UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Proyecto Cheat Sheet

Programación / Laboratorio I

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Índice de contenido

Alcance del proyecto

La idea de este proyecto es ir construyendo clase a clase el mejor machete (cheatsheet) jamás creado para Python 3. La característica principal es que se componga de notas cortas y algún ejemplo que aclare la nota. Pueden ser pasos, fórmulas, atajos de teclado, funciones, referencias o cualquier cosa que entre todos entendamos que es lo que mejor resume el tema explicado. A continuación un ejemplo:

Variables Numéricas

Python tiene tres tipos primitivos de variables numéricas enteras, flotantes y complejas.

```
numero_entero = 15   # int
numero_negativo = -3 # int
numero_flotante = 3.14 # float
numero_complejo = 8.2 + 2j # complex
```

Operadores Relacionales

En todo los casos estos dan como resultado un booleano **True** o **False**.

Igual a	==
Diferente a	!=
Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=

Operadores Aritméticos

Suma	+	3 + 2 => 5
Resta	-	3 - 2 => 1
Multiplicación	*	3 * 2 => 6
División	/	3 / 2 => 1.5
Módulo	%	3 % 2 => 1
División entera	//	3 // 2 => 1
Potencia	**	3 ** 2 => 9

Operadores Lógicos

and	compara 2 elementos y devuelve **True** si ambos son verdadero
or	compara 2 elementos y devuelve **True** si uno de ellos es verdadero
not	devuelve el valor opuesto si es **True** devuelve **False**

Operadores Asignación

Se utiliza para asignar valores a una variable.

Asigna un valor en la variable	var = 5
var = var + 5	var += 5
var = var - 5	var -= 5
var = var * 5	var *= 5
var = var / 5	var /= 5
var = var % 5	var %= 5

type()

La función type permite comprobar el tipo de variable

```
x = 3.1415
print(type(x)) #<class 'float'>
```

print()

print al ser función siempre se utiliza con paréntesis

```
print('Hola Mundo')
```

input()

Permite obtener texto escrito por teclado.

```
nombre = input('¿Cual es tu nombre?')
print('Hola ' + nombre)
```

while

El bucle **while** (mientras) ejecuta un fragmento de código mientras se cumpla una condición.

```
respuesta = 's'
while(respuesta == 's'):
  respuesta = input("¿Desea continuar? (s/n) ")
```

for

El bucle for se utiliza para recorrer los elementos de un objeto iterable (lista, tupla, conjunto, diccionario, …) y ejecutar un bloque de código. En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable.

```
lista_numeros = [1,2,4,5,77,-1]
for numero in lista_numeros:
  print(numero)
```

```
#Salida: 1 2 4 5 77 -1
```

break

Permite terminar con la ejecución del bucle.

```
lista_numeros=[1,2,4,5,77,-1]
for numero in lista_numeros:
  if(numero==5):
     break
  print(numero)
 #Salida 1 2 4
```

continue

Permite forzar una nueva iteración del bucle.

```
lista_numeros=[1,2,4,5,77,-1]
for numero in lista_numeros:
  if(numero==5):
     continue
  print(numero,end=" ")

#Salida 1 2 4 77 -1
```

range()

Genera una secuencia de números que van desde cero por defecto hasta el número que se pasa como parámetro menos uno.

```
lista_numeros = list(range(5))
print(lista_numeros) # [0,1,2,3,4]
```

lista

Es una colección de objetos, que pueden ser de distinto tipo.

Se define entre corchetes :

```
lista = ["Texto", 22 , 3.5]
```

Es mutable y puede contener diferentes tipos de objetos.

Se puede acceder a los valores a través del índice.

```
lista[2] = 'Hola'
print(lista[2]) # Hola
```

Se agregan valores al final de la lista con .append()

```
lista.append('nuevo')
print(lista) # ["Texto", 22 , "Hola", "nuevo"]
```

El método **len()** permite conocer cuántos elementos tiene la lista

```
print(len(lista)) # 4
```

Las listas son iterables, las puedo recorrer con un **for**

```
for elemento in lista:
      print(elemento)
# Texto
# 22
# Hola
# nuevo
```

. insert()
Permite agregar un elemento a una lista pero especificando la posición de la lista para insertar el elemento

. extend()
Agrega una lista a otra lista. Extiende una lista agregando los elementos de otra lista que se pase por parámetro

. pop()
Quita un elemento de una lista y lo retorna. Si no se especifica el índice del elemento a quitar como parámetro, quita y retorna el último elemento. Si se especifica la posición del índice, quita dicho elemento y lo retorna

. remove()
Quita un elemento de una lista y NO lo retorna. A diferencia de pop, para quitar un elemento específico de la lista deberá pasarse la coincidencia de dicho elemento y no el índice del mismo. También, remueve solamente la PRIMER coincidencia que encuentre (si hay más elementos que coincidan, no se eliminarán).

. enumerate()
Recorre una lista que se le pase por parámetro y devuelve dos elementos: el índice y el elemento.

```
for indice,elemento in enumerate(lista_numeros):
      print(indice, elemento):
```

. **zip()**: Itera varias listas en paralelo.

```
for elem_lista_a, elem_lista_b in zip(lista_a, lista_b):
      print(elem_lista_a, elem_lista_b):
```

. map()
Recorre una lista y permite realizar una determinada acción por cada elemento. Este método retorna un objeto map, pero es posible castearlo a list() para poder tener nuevamente una lista. Por ejemplo, recorrer una lista con strings para capitalizar cada elemento:

```
lista_capitalize = list(map(lambda elem : elem.capitalize(), lista))
```

. filter()
Función que retorna una lista con los elementos que cumplan con la coincidencia que se pase por parámetro. El primer parámetro es la condición para buscar una coincidencia de los elementos de la lista (retorna valor booleano), y como segundo parámetro se pasa la lista a evaluar

.reduce()
Suele ser utilizado con funciones que llevan a cabo un cálculo acumulativo por cada elemento que recorre. Se utiliza la función lambda donde deberá tener dos parámetros: uno para acumular la suma y otro que será el elemento actual que estará recorriendo.

```
lista = [17, 71, 18]
lista_resultado = reduce(lambda x, y : x + y, lista)
```

* Requiere el módulo "functools"
* En la función lambda: "x" parámetro que estará acumulando / "y": parámetro del elemento actual de la lista
* Tiene un tercer parámetro (opcional) donde se puede especificar a partir de qué valor comenzar a reducir:

```
lista = [17, 71, 18]
# Comenzar a contar a partir de valor 10:
print(reduce(lambda x, y : x + y, lista, 10)) # 116
```

. shuffle()
Función que mezcla una lista (que se pase por parámetro) cambiando las posiciones de los elementos de forma random

* Requiere el módulo "random"
* Modifica la lista original

. sort()
Función que ordena de menor a mayor una lista.

* Para realizarlo al revés, puede pasarse por parámetro "reverse = True" para invertirla.
* Para ordenar bajo un criterio específico, puede especificarse el parámetro "key = ..." con la función lambda/criterio necesario
* Modifica la lista original

. reverse()
Función que invierte el orden de una lista.

```
lista = [1, 2, 3, 4, 6, 1, 5, 4, 7]
lista.reverse()
print(lista) # [7, 4, 5, 1, 6, 4, 3, 2, 1]
```

tupla

Las tuplas son similares a las listas, simplemente que una vez definida la tupla, NO puede ser modificada ya que no es mutable. Una lista se puede catear a tupla y una tupla se puede castear a lista.

Declarar una tupla

```
tupla = (1,2,3)
print(type(tupla)) # <class 'tuple'>
```

Convertir tupla en lista

```
lista = list(tupla)
print(type(lista)) # <class 'list'>
```

Convertir lista a tupla

```
nueva_tupla = tuple(lista)
print(type(nueva_tupla)) # <class 'tuple'>
```

diccionarios

Son una estructura de datos que almacena valores utilizando otros como referencia para su acceso y almacenamiento, es iterable, mutable y puede contener elementos de diferente tipo; se declara entre llaves {clave:valor}

```
dic_test = {"a": 22, "b": "Hola", "c": 3}
print(dic_test) # {'a': 22, 'b': 'Hola', 'c': 3}
```

Acceder a un elemento utilizamos el índice

```
print(dic_test['a']) # 22
```

Modificar un valor

```
dic_test['a'] = 28
print(dic_test)
```

Nuevos elementos añadimos una clave no existente

```
dic_test['j'] = 4
print(dic_test) # {'a': 28, 'b': 'Hola', 'c': 3, 'j': 4}
```

funciones

Una función es simplemente un "fragmento" de código que se puede usar una y otra vez, en lugar de escribirlo varias veces.

El uso de **def** nos permite crear una función para que la función devuelva uno o varios valores, debemos usar **return**.

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

Retorno: Una función puede o no retornar un resultado. Cuando una función, haga un retorno de datos, éstos, pueden ser asignados a una variable:

```
suma = funcion_suma(33,1)
print("La suma es", suma)
```

Parámetro: Es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def funcion_suma(primer_parametro, segundo_parametro):
    return primer_parametro + segundo_parametro
```

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local. Es decir, que los parámetros serán variables locales, a las cuáles sólo la función podrá acceder.

Parámetros por defecto: Es posible asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def funcion_suma(primer_parametro, segundo_parametro=1):
    return primer_parametro + segundo_parametro

suma = funcion_suma(33)
print("La suma es", suma) # La suma es 34
```

Documentar: Es de gran utilidad explicar que hace la función, que parámetros recibe y que retorna. También es posible documentar tanto los tipos de los parámetros que espera recibir una función como el tipo de valor que retorna.

```
def resta(variable_a:int, variable_b:int=5) -> int:
    '''
    Indicar que hace
    Qué parámetros acepta
    Qué devuelve
    '''
    return variable_a-variable_b

print(resta(variable_a=15)) # 10
```

Scope: Tanto los parámetros como las variables definidas dentro de la función solo son accesibles dentro de esa función.

```
def resta(variable_a:int, variable_b:int=5) -> int:
    int resultado
    resultado = variable_a-variable_b
    return resultado
```

```
print(resta(variable_a=15)) # 10
```

Fuera de la función la variable “resultado” no está declarada.

string

Los cadenas (o strings) son un tipo de datos compuestos por secuencias de caracteres que representan texto. Estas cadenas de texto son de tipo str y se delimitan mediante el uso de comillas simples o dobles.

strip: elimina caracteres en blanco

variable.strip()

lower: convierte a las letras a minúscula

variable.lower()

upper: convierte la primer letra en mayúscula

variable.upper()

replace: buscar y reemplazar

variable.replace("casa", "CASAS")

-Parametro1: ingreso el valor que quiero buscar

-Parametro2: ingreso el valor que reemplazará al valor buscado

split: Este método se encarga de separar la cadena utilizando como separador el parámetro ingresado. Este método NO DEVUELVE un STRING, sino que devuelve una lista con los valores.

lista = variable.split("***separador***")

join: Permite concatenar en un nuevo string los elementos de una lista pasada como parámetro, usando como separador el valor contenido en la cadena a la que se le ejecuta el método join.

variable = “+”
variable.join(['A','B','C']) --> devuelve "A+B+C"

Slice: Otra operación que podemos realizar a una cadena es seleccionar solamente una parte de ella. Para ello se usa la notación [inicio:fin:paso] también en el nombre de la variable que almacena la cadena, donde:

- Inicio: es el índice del primer carácter de la porción de la cadena que queremos seleccionar.
- Fin: es el índice del último carácter no incluido de la porción de la cadena que queremos seleccionar.
- Paso: indica cada cuantos caracteres seleccionamos entre las posiciones de inicio y fin.

zfill : agrega y rellena con 0 hasta completar el número de caracteres ingresado en el parámetro(se debe ingresar un entero)

variable.zfill(parámetro)--> variable = "06"
variable.zfill(5) # "00006"

isalpha: devuelve un booleano, evalúa si todos los caracteres son alfabéticos
variable.isalpha()

isalnum: devuelve un booleano, evalúa si todos los caracteres son alfanuméricos
variable.isalnum()

count: devuelve la cantidad de ocurrencias que aparecen !!!Devuelve un INT¡¡¡¡

variable.count("lo_que_quiero_contar")

regExp

Las expresiones regulares, a menudo llamada también regex, son unas secuencias de caracteres que forman un patrón de búsqueda, las cuales son formalizadas por medio de una sintaxis específica.

```
import re
```

Métodos para buscar coincidencias

match(): Determinada si la regex tiene coincidencias en el comienzo del texto.

search(): Escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.

findall(): Encuentra todos los subtextos donde haya una coincidencia y nos devuelve estas coincidencias como una lista.

Sintaxis de expresiones:

Los corchetes [] permiten describir un conjunto de caracteres a ha aceptar o en el caso de comenzar con el acento circunflejo [^] describira el conjunto de caracteres a no ser incluidos.

`[a-z]`	Letra de la a hasta la z en minúscula
`[A-Z]`	Letra de la A hasta la Z en mayúscula
`[a-zA-Z]`	De la a-z en minúscula o de la A-Z en mayúscula
`[0-9]`	Número del 0 al 9
`[ _0-9 ]`	Espacio (" "), guión bajo ("_") o número del 0-9
`[.]`	Cualquier carácter
`[^,]`	Cualquier carácter excepto la coma
`"^Hola"`	Empieza por Hola
`"Chau$"`	Termina por Chau

Cantidad de ocurrencias

Estos modificadores indican la cantidad de veces que puede aparecer el carácter o junto de caracteres anterior a donde aparece el modificador.

`[a-z]*`	Cero o más, similar a {0,}
`[a-z]+`	Una o más, similar a {1,}
`[a-z]?`	Cero o una, similar a {0,1}
`[a-z]{n}`	Exactamente **n** veces
`[a-z]{n,}`	Por lo menos **n** veces.
`[a-z]{n,m}`	Por lo menos **n** pero no más de **m** veces.

Escape de caracteres

Si necesitamos buscar un carácter que el sistema RegEX utiliza para sus expresiones (como pueden ser los siguientes: ".", "?", "|", "*", etc), debemos escapar los caracteres con "****"

raw strings

En el caso de requerir utilizar **** de manera literal y no como un carácter de escape, se puede hacer uso de las raw strings. Una cadena de este tipo comienza anteponiendo el carácter r a las comillas (simples o dobles).

```
r" \Esto es una cadena cruda\"
```

quick sort

El ordenamiento rápido (quicksort en inglés) es un algoritmo de ordenamiento creado por el científico británico en computación C. A. R. Hoare.

Pasos:

1. Generamos una copia de la lista original y creamos dos listas vacías (izquierda y derecha
2. Si el tamaño de la lista recibida por la función es menor o igual a 1 se retorna la lista.
3. Si no se realizan los siguientes pasos.
    a. - Se genera un pivote al azar de la lista. (puede ser el primer elemento)
    b. - Se recorre la lista desde el elemento 1 (próximo al pivote) hasta el final.
    c. - Si el elemento es menor al pivote se appendea a la lista izquierda, si es mayor a la lista derecha (ordenamiento ascendente)
4. Luego ordenamos ambas listas llamando a la misma función (RECURSIVIDAD)
5. Finalmente retornamos la lista izquierda + pivote + la lista derecha
6. El resultado debería ser la lista ordenada de forma ascendente.

archivos

Los archivos de texto contienen caracteres legibles, es posible no solo leer dicho contenido sino también modificarlo usando un editor de texto.

Modos de apertura de un archivo:

`r`	Abre el archivo sólo para lectura
`w`	Abre un archivo en modo escritura. Si no existe el archivo, lo crea. Si existe, lo sobreescribe completamente
`a`	Abre un archivo en modo escritura, pero no sobrescribe el archivo, sino que le agrega nueva información

Para abrir un archivo con Python, podemos usar la función open.

```
archivo = open(nombre_archivo, modo)
```

- `archivo` objeto file, el cual será utilizado para llamar a otros métodos asociados con el archivo.
- `nombre_archivo` string que contiene el nombre del archivo al que queremos acceder.
- `modo` string que contiene el modo de apertura del archivo.

Para cerrar un archivo con Python, podemos usar la función close.

```
archivo.close()
```

El método **read** permite extraer un string que contenga todos los carácteres del archivo.

```
# Abrimos el archivo en modo lectura y escritura
archivo = open('archivo.txt', 'r+')
texto = archivo.read()
print('El contenido del archivo es: ' + texto)
# Cerramos el archivo
archivo.close()
```

El método **readlines** permite obtener una lista con todas las líneas que contiene el archivo.

```
archivo = open('archivo.txt', 'r+')
lista_lineas = archivo.readlines()
for linea in lista_lineas:
   print(linea, end="")
# Cerramos el archivo
archivo.close()
```

Si solamente requerimos recorrer el archivo línea por línea, el objeto file es iterable.

```
archivo = open('archivo.txt', 'r+')
for linea in archivo:
   print(linea, end="")
# Cerramos el archivo
archivo.close()
```

El método **write** permite escribir una cadena de caracteres dentro del archivo.

```
archivo = open('archivo.txt', 'w')
archivo.write('Primer linea de texto\n')
archivo.write('segunda linea\n')
archivo.write('tercera linea\n')
archivo.close()
```

Podemos usar la sentencia **with** para abrir archivos en Python y dejar que el intérprete se encargue de cerrar el mismo.

```
path = './reg/logs.txt'
with open(path, 'r') as archivo:
    texto = archivo.read()
print(texto)
```

json

Es un formato para el intercambio de datos basado en texto.

El paquete json permite traducir un diccionario a formato JSON utilizando el método **dump**.

```
import json

data = {}
data['clientes'] = []
data['clientes'].append({ 'nombre': 'Juan', 'edad': 27})
data['clientes'].append({ 'nombre': 'Ana', 'edad': 26})

with open('data.json', 'w') as file:
    json.dump(data, file, indent=4, ensure_ascii=False )
```

La lectura es similar al proceso de escritura, se debe abrir un archivo y procesar esté utilizando el método load.

```
import json

with open('data.json') as file:
    data = json.load(file)
```

lambda

Es la palabra reservada para definir una función anónima, una función anónima debe ser un tipo de función muy abreviada la cual se escribe en solo una línea de código.

Sintaxis:

```
lambda num: pow(num,2)
```

Partes:

- "**lambda**": Es la palabra reservada para definir una función anónima
- "**num**": es el parámetro que tendrá la función anónima (como los parámetros de las funciones normales)
- El separador " **:** " entre los parámetros y el cuerpo de la función (donde se coloca el bloque de código).
- "pow(num, 2)": La función lambda no requiere escribir la sentencia "return". Retorna el código que se escriba del lado derecho (siendo en este caso la potencia del parámetro num por 2)

Pygame

Pygame es un contenedor de Python para la biblioteca SDL (Simple DirectMedia Layer)
** **SDL**: Biblioteca de C pensada para el desarrollo de juegos. Puede tener acceso a componentes hardware de la computadora (sonido, video, mouse, teclado, joystick, entre otros)*

Instalar pygame

Para instalar pygame y poder utilizar el módulo, en la consola escribimos:

```
pip install pygame
```

Inicializar y pantalla

- Inicializar Pygame:

```
pygame.init()
```

- Inicializa la pantalla:

```
pantalla = pygame.display.set_mode((ANCHO,ALTO))
```

- Nombre de ventana:

```
pygame.display.set_caption("nombre ventana")
```

- Colorea la pantalla en base a un color RGB (valores enteros):

```
pantalla.fill((R,G,B))
```

- Actualiza elementos (superficies) en pantalla

```
pygame.display.flip()
```

Cuadrados, círculos y texto

- Dibujar rectángulo:

```
pygame.draw.rect(pantalla,(R,G,B),(x,y))
```

- Dibujar círculo:

```
pygame.draw.circle(pantalla,(R,G,B),r)
```

- Crear fuente para texto:

```
fuente = pygame.font.SysFont("Nombre Fuente",tamaño)
```

- Renderizar texto:

```
texto = fuente.render("TEXTO A MOSTRAR",True,(R,G,B))
```

- Fundir texto en una superficie:

```
superficie.blit(texto, (x, y))
```

Superficies, rectángulos y fundir

Superficie: Una superficie representa en memoria un buffer de píxeles. Cada elemento que se representa en una pantalla es una superficie (imagen, texto, entre otros)

Rectángulo: Es una representación abstracta de un rectángulo sus parámetros básicos son: (x, y, ancho, alto)

Fundir: Se trata de unir una capa con otra. Para fundir en pygame usamos *blit*().

- Genera superficie en base a imagen:

```
superficie = pygame.transform.load('path_de_imagen')
```

- Modifica el tamaño de la superficie:

```
superficie = pygame.transform.scale
```

- Obtiene el rectángulo de la superficie:

```
rectangulo = superficie.get_rect()
```

- Funde superficie y rectángulo en la pantalla:

```
pantalla.blit(superficie, rectangulo)
```

Eventos

```
# Setear eventos de tiempo:
tick_1s = pygame.USEREVENT + 0
pygame.time.set_timer(tick_1s, 1000) # setea un timer en milisegundos
tick_2s = pygame.USEREVENT + 1 # crear un segundo timer
pygame.time.set_timer(tick_2s, 2000)

tiempo = pygame.time.get_ticks() # obtener tiempo actual

# guardar lista de eventos
eventos = pygame.event.get # guardar lista de eventos
for event in eventos:
    if event.type == pygame.QUIT # salir del juego
            hacer algo

    if event.type == pygame.MOUSEBUTTONDOWN: # presionar botón del mouse
        posicion = list(event.pos) # posicion del mouse

    if event.type == pygame.USEREVENT: # evento de usuario
        if event.type == tick_1s # USEREVENT de 1 segundo
            hacer algo

    if event.type == pygame.KEYDOWN: # evento de teclado (DOWN;UP;etc)
        if event.key == pygame.K_x: #detecta una sola tecla "x"
            hacer algo
```

Para importar varias teclas:

```
Importar el módulo:
      from pygame.locals import K_x

Dentro del loop:
    # Lista de teclas presionadas o no presionadas:
    teclas_presionadas = event.type == pygame.key_get_pressed()
    if True in teclas_presionadas:
        if teclas_presionadas[K_x]:
            hacer algo
```

Sonidos

- Inicializar mixer sonidos:

```
pygame.mixer.init()
```

- Setear volumen general:

```
pygame.mixer.music.set_volume(0,5)
```

- carga sonido de sistema:

```
sonido = pygame.mixer.sound("fondo.wav")
```

- volumen sonido:

```
sonido.set_volume(0,5)
```

- iniciar sonido:

```
sonido.play()
```

- parar sonido:

```
sonido.stop()
```

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Programación Orientada a Objetos

*es: **POO**, *en:* **OOP**

Puntos principales

1. **Declaración de una clase**

Para declarar una clase, se usa la palabra reservada class, seguida del nombre que queramos ponerle (sin paréntesis al final)
Sintaxis:

```
class Boton:
    pass
```

2. **Declaración de un constructor**

El constructor va a ser el encargado de recibir todos los datos de afuera y ponerlos a disposición en la clase según su nivel de protección. Se define como init.
Sintaxis:

```
class Boton:
    def __init__(self, path_imagen, x, y):
        self.path_imagen = path_imagen
        self.x = x
        self.y = y
```

3. **Atributos del objeto**

Son los atributos (variables) que se definen dentro del constructor.
Estos atributos podrán ser llamados (luego de instanciarse) desde dentro o fuera de la clase según su nivel de protección.

Protección de atributos y métodos

1. **Atributos y métodos protegidos**

Son atributos y/o métodos que por convención no deben ser modificados, pero permiten ser accedidos (y también modificados) tanto dentro de la clase, como por fuera. Se declaran con un guión bajo al comienzo del nombre:

```
class Personaje:
    def __init__(self, id, nombre):
        self.id = id
        self._nombre = nombre # Atributo protegido
personaje_A = Personaje(0, 'Marty') # Instanciar
print(personaje_A._nombre) # Marty
```

2. **Atributos y métodos privados**

A diferencia del punto anterior, los atributos y/o métodos privados son aquellos donde solamente tienen alcance dentro de la clase donde se declararon. Éstos no podrán ser llamados por fuera de la clase. Se declaran con doble guion bajo al comienzo del nombre:

```
class Personaje:
    def __init__(self, id, nombre):
        self.id = id
        self.__nombre = nombre # atributo privado

personaje_A = Personaje(0, 'Marty')
print(personaje_A.__nombre)
# AttributeError: 'Personaje' object has no attribute '__nombre'
```

Creación y manipulación de métodos

1. **Crear un método**

Un método es una función que se crea dentro de la clase donde, luego de haberse instanciado, se podrá utilizar. Por ejemplo:

```
class Personaje:
    def __init__(self, nombre):
        self.__nombre = nombre

    def presentacion(self):
        return 'Hola! Mi nombre es {0}'.format(self.__nombre)

personaje_A = Personaje('Marty')
print(personaje_A.presentacion()) # Hola! Mi nombre es Marty
```

2. **Getter**:

Un getter permite crear una función en una propiedad. Es decir, fuera de la clase, con el getter se puede llamar a la función como si fuera una propiedad del objeto instanciado:

```
class Personaje:
    def __init__(self, nombre):
        self.__nombre = nombre

    @property
    def presentacion(self):
        return 'Hola! Mi nombre es {0}'.format(self.__nombre)

personaje_A = Personaje('Marty')
print(personaje_A.presentacion) # Hola! Mi nombre es Marty
```

3. **Setter**

Al igual que getter, crea una propiedad donde puede asignarse un nuevo valor:
class Personaje:

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

    @property
    def edad(self):
        return 'Hola! Mi nombre es {0}'.format(self.__nombre)

    @edad.setter
    def set_edad(self, value):
        return self.edad = value

personaje_A = Personaje('Marty', 17)
personaje_A.set_edad = 18
print(personaje_A.presentacion)
# Hola! Mi nombre es Marty y tengo 18 años
```

Métodos especiales/mágicos/dunder de los objetos (Python)

1. **__str__**

Muestra un objeto. Puede utilizarse para devolver un string cuyo propósito sea mostrar el objeto que se creó, por ejemplo:

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

    def __str__(self) -> str:
        return 'Nombre: {0}; Edad: {1}'.format(self.__nombre, self.edad)

personaje_A = Personaje('Marty', 17)
print(personaje_A) # Nombre: Marty; Edad: 17
```

2. **__len__**

Permite utilizar la función ***len()*** sobre una instancia creada de la clase.

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

    def __len__(self) -> str:
        return self.edad
personaje_A = Personaje('Marty', 17)
print(len(personaje_A))  # '18'
```

3. **__getitem__**

Permite llamar un elemento de una lista que esté dentro de la clase:

```
class Personaje:
    def __init__(self, nombre, apellido, edad):
        self.__nombre = nombre
        self.__apellido = apellido
        self.edad = edad
        self._lista = [nombre, apellido, edad]

    def __getitem__(self, i):
        return self._lista[i]

personaje_A = Personaje('Marty', 'McFly', 18)
print(personaje_A[0])  # 'Marty'
```

4. **__setitem__**

Es el mismo que el anterior, pero permite setear un valor en la posición de la lista que esté dentro de la clase:

```
class Personaje:
    def __init__(self, nombre, apellido, edad):
        self.__nombre = nombre
        self.__apellido = apellido
        self.edad = edad
        self._lista = [id, nombre, apellido, edad]

    def __setitem__(self, i, value):
        self._lista[i] = value

personaje_A = Personaje('Marty', 'McFly', 18)
personaje_A[1] = 'Mc'
print(personaje_A[1])  # 'Mc'
```

5. **__contains__**

Consulta si un ítem se encuentra (in) en una lista que esté dentro de la clase. Devuelve como resultado un bool:

```
class Personaje:
    def __init__(self,nombre, apellido, edad):
        self.__nombre = nombre
        self.__apellido = apellido
        self.edad = edad
        self._lista = [id, nombre, apellido, edad]

    def __contains__(self, value):
        return value in self._lista

personaje_A = Personaje('Marty', 'McFly', 18)
print('Marty' in personaje_A) # True
print('Marta' in personaje_A) # False
```

6. **__iter__**

Método que permite iterar sobre una lista de la clase:

```
class Personaje:
    def __init__(self,nombre, apellido, edad):
        self.__nombre = nombre
        self.__apellido = apellido
        self.edad = edad
        self._lista = [id, nombre, apellido, edad]

    def __iter__(self):
        for i in range(len(self._lista)):
            yield self._lista[i]

personaje_A = Personaje('Marty', 'McFly', 18)
for el in personaje_A:
    print(el)
```

** **yield** es una función que retorna un diccionario iterable. Dicho diccionario se loopea fuera de la clase*

Operadores de comparación (POO)

Método	Operador	Significado
`__lt__`	`<`	Menor que
`__le__`	`<=`	Menor o igual que
`__eq__`	`==`	Igual que
`__ne__`	`!=`	Diferente que
`__gt__`	`>`	Mayor que
`__ge__`	`>=`	Mayor o igual que

Ejemplo:

```
class Personaje:
   def __init__(self, id, nombre, apellido, edad) -> None:
      self.id = id
      self.nombre = nombre
      self._lista= [id, nombre, apellido, edad]
   def __lt__(self, item):
       return self.edad < item.edad

personaje_A = Personaje(0,'Marty','McFly', 18)
personaje_B = Personaje(0,'Emmet','Brown', 54)
print(personaje_A < personaje_B) # True
```