
Curso de programación Documentation

Versión v 0.4.6

Andrés Vázquez

17 de octubre de 2022

Contents

1	¿Que es programar?	3
1.1	Ejemplo	4
2	¿Que hace un programador?	5
3	¿Donde escribimos nuestro código?	7
4	Escribiendo nuestro primer código	9
4.1	¿Qué es una variable?	10
5	Números enteros <int>	11
5.1	Tareas	13
6	Cadenadas de caracteres o <i>strings</i> <str>	15
6.1	Tareas	16
6.2	Algunos ejemplos de uso	16
7	Recibir datos del usuario <code>input</code>	19
7.1	Tareas	20
8	Entorno de desarrollo: Visual Studio Code	21
8.1	Visual Studio Code	22
9	Funciones en Python	23
9.1	Anatomía de una función simple	23
9.2	Funciones con parámetros	24
9.3	Funciones con parámetros opcionales	25
9.4	Tareas	26
9.5	Algunos ejemplos de uso	26
10	Funciones incluidas en Python	29
10.1	Tareas	30
11	Booleanos <bool> + operadores de comparación + <code>if</code>	31

11.1	Control de flujo -> <code>if / elif / else</code>	32
12	Listas en Python <list>	35
12.1	Los <i>strings</i> tambien son listas	38
12.2	Función <code>split</code> de los <i>strings</i>	38
12.3	Tareas	39
12.4	Algunos ejemplos de uso	40
13	Iterando ando: <code>for</code>	43
13.1	Anatomía de un <code>for</code>	44
13.2	<code>continue</code> y <code>break</code> en ciclos <code>for</code>	45
13.3	<code>if + in</code>	46
13.4	Iterando <i>strings</i>	46
13.5	Agregado: <code>range</code>	46
13.6	Iterar <i>mientras</i> que algo suceda: <code>while</code>	47
14	Cuidando nuestro código: <code>git</code>	49
14.1	Control de versiones	49
14.2	Descentralizado	49
14.3	Modelo de ramificación	49
15	Git en la nube: GitHub	51
15.1	GitHub desde Visual Studio Code	52
16	Mi primer <i>PR</i>	55
16.1	Pull request paso a paso	55
17	Diccionarios: <code>dict</code>	63
17.1	Tareas	66
17.2	Algunos ejemplos de uso	74
18	Librerías incluida: <code>random</code>	77
18.1	Python al azar: <code>random</code>	77
18.2	<code>choice</code> : Elegir un opción al azar de una lista.	78
18.3	<code>shuffle</code> : Mezclar una lista	78
19	Librerías incluida: <code>datetime</code>	85
19.1	Fecha y hora: <code>datetime</code>	85
19.2	Fechas simples con <code>date</code>	85
19.3	Variación de tiempo con <code>timedelta</code>	86
19.4	Fecha + hora = momento exacto con <code>datetime</code>	86
19.5	fecha <—> string	87
20	Clases y objetos	89
20.1	Mi primera clase	89
21	Propiedades controladas	93
21.1	¿Que es <code>@property</code> ?	93
22	Funciones de mi clase	97

22.1	Tareas	100
22.2	Algunos ejemplos de uso	100
23	Funciones especiales de las clases	101
23.1	__add__	101
23.2	__str__	102
23.3	__eq__	103
24	Código final de nuestra clase	105
25	Otras funciones especiales	107
25.1	Tareas	107
25.2	Algunos ejemplos de uso	108
26	Paquetes y módulos	111
27	Indices y tablas	113

El presente curso esta orientado a comenzar a programar desde cero usando el lenguaje Python.

CHAPTER 1

¿Que es programar?

Programar es escribir las *instrucciones* necesarias para que una *computadora* realice alguna tarea. Las *instrucciones* son diferentes según el entorno donde se usan o la finalidad que se busca. Existen muchos lenguajes de programación que reflejan esa variedad. Excede a este manual definir estrictamente que es una *computadora*. Diremos que al nombrarla incluimos a:

- La clásica computadora que podemos tener en nuestra casa u oficina.
- Una portatil como las conocidas notebooks
- Un teléfono celular

Podemos extender lo que entendemos por computadoras agregando:

- Un lavarropas moderno en tanto que tienen programas variados en un mini-computadora interna.
- La alarma de una casa o auto que según diferentes acciones pre-establecidas dispara acciones programadas (luces y bocinas).

En general muchos electrodomésticos ya incluyen computadoras y por lo tanto programas. Estas *instrucciones* organizadas con alguna finalidad específica conforman lo que denominamos *software*.

1.1 Ejemplo

La computadora interna de un lavarropas que gestiona los dispositivos y conexiones internas podría tener estas instrucciones:

1. Abrir el conducto de agua hasta que el sensor detecte que se llegó al nivel esperado
2. – Si no se cumple en 20 segundos mostrar en pantalla código de error ERR01
3. Abrir el conducto de jabón líquido 10 segundos
4. Girar a velocidad normal el tambor 20 vueltas hacia la derecha
5. Girar a velocidad normal el tambor 20 vueltas hacia la izquierda
6. Abrir el desagote del tambor mientras gira durante 1 minuto
7. – Si el sensor detecta que todavía hay agua mostrar en pantalla código de error ERR02
8. Abrir el conducto de agua durante 20 segundos (para enjuague)
9. Girar a velocidad normal el tambor 20 segundos
10. Abrir el desagote del tambor
11. – Si el sensor detecta que todavía hay agua mostrar en pantalla código de error ERR03
12. Girar a velocidad rápida el tambor 2 minutos hacia la derecha
13. Girar a velocidad rápida el tambor 2 minutos hacia la izquierda
14. Destabar la puerta del tambor, trabajo terminado.

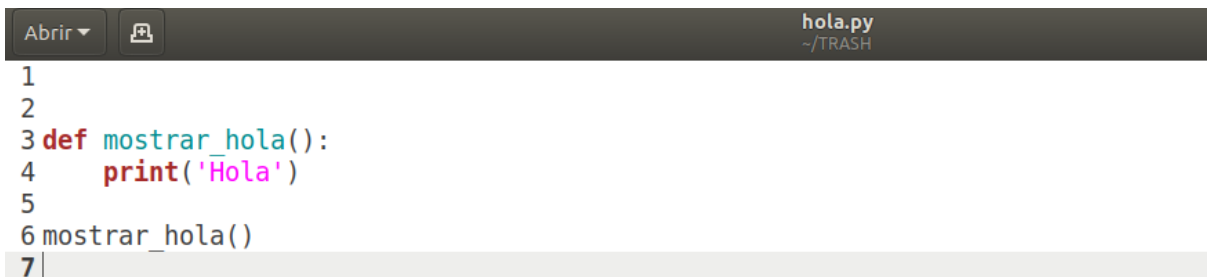
CHAPTER 2


¿Que hace un programador?

El trabajo de un programador de software es en general **escribir instrucciones en archivos de texto simples** en un lenguaje de programación específico. Un programador conoce uno o más *lenguajes de programación* y puede desempeñarse con ellos en múltiples entornos de trabajo. Estos entornos incluyen:

- Una página web
- Una aplicación para tu celular
- Un sistema que funciona en una computadora estándar (como Word, Google Chrome, Excel o cualquier programa que ves en el *escritorio* de tu computadora)
- Micro-sistemas que sirvan de soporte a sistemas más grandes y que *no se ven*

Acotar estos entornos donde un software se ejecuta es difícil, son múltiples.



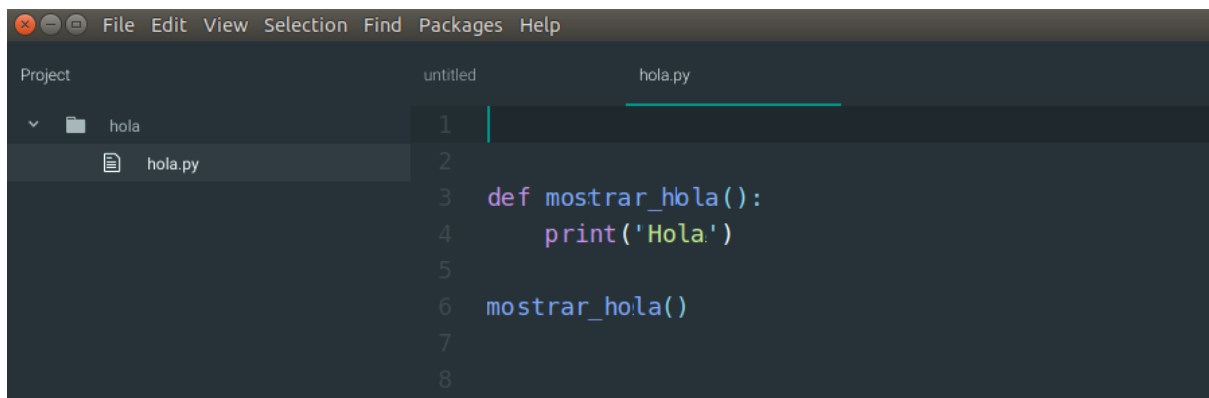
```
Abrir ▾  hola.py  
~ / TRASH  
1  
2  
3 def mostrar_hola():  
4     print('Hola')  
5  
6 mostrar_hola()  
7 |
```


CHAPTER 3

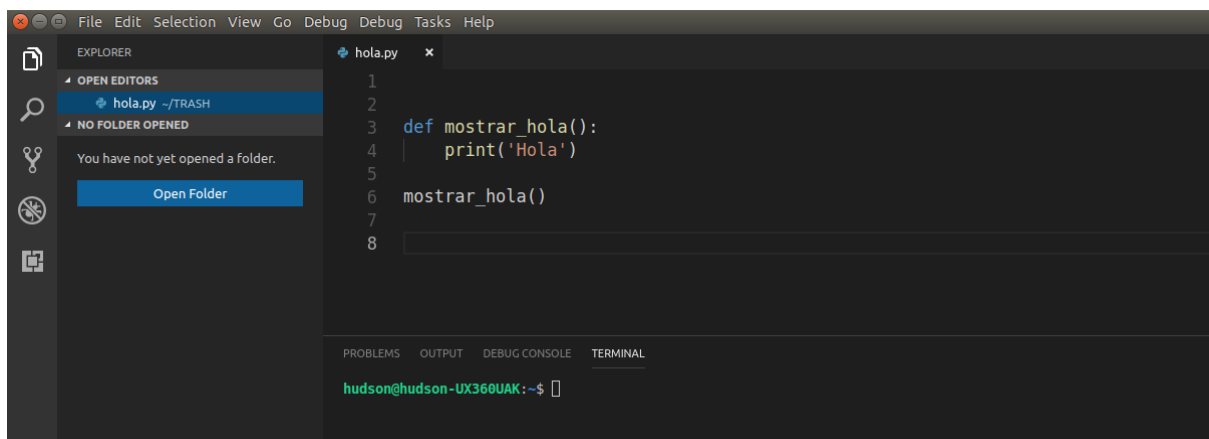
¿Donde escribimos nuestro código?

Como dijimos programar es escribir instrucciones (las denominaremos código o *código fuente*). Para esto podríamos usar algún software de edición de texto pero existen herramientas específicas para esta tarea.

Atom

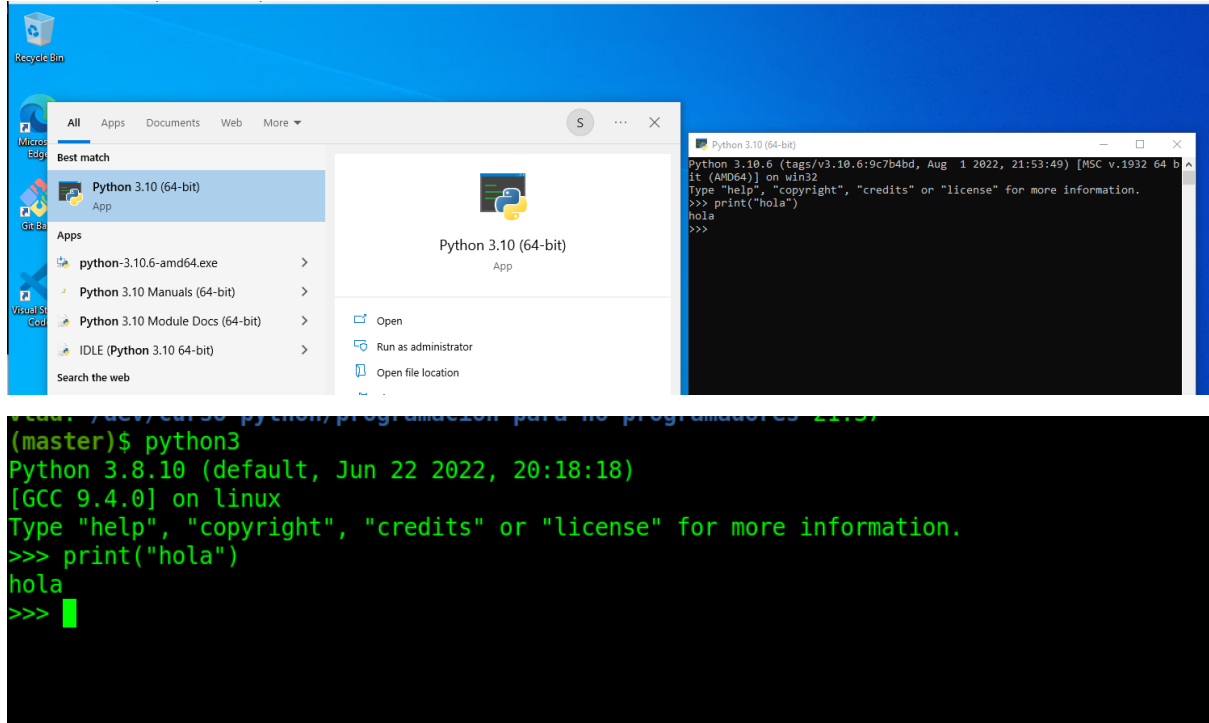


Visual Studio (de Microsoft)



A estos entornos de trabajo se los conoce *Entornos de desarrollo Integrado* o IDE por sus siglas en inglés (Integrated Development Environment). Estas herramientas proveen funcionalidades que simplifican el trabajo de un programador.

Es también posible ejecutar código Python línea a línea con la consola interactiva de Python.



La consola interactiva nos permite escribir código Python al mismo tiempo que se ejecuta. Es ideal para probar pequeñas porciones de código. Una vez que la cerramos, todo lo que hemos escrito se pierde.

CHAPTER 4

Escribiendo nuestro primer código

Cuando programamos parte de nuestro trabajo es tomar datos de entrada, procesarlos y finalmente transformarlos en datos de salida que necesitamos. Esta es una parte muy importante de nuestro trabajo como programadores. Es por esto que todos los lenguajes de programación incluyen formas de procesar datos de diferentes tipos.

tipos de datos

Como permanentemente vamos a recibir, procesar y devolver datos es muy importante conocer cuales son y de que herramientas dispone cada uno para resolver problemas comunes.

En Python (y en casi todos los lenguajes de programación) se utiliza el operador = como forma de asignar un valor (el de la izquierda) a una *variable* (a la derecha). De tal forma el código.

```
a = 1
```

quiere decir: *asignar el valor 1 a la variable llamada a*. Si mas adelante en el código usamos *a* nos estaremos refiriendo a su contenido: *1*. Entonces ...

```
b = a + 1
```

quiere decir *asignar el valor a+1 a la variable llamada b*. En este caso b será igual a 2 (1+1).

4.1 ¿Qué es una variable?

Las *variables* son los instrumentos que usan los lenguajes de programación para almacenar datos de diferentes tipos. Deben tener un identificador o nombre (en los ejemplos anteriores a y b).

Estos identificadores deben ser letras, números y el símbolo `_` (guión bajo) con estos límites:

- no puede tener espacios.
- no empezar con un número (si puede usarse después del primer carácter).
- no puede ser una palabra que Python ya tiene reservada para otras funciones como `if`, `for`, etc.

De tal forma, los siguientes identificadores de variables son válidos:

- `nombre`
- `n3`
- `nombre_y_apellido`
- `_dato_privado`
- `f910293`

y los siguientes no son válidos:

- `3n` (no se puede empezar con números)
- `while` (es una palabra reservada de Python)
- `nombre apellido` (no se pueden usar espacios)

4.1.1 ¿Cómo puede saber que tipo de datos almacena una variable?

Python incluye una herramienta llamada `type` que informa que tipo de datos contiene una variable dada.

```
a = 5
type(a)
# devuelve
# <class 'int'>
```

En este caso, la variable `a` es del tipo `int` (que veremos a continuación). *Nota: La palabra ``class`` cobrará sentido más adelante.*

Los siguientes son ejemplo de uso de los tipos básicos de datos de los que disponemos en Python.

CHAPTER 5

Números enteros <int>

Desde la consola interactiva de Python podemos usar numeros enteros. Estos son llamados `int` en Python (vienen de *Integer* en inglés). Con los enteros podemos hacer operaciones de cálculo básicas en Python.

```
edad = 31
calculo = 291 + 56 - 12
# tambien es posible mostrar (imprimir) el resultado simplemente
# escribiendo
# el nombre de la variable que deseamos conocer
calculo
355
# fuera de la consola interactiva, es necesario usar la funcion "print"
# para
# mostrar el valor contenido en una variable
print(calculo)
355

una_multiplicacion = 3 * 4
potencia = 2 ** 3 # ** hace calculos de potencias, en este caso: 2 al
# cubo
division = 8 / 4

resto = 5 % 2 # (calcula el resto de la division, en este caso 5/2 es
# 2 con resto 1)
```

Nota: Las palabras a la izquierda del signo igual son los nombres que elegimos para nuestras variables. Son arbitrarios y no representan nada más que un nombre interno, no tienen un significado especial para Python.

Es tambien posible usar las variables para hacer cálculos

```
unidades = 3
precio = 11
precio_final = unidades * precio

# Mostrar resultado
precio_final
33
```

variables y objetos

En Python, todo es un *objeto*. El concepto de *objeto* lo vamos a ver en profundidad más adelante. Por lo pronto diremos que en Python una variable es un *objeto* de un tipo específico. Este objeto tiene propiedades (también llamados *atributos*) y funciones.

Por ejemplo, en el código anterior la variable *unidades* (definida en la linea que dice `unidades = 3`) es en realidad un *objeto* de tipo `int`.

Los objetos de tipo `int` no tienen muchas propiedades y funciones. A modo de ejemplo, la función `bit_length` permite saber el número de dígitos que este número tiene en su versión binaria.

```
unidades = 3
unidades.bit_length()
2 # sería = 11 en binario (2 digitos binarios)
# Es tambien posible asignar el resultado de esa funcion a una variable
bits_en_unidades = unidades.bit_length()
# ver el resultado
bits_en_unidades
2
```

Nota importante: La forma de llamar a las propiedades y funciones de un objeto es usando el `.` (punto). Se hace de la forma `objeto.propiedad` o `objeto.funcion()`. Nótese que para llamar a las funciones son necesarios los paréntesis.

Si tienes curiosidad por conocer todas las propiedades y funciones de un *objeto* (del tipo que sea) puedes usar la función `__dir__()`

```
unidades = 3
unidades.__dir__() # tambien puede obtenerse esta lista con
→ dir(unidades)

['__repr__', '__hash__', '__getattribute__', '__lt__', '__le__', '__eq__',
→ '__ne__', '__gt__', '__ge__', '__add__', '__radd__', '__sub__', '__
→ rsub__',
'__mul__', '__rmul__', '__mod__', '__rmod__', '__divmod__', '__
```

(continúe en la próxima página)

(proviene de la página anterior)

```

→ rdivmod__,
  '__pow__', '__rpow__', '__neg__', '__pos__', '__abs__', '__bool__', '_
→ invert__',
  '__lshift__', '__rlshift__', '__rshift__', '__rrshift__', '__and__',
→ '__rand__',
  '__xor__', '__rxor__', '__or__', '__ror__', '__int__', '__float__', '_
→ floordiv__',
  '__rfloordiv__', '__truediv__', '__rtruediv__', '__index__', '__new__
→ ',
  'conjugate', 'bit_length', 'to_bytes', 'from_bytes', 'as_integer_ratio
→ ',
  '__trunc__', '__floor__', '__ceil__', '__round__', '__getnewargs__',
→ '__format__',
  '__sizeof__', 'real', 'imag', 'numerator', 'denominator', '__doc__',
→ '__str__',
  '__setattr__', '__delattr__', '__init__', '__reduce_ex__', '__reduce__
→ ',
  '__subclasshook__', '__init_subclass__', '__dir__', '__class__']

```

No te preocupes por esa larga lista y por todos esos guiones bajos, gradualmete iremos comprendiendo de que se tratan.

5.1 Tareas

Calcular cuantos segundos tiene un día definiendo las variables:

- segundos_en_minuto
- minutos_en_hora
- horas_en_dia

Finalmente asignar el resultado a una variable llamada segundos_en_dia

CHAPTER 6

Cadenadas de caracteres o *strings* <str>

Las cadenas de caracteres o *strings* son el tipo de dato para almacenar textos. Estos son llamados `str` en Python.

```
nombre = "Juana Velez"
# tambien es posible mostrar (imprimir) el contenido
print(nombre)
Juana Velez
type(nombre)
# devuelve <class 'str'>
```

Nota: Como los textos suelen naturalmente tener espacios es necesario delimitar donde empiezan y terminan con las " o ' (comillas dobles o simples).

Si intentamos definir una variable de tipo `str` sin comillas vamos a recibir un error de sintaxis.

```
nombre = Juana Velez
File "<stdin>", line 1
    nombre = Juana Velez
                ^
SyntaxError: invalid syntax
```

Con los *strings* podemos hacer también algunas operaciones en Python. La suma en *strings* (se llama *concatenar*) es posible:

```
nombre = "Juana"
apellido = "Velez"

nombre_completo = nombre + " " + apellido
```

Nota: esta suma incluye tres strings, dos tienen nombre y otro es un espacio definido directamente.

La multiplicación también está definida para *strings*:

```
letra = "a"
letra * 4
aaaa
```

Otras funciones disponibles para los *strings*:

```
nombre = "Juana Velez"
# funcion lower -> pasar a minúsculas
nombre.lower()
'juana velez'
# funcion upper -> pasar a mayúsculas
nombre.upper()
'JUANA VELEZ'
# funcion format -> completar las llaves dentro de un string con
# valores definidos fuera
saludo = "Hola, {}".format(nombre)
# otra forma de hacer lo mismo (se le llama "f strings")
saludo = f"Hola, {nombre}"
```

Los objetos de tipo `str` tienen muchas propiedades o funciones

6.1 Tareas

Investigar, usar y describir para qué sirven las siguientes funciones para objetos `str` en Python.

- `replace`:
- `capitalize`:
- `title`:
- `strip`:

Se espera un archivo de Python con estas funciones en uso como ejemplo.

6.2 Algunos ejemplos de uso

```
"""
Opciones para concatenar strings con variables
"""

nombre = 'Pedro'
```

(continúe en la próxima página)

(proviene de la página anterior)

```
pais = 'Chile'

print("Hello world {} de {}".format(nombre, pais))
# Hello world Pedro de Chile!

# valores enumerados
print("Valores enumerados. Hello world {0} de {1} ({0}-{1})!".
      ↪format(nombre, pais))
# Valores enumerados. Hello world Pedro de Chile (Pedro-Chile)!

# valores con nombre
print("Valores con nombre. Hello world {name} de {country}!".
      ↪format(name=nombre, country=pais))
# Valores con nombre. Hello world Pedro de Chile!

# Estilo C
print("Estilo C. Hello world %s de %s !" % (nombre, pais))
# Estilo C. Hello world Pedro de Chile !

# Nueva opcion desde python 3.6
print(f"Hello world {nombre} de {pais}!")
# Hello world Pedro de Chile!
```


CHAPTER 7

Recibir datos del usuario input

Es posible detener la ejecución de tu programa para solicitar al usuario de nuestro programa que ingrese datos.

```
nombre = input('Ingresa tu nombre: ')
apellido = input('Ingresa tu apellido: ')
print(f'Hola {nombre} {apellido}!')
```

La función `input` devuelve como *string* lo que el usuario ingresa. Si necesitaras un objeto de tipo `int` (por ejemplo para hacer cálculos) puedes hacer la transformación con `int(variable_string)`.

```
print('POTENCIAS')
nro = input('Ingresa un numero: ')
base = int(nro)
print(f'{base}2 = {base**2}')
print(f'{base}3 = {base**3}')
print(f'{base}4 = {base**4}')
print(f'{base}5 = {base**5}')
```

```
""" Ejemplo
POTENCIAS
Ingresa un numero: 7
72 = 49
73 = 343
74 = 2401
75 = 16807
"""
```

7.1 Tareas

- ¿Qué pasa si en el último ejemplo el usuario inserta una letra en lugar de un número?
¿Por qué?.
- Escribir un programa que le pida al usuario que ingrese los datos necesarios y calcule el [índice de masa corporal](#).

CHAPTER 8

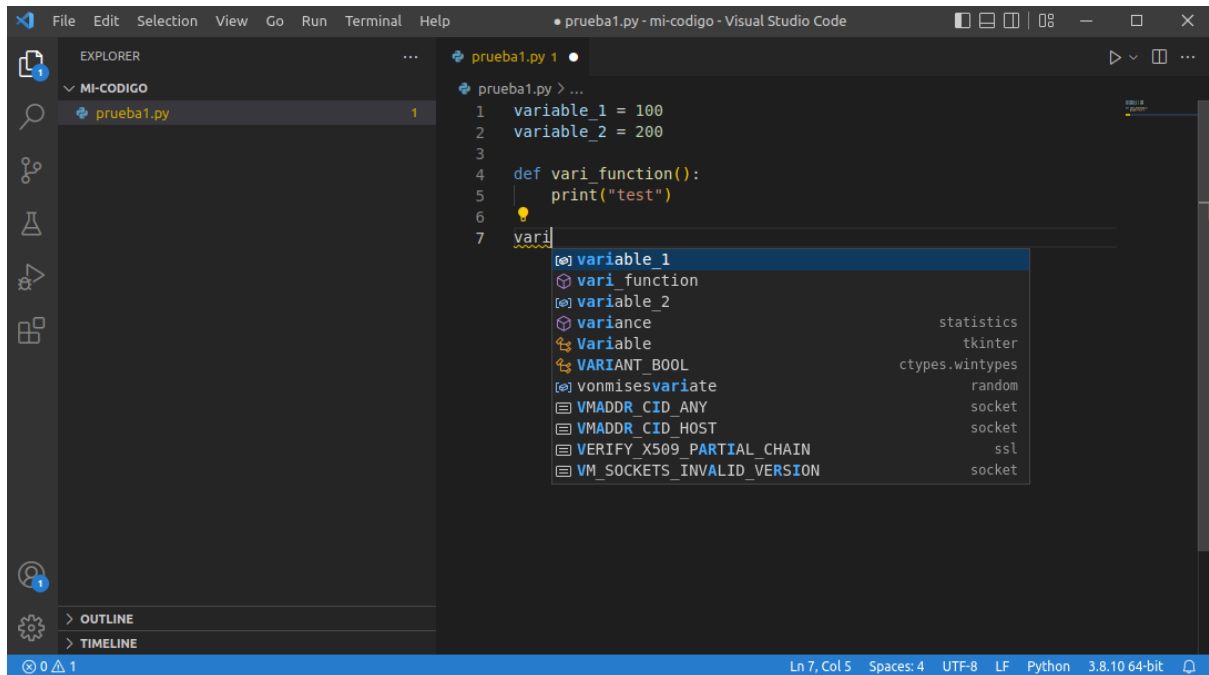
Entorno de desarrollo: Visual Studio Code

La consola interactivo de Python es muy util para hacer pruebas pero si queremos pasar al siguiente nivel y escribir código fuente más complejo es necesario contar con un entorno de desarrollo con la posibilidad de guardar nuestro trabajo y acceder a herramientas que lo simplifiquen.

Como desarrolladores de software vamos a pasar mucho tiempo escribiendo texto y si bien un editor de texto simple sería suficiente vamos a usar una herramienta hecha específicamente para esto.

Hay muchos productos disponibles y todos tienen funcionalidades similares.

8.1 Visual Studio Code



Algunas funcionalidades interesantes:

- Estilos y coloreo del código.
- Autocompletado: Ayuda para completar el código que estas escribiendo.
- Posibilidad de ejecutar y depurar de código.
- Extensión de la funcionalidad mediante plugins desarrollados por terceros.
- Posibilidad de integrarte con sistemas de controles de versiones de código (que veremos más adelante).

CHAPTER 9

Funciones en Python

En este curso hemos nombrado y usado sin definir todavía a las *funciones*. Una función en Python es una porción de código que cumple una tarea determinada y opcionalmente devuelve un resultado.

Veamos un ejemplo:

```
def segundos_en_un_dia():
    segundos_en_una_hora = 60
    minutos_en_una_hora = 60
    horas_en_un_dia = 24
    segundos_en_un_dia = segundos_en_una_hora * minutos_en_una_hora *
    horas_en_un_dia
    return segundos_en_un_dia

segundos = segundos_en_un_dia()
print(segundos)
86400
```

9.1 Anatomía de una función simple

- `def` es una palabra reservada de Python (no la podemos usar como nombre de variable) que usamos para indicar que estamos definiendo una función.
- Después de `def` agregamos el nombre de nuestra función. Se deben cumplir las mismas reglas que para los nombres de variables (no pueden empezar con números, no pueden tener espacios, etc).

- Después del nombre de la función colocamos parentesis (es obligatorio). En el futuro vamos a usarlas para agregar los que se llaman *parámetros*. Por ahora solo es importante no olvidar agregarlos.
- Finalmente agregamos : (dos puntos) para indicar que terminamos de definir el encabezado de la función y vamos a comenzar con el código.
- El código de la función debe estar tabulado hacia la derecha. **Esta es una de las grandes diferencias que Python tiene con los demás lenguajes de programación.** El código propio de la función comienza tabulado y termina cuando el código vuelve a la izquierda. Muchos otros lenguajes de programación usan las llaves {} para delimitar donde empiezan y terminan los bloques de código.
- `return` se usa para indicar cual es el valor que devolverá nuestra función cuando sea llamada. En este caso es el resultado de un cálculo. No es obligatorio usarla, a veces simplemente necesitamos procesar datos sin entregar resultados.
- Una vez definida (y terminada anulando la tabulación y volviendo a la izquierda el código), una función se puede llamar simplemente con su nombre y los paréntesis.

9.2 Funciones con parámetros

Muchas funciones procesan datos que ya conocemos pero en muchos casos necesitamos que nuestra función procese datos que pueden variar. Para estos casos necesitamos darle a nuestra función la posibilidad de agregar valores variables llamados *parámetros*. Estos parámetros se incluyen dentro de los paréntesis del encabezado de nuestra función **separados por comas**.

Ejemplo de una función con parámetros.

```
def sumar(a, b):  
    print(f'Se llamo a la funcion sumar con {a} y {b}')    return a + b  
  
resultado = sumar(10, 20)  
print(f'Resultado de la suma: {resultado}')Resultado de la suma: 30  
  
resultado = sumar(7, 27)  
print(f'Resultado de la suma: {resultado}')Resultado de la suma: 34  
  
resultado = sumar("10", "20")  
print(f'Resultado de la suma: {resultado}')Resultado de la suma: 1020  
# Para pensar: ¿Que sucedió?)  
  
# ¿Qué pasa si no completamos todos los parámetros esperados?  
sumar(10)
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sumar() missing 1 required positional argument: 'b'

# Obtenemos un error. Los parámetros así como están definidos son
↳ obligatorios.
```

Nótese que dentro del código de la función los parámetros son variables disponibles para usar libremente. Fuera de ella (por ejemplo tratar de usar `a` fuera de la función `suma`) no están definidas y darán error. El alcance y validez de `a` es solo dentro de la función que la declaró como parámetro.

9.3 Funciones con parámetros opcionales

En algunos casos no necesitamos que todos los parámetros sean obligatorios. Muchas veces hay valores que son de uso más frecuente y el usuario no querrá definirlos cada vez que llama a la función. Para esto existen parámetros *opcionales*. Simplemente colocamos cual es el valor predeterminado para los parámetros en el encabezado de la función y es suficiente. Tomemos el ejemplo de la función *potencia* y asumamos que en general los usuarios querrán elevar números al cuadrado.

```
def potencia(a, b=2):
    print(f'Se llamo a la funcion potencia con {a} y {b}')
    return a ** b

resultado = potencia(3)
print(f'Resultado de la potencia 3 ** 2: {resultado}')

resultado = potencia(3, 3)
print(f'Resultado de la potencia 3 ** 3: {resultado}')

# Es tambien posible cambiar el orden de los parametros al
# llamar a la función usando su nombre.
# Las siguientes opciones devolverán el mismo resultado.
resultado = potencia(3, 4)
resultado = potencia(a=3, b=4)
resultado = potencia(b=4, a=3)
```

9.4 Tareas

- Un mago se para frente a su audiencia y les dice:
 - **Piensen un numero cualquiera**
 - Ahora súmenle 2
 - Multipliquen el resultado por 2
 - Resten al resultado el numero que pensaron al inicio
 - Sumen al resultado 8
 - Resten otra vez el numero pensado al principio.

Finalmente les pregunta a todos el resultado final y **aplauden**

Tarea: Crear una funcion que haga paso a paso los calculos del mago para llegar al resultado final. Ejecutar la función para 5 numeros distintos y observar los resultados para entender que paso (editado)

- Crear una funcion que calcule y devuelva la superficie de un rectangulo dados (como parámetros) los valores de sus lados. Usar esta funcion con valores ingresados por el usuario con `input`.
- Crear una funcion que dados un nombre y un apellido imprima en pantalla «*Hola NOMBRE APELLIDO!*». La función no debe devolver ningun valor.
- Crear una función que dado un texto pasado como parámetro, devuelva el mismo texto pero con todas las vocales cambiadas por un asterisco.
- Crear una funcion que dada una temperatura en grados Celsius devuelva el equivalente en grados Fahrenheit. Usar esta funcion con valores ingresados por el usuario con `input`.

En todos los casos usar la función para asegurarse que funciona como es esperado.

9.5 Algunos ejemplos de uso

```
def sumar(a, b):  
    print(f'Se llamo a la funcion sumar con {a} y {b}')  
    return a + b  
  
resultado = sumar(10, 20)  
print(f'Resultado de la suma: {resultado}')
```

```
def potencia(a, b=2):  
    print(f'Se llamo a la funcion potencia con {a} y {b}')  
    return a ** b
```

(continué en la próxima página)

(proviene de la página anterior)

```
resultado = potencia(3)
print(f'Resultado de la potencia 3 ** 2: {resultado}')

resultado = potencia(3, 3)
print(f'Resultado de la potencia 3 ** 3: {resultado}')
```

```
def raiz(numero, raiz=2):
    print(f'Se llamo a la funcion raiz con {numero} y {raiz}')
    return numero ** (1/raiz)

resultado = raiz(numero=64)
print(f'Resultado de la raiz cuadrada de 64: {resultado}')

resultado = raiz(raiz=3, numero=27)
print(f'Resultado de la raiz cúbica de 27: {resultado}')
```

```
def mi_function(*args, **kwargs):
    print(f'Argumentos sin nombre: {args}')
    print(f'Argumentos con nombre: {kwargs}')

mi_function(3, "algo", nombre='Luis', edad=91)

# Argumentos sin nombre: (3, 'algo')
# Argumentos con nombre: {'nombre': 'Luis', 'edad': 91}
```


CHAPTER 10

Funciones incluidas en Python

Así como nosotros definimos nuestras propias funciones, Python incluye algunas funciones por defecto. Ya hemos usado una de ella en nuestros códigos de ejemplo: `print`.

A estas funciones incorporadas y disponibles en Python se las conoce como *built-ins*.

La función `print` simplemente *imprime* en nuestra terminal cualquier valor que se le pase como parámetro.

Veamos algunos ejemplos:

```
print('Hola')
nombre = 'Juan'
print(nombre)
```

Pero `print` no es la única función disponible, hay muchas.

Nota: La lista de todas las funciones built-ins de Python está disponible [aquí](#).

Algunos ejemplos:

```
# abs -> obtener el valor absoluto de un numero
abs(-3)
3
# len -> obtener el largo de un objeto. No disponible para cualquier
# objeto. En el caso de los strings, cuenta las letras
len('hola')
4
nombre = 'Victor'
len(nombre)
6
# type -> devuelve el tipo de un objeto
```

(continué en la próxima página)

(proviene de la página anterior)

```
type('hola')
# devuelve <class 'str'>
type(nombre)
# devuelve <class 'str'>
# max y min -> devuelven el elemento maximo y minimo de una lista de
# elementos
max(3, 5, 15, 1)
15
min(3, 5, 15, 1)
1
# en caso de strings, max y min resuelven ordenando alfabéticamente.
max("hola", "chau")
'hola'
```

10.1 Tareas

- Escribir una función que dadas tres palabras devuelva el largo total de todas ellas juntas. Por ejemplo (si la función se llamara `largo_total`) la llamada `largo_total('hola', 'chau', 'tercera')` debe devolver 15.
- Escribir una función que dados dos números devuelva el valor absoluto del menor de ambos.

En todos los casos usar la función para asegurarse que funciona como es esperado.

CHAPTER 11

Booleanos <bool> + operadores de comparación + if

El tipo de datos <bool> (nombrados así en memoria de matemático inglés George Boole) solo usa dos valores: True y False (*verdadero* y *falso*). Es el tipo de datos más simple y uno de los más usados. Se puede asignar directamente o puede ser el resultado de una comparación. Los operadores de comparación en Python son:

- ==: Compara si dos variables valen lo mismo.
- !=: Compara si dos variables son distintas. En general el signo ! se usa para negar. Podemos pensar a este operador como *no igual*.
- > y <: Compara si una variable es mayor o menor que otra.

Para los números estas comparaciones son fáciles de intuir pero pueden significar diferentes cosas para diferentes tipos de datos. Como vimos antes en las funciones `max` y `min` usadas para *strings*, las comparaciones son alfabéticas. Es por esto que en Python podemos decir que “alpha” < “beta” es True.

Veamos algunos ejemplos con código.

```
# "a" es igual a uno.
a = 1
b = 2

# ¿Es "a" igual a 1?
a == 1
# Si, es verdadero
True

# ¿Es "a" mayor que "b"?
a > b
# No, es falso
```

(continué en la próxima página)

(proviene de la página anterior)

False

```
# ¿Es "a" distinto de "b"?
```

```
a != b
```

```
# Si, es verdadero
```

True

11.1 Control de flujo -> if / elif / else

El flujo de un programa no tiene que ser siempre lineal. Diferentes porciones de código pueden ejecutarse si alguna condición esperada se cumple. Para esto existe la sentencia `if` (traducido, es el *si* condicional del español).

Veamos su funcionamiento con simples ejemplos:

11.1.1 if

Uso de `if` solo:

```
a == 1
if a > 0: # nótese que la línea debe terminar en ":" como las
    ↪funciones
    print('"a" es mayor que cero')
```

Nota importante: la porción de código a ejecutarse **si** se cumple la condición debe estar tabulada (tal como lo hacemos en las funciones).

11.1.2 else

Opcionalmente, podemos usar `else` para capturar los casos que no cumplen la condición.

```
edad = 33
if edad >= 18:
    print('Es mayor de edad')
    prohibir_entrada = False
else:
    print('No es mayor de edad')
    prohibir_entrada = True
```

Nota importante: Las tabulaciones definen el inicio y el fin de las porciones de código a ejecutar según la condición. Puede ser más de una línea.

11.1.3 elif

Entre if y else podemos usar uno o varios elif para identificar mas casos buscados. Nota: elif es una forma abreviada para else if.

Veamos un ejemplo un poco mas complejo de una función que usa if/elif/else.

```
def identificar_bicho(patas):
    if patas == 6:
        print('Es un insecto')
    elif patas == 8:
        print('Es un arácnico')
    else:
        print('Bicho no identificado')

identificar_bicho(6)
'Es un insecto'
identificar_bicho(8)
'Es un arácnico'
identificar_bicho(5)
'Bicho no identificado'
```

11.1.4 and y or

Si queremos consultar mas de una condición simultaneamente podemos usar and u or (y u o).

Por ejemplo:

```
a = 1
b = 2

if a == 2 or b == 2:
    print('"a" o "b" valen dos (pueden ser ambos o cualquiera de los_
↪dos')

if a == 2 and b == 2:
    print('"a" Y "b" valen dos')
```

11.1.5 Tareas

- Escribir una función que se llame es_par y que dado un número devuelva True o False según corresponda. Tip: Los numeros pares tienen resto (operador %) cero al dividirlos por 2.
- Escribir una función que reciba 3 parametros: nombre, edad y termino_secundario. Si la edad es mayor que 18, termino_secundario es True y el nombre termina (tip:

función `endswith` de los *strings*) con «s» devuelve `True`. En cualquier otro caso, devuelve `False`.

- Escribir un programa que le pida al usuario que ingrese los datos necesarios para calcular el *índice de masa corporal* y finalmente informe (imprima algún mensaje) si el usuario tiene *Peso bajo*, *Normal*, *Sobrepeso* u *Obesidad*.

CHAPTER 12

Listas en Python <list>

Las listas en Python se usan para representar conjuntos ordenados de elementos. Los objetos contenidos pueden ser de cualquier tipo (incluidas otras listas).

Nota: *Ordenado* quiere decir que la lista preserva el orden en que los elementos se definieron, no que sus elementos se van a ordenar automáticamente.

Veamos un ejemplo simple de como definir una lista:

```
mi_lista = [1, 2, 4]
otra_lista = ["Jose", "Luisa", "Rafaela"]

# tambien puede contener tipos diferentes
variado = [1, "Pedro", 0, "Nombre"]

# Puede contener variables
listas = [mi_lista, otra_lista, variado]

print(listas)
[[1, 2, 4], ['Jose', 'Luisa', 'Rafaela'], [1, 'Pedro', 0, 'Nombre']]
```

Python usa los corchetes [] para delimitar donde empieza y donde termina una lista. Para separar los elementos se usa la coma.

Veamos algunas funciones de las listas:

```
lista = [8, 9]
# agregar un elemento al final
lista.append(3)
print(lista)
# [8, 9, 3]
```

(continué en la próxima página)

(proviene de la página anterior)

```

# eliminar un elemento específico
lista.remove(9)
print(lista)
# [8, 3]
# Cuidado! si se intenta eliminar un elemento que no existe Python
→ lanzar un error

# También es posible eliminar por número de índice y devolver lo
→ eliminado
primer_elemento = lista.pop(0)
print(f'El primero era: {primer_elemento}')

# Para obtener el largo de una lista, se usa la función len()
apellidos = ["gonzalez", "gomez", "rodriguez", "lopez", "garcia"]
total_apellidos = len(apellidos)
print(f'Total de apellidos: {total_apellidos}')
# Total de apellidos: 5

# Ordenar los elementos de una lista
lista.sort()
# Si hay tipos que no son comparables, Python lanzará un error.
# Si todos son números se ordenarán de mayor a menor
# Si todos son strings, se ordenarán alfabéticamente

# Es posible modificar directamente algún elemento de la lista
# usando su número de índice.

lista = [3, 4, 5, 7]
lista[1] = 9
print(lista)
[3, 9, 5, 7]

```

Las listas también permiten obtener rápido cada uno de sus elementos u obtener sub listas incluidas. Usar los corchetes junto a la variable permite acceder a cualquier elemento de la lista por su número de orden (comenzando por cero). Por ejemplo

- `nombre_de_mi_lista[0]` devuelve el **primer** elemento de una lista.
- `nombre_de_mi_lista[1]` devuelve el **segundo** elemento de una lista.

Nota: si se pide un número de elemento (se les llama índices) mayor a los disponibles, Python lanzará un error.

Comprensión de Diccionario en Python: Explicado con ejemplos

Es posible también acceder a los elementos en orden inverso:

- `nombre_de_mi_lista[-1]` devuelve el **último** elemento de una lista.

- `nombre_de_mi_lista[-2]` devuelve el **anteúltimo** elemento de una lista.

Es posible también obtener una sublista: **Nota: esto puede parecer anti-intuitivo y confuso. Requiere práctica habituarse**

- `nombre_de_mi_lista[0:3]` devuelve una nueva lista cuyo primer elemento es `nombre_de_mi_lista[0]` y el último es `nombre_de_mi_lista[2]` (si, hasta el 2). El segundo *parámetro* no está incluido, el primero sí. `_(-_-)_`
- `nombre_de_mi_lista[2:-2]` devuelve una nueva lista cuyo primer elemento es `nombre_de_mi_lista[2]` (el tercer elemento) y el último es `nombre_de_mi_lista[-3]` `_(-_-)_`
- `nombre_de_mi_lista[-2:]` devuelve los últimos dos elementos, desde `nombre_de_mi_lista[-2]` hasta el final (no especificar nada significa hasta el final).

Veamos algunos ejemplos:

```
apellidos = ["gonzalez", "gomez", "rodriguez", "lopez", "garcia"]
primer_apellido = apellidos[0]
print(f'El primer apellido es: {primer_apellido}')
# El primer apellido es: gonzalez

ultimo_apellido = apellidos[-1]
print(f'El último apellido es: {ultimo_apellido}')
# El último apellido es: garcia

primeros_2 = apellidos[0:2]
print(f'Los primeros dos: {primeros_2}')
# Los primeros dos: ['gonzalez', 'gomez']

ultimos_2 = apellidos[-2:]
print(f'Los últimos dos son: {ultimos_2}')
# Los últimos dos son: ['lopez', 'garcia']

# ordenar
apellidos.sort()
print(f'Lista ordenada: {apellidos}')
# Lista ordenada: ['garcia', 'gomez', 'gonzalez', 'lopez', 'rodriguez']

# invertir orden
apellidos.reverse()
print(f'Lista invertida: {apellidos}')
# Lista invertida: ['rodriguez', 'lopez', 'gonzalez', 'gomez', 'garcia']
```

funciones que transforman y funciones que devuelven

Vale la pena notar que `reverse` y `sort` en las listas transforman al objeto que lo llama y no devuelven ningún (`None`) resultado; en cambio en los *strings*, las funciones `replace`, `title` y

otras devuleven un resultado pero no cambian al objeto que los llamó.

¿Puede ser más complicado? Si, un poco más. Podemos usar un tercer parámetro. Este indica los saltos que damos para seleccionar elementos. Predeterminado es 1 (vamos de un elemento al otro).

De esta forma `nombre_de_mi_lista[1:6:2]` significa *los elementos desde el segundo al sexto de dos en dos* (desde `nombre_de_mi_lista[1]` a `nombre_de_mi_lista[5]`). Y `nombre_de_mi_lista[::-1]` significa toda la lista completa en sentido inverso (tambien podemos usar `nombre_de_mi_lista.reverse()` o `nombre_de_mi_lista.sort(reverse=True)`).

12.1 Los *strings* tambien son listas

Python permite tratar a los *strings* como listas. Podemos pensar que una palabra es una lista de letras.

Veamos algunos ejemplos:

```
nombre = "Pedro"
print(f"La primera letra de mi nombre es {nombre[0]}")
print(f"La última letra de mi nombre es {nombre[-1]}")
```

12.2 Función `split` de los *strings*

Si quiero separar una frase en palabras Python ya incluye la funcion `split` en los *strings*. Esta función devuelve un objeto de tipo lista.

Veamos un ejemplo:

```
frase = "Era el mejor de los tiempos y era el peor de los tiempos"
palabras = frase.split()
print(palabras)
['Era', 'el', 'mejor', 'de', 'los', 'tiempos', 'y', 'era', 'el', 'peor', 'de', 'los', 'tiempos']
```

La función `split` tiene un parámetro llamado `separator` que tiene como valor predeterminado " " (un espacio, su uso más común). Esté parámetro indica que *caracter* se va a usar para separar los elementos de la lista resultande.

Existen casos en que necesitamos separar por otros carcateres.

Veamos un ejemplo:

```
raw_data = "juana,pedro,fabiana,victor,jose,laura"
nombres = raw_data.split(',')
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(nombres)
['juana', 'pedro', 'fabiana', 'victor', 'jose', 'laura']
```

12.3 Tareas

- Escribir una funcion que dada una palabra devuelva su tercera letra.
- Escribir una funcion que reciba cuatro parametros obligatorios.
 - Devuelva una lista
 - Esta lista debe contener tres de los cuatro elementos (hay que quitar el que sea más pequeño)
 - La lista devuelta debe estar ordenada de mayor a menor.
 - Ejemplo: `ordenar_y_quitar(4, 8, 9, 12)` debe devolver `[12, 9, 8]`.
- Escribir una funcion que reciba un parametro llamado `nombre_completo` y devuelva una lista de tres elementos (siempre con tres elementos).
 - El primero de los elementos de la lista devuelta debe ser la primera palabra de `nombre_completo` (separada con la función `split`)
 - Si `nombre_completo` separado con `split` tiene solo un elemento, agregar dos *strings* vacios para cumplir el requisito de devolver una lista con tres elementos.
 - Si el `nombre_completo` tiene solo dos palabras, devolver una lista de la forma `['palabra1', '', 'palabra2']`
 - Si el `nombre_completo` tiene tres palabras, devolver una lista de la forma `['palabra1', 'palabra2', 'palabra3']`
 - Si el `nombre_completo` tiene **más** de tres palabras, devolver una lista de la forma `['palabra1', 'palabra2', 'palabra3']`
 - Algunos ejemplos (suponiendo que la funcion se llame `separar_nombre`, puede ser otro). Probarlos todos para asegurarse que funciona como es esperado.
 - * `separar_nombre('Juan')` debe devolver `['Juan', '', '']`
 - * `separar_nombre('Juan Perez')` debe devolver `['Juan', '', 'Perez']`
 - * `separar_nombre('Juan Carlos Perez')` debe devolver `['Juan', 'Carlos', 'Perez']`
 - * `separar_nombre('Juan Carlos Perez Valdez')` debe devolver `['Juan', 'Carlos', 'Perez']`

12.4 Algunos ejemplos de uso

```
lista = [1, 2]
lista.append(3)

print(lista)
# [1, 2, 3]

# pueden contener cualquier tipo de dato
lista = [1, "a", []]
lista.remove("a")
print(lista)
# [1, []]
```

```
lista = ["Juan", "Pedro", "Maria"]

for persona in lista:
    print(f'Hola {persona}')

""" Muestra
Hola Juan
Hola Pedro
Hola Maria
"""
```

```
# Listas por/de compresion
# mas ejemplos https://www.analyticslane.com/2019/09/23/listas-por-compresion-en-python/

lista_base = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista = [x for x in lista_base if x > 5]

""" También podría escribirse como
lista = [
    x
    for x in lista_base
    if x > 5
]
"""

# Equivalente a:
lista2 = []
for x in lista_base:
    if x > 5:
        lista2.append(x)
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(f"Lista 1 {lista}")
print(f"Lista 2 {lista2}")

print(f"lista == lista2 {lista==lista2}")

""" muestra
Lista 1 [6, 7, 8, 9, 10]
Lista 2 [6, 7, 8, 9, 10]
lista == lista2 True
"""
```


CHAPTER 13

Iterando ando: for

En Python algunos objetos se pueden iterar. Esto es: **recorrer su elementos de uno a la vez**. Muchos objetos tiene definida una forma de iterarse, por ejemplo las listas.

Veamos un ejemplo:

```
mi_lista = ["Juan", "Pedro", "María"]

for persona in mi_lista:
    saludo = f'Hola {persona}!'
    print(saludo)
print('Iteración terminada')

""" Imprime
Hola Juan
Hola Pedro
Hola María
Iteración terminada
"""
```

Como podemos imaginar, la línea `print(saludo)` se ejecuta tres veces. Estos es, una por cada uno de los elementos de `mi_lista`. La sentencia `for` inicia un loop que **va a terminar cuando se quede sin elementos que recorrer**.

13.1 Anatomía de un for

La línea `for persona in mi_lista:` podría traducirse como: *por cada persona en mi_lista hacer lo siguiente:*

Analicémosla:

- La sentencia `for` indica que vamos a iniciar una iteración.
- A continuación se requiere un nombre de variable (en nuestro ejemplo `persona`) para contener cada uno de los elementos de la iteración dentro de la porción de código que se va a ejecutar en cada vuelta del bucle.
- Luego sigue la palabra `in` para que esta línea sea fácil de leer y como indicador de que vamos a continuar con algún objeto iterable.
- Finalmente se coloca el objeto sobre el que deseamos iterar.
- Como todas las líneas que preceden a un bloque de código termina con `:`.
- Todo el código que necesitamos que se ejecute para cada uno de los elementos debe estar tabulado. **La forma de saber donde termina esta porción de código es que se termina la tabulación.**
- Nota: la variable `persona` solo está definida dentro del bloque del código del `for`. Fuera de él no está definida y dará un error si se intenta usar.

¿Se puede anidar iteraciones? Si, claro. Veamos un ejemplo:

```
lista_con_listas = [ [1, 12], [20, 22], [5, 8] ]

for lista in lista_con_listas:
    for numero in lista:
        print(numero)

# imprimirá:
1
12
20
22
5
8
```

13.2 continue y break en ciclos for

Dentro de una iteración, existen formas de salir de ella (break) y de saltarse un elemento (continue).

Veamos un ejemplo:

```
# for / continue / break
países = ["Argentina", "Brasil", "Chile", "Uruguay", "Venezuela",
→ "Colombia", "Japón"]
print('Países')
for país in países:
    if país == "Chile":
        continue
    print(f" - {país}")
    if país == "Venezuela":
        break
print('FIN')
""" imprime
Países
- Argentina
- Brasil
- Uruguay
- Venezuela
FIN
"""
```

Algunos elementos no pueden iterarse, por ejemplo los int. El error 'XXX' object is not iterable es la forma de informar esto.

```
a = 90
for x in a:
    print(x)

""" devolverá el error.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
"""
```

13.3 if + in

Tambien podemos combinar if con in para saber si un objeto esta contenido en una lista.

```
lista = [1, 2, 3]

if 4 in lista:
    print("4 está en la lista")
else:
    print("4 no está en la lista")
# imprime
# 4 no está en la lista
```

13.4 Iterando strings

Como ya vimos antes, los strings tambien pueden usarse como listas y por lo tanto se puede iterar.

```
nombre = "Victor"

for letra in nombre:
    print(letra)

""" imprimirá
V
i
c
t
o
r
"""
```

13.5 Agregado: range

Python tiene una funcion (técnicamente no es una función pero por ahora podemos pensarla como tal) incluida (*built-in*) que permite iterar sobre una serie de números. Se llama range y podemos ver su funcionamiento con algunos ejemplos. Cuando llamamos a range con un solo un parámetro (siempre numeros) este devuelve un objeto que se puede iterar. Incluye los números desde cero hasta el valor pasado como parámetro menos uno (comportamiento similar a los indices de las listas).

```
for n in range(3):
    print(n)
# imprimira (en lineas diferentes): 0 1 2
```

Cuando llamamos a `range` con dos parámetros (siempre numeros) este devuelve un objeto que se puede iterar. Incluye los números desde el primer parámetro hasta el valor pasado como segundo parámetro menos uno.

```
for n in range(2, 8):
    print(n)
# imprimira (en lineas diferentes): 2 3 4 5 6 7
```

Cuando llamamos a `range` con tres parámetros (siempre numeros) este devuelve un objeto que se puede iterar. Incluye los números desde el primer parámetro hasta el valor pasado como segundo parámetro menos uno. El último parámetro indica el tamaño del salto entre un elemento y otro (por ejemplo 2 hará que el resultado vaya de dos en dos).

```
for n in range(3, 10, 2):
    print(n)
# imprimira (en lineas diferentes): 3 5 7 9
```

Tambien es posible convertir el resultado de `range` a una lista (que ya conocemos).

```
pares = range(0, 100, 2)
lista_pares = list(pares)
print(lista_pares)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64,
66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
```

13.6 Iterar *mientras* que algo suceda: `while`

En algunas ocasiones necesitamos iterar hasta que algo cambie. Por ejemplo, hasta que el usuario ingrese un número válido.

Para estos casos existe la sentencia `while`.

```
numero_final = 0

# No salimos hasta que el numero sea válido
while numero_final == 0:
    numero = input("Ingrese un número entre 1 y 10: ")
    if not numero.isdigit():
        print("No ingresaste un número válido")
        continue
    if int(numero) < 1 or int(numero) > 10:
        print("El número debe estar entre 1 y 10")
        continue
    numero_final = int(numero)

print(f"Ingresaste el número {numero_final}")
```

(continué en la próxima página)

(proviene de la página anterior)

```
""" TEST

Ingrese un número entre 1 y 10: a
No ingresaste un número válido
Ingrese un número entre 1 y 10: 11
El número debe estar entre 1 y 10
Ingrese un número entre 1 y 10: 0
El número debe estar entre 1 y 10
Ingrese un número entre 1 y 10: 3
Ingresaste el número 3

"""
```

13.6.1 Tareas

- Escribir un programa que:
 - Incluya al inicio una funcion llamada `es_par` y que devuelva *verdadero* o *falso* según corresponda
 - Le solicite al usuario que ingrese una lista de números separados por coma.
 - Transformarme este texto ingresado a lista con `split(',')`
 - Iterere todos ellos e imprima solo aquellos que son pares.
- Escribir un programa que itere todos los múltiplos de 3 desde el cero al 500000 e imprima solo aquellos que además son divisibles (el resto de la division es cero) por 13.
- Escribir un programa que responda estas preguntas:
 - ¿Cuantos números multiplos de siete menores a 10.000 terminan en 999?
 - ¿Cuales son?

CHAPTER 14

Cuidando nuestro código: git

Git es un sistema de control de versiones para repositorios de código fuente descentralizado basado en un modelo de ramificación. Actualmente es el instrumento más usado en la industria del software para administrar código fuente en equipos de trabajo.

14.1 Control de versiones

Git permite hacer seguimiento del código que escribimos y de todos los cambios que le hacemos. Es posible saber en cada momento cuando y quien escribió cada línea de código. También es posible ver y volver a versiones anteriores cuando sea necesario.

14.2 Descentralizado

No necesitamos un servidor central donde almacenar todo el código junto a sus ramas y versiones. Cada usuario puede tener copias completas e independientes del material.

14.3 Modelo de ramificación

Git permite (y te alienta) a que uses múltiples *ramas* de trabajo (usaremos indistintamente la palabra *branch* del inglés). Cada cambio en el código (para corregir errores o agregar nuevas funcionalidades) puede hacerse en una rama de trabajo independiente que fácilmente puede integrarse a otras ramas de trabajo.

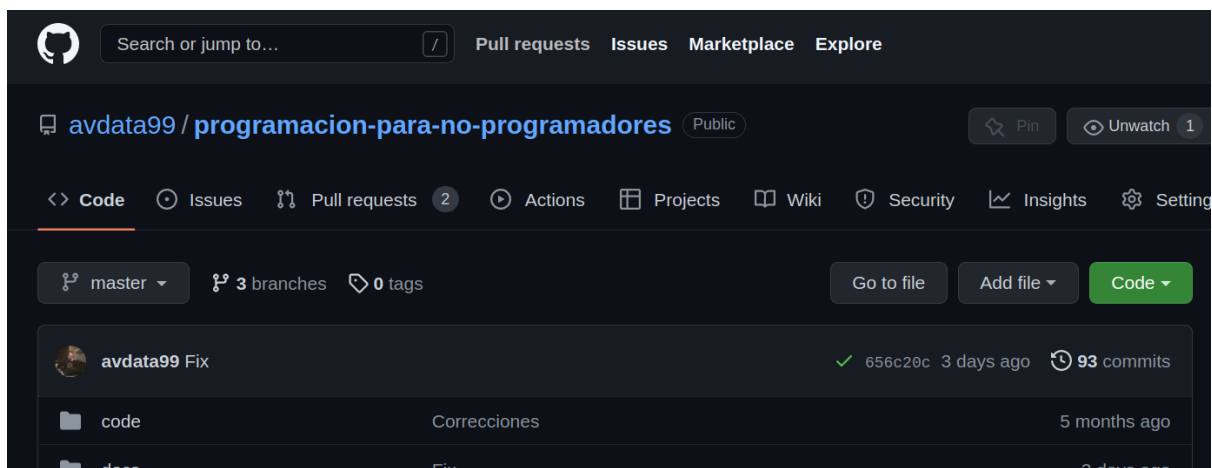
CHAPTER 15

Git en la nube: GitHub

Luego de la aparición de Git surgieron muchos servicios que permitían mantener una copia de tus repositorios en internet. Todas las empresas de software usan alguno de estos servicios por lo que trabajar con desarrollador requiere siempre conocer y usar estos servicios.

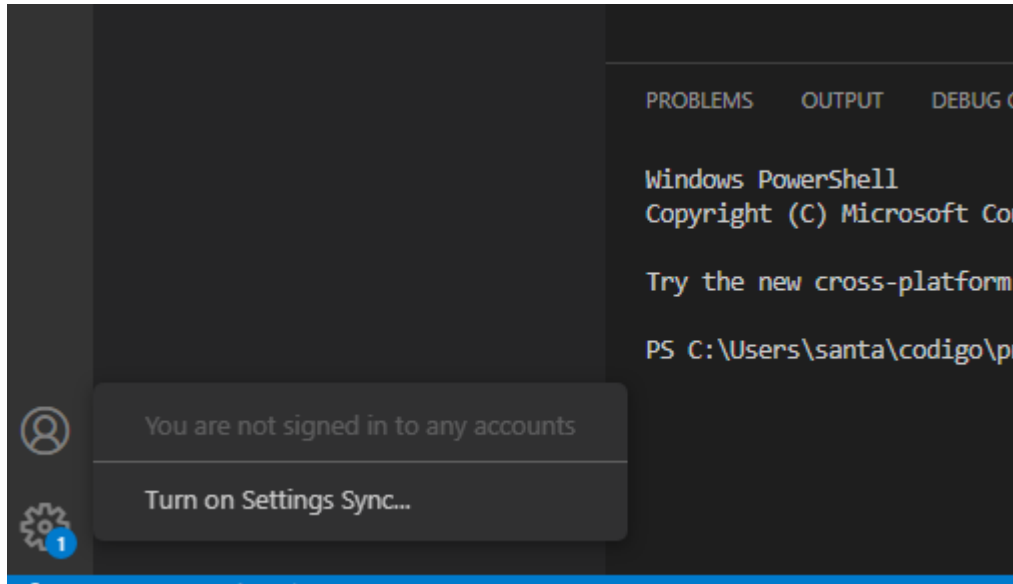
El más popular (pero no el único) de ellos actualmente es [GitHub](#) y es por esto que lo usaremos en este curso. Es importante que accedas al sitio y crees una cuenta (son gratuitas para la mayoría de sus funciones).

Un ejemplo de repositorio de código es este mismo curso. Puedes ver el repositorio que incluye este manual y documentación [aquí](#).

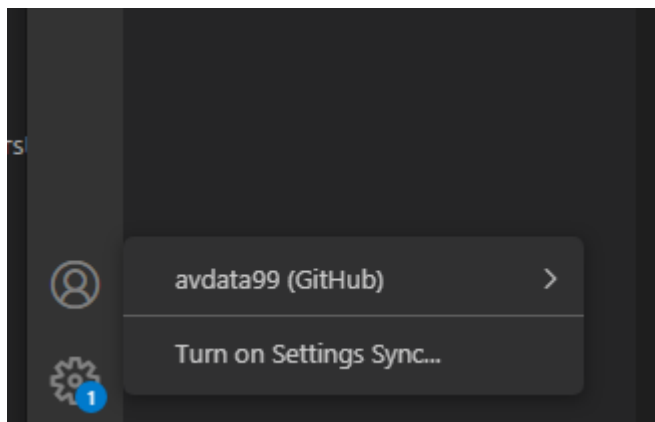


15.1 GitHub desde Visual Studio Code

Visual Studio Code incluye la posibilidad de conectar tu cuenta de GitHub. Esto es muy útil para conectarte a repositorios de código y enviar tus sugerencias o cambios.



Después de un proceso de conectar tu cuenta de GitHub, esta ya (casi) quedará disponible para usar desde Visual Studio Code.



Finalmente, debes ir al menú *View -> Terminal* y colocar:

```
git config --global user.email "tu-email@xxx.com"
git config --global user.name "tu_nombre_de_usuario"
```

The screenshot shows the Visual Studio Code interface. The editor has a file named 'ejercicio.py' open, containing a Python script with a task description in Spanish. The terminal window at the bottom shows the execution of 'git config' commands to set the user's email and name.

```

ejercicios > ejercicio-040 > ejercicio.py > process_data
1 """
2 Tarea: corregir la funcion para que dado el parametro "data"
3 lo separe en líneas y lo transforme en una lista sin elementos vacios.

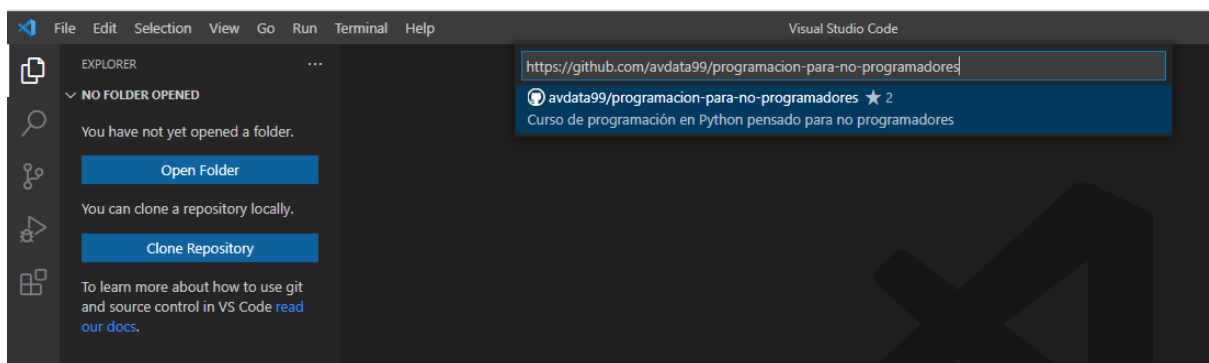
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

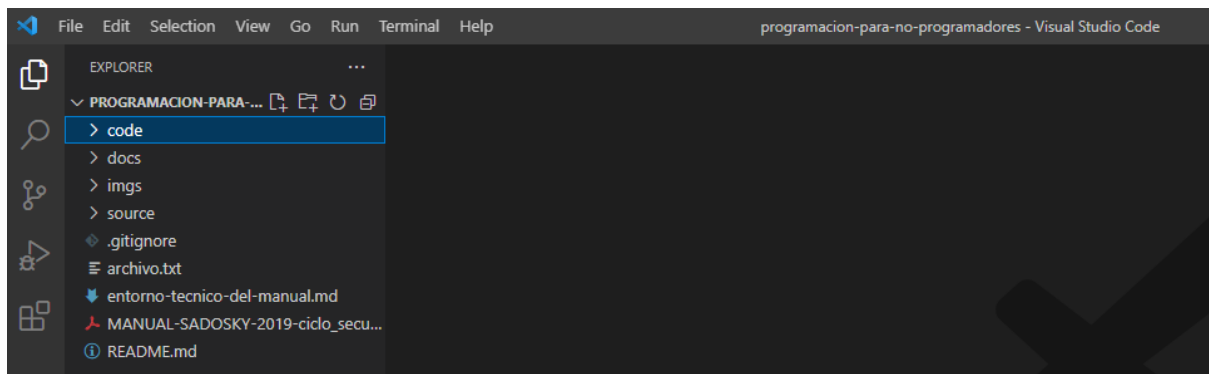
Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\santa\Documents\codigo\ejercicios-python-101> git config --global user.email "andres@data99.com.ar"
PS C:\Users\santa\Documents\codigo\ejercicios-python-101> git config --global user.name "avdata99"
PS C:\Users\santa\Documents\codigo\ejercicios-python-101>
    
```

Para comenzar a trabajar sobre un repositorio de código debes usar el buscador o colocar la URL del repositorio: <https://github.com/avdata99/programacion-para-no-programadores>



Luego de definir en que carpeta quedará tu copia de este repositorio podrás comenzar a usarlo.



15.1.1 Tarea

- Compartí con tus compañeros de curso tu nombre de usuario en [GitHub](#) para comenzar a trabajar en equipo.
- Conectar tu cuenta de GitHub a Visual Studio Code.
- Clonar el [repositorio del curso](#)

CHAPTER 16

Mi primer *PR*

Como dijimos anteriormente, Git nos permite tener diferentes ramas de trabajo. De esta forma, existe una rama principal (llamada *main*, *master* o como cada equipo prefiera) de trabajo donde todos los cambios aceptados se integran consolidando la versión oficial de tu producto de software.

La forma de enviar tus sugerencias de cambios al equipo es crear una nueva rama de trabajo con tus cambios y solicitar la revisión a otros integrantes del equipo.

Esta solicitud es conocida como *PR* y viene de *Pull request*. En otras plataformas se le llama *MR* (por *Merge Request*).

16.1 Pull request paso a paso

Ten en cuenta estos pasos cada vez que quieras proponer cambios en un proyecto de software.

16.1.1 Paso 1: Actualizar tu repositorio

En primer lugar debes asegurarte que tu versión del código esta actualizada para que tus cambios sean propuestos sobre la última versión del código. Comenzar una rama a partir de una versión *vieja* de la rama principal es un error común.

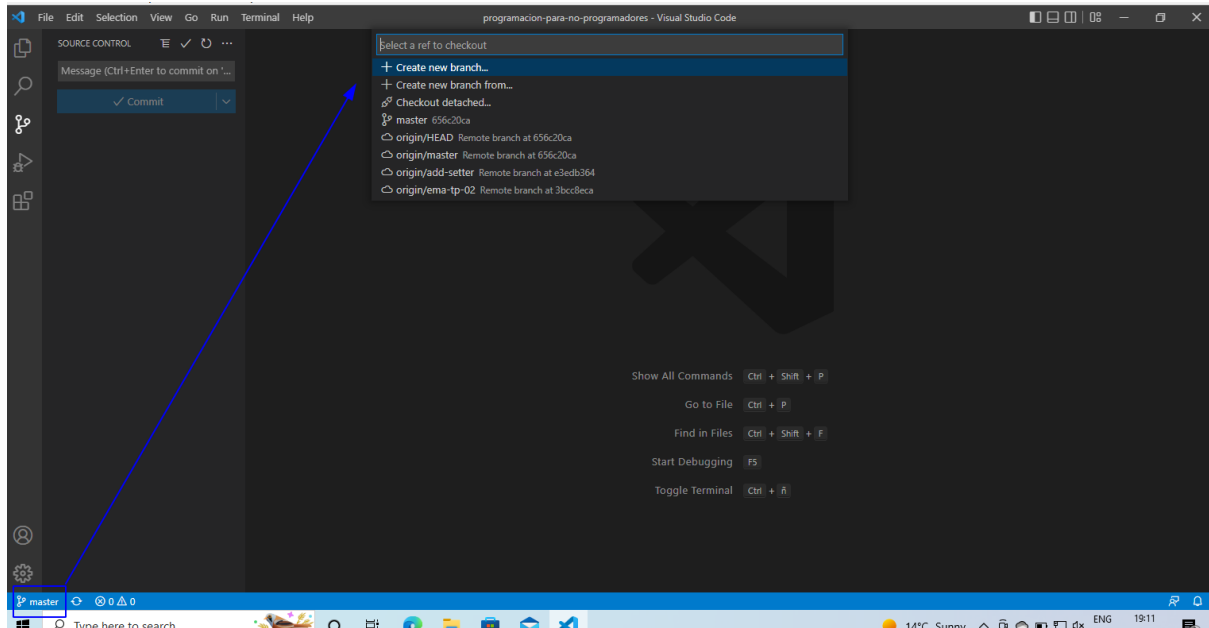
Antes de comenzar debes asegurarte de estar en la rama principal y de que esta este actualziada. Esto se consigue con estos dos pasos (asumimos que la rama principal se llama *master* pero podría ser otra).

Desde tu terminal esto puede hacerse así:

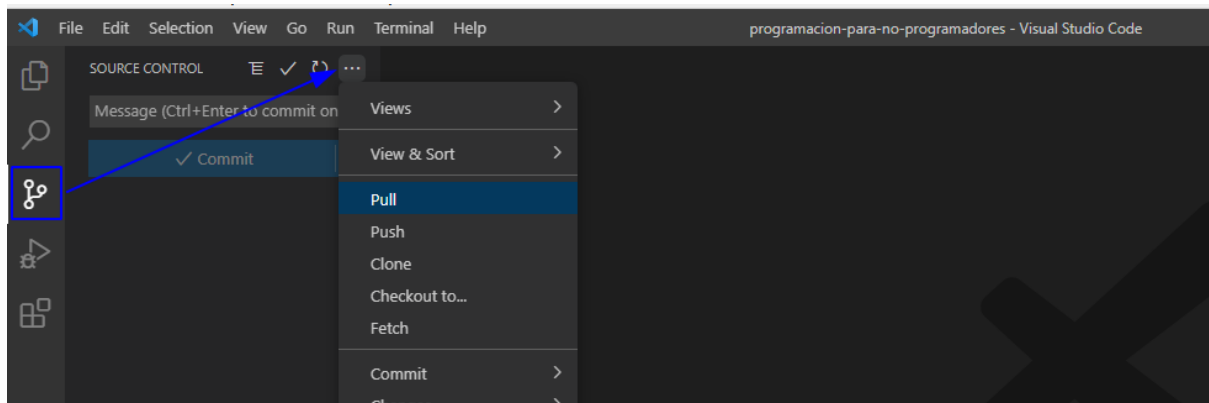
```
# posicionarme en la rama principal
git checkout master
# traer (pull) los ultimos cambios
git pull
```

Desde Visual Studio Code puede hacerse gráficamente.

Checkout (posicionarse en una rama):

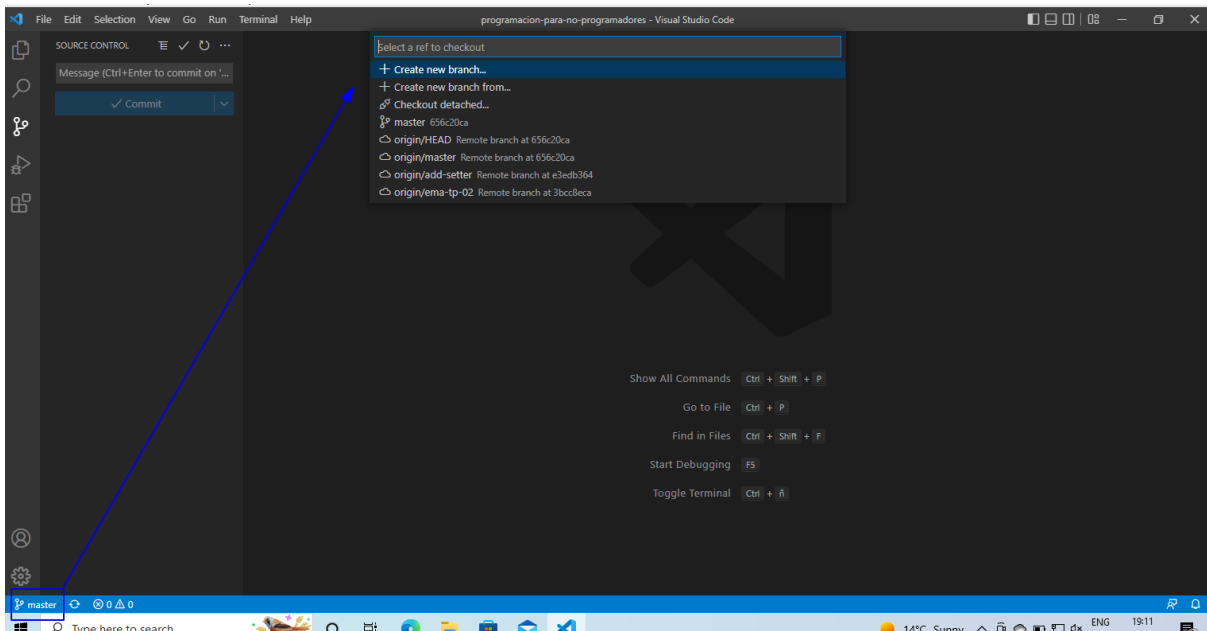


Pull (traer los ultimos cambios):



16.1.2 Paso 2: Crear una nueva rama

Crear nueva rama (*Create new branch*) con un nombre descriptivo de los cambios que pensas implementar.



Desde tu terminal esto puede hacerse con:

```
# posicionarme en la rama principal
git checkout -b nombre_de_mi_rama
```

La rama de trabajo actual estará siempre visible en VSC abajo a la izquierda en la pantalla. Es importante saber en cada momento en que rama de trabajo estamos.

16.1.3 Paso 3: Hacer los cambios en el código

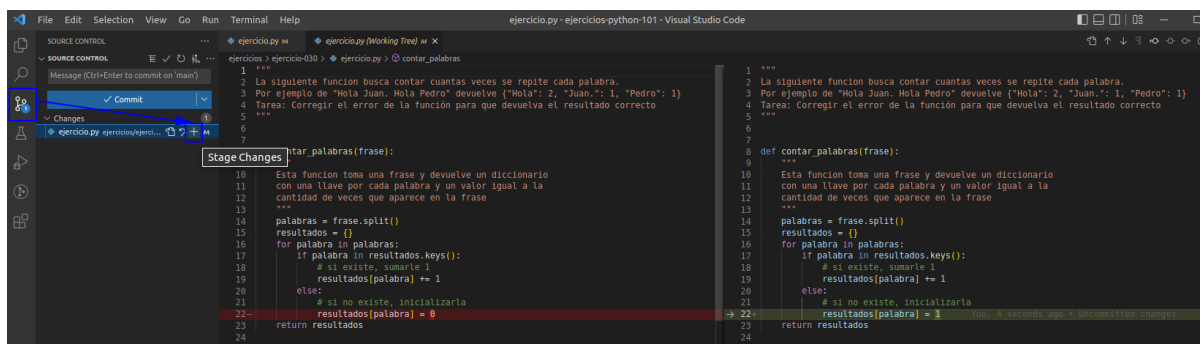
Modificar o crear uno o más archivos con los cambios que esperamos proponer. Es conveniente asegurarnos de probar los cambios localmente para estar seguros que funcionan según esperamos.

16.1.4 Paso 4: Agregar los archivos con cambios

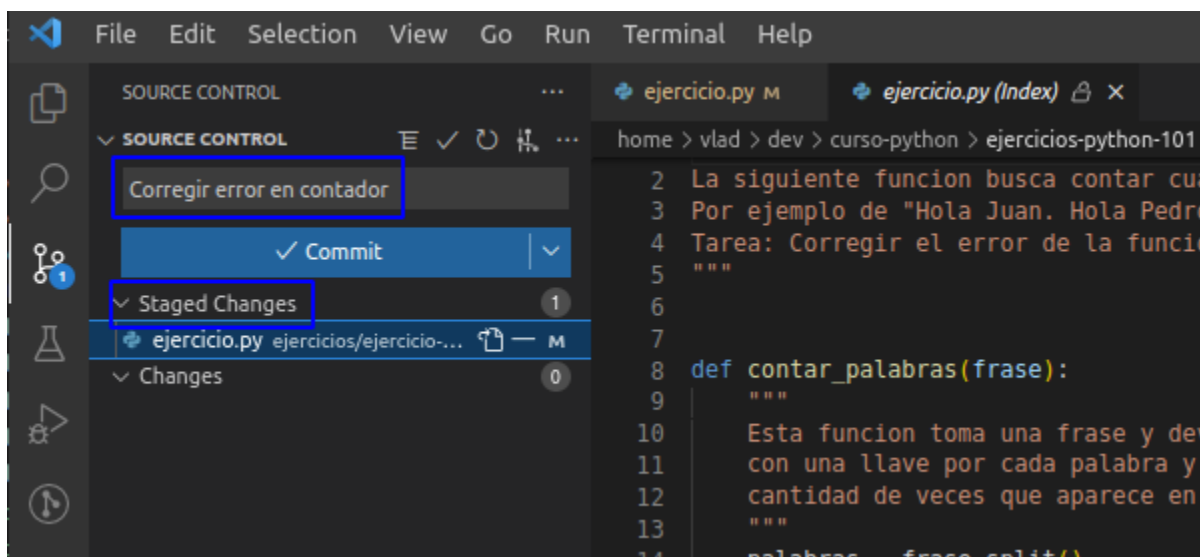
Una vez hechos los cambios debemos agregar estos archivos a lo que será nuestro *commit*. Un *commit* en Git es un conjunto de cambios en uno o más archivos.

Desde tu terminal esto puede hacerse con:

```
# agregar los cambios en un archivo a lo que será nuestro commit
git add archivo_que_cambio.py
```



Todos los archivos agregados con *add* pasan a estar preparados para hacer tu *commit* (se llama *staged* a esta *area de preparación*). En la parte superior debes colocar un mensaje que describa el cambio que estas haciendo.



16.1.5 Paso 5: Commit

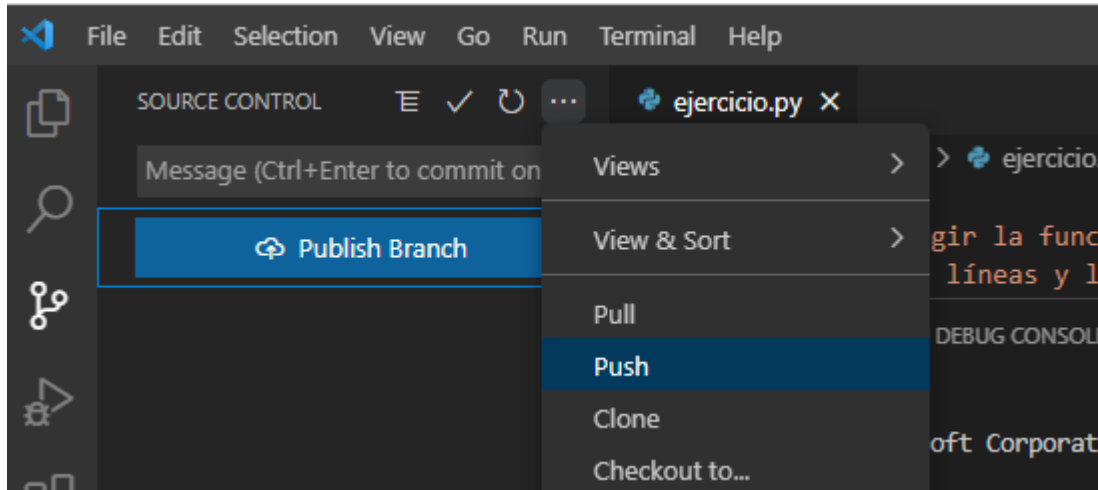
Finalmente con el boton *commit* se consolida este cambio en la rama actual. Desde tu terminal esto puede hacerse con:

```
# agregar los cambios en un archivo a lo que será nuestro commit
git commit -m "Descripcion de mi cambio"
```

Nota importante Este *commit* solo existe en tu computadora. Al clonar el repositorio de GitHub, creamos un nodo local e independiente de aquel y es allí donde todos estos cambios quedan registrados hasta que finalmente hacemos un *push*.

16.1.6 Paso 6: Push

En este paso enviamos los cambios que tenemos localmente al repositorio en GitHub. De esta forma otros usuarios podrían descargarse nuestra rama (por ejemplo para colaborar en ella).



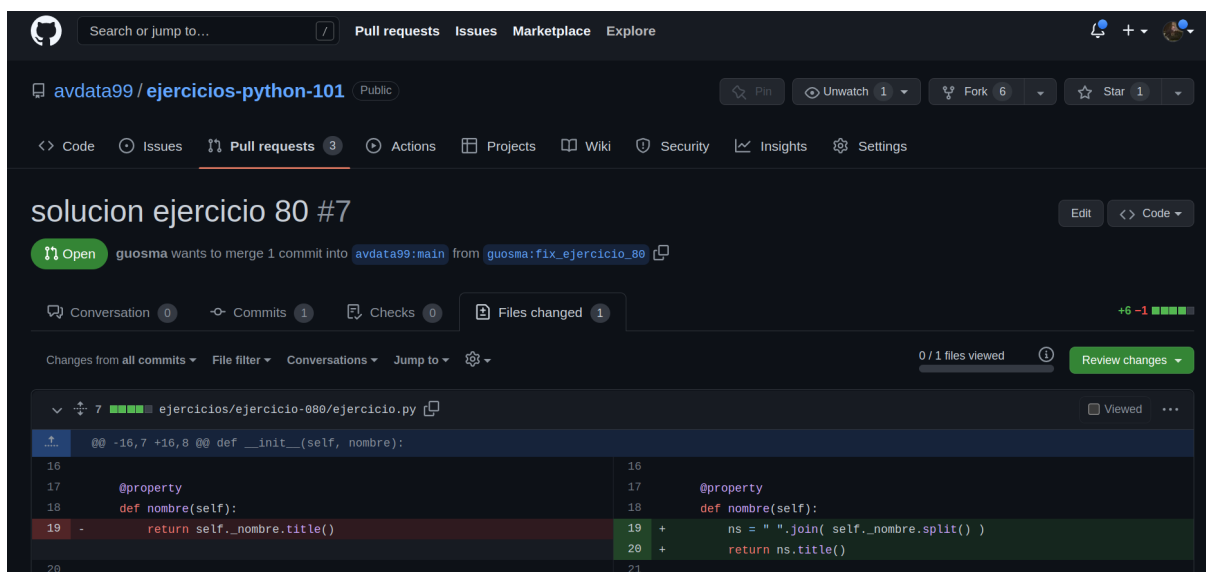
Desde tu terminal esto puede hacerse con:

```
# agregar los cambios en un archivo a lo que será nuestro commit
git push
```

Este ciclo de hacer cambios (paso 3) hasta el push puede hacerse las veces que sea necesario, incluso después de creado el PR en el paso siguiente.

16.1.7 Paso 7: Pull Requets

Finalmente desde VSC o desde la página web del repositorio podemos iniciar un *PR* y asignar uno o más usuarios como revisores.



Es posible que los revisores soliciten nuevos cambios o correcciones. En caso de ser necesario podemos volver a nuestra rama más tarde y actualizar el PR haciendo nuevos *commits* hasta que

finalmente nuestro trabajo quede resuelto.

16.1.8 Tarea

- Buscar en el [repositorio de este curso](#) (al que ya nos conectamos) cualquiera de los archivos `.rst` de la carpeta `source` algún error ortográfico o sugerencia sobre el texto de este manual y enviarlo como PR.
- La carpeta [ejercicios](#) de este repositorio tiene ya problemas para resolver mediante *pull requests*. Se espera un *PR* para cada ejercicio, por lo tanto cada uno requiere una rama distinta **que siempre** salga desde master. **Asegurarse de seguir todos los pasos para cada PR.**
 - Hacer un PR con una propuesta de solución para el [ejercicio 001](#) (contenido en este repositorio)

```
"""
Completar la funcion para que devuelva "Hola NOMBRE"
segun el "nombre" (string) que se pasa como parametro.
"""

def hola(nombre):
    pass

# -----
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se
# considera completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
# continuacion es falso.
# Si usas GitHub (o similares) puedes hacer una nueva rama con esta
# solución,
# crear un "pull request" y solicitar revision de un tercero.

assert hola("Juan") == "Hola Juan"
assert hola("Pedro") == "Hola Pedro"

print('Ejercicio terminado OK')
```

- Hacer un PR con una propuesta de solución para el [ejercicio 002](#) (contenido en este repositorio)

```
"""
Completar la funcion para que devuelva el resultado de la suma
de los parámetros pasados a y b.
"""

def suma(a, b):
    pass

# -----
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se
# considera completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
#   continuacion es falso.
# Si usas GitHub (o similares) puedes hacer una nueva rama con esta
# solución,
#   crear un "pull request" y solicitar revision de un tercero.

assert suma(2, 6) == 8
assert suma(3, 3) == 6

print('Ejercicio terminado OK')
```


CHAPTER 17

Diccionarios: dict

Los diccionarios en Python son una estructura de datos preparadas para almacenar colecciones de elementos (separados por coma) compuestos de una clave y un valor (suele llamarlos en ingles *key*, *value*). Los diccionarios se delimitan con llaves {}.

Veamos un ejemplo:

```
persona = {
    "nombre": "Luis",
    "apellido": "Colon",
    "edad": 32
}
type(persona)
# devuelve <class 'dict'>
```

Los valores pueden accederse directamente desde sus claves. Esto es así tanto para leer como para modificar cada elemento. Las claves son únicas, no puede haber dos elementos con la misma clave.

```
persona = {
    "nombre": "Luis",
    "apellido": "Colon",
    "edad": 32
}

print(f'Nombre: {persona["nombre"]} {persona["apellido"]} tiene
↪{persona["edad"]} años')
'Luis Colon tiene 32 años'

print(persona["nombre"])
```

(continué en la próxima página)

(proviene de la página anterior)

```
'Luis'

edad = persona["edad"]
print(edad)
32

persona["apellido"] = "Gonzalez"
print(persona)
{"nombre": "Luis", "apellido": "Gonzalez", "edad": 32}
```

En algunas ocasiones no vamos a tener certeza de que las claves a las que queremos acceder existen.

```
persona = {
    "nombre": "Luis",
    "apellido": "Colon",
    "edad": 32
}

# La siguiente línea lanzará un error (la clave no existe)
persona["clave_que_no_exite"]

# Para acceder a elementos de los que no sabemos si su clave existe,
# usamos la función "get" de los diccionarios
persona.get("clave_que_no_exite")
# Si la clave (key) no existe la anterior línea no va a lanzar
# un error. Simplemente devolverá None (es el "nada" de Python)

# Si queremos un valor predeterminado para cuando la clave no
# esta definida podemos usar el segundo parámetro opcional de "get"
persona.get("clave_que_no_exite", "valor_predeterminado")

# Tambien es posible "preguntar" si una clave existe
if "clave_buscada" in persona:
    print("clave_buscada SI existe como clave en 'persona'")

# Esto es posible porque los diccionarios definen un iterador con la
# lista de sus claves

for key in persona:
    print(f"Clave en persona: {key}")
```

Cada par de clave-valor es denominado como elemento (item). La función `items` de los diccionarios devuelve un objeto iterable que podemos navegar con `for` y que devuelve en cada paso una clave y un valor.

```

persona = {
    "nombre": "Juan",
    "apellido": "Colon",
    "edad": 32
}

for k, v in persona.items():
    print(f'Item encontrado: Key:{k}, Value: {v}')

# Item encontrado: Key:nombre, Value: Juan
# Item encontrado: Key:apellido, Value: Colon
# Item encontrado: Key:edad, Value: 32

```

Los valores puede ser de cualquier tipo e incluso conformar estructuras muy complejas.

Ejemplo:

```

persona = {
    "nombre": "Luis",
    "apellido": "Colon",
    "edad": 32,
    "estudios": {
        "primario": True,
        "secundario": True,
        "terciario": False,
        "universitario": False
    },
    "experiencia_laboral": {
        "2005-2008": {
            "empresa": "Ferreteria el cosito del coso",
            "cargo": "Vendedor",
            "sueldo": 45000,
            "tareas": ["atender publico", "compras"]
        },
        "2009-2011": {
            "empresa": "Escuela Tecnica San Martin",
            "cargo": "profesor",
            "sueldo": 75000,
            "tareas": ["dictar clases", "elaborar cursos"]
        }
    }
}

```

En el ejemplo anterior `persona["estudios"]` es a su vez un diccionario por lo que tiene a su vez las propiedades de un nuevo objeto de este tipo. Las siguientes líneas son válidas para la estructura recién definida.

```
print(persona["estudios"]["primario"])
True

if persona["estudios"]["secundario"]:
    print('La persona termino el secundario')
```

En el ejemplo anterior `persona["experiencia_laboral"]["2005-2008"]["tareas"]` es a su vez una lista por lo que tiene a su vez las propiedades de estas. Las siguientes líneas son válidas para la estructura recién definida.

```
for tareas in persona["experiencia_laboral"]["2005-2008"]["tareas"]:
    print(f"Tarea: {tarea}")
```

También es posible usar `dict()` para crear un diccionario.

```
d3 = dict(
    nombre='Laura',
    edad=47,
    documento=221029489
)
type(d3)
# muestra <class 'dict'>
print(d3)
{'nombre': 'Laura', 'edad': 47, 'documento': 221029489}
```

No confundir diccionarios con set

En Python también se usan las llaves `{}` para definir conjuntos (*sets*).

Por ejemplo `s = {1, 2, 3, 4, 5}` no es un diccionario, es un *set* y `type(s)` mostrará `<class 'set'>`

Los *sets* son listas no ordenadas de elementos únicos (los duplicados se eliminan automáticamente).

17.1 Tareas

- Hacer un PR con una propuesta de solución para el [ejercicio 020](#)

```
"""
Completar la funcion para que devuelva la "frase" pasada como parámetro
reemplazadas todas sus vocales con la "a" (o cualquier otra "vocal"
→ que se
pase como parámetro)
"""
```

(continué en la próxima página)

(proviene de la página anterior)

```

def cambia_vocales(frase, vocal="a"):
    pass

# -----
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
#   continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta
# solución,
#   crear un "pull request" y solicitar revision de un tercero.

assert cambia_vocales("hola") == "hala"
assert cambia_vocales("Juan Carlos") == "Jaan Carlas"
assert cambia_vocales("Pepito", "e") == "Pepete"
assert cambia_vocales(vocal="i", frase="me llamo juan") == "mi llimi
# revisar mayúsculas y minúsculas
assert cambia_vocales("Holi") == "HALa"

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el ejercicio 030

```

"""
La siguiente funcion busca contar cuantas veces se repite cada palabra.
Por ejemplo de "Hola Juan. Hola Pedro" devuelve {"Hola": 2, "Juan.": 1,
# "Pedro": 1}
Tarea: Corregir el error de la función para que devuelva el resultado
# correcto.
Se espera que los signos de puntuacion no afecten el resultado y que
# las mayusculas
y minusculas no cuenten como palabras diferentes.
"""

```

(continué en la próxima página)

(proviene de la página anterior)

```

def contar_palabras(frase):
    """
    Esta funcion toma una frase y devuelve un diccionario
    con una llave por cada palabra y un valor igual a la
    cantidad de veces que aparece en la frase
    """
    palabras = frase.split()
    resultados = {}
    for palabra in palabras:
        if palabra in resultados.keys():
            # si existe, sumarle 1
            resultados[palabra] += 1
        else:
            # si no existe, inicializarla
            resultados[palabra] = 0
    return resultados

# -----
# ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# ---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
# continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta
# solución,
# crear un "pull request" y solicitar revision de un tercero.

f1 = contar_palabras("Hola dijo Juan. Hola dijo pedro")
assert f1['Hola'] == 2

himno = "Oid mortales el grito sagrado. Libertad, libertad, libertad.
Oid el ruido de rotas cadenas"
palabras_himno = contar_palabras(himno)
assert palabras_himno['Oid'] == 2
assert palabras_himno['libertad'] == 3

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el ejercicio 031

```

"""
La siguiente funcion toma como parámetro una lista de diccionarios
y cuenta la cantidad de elementos que tiene un valor especifico en
una propiedad definida.
Por ejemplo de la lista
lista = [
    {"genero": "M", "nombre": "Juan"},
    {"genero": "F", "nombre": "Pablo"},
    {"genero": "F", "nombre": "Juana", "apellido": "Gomez"},
    {"genero": "M", "nombre": "Victor"},
    {"genero": "M", "nombre": "Juan Pablo", "apellido": "Velez"},
    {"genero": "F", "nombre": "Juana"},
    {"genero": "F", "nombre": "Victoria"}
]
Se esperan estos posibles resultados

contar_si(lista, "genero", "M") = 3
contar_si(lista, "genero", "F") = 4
contar_si(lista, "nombre", "Juana") = 2

pero la funcion da error en algunos casos como
contar_si(lista, "apellido", "Gomez")
donde en realidad esperamos que devuelva 1

Tarea: Mejorar la función para que no de errores cuando una clave no
→ existe
"""

def contar_si(lista, propiedad, valor):
    """
    Esta funcion cuenta la cantidad de elementos (diccionarios) en
    → "lista"
    que tiene una "propiedad" con un "valor" específico.
    """
    contador = 0
    for elemento in lista:
        if elemento[propiedad] == valor:
            contador += 1

    return contador

# -----
→ ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
→ ---

```

(continué en la próxima página)

(proviene de la página anterior)

```

# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
→completado.
# La instruccion "assert" de Python lanzará un error si lo que se
→indica a
#   continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta
→solución,
#   crear un "pull request" y solicitar revision de un tercero.

lista = [
    {"genero": "M", "nombre": "Juan"},
    {"genero": "F", "nombre": "Pablo"},
    {"genero": "F", "nombre": "Juana", "apellido": "Gomez"},
    {"genero": "M", "nombre": "Victor"},
    {"genero": "M", "nombre": "Juan Pablo", "apellido": "Velez"},
    {"genero": "F", "nombre": "Juana"},
    {"genero": "F", "nombre": "Victoria"}
]

assert contar_si(lista, "genero", "M") == 3
assert contar_si(lista, "genero", "F") == 4
assert contar_si(lista, "nombre", "Juana") == 2
assert contar_si(lista, "apellido", "Gomez") == 1
assert contar_si(lista, "apellido", "Perez") == 0

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el ejercicio 032

```

"""
El siguiente código permite crear facturas de ventas y agregar items.
El código no tiene fallas pero el cliente desea que la lista de items
no tenga productos duplicados. Si se intenta agregar un producto por
segunda vez, la función debería darse cuenta y actualizar los valores
de ese item y no agregarlo como nuevo item.
"""

def agregar_item(factura, producto, precio_unitario, cantidad=1):
    """
    Agregar un item a una factura que se pasa como parámetro
    """
    precio_total = cantidad * precio_unitario
    item = {

```

(continué en la próxima página)

(proviene de la página anterior)

```

        'producto': producto,
        'precio_unitario': precio_unitario,
        'cantidad': cantidad,
        'precio_total': precio_total,
    }

    factura['items'].append(item)
    factura['total'] += precio_total

def crear_factura():
    """ Crear una factura """
    factura = {
        'total': 0,
        'items': [] # lista de items
    }

    return factura

mi_factura = crear_factura()
agregar_item(mi_factura, 'Alfajor', 150, 3)
agregar_item(mi_factura, 'Turrón', 53)
agregar_item(mi_factura, 'Turrón', 53)

# -----
#>---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
#>---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
#>completado.
# La instruccion "assert" de Python lanzará un error si lo que se
#>indica a
#   continuacion es falso.
# Si usas GitHub (o similares) puedes hacer una nueva rama con esta
#>solución,
#   crear un "pull request" y solicitar revision de un tercero.

assert mi_factura['total'] == 556

items_en_factura = mi_factura['items']
assert len(items_en_factura) == 2

print('Ejercicio terminado OK')
```

- Hacer un PR con una propuesta de solución para el ejercicio 041

```

"""
La funcion "crear_mazo_cartas_espaniolas" funciona casi bien.
Por algun motivo faltan algunas cartas.
La tarea de este ejercicio es reparar esta función para que el mazo_
→este completo
"""

def crear_mazo_cartas_espaniolas():
    palos = ['oro', 'copa', 'espada', 'basto']
    mazo = []
    for n in range(1, 12):
        for palo in palos:
            carta = {'numero': n, 'palo': palo}
            mazo.append(carta)
    return mazo

mazo_esp = crear_mazo_cartas_espaniolas()
print(mazo_esp)

# -----
# ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# ---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera_
→completado.
# La instruccion "assert" de Python lanzará un error si lo que se_
→indica a
#   continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta_
→solución,
#   crear un "pull request" y solicitar revision de un tercero.

assert {'numero': 10, 'palo': 'oro'} in mazo_esp
assert {'numero': 11, 'palo': 'oro'} in mazo_esp
assert {'numero': 12, 'palo': 'oro'} in mazo_esp

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el ejercicio 042

```

"""

La funcion "crear_mazo_cartas_poker" esta incompleta y necesita ser_

```

(continué en la próxima página)

(proviene de la página anterior)

→completada para
devolver una lista de diccionarios con todas las cartas disponibles en
→un mazo de poker.

Nota: El ejercicio 041* ya muestra una función similar que puede
→usarse como ayuda

* <https://github.com/avdata99/programacion-para-no-programadores/blob/master/ejercicios/ejercicio-041/ejercicio.py>

"""

```
def crear_mazo_cartas_poker():
    palos = ['pica', 'trebol', 'corazon', 'diamante']
    mazo = []

    # COMPLETAR la lista con un diccionario por cada carta de
    # la forma {'numero': X, 'palo': Y}
    # El test de la parte inferior de este archivo ayuda a validar
    # el resultado esperado

    return mazo

# -----
# ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# ---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
#   continuacion es falso.
# Si usas GitHub (o similares) puedes hacer una nueva rama con esta
# solución,
#   crear un "pull request" y solicitar revision de un tercero.

mazo_poker = crear_mazo_cartas_poker()

assert {'numero': 9, 'palo': 'pica'} in mazo_poker
assert {'numero': 10, 'palo': 'pica'} in mazo_poker
assert {'numero': 'J', 'palo': 'pica'} in mazo_poker
assert {'numero': 'Q', 'palo': 'pica'} in mazo_poker
assert {'numero': 'K', 'palo': 'pica'} in mazo_poker
```

(continué en la próxima página)

(proviene de la página anterior)

```
assert {'numero': 9, 'palo': 'diamante'} in mazo_poker
assert {'numero': 10, 'palo': 'diamante'} in mazo_poker
assert {'numero': 'J', 'palo': 'diamante'} in mazo_poker
assert {'numero': 'Q', 'palo': 'diamante'} in mazo_poker
assert {'numero': 'K', 'palo': 'diamante'} in mazo_poker

print('Ejercicio terminado OK')
```

17.2 Algunos ejemplos de uso

```
data = {
    'edad': 32,
    'nombre': 'Juan'
}

nombre = data['nombre'] # equivalente a data.get(nombre)
edad = data['edad']
print(f'Nombre: {nombre}, edad {edad} años')

"""
data['algo-que-no-existe']
# genera un error
# KeyError: 'algo-que-no-existe'
# mientras que
data.get('algo-que-no-existe')
devuelve None
"""

# Nombre: Juan, edad 32 años

for k, v in data.items():
    print(f'Item encontrado: Key:{k}, Value: {v}')

# Item encontrado: Key:edad, Value: 32
# Item encontrado: Key:nombre, Value: Juan
```

```
data = {
    'edad': 32,
    'nombre': 'Juan',
    'educacion': {
        'secundario': 'Montserrat',
```

(continué en la próxima página)

(proviene de la página anterior)

```

        'universidad': 'UNC',
    }
}

nombre = data['nombre']
edad = data['edad']
universidad = data['educacion']['universidad']

print(f'Nombre: {nombre}, edad {edad} años. Universidad: {universidad}
→')
# Nombre: Juan, edad 32 años. Universidad: UNC

# Agregarle datos
data['ocupacion'] = 'Desarrollador'
data['educacion']['primario'] = 'San Juan'

print(data)
# {'edad': 32, 'nombre': 'Juan', 'educacion': {'secundario': 'Monserrat',
→'universidad': 'UNC', 'primario': 'San Juan'}, 'ocupacion': 'Desarrollador'}

print(data.get('ocupacion'))
# 'Desarrollador'

print(data.get('NO EXISTE'))
# None

print(data.get('NO EXISTE', 'valor predeterminado'))
# valor predeterminado

```

```

# lista de diccionarios
data = {
    'personas': [
        {'nombre': 'Juan', 'edad': 20},
        {'nombre': 'Pedro', 'edad': 30},
        {'nombre': 'María', 'edad': 40},
    ]
}

print('PERSONAS:')
for persona in data['personas']:
    nombre = persona['nombre']
    edad = persona['edad']
    print(f' - Nombre: {nombre}, edad {edad} años')

""" resultados

```

(continué en la próxima página)

(proviene de la página anterior)

PERSONAS:

- *Nombre: Juan, edad 20 años*
- *Nombre: Pedro, edad 30 años*
- *Nombre: María, edad 40 años*

""""

CHAPTER 18

Librerías incluida: random

Así como hay funciones incluidas (*built-ins*) que se pueden usar sin estar declaradas también hay más herramientas de Python disponibles pero que requieren ser *importadas*.

Importar (con el comando `import` o de la forma `from X import Y`) en Python es disponibilizar nuevas herramientas (funciones y otras) en nuestro código.

18.1 Python al azar: random

El módulo `random` incluye una serie de funciones que permiten darle aleatoriedad a nuestro código.

Veamos un ejemplo. La función `randint` genera números al azar entre dos valores pasados como parámetros.

```
from random import randint

al_azar = randint(1, 100)
print(al_azar)

# otra forma de usarlo podría ser
import random
al_azar = random.randint(1, 100)
print(al_azar)
```

Así como el `.` se usa en `objeto.propiedad` también se usa en `modulo.objeto_en_modulo`.

18.2 choice: Elegir un opción al azar de una lista.

La función `choice` recibe como parámetro una lista de la cual devolverá un elemento al azar.

```
from random import choice

opciones = ["piedra", "papel", "tijeras"]
opcion_elegida = choice(opciones)
print(opcion_elegida)
# tijeras
```

18.3 shuffle: Mezclar una lista

Otra función del módulo `random` es `shuffle` que recibe una lista como parámetro y la desordena aleatoriamente.

Ejemplo:

```
from random import shuffle

mazo = [
    {"numero": 1, "palo": "espada"},
    {"numero": 2, "palo": "espada"},
    # ...
    {"numero": 11, "palo": "oro"},
    {"numero": 12, "palo": "oro"},
]
shuffle(mazo)
print(mazo[0])
# {"numero": 11, "palo": "oro"}
```

18.3.1 Tareas

- Escribir un programa que elija un numero al azar entre 1 y 100 y le pida al usuario que lo adivine. El programa debe decirle al usuario si el numero que ingresó es mayor o menor al numero que se eligió al azar. El programa debe terminar cuando el usuario adivine el numero elegido al azar.
- Hacer un PR con una propuesta de solución para el [ejercicio 045](#) (contenido en este repositorio)

```
"""
La funcion "carta_poker_al_azar" ya funciona como es esperado
La tarea aquí es completar la funcion "carta_espaniola_al_azar"
para que devuleva resultados válidos.
```

(continué en la próxima página)

(proviene de la página anterior)

```

Nota: se debe importar y usar la funcion "randint" de "random"
"""

from random import choice

def carta_poker_al_azar():
    palos = ['pica', 'trebol', 'corazon', 'diamante']
    numeros = list(range(1, 11)) + ['J', 'Q', 'K']
    nro = choice(numeros)
    palo = choice(palos)

    carta = {"numero": nro, "palo": palo}
    return carta

def carta_espaniola_al_azar():
    palos = ['oro', 'basto', 'espada', 'copa']

    # corregir estas dos lineas para devolver valores válidos
    nro = 0
    palo = ""

    carta = {"numero": nro, "palo": palo}
    return carta

# mirar ejemplos de ambas funciones
for n in range(20):
    carta1 = carta_poker_al_azar()
    carta2 = carta_espaniola_al_azar()
    print(carta1, carta2)

# -----
# ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# ---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
# continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta
# solución,
# crear un "pull request" y solicitar revision de un tercero.

```

(continué en la próxima página)

(proviene de la página anterior)

```

carta = carta_espaniola_al_azar()
assert type(carta['numero']) == int
assert carta['numero'] > 0
assert carta['numero'] <= 12
assert carta['palo'] in ['oro', 'basto', 'espada', 'copa']

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el ejercicio 046

```

"""
La siguiente funcion pretende ser usada para generar
10 numeros como resultados del sorteo de quiniela.
Como podrán notar es bastante mala.
Tarea:
- Asegurarse que los resultados sean aleatorios
- Asegurarse que la función respete los tres parametros
"""

def generar_quiniela(minimo=0, maximo=9999, total_numeros=10):
    """ Generar varios numeros al azar definidos
        entre máximo y mínimo (sin numeros duplicados) """
    return [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numeros_quiniela = generar_quiniela()
print(f'Numeros resultantes {numeros_quiniela}')

# -----
# ---
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----
# ---
# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
#   continuacion es falso.
# Si usas GitHub (o similares) podes hacer una nueva rama con esta
# solución,
#   crear un "pull request" y solicitar revision de un tercero.

# Prueba de resultados basicos
assert type(numeros_quiniela) == list

```

(continué en la próxima página)

(proviene de la página anterior)

```

assert len(numeros_quiniela) == 10
for n in numeros_quiniela:
    assert type(n) == int
    assert n >= 0
    assert n <= 9999

# Pruebas con otros parametros
minimo = 90
maximo = 200
total_numeros = 7
numeros_quiniela = generar_quiniela(minimo=minimo, maximo=maximo,
    ↪total_numeros=total_numeros)
assert type(numeros_quiniela) == list
assert len(numeros_quiniela) == total_numeros
for n in numeros_quiniela:
    assert type(n) == int
    assert n >= minimo
    assert n <= maximo

minimo = 0
maximo = 100
total_numeros = 3
numeros_quiniela = generar_quiniela(minimo=minimo, maximo=maximo,
    ↪total_numeros=total_numeros)
assert type(numeros_quiniela) == list
assert len(numeros_quiniela) == total_numeros
for n in numeros_quiniela:
    assert type(n) == int
    assert n >= minimo
    assert n <= maximo

print('Ejercicio terminado OK')

```

- Hacer un PR con una propuesta de solución para el [ejercicio 047](#)

```

"""
La funcion "detectar_escaleras" funciona bastante bien.
Toma una lista de numeros y busca escaleras ascendentes
dentro de la secuencia de numeros.
Esta funcion detecta casi todas las secuencias pero no todas
Tarea:
- Arreglar la funcion para que pase los tests y encuentre
  todas las escaleras existentes
Nota: probablemente vas a necesitar depurar el código para
entender como funciona y por que falla.
"""

```

(continué en la próxima página)

(proviene de la página anterior)

```
from random import randint

def tirar_dados(lados=6, veces=100):
    """ Generar aleatoriamente los tiros de un lado de cantidad
        de lados variable.
    """

    tiros = []
    for _ in range(veces):
        numero = randint(1, lados)
        tiros.append(numero)

    return tiros

def detectar_escaleras(lista_tiros):
    """ Detectar las escaleras ascendentes encontradas
        en los tiros de al menos 3 elementos """

    # aqui guardamos las escaleras que se consiguieron con 3 o más
    ➔ elementos
    escaleras_buenas = []
    # lista temporal para ir testeando todos los tiros
    escalera = []
    for tiro in lista_tiros:
        if len(escalera) == 0:
            # recién empiezo a buscar
            escalera.append(tiro)
        elif escalera[-1] == tiro - 1:
            # La escalera de prueba ya empezó y el número actual
            # está en escalera con el último encontrado
            escalera.append(tiro)
        else:
            # este tiro no está en escalera con el anterior
            # ver si la escalera conseguida tiene 3 o más elementos
            # para guardarle
            if len(escalera) >= 3:
                escaleras_buenas.append(escalera)
            # como este número no sirve para la escalera anterior
            # es el que inicializa la detección de la nueva
            escalera = [tiro]

    return escaleras_buenas

tiros = tirar_dados(veces=10000)
```

(continúe en la próxima página)

(proviene de la página anterior)

```

escaleras = detectar_escaleras(tiros)
print(f'Escaleras encontradas {len(escaleras)}')

# -----
# NO BORRAR O MODIFICAR LAS LINEAS QUE SIGUEN
# -----

# Una vez terminada la tarea ejecutar este archivo.
# Si se ve la leyenda 'Ejercicio terminado OK' el ejercicio se considera
# completado.
# La instruccion "assert" de Python lanzará un error si lo que se
# indica a
# continuacion es falso.
# Si usas GitHub (o similares) puedes hacer una nueva rama con esta
# solución,
# crear un "pull request" y solicitar revision de un tercero.

# Probar tiros donde yo ya conozco los resultados
tiros_test = [1, 5, 6,
              3, 4, 5,
              1, 5, 6,
              1, 2, 3,
              2, 2, 2]

escaleras = detectar_escaleras(tiros_test)
assert escaleras == [
    [3, 4, 5],
    [1, 2, 3],
]

tiros_test = [1, 5, 2, 3, 2, 1, 4, 5, 5, 1, 2, 6, 3, 3, 3, 1, 2, 5, 5,
              2, 4,
              3, 4, 5,
              2, 3, 4, 5,
              1, 5, 2, 4, 2, 5, 6, 6, 5, 5, 4, 2, 6, 1, 4, 5, 3, 1, 6,
              2, 2,
              1, 2, 3, 4, 5,
              3, 1, 2, 6, 5, 6, 2, 5, 1, 4, 4, 5, 4, 3, 2, 1, 1, 3, 5,
              6, 1,
              1, 2, 3]

escaleras = detectar_escaleras(tiros_test)
assert escaleras == [
    [3, 4, 5],

```

(continué en la próxima página)

(proviene de la página anterior)

```
[2, 3, 4, 5],  
[1, 2, 3, 4, 5],  
[1, 2, 3]  
]  
  
tiros_test = [1, 2, 3]  
escaleras = detectar_escaleras(tiros_test)  
assert escaleras == [[1, 2, 3]]  
  
print('Ejercicio terminado OK')
```

CHAPTER 19

Librerías incluida: datetime

Así como hay funciones incluidas (*built-ins*) que se pueden usar sin estar declaradas también hay más herramientas de Python disponibles pero que requieren ser *importadas*.

Importar (con el comando `import` o de la forma `from X import Y`) en Python es disponibilizar nuevas herramientas (funciones y otras) en nuestro código.

19.1 Fecha y hora: datetime

El manejo de fechas y horas básico en Python se hace con la librería `datetime`.

19.2 Fechas simples con date

Si solo necesitamos una fecha general solo con día + mes + año podemos usar objetos de tipo `date`.

```
from datetime import date

hoy = date.today()
print(hoy)
# 2022-02-12
print(type(hoy))
# <class 'datetime.date'>

# date (año, mes, día)
agosto_27_2022 = date(2022, 8, 27)
```

19.3 Variación de tiempo con timedelta

Para sumar o restar periodos de tiempo a una fecha existe timedelta:

```
from datetime import date, timedelta
hoy = date.today()
manana = hoy + timedelta(days=1)
print(manana)
# 2022-02-13
```

19.4 Fecha + hora = momento exacto con datetime

Si necesitamos más precision que solo día + mes + año debemos usar datetime.datetime

Nota: datetime como libreria incluye un objeto con el mismo nombre pero son cosas distintas. Uno es la libreria y otro es la clase para construir fecha/horas o datetimes.

```
from datetime import datetime, timedelta

ahora = datetime.now()
print(ahora)
# 2022-02-12 14:35:16.687589

print(type(ahora))
# <class 'datetime.datetime'>

en_15 = ahora + timedelta(minutes=15)
print(en_15)
# 2022-02-12 14:50:16.687589

# la diferencia entre dos fechas, es un timedelta
a = datetime(2022,3,4)
b = datetime(1988,2,11,6,7,8)
c = a - b
print(c)
# imprime 12439 days, 17:52:52
type(c)
# imprime <class 'datetime.timedelta'>
```

19.5 fecha <—> string

En muchos casos es requerido pasar de fecha a strings y viceversa. Para estos casos se usan las funciones `strftime` (fecha a string) y `strptime` (string a fecha).

```
from datetime import date, datetime

hoy = date.today()
print(hoy.strftime("%d/%m/%Y"))
# 12/02/2022
print(hoy.strftime("%Y-%m-%d"))
# 2022-02-12
print(hoy.strftime("%d de %B de %Y"))
# 12 de February de 2022

# pasar de string a fecha
fecha_str = '2022-09-21'
fecha = datetime.strptime(fecha_str, "%Y-%m-%d")
print(fecha)
# 2022-09-21 00:00:00
```

19.5.1 Tareas

- Hacer una función que le pida al usuario que inserte su fecha de nacimiento y se le devuelva hace cuantos días nació.

CHAPTER 20

Clases y objetos

En Python todo es un objeto. Por ejemplo, todas las variables para almacenar textos son *objetos* del tipo `str` (les decimos *strings*). Hemos visto ya que estos objetos tienen propiedades y funciones (`upper`, `lower`, `strip`, etc). ¿Te preguntaste dónde están definidas esas funciones? ¿Quién decide que puede y que no puede hacerse con un objeto?

La respuestas a todo son las **clases**. En Python (y en cualquier lenguaje que use *Programación Orientada a Objetos*) podemos definir nuestros propios tipos de objetos. Cuando los definimos hacemos la elección de que como funcionará y que propiedades y funciones incluirá.

20.1 Mi primera clase

Veamos un ejemplo basico. Queremos definir un tipo de dato nuevo: `Persona`.

```
1 class Persona:
2     def __init__(self, nombre, apellido):
3         self.nombre = nombre
4         self.apellido = apellido
5
6 # ----- FIN DE LA CLASE -----
7
8 # Crear nuestros objetos de tipo Persona
9 juan = Persona(nombre='juan carlos', apellido='perez')
10 pedro = Persona(nombre='pedro ', apellido='gomez')
11 luis = Persona('Luis', 'Velez ')
12
13 # Ver el tipo
```

(continué en la próxima página)

(proviene de la página anterior)

```
14 type(juan)
15 # imprime <class '__main__.Persona'>
16
17 # imprimir propiedades de nuestro objeto
18 print(f'{luis.nombre} {luis.apellido}')
19 'Luis Velez'
```

20.1.1 Anatomía de una clase

- `class` es una palabra reservada de Python (no la podemos usar como nombre de variable) que usamos para indicar que estamos definiendo una clase.
- Después de `class` agregamos el nombre de nuestra clase. Se deben cumplir las mismas reglas que para los nombres de variables (no pueden empezar con números, no pueden tener espacios, etc). Se suele usar aquí el formato *CamelCase* y siempre con mayúscula inicial. No es obligatorio.
- Después del nombre de la clase podemos colocar parentesis (no es obligatorio). Se usarán en caso de querer aprovechar algo llamado *Herencia* que veremos más adelante.
- Finalmente agregamos `:` (dos puntos) para indicar que terminamos de definir el encabezado de la clase y vamos a comenzar con el bloque de código de ella.
- El código de la clase debe estar tabulado hacia la derecha. **Esta es una de las grandes diferencias que Python tiene con los demás lenguajes de programación.** El código propio de la clase comienza tabulado y termina cuando el código vuelve a la izquierda. Muchos otros lenguajes de programación usan las llaves `{}` para delimitar donde empiezan y terminan los bloques de código.
- Dentro del código de la clase debemos definir las propiedades y funciones que queremos que tengan nuestros objetos. Podemos agregar tantas propiedades y funciones como sean necesarias. Veremos más detalles a continuación.
- `__init__` es hasta aquí nuestra única función (nos damos cuenta porque usa el `def` que ya conocemos junto a un grupo de parámetros). Cuando veamos estos guiones bajos dobles debemos interpretar que es una herramienta interna de Python. En este caso, todas las clases tienen una función de inicialización (y siempre se llama `__init__`). Ser la función inicializadora quiere decir que se va a ejecutar cuando los usuarios de la clase la usen para construir un objeto de este tipo.
- `self` es el primer parámetro de la función `__init__`. Hablaremos en particular de este parámetro más adelante. Por ahora podemos decir que **es obligatorio que todas las funciones de las clases tengan un parámetro que represente a cada objeto creado con esta clase.** Es por esto que veremos a `self` en (*casi*, hay situaciones especiales) todas las funciones de una clase.
- Una vez definida la clase (y terminada anulando la tabulación y volviendo a la izquierda el código), esta se puede llamar simplemente con su nombre y entre paréntesis todos los parámetros que se hayan definido en `__init__` (después de `self`).

20.1.2 Clases y objetos

Las clases se usan para definir el comportamiento de los objetos que podremos crear a partir de ellas. Al proceso de crear un nuevo objeto se le dice *instanciar* y a cada objeto se le puede llamar *instancia*. En las siguientes líneas ...

```
juan = Persona(nombre='juan carlos', apellido='perez')
pedro = Persona(nombre='pedro', apellido='gomez')
# como en las funciones, no es obligatorio nombrar los parámetros
luis = Persona('Luis', 'Velez ')
```

se crean tres *instancias* del tipo `Persona`. Cada una de ellas es un objeto distinto y por lo tanto las propiedades que contienen son independientes y se procesan de manera aislada.

```
print(juan.nombre)
'juan carlos'
print(pedro.nombre)
'pedro'
```

Podemos pensar a los objetos o *instancias* como una versión concreta de una clase.

20.1.3 ¿self?

Salvo algunas excepciones todas las funciones de las clases deben tener como primer parámetro a `self`. De esta forma, todo el objeto estará disponible dentro de cada función de la clase. Esto es obligatorio y olvidar colocarla generará errores difíciles de detectar en nuestras primeras experiencias con clases.

Cuando llamamos a funciones de la clase que usan `self`, no debemos pasar nada en lugar de este parámetro. Debemos ignorarlo cuando estamos usando nuestro objeto. Esto es visible en todos los ejemplos usados en este manual.

20.1.4 Contenido de un clase

Dentro de la clase podemos definir las propiedades y funciones que nuestros objetos tendrán cuando sean instanciados.

Cuando escribimos `self.PROPIEDAD = VALOR` estamos indicando que los usuarios podrán usar estas propiedades en los objetos definidos.

Estas líneas ...

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

indican que todos los objetos de tipo `Persona` tendrán disponibles las propiedades `nombre` y `apellido`. Además el valor de ellas será inicializado con los parámetros que usuario pase al construir estos objetos. Al no tener valores predeterminados (funciona igual que las funciones) ambas propiedades son **obligatorias**. Si intentamos crear una `Persona` con algo como: `juan = Persona(nombre='juan carlos')` obtendremos un error, el `apellido` es obligatorio.

Así como estas propiedades están definidas además de poder ser leídas como hemos visto, también pueden modificarse libremente.

```
diego = Persona('Diego', 'Algun apellido')
diego.apellido = 'Solis'
print(diego.apellido)
'Solis'
```

Esta forma de definir y usar las propiedades en nuestros objetos es poco segura. En nuestra clase, el siguiente comportamiento si está permitido:

```
diego = Persona('Diego', 'Algun apellido')
diego.apellido = 17 # sería bueno evitar esto
print(diego.apellido)
17
```

20.1.5 Tareas

- Crear una clase llamada `Auto` que se inicialice con algunas propiedades obligatorias y otras opcionales

CHAPTER 21

Propiedades controladas

Es posible tomar los valores que el usuario nos pasa al inicializar los objetos (en `__init__`) y guardarlos en variables privadas. Usamos el guión bajo inicial en la propiedades para indicar que estas son internas de la clase y que no deberían ser usadas. Esto es una convención pero no es obligatorio. Python no lanzará un error si los usuarios de nuestra clase usan estas propiedades privadas.

```
1 class Persona:
2     def __init__(self, nombre, apellido):
3         self._nombre = nombre
4         self._apellido = apellido
5
6     @property
7     def nombre(self):
8         return self._nombre
9
10    @property
11    def apellido(self):
12        return self._apellido
```

21.1 ¿Que es @property?

Antes de la definición de una función (esto puede hacerse dentro y fuera de las clases) es posible agregar lo que llamamos un decorador (o función decoradora). La sintaxis para hacerlo es simplemente agregar esta línea sobre la definición de una función y comenzarla con un `@`. Estos decoradores se usan para modificar a la función de formas que por el momento exceden lo que necesitamos conocer. En particular, `@property` es usado por las clases en Python para marcar

que una propiedad **existe** y que la función *decorada* será la encargada de atender las llamadas de **lectura** de esta propiedad. La escritura/modificación de cada propiedad se hace de otra forma.

Hasta aquí estas propiedades son *solo lectura*

```
1 victor = Persona('Victor', 'Fernandez')
2 print(victor.apellido)
3 # funciona y devuelve
4 'Fernandez'
5 # La siguiente línea fallará porque no está todavía definido como...
  ↳ funcionará
6 # la asignación de esta propiedad
7 victor.apellido = 'Gonzalez'
```

A las funciones de una clase para **leer** una propiedad se les llama **getter** y las funciones para escribir un nuevo valor a una propiedad se las llama **setter** (por *get* y *set* del inglés: *obtener* y *definir*).

Formalmente ahora podemos decir que nuestras propiedades **nombre** y **apellido** tienen **getter** pero no **setter**.

Veamos un ejemplo de **setter** para la propiedad **nombre** de la clase **Persona**:

```
1 @nombre.setter
2 def nombre(self, value):
3     # Antes de escribir mi variable privada _nombre, revisar que
4     # cumpla con los requisitos definidos
5     if type(value) != str:
6         # Si no es del tipo *string* lanzaremos (raise) un error
7         # (excepción) del Tipo Exception (hay otros tipos).
8         raise Exception('Nombre inválido. Solo string permitido')
9
10    # solo si pasa las validaciones (podrían ser varias)
11    # sobrescribimos nuestra variable privada con el nuevo valor.
12    self._nombre = value
```

La función definida para ser **setter** debe cumplir las siguientes condiciones:

- Tener un decorador de la forma `@NOMBRE_DE_LA_PROPIEDAD.setter`.
- Tener el mismo nombre que la función **getter**.
- Incluir un parámetro para recibir el valor que el usuario quiere definir (usualmente lo llamaremos **value**).

Es posible también definir propiedades personalizadas a gusto.

```
1 @property
2 def nombre_completo(self):
3     """ devuelve el nombre completo """
4     return f'{self._nombre} {self._apellido}'
```

(continúe en la próxima página)

(proviene de la página anterior)

```
5
6 @property
7 def nombre_formal(self):
8     """ devuelve el nombre completo """
9     return f'{self._apellido}, {self._nombre}'
```

Estas propiedades solo tienen sentido para ser leídas. Es por esto que no tienen una función setter.

Ejemplos de uso:

```
juan = Persona('juan carlos', 'perez')
print(juan.nombre_completo)
'juan carlos perez'
print(juan.nombre_formal)
'perez, juan carlos'
# Si intento asignar una propiedad que es solo lectura (no tienen una
→funcion setter)
# dará un error "can't set attribute" (no se puede asignar esta
→propiedad)
juan.nombre_completo = 'Nuevo nombre completo'
```

Las propiedades nombre y apellido se pueden leer y escribir. Las propiedades nombre_completo y nombre_formal son simplemente combinaciones útiles de otras propiedades básicas. Solo se puede leer.

CHAPTER 22

Funciones de mi clase

Es también posible definir funciones

```
def limpiar(self):
    """ Mejorar el nombre y el apellido """
    self._nombre = self._nombre.strip().title()
    self._apellido = self._apellido.strip().title()

def encabezado(self, titulo, limpiar=True):
    """ Genera y devuelve el nombre completo con
        "Sr." "Sra." o algun otro titulo.
        Opcionalmente se puede limpiar el nombre """
    # limpiar el nombre si se solicita
    if limpiar:
        self.limpiar()
    return f'{titulo} {self.nombre_completo}'

# podemos tambien emular el comportamiento de los strings
# e incluso copiar nombres de funciones de ellos
def lower(self):
    """ devuelve el nombre completo en minusculas """
    return self.nombre_completo.lower()

def upper(self):
    """ devuelve el nombre completo en minusculas """
    return self.nombre_completo.upper()
```

Algunos ejemplos de uso con estas nuevas funciones:

```
juan = Persona('juan carlos', 'perez')
print(juan.nombre_completo)
# 'juan carlos perez'
print(juan.nombre_formal)
# 'perez, juan carlos'

enc = juan.encabezado('Sr.', limpiar=False)
print(enc)
# 'Sr. juan carlos perez'

enc = juan.encabezado('Sr.')
print(enc)
# 'Sr. Juan Carlos Perez'

print(juan.nombre)
# El nombre fue limpiado
# 'Juan Carlos'

print(juan.upper())
# 'JUAN CARLOS PEREZ'
```

Nota importante: Las funciones se llaman con los parentesis (y parámetros si se requieren) y las propiedades se llaman sin ellos (y no puede requerir parámetros).

Código de la clase final [aquí](#).

```
class Persona:
    def __init__(self, nombre, apellido):
        self._nombre = nombre
        self._apellido = apellido

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, value):
        if type(value) != str:
            raise Exception('Nombre inválido. Solo string permitido')
        self._nombre = value

    @property
    def apellido(self):
        return self._nombre

    @nombre.setter
```

(continué en la próxima página)

(proviene de la página anterior)

```
def apellido(self, value):
    if type(value) != str:
        raise Exception('Apellido inválido. Solo string permitido')
    self._nombre = value

@property
def nombre_completo(self):
    """ devuelve el nombre completo """
    return f'{self._nombre} {self._apellido}'

@property
def nombre_formal(self):
    """ devuelve el nombre completo en modo formal """
    return f'{self._apellido}, {self._nombre}'

def limpiar(self):
    """ Mejorar el nombre y el apellido """
    self._nombre = self._nombre.strip().title()
    self._apellido = self._apellido.strip().title()

def encabezado(self, titulo, limpiar=True):
    """ Genera y devuelve el nombre completo con
        "Sr." "Sra." o algun otro titulo.
        Opcionalmente se puede limpiar el nombre """
    # limpiar el nombre si se solicita
    if limpiar:
        self.limpiar()
    return f'{titulo} {self.nombre_completo}'

# podemos tambien emular el comportamiento de los strings
# e incluso copiar nombres de funciones de ellos
def lower(self):
    """ devuelve el nombre completo en minusculas """
    return self.nombre_completo.lower()

def upper(self):
    """ devuelve el nombre completo en minusculas """
    return self.nombre_completo.upper()
```

22.1 Tareas

- Hacer un PR a la [clase Persona](#) para agregar la propiedad edad.
- Hacer un PR a la [clase Carta](#) para validar que el palo es string en su función `setter`.
- Hacer un PR a la [clase Carta](#) para validar que el numero es mayor que cero y menor o igual que 12 en su función `setter`.

22.2 Algunos ejemplos de uso

```
"""
Clase Carta para juegos de cartas
"""

class Carta:
    def __init__(self, numero, palo):
        self._numero = numero
        self._palo = palo

    @property
    def numero(self):
        return self._numero

    @numero.setter
    def numero(self, value):
        if type(value) != int:
            raise Exception('Solo están permitidos numeros')
        self._numero = value

    @property
    def palo(self):
        return self._palo

    @palo.setter
    def palo(self, value):
        self._palo = value

    def __str__(self):
        return f'{self.numero} de {self.palo}'

# Pruebas

carta1 = Carta(3, 'espada')
print(str(carta1))
'3 de espada'
```

CHAPTER 23

Funciones especiales de las clases

También conocidos como *métodos mágicos* (podemos pensar a la palabra método como sinónimo de *función*) estas funciones se aplican a situaciones habituales de otros objetos de Python. Estas situaciones son la suma, resta, división, comparación, etc.

23.1 `__add__`

Que pasara si quisiéramos sumar dos Personas tal como las vimos en la clase anterior:

```
juan = Persona('Juan', 'Perez')
victor = Persona('Victor', 'Gutierrez')

a = juan + victor
```

Este código daría un error porque no está definida la función especial (o mágica) `__add__`. No está definida porque esta suma no tendría sentido en estos objetos. Veamos un ejemplo donde sí pudiera ser útil.

```
class FacturaServicio:
    """ Cada factura para el pago de servicios hogareños """
    def __init__(self, monto, servicio):
        self.monto = monto
        self.servicio = servicio

    def __add__(self, otro):
        """ Sumar esta factura a otra factura
            Notar que el resultado no es otro objeto de este tipo,
            es solo un numero. """
```

(continué en la próxima página)

(proviene de la página anterior)

```
if type(otro) != FacturaServicio:
    raise Exception('La suma solo está permitida para objetos_
↪del mismo tipo')

return self.monto + otro.monto
```

La función especial `__add__` debe incluir un parámetro después de `self` en el que recibiremos cualquiera sea el objeto al que debemos sumarnos.

Veamos este código en acción:

```
f1 = FacturaServicio(3500.90, 'Internet')
f2 = FacturaServicio(1806.06, 'Telefono')

print(f1 + f2)
5306.96
```

La función `__add__` devuelve un número pero podría haber casos donde se devuelvan otros tipos de datos. Muchas veces podemos esperar que dos objetos del mismo tipo al sumarse devuelvan un nuevo objeto de ese tipo pero no es siempre el caso. Esto puede definirse a gusto. De la misma forma, podríamos sumar nuestros objetos con lo de otra clase. Nosotros lo hemos bloqueado (lanzando un error) pero podríamos hacerlo si fuera necesario. Incluso podríamos devolver resultados distintos por cada tipo de objeto al que sumamos nuestro objeto.

23.2 `__str__`

Es probablemente la función especial más usada. Se usa para definir qué texto se va a devolver cuando el usuario necesite una representación *string* de este objeto.

```
def __str__(self):
    return f'$ {self.monto} a pagar por el servicio de {self.servicio}'
```

Ejemplo de uso:

```
f1 = FacturaServicio(3500.90, 'Internet')
f1_str = str(f1)
print(f1_str)

# o directamente cuando se quiere imprimir nuestro objeto
f1 = FacturaServicio(3500.90, 'Internet')
print(f1)
```

23.3 `__eq__`

Si queremos permitir la comparación de objetos de nuestra clase se puede definir esta función. Esta función será llamada cuando nuestro objeto sea comparado con otro mediante el operador `==`. Al igual que `__add__`, podríamos comparar nuestro objetos con lo de otra clase si fuera necesario (este no es el caso).

```
def __eq__(self, otro):
    """ Revisar si son iguales a otra factura """

    if type(otro) != FacturaServicio:
        raise Exception('La comparacion solo está permitida para
↪objetos del mismo tipo')

    montos_iguales = self.monto == otro.monto
    servicios_iguales = self.servicio == otro.servicio

    return montos_iguales and servicios_iguales
```

Veamoslo en acción:

```
f1 = FacturaServicio(1500.90, 'Internet')
f2 = FacturaServicio(1500.90, 'Internet')
f3 = FacturaServicio(3500.90, 'Internet')

if f1 == f2:
    print('f1 y f2 SI son iguales')
else:
    print('f1 y f2 NO son iguales')

if f2 == f3:
    print('f2 y f3 SI son iguales')
else:
    print('f2 y f3 NO son iguales')

"""
f1 y f2 SI son iguales
f2 y f3 NO son iguales
"""
```


CHAPTER 24

Código final de nuestra clase

Disponible aquí.

```
class FacturaServicio:
    """ Cada factura para el pago de servicios hogareños """
    def __init__(self, monto, servicio):
        self.monto = monto
        self.servicio = servicio

    def __add__(self, otro):
        """ Sumar esta factura a otra factura
            Notar que el resultado no es otro objeto de este tipo,
            es solo un numero. """
        if type(otro) != FacturaServicio:
            raise Exception('La suma solo está permitida para objetos,
→del mismo tipo')

        return self.monto + otro.monto

    def __str__(self):
        return f'$ {self.monto} a pagar por el servicio de {self.
→servicio}'

    def __eq__(self, otro):
        """ Revisar si son iguales a otra factura """

        if type(otro) != FacturaServicio:
            raise Exception('La comparacion solo está permitida para,
→objetos del mismo tipo')
```

(continué en la próxima página)

(proviene de la página anterior)

```
montos_iguales = self.monto == otro.monto
servicios_iguales = self.servicio == otro.servicio

return montos_iguales and servicios_iguales
```


CHAPTER 25

Otras funciones especiales

Estas algunas otras de las funciones especiales.

- `__mul__`: Multiplicación
- `__sub__`: Resta (*Substraction*)
- Para que nuestros objetos se comporten como diccionarios
- `__getitem__`: Obtener un item con la clave que se pasa como parámetro
- `__setitem__`: Definir un item con la clave y el valor que se pasan como parámetro
- `__delitem__`: Eliminar el item que tiene la clave que se pasa como parámetro
- `__ne__`: No igual (*Not equal*) `!=`
- `__lt__`: Menor que (*less than*) `<`
- `__gt__`: Mayor que (*grater than*) `>`
- `__neg__`: Negativo (para cuando usan `-MY-OBJETO`)

Y hay muchas más.

25.1 Tareas

- Hacer un PR a la `clase Carta` para agregar la función `__add__` para que devuelva un `int` calculando el envido **solo** de esas dos cartas. **Incluir multiples asserts al final que pruebe al menos tres sumas (diferentes y variadas) y sus resultados (envidos) esperados.**
- Hacer un PR a la `clase Carta` para agregar la función `__eq__` para que devuelva `True` solo cuando el numero y el palo sean iguales.

25.2 Algunos ejemplos de uso

```

"""
Ejemplo de una clase para manejar fracciones
(de numerador y denominador entero y positivo)
"""

class Fraccion:
    """ Clase para manejar fracciones de numeros enteros positivos """

    def __init__(self, numerador, denominador):
        if type(numerador) != int or type(denominador) != int:
            raise ValueError('Solo numeros enteros aceptados')
        if numerador < 0 or denominador < 1:
            raise ValueError('Solo numeros positivos aceptados')

        self._numerador = numerador
        self._denominador = denominador
        self._simplificar()

    @property
    def numerador(self):
        return self._numerador

    @numerador.setter
    def numerador(self, num):
        if type(num) != int:
            raise ValueError('Tipo de dato no admitido para numerador')
        if num < 1:
            raise ValueError('Valor de numerador no admitido')
        self._numerador = num
        self._simplificar()

    @property
    def denominador(self):
        return self._denominador

    @denominador.setter
    def denominador(self, den):
        if type(den) != int:
            raise ValueError('Tipo de dato no admitido para denominador
→ ')
        if den <= 0:
            raise ValueError('Valor de denominador no admitido')
        self._denominador = den

```

(continué en la próxima página)

(proviene de la página anterior)

```

    self._simplificar()

    def _simplificar(self):
        """ Simplificar la fraccion a los numeros mas bajos posibles """
        ↪ "
        men = min(self._numerador, self._denominador)
        for n in range(men, 1, -1):
            # Si este numero divide a los dos, entonces los divido
            if self._numerador % n == 0 and self._denominador % n == 0:
                self._numerador = int(self._numerador / n)
                self._denominador = int(self._denominador / n)
                break

    def __add__(self, otro):
        """ Sumar fracciones """
        if type(otro) != Fraccion:
            raise ValueError('Solo suma aceptada entre fracciones')

        nuevo_denominador = self._denominador * otro.denominador
        nuevo_numerador = self._numerador * otro.denominador + otro.
        ↪ numerador * self._denominador

        return Fraccion(nuevo_numerador, nuevo_denominador)

    def __eq__(self, otro):
        if type(otro) != Fraccion:
            raise ValueError('No son objetos iguales')

        return self._numerador == otro.numerador and self._denominador_
        ↪ == otro.denominador

    def __str__(self):
        return f'({self._numerador} / {self._denominador})'

    def __repr__(self):
        return f'<Fraccion {self._numerador} / {self._denominador}>'

# Pruebas de funcionamiento

assert Fraccion(2, 3) + Fraccion(1, 3) == Fraccion(1, 1)
assert Fraccion(4, 5) + Fraccion(3, 5) == Fraccion(7, 5)
assert Fraccion(4, 5) + Fraccion(6, 5) == Fraccion(2, 1)
assert Fraccion(5, 12) + Fraccion(4, 19) == Fraccion(143, 228)
assert Fraccion(3, 2) + Fraccion(8, 11) == Fraccion(49, 22)

```

(continué en la próxima página)

(proviene de la página anterior)

```
# Probar la simplificacion al inicio
assert Fraccion(8, 4) == Fraccion(2, 1)

f1 = Fraccion(2, 4)
assert f1 == Fraccion(1, 2)

f1.numerador = 2
assert f1 == Fraccion(1, 1), f'{f1} no es igual a {Fraccion(1, 1)}'

f1.denominador = 8
assert f1 == Fraccion(1, 8)

print('TODO OK')
```

CHAPTER 26

Paquetes y módulos

Hasta aquí hemos ejecutado nuestro código en un solo archivo.

no es exactamente así

En realidad cuando usamos algo como `from random import randint` estamos usando (importando) código que esta en otros archivos que no vemos (pero podríamos, [aquí esta el modulo interno de python random.py](#)).

En la medida que el código que hacemos crece, es necesario mantener un orden. Es por esto que conviene empaquetar el código que hacemos. Esto incluso nos permite reutilizarlo en el futuro.

reutilizar y compartir

Además de reutilizarlo nosotros lo podemos compartir abiertamente. La comunidad de Python es una de las más grandes en el desarrollo de software abierto. Al momento de escribir estas líneas, hay alrededor de 400.000 paquetes abiertos en [Pypi](#) (*The Python Package Index*). Todo este código esta disponible para nosotros.

Podemos pensar a los paquetes Python como carpetas que pueden contener más paquetes (sub-carpetas) y modulos (archivos de Python `.py`) con funciones y clases para reutilizar.

Para indicar que una carpeta es un paquete alcanza con agregarle un archivo llamado `__init__.py`. Por el momento alcanza con que este archivo este vacío.

De esta forma es posible mantener tu código ordenado y fácil de mantener.

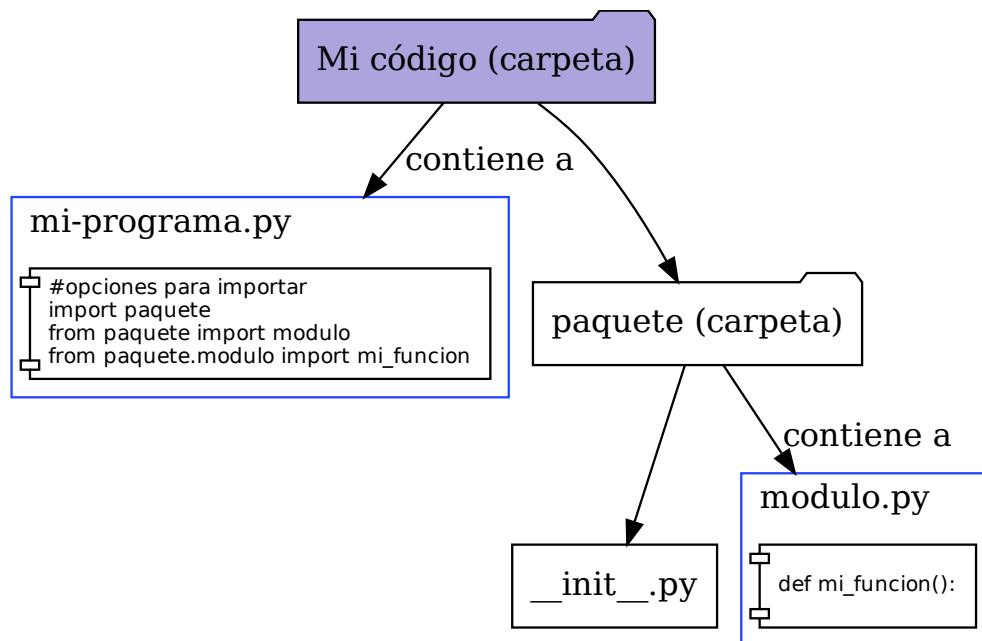


Figure1: Estructura de archivos y carpetas

Listing 1: funciones.py

```
def suma(a, b):
    return a + b
```

Listing 2: otro_archivo_en_la_misma_carpeta.py

```
import funciones
a = funciones.suma(4, 5)

from funciones import suma
a = suma(4, 5)
```

CHAPTER 27

Indices y tablas

- `genindex`
- `modindex`
- `search`