del movimiento de los objetos astronómicos. Para describir totalmente el movimiento de un objeto se necesita obtener su posición y sus componentes de velocidad en las tres coordenadas espaciales. Para describir la posición de todos los objetos se utiliza un sistema de referencia llamado

«coordenadas celestes», en los cuales se debe determinar su ascensión recta (RA, por sus siglas en inglés), su declinación y la dirección a lo largo de la línea de visión (LOS, por sus siglas en inglés). Para más información sobre este sistema de coordenadas puedes referirte al libro «Fundamental Astronomy».

Cuando la distancia a un objeto es conocida, las componentes de velocidad perpendiculares al LOS pueden calcularse midiendo los cambios en las coordenadas celeste con respecto al tiempo. Estas componentes de velocidad son llamadas «*movimientos propios*» y son inversamente proporcionales a la distancia. Por lo tanto, son más fáciles de medir para objetos cercanos. La tercera componente de velocidad se mide con respecto al LOS y es llamada la velocidad radial, ya que nos indica si el objeto se acerca o se aleja de nosotros.

Por convención en astronomía, la velocidad radial se define tal que es positiva si la distancia entre el objeto y la Tierra está incrementando (es decir, se alejan) y negativa si la distancia está disminuyendo (se acercan). Esta velocidad puede obtenerse a partir de observaciones espectroscópicas en términos del efecto Doppler.

7.2.1 El efecto Doppler

El efecto Doppler describe el cambio aparente en la longitud de onda (y la frecuencia) de una onda cuando existe movimiento relativo entre la fuente y el observador. El ejemplo más común es el de las ondas sonoras cuando pasa una ambulancia.

En el contexto de las ondas electromagnéticas, se utiliza el término «desplazamiento Doppler» para referirse al cambio en la longitud de onda. Por ejemplo, la serie de Balmer para el hidrógeno ocurre a las longitudes de onda mostradas en la Figura 7.2. Si medimos el espectro de una fuente que se encuentra en reposo con respecto a la Tierra, entonces las líneas espectrales de la serie de Balmer para esa fuente serán exactamente iguales a las que se miden en el laboratorio.

Un ejemplo más realista involucra el movimiento relativo entre una fuente astronómica y la Tierra. En este caso hay dos posibles escenarios. Primero, si un objeto se mueve de tal modo que se aleja de nosotros, entonces sus líneas espectrales parecerán desplazadas hacia longitudes de onda más largas, o a la región más roja del espectro electromagnético. Este efecto recibe el nombre de «desplazamiento al rojo» (redshift en inglés). Mientras más rápido el objeto se aleje de nosotros, mayor será el desplazamiento al rojo. Segundo, si el objeto se mueve de modo que se acerca a nosotros, entonces sus líneas espectrales parecerán

desplazadas hacia longitudes de onda más cortas, o a la región más azul del espectro. Debido a eso, este efecto se llama «desplazamiento al azul» (blueshift en inglés). Nuevamente, qué tan grande es el desplazamiento al azul depende de la velocidad del objeto. En la Figura 7.5 se muestra un diagrama bastante sencillo explicando el desplazamiento Doppler.



Figura 7.5: Líneas espectrales de un sistema en reposo, un objeto alejándose de nosotros y otro acercándose. Fuente: [Australia Telescope National Facility \(ATNF\)](#).

Cuando una estrella o sistema astronómico se mueve con respecto a la Tierra con una velocidad radial v_r y emite un fotón a una longitud de onda λ_0 , entonces un observador en la Tierra detectará a ese fotón con una longitud de onda desplazada debido al efecto Doppler, λ_{obs} , donde $\lambda_{\text{obs}} \neq \lambda_0$. El desplazamiento Doppler se obtiene realizando el cociente entre la diferencia de longitudes de onda y la longitud de onda real a la que el fotón fue emitido: $(\lambda_{\text{obs}} - \lambda_0)/\lambda_0$. A su vez, estas dos cantidades se relacionan con la velocidad radial mediante la ecuación:

$$\frac{\lambda_{\text{obs}} - \lambda_0}{\lambda_0} = \frac{v_r}{c},$$

donde v_r es la velocidad radial de la fuente y c es la velocidad de la luz. Si la fuente se acerca a nosotros, entonces λ_{obs} se desplaza hacia el azul y es menor que λ_0 y por lo tanto v_r es negativa. En cambio, si la fuente se aleja de nosotros, λ_{obs} se desplaza hacia el rojo y es mayor que λ_0 y v_r es positiva. Una de sus aplicaciones es el estudio de sistemas binarios donde la única información que tenemos son sus líneas de emisión o absorción.

7.2.2 Sistemas binarios espectroscópicos

Este tipo de sistemas binarios se caracteriza porque están muy lejos y sus componentes estelares no pueden resolverse incluso con los telescopios más grandes. Sin embargo, es posible obtener información sobre el sistema gracias al análisis de su espectro. En realidad, la mayoría de los sistemas binarios detectados son gracias al desplazamiento Doppler y la

espectroscopía.

La Figura 7.6 muestra una representación esquemática de un sistema binario espectroscópico. La elipse de color azul representa la órbita de la estrella A, mientras que la elipse de color negro representa la órbita de la estrella B. El punto rojo representa el centro de masa alrededor del que ambas estrellas orbitan. La dirección hacia la Tierra es hacia abajo. En la etapa 1, la estrella A se encuentra en un punto de la órbita en la que se mueve en dirección a la Tierra, por lo tanto, al revisar su espectro se observa que está desplazado hacia el azul. Al mismo tiempo, la estrella B se encuentra en un punto de su órbita en la que se mueve alejándose de la Tierra, por lo tanto, su espectro está desplazado hacia el rojo.

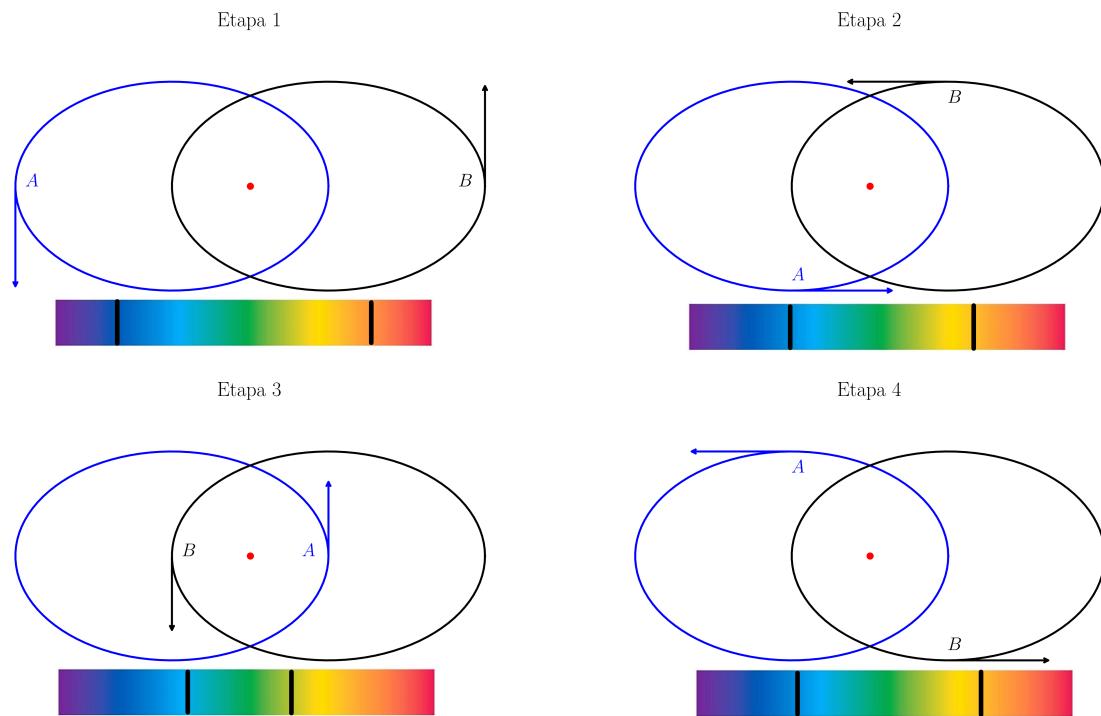


Figura 7.6: Esquema de las líneas espectrales observadas en un sistema binario espectroscópico.

En la etapa 2, ambas se encuentran en una posición en su órbita donde no parece que se alejen ni que se acerquen a la Tierra. Su espectro no se ve afectado por el efecto Doppler. En la etapa 3, la situación es opuesta a la etapa 1: la estrella A parece alejarse y su espectro está desplazado hacia el rojo, mientras que la estrella B parece acercarse y su espectro está desplazado hacia el azul. En la etapa 4, la situación es similar a la etapa 2 y el espectro no se ve afectado por el efecto Doppler.

Ya que la velocidad radial de las estrellas puede calcularse con el efecto Doppler, es

possible construir una gráfica de cómo varía la velocidad con respecto al tiempo. Dicha curva de luz se muestra en la Figura 7.7, donde se observa un comportamiento periódico. Estas son características únicas de los sistemas binarios y gracias a la espectroscopía es posible detectarlos aún cuando se encuentran demasiado lejos para ser vistos por telescopios.

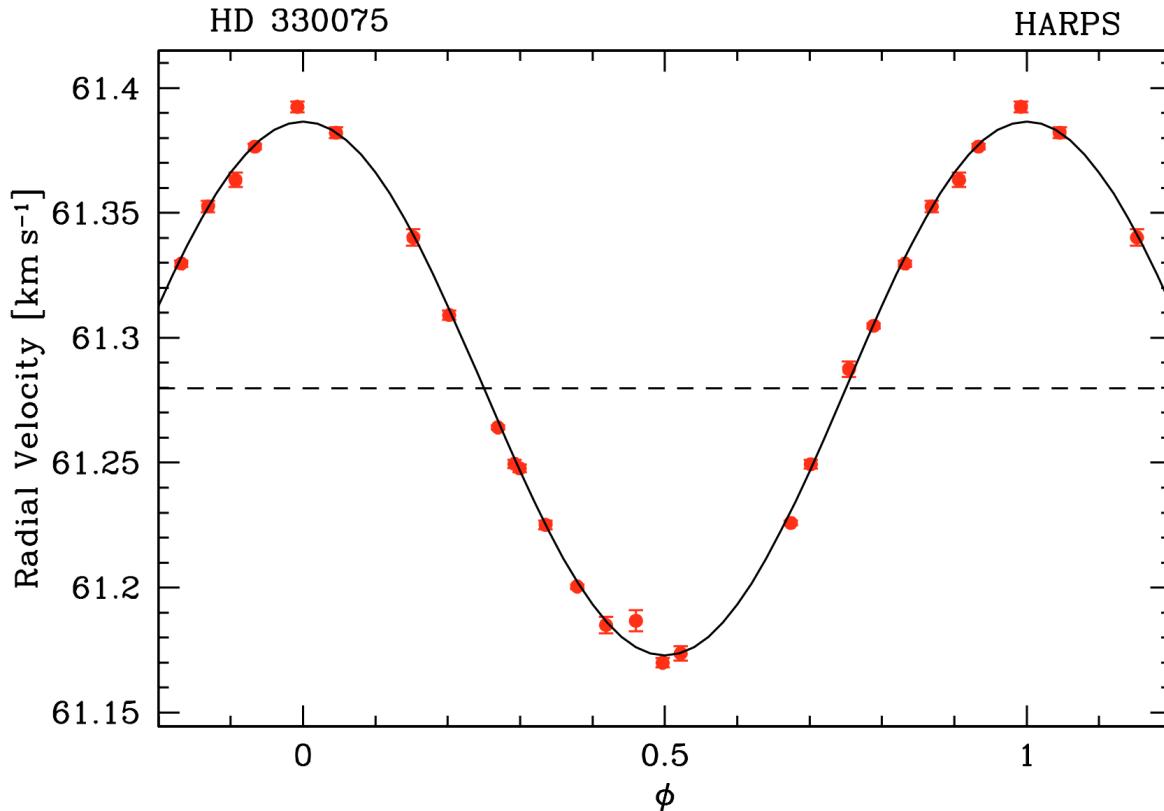


Figura 7.7: Curva de velocidad radial para el sistema binario HD 330075 tomado con el espectrógrafo «High-Accuracy Radial-velocity Planet Searcher» (HARPS). Fuente: [Pepe et al \(2004\)](#).

Clase 8 | Detectores CCD y calibración

Unidad 3

Python para datos CCD

Clase 9 | Visualización de datos con Python

Una de las múltiples aplicaciones de Python es la visualización de datos, que permite crear gráficas de funciones con una o dos variables y generar imágenes a partir de matrices de datos. En Python, estas matrices se representan mediante arreglos numéricos multidimensionales utilizando el módulo `numpy`. Las imágenes astronómicas obtenidas con dispositivos CCD también consisten en arreglos numéricos de dos o más dimensiones, por lo que es esencial aprender a manipular este tipo de datos de manera eficiente antes de visualizarlos.

En esta clase, abordaremos el uso de los módulos `numpy` y `matplotlib`. Con `numpy`, aprenderemos a crear y manejar arreglos numéricos de diversas dimensiones, facilitando el procesamiento y análisis de datos complejos. Posteriormente, utilizaremos `matplotlib` para visualizar estos datos de forma clara y efectiva, creando gráficos e imágenes que nos permitan interpretar y comunicar mejor la información obtenida.

8 Numpy y Matplotlib

8.1 Arreglos numéricos con numpy

Los arreglos, en inglés llamados *arrays*, son una estructura de datos que permiten almacenar múltiples valores en una sola variable. Son similares a las listas, con la diferencia que todos sus elementos son del mismo tipo de dato. El módulo `numpy`, que es una abreviación de *NUMerical PYthon*, está diseñado para realizar tareas matemáticas de cualquier tipo sobre arreglos de manera eficiente. Si por algún motivo tu versión de Python no cuenta con este módulo, puedes instalarlo escribiendo `pip install numpy` en una terminal.

Para usarlo, primero debemos importarlo de la misma manera que hicimos con el módulo `math` anteriormente. Por convención, `numpy` siempre se importa asignándole el alias `np`. Para crear una arreglo, se utiliza la función `array()` definida dentro de `numpy`. Los elementos se escriben separados por comas y encerrados entre corchetes dentro de `array()`. Por ejemplo, para crear un arreglo unidimensional, se hace de la siguiente manera:

```
[1]: import numpy as np  
  
#- Crear un arreglo  
arreglo_1 = np.array([1, 2, 3, 4, 5])
```

Los arreglos también admiten acceder a tus elementos mediante índices. Por ejemplo, para acceder al tercer elemento de arreglo_1:

```
[2]: #- Acceder al tercer elemento (índice 2)  
print(arreglo_1[2])
```

3

Presta atención a la siguiente sintaxis, particularmente a la cantidad de corchetes que se necesitan y en qué ubicación se encuentran para definir un arreglo de dos dimensiones (una matriz):

```
[3]: #- Crear un arreglo bidimensional (matriz de 2x3 elementos)  
arreglo_bidimensional = np.array([[1, 2, 3], [4, 5, 6]])  
arreglo_bidimensional
```

```
[3]: array([[1, 2, 3],  
           [4, 5, 6]])
```

Para acceder a los elementos de una matriz también se utilizan índices, pero se necesitan dos. El primero índice determina el número de fila y el segundo, el número de columna:

```
[4]: #- Acceder al elemento de la primera fila y segunda columna  
print(arreglo_bidimensional[0, 1])
```

2

8.1.1 Arreglos de ceros y unos

Python también permite crear arreglos de manera automática con diversas funciones. Algunas de las más utilizadas son las funciones zeros() y ones(), que generan arreglos que solo contienen ceros y unos, respectivamente. Para crear un arreglo unidimensional de ceros y unos, el argumento de ambas funciones es la cantidad de elementos deseados:

```
[5]: #- Crear un arreglo de ceros con 5 elementos  
np.zeros(5)
```

```
[5]: array([0., 0., 0., 0., 0.])
```

```
[6]: #- Crear un arreglo de unos con 6 elementos  
np.ones(6)
```

```
[6]: array([1., 1., 1., 1., 1., 1.])
```

Para crear un arreglo bidimensional de ceros y unos, entonces el argumento debe ser una tupla con dos elementos. El primero indica la cantidad de filas y el segundo, la cantidad de columnas:

```
[7]: #- Crear una matriz de ceros de 2x3  
np.zeros((2,3))
```

```
[7]: array([[0., 0., 0.],  
          [0., 0., 0.]])
```

```
[8]: #- Crear una matriz de unos de 3x4  
np.ones((3, 4))
```

```
[8]: array([[1., 1., 1., 1.],  
          [1., 1., 1., 1.],  
          [1., 1., 1., 1.]])
```

8.1.2 Arreglos ordenados

También es posible crear arreglos ordenados, que van desde un punto de partida hasta un punto de llegada y con una separación o cantidad de elementos específicos. La primera opción es utilizar la función `arange()`, cuyo comportamiento depende de la cantidad de argumentos que se ingresen. Si no se especifica, `arange()` define por defecto una separación de `1` entre los elementos. Revisa los siguientes ejemplos:

```
[9]: #- Arreglo inicia en 0 y termina en 9  
np.arange(10)
```

```
[9]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[10]: #- Arreglo inicia en 3 y termina en 9  
np.arange(3, 10)
```

```
[10]: array([3, 4, 5, 6, 7, 8, 9])
```

```
[11]: #- Arreglo inicia en 3, termina en 9 con separación 2  
np.arange(3, 10, 2)
```

```
[11]: array([3, 5, 7, 9])
```

Otra opción es utilizar la función `linspace()`, que como su nombre lo indica, genera arreglos linealmente espaciados. Sus argumentos son el punto de partida, el punto de llegada y la cantidad de elementos que tendrá el arreglo. La función `linspace()` sí incluye al punto de llegada:

```
[12]: #- Arreglo inicia en 10, termina en 20, 5 elementos  
np.linspace(10, 20, 5)
```

```
[12]: array([10. , 12.5, 15. , 17.5, 20. ])
```

8.1.3 Arreglos aleatorios

El módulo `numpy` también permite crear arreglos aleatorios que siguen alguna distribución de probabilidad. Para esto se utiliza el submódulo `numpy.random`. En este submódulo existen muchísimas distribuciones de probabilidad que pueden utilizarse. Nosotros solo revisaremos la de la distribución normal y de Poisson.

La distribución normal o gaussiana tiene la forma

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x - \mu)^2}{2\sigma^2}\right],$$

Donde μ es la media de la distribución y σ es la desviación estándar. Para generar un arreglo que siga esta distribución se utiliza la función `numpy.random.normal()` y recibe los siguientes parámetros de entrada en orden: la media, desviación estándar y el tamaño del arreglo. De este modo, para crear un arreglo de 25 elementos que siguen la distribución normal con una media igual a 100 y desviación estándar igual a 5, se usa la siguiente sintaxis:

```
[13]: #- Media 100, desviación 5 y 25 elementos  
np.random.normal(100, 5, 25)
```

```
[13]: array([ 91.39608103,  96.73532073,  99.76330928,  94.25546655,  
          106.48711238,  98.9051047 ,  93.13512854,  95.64941428,  
          99.317889 , 105.10466668, 101.33295227,  98.9041564 ,  
         104.5538657 , 107.75122643, 107.47218624, 103.36162503,  
         97.9425419 ,  98.88489139, 101.26250893, 105.16440413,  
        103.02281608,  93.13852985,  98.38352758, 102.4490247 ,  
        113.70968138])
```

Por otro lado, la distribución de Poisson tiene la forma

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!},$$

y se utiliza para describir la probabilidad de que k eventos ocurran en un intervalo con separación λ . Para crear un arreglo que siga esta distribución se utiliza la función `numpy.random.poisson()` que acepta como parámetros de entrada el valor esperado de que k eventos ocurran y el tamaño del arreglo. Para crear un arreglo de 20 elementos siguiendo la distribución de Poisson con un valor esperado de 5 eventos, se usa la sintaxis:

```
[14]: #- Arreglo poissoniano de 20 elementos con valor esperado 5
np.random.poisson(5, 20)
```

```
[14]: array([2, 5, 7, 3, 7, 4, 6, 8, 4, 7, 2, 4, 4, 5, 1, 9, 4, 6, 4, 1])
```

8.2 Visualización con Matplotlib

Matplotlib es uno de los paquetes más populares en Python para la creación de gráficos y visualización de datos. Es ampliamente utilizada en análisis de datos, ciencia de datos, y otras disciplinas que requieren visualizaciones gráficas para interpretar resultados de manera visual. Por lo general se utiliza extensamente en conjunto con el módulo NumPy.

Es bastante común que en lugar de importar el módulo `matplotlib`, se importe uno de sus submódulos, `pyplot`, ya que es el que nos ayuda a visualizar datos. Por ejemplo, primero importamos `numpy` en conjunto con `matplotlib.pyplot` para poder usarlos en conjunto.

```
[15]: import numpy as np
import matplotlib.pyplot as plt
```

Al igual que con `numpy`, el módulo `matplotlib.pyplot` se importa por convención siempre con el alias `plt`.

8.2.1 Gráfico de funciones

El tipo de gráfico más sencillo de crear es el de una línea recta. Podemos por ejemplo graficar la funciones $y_1 = 5x - 1$ y $y = -2x + 7$. Para eso, definimos a x y las funciones y_1, y_2 de la siguiente manera:

```
[16]: #- Definiendo La variable independiente
x = np.arange(100)

#- Definiendo Las variables dependientes
y_1 = 3 * x - 1
y_2 = -2 * x + 5
```

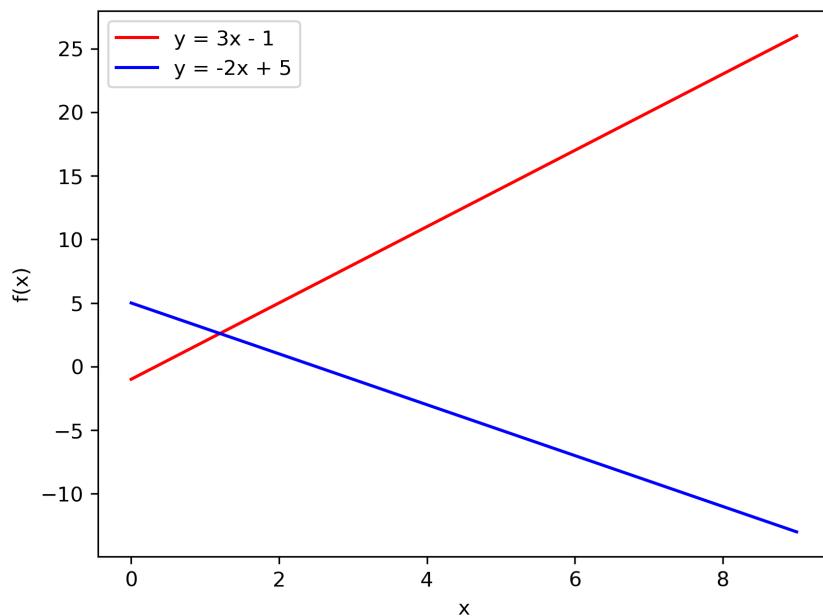
Para visualizarlas, utilizamos la función «`plot()`» definida dentro del submódulo `matplotlib.pyplot` de la siguiente manera:

```
[17]: #- Graficando La primera Línea
plt.plot(x, y_1, c='red', label='y = 3x - 1')
plt.plot(x, y_2, c='blue', label='y = -2x + 5')

#- Muestra las etiquetas
plt.legend()

#- Asigna títulos a los ejes
plt.xlabel('x')
plt.ylabel('f(x)')

#- Muestra la figura
plt.show()
```



8.2.2 Gráfico de histogramas

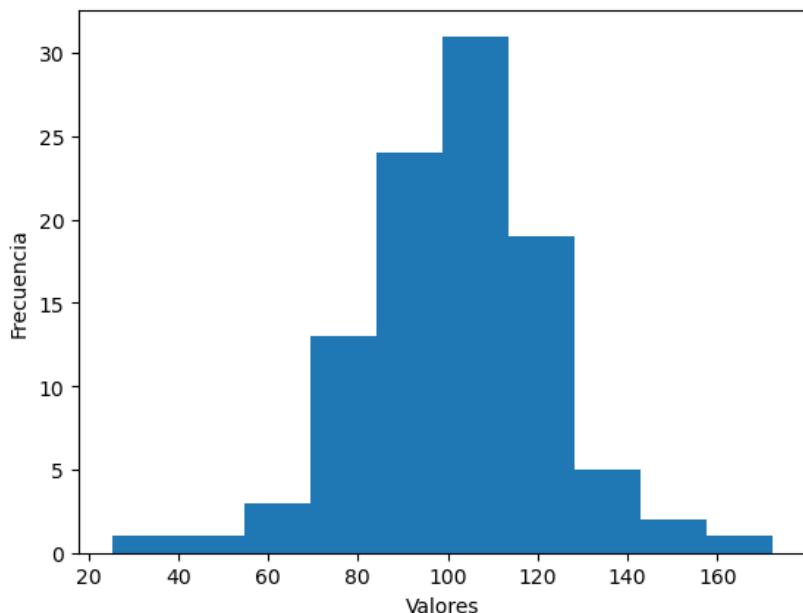
Para crear un histograma se utiliza la función «`hist()`». Por ejemplo, podemos crear un arreglo aleatorio que sigue una distribución normal y visualizarlo con `matplotlib`:

```
[18]: #- Distribución normal
x = np.random.normal(100, 20, 100)

#- Dibujar el histograma
plt.hist(x)

#- Añadir etiquetas a los ejes
plt.xlabel('Valores')
plt.ylabel('Frecuencia')

#- Muestra la figura
plt.show()
```



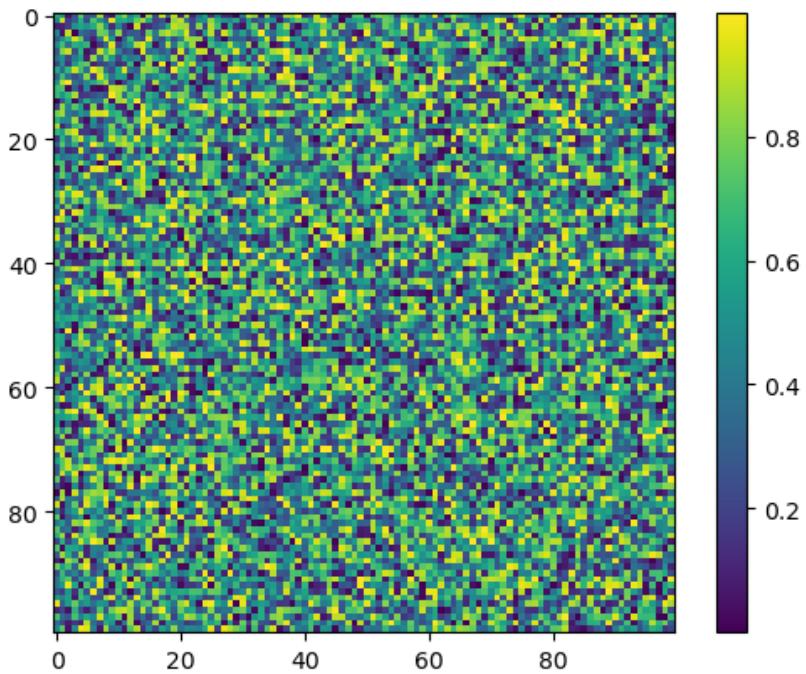
8.2.3 Visualización de matrices

Para visualizar matrices en Python, se utiliza la función «imshow()» definida en `matplotlib.pyplot`. Por ejemplo, vamos a usar una de las funciones de `numpy.random` para generar una matriz con números aleatorios y luego visualizarlos.

```
[19]: #- Matriz de 100x100 con números aleatorios
matrix = np.random.random((100, 100))

#- Dibuja la matriz
plt.imshow(matrix)

#- Muestra la barra de colores
plt.colorbar()
plt.show()
```



8.2.4 Gráfico de una función de dos variables

Matplotlib también permite visualizar gráficas de contorno para funciones de la forma $z = f(x, y)$. Por ejemplo, comencemos definiendo una función en Python que represente este tipo de funciones matemáticas:

```
[20]: def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Para crear un gráfico de contornos se utiliza la función «`plt.contour()`» que toma tres argumentos: una malla de valores de x , otra malla de valores de y y una malla de valores de z . Para crear este tipo de arreglos necesitamos de la función «`np.meshgrid()`», que genera una malla bidimensional a partir de arreglos unidimensionales.

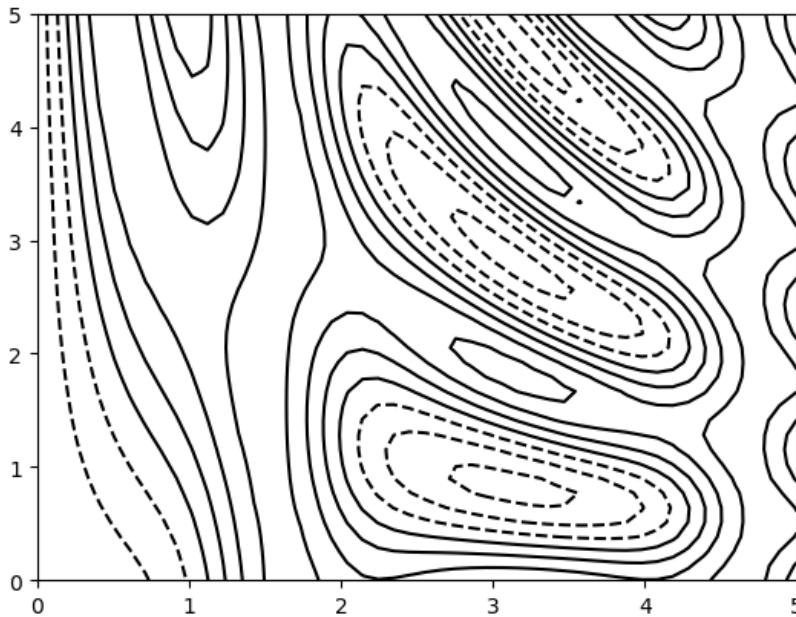
```
[21]: #- Valores de x e y
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

#- Malla de x e y
X, Y = np.meshgrid(x, y)

#- Malla de valores de z
Z = f(X, Y)
```

Y ahora podemos visualizar la función:

```
[22]: plt.contour(X, Y, Z, colors='black')
plt.show()
```



Como último ejemplo, veamos cómo graficar una función gaussiana en dos dimensiones, que tiene la forma:

$$f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{1}{2}\left[\left(\frac{x - \mu_x}{\sigma_x}\right)^2 + \left(\frac{y - \mu_y}{\sigma_y}\right)^2\right]\right).$$

Comenzamos definiendo los valores de x y y :

```
[23]: #- Definir la cuadricula de puntos
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
```

Y ahora podemos establecer los parámetros de la distribución gaussiana en dos dimensiones:

```
[24]: #- Media
mu_x, mu_y = 0, 0

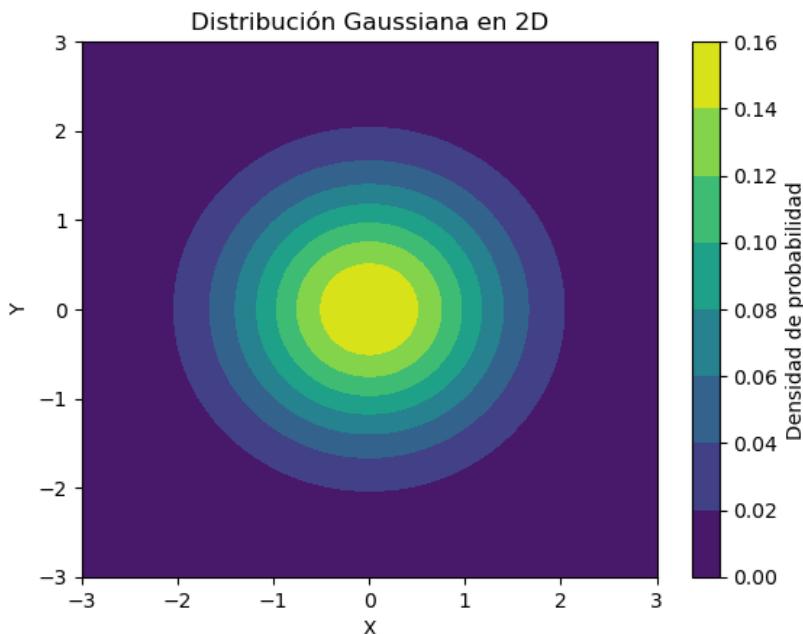
#- Desviación estándar
sigma_x, sigma_y = 1, 1
```

Y ahora calculamos los valores de la distribución:

```
[25]: Z = (1 / (2 * np.pi * sigma_x * sigma_y)) * np.exp(-((X - mu_x)**2 / (2 * sigma_x**2) + (Y - mu_y)**2 / (2 * sigma_y**2)))
```

Finalmente, visualizamos. Esta vez utilizaremos la función «`plt.contourf()`» que es igual a `plt.contour`, pero rellena de color los contornos. Además, usaremos el mapa de colores llamado «`viridis`»:

```
[26]: plt.contourf(X, Y, Z, cmap='viridis')
plt.colorbar(label='Densidad de probabilidad')
plt.title('Distribución Gaussiana en 2D')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Clase 10 | Reducción de datos con Python

Para realizar las tareas básicas de reducción de datos astronómicos usando Python se utiliza el módulo llamado «`ccdproc`». Este módulo es una herramienta poderosa diseñada para la reducción y el procesamiento de datos de imágenes CCD. Está desarrollado como parte del ecosistema de «`astropy`» y proporciona una serie de funciones y clases para manejar tareas comunes en el preprocesamiento de datos astronómicos, como la calibración de imágenes, la corrección de bias, la corrección de flat-field, y la combinación de imágenes. Primero revisaremos el módulo `astropy`.

9 El paquete Astropy

Astropy es un paquete diseñado específicamente para la astronomía y la astrofísica. Cuenta con muchas funciones y módulos, pero nosotros revisaremos algunos de los más utilizados y que nos serán de ayuda para comprender el funcionamiento de `ccdproc`. Para instalar `astropy`, si es que no lo tienes aún, es suficiente con escribir `pip install astropy` en una terminal.

Algunos de los módulos más populares dentro del ecosistema de `astropy` son `units`, `Coordinates`, y `fits`. A continuación realizaremos algunas tareas básicas con estos módulos.

9.1 El módulo `astropy.units`

El módulo «`astropy.units`» permite trabajar con unidades físicas tales como los kilogramos, metros, segundos y cualquiera de sus múltiplos, submúltiplos y además hace posible transformar entre diferentes unidades equivalentes de otros sistemas de medición.

Primero debemos importarlo:

```
[1]: import astropy.units as u
```

Ahora podemos definir una variable que tenga unidades de distancia. Por ejemplo, podemos asignarle unidades de kilopársec:

```
[2]: #- Distancia en kilopársec  
distance = 1 * u.kpc  
distance
```

```
[2]: <Quantity 1. kpc>
```

Como se ve en la celda de código anterior, se crea una «cantidad» al multiplicar cualquier variable numérica por una unidad definida en astropy. En este caso, el kilopársec se obtiene con «u.kpc». Si ahora queremos saber a cuántos metros equivale un kilopársec, entonces podemos usar la función «u.to()» de la siguiente manera:

```
[3]: #- Convertir a metros  
distance.to(u.m)
```

```
[3]: <Quantity 3.08567758e+19 m>
```

Existe otra forma de utilizar la función u.to(), que es un poco más flexible. Para ilustrarlo, intentemos convertir la variable `distance` a varias unidades de distancia, tales como los centímetros, metros, kilómetros, años luz, pársec, nanómetros y unidades astronómicas. Una forma de hacerlo es la siguiente:

```
[4]: distance_units = ['cm', 'm', 'km', 'lyr', 'pc', 'nm', 'au']  
  
#- Distancia equivalente en varias unidades  
print(f'{distance} es equivalente a:\n=====')
```

```
for unit in distance_units:  
    print( distance.to(u.Unit(unit)) )
```

```
1.0 kpc es equivalente a:  
=====  
3.0856775814913673e+21 cm  
3.085677581491367e+19 m  
3.085677581491367e+16 km  
3261.5637771674333 lyr  
1000.0 pc  
3.085677581491367e+28 nm  
206264806.24709633 AU
```

Por supuesto que se pueden utilizar unidades derivadas, tales como las de la velocidad (m/s , por ejemplo) o las de la fuerza ($N \equiv kg\ m\ s^{-2}$). Para esto usamos los operadores de multiplicación y división según sea el caso. Específicamente, para definir cantidades con unidades de velocidad:

```
[5]: velocity = 60 * u.km / u.h  
velocity
```

```
[5]: <Quantity 60. km / h>
```

Obteniendo su equivalente en m/s :

```
[6]: velocity.to(u.m/u.s)
```

```
[6]: <Quantity 16.6666667 m / s>
```

9.2 Trabajando con tablas

Con astropy se pueden leer tablas casi en cualquier formato. Un formato muy popular en astronomía son los archivos con extensión fits (**F**lexible **I**mage **T**ransport **S**ystem). Una de las formas de leer este tipo de archivos es usando la función `Table()` del módulo `astropy.table`. Las imágenes CCD se almacenan en archivos `.fits`. Sin embargo, no es posible leerlas con `Table()`. Afortunadamente, existen más opciones.

Para poder trabajar con imágenes `.fits` provenientes de archivos CCD, se puede utilizar el módulo `fits` que está dentro de `astropy.io`. Toma como ejemplo las imágenes disponibles en este enlace de Google drive: https://drive.google.com/drive/folders/1TxopD_fIa1ZHplm_SH0Gvhe663psuJn-?usp=sharing.

El enlace contiene una carpeta organizada de la siguiente manera: hay un directorio llamado `bias`, otro llamado `flats`, otro llamado `object` y uno llamado `stds`. Los directorios `bias` y `flat` contienen las imágenes `bias` y `flat`, respectivamente. El directorio `object` contiene las imágenes del objeto de interés, también llamadas «*science images*» o imágenes de ciencia. El directorio `stds` contiene imágenes de una estrella estándar, que no usaremos por el momento.

Recomiendo que extraigas el contenido de la carpeta en tu mismo directorio de trabajo donde utilizas los Notebooks de Jupyter para que puedas acceder a sus datos de manera más simple y con las rutas especificadas en el Notebook llamado «Astropy-package».

Primero establecemos las rutas a las imágenes:

```
[7]: #- Ruta a la ubicación de los datos  
bias_image = 'OB0001/bias/0002611406-20200712-OSIRIS-OsirisBias1.fits'
```

```
[8]: flat_image =  
'OB0001/flat/0002615401-20200712-OSIRIS-OsirisSkyFlat1.fits'  
object_image =  
'OB0001/object/0002611437-20200712-OSIRIS-OsirisBroadBandImage1.fits'
```

Para leer estas imágenes con el módulo fits, se hace de la siguiente manera:

```
[9]: #- Leyendo la imagen con astropy  
bias_data = fits.open(bias_image)[0]  
flat_data = fits.open(flat_image)[0]  
object_data = fits.open(object_image)[0]
```

Debes tener en cuenta que las instrucciones anteriores generarán errores si no guardaste los datos en tu directorio actual de trabajo. Si en cualquier momento ocurre un error de compilación, siempre revisa el mensaje de error. Como hemos visto en ocasiones anteriores, Python brinda mucha información en sus mensajes de errores para que puedas corregirlos.

Puedes verificar la información contenida en cada imagen aplicando el atributo header. Por ejemplo para la imagen bias:

```
[10]: bias_data.header
```

SIMPLE	=	T / Fits standard
BITPIX	=	16 / Bits per pixel
NAXIS	=	2 / Number of axes
NAXIS1	=	1049 / Axis length
NAXIS2	=	2051 / Axis length
EXTEND	=	F / File may contain extensions
BSCALE	=	1.000000E0 / REAL = TAPE*BSCALE + BZERO
BZERO	=	3.276800E4 /
ORIGIN	=	'NOAO-IRAF FITS Image Kernel July 2003' / FITS file originator
DATE	=	'2021-10-07T01:52:03' / Date FITS file was generated
IRAF-TLM	=	'2021-10-07T01:52:03' / Time of last modification
OBJECT	=	'BIAS' / Name of the object observed
ORIGFILE	=	'Jul12_204354.fits' / Filename
INSTRUME	=	'OSIRIS' / Instrument Name
DETECTOR	=	'E2V CCD44_82_BI' / Detectors Model
DETSIZE	=	'[1:4096,1:4102]' / Maximum Imaging Pixel Area
DATE-OBS	=	'2020-07-12T20:43:39.592' / Time when starts the first exposure (in fr
ELAPSED	=	'22.969' / Total elapsed time from start to end (s)
...		

Para acceder a la imagen se necesita del atributo data:

```
[11]: bias_data.data
```

```
array([[1039, 1137, 1156, ..., 1203, 29, 881],
       [1037, 1133, 1171, ..., 1192, 28, 892],
       [1044, 1135, 1170, ..., 1204, 29, 896],
       ...,
       [1069, 1160, 1184, ..., 1224, 28, 897],
       [1062, 1157, 1180, ..., 1231, 29, 892],
       [1068, 1160, 1188, ..., 1221, 1224, 102]], dtype=uint16)
```

Como puedes darte cuenta, la imagen CCD es un arreglo de numpy bidimensional. Para poder visualizarlo, necesitamos usar la función `imshow()`. Sin embargo, los datos tienen una cantidad bastante grande de pixeles:

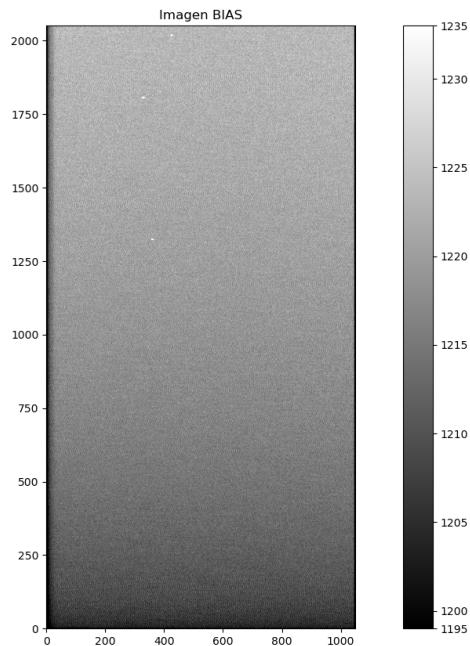
```
[12]: bias_data.data.size
```

```
[12]: 2151499
```

Dada la cantidad de pixeles, es necesario aplicar una función para escalar la imagen. Para eso utilizamos la función `show_image` que está definida dentro del archivo llamado `show_image.py` en el directorio de Notebooks. Importamos esa función y la utilizamos:

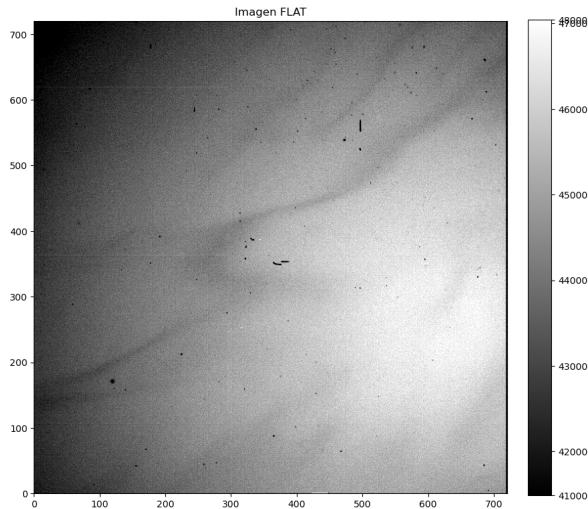
```
[13]: from show_image import show_image
```

```
show_image(bias_data.data, cmap='gray')
plt.title('Imagen BIAS')
plt.show()
```



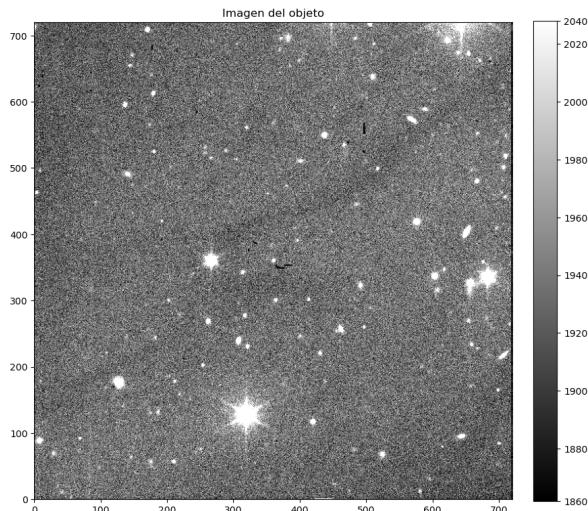
De igual manera podemos visualizar la imagen de campo plano:

```
[14]: show_image(flat_data.data, cmap='gray')
plt.title('Imagen FLAT')
plt.show()
```



Y también la imagen del objeto:

```
[15]: show_image(object_data.data, cmap='gray')
plt.title('Imagen del objeto')
plt.show()
```



9.3 Un ejemplo un poco realista

Ahora revisemos un ejemplo bastante sencillo sobre el proceso de reducir una imagen CCD.

Para eso vamos a generar una imagen sintética, o en otras palabras, a simular una imagen

astronómica. Utilizaremos las funciones definidas dentro del módulo `image_sim` para generar una imagen de bias, una flat y una de dark current. Primero importamos el módulo:

```
[16]: import image_sim as imsim
```

Comenzamos definiendo los parámetros de cada imagen:

```
[17]: stars_exposure = 30.0
dark_exposure = 60.0
dark = 0.1
sky_counts = 20
bias_level = 1100
read_noise = 700
max_stars_counts = 2000
```

Generamos las imágenes de bias, dark current y flat:

```
[18]: bias_with_noise = (imsim.bias(image, bias_level, realistic=True) +
                      imsim.read_noise(image, read_noise))

dark_frame_with_noise = (imsim.bias(image, bias_level, realistic=True) +
                         imsim.dark_current(image, dark, dark_exposure,
                                             hot_pixels=True) +
                         imsim.read_noise(image, read_noise))

flat = imsim.sensitivity_variations(image)
```

Creamos la imagen con estrellas y ruido:

```
[19]: realistic_stars = (imsim.stars(image, 50, max_counts=max_stars_counts) +
                       imsim.dark_current(image, dark, stars_exposure,
                                          hot_pixels=True) +
                       imsim.bias(image, bias_level, realistic=True) +
                       imsim.read_noise(image, read_noise)
                     )
```

Ya que las imágenes simuladas para bias y dark frames no tienen el mismo tiempo de exposición, se necesita hacer una escalación:

```
[20]: scaled_dark_current = stars_exposure * (dark_frame_with_noise -
                                              bias_with_noise) / dark_exposure
```

Y este es nuestro primer intento de reducir una imagen astronómica:

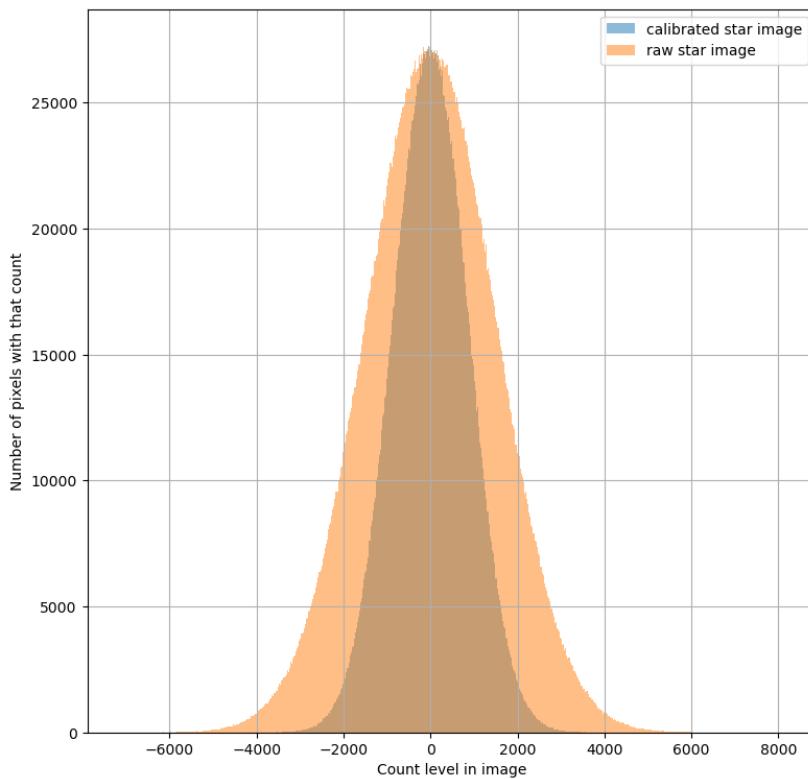
```
[21]: calibrated_stars = (realistic_stars - bias_with_noise -
                           scaled_dark_current) / flat
```

Visualizar la imagen tal vez no nos brinde mucha información sobre si la imagen mejoró o no. Pero podemos observar el histograma de cuentas. Para lograrlo, importamos el siguiente módulo:

```
[22]: from astropy.visualization import hist
```

Ahora graficamos el histograma para la imagen simulada y su correspondiente calibración:

```
[23]: plt.figure(figsize=(9, 9))
hist(calibrated_stars.flatten(), bins='freedman', label='Imagen
calibrada', alpha=0.5)
hist(stars_with_noise.flatten(), bins='freedman', label='Imagen cruda',
alpha=0.5)
plt.legend()
plt.grid()
plt.xlabel('Nivel de cuentas en la imagen')
plt.ylabel('Número de pixeles con esas cuentas')
```



En la gráfica anterior, el ruido de la imagen CCD está representado por el espesor de la distribución del histograma. Vemos que el espesor de la imagen calibrada es menor al de la imagen sin calibrar. En otras palabras, hemos logrado reducir el ruido de la imagen CCD. En las siguientes clases veremos cómo realizar una calibración de manera correcta.

Clase 11 | Imágenes BIAS y DARK

El primer tipo de imágenes con las que trabajaremos son las imágenes de bias y las de corriente oscura. En este documento se presenta un resumen de ambos casos, para más detalles dirígete a los Notebooks `Example-1-BIAS.ipynb` y `Example-1-DARK.ipynb`. Los datos que usaremos se encuentran en la siguiente enlace: <https://zenodo.org/records/3254683>. Debes descargar el archivo y descomprimirlo en tu misma carpeta de trabajo para que las rutas mostradas en los Notebooks funcionen.

10 El módulo ccdproc

Como se mencionó anteriormente, el módulo que se utiliza en Python para la reducción de datos astronómicos es `ccdproc`. Para instalarlo, simplemente escribe `pip install ccdproc` en la terminal cuando tu entorno de Python esté activado.

El módulo `ccdproc` permite combinar archivos de bias, dark current y flat fields, además de corregir las imágenes de objetos por cada una de estas fuentes de ruido. Las imágenes combinadas de cada tipo suelen denominarse "master"(master bias, master dark, master flat). En esta clase aprenderemos a crear este tipo de archivos.

Es importante tener en cuenta que cada análisis de datos CCD es único. Los pasos que revisaremos para este conjunto de datos son apropiados para el instrumento específico, pero podrían no ser aplicables a otros telescopios o instrumentos. Para determinar los pasos adecuados en cada caso, es fundamental conocer la naturaleza de los datos y su organización. En realidad, el aprendizaje de la reducción de datos es una cuestión de práctica y experiencia. Por ello, cuando utilicemos IRAF, trabajaremos con datos de otro instrumento, de modo que puedas ver que, aunque los principios básicos son los mismos, cada procedimiento varía según el instrumento y las características de los datos.

10.1 Creando un archivo master bias

Este ejemplo está basado en la guía «CCD Data reduction guide» de Python. Los datos corresponden al instrumento «Large Field Camera» (LFC) del Observatorio de Palomar. Los

datos se encuentran dentro de la carpeta llamada '`example-cryo-LFC`' y dentro de ella existen imágenes de bias, dark y flat. Comencemos leyendo un archivo de cada tipo:

```
[1]: from ccdproc import CCDData, ImageFileCollection
      import pathlib
      import numpy as np
      from astropy.stats import mad_std

[2]: #- Definiendo La ruta a Los datos
      cryo_path = pathlib.Path('example-cryo-LFC')

      #- Leyendo unas cuantas imágenes
      bias_lfc    = CCDData.read(cryo_path / 'ccd.001.0.fits', unit='count')
      object_lfc  = CCDData.read(cryo_path / 'ccd.037.0.fits', unit='count')
      flat_lfc    = CCDData.read(cryo_path / 'ccd.014.0.fits', unit='count')
```

El CCD de este instrumento cuenta con una sección llamada «overscan», que consiste de una parte cubierta del CCD. Se trata de una región adicional que no corresponde a ninguna parte de la imagen física, sino a píxeles ficticios que se utilizan para medir y corregir el nivel de bias. Los detalles técnicos del LFC pueden encontrarse en su [página oficial](#), donde se muestra que sus imágenes cuentan con 2048×4096 píxeles. Sin embargo, cuando leemos los datos nos damos cuenta que existen píxeles extra, los cuales conforman el overscan:

```
[3]: #- Obteniendo La cantidad de pixeles
      bias_lfc.shape
```

```
[3]: (4128, 2080)
```

Al hacer un gráfico de la intensidad de la señal con respecto a cada pixel obtenemos lo mostrado en la Figura 11.1. Lo que observamos es que a partir del pixel 2048, la señal detectada para el objeto y para las imágenes flat comienzan a disminuir y eventualmente alcanzan la misma intensidad que las imágenes de bias. El hecho de que se observe esta caída y que además se mantiene constante nos indica que sí podemos utilizar la información que nos brinda el overscan. Lo que debemos hacer es restar el valor constante que se alcanza luego del pixel ≥ 2048 a todas nuestras imágenes, con eso estaríamos al mismo tiempo corrigiendo por bias.

Primero debemos crear una carpeta donde colocaremos las imágenes a las que aplicaremos esta corrección. Preferiblemente deberá estar ubicada en tu mismo directorio de trabajo. Puedes por ejemplo llamar a esta carpeta '`example-reduced`' y crearla de la siguiente manera:

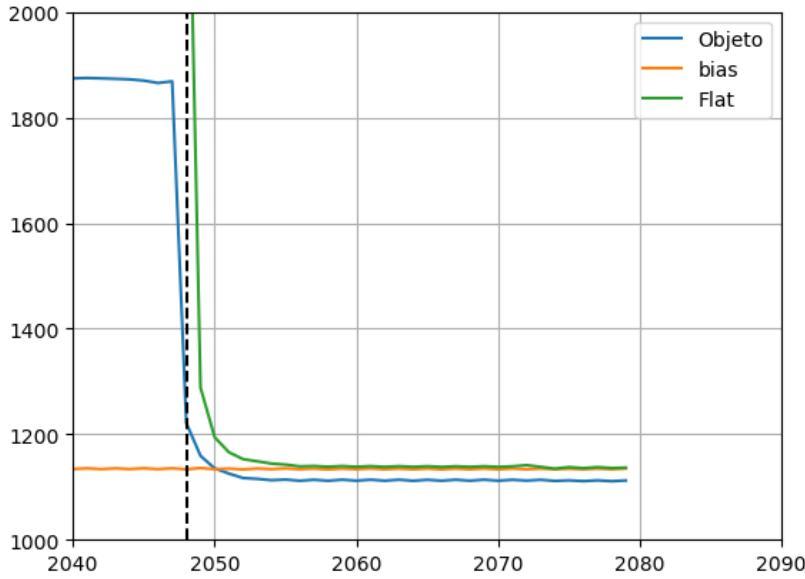


Figura 11.1: Intensidad con respecto a los pixeles

```
[4]: # Crea una nueva carpeta
calibrated_data = Path('.', 'example-reduced')
calibrated_data.mkdir(exist_ok=True)
```

Ahora creamos una colección de archivos con todas las imágenes dentro de la carpeta '`example-cryo-lfc`'. Esta colección tendrá las imágenes bias, flat y las de objeto.

```
[5]: files = ImageFileCollection(cryo_path)
```

Para quedarnos únicamente con las de bias, que son las que nos interesan por el momento, debemos filtrarlas con una de las funciones dentro del módulo `ImageFileCollection`. Dicha función se llama `files_filtered()` y podemos usarla de la siguiente manera:

```
[6]: # Seleccionando solo imágenes tipo bias
biases = files.files_filtered(include_path=True, imagetyp='BIAS')
```

Para entender qué es lo que haremos con todas estas imágenes, primero aplicaremos el procedimiento a una sola imagen y veremos el resultado. Comencemos seleccionando la primera imagen de nuestra lista:

```
[7]: # Seleccionando una sola imagen
first_bias = CCDData.read(biases[0], unit='adu')
```

Como vemos en la Figura 11.1, a pesar que el overscan de este instrumento inicia a partir del píxel 2048, la intensidad se mantiene constante para los tres tipos de imágenes a partir

del píxel 2055 aproximadamente. Por lo tanto, debemos sustraer ese valor de intensidad a nuestra imagen. Primero importamos la función que nos permite realizar esa tarea:

```
[8]: from ccdproc import subtract_overscan
```

Para estos datos específicos, la usamos de la siguiente manera:

```
[9]: bias_overscan_subtracted = subtract_overscan(first_bias,  
                                                overscan=first_bias[:, 2055:],  
                                                median=True)
```

Ahora debemos recortar la imagen para no utilizar los píxeles extra y quedarnos solo con los píxeles físicos. Primero importamos la función que utilizaremos:

```
[10]: from ccdproc import trim_image
```

Y la utilizamos:

```
[11]: trimmed_bias = trim_image(bias_overscan_subtracted[ :, :2048])
```

En la Figura 11.2 se compara la imagen inicial, llamada `first_bias` y la imagen final, llamada `trimmed_bias`. Como puedes notar, se redujo la intensidad de la imagen además que en el eje horizontal ahora hay menos píxeles.

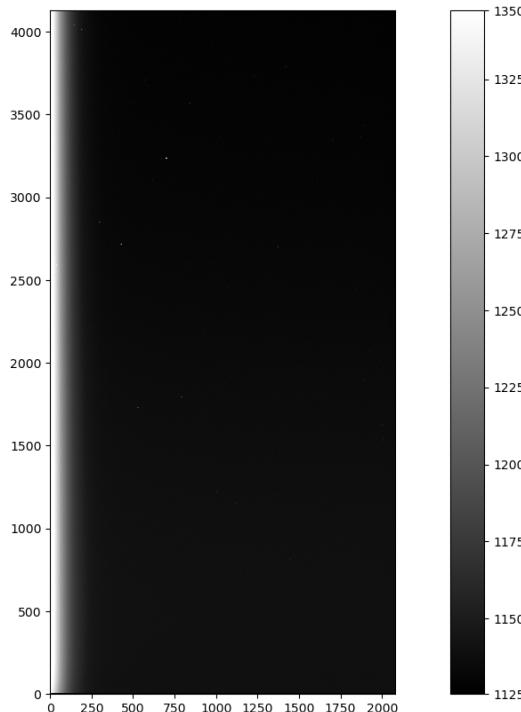


Figura 11.2: Comparación de una imagen de bias antes y después de sustraer el overscan.

Ahora que hemos comprendido el resultado de este procedimiento, podemos aplicarlo a todas las imágenes de bias usando un ciclo **for**:

```
[12]: for ccd_file, file_name in files.ccds(imagetyp='BIAS',
                                             ccd_kwarg...  
#- Restar el overscan  
ccd_file = subtract_overscan(ccd_file,  
                             overscan=ccd_file[:, 2055:],  
                             median = True)  
  
#- Recortar los archivos  
ccd_file = trim_image(ccd_file[:, :2048])  
  
#- Guardar el resultado  
ccd_file.write(calibrated_data / file_name )
```

Con los datos corregidos, vamos a crear un archivo master bias, que es simplemente un promedio de todas las imágenes de bias. Primero seleccionamos las imágenes que acabamos de corregir:

```
[13]: #- Creando una colección de imágenes  
reduced_images = ImageFileCollection(calibrated_data)  
  
calibrated_biases = reduced_images.files_filtered(imagetyp='bias',  
                                                 include_path=True)
```

Para crear el master bias, debemos usar la función `combine()` del módulo `ccdproc`. Primero la importamos:

```
[14]: from ccdproc import combine
```

La utilizamos de la siguiente manera:

```
[15]: #- Creando un master bias  
combined_bias = combine(calibrated_biases,  
                         method='average',  
                         sigma_clip=True, sigma_clip_low_thresh=5,  
                         sigma_clip_high_thresh=5,  
                         sigma_clip_func=np.ma.median,  
                         sigma_clip_dev_func=mad_std,  
                         mem_limit=350e6  
)
```

Actualizamos la información del archivo y además lo guardamos:

```
[16]: #- Actualizando la meta data  
combined_bias.meta['combined'] = True  
  
#- Guardando el archivo  
combined_bias.write(calibrated_data / 'master_bias.fits')
```

10.2 Creando un archivo master dark

Para crear el archivo master dar, primero debemos sustraer el overscan y recortar las imágenes de dark current. Ya hemos definido la variable `cryo_path` como la ubicación de nuestros archivos. Dentro de esa ubicación se encuentra el directorio llamado `darks`. Para acceder a esos archivos, por lo tanto, debemos definir esa ruta:

```
[17]: #- Ruta a las imágenes dark  
lfc_darks_raw = ImageFileCollection(cryo_path / 'darks')
```

Aplicamos el mismo ciclo `for` a todas las imágenes dark para restarles el overscan y recortarlas:

```
[18]: for ccd, file_name in lfc_darks_raw.ccds(imagetype='DARK',  
                                              ccd_kwarg={ 'unit': 'adu' },  
                                              return_fname=True  
                                              ):  
    # Subtract the overscan  
    ccd = subtract_overscan(ccd, overscan=ccd[:, 2055:],  
                           median=True)  
  
    # Trim the overscan  
    ccd = trim_image(ccd[:, :2048])  
  
    # Save the result  
    ccd.write(calibrated_data / file_name, overwrite=True)
```

Ahora debemos actualizar nuestra colección de imágenes reducidas:

```
[19]: reduced_images.refresh()
```

Las imágenes dark tienen diferentes tiempos de exposición, por lo que debemos crear un archivo master dark para cada uno de ellos. Primero creamos un set con dichos tiempos:

```
[20]: darks = reduced_images.summary['imagetype'] == 'DARK'  
dark_times = set(reduced_images.summary['exptime'][darks])  
print(dark_times)
```

```
{300.0, 70.0, 7.0}
```

Ahora usamos un ciclo for que itera sobre cada tiempo de exposición y luego:

- Selecciona todas las imágenes dark para el tiempo de exposición dado
- Combina esas imágenes dark
- Actualiza su información
- Guarda los archivos combinados

```
[21]: for exp_time in sorted(dark_times):  
  
    # Seleccióna las imágenes dark dentro de la carpeta  
    calibrated_darks = reduced_images.files_filtered(imagetype='dark',  
                                                       exptime=exp_time,  
                                                       include_path=True)  
  
    # Combina las imágenes  
    combined_dark = combine(calibratwrite(ed_darks,  
                                           method='average',  
                                           sigma_clip=True, sigma_clip_low_thresh=5,  
                                           sigma_clip_high_thresh=5,  
                                           sigma_clip_func=np.ma.median,  
                                           sigma_clip_dev_func=mad_std,  
                                           mem_limit=350e6  
                                           )  
  
    # Actualiza la información  
    combined_dark.meta['combined'] = True  
  
    # Guarda Los archivos  
    dark_file_name = 'combined_dark_{:6.3f}.fits'.format(exp_time)  
    combined_dark.write(calibrated_data / dark_file_name)
```

Con esto hemos creado los tres archivos master dark, uno para cada tiempo de exposición.

Clase 12 | Master flat y calibración final

En las clases anteriores aprendimos a crear archivos master bias y master dark. Lo único que queda por hacer es crear un archivo master flat con las imágenes de flat fields y así podremos finalmente realizar la calibración básica de nuestros datos. Esta tarea también se consigue utilizando el paquete ccdproc. También utilizaremos unas cuantas funciones de astropy y numpy, así como las herramientas de visualización de Python.

11 Calibración de imágenes

Ahora trabajaremos con las imágenes de flat fields, con las que vamos a necesitar realizar un poco más de tareas, pero nada muy complicado. Recuerda que este documento es solo un resumen de los pasos necesarios para realizar la calibración, para más detalles de los pasos y los códigos, debes revisar el Notebook llamado Example-FLAT.ipynb.

11.1 Creando un archivo master flat

Volveremos a utilizar el módulo ccdproc junto con sus submódulos y funciones. Por lo tanto, como es usual, primero importamos todo lo que usaremos en nuestro código:

```
[1]: #- Para trabajar con rutas
      from pathlib import Path

      #- Modulos de astropy y ccd proc
      from astropy import units as u
      from astropy.nddata import CCDData
      from ccdproc import ImageFileCollection
      from ccdproc import subtract_overscan, trim_image, subtract_dark

      #- Módulos para estadística
      import numpy as np
      from astropy.stats import mad_std

      #- Modulos para visualización
      from show_image import show_image
```

Como encontramos anteriormente, las imágenes de corriente oscura tienen diferentes tiempos de exposición. Las imágenes de flat también tienen diferentes tiempos de exposición. Para poder corregir las imágenes flat por corriente oscura, necesitamos agruparlas de acuerdo al tiempo de exposición para que coincidan con las de corriente oscura. La función que nos ayudará a conseguirlo se llama `find_nearest_dark_exposure()` y debemos importarla.

```
[2]: from functions import find_nearest_dark_exposure
```

Ya que hemos definido los módulos a utilizar, podemos ahora definir las variables que nos darán acceso a los archivos:

```
[3]: # Definiendo la ruta a los datos
cryo_path = pathlib.Path('example-cryo-LFC')
calibrated_data = Path('example-reduced')

# Selecciona las imágenes
files = ImageFileCollection(cryo_path)
reduced_images = ImageFileCollection(calibrated_data)
combined_dark_files = reduced_images.files_filtered(imagetyp='dark',
                                                       combined=True)
```

Al igual que con las imágenes de bias y corriente oscura, necesitaremos restar el overscan y recortar las imágenes flat. Adicionalmente, también debemos corregir por corriente oscura. Primero veamos cómo hacerlo para una sola imagen y luego lo aplicaremos a todas. Seleccionamos una sola imagen flat:

```
[4]: # Seleccionando una imagen flat
a_flat = CCDData.read(files.files_filtered(imagetyp='flatfield',
                                             include_path=True)[0], unit='adu')
```

Y ahora podemos corregirla por overscan y recortarla:

```
[5]: # Restar el overscan
a_flat_reduced = subtract_overscan(a_flat,
                                      overscan=a_flat[:, 2055:], median=True)

# Recortar
a_flat_reduced = trim_image(a_flat_reduced[:, :2048])

# Visualizar
show_image(a_flat_reduced, cmap='gray')
plt.title('Single flat frame, overscan subtracted and trimmed')
```

En la Figura 12.1 se muestra una comparación entre la imagen flat sin procesar y luego de ser procesada.

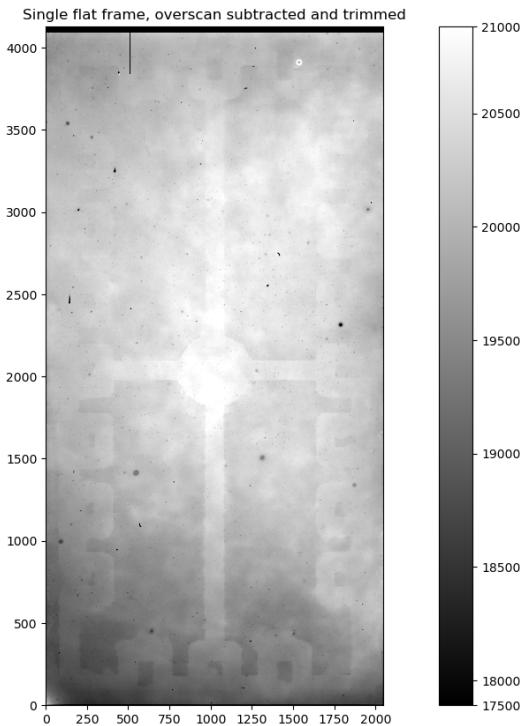


Figura 12.1: Una imagen flat antes y después de restar el overscan y recortarla

Revisamos los tiempos de exposición para las imágenes flat:

```
[6]: set(files.summary['exptime'][files.summary['imagetyp'] == 'FLATFIELD'])  
  
{7.0, 70.001, 70.011}
```

Y comparamos con los de corriente oscura:

```
[7]: actual_exposure_times = set(h['exptime'] for h in  
reduced_images.headers(imagetyp='dark', combined=True))  
actual_exposure_times  
  
{7.0, 70.0, 300.0}
```

Comprobamos a cuál de estos tiempos de exposición está más cercano el de la imagen flat con ayuda de la función `find_nearest_dark_exposure`:

```
[8]: closest_dark = find_nearest_dark_exposure(a_flat_reduced,  
actual_exposure_times)  
closest_dark
```

```
[8]: 70.0
```

Ahora creamos un diccionario con las tres imágenes master dark que generamos anteriormente:

```
[9]: combined_darks = {ccd.header['exptime']: ccd for ccd in  
reduced_images.ccds(imagetype='dark', combined=True)}
```

Y ahora restamos el master dark correspondiente a los 70 s de exposición a la imagen flat que seleccionamos:

```
[10]: a_flat_reduced = subtract_dark(a_flat_reduced,  
combined_darks[closest_dark],  
exposure_time='exptime',  
exposure_unit=u.second, scale=False)
```

Este último archivo está corregido por overscan, está recortado y además está corregido por corriente oscura. Ahora podemos aplicar este procedimiento a todas las imágenes flat en un bucle **for** de la siguiente manera:

```
[11]: for ccd, file_name in files.ccds(imagetype='FLATFIELD',  
ccd_kwarg={'unit': 'adu'},  
return_fname=True):  
    # Subtract the overscan  
    ccd = subtract_overscan(ccd, overscan=ccd[:, 2055:], median=True)  
  
    # Trim the overscan  
    ccd = ccdp.trim_image(ccd[:, :2048])  
  
    # Find the correct dark exposure  
    closest_dark = find_nearest_dark_exposure(ccd,  
                                              actual_exposure_times)  
  
    # Subtract the dark current  
    ccd = subtract_dark(ccd, combined_darks[closest_dark],  
                       exposure_time='exptime',  
                       exposure_unit=u.second)  
  
    # Save the result;  
    ccd.write(calibrated_data / ('flat-' + file_name))
```

Antes de combinar los flats, debemos actualizar la carpeta de imágenes calibradas y separarlas por sus filtros. En este conjunto de datos en particular, se utilizaron los filtros i y g:

```
[12]: #- Actualizando el contenido de la carpeta
reduced_images.refresh()

#- Definiendo el tipo de imagen
flat_imagetyp = 'flatfield'

#- Buscando los filtros usados en este tipo de imagen
flat_filters = set(h['filter'] for h in
                    reduced_images.headers(imagetyp=flat_imagetyp))
flat_filters

{"g'", "i'"}  


```

Vamos a definir una función que nos ayudará a mejorar la estadística al momento de combinar los flats.

```
[13]: def inv_median(a):
    return 1 / np.median(a)
```

Ahora creamos el master flat para cada filtro:

```
[14]: for filt in flat_filters:
    to_combine = reduced_images.files_filtered(
                    imagetyp=flat_imagetyp,
                    filter=filt, include_path=True)

    combined_flat = ccdp.combine(to_combine,
                                method='average',
                                scale=inv_median,
                                sigma_clip=True,
                                sigma_clip_low_thresh=5,
                                sigma_clip_high_thresh=5,
                                sigma_clip_func=np.ma.median,
                                signma_clip_dev_func=mad_std,
                                mem_limit=350e6
                                )

    combined_flat.meta['combined'] = True

    flat_file_name =
    'combined_flat_filter_{}.fits'.format(filt.replace("'", "p"))
    combined_flat.write(calibrated_data / flat_file_name)
```

Actualizamos nuevamente la carpeta de imágenes calibradas:

```
[15]: #- Actualizando el contenido de la carpeta  
reduced_images.refresh()
```

En la Figura 12.2 se muestran las imágenes master flat para cada filtro:

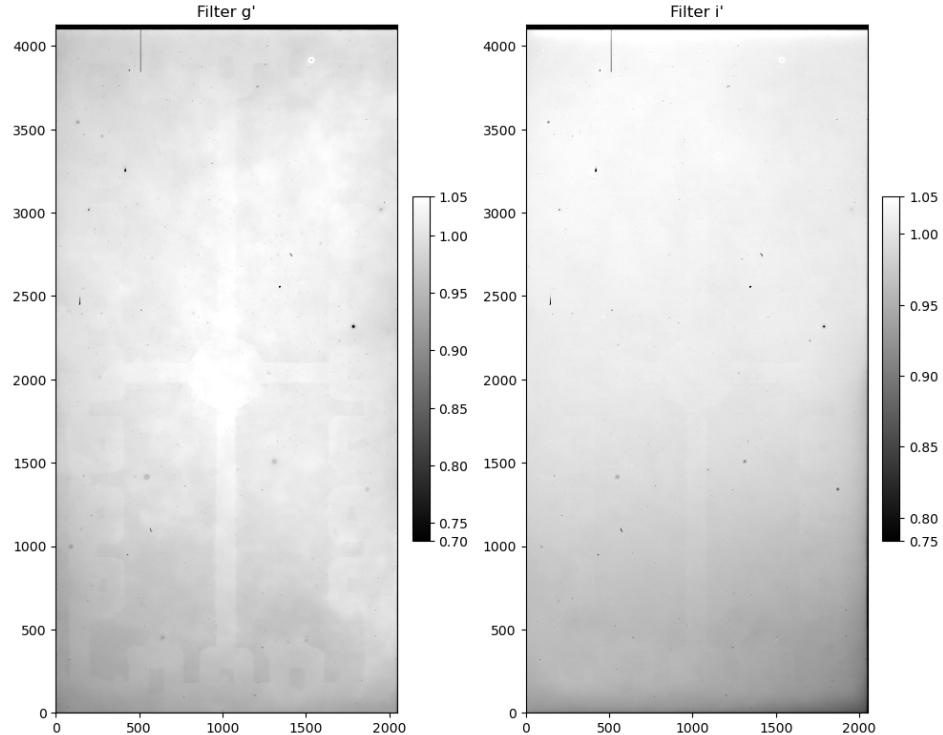


Figura 12.2: Archivos master flats para cada filtro.

11.2 Calibrando las imágenes de objeto

En nuestra carpeta de imágenes sin procesar, solo hay dos imágenes de objeto. Una para el filtro *g* y otra para el filtro *i*. Estas dos imágenes son a las que les aplicaremos las correcciones con los archivos master bias, master dark y master flat que ya hemos creado. Primero definimos un par de variables:

```
[16]: #- Definiendo algunas variables  
science_imagetyt = 'object'  
exposure = 'exptime'
```

Definimos un diccionario con los master flat:

```
[17]: combined_flats = {ccd.header['filter']: ccd for ccd in  
reduced_images.ccdfs(imagetyt=flat_imagetyt, combined=True)}
```

Y finalmente realizamos la calibración básica con un bucle **for**:

```
[18]: #- Definiendo listas vacías
all_reds = []
light_ccds = []

#- Ciclo for para reducir las imágenes
for light, file_name in files.ccds(imagetyp=science_imagetyp,
return_fname=True, ccd_kwargs=dict(unit='adu')):

    #- Incluyendo un nuevo elemento a la lista
    light_ccds.append(light)

    #- Restando overscan de la image de objeto
    reduced = ccdp.subtract_overscan(light, overscan=light[:, 2055:], median=True)

    #- Cortando la imagen de objeto
    reduced = ccdp.trim_image(reduced[:, :2048])

    #- Decidiendo cuál tiempo de exposición usar
    closest_dark = find_nearest_dark_exposure(reduced,
combined_darks.keys())

    #- Corrigiendo por corriente oscura
    reduced = ccdp.subtract_dark(reduced,
combined_darks[closest_dark],
exposure_time=exposure,
exposure_unit=u.second)

    #- Seleccionando el flat adecuado de acuerdo al filtro
    good_flat = combined_flats[reduced.header['filter']]

    #- Corrigiendo por flat
    reduced = ccdp.flat_correct(reduced, good_flat)

    #- Incluyendo un nuevo elemento a la lista
    all_reds.append(reduced)

    reduced.write(calibrated_data / file_name)
```

Los resultados de la calibración se muestran en la Figura 12.3.

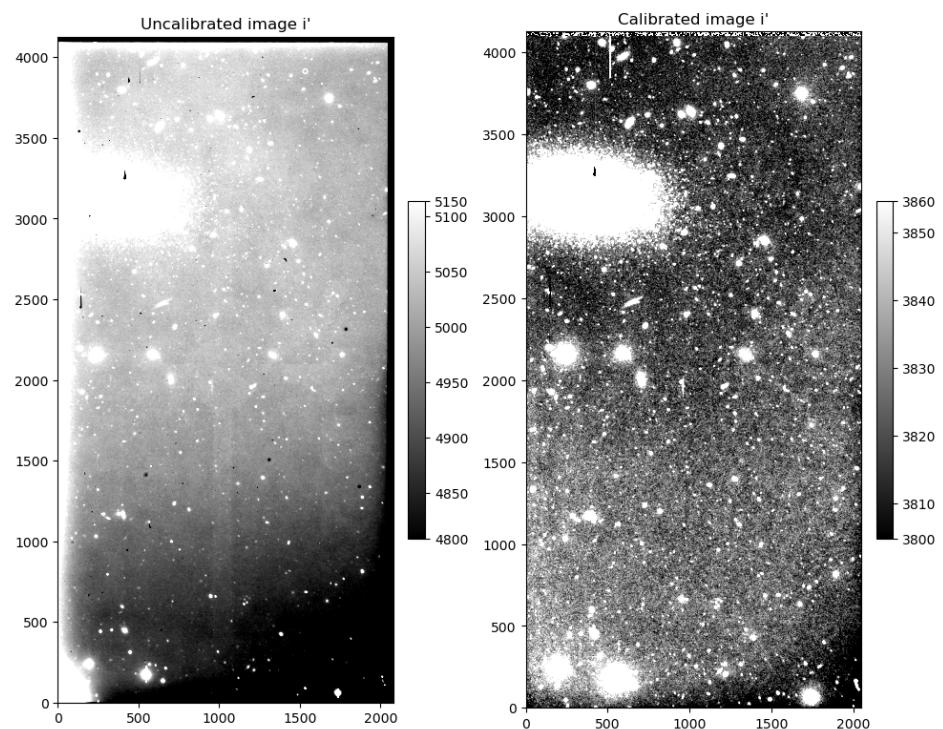
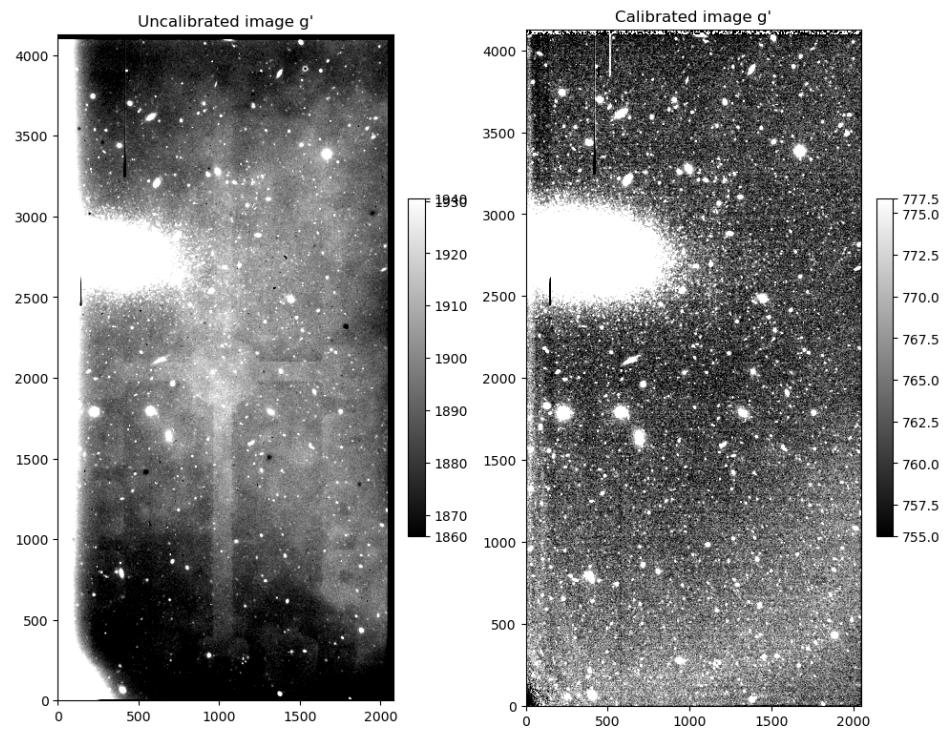


Figura 12.3: Comparación de imágenes calibradas y no calibradas en diferentes filtros