1. Objetos de instancia y de clase

Como se mencionó anteriormente los objetos que interactúan en un sistema OO tienen un estado que está dado por los valores de sus "atributos" (o "variables de instancia") en determinado momento.

Generalmente, los estados de los objetos varían en el tiempo, ya que desde sus métodos es posible alterar los valores de sus atributos. Estos no deberían de ser manipulados directamente por el resto de los objetos del sistema, por eso se los trata de "esconder" nombrándolos con un doble guion bajo al nombrarlos. Este asunto se verá más detalladamente en el apunte de **Encapsulamiento**.

1.1. Variables de instancia

En `Python` los atributos de un objeto suelen declararse en el *inicializador* (__init__) anteponiendo la referencia de instancia `self`. Esto ocurre porque en `Python` todos son objetos y son dinámicos. Esto implica 2 cosas, una que cada atributo y método son objetos y que, además, a cada instancia se le agregan los atributos o métodos dinámicamente como se observa en el Programa 1.

Programa 1: Agregar atributos dinámicamente

```
1  class MiClase:
2    def __init__(self):
3      self.__atributo1 = "agregado al inicializar la instancia"
4
5    def mostrar_atributo1(self):
6      print(f"atributo1 -> {self.__atributo1}")
7
8  mi_objeto = MiClase() # se crea una instancia de MiCalse
9  mi_objeto.nuevo_atributo = "agregado luego de inicializar"
10 mi_objeto.mostrar_atributo1()
11 print(f"nuevo_atributo -> {mi_objeto.nuevo_atributo}")
```

En este ejemplo, la instancia `mi_objeto` de `MiClase` es inicializada con un `atributo1`, al igual que todas las demás al ser creadas como tal, ya que este atributo es agregado en el método `__init__` utilizando la referencia a la instancia `self`. Pero a este objeto en particular, se le agrega un atributo que no es compartido por ninguna otra instancia de `MiClase` u otra clase en la *línea 9*, llamado `nuevo_atributo`. Este solo existe en la presente

instancia y nada más que para ella, a estos se los conocen como *atributos ad-hoc*. Esta no es una práctica común, y puede ser confusa, por eso será **desaconsejada**.

1.2. Variables de clase

Dentro del ámbito de una clase, se pueden definir objetos que son considerados **variables de clase**. A diferencia de los atributos declarados en `__init__`, que son únicos para cada instancia, las variables de clase son compartidas por todas las instancias de la clase. Estos objetos incluyen tanto métodos como atributos.

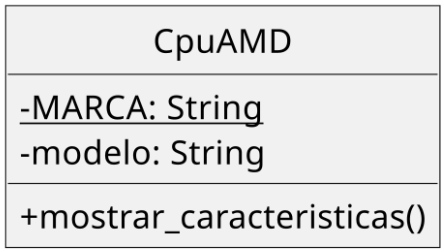
Las variables de clase permiten mantener datos comunes a todas las instancias, lo cual es útil para compartir comportamientos a través de los métodos o para utilizar valores que deben ser iguales para todas las instancias.

Programa 2: Atributos de clase

```
1  class CpuAMD:
2    __MARCA = 'AMD'
3
4    def __init__(self, modelo : str) -> None:
5      self.__modelo = modelo
6
7    def mostrar_caracteristicas(self):
8      print(f"CPU -> {CpuAMD.__MARCA} {self.__modelo}")
9
10 if __name__ == "__main__":
11   cpu1 = CpuAMD('Phenom II x6')
12   cpu2 = CpuAMD('Ryzen 5')
13   cpu1.mostrar_caracteristicas()
14   cpu2.mostrar_caracteristicas()
```

Debe notarse que para acceder a este atributo lo correcto es accederlo a través de `NombreClase.Atributo`, si bien puede hacerse desde `self.__MARCA` y en este ejemplo daría el mismo resultado, no se comporta exactamente igual. Esto se analizará en el apunte 4 sobre **herencia**.

Este tipo de atributos en otro lenguaje suelen llamarse **atributos estáticos** (*static*). En los diagramas **UML**, se indican subrayándolos.



Nota

Para conocer todos los objetos dentro de un módulo, clase o instancia puede utilizar la función `dir()` que devuelve una lista de *strings* con los nombres de atributos y métodos dentro de la clase.

```
>>> dir(CpuAMD('Ryzen 5'))
['_CpuAMD__MARCA', '_CpuAMD__modelo', '__class__', '
    __delattr__', '__dict__', '__dir__', '__doc__', '
    __eq__', '__format__', '__ge__', '__getattribute__',
    '__getstate__', '__gt__', '__hash__', '__init__', '
    __init_subclass__', '__le__', '__lt__', '__module__',
     '__ne__', '__new__', '__reduce__', '__reduce_ex__',
    '__repr__', '__setattr__', '__sizeof__', '__str__', '
    __subclasshook__', '__weakref__', '
    mostrar_caracteristicas']
```

2. Tipos de datos integrados

Hay muchos tipos de datos que ya son proporcionados con `Python`, los más comunes son los siguientes.

2.1. Tipos de datos simples

Estos objetos son los que representan datos únicos.

Tipos numéricos: Los tipos de datos/objetos más comunes para representar valores numéricos son `int`, `float` y `complex`.

Tipo booleano: Existe el tipo `bool` el cual usualmente toma los valores constantes `True` o `False` (empezados ambos en mayúscula)

2.2. Tipos secuencia

En su interior, estos objetos contienen muchos valores en un orden.

Listas: Las listas se declaran con corchetes `[]` o con el constructor explicito `list()`, son parecidas a los arreglos de datos en otros lenguajes de programación, estos son de una longitud variable y admiten cambios en sus elementos **son mutables**.

Tuplas: Se declaran con `()` o `tuple()`, al igual que las listas, pueden tener varios datos y acceder como si se tratara de un arreglo, pero los elementos en una tupla no pueden ser cambiados **son inmutables**.

Rangos: Los tipo de dato `range` son utilizados habitualmente para hacer bucles del tipo `for`. Generan un rango **inmutable** de valores enteros.

2.2.1. Listas

Creación de listas

Las listas se crean habitualmente utilizando corchetes `[]`.

```
lista_vacia = []
lista_con_elementos = [1, 2, 3, "cuatro", 5.0]
```

Acceso a elementos

Se puede acceder a los elementos de una lista utilizando índices.

```
lista = [1, 2, 3, 4, 5]
elemento = lista[0]
# elemento = 1
```

Modificación de elementos

Las listas son mutables, por lo que se pueden modificar sus elementos.

```
lista = [1, 2, 3, 4, 5]
lista[0] = 10
# lista = [10, 2, 3, 4, 5]
```

Añadir elementos

Se pueden añadir elementos a una lista usando los métodos `append()` y `extend()`.

```
lista = [1, 2, 3]
lista.append(4)
# lista = [1, 2, 3, 4]
lista.extend([5, 6])
# lista = [1, 2, 3, 4, 5, 6]
```

Eliminar elementos

Los elementos se pueden eliminar utilizando los métodos `remove()`, `pop()` o la instrucción `del`.

```
lista = [1, 2, 3, 4, 5]
lista.remove(3)
# lista = [1, 2, 4, 5]
elemento = lista.pop(1)
# lista = [1, 4, 5]
# elemento = 2
del lista[0]
# lista = [4, 5]
```

Índices y Slicing

Se puede acceder a sublistas utilizando slicing.

```
lista = [1, 2, 3, 4, 5]
sublista = lista[1:4]
# sublista = [2, 3, 4]
```

El slicing también permite omitir el índice inicial o final para seleccionar desde el principio o hasta el final de la lista.

```
inicio = lista[:3]
# inicio = [1, 2, 3]
fin = lista[2:]
# fin = [3, 4, 5]
```

También se puede utilizar un tercer parámetro para especificar el paso del slicing.

```
pasos = lista[::2]
# pasos = [1, 3, 5]
```

Longitud de una lista

La función `len()` devuelve la longitud de una lista.

```
lista = [1, 2, 3, 4, 5]
longitud = len(lista)
# longitud = 5
```

Métodos comunes

Las listas en Python tienen varios métodos incorporados para manipularlas:

- `sort()` para ordenar la lista.
- `reverse()` para invertir el orden de la lista.
- `index()` para encontrar la posición de un elemento.
- `count()` para contar las ocurrencias de un elemento.

```
lista = [3, 1, 4, 1, 5, 9, 2]
lista.sort()
# lista = [1, 1, 2, 3, 4, 5, 9]
lista.reverse()
# lista = [9, 5, 4, 3, 2, 1, 1]
posicion = lista.index(4)
# posicion = 2
cuenta = lista.count(1)
# cuenta = 2
```

2.2.2. tuplas

Creación de tuplas

Las tuplas se crean utilizando paréntesis `()` y pueden contener elementos de cualquier tipo.

```
tupla_vacia = ()
tupla_con_elementos = (1, 2, 3, "cuatro", 5.0)
# También se pueden crear sin paréntesis (tuple packing)
tupla_sin_parentesis = 1, 2, 3, "cuatro", 5.0
# Para tuplas con un solo elemento, se debe incluir una coma
   final
tupla_un_elemento = (1,)
```

Acceso a elementos

Se puede acceder a los elementos de una tupla utilizando índices.

```
tupla = (1, 2, 3, 4, 5)
elemento = tupla[0]
# elemento = 1
```

Inmutabilidad

Las tuplas son inmutables, por lo que no se pueden modificar sus elementos una vez creadas.

```
tupla = (1, 2, 3)
tupla[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Índices y Slicing

Se puede acceder a subtuplas utilizando slicing.

```
tupla = (1, 2, 3, 4, 5)
subtupla = tupla[1:4]
# subtupla = (2, 3, 4)
```

El slicing también permite omitir el índice inicial o final para seleccionar desde el principio o hasta el final de la tupla.

```
inicio = tupla[:3]
# inicio = (1, 2, 3)
fin = tupla[2:]
# fin = (3, 4, 5)
```

También se puede utilizar un tercer parámetro para especificar el paso del slicing.

```
pasos = tupla[::2]
# pasos = (1, 3, 5)
```

Longitud de una tupla

La función `len()` devuelve la longitud de una tupla.

```
tupla = (1, 2, 3, 4, 5)
longitud = len(tupla)
# longitud = 5
```

Desempaquetado de tuplas

Las tuplas pueden ser desempacadas en variables individuales.

```
tupla = (1, 2, 3)
a, b, c = tupla
# a = 1, b = 2, c = 3
```

Métodos comunes

Las tuplas en Python tienen menos métodos que las listas debido a su inmutabilidad. Los métodos más comunes son:

- `count()` para contar las ocurrencias de un elemento.
- `index()` para encontrar la posición de un elemento.

```
tupla = (1, 2, 3, 1, 4, 1)
cuenta = tupla.count(1)
# cuenta = 3
posicion = tupla.index(3)
# posicion = 2
```

Conversión entre listas y tuplas

Se puede convertir una lista a una tupla y viceversa usando las funciones `tuple()` y `list()`.

```
lista = [1, 2, 3]
tupla = tuple(lista)
# tupla = (1, 2, 3)
lista = list(tupla)
# lista = [1, 2, 3]
```

2.2.3. range()

Lo más común es utilizarlos para bucles controlados con la sentencia `for`.

```
>>> # Uso de range
>>> for i in range(10):
        print(i, end=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
>>>
```

Sobre el tipo `range()`

Los objetos tipo `range()` sirve para recorrer rangos, tiene algunas variantes:

```
for i in range(0, 10, 1): # range(start, stop, step)
    print(i)

for i in range(1,11): # por default el paso es 1
    print(i)          # muestra del 1 al 10

for i in range(10):   # por default inicio en 0
    print(i)          # muestra del 0 al 9
```

Tenga en cuenta que si quiere hacer rangos decrecientes o de pasos distintos de `+1`, entonces no puede omitir ningún dato:

```
for i in range(0,-10,-1):
    print(i)   # muestra del 0 al -9

for i in range(0,11,2):
    print(i)   # 0,2,4,6,8,10
```

Además puede generar listas en base a un `range`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.3. Strings

La información textual se representa en **Python** con objetos de tipo `str`, normalmente llamados cadenas de caracteres o simplemente *strings*. Son secuencias inmutables de puntos de código **Unicode**. Las cadenas se pueden definir de diferentes maneras:

- Comillas simples: `'permite incluir comillas "dobles"'`
- Comillas dobles: `"permite incluir comillas 'simples'"`
- Triples comillas: ya sea con comillas simples `'''Triples comillas simples'''` o dobles `"""Triples comillas dobles"""`

A continuación se describirán algunas de las operaciones posibles con *strings* en **Python**.

2.3.1. Concatenación

La concatenación de cadenas se puede realizar utilizando el operador +.

```
cadena1 = "Hola"
cadena2 = "Mundo"
resultado = cadena1 + " " + cadena2
# resultado = "Hola Mundo"
```

2.3.2. Repetición

Las cadenas se pueden repetir utilizando el operador *.

```
cadena = "Hola"
resultado = cadena * 3
# resultado = "HolaHolaHola"
```

2.3.3. Índices y Slicing

Se puede acceder a caracteres individuales y subcadenas usando índices y slicing.

```
cadena = "Hola Mundo"
caracter = cadena[1]
# caracter = 'o'
subcadena = cadena[1:5]
# subcadena = 'ola '
```

2.3.4. Longitud de una cadena

La función `len()` devuelve la longitud de una cadena.

```
cadena = "Hola Mundo"
longitud = len(cadena)
# longitud = 10
```

2.3.5. Métodos comunes

Las cadenas en Python tienen varios métodos incorporados para manipularlas:

- `lower()` y `upper()` para cambiar el caso.
- `strip()` para eliminar espacios en blanco al inicio y al final.

- `replace()` para reemplazar subcadenas.
- `find()` para encontrar la posición de una subcadena.

```
cadena = " Hola Mundo "
resultado = cadena.lower()
# resultado = " hola mundo "
resultado = cadena.upper()
# resultado = " HOLA MUNDO "
resultado = cadena.strip()
# resultado = "Hola Mundo"
resultado = cadena.replace("Mundo", "Python")
# resultado = " Hola Python "
posicion = cadena.find("Mundo")
# posicion = 6
```

2.3.6. Formateo de cadenas

El formateo de cadenas se puede realizar de varias maneras:

- Usando el operador `%`.
- Usando el método `format()`.
- Usando f-strings (Python 3.6+).

```
nombre = "Juan"
edad = 30

# Operador %
resultado = "Hola, %s. Tienes %d años." % (nombre, edad)
# resultado = "Hola, Juan. Tienes 30 años."

# Método format
resultado = "Hola, {}. Tienes {} años.".format(nombre, edad)
# resultado = "Hola, Juan. Tienes 30 años."

# f-strings
resultado = f"Hola, {nombre}. Tienes {edad} años."
# resultado = "Hola, Juan. Tienes 30 años."
```

En general se optará por la última opción, es decir los f-strings.

2.4. Tipos conjunto y mapa

En Python vienen integrados muchos tipos de datos que son colecciones dinámicas (como las listas), pero que a su vez organizan la información de una manera estructurada. Aquí se darán dos ejemplos concretos que son muy utilizados.

2.4.1. Conjuntos (set())

Los tipos de dato set() acumulan valores, al igual que las listas, pero no admiten repetidos y siempre se encuentran ordenados. Podemos inicializar un set vacío construyendo un objeto, o uno inicializado entre {}.

Programa 3: Ejemplos con set()

```
# Inicialización de un set vacío
set_uno = set()

# Se añaden los números del 0 al 9 en orden:
for i in range(10):
    set_uno.add(i)

# set inicialmente cargado con un 1,2,3 ("desordenado")
set_dos = {3,2,1}

# Se añaden los números del 9 al 0:
n = 9
while n >= 0:
    set_dos.add(n) # el 1, 2 y 3 se repiten...
    n -= 1

# Se muestran ambos sets
print(f"set_uno => {set_uno}")
print(f"set_dos => {set_dos}")
print(f"set_uno == set_dos => {set_uno == set_dos}")
```

El resultado de este código son dos set exactamente iguales:

```
set_uno => {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
set_dos => {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
set_uno == set_dos => True
```

Como se observa en el Programa 3, no importa el orden en que se ingresen los valores, o si intentamos agregar el mismo valor más de una vez, los sets terminan siendo iguales.

2.4.2. Mapas/Diccionarios (`dict()`)

En `Python` solo hay un tipo estándar de mapa: los *dictionary*. Estos toman un valor como clave (key) y están asociados a un valor que es accedido normalmente a través de la primera.

Los diccionarios se pueden construir de diferentes formas:

- Usando una lista separada por comas de pares '`clave: valor`' entre llaves:
  `{'jack': 4098, 'sjoerd': 4127}` o `{4098: 'jack', 4127: 'sjoerd'}`
- Usando un constructor de tipo:
  `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

Los diccionarios pueden tener alternados los pares '*clave-valor*', pero aún así se consideran iguales si tienen las mismas combinaciones:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

Los valores en un diccionario son fácilmente accesibles y asignables a través de su clave:

```
>>> d = dict()
>>> d['uno'] = 1 # se agrega 'uno':1 al diccionario
>>> d
{'uno': 1}
>>> d.get('uno') # Se puede usar el método get()
1
>>> d['uno'] # o los corchetes sin asignación
1
>>> del d['uno'] # Se borra el elemento con clave 'uno'
>>> d            # diccionario vacío
{}
>>> d = {"uno": 1, "dos": 2, "tres": 3, "cuatro": 4}
>>> list(d.keys())
['uno', 'dos', 'tres', 'cuatro']
>>> list(d.values())
[1, 2, 3, 4]
```

Se pueden recorrer los diccionarios de muchas formas, por clave, por valor, o por ambos utilizando el método `items()` que devuelve una lista de tuplas `(clave,valor)`:

Programa 4: Recorrer un diccionario utilizando `items()`

```
d = d = {"uno": 1, "dos": 2, "tres": 3, "cuatro": 4}
for k, v in d.items():
  print(f'{k} => {v}')
```

Por la pantalla saldrá la siguiente salida:

```
uno => 1
dos => 2
tres => 3
cuatro => 4
```

3. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: `https://docs.python.org/3/library/stdtypes.html`
- UBA – Algoritmos I: `https://algoritmos1rw.ddns.net/`
- Interprete Visual de Python: `http://www.pythontutor.com/visualize.html`