Capítulo 2

Programación escencial en Python

El proceso de programar involucra el idear acciones, y ordenarlas de una forma tal que nos permita resolver un problema de forma adecuada. En el ámbito de la computación, cuando elaboramos programas tenemos como objetivo darles una utilidad a éstos tal que nos permitan resolver problemas específicos. En este capítulo nos enfocaremos en el proceso de programar utilizando el lenguaje Python. el objetivo es conocer los elementos básicos de programación enfocados en la aplicabilidad a problemas relacionados a las ciencias físicas y astronómicas.

2.1. Acerca de Python

Python es un lenguage de programación interpretado, multiparadigma, y que hace incapié en su legibilidad, y simplicidad. Cuenta con un set bastante acotado de palabras clave propias del lenguaje, y una cantidad muy grande de librerias adicionales diseñadas para distintas labores. Este lenguaje ha ganado gran relevancia entre la comunidad científica por su fácil implementación, su aplicabilidad en distintos OS, y la gran cantidad de colaboradores en el desarrollo de distintas tareas.

Python nace en los años noventa, y el ser un lenguaje interpretado genera simplicidad en su implementación pero una baja en el rendimiento, lo que ha ido mejorando con el tiempo pero sigue bajo los rendimientos de los lenguajes compilados, tales como C, C++, y Fortran. Cuando ejecutamos python, por *default* se iniciará el interprete en nuestro sistema, por ejemplo:

```
username@machine:$ python
Python 3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 16:07:37)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Acá vemos que los símbolos $>>>$ indican que el intérprete está a la espera de instrucciones. Escribamos entonces nuestra primera instrucción para comprender cómo Python **intepreta y ejecuta** las instrucciones asignadas. Para comenzar utilizaremos la palabra clave `print` de la siguiente forma –recuerde presionar `Enter`, luego de dar la instrucción–:

```
username@machine:$ python
Python 3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 16:07:37)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hola Mundo!")
Hola Mundo!
>>>
```

Observamos entonces que nuestra instrucción `print("Hola Mundo!")` es interpretada y ejecutada, para luego desplegar en pantalla el texto `"Hola Mundo!"`. Fe-

licitaciones, has escrito tu primera instrucción de Python. Python, cuenta además con una extensa documentación que puede ser accedida online, o bien dentro del mismo interprete, por ejemplo y tal como lo menciona el mensaje de bienvenida, pordemos escribir help para mayor información

```
username@machine:$ python
Python 3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 16:07:37)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> help
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Ahora el intérprete ha cambiado, y podemos ver que el sistema esta en `help>` a la espera de instrucciones. Podemos listar modulos, palabras reservadas, simbolos, o tópicos simplemente escribiendo sobre esto, como se observa en el mensaje anterior. Por ejemplo podemos ver las palabras reservadas con la instrucción `keyword`.

```
help> keywords
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not

help>
```

Luego podemos pedir más ayuda, por ejemplo con la palabra clave `def`. Note que la documentación puede mostrarse *paginada*, por lo que puede terminar de *leer* la documentación, debe presionar la tecla `q` (del inglés, *quit*). Siempre es recomendable conocer del interprete, como acceder a él, como salir de él, como definir variables, como usarlo rapidamente para pequeñas *pruebas*. El intérprete también puede ser utilizado como una calculadora, para resolver algún problema de forma rápida. Consideremos por ejemplo los siguientes casos :

```
username@machine:$ python
Python 3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 16:07:37)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 5 + 254.5
259.5
>>> 1 + 5
7
>>>
```

Vemos que en ambos casos `python` reconoce automáticamente si escribimos numeros enteros o reales en una operación (por lo que entrega, o *retorna* un entero o real). A continuación discutiremos brevemente sobre los tipos de datos que soporta python.

2.2. Tipos de datos

Python es un lenguaje de programación de tipado dinámico, lo que significa que no es necesario declarar explícitamente el tipo de una variable antes de usarla. Python infiere automáticamente el tipo de la variable según el valor asignado. A continuación, se mencionan algunos tipos de variables comunes en Python:

- Enteros (int): Representan números enteros, como 1, 42 o -100.
- Flotantes (float): Representan números reales, como 3.14, 0.001 o -1.5.
- Cadenas de caracteres (str): Representan secuencias de carácteres, como `"Hola, mundo!"`, o `"Python"`.
- Listas (list): Representan secuencias ordenadas de elementos, como `[1, 2, 3]`.
- Tuplas (tuple): Representan secuencias ordenadas e inmutables de elementos, como `(1, 2, 3)` o `("a", "b", "c")`.
- Diccionarios (dict): Representan conjuntos de pares clave-valor, como `{"key1": "valor", "key2": "valor"}`.
- Conjuntos (set): Representan conjuntos no ordenados de elementos únicos, como `{1, 2, 3}` o `{"a", "b", "c"}`.

2.2.1. Ejemplo 1: Conversión entre tipos de variables

El siguiente código, ha sido escrito en un archivo de texto, que posee el *set* de instrucciones de python que deseamos ejecutar. Los ficheros con códigos python suelen llevar una extensión `.py`. Puede utilizar el editor o gestor IDE que más le acomode para escribir su programa, y luego ejecutarlo como se muestra a continuación. Este código convierte entre diferentes tipos de variables en Python:

```
numero_entero = 42
numero_flotante = float(numero_entero)
cadena = str(numero_entero)

print(f'Número Entero: {numero_entero}')
print(f'Número Flotante: {numero_flotante}')
print(f'Cadena: {cadena}')
```

Al ejecutar este programa, obtenemos la salida :

```
username@machine:$ python3 example.py
Número Entero: 42
Número Flotante: 42.0
Cadena: 42
uername@machine:$
```

Podemos observar que el valor permanece constante, y la impresión del número flotante presenta un decimal de precisión en su presentación standard. Para un usuario final la muestra del número entero y la cadena de texto no parecen tener diferencia, sin embargo desde el punto de vista de la programación son completamente distintas.

Ejercicio:

Escriba un programa en Python, similar al ejemplo anterior, pero que a partir de un texto incial convierta éste a diferentes tipos de variables, tales como enteros, reales, y complejos.

Hechos interesantes sobre las variables en Python

- A diferencia de otros lenguajes de programación, en Python las variables son en realidad referencias a objetos, lo que significa que cuando se asigna una variable a otra, ambas variables apuntan al mismo objeto en memoria.
- Python tiene una función incorporada llamada `type()` que devuelve el tipo de una variable. Por ejemplo, `type(42)` devuelve `<class 'int'>`.
- Python también admite operaciones con números complejos, utilizando el tipo complejos (complex), que representa números complejos, como 1 + 2j o 3 - 4j. Por ejemplo, para crear un número complejo en Python, se puede utilizar la siguiente sintaxis: `numero_complejo = 1 + 2j`.
- Las variables en Python pueden ser reasignadas a valores de diferentes tipos en cualquier momento. Por ejemplo, una variable que originalmente contenía un entero puede ser reasignada a una cadena de caracteres: `mi_variable = 42; mi_variable = "Hola, mundo!"`.
- Se pueden almacenar variales en listas, tuplas y diccionarios. Estas pueden contener elementos de diferentes tipos, lo que permite estructuras de datos muy flexibles y personalizables.

2.3. Listas en Python

Las listas son una de las estructuras de datos más utilizadas en Python. Son secuencias ordenadas de elementos, que pueden ser de diferentes tipos (números, cadenas de caracteres, otras listas, etc.). Las listas son mutables, lo que significa que sus elementos pueden ser modificados después de su creación.

2.3.1. Creación y manipulación de listas

Las listas se crean utilizando corchetes (`[]`) y separando sus elementos por comas. Algunas operaciones básicas de manipulación de listas incluyen agregar elementos al final (`append()`), insertar elementos en una posición específica (`insert()`), eliminar elementos por valor (`remove()`) y contar la cantidad de ocurrencias de un

elemento (`count()`). A continuación se muestra un procedimiento en el intérprete de python[18], mostrando los procedimientos básicos para la creación y manejo de listas.

[18] La opción `-q` al ejecutar python iniciará el intérprete sin el mensaje de bienvenida.

```
username@machine:~$ python3 -q
>>> lista_vacia = []
>>> lista_numeros = [1, 2, 3, 4, 5]
>>>
>>> lista_numeros.append(6)
>>> print(lista_numeros)
[1, 2, 3, 4, 5, 6]
>>>
>>> lista_numeros.insert(1, 1.5)
>>> print(lista_numeros)
[1, 1.5, 2, 3, 4, 5, 6]
>>>
>>> lista_numeros.remove(3)
>>> print(lista_numeros)
[1, 1.5, 2, 4, 5, 6]
>>>
>>> cantidad_de_dos = lista_numeros.count(2)
>>> print(cantidad_de_dos)
1
>>>
username@machine:~$
```

Las listas, al igual que todos los iteradores en python pueden ser indexados a sus elementos a partir del índice 0, así el primer elemento de la lista `l` [19] sera `l[0]`. Además si la lista tiene un largo `N`, entonces podremos acceder a su ultimo elemento con `l[N-1]`. El acceso a los elementos también se puede realizar por una técnica llamada *slicing*, que permite recortar y obtener *trozos* de una lista original y copiarlo a nuevas variables.

[19] No confunda la lista `l` con el número 1.

2.3.2. Slicing en listas

El slicing es una técnica que permite extraer una parte de una lista, creando una nueva lista a partir de un rango especificado de índices. La sintaxis general para el slicing es `lista[inicio:final:pasos]`, donde `inicio` es el índice del primer elemento del subconjunto, `final` es el índice del elemento siguiente al último elemento del subconjunto, y `pasos` es la cantidad de índices entre los elementos del subconjunto.

```
lista_numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

primeros_tres = lista_numeros[0:3] # Extrae los tres primeros elementos
pares = lista_numeros[0::2] # Extrae los elementos pares
impares = lista_numeros[1::2] # Extrae los elementos impares
reversa = lista_numeros[::-1] # Invierte el orden de la lista

print(f'Lista Original = {lista_numeros}')
print(f'Lista de primeros tres = {primeros_tres}')
print(f'Lista de pares = {pares}')
print(f'Lista de impares = {impares}')
print(f'Lista reversa = {reversa}')

```

Al ejecutar el código anterior veremos entonces como se han generado listas nuevas a partir del slicing de la lista original

```
username@machine:$ python3 example.py
Lista Original = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista de primeros tres = [0, 1, 2]
Lista de pares = [0, 2, 4, 6, 8]
Lista de impares = [1, 3, 5, 7, 9]
Lista reversa = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
username@machine:$
```

Hechos interesantes sobre las listas en Python

- Las listas pueden ser anidadas, lo que significa que una lista puede contener otras listas como elementos. Esto permite crear estructuras de datos multidimensionales, como matrices. Sin embargo utilizaremos otras librerías especializadas para el manejo de matrices.
- Las listas en Python se implementan como arreglos dinámicos, lo que significa que pueden cambiar de tamaño durante la ejecución del programa y que su acceso es de tiempo constante.
- La función `len()` devuelve la cantidad de elementos en una lista.
- Las listas pueden ser concatenadas utilizando el operador + y replicadas utilizando el operador `*`. Por ejemplo, `[1, 2, 3] + [4, 5, 6]` crea una nueva lista `[1, 2, 3, 4, 5, 6]`, y `[1, 2, 3] * 3` crea una nueva lista `[1, 2, 3, 1, 2, 3, 1, 2, 3]`.
- La función `list()` puede utilizarse para convertir otros tipos de secuencias, como tuplas o cadenas de caracteres, en listas. Por ejemplo, `list((1, 2, 3))` devuelve `[1, 2, 3]`, y `list("Python")` devuelve `['P', 'y', 't', 'h', 'o', 'n']`.
- Es posible utilizar la función `sorted()` para obtener una nueva lista con los elementos de la lista original ordenados, sin modificar la lista original. Para ordenar la lista original en su lugar, se puede utilizar el método `sort()`.

2.4. Control de flujo

Los programas computacionales son diseñados para almacenar y trabajar datos. Para esto los lenguajes incorporan estructuras que permiten **controlar el flujo** de ejecución de un código. Existen cuatro estructuras básicas que todos los lenguajes ofrecen de una forma u otra: ejecución secuencial, selección, repetición e invocación [20].

La estructura más simple es la ejecución secuencial, en la cuál las intrucciones van siendo realizadas por el computador de forma secuencial, es decir una a la vez, partiendo de la primera instrucción hasta llegar a la última. La verdadera magia comienza a ocurrir cuando las otras estructuras comienzan a ser parte del código. Sin embargo debemos comprender que son los ámbitos dentro de Python antes de saltar a los tipos de control de flujo.

[20] Joyanes Aguilar, Luis (2013). Fundamentos generales de programación (1^{era} edición). Ciudad de México, México: McGraw Hill. p. 368. ISBN 978-607-15-0818-8.

2.4.1. Ámbitos en Python

El ámbito (*scope*) en Python se refiere a la región en la que una variable es accesible y visible dentro del código. El concepto de ámbito es importante en la programación, ya que permite controlar la visibilidad y el acceso a las variables y funciones en diferentes partes del código. En Python, hay cuatro niveles de ámbito:

1. Ámbito local (Local)
2. Ámbito de la función que lo encierra (Enclosing)
3. Ámbito global (Global)
4. Ámbito incorporado (Built-in)

Estos ámbitos forman la regla LEGB, que es el orden en el que Python busca una variable en los diferentes niveles de ámbito. A continuación sólo mencionaremos el ámbito local (L) y a medida que se incorporen más elementos de programación se irán complementando estos conceptos.

El ámbito local es el nivel más interno de ámbito en Python. Se crea cuando se define una función, y las variables declaradas dentro de la función pertenecen al ámbito local. Estas variables son accesibles únicamente dentro de la función y no son visibles fuera de ella. Si bien no hemos cubierto nada sobre funciones, observemos el siguiente código :

```
1  def funcion():
2      x = 5
3
4  print(f'x = {x}')
```

La línea uno construye una función que llamaremos `funcion`. La seguna línea es clave ya que la indentación (espacio en blanco o sangría) es lo que define lo que pertenecen al ámbito de `funcion`. Todo instrucción que pertenezca a la función – i.e., *se encuentre dentro del ámbito* de la función – **debe** estar indentada. Si ejecutamos este programa, veremos el siguiente error :

```
1  user@machine:$ python3 ambito.py
2  Traceback (most recent call last):
3  File "/home/user/ambito.py", line 4, in <module>
4  print(f'x = {x}')
5  ^
6  NameError: name 'x' is not defined
7  user@machine:$
```

Podemos observar que el llamado a la variable `x` dentro de la función `print` genera un error, debido a que esa variable no existe fuera del ámbito de la función.

Al igual que esta función los controladores de flujo definirán ámbitos, que serán ejecutados (o no) sólo si las condiciones se cumplen.

2.4.2. Selección if/elif/else

Las estructuras de control condicionales en Python permiten que el flujo del programa siga diferentes caminos según las condiciones dadas. Estas estructuras incluyen `if`, `elif` y `else`. La estructura general es la siguiente:

```
1  if condición_1:
2      # Código que se ejecuta si condición_1 es verdadera
3  elif condición_2:
4      # Código que se ejecuta si condición_2 es verdadera
5  else:
6      # Código que se ejecuta si ninguna de las condiciones anteriores es verdadera
```

Acá podemos observar que el código **debe** ser indentado para que se encuentre dentro del ámbito de los condicionales `if`, `elif`, o `else`.

2.4.2.1. Ejemplo 1: Determinar el mayor de tres números

El siguiente ejemplo incluye la función `input`, que esta diseñada para recibir información del usuario una vez que el código es ejecutado. Puede notar además los niveles de indentación de cada una de las condiciones utilizadas. Este ejemplo determina cuál es el mayor de tres números ingresados por el usuario:

```
a = int(input("Ingrese el primer número: "))
b = int(input("Ingrese el segundo número: "))
c = int(input("Ingrese el tercer número: "))

if a >= b and a >= c:
    print("El número mayor es:", a)
elif b >= a and b >= c:
    print("El número mayor es:", b)
else:
    print("El número mayor es:", c)
```

2.4.2.2. Ejemplo 2: Categorizar la edad de una persona

Este ejemplo categoriza la edad de una persona en función de un rango de edades:

```
edad = int(input("Ingrese su edad: "))

if edad < 13:
    print("Eres un niño.")
elif edad >= 13 and edad < 18:
    print("Eres un adolescente.")
elif edad >= 18 and edad < 65:
    print("Eres un adulto.")
else:
    print("Eres un adulto mayor.")
```

Ejercicio:

Escriba un programa en Python que pregunte por el año de nacimiento, al usuario. Luego el programa debe mostrar en pantalla a que generación a la que pertenece. Puede usar la siguiente información

- Generación del Baby Boom: Personas nacidas entre 1946 y 1964, durante el período de aumento en la tasa de natalidad que siguió a la Segunda Guerra Mundial.
- Generación X: A veces llamada la generación del ”baby bust”porque la tasa de natalidad cayó después del baby boom. Esta generación incluye a las personas nacidas entre 1965 y 1980.
- Generación del Milenio o Generación Y: Personas nacidas entre 1981 y 1996. A menudo son los hijos de la generación del baby boom.
- Generación Z o Centennials: Nacidas entre 1997 y 2012. Muchos de ellos son hijos de la Generación X y algunos de los más jóvenes de la generación del milenio.
- Generación Alfa: Nacidos entre 2013 y 2025, aproximadamente. Son los hijos más jóvenes de la generación del milenio y los primeros hijos de la Generación Z.

Hechos interesantes sobre if/elif/else

- Las estructuras de control condicionales no necesitan paréntesis alrededor de las condiciones, lo cual es diferente a otros lenguajes de programación como C++ y Java.
- No hay una estructura de control `switch` en Python, pero se pueden utilizar múltiples bloques `elif` para lograr un comportamiento similar.
- Python no tiene operadores ternarios como C++ o Java, pero se pueden lograr resultados similares utilizando la expresión `valor_si_verdadero if condición else valor_si_falso`.

2.4.3. Repetición For/While

Las estructuras de control de bucles en Python permiten ejecutar bloques de código varias veces, según las condiciones dadas. Estas estructuras incluyen `for` y `while`. La estructura general es la siguiente:

```
for variable in iterable:
        # Código que se ejecuta para cada elemento del iterable

while condición:
        # Código que se ejecuta mientras la condición sea verdadera
```

Acá un iterable corresponde a un objeto capaz de devolver sus elementos uno a uno, lo que permite ser recorrido o *iterado* en un bucle, por ejemplo `for` y `while`. Entre los iterables más comunes estan las lista, tuplas, arreglos, y strings.

2.4.3.1. Ejemplo 1: Suma de los primeros N números naturales

Este ejemplo calcula la suma de los primeros N números naturales usando un bucle `for`:

```
N = int(input("Ingrese el valor de N: "))
suma = 0

for i in range(1, N + 1):
    suma += i

print("La suma de los primeros", N, "números naturales es:", suma)
```

2.4.3.2. Ejemplo 2: Encontrar el primer número primo mayor que N

Este ejemplo encuentra el primer número primo mayor que N usando un bucle `while`:

```
def es_primo(numero):

    if numero < 2:
        return False

    for i in range(2, numero):
        if numero % i == 0:
            return False

    return True


N = int(input("Ingrese el valor de N: "))
```

```
14  numero = N + 1
15
16  while not es_primo(numero):
17      numero += 1
18
19  print("El primer número primo mayor que", N, "es:", numero)
```

Ejercicio:

Escriba un programa que pregunte por un valor N y luego imprima sólo los números primos entre $1 \ldots N$.

Hechos interesantes sobre for/while

- El bucle `for` en Python es en realidad un bucle ”for each”que itera sobre los elementos de un objeto iterable (como listas, tuplas, conjuntos, diccionarios o rangos).
- La función `range()` en Python es muy útil para generar secuencias numéricas en bucles `for`, pero es importante recordar que el segundo argumento de `range()` es exclusivo.
- Se pueden utilizar las palabras clave `break` y `continue` para controlar el flujo de un bucle. `break` sale del bucle inmediatamente, mientras que `continue` omite el resto del bloque de código y pasa a la siguiente iteración.
- Se puede utilizar la palabra clave `else` junto con un bucle `for` o `while` para ejecutar un bloque de código cuando el bucle se completa sin interrupciones (sin encontrar un `break`).

2.5. Clases

Las clases son la base de la programación orientada a objetos en Python. Permiten definir estructuras de datos y comportamientos personalizados mediante la creación de objetos. Una clase actúa como una plantilla para crear objetos con atributos y métodos específicos.

2.5.1. Definición y uso de clases

Para definir una clase en Python, se utiliza la palabra clave `class`, seguida del nombre de la clase y dos puntos. Los atributos y métodos de la clase se definen dentro del bloque de la clase, utilizando la palabra clave `self` para referirse a la instancia actual de la clase.

```
1   class Persona:
2       def __init__(self, nombre, edad):
3           self.nombre = nombre
4           self.edad = edad
5
6       def presentarse(self):
7               print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
8
9   juan = Persona("Juan", 30)
10  juan.presentarse()
```

2.5.2. Herencia

La herencia es un concepto clave en la programación orientada a objetos que permite a una clase heredar atributos y métodos de otra clase. La clase de la cual se hereda se denomina clase base o superclase, mientras que la clase que hereda se denomina clase derivada o subclase. En Python, la herencia se indica colocando el nombre de la clase base entre paréntesis después del nombre de la clase derivada.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
            print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

#Construimos la clase
class Empleado(Persona):
    def __init__(self, nombre, edad, cargo, salario):
        super().__init__(nombre, edad)
        self.cargo = cargo
        self.salario = salario
    def presentar_cargo(self):
        print(f"Soy {self.nombre} y mi cargo es {self.cargo}.")

#Damos uso a la clase
ana = Empleado("Ana", 28, "Desarrolladora", 60000)
ana.presentarse()
ana.presentar_cargo()
```

2.5.3. Hechos interesantes sobre las clases en Python

- El método especial `__init__()` se denomina constructor y se utiliza para inicializar atributos de una instancia de la clase al momento de su creación.
- La función `super()` permite llamar a un método de la clase base desde la clase derivada. Por ejemplo, se puede utilizar en el constructor de la clase derivada para inicializar atributos heredados.
- Python admite herencia múltiple, lo que significa que una clase puede heredar de más de una clase base. Los nombres de las clases base deben estar separados por comas entre paréntesis después del nombre de la clase derivada.
- La encapsulación es un concepto en programación orientada a objetos que se refiere a ocultar detalles de implementación de una clase y permitir el acceso a sus atributos y métodos solo a través de métodos específicos. En Python, no hay una forma estricta de aplicar la encapsulación, pero se puede lograr utilizando guiones bajos para indicar que un atributo o método es ”privadoz no debe accederse directamente (porejemplo, `_atributo_privado` o `__metodo_privado()`).
- Los decoradores, como `@property` y `@staticmethod`, se pueden utilizar para modificar el comportamiento de atributos y métodos de una clase. Por ejemplo, el decorador `@property` permite definir un método como un atributo de solo lectura, y el decorador `@staticmethod` permite definir un método que no depende de una instancia específica de la clase y, por lo tanto, no requiere el argumento `self`.

- En Python, todo es un objeto, incluso las clases, los números y las funciones. Esto significa que las clases pueden ser manipuladas, pasadas como argumentos y utilizadas como atributos de otras clases, de la misma manera que cualquier otro objeto.

2.5.4. Ejemplo

Consideremos un ejemplo simple de clases. Este ejemplo define una clase Auto, la cual tiene atributos para marca, modelo y año. Además, tiene dos métodos: descripcion() para imprimir información acerca del auto y envejecer() para aumentar el año del auto.

```
class Auto:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def descripcion(self):
        print(f"Este auto es un {self.marca} {self.modelo} del año {self.año}.")

    def envejecer(self):
        self.año += 1

# Crear una instancia de la clase Auto
mi_auto = Auto('Toyota', 'Corolla', 2010)

# Llamar a los métodos de la instancia
mi_auto.descripcion()
mi_auto.envejecer()
mi_auto.descripcion()

```

2.6. Entrada y salida

La entrada y salida de datos (E/S, ó I/O) es un aspecto fundamental de la programación en Python. Permite a los programas interactuar con los usuarios, archivos y otros programas.

2.6.1. Entrada y salida estándar

La función `input()` se utiliza para leer datos del usuario, como hemos visto en algunos de los ejemplos anteriores. Por defecto, devuelve una cadena de caracteres, pero se puede utilizar la función de conversión apropiada para obtener un valor de otro tipo.

La función `print()` se utiliza para mostrar datos al usuario. Puede aceptar cualquier cantidad de argumentos y los convierte automáticamente en cadenas de caracteres.

```
nombre = input("¿Cómo te llamas? ")
print("Hola,", nombre, "!")

numero = int(input("Ingresa un número: "))
print("El cuadrado de tu número es", numero ** 2)
```

2.6.2. Lectura y escritura de archivos

Python tiene funciones incorporadas para leer y escribir archivos. La función `open()` se utiliza para abrir un archivo y devuelve un objeto de archivo que proporciona métodos para manipular el archivo.

El método `read()` lee todo el contenido del archivo, y el método `write()` escribe en el archivo. Es importante recordar cerrar el archivo después de su uso con el método `close()` para liberar los recursos del sistema.

```
#Escritura de archivo
with open('test.txt', 'w') as f:
    f.write("Hola, mundo!")

#Lectura de archivo
with open('test.txt', 'r') as f:
    contenido = f.read()
    print(contenido)
```

Desde un archivo, cada una de las lineas pueden ser separadas segun algun carácter separador. Consideremos por ejemplo el fichero online `http://go.gitarra.cl/w0N8He`, este fichero lo podemos descargar en nuestro directorio de trabajo y allí escribir un programa en python como el que se muestra a continuación.

```
# Abre el archivo en modo lectura
with open('datos.txt', 'r') as file:
    counter = 0
    # Lee cada línea del archivo
    for line in file:
        # Usa split(',') para dividir la línea en palabras
        words = line.split(',')

        # Imprime las palabras
        print(words)
        counter = counter + 1
        #Si alcanzamos las 5 líneas se acaba el ciclo y cerramos el fichero
        if counter == 5:
            file.close()
            break

```

Este código considera que el fichero `datos.txt` ya ha sido descargado y se encuentra en el mismo directorio que el código. Podemos observar que la línea 3 añade un contados y que por cada lectura de linea del archivo este contador se incrementa en 1, así al llegar a un valor de 5, el iterador termina y cierra el archivo produciento el fin del ciclo principal.

La salida posterior a la ejecución es :

```
username@machine:~$ python3 263.py
['Christina Kennedy', 'thomastimothy@gmail.com', '(034)785-7057x026\n']
['1308609478', 'Kelly Williams', 'steven84@gmail.com\n']
['gregg52@mccarthy.biz', 'Sandy Luna', '2557084635\n']
['Charles Martinez', '162-908-2769x009', 'matthew46@parrish.com\n']
['+1-584-693-9789x0106', 'theodore96@walker.biz', 'Erik Johnson\n']
username@machine:~$
```

En el resultado podemos observar como cada linea del archivo es separada en base a las comas que hay en el archivo. pueden entregarse distintos carácteres a la función split para separar de distintas formas, o bien entregarla sin argumentos para utilizar el espacio simple como separador.

Ejercicio:

En base al archivo del último ejemplo, ordene la información para que cada línea contenga primero el nombre, luego el correo electrónico y finalmente el número de teléfono, y luego escribir los datos ordenados en un nuevo archivo de texto llamado ”datos_ordenados.txt”.

Considere que : Los nombres siempre constan de dos palabras separadas por un espacio, las direcciones de correo electrónico siempre contienen un ”@z los números de teléfono siempre están en el formato ”xxx-xxx-xxxx”.

Hechos interesantes sobre la entrada y salida en Python

- Python tiene una sintaxis especial, `with`, que se utiliza con los archivos. Garantiza que el archivo se cierre correctamente cuando se termina de usar, incluso si ocurren errores durante su uso.
- Además del método `read()`, los objetos de archivo proporcionan otros métodos útiles para la lectura, como `readline()` para leer una línea a la vez y `readlines()` para leer todas las líneas en una lista.
- La función `print()` puede usarse para escribir en un archivo especificando el archivo como el argumento de la palabra clave `file`. Por ejemplo, `print("Hola, mundo!", file=f)` escribe ”Hola, mundo!.[en] el archivo representado por `f`.
- Python soporta una amplia gama de operaciones de formateo de cadenas, que son especialmente útiles para la salida. Por ejemplo, la función `format()` y las cadenas f (por ejemplo, `f"Hola, nombre!"`) proporcionan una forma conveniente y legible de incrustar expresiones dentro de las cadenas de caracteres.

2.7. Módulos

Los módulos en Python son archivos que contienen definiciones y declaraciones de funciones, variables y clases. Permiten organizar y reutilizar el código en diferentes programas. Para usar un módulo en un programa Python, primero debe ser importado utilizando la declaración `import`.

2.7.1. Módulo `math`

El módulo `math` proporciona funciones matemáticas y constantes.

```
import math

#Uso de la constante pi
circunferencia = 2 * math.pi * 5
print(f"La circunferencia de un círculo de radio 5 es: {circunferencia}")

#Uso de la función sqrt
raiz = math.sqrt(16)
print(f"La raíz cuadrada de 16 es: {raiz}")
```

2.7.2. Módulo `random`

El módulo `random` proporciona funciones para generar números aleatorios.

```python
import random

#Generar un número aleatorio entre 0 y 1
numero = random.random()
print(f"Un número aleatorio entre 0 y 1 es: {numero}")

#Elegir un elemento aleatorio de una lista
lista = [1, 2, 3, 4, 5]
eleccion = random.choice(lista)
print(f"Un elemento aleatorio de la lista {lista} es: {eleccion}")
```

2.7.3. Módulo `sys`

El módulo `sys` proporciona funciones y variables que se utilizan para manipular diferentes partes del entorno de ejecución de Python.

```python
import sys

#Imprimir la versión de Python
print(f"Versión de Python: {sys.version}")

#Imprimir la ruta de búsqueda de módulos
print(f"Ruta de búsqueda de módulos: {sys.path}")
```

Hechos interesantes sobre los módulos en Python

- Puede importar solo partes específicas de un módulo utilizando la sintaxis `from modulo import nombre`. Por ejemplo, `from math import pi` importaría solo la constante `pi` del módulo `math`.
- Puede cambiar el nombre de un módulo al importarlo utilizando la sintaxis `import modulo as nombre`. Esto puede ser útil si el nombre del módulo es largo o si existe un conflicto de nombres con otro módulo o variable.
- Los módulos solo se cargan una vez por sesión de Python. Si modifica un módulo después de importarlo, deberá reiniciar el intérprete de Python o usar la función `reload()` del módulo `importlib` para refrescar el módulo.
- Puede crear sus propios módulos simplemente creando nuevos archivos Python con las definiciones y declaraciones que desee. Luego puede importarlos usando el nombre del archivo (sin la extensión .py). Por ejemplo, si tiene un archivo llamado `mimodulo.py`, puede importarlo en otro programa Python con `import mimodulo`.
- Python viene con una gran biblioteca estándar que incluye módulos para una amplia variedad de tareas, incluyendo el procesamiento de archivos, protocolos de red, análisis de datos y mucho más. También hay miles de módulos de terceros disponibles que puede instalar y usar en sus programas.

2.8. Ejemplos

2.8.1. Fuerza Gravitacional

En este ejemplo calcularemos la fuerza gravitacional entre la Tierra y la Luna.

```python
import math

class CuerpoCeleste:
```

```
    def __init__(self, nombre, masa, radio):
        self.nombre = nombre
        self.masa = masa
        self.radio = radio

def calcular_fuerza_gravitacional(cuerpo1, cuerpo2, distancia):
    G = 6.67430e-11
    return G * cuerpo1.masa * cuerpo2.masa / distancia**2

tierra = CuerpoCeleste("Tierra", 5.972e24, 6.371e6)
luna = CuerpoCeleste("Luna", 7.342e22, 1.737e6)

distancia_tierra_luna = 3.844e8

fuerza_gravitacional = calcular_fuerza_gravitacional(tierra, luna,
↪  distancia_tierra_luna)

print(f"La fuerza gravitacional entre la {tierra.nombre} y la {luna.nombre} es de
↪  {fuerza_gravitacional} N.")

```

En este ejemplo, primero importamos el módulo `math`. Luego, definimos una clase `CuerpoCeleste` que representa un cuerpo celeste con atributos para el nombre, la masa y el radio.

A continuación, definimos una función `calcular_fuerza_gravitacional()` que calcula la fuerza gravitacional entre dos cuerpos celestes utilizando la ley de gravitación universal de Newton.

Después, creamos dos instancias de la clase `CuerpoCeleste` que representan la Tierra y la Luna, y especificamos la distancia entre ellos.

Finalmente, calculamos la fuerza gravitacional entre la Tierra y la Luna utilizando la función que definimos, y mostramos el resultado con la función `print()`.

Este ejemplo demuestra cómo los módulos, las clases, las variables y las funciones pueden trabajar juntos para resolver un problema real. En particular, muestra cómo una clase puede ser útil para agrupar datos relacionados y cómo una función puede ser útil para realizar un cálculo que se basa en esos datos.

2.8.2. Ley de Kepler

Este programa calcula el período de un planeta en una órbita alrededor del sol, dada la longitud del semi-eje mayor de la órbita.

```
import math

class Planeta:
    def __init__(self, nombre, semi_eje_mayor):
        self.nombre = nombre
        self.semi_eje_mayor = semi_eje_mayor

def calcular_periodo(planeta):
    # Constante de gravitación (m^3 kg^-1 s^-2)
    G = 6.67430e-11
    # Masa del Sol (kg)
    M = 1.989e30
    # Usamos la Tercera Ley de Kepler para calcular el período
    periodo = 2 * math.pi * math.sqrt(planeta.semi_eje_mayor**3 / (G * M))
    # Convertimos el período a días terrestres
    periodo_dias = periodo / (60 * 60 * 24)
    return periodo_dias

tierra = Planeta("Tierra", 1.496e11)

periodo = calcular_periodo(tierra)

```

```
23  print(f"El período de la {tierra.nombre} alrededor del sol es de aproximadamente
    ↪  {round(periodo)} días terrestres.")
24
```

n este ejemplo, primero importamos el módulo `math`. Luego, definimos una clase `Planeta` que representa un planeta con atributos para el nombre y la longitud del semi-eje mayor de su órbita.

A continuación, definimos una función `calcular_periodo()` que calcula el período de un planeta en una órbita alrededor del Sol utilizando la tercera ley de Kepler. Esta función también convierte el período de segundos a días terrestres.

Después, creamos una instancia de la clase `Planeta` que representa la Tierra.

Finalmente, calculamos el período de la Tierra utilizando la función que definimos, y mostramos el resultado con la función `print()`.

2.8.3. Energía Cinética

En este ejemplo, se calcula la energía cinética de un objeto en movimiento usando la fórmula clásica de la física $E_k = \frac{1}{2}mv^2$, donde m es la masa del objeto y v es su velocidad.

```
1   class ObjetoFisico:
2       def __init__(self, nombre, masa, velocidad):
3           self.nombre = nombre
4           self.masa = masa
5           self.velocidad = velocidad
6
7   def calcular_energia_cinetica(objeto):
8       return 0.5 * objeto.masa * objeto.velocidad**2
9
10  pelota = ObjetoFisico("Pelota de fútbol", 0.43, 30)
11
12  energia_cinetica = calcular_energia_cinetica(pelota)
13
14  print(f"La energía cinética de una {pelota.nombre} con masa de {pelota.masa} kg y
    ↪  velocidad de {pelota.velocidad} m/s es {energia_cinetica} J.")
15
```

Primero definimos una clase `ObjetoFisico` que representa un objeto con atributos para el nombre, la masa y la velocidad.

A continuación, definimos una función `calcular_energia_cinetica()` que calcula la energía cinética de un objeto utilizando la fórmula de la energía cinética en la física clásica.

Después, creamos una instancia de la clase `ObjetoFisico` que representa una pelota de fútbol.

Finalmente, calculamos la energía cinética de la pelota utilizando la función que definimos y mostramos el resultado con la función `print()`.