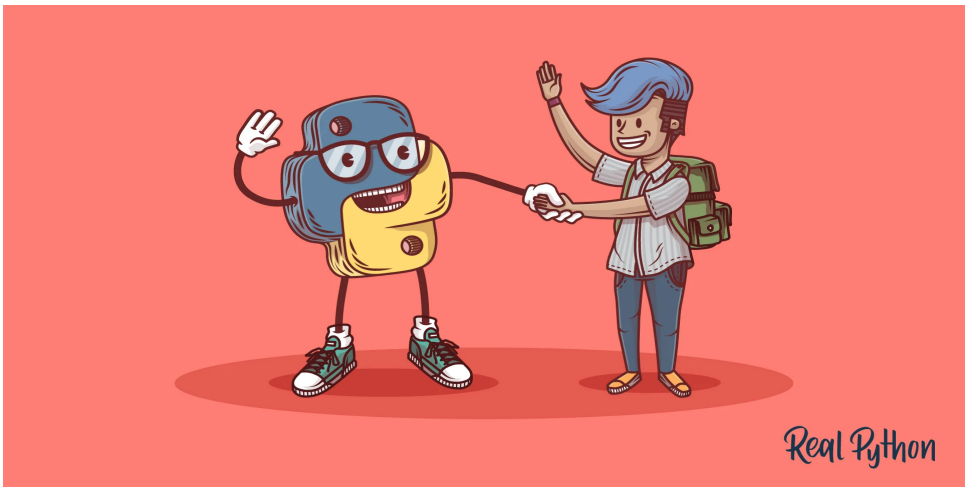
Revisitando Python



Fuente: RealPython (https://realpython.com/python-first-steps/)

Antes de iniciar en los temas propios del curso, es necesario que volvamos a revisar algunos detalles que quiza se quedaron en el tintero (o con mayor rigor, en el teclado) o que no se tomaron en cuenta o que sencillamente han sido olvidados. Así que vamos a repasar las cosas que debieron haber aprendido y quiza alguno olvido en el camino.

Luis A. Muñoz

Funciones definidas por el usuario

Las funciones son codigos autónomos que implementan soluciones puntuales. Son parte del rompecabezas que puede ser un proyecto de programación final. Consisten en un bloque de codigo que toma valores de entrada (parametros) para ser parte del proceso y retornar un valor o varios como resultado.

In [7]:

```
def factores_primos(n):
    '''factores primos(n)   Funcion que retorna los factores primos de un numero

    Parametros:
        - n: int

    Uso:
        factores_primos(12)  -> [2, 2, 3]'''
    if n < 1 or not isinstance(n, int):
        return []

    div = 2
    factores = []
    # Lazo que encuentra los divisores del numero (factores)
    while n is not 1: "while n!=1:"
        # Si se tiene un factor, se guarda en la lista de salida
        if n % div == 0:
            factores.append(div)
            n //=  div
        else:
            # ...de lo contrario se pasa a considerar el siguiente posible divisor
            div += 1

    return factores
```

In [8]:

```
print(factores_primos(1.5))
```

```
[]
```

In [4]:

```
help(factores_primos)
```

```
Help on function factores_primos in module __main__:

factores_primos(n)
    factores primos(n)   Funcion que retiorna los factores primos de un numero

    Parametros:
        - n: int

    Uso:
        factores_primos(12)  -> [2, 2, 3]
```

Excepciones

Las funciones pueden imprimir información pero no suelen imprimir resultados, ya que estos serán interpretados por otros programas como parte de un proyecto de programación. La impresión de un error puede ser util para el programador, pero si un código llama a una función debe de saber si ha ocurrido un error al momento de llamar a la función y para esto debe de recibír un mensaje de error que el proceso pueda interpretar como tal. Por eso es preferible generar Excepciones a imprimir mensajes de error.

In [52]:

```
def factorial(n):
    '''factorial(n)     Retorna el factorial de n (forma recursiva)

    Parametros:
        - n: int

    Uso:
        factorial(5) -> 120'''
    if not isinstance(n, int):
        raise TypeError("El valor 'n' debe ser un 'int'") '''Retorna una excepcion'''

    if n < 0:
        raise ValueError("El valor 'n' debe ser un 'int' >= 0")

    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In [53]:

```
print(factorial(5))
```

```
120
```

In [54]:

```
print(factorial(1.5))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-3520c8deda00> in <module>
----> 1 print(factorial(1.5))

<ipython-input-52-0727e516acbe> in factorial(n)
      8         factorial(5) -> 120'''
      9     if not isinstance(n, int):
---> 10         raise TypeError("El valor 'n' debe ser un 'int'")
     11
     12     if n < 0:

TypeError: El valor 'n' debe ser un 'int'
```

In [55]:

```
print(factorial(-1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-55-8f4cb14927b2> in <module>
----> 1 print(factorial(-1))

<ipython-input-52-0727e516acbe> in factorial(n)
     11
     12     if n < 0:
---> 13         raise ValueError("El valor 'n' debe ser un 'int' >= 0")
     14
     15     if n == 0:

ValueError: El valor 'n' debe ser un 'int' >= 0
```

Bloque try... except

Cuando un script llame a la función `factorial` puede hacer un "intento" con la instrucción `try` y si es que esta función genera una Excepción esta es capturada con la instrucción `except` y se puede mostrar un error propio del script.

In [19]:

```
num = 1.5        # Prueba float
# num = -5       # Prueba negativo

try:
    factorial(num)
except TypeError:
    print("Error: 'num' debe ser entero")
except ValueError:
    print("Error: 'num' debe ser mayor o igual a 0")
except:
    print("Error: Error general")
```

```
'num' debe ser entero
```

*args y **kwargs

En algunas ocasiones se requiere que una función tenga un número de parametros aleatorios, ya sea en forma de argumentos posicionales o por `keywords`. Esto se logra con los symbolos especiales `*` (desempaquetado en tuplas) y `**` (desempaquetado en diccionarios) y normalmente se especifican de la forma `*args` y `**kwargs`.

In [31]:

```
def foo(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print(k, '->', v)
```

In [38]:

```
foo('A')
```

```
A
```

In [39]:

```
foo('A', 'B', 'C')
```

```
A
B
C
```

In [40]:

```
foo(num1=1)
```

```
num1 -> 1
```

In [41]:

```
foo(num1=1, num2=2, num3=3)
```

```
num1 -> 1
num2 -> 2
num3 -> 3
```

In [42]:

```
foo('A', 'B', 'C', num1=1, num2=2, num3=3)
```

```
A
B
C
num1 -> 1
num2 -> 2
num3 -> 3
```

Esta especificación permite reducir la definición de los argumentos de una función y se utiliza mucha en la documentación de Python. Un buen ejemplo es el método `format` de la clase `str` que soporta diferentes argumentos y keywords:

In [47]:

```
help(str.format)
```

```
Help on method_descriptor:

format(...)
    S.format(*args, **kwargs) -> str

    Return a formatted version of S, using substitutions from args and kwargs.
    The substitutions are identified by braces ('{' and '}').
```

Generadores con yield

Una función puede ser un *generador*, esto es un objeto que no retorna una secuencia de elementos sino un *motor* que genere valores según la regla de la función y que será controlado por un iterador como un lazo `for` . Para esto, el lugar de retornar una lista o una tupla, la función retorna los valores por separado utilizando la instrucción `yield` en lugar de `return` :

In [103]:

```
def range_letters(ini='A', end='Z', case='upper', reverse=False):
    if not isinstance(ini, str) or not isinstance(end, str):
        raise TypeError

    if all([ini.isalpha(), end.isalpha()]):     # all: True si [True, True]
        if ini.upper() < end.upper():
            if not reverse:
                letter = ord(ini.upper()) - 1
                while letter < ord(end.upper()):
                    letter += 1
                    if case == 'upper':
                        yield chr(letter).upper()
                    elif case == 'lower':
                        yield chr(letter).lower()
                    else:
                        raise AttributeError
            else:
                letter = ord(end.upper()) + 1
                while letter > ord(ini.upper()):
                    letter -= 1
                    if case == 'upper':
                        yield chr(letter).upper()
                    elif case == 'lower':
                        yield chr(letter).lower()
                    else:
                        raise AttributeError
        else:
            raise ValueError
    else:
        raise ValueError
```

Si se llama a la función lo que retornará en un objeto generador:

In [110]:

```
print(range_letters('A', 'C'))
```

```
<generator object range_letters at 0x0000027F1A17E548>
```

Se puede utilizar la función `next` para que retorna el siguiente valor del generador. `next` ira retornando los valores del generados hasta que genere una excepción por haber agotado con los valores.

In [112]:

```
gen = range_letters('A', 'C')
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))   # Esta linea genera un Excepcion StopIteration
```

```
A
B
C
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-112-c010a437df3e> in <module>
      3 print(next(gen))
      4 print(next(gen))
----> 5 print(next(gen))   # Esta linea genera un Excepcion StopIteration

StopIteration:
```

¿Ahora entiende porque la función `range` al ser impresa no retorna el rango de números sino la especificación del rango a generar? Esto es porque `range` es un generador y si se especifica que se quieren generar 10, 100 o 1000 números, el uso de los recursos es el mismo pues la función solo retorna un número por vez y no una lista de números.

In [115]:

```
print(range(1, 11))
```

```
range(1, 11)
```

¿Ahora también entiende por que el lazo `for` en Python es tan extraño por no utilizar indices o controles de fin de iteración? Porque es un lazo `while` con la instrucción `next` en el interior y se mantiene hasta que se genera la exception `StopIteration` (siempre y cuando el elemento a iterar sea un generador).

In [118]:

```
for letter in range_letters('C', 'H', reverse=True):
    print(letter)
```

```
H
G
F
E
D
C
```

enumerate y zip

Las funciones `enumerate` y `zip` agregan funcionalidades a un iterador, es decir, a un lazo `for` . Por ejemplo, `enumerate` retorna una lista enumerada que es desempaquetada por el lazo:

In [120]:

```
# Por cada elemento iterado se le agrega una etiqueta numerica (por defecto, start=0)
for idx, letter in enumerate(range_letters('A', 'C'), start=1):
    print(idx, ':', letter)
```

```
1 : A
2 : B
3 : C
```

Por otro lado, `zip` es una *cremallera* que permite extraer los datos de dos o mas iterables que puude ser desempaquetada por el lazo

In [121]:

```
minusculas = range_letters('A', 'C', case='lower')
mayusculas = range_letters('A', 'C', case='upper')

for min_, may in zip(minusculas, mayusculas):    # "min_" ya que "min" es una funcion de Python (BIF)
    print(min_, '-', may)
```

```
a - A
b - B
c - C
```

map y filter

Otra operación útil en las colecciones de elementos es la capacidad de modificar los valores o filtrarlos bajo un criterio. Esto se lográ con las instrucciones `map` y `filter` , combinadas con las funciones anónimas `lambda` :

In [124]:

```
numeros = [1, 2, 3, 4, 5]

numeros_cuad = list(map(lambda x: x**2, numeros))
numeros_pares = list(filter(lambda x: x%2 == 0, numeros))"Devuelve tru o false"

print("numeros =", numeros)
print("numeros_cuad = ", numeros_cuad)
print("numeros_pares =", numeros_pares)
```

```
numeros = [1, 2, 3, 4, 5]
numeros_cuad =  [1, 4, 9, 16, 25]
numeros_pares = [2, 4]
```

Listas por comprehension

Las listas por comprehensión es de esas cosas que una vez que se pruebas y se entienden ya no se pueden dejar de utilizar. Es una forma de lograr operaciones de lazo para generar una lista utilizando una construcción diferente pero más legible.

Así, en lugar de escribir:

In [125]:

```
lista = []

for i in range(1, 10):
    lista.append(i)

print(lista)
```

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

Escibimos:

In [126]:

```
lista = [i for i in range(1, 10)]
print(lista)
```

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

La generación por medio de listas por comprehensión no solo es más legible sino que mucho más rápida. Se puede combinar con `if` e inclusive con `for` anidados en una sola instrucción. Utilizando esto se puede realizar lo mismo que con `map` y `filter` sin tener que recurrir a funciones `lambda` y se consigue un código más légible:

In [128]:

```
numeros = [1, 2, 3, 4, 5]

numeros_cuad = [n**2 for n in numeros]
numeros_pares = [n for n in numeros if n%2 == 0]

print("numeros =", numeros)
print("numeros_cuad = ", numeros_cuad)
print("numeros_pares =", numeros_pares)
```

```
numeros = [1, 2, 3, 4, 5]
numeros_cuad =  [1, 4, 9, 16, 25]
numeros_pares = [2, 4]
```

Una lista por comprehensión puede tener lazos anidados:

In [145]:

```
lista2D = [(a, chr(b)) for a in range(1, 5)
                       for b in range(ord('A'), ord('E'))]
print(lista2D)
```

```
[(1, 'A'), (1, 'B'), (1, 'C'), (1, 'D'), (2, 'A'), (2, 'B'), (2, 'C'), (2, 'D'), (3, 'A'), (3,
'B'), (3, 'C'), (3, 'D'), (4, 'A'), (4, 'B'), (4, 'C'), (4, 'D')]
```

Conjuntos y Diccionarios por comprehension

Tambien se pueden generar `set` y `dict` por comprehension.

In [147]:

```
print("Conjunto =", {n for n in range(1, 6)})
print("Diccionario =", {k: v for k, v in zip(['A', 'B', 'C'], ['a', 'b', 'c'])})
```

```
Conjunto = {1, 2, 3, 4, 5}
Diccionario = {'A': 'a', 'B': 'b', 'C': 'c'}
```

Sin embargo, un diccionario se puede generar de forma más sencilla directamente con la instrucción `zip y dict` :

In [150]:

```
print("Diccionario =", dict(zip(['A', 'B', 'C'], ['a', 'b', 'c'])))
```

```
Diccionario = {'A': 'a', 'B': 'b', 'C': 'c'}
```

Valores de un diccionario con `get`

Para obtener el valor correspondiente a la llave de un diccionario se puede especificando la llave entre `[]` :

In [151]:

```
may2min = dict(zip(['A', 'B', 'C'], ['a', 'b', 'c']))
print(may2min['A'])
```

```
a
```

Sin embargo, cuando se busca el valor asociado a una llave que no exista se generará una excepción:

In [152]:

```
print(may2min['M'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-152-9b2c01cb249e> in <module>
----> 1 print(may2min['M'])

KeyError: 'M'
```

Esto require que en el script se tenga que asegurar este procedimiento colocando las consultas a un diccionario en un bloque `try...except` . Sin embargo, hay una mejor forma y es utilizando el método `dict.get()` ya que este retorna un objeto especificado como segundo parametro:

In [154]:

```
print(may2min.get('M', 'Esta llave no existe'))
```

```
Esta llave no existe
```

Esto se puede combinar con un `if` y asi tener una consulta legible y más sencilla:

In [156]:

```
letra = input("Llave: ")

if may2min.get(letra, False):
    print(may2min[letra])
else:
    print("Llave no existe")
```

```
Llave no existe
```

f-strings

Un f-string es una cadena de texto con formato incluido y es la forma preferida de generar cadenas de texto desde la version 3.6 de Python. Se utiliza el caracter `f` antes de definir una cadena con comodines de formato `{}` y en el interior se colocan los valores a ser asignados.

In [368]:

```
peso = 80
altura  = 1.60
imc = peso / altura**2

# Uso del método format de los strings
print("Para una persona de {:.2f} m y {} kg, el IMC es de {:.1f}".format(altura, peso, imc))
```

```
Para una persona de 1.60 m y 80 kg, el IMC es de 31.2
```

In [369]:

```
# Uso de un f-string
print(f"Para una persona de {altura:.2f} m y {peso} kg, el IMC es de {imc:.1f}")
```

```
Para una persona de 1.60 m y 80 kg, el IMC es de 31.2
```

Programación Orientada a Objetos (OOP)

Es un paradigma de programacion en el que se crea una entidad llamada "clase" que encapsula en una sola estructura propiedades (variables) y funciones (métodos) asociados a un "objeto". Luego, estos objetos pueden ser "instanciados" a partir de una clase que sirve como plantilla.

En el siguiente código se crear una clase Alumno en terminos de nombre, apellido, codigo y estado de matricula. Se definen los *setters* y *getters* para el control de atributos (a excepción de *presente* ya que es un atributo que se define internamente). Tambien se define el método *esta_matriculado* para conocer el estado de la matricula (el *getter* de self.maricula) asi como el método *mágico* `__repr__` que retiorna una cadena que describe el objeto.

`__repr__` y `__str__` son ligeramente diferentes: el primero es invocado cuando se consulta por un objeto y el segundo cuando se imprime un objeto. Sin embargo, si `__str__` no está definido, se utiliza de forma automática `__repr__` por lo que es preferible definir este que `__str__` .

In [192]:

```
class Alumno:
    def __init__(self, nombre='', apellido='', codigo=''):
        self.nombre = nombre
        self.apellido = apellido
        self.codigo = codigo
        self.matricula = False

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

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

    @property
    def codigo(self):
        return self.__codigo

    @nombre.setter
    def nombre(self, val):
        if isinstance(val, str):
            self.__nombre = val
        else:
            raise TypeError("El campo 'nombre' debe ser tipo 'str'")

    @apellido.setter
    def apellido(self, val):
        if isinstance(val, str):
            self.__apellido = val
        else:
            raise TypeError("El campo 'apellido' debe ser tipo 'str'")

    @codigo.setter
    def codigo(self, val):
        if isinstance(val, str):
            self.__codigo = val
        else:
            raise TypeError("El campo 'codigo' debe ser tipo 'str'")

    def esta_matriculado(self):
        return self.matricula

    def __repr__(self):
        return f"Alumno(nombre={self.nombre}, apellido={self.apellido}, codigo={self.codigo})"
```

In [193]:

```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
print(alumno1)
print(alumno1.esta_matriculado())
```

```
Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
False
```

Herencia

Se puede crear una clase a partir de otra, colocando la clase *padre* entre () a momento de definirla. Esta clase heredara todas las propiedades y métodos de la clase padre.

In [194]:

```
# Clase Delegado es hija de Clase Alumno
class Delegado(Alumno):
    def __init__(self, nombre, apellido, codigo, curso):
        super().__init__(nombre, apellido, codigo)
        self.curso = curso

    def __repr__(self):
        return f"Alumno(nombre={self.nombre}, apellido={self.apellido}, codigo={self.codigo}, curso={self.curso
})"
```

In [195]:

```
delegada = Delegado('Dina', 'Mita', 'u20181811', 'Programacion')
print(delegada)
print(delegada.esta_matriculado())

delegada.matricula = True
print(delegada.esta_matriculado())
```

```
Alumno(nombre=Dina, apellido=Mita, codigo=u20181811, curso=Programacion)
False
True
```

Observe que la clase `Delegado` no define la propiedad `self.matricula` ni el método `self.esta_matriculado` ; sin embargo estan disponiibles para ser utilizado por el objeto `Delegado` instanciado.

staticmethod y classmethod

Una características de la definición de una clase es que la palabra reservada `self` es reemplazada por la instancia del objeto. Esto significa que todos los métodos definidos en una clase están asociados al objeto. A esto se le llama métodos estáticos:

Por ejemplo:

```
alumno = Alumno('ELvio', 'Lado', 'a81277222')
alumno.esta_presente()
```

El método `self.esta_presente` de la clase `Alumno` se ejecuta sobre el objeto instanciado `alumno` .

Hay casos en los que se quiere que un método se ejecute sobre la clase. Esto sucede, por ejemplo, en `numpy` :

```
array = np.array([1, 2, 3, 4, 5])
array = np.append(array, 6)
print(array

[1, 2, 3, 4, 5, 6]
```

Como se observa, en `numpy` es la clase `np` la que llama al método y no el objeto instanciado `array` .

En el primer caso estamos ante lo que so conoce como `staticmethod` y en el segundo caso tenemos un `classmethod` . Por defecto los métodos que se definen en una clase son del tipo `staticmethod` . Probemos crear un `classmethod` en la clase Alumnos (y vamos a olvidar los *setters* y *getters* para no complicar el código):

In [213]:

```
class Alumno:
    matriculado = False

    def __init__(self, nombre='', apellido='', codigo=''):
        self.nombre = nombre
        self.apellido = apellido
        self.codigo = codigo

    def esta_matriculado(self):
        return matriculado

    def __repr__(self):
        return f"Alumno(nombre={self.nombre}, apellido={self.apellido}, codigo={self.codigo})"
```

In [214]:

```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
print(alumno1)
print(alumno1.matriculado)
```

```
Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
False
```

Observe que la propiedad `matriculado` es una propiedad de la clase y por eso se puede llamar directamente. De hecho, se puede llamar de varias formas:

```
return matriculado
return self.matriculado
return cls.matriculado
```

Esto porque una vez instanciada la clase en un objeto, el propiedad es generica, de la clase y tambien del objeto...

Pero veamos el siguiente codigo para entender el alcance de esta definición:

In [217]:

```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
alumno2 = Alumno('Dina', 'Mita', 'u20181811')
print(alumno1.matriculado)
print(alumno2.matriculado)

Alumno.matriculado = True
print(alumno1.matriculado)
print(alumno2.matriculado)
```

```
False
False
True
True
```

Como se observa, al modificar la propiedad de la clase `matriculado` se modifica el estado de todos los objetos instanciados tipo `Alumno`. Se puede crear un método que afecte esta propiedad de la clase utilizando el decorador @classmethod, pero eso no tiene mucho sentido ya que podría ser llamada por lo objetos (se decir, ¡si se matricula el `alumno1`, matricula a todos los alumnos!).

Uno de sus usos más utiles es para generar nuevos contructores. Por ejemplo, digamos que se quiere generar alumnos a partir de cadenas de texto de la forma "nombre apellido codigo" que se extraen de una archivo. Entonces podemos definir un `classmethod` que afecte la forma como funciona el instanciamiento del objeto en la clase:

In [222]:

```
class Alumno:
    matriculado = False

    def __init__(self, nombre='', apellido='', codigo=''):
        self.nombre = nombre
        self.apellido = apellido
        self.codigo = codigo

    def esta_matriculado(self):
        return matriculado

    @classmethod
    def from_string(cls, string):
        nombre, apellido, codigo = string.split()
        return cls(nombre, apellido, codigo)      # cls: Equivalente a Alumno(nombre, apellido, codigo)

    def __repr__(self):
        return f"Alumno(nombre={self.nombre}, apellido={self.apellido}, codigo={self.codigo})"
```

In [223]:

```
alumno_str = "Elvio Lado a81277222"
alumno1 = Alumno.from_string(alumno_str)
print(alumno1)
```

```
Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
```

namedtuples

Una de las ventajas obvias de la OOP es el encapsulamiento de las propiedades y métodos en una sola entidad. Sin embargo, a veces solo se quiere utilizar el encapsulamiento de propiedades ya que no se requieren métodos: se quiere utilizar la clase como un sistema de almacenamiento eficiente de forma tal que se puede llamar a las propiedades en lugar de tener muchas variables en el código.

Por ejemplo, dejando de lado los ejemplos para demostrar algunos aspectos teoricos, la clase `Alumnos` no tiene un método útil: es solo un buen elemento para almacenar datos de un alumno.

Se puede contruir una clase sencilla que solo contemple propiedades utilizando `namedtuples` .

In [230]:

```
from collections import namedtuple

Alumno = namedtuple('Alumno', ['nombre', 'apellido', 'codigo'])
```

In [231]:

```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
print(alumno1.nombre)
print(alumno1)
```

```
Elvio
Alumno(nombre='Elvio', apellido='Lado', codigo='a81277222')
```

¡Y tiene su propio `__repr__` !

Metodos mágicos o "dunder methods"

Los métodos que inician y terminan con "__" de denominan "under-under" o "doble-under" o "dunder" o simplemente métodos mágicos. Estos son la columna vertebral del Modelo de Datos de Python. El estudio de esto escapa el alcance de este curso, pero conocer la idea general de esto ayuda a entender la forma como Python funciona como lenguaje de programación.

Los métodos mágicos son heredados de una *superclase* que es la clase Padre de todas las clases que se definen en Python. Muchas instrucciones y operaciones estan asociadas a estos métodos. Por ejemplo, cuando se numas dos variables tipo `int` como:

```
a, b = 3, 5
print(a + b)
```

Lo que realmente sucede es lo siguiente:

```
print(a.__add__(b))
```

Se esta invocando el método estático `__add__` sobre el objeto `a` con el parametro de entrada `b`. Esto es muy útil pues permite definir la operación suma en un objeto. Por ejemplo, si definimos una clase llamada Grupo:

In [251]:

```
class Grupo:
    def __init__(self, *args):
        self.miembros = [arg for arg in args]

    def __repr__(self):
        str_out = "Miembros del grupo:\n"
        for idx, miembro in enumerate(self.miembros, start=1):
            str_out += f'{idx} - {miembro}\n'

        return str_out
```

Luego, podemos modificar nuestra clase Alumnos para hacer algo bastante natural:

In [252]:

```
class Alumno:
    def __init__(self, nombre='', apellido='', codigo=''):
        self.nombre = nombre
        self.apellido = apellido
        self.codigo = codigo

    def __add__(self, other):
        return Grupo(self, other)

    def __repr__(self):
        return f"Alumno(nombre={self.nombre}, apellido={self.apellido}, codigo={self.codigo})"
```

In [253]:

```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
alumno2 = Alumno('Dina', 'Mita', 'u20181811')
grupo = alumno1 + alumno2
print(grupo)
```

```
Miembros del grupo:
1 - Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
2 - Alumno(nombre=Dina, apellido=Mita, codigo=u20181811)
```

Hemos utilizado el operador `+` para sumar estudiantes y meterlos en un grupo de trabajo. Esto resulta mejor que tratar de crear un método en la clase `Grupo` que permita agregar alumnos a una clase, ¿no? Pero esto puede ponerse mejor. Modifiquemos la clase `Grupo` con dos métodos mágicos adicionales:

In [269]:

```
class Grupo:
    def __init__(self, *args):
        self.miembros = [arg for arg in args]

    def __len__(self):
        return len(self.miembros)

    def __getitem__(self, idx):
        return self.miembros[idx]

    def __repr__(self):
        str_out = "Miembros del grupo:\n"
        for idx, miembro in enumerate(self.miembros, start=1):
            str_out += f'{idx} - {miembro}\n'

        return str_out
```

Parece poco pero ahora podemos hacer muchas cosas con un objeto clase `Grupo` . Primero, creemos un Grupo de cinco alumnos:

In [272]:

```
alum1 = Alumno('Elvio', 'Lado', 'a81277222')
alum2 = Alumno('Dina', 'Mita', 'u20181811')
alum3 = Alumno('Elmer', 'Curio', 'u2019122')
alum4 = Alumno('Alan', 'Brito', 'u20192816')
alum5 = Alumno('Susana', 'Oria', 'u20162722')

grupo = Grupo(alum1, alum2, alum3, alum4, alum5)
print(grupo)
```

```
Miembros del grupo:
1 - Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
2 - Alumno(nombre=Dina, apellido=Mita, codigo=u20181811)
3 - Alumno(nombre=Elmer, apellido=Curio, codigo=u2019122)
4 - Alumno(nombre=Alan, apellido=Brito, codigo=u20192816)
5 - Alumno(nombre=Susana, apellido=Oria, codigo=u20162722)
```

Ahora podemos sacar cuantos miembros hay en el grupo:

In [273]:

```
print(len(grupo))
```

```
5
```

Ademas, gracias al método `__getattr__` tenemos indices:

In [276]:

```
print(grupo[0])
print(grupo[-1])
```

```
Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
Alumno(nombre=Susana, apellido=Oria, codigo=u20162722)
```

Inclusive index slicing:

In [277]:

```
print(grupo[::2])
```

```
[Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222), Alumno(nombre=Elmer, apellido=Curio, codig
o=u2019122), Alumno(nombre=Susana, apellido=Oria, codigo=u20162722)]
```

Y nuestro grupo es iterable!:

In [279]:

```
for alumno in grupo:
    print(alumno.nombre)
```

```
Elvio
Dina
Elmer
Alan
Susana
```

¿Y si queremos elegir un alumno al azar como delegado del grupo? No necesitamos definir un método en la clase `Grupo`. Podemos utilizar el método `choice` de la librería `random`:

In [295]:

```
from random import choice

delegado = choice(grupo)
print(f"Delegado: {delegado.nombre} {delegado.apellido}")
```

```
Delegado: Elmer Curio
```

¿Y si queremos la lista de los alumnos del grupo ordenados por orden alfabetico en un función del nombre?...

In [301]:

```
for alumno in sorted(grupo, key=lambda x: x.nombre):    # key: funcion que retorna el elemento de referencia pa
ra el ordenamiento
    print(alumno)
```

```
Alumno(nombre=Alan, apellido=Brito, codigo=u20192816)
Alumno(nombre=Dina, apellido=Mita, codigo=u20181811)
Alumno(nombre=Elmer, apellido=Curio, codigo=u2019122)
Alumno(nombre=Elvio, apellido=Lado, codigo=a81277222)
Alumno(nombre=Susana, apellido=Oria, codigo=u20162722)
```

La simple definición de los métodos `__len__` y `__getitem__` permiten que nuestra clase pueda interactuar con el Modelo de Datos de Python.

time y datetime

La gestión del tiempo en Python es un dolor de cabeza. Esto porque hay diferentes maneras de hacer los mismo con diferentes módulos... Asi que para encontrar un orden, se puede tomar la siguiente guia de referencia:

- time: información del tiempo (horas, minuto segundo)
- datetime: manipulacion de fechas(operaciones con el tiempo)

time

El módulo `time` permite obtener información del tiempo. Por ejemplo, se puede obtener una *estructura* del tiempo con `time.localtime` de tal forma que los diferentes campos se puede extraer con el operador `.`:

In [318]:

```
import time

time_now = time.localtime()
print(time_now)

print(f"{time_now.tm_hour}:{time_now.tm_min}")
print(f"Han pasado {time_now.tm_yday} dias desde el 1 de Enero de {time_now.tm_year}")
```

```
time.struct_time(tm_year=2020, tm_mon=7, tm_mday=20, tm_hour=15, tm_min=30, tm_sec=26, tm_wday=0,
tm_yday=202, tm_isdst=0)
15:30
Han pasado 202 dias desde el 1 de Enero de 2020
```

La información del tiempo se puede obtener (en inglés) como una cadena de impresión con `time.asctime` ("asc" de ASCII):

In [319]:

```
time.asctime()    # Por defecto es el tiempo actual. Prueba como argumento time_now
```

Out[319]:

```
'Mon Jul 20 15:30:33 2020'
```

Sin embargo, una buena practica de programación no es usar el tiempo local sino el tiempo universal (UTC). Esto es, información de tiempo referenciado a un meridiano terreste (UTC-0). Esto permite generar un valor de tiempo que será reconocido y convertido a tiempo local en la zona de consulta. Para esto se necesita un tiempo de referencia donde todos los husos horarios consideren "el inicio del tiempo". Eso existe, es el "UNIX Time" referenciado a un "epoch": 1 de enero de 1970.

In [323]:

```
time_now = time.time()
print(time_now)
```

```
1595277085.9266865
```

Ese es el número de segundos que han transcurrido desde el 1 de enero de 1970. Si queremos retornar esto a una estructure de tiempos debemos llamar al método `time.gmtime` ("gm" de "Greenwich Mean Time" o "GMT"):

In [333]:

```
time.gmtime(time_now)
```

Out[333]:

```
time.struct_time(tm_year=2020, tm_mon=7, tm_mday=20, tm_hour=20, tm_min=31, tm_sec=25, tm_wday=0,
tm_yday=202, tm_isdst=0)
```

Estas estrcturas de tiempo se pueden mostrar con un formato personalizado (ya que al usar `asctime` obtendremos una respuesta en inglés y con un formato fijo) con el método `time.strftime` que tiene la siguiente forma:

(Extraido de la ayuda del método `time.strftime`)

```
    strftime(...)
        strftime(format[, tuple]) -> string

        Convert a time tuple to a string according to a format specification.
        See the library reference manual for formatting codes. When the time tuple
        is not present, current time as returned by localtime() is used.

        Commonly used format codes:

        %Y  Year with century as a decimal number.
        %m  Month as a decimal number [01,12].
        %d  Day of the month as a decimal number [01,31].
        %H  Hour (24-hour clock) as a decimal number [00,23].
        %M  Minute as a decimal number [00,59].
        %S  Second as a decimal number [00,61].
        %z  Time zone offset from UTC.
        %a  Locale's abbreviated weekday name.
        %A  Locale's full weekday name.
        %b  Locale's abbreviated month name.
        %B  Locale's full month name.
        %c  Locale's appropriate date and time representation.
        %I  Hour (12-hour clock) as a decimal number [01,12].
        %p  Locale's equivalent of either AM or PM.

        Other codes may be available on your platform.  See documentation for
        the C library strftime function.
```

In [338]:

```
time.strftime("%I:%M:%S %p", time.localtime())
```

Out[338]:

```
'03:41:21 PM'
```

datetime

El módulo `datetime` se utiliza para hacer operaciones con las fechas. Esto resulta muy útil pues evita tener que hacer calculos complejos para calcular una fecha en el futuro o en el pasado, tomando en consideración los años bisiestos, los meses de 30 o 31 dias, etc.

Para obtener la fecha de hoy como un objeto DateTime llamamos al método `datetime.datetime.now` :

In [3]:

```
import datetime

time_now = datetime.datetime.now()
print(time_now)
```

```
2020-07-23 19:14:29.127380
```

O podemos definir una fecha en el futuro:

In [4]:

```
time_past = datetime.datetime(2001, 1, 1, 0, 0)    # 1 de Enero 2001, 00:00 horas
print(time_past)
```

```
2001-01-01 00:00:00
```

Si hacemos una operaciones de suma o resta obtendermos un nuevo objeto: `timedelta` , una diferencia de tiempo:

In [5]:

```
time_delta = time_now - time_past     # Cuanto tiempo ha transcurido del siglo XXI?
print(time_delta)
```

```
7143 days, 19:14:29.127380
```

Se puede definir un `timedelta` para calcular una fecha (se define con información de segundos, minutos, horas o dias).

In [6]:

```
time_delta = datetime.timedelta(days=100)
time_future = datetime.datetime.now() + time_delta
print(time_future)
```

```
2020-10-31 19:14:32.585384
```

In [ ]:

```

```