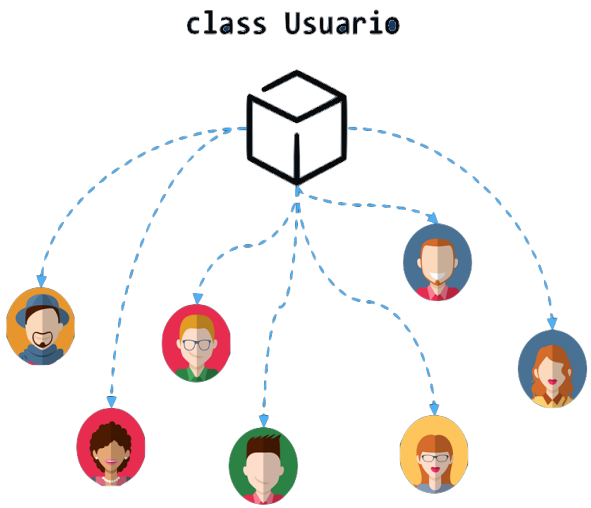
UNIVERSIDAD DE MURCIA

Clases y Objetos

Tecnología de la Programación

L. Daniel Hernández < *ldaniel@um.es* >

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
19 de septiembre de 2023



Índice de Contenidos

Desarrollo

Profundizamos en ...

Resumen sobre la abstracción

- La **abstracción** es el proceso mental por el que captamos las **características principales** de un concepto o proceso descartando los detalles aíslando conceptualmente las distintas partes, propiedades o cualidades de un objeto para estudiarlas por separado.
- Genera grupos jerárquicos: cada grupo es una abstracción que ignora algunas características de los elementos que forman parte de él.
- Mecanismos de abstracción:
  - **Parametrización**. Se usan parámetros para representar a un conjunto de elementos.
  - **Especificación**. Cumplimentar documentos indicando nombre, descripción y condiciones.
- Tipos de Abstracción: procedimental, iteración y abstracción de datos.
- Hay 3 niveles de abstracción de datos: tipos de datos integrados, tipos de datos definidos por el usuario, Tipos de Datos Abstractos.

Resumen sobre los TDAs

- Los **Tipos de Datos Abstractos** (TDA) son **modelos matemáticos** que constan de
  - un nombre publico para
  - identificar a un conjunto de datos (valores), junto con
  - un conjunto de operaciones bien definidas sobre los datos (como en una estructura algebraica).
- Como modelo, le es **irrelevante** cómo se almacenan o estructuren los datos y cómo se implementan las operaciones.
- Para todo TDA hemos de abordar **tres tareas**:
  - **Especificación:** definición del TDA
    - La **especificación** de un TDA (Datos+Operaciones) se rige por las normas generales de la **Abstracción por Especificación**
    - Se debe dar la especificación de los datos y de los operadores (constructores, modificadores y de consulta).
    - Hay que distinguir los operadores: fundamentales vs no fundamentales, públicos vs privados.
  - **Representación:** estructura con la que representar el TDA. Debe definir:
    - **La función de abstracción.** Una función sobreyectiva $Abst : \mathbf{rep} \longrightarrow \mathcal{A}$
    - **El invariante de la representación.** Es un predicado $I : \mathbf{rep} \longrightarrow \mathbb{B}$ que es cierto para los objetos de **rep** que sean legítimos.
  - **Implementación:** cómo implementar la estructura en un lenguaje de programación.

Desarrollo

Profundizamos en ...

Principios de la POO

- **Nuestro Objetivo:**

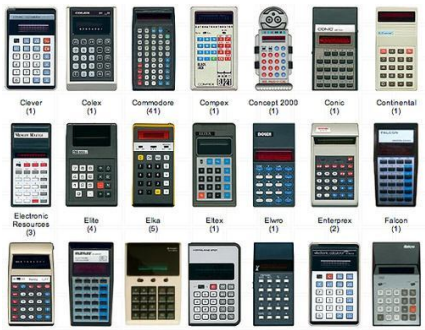
  Usar la POO para implementar TDAs (que son modelos matemáticos) en `Python`.

- Conceptos asociados a la POO
  - **Abstracción** es un proceso mental de extracción de las características esenciales de un concepto o proceso descartando los detalles. Hay dos tipos de abstracción:
    - TDA: Abstracción de tipo
    - POO: Abstracción operacional
  - **Encapsulamiento.** Proceso por el que se agrupan datos y operaciones, ocultando los detalles internos.
    - No se oculta la información, sino su soporte.
    - La información se accede por una interface.
  - **Jerararquización**. Estructurar por niveles (jerarquía) los elementos que intervienen en el proceso.
    - Jerarquía de clasificación (**Herencia**).
    - Jerarquía de composición (**Asociación, agregación**)
  - **Modularidad**. Descomposicón del sistema en conjunto de módulos poco acoplados (independientes) y cohesivos (con significado propio)

Profundizamos en ...

TDA ⇒ Clase. Clase ⇏ TDA

- Diseñar un TDA es **abstraer** lo que tienen de común entes parecidos: calculadoras, estudiantes, coches, ...



- Ejemplo: Consideremos **dos estudiantes**. Ambos parecen tener **coincidencias** en
  - Los mismos atributos: están en un curso, tienen una edad, ...
  - Los mismos comportamientos: se desplazan, estudian, cambian objetos en la mochila, ...

  Realmente dos estudiantes se **diferencian** en los valores de algunos atributos pero tienen los **mismos comportamientos/métodos** (con resultados posiblemente **diferentes** dependiendo de sus atributos).
- Cuando **un conjunto de objetos presentan los mismos métodos y se diferencian solo en los estados**, diremos que dicho conjunto es una **clase** en POO.
  TDA es el término formal. Clase es el término en POO. Pero no son lo mismo.
- **Los TDAs y las clases son tipos de datos.**

Declaración de clases

- Un **clase** es el término usado en programación para implementar un TDA en POO.
- Informalmente, **una clase es una plantilla**/molde para construir los objetos.
- En Pseudocódigo, una primera aproximación es el siguiente esquema:

```
1  class Car:
2      # Variables posibles para definir un objeto
3      luces: bool   # booleano on/off
4      color: str    # string
5      #
6      # Métodos
7      None turnHeadLights():
8          luces ← not luces
9      None moveForward():
10         ...
```

1. Se usa la palabra reservada class
2. Se da un nombre a la clase Car (conjunto de objetos)
3. Se indican los atributos comunes de todos los objetos: variables
4. Se indican los métodos comunes de todos los objetos: similar a las funciones.

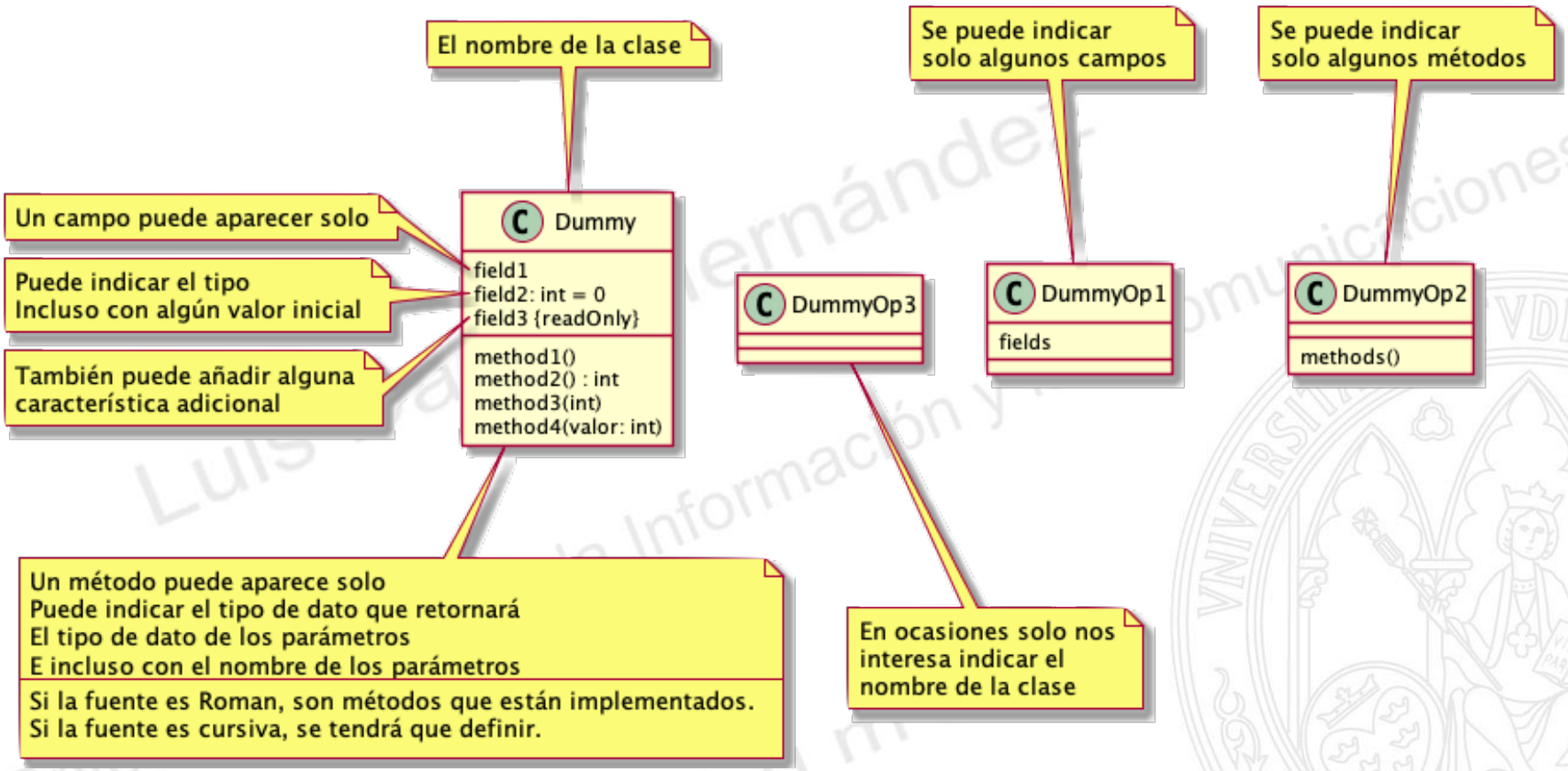
Ejercicio

Ejemplo

Desarrolla una aproximación inicial para declarar las clases correspondientes a los siguientes tipos de objetos:

1. funciones matemáticas,
2. conjuntos,
3. estadística descriptiva básica.

Representación UML de una clase



Ejercicio.

Haz la representación UML de la clase `Car` de la transparencia anterior.

Profundizamos en ...

Encapsulamiento

- En POO **una clase** es una plantilla y **define** un tipo de dato.
- En POO **un objeto** representa a una entidad (física o abstracta) que viene dado por la **particularización de una abstracción** (definida por una clase/TDA).
- Un objeto presenta:
  - Un **estado**, dado por los valores concretos de sus **variables** de la representación. Las variables también se llaman **atributos** o **campos**.
  - Un **comportamiento**, representado por los **métodos** (realmente atributos precedimentales).
- Estas dos componentes, juntas (o agrupadas) definen al objeto.
- La agrupación se llama **encapsulamiento**:

Fuente www.lineaysalud.com

- El **encapsulamiento** es el proceso por el que **un objeto** tiene sus propios datos y métodos.
- La encapsulamiento requiere que su atributos queden **ocultos** (pero se estudiará con más detenimiento con los modificadores de acceso)

Profundizamos en ...

Constructores de Objetos

- Para trabajar con clases y objetos:
  - Hay que **indicar antes** cuál es **la clase**.
  - y **después los objetos** que la componen indicando cuál es el estado de cada uno.
- Para construir objetos, se necesitan **constructores**.
- Un **constructor** es un método especial que se caracteriza porque:
  - No retorna nunca un valor
  - Se llama igual que la clase
  - Sus parámetros se identifican con algunos atributos.
    Por tanto, sus argumentos establecen el estado inicial del objeto.
- La construcción genera **una referencia** al objeto (ver ciclo de vida).
- Algunos ejemplos para la clase `Car` en `pseudocódigo` son:

```
1  Car()                          {luces ← false;  color←"rojo";}
2  Car(String c)                  {luces ← false;  color←c;}
3  Car(boolean l)                 {luces ← l;  color←"blanco";}
4  Car(boolean l, String c) {luces ← l;  color←c;}
```

- Para construir un objeto, en `pseudocódigo`:

```
1 c1 ← Car();              // Luces apagadas, color rojo
2 c2 ← Car("green");       // Luces apagadas, color verde
3 c3 ← Car(true);          // Luces encendidas, color blanco
4 c4 ← Car(true, "blue");// Luces encendidas, color azul
```

Cambiando el Estado de un Objeto

- Se necesita una referencia, `nombreObjeto`, para acceder a un objeto.
- Para usar un atributo o un método de un objeto se usa la **notación punto**.
  - `nombreObjeto.atributo` referencia a un atributo del objeto.
  - `nombreObjeto.metodo()` referencia a un método del objeto.

Ejemplo:

```
c1.lights          # Referencia a la variable lights del coche c1
c2.moveForward()   # Referencia al método del coche c2.
```

- **Modificar el estado** de un objeto es cambiar los valores de sus variables/atributos.
  - **No se deben modificar** los valores de los atributos **directamente.**
    Los motivos se explicarán en el siguiente tema.
  - **La modificación del estado de un objeto se debe de realizar mediante sus métodos**
    Ejemplo: Acceso a un método para cambiar el estado

```
car.turnHeadLightsTo(true) # Cambia el estado de la luz
car.ligths = true          # Esto funciona pero está mal
```

El objeto self

- En POO se suele usar un objeto especial (this, self o Me)
- self se refiere al **objeto** que actualmente está ejecutando el código.
  - Recuerda que obj.atributo y obj.metodo() hace referencia al atributo .atributo y al método .metodo() del objeto obj.
  - Por tanto, self.atributo y self.metodo() hace referencia al atributo .atributo y al método .metodo() **del objeto que esté ejecutando el código** en ese momento.
- Ejemplo: Considera el siguiente código Python

```
1  class Estudiante:
2      ...
3      def notaPOO (self, nota):
4          self.nota = nota  # self.variable = parámetro
5
6  maria = Estudiante()
7  maria.notaPOO(10)
```

  - maria es el objeto que ejecuta el código (línea 7).
  - maria.notaPOO(10) hace referencia al método notaPOO() para el objeto maria.
  - En la ejecución, se realizará la instrucción self.nota = 10 (línea 4)
  - Como el objeto que lo invoca es maria, el objeto self es el objeto maria (línea 4)
    Es como si ejecutáramos: maria.nota = 10
  - En consecuencia, *al objeto maria se le asignará un 10 a su atributo nota*.

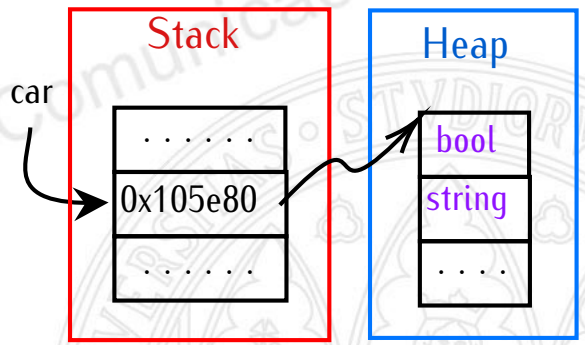
Ciclo de vida de un objeto

- Es distinto un objeto que una referencia a un objeto.
- El objeto está en el **Heap** y la referencia en **Stack**.
- Para que un objeto exista, éste necesita una referencia.
- Para la **creación** se necesita
  - Hacer una **declaración** del "objeto", indicando a qué clase pertenece – opcional, según Lenguaje.
  - **Construir** un objeto, invocando al constructor de objetos.
  - **Almacenar** la referencia retornada por el constructor.

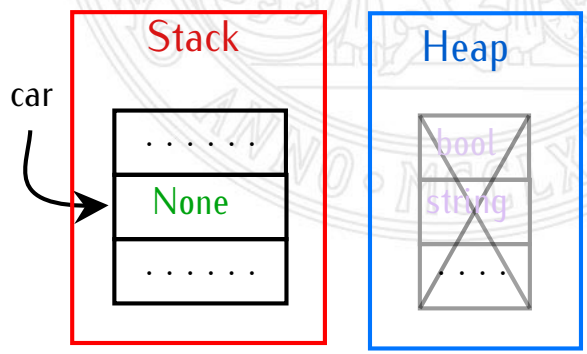
  Cada objeto se llama también **instancia de clase**.
- Mientras que exista se puede **modificar** su estado.
  - El estado se modifica siempre **a través de los métodos**
- Un objeto **deja de existir** si no tiene una referencia a él.
  - El **recolector de basura** lo borrará de la memoria.
  - No es lo mismo `car=None` que `del(car)`

  No haremos distinción entre objeto y referencia de objeto (se entenderá por el contexto)

```
1 // Si procede
2 Car car;

4 car ← Car();
```



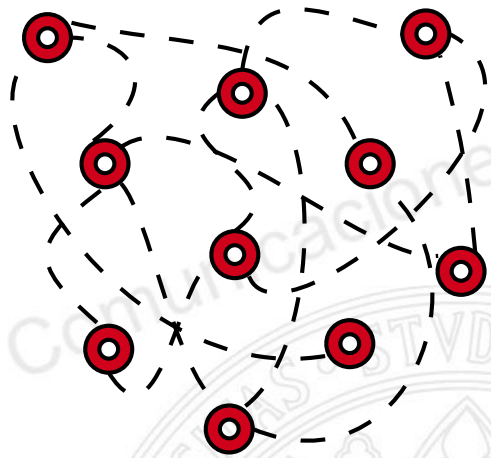

```
1 car.turnHeadLights();

3 car ← None;
```



Profundizamos en ...

Cómo se usa la POO para Resolver Problemas

- En todo problema se pueden identificar **conceptos**.
- Cada concepto definirá una **clase** (define antes un TDA).
  - Define un plantilla:
    estructura (datos) + métodos (operaciones)
- En un problema se identificarán entes/**objetos** concretos.
  - Un objeto se crea con un estado inicial.
  - **Estado:** valores de sus atributos/estructura.



- **Resolver un problema en POO es resolver un problema de Espacio de Estados**
  - La situación de un problema queda definida por los **estados** de los objetos.
  - Cada paso a la solución cambiará los estados de los objetos que interviene en el problema.
  - **Solución:** el estado de algunos objetos es el deseado.
- Para la resolución, los objetos se envían **mensajes** entre sí.
  - `usuario.saca_dinero(cajero)`
  - El usuario manda un mensaje al cajero.
  - Modificará el estado del usuario y del cajero.

Desarrollo

Profundizamos en ...

Iteradores en Python

- La abstracción por iteración se basa en usar **iteradores** para recorrer **contenedores**.
- En Python un **contenedor es un objeto** que almacena objetos.
- En Python un **iterador es un objeto** que sabe cómo recorrer un contenedor.
- El iterador se puede construir de forma explícita. `iter(contenedor)`
- Pero también se puede construir de forma implícita.

```
elem in contenedor
```

Retorna `True` si el objeto `elem` se encuentra en el contenedor y `False` en otro caso.

```
for elem in contenedor
    acción sobre elem
```

En cada bucle el iterador retorna un siguiente elemento llamado `elem`.

Contenedores e Iteradores en Python

Para entender como se usan los iteradores se necesita:

1. Conocer los tipos de contenedores que existen en Python.
   - Mutables: listas, conjuntos, diccionarios.
   - Inmutables: rangos, tuplas.
2. Saber construir nuestros propios contenedores.
   - Se definirá una clase `Container` cuyos atributos permitan almacenar secuencias de objetos.
   - Se definirá el método `.__iter__()` que retornará un objeto de tipo `Iterator`.
   - Será usado por la función `iter(container)`.
3. Saber construir iteradores en Python (o clases).
   - Se definirá una clase `Iterator` con un atributo que referencia a `container`.
   - También se definirán dos métodos:
     - `.__iter__()`, que retorna la referencia del propio iterador.
     - `.__next__()`, que retorna el siguiente elemento del contenedor. Cuando no hay más elementos lanza un error con la sentencia `raise StopIteration`.
     - Serán usados por las funciones `iter(iterator)` y `next(iterator)`.

Documentación oficial: https://docs.python.org/3/library/stdtypes.html#iterator-types

Profundizamos en ...

Funciones. Parámetros Posicionales

La Tabulación es Fundamental

- **Función:** una **secuencia** de instrucciones identificada con un **nombre** que retorna un valor.

```
def nombre_funcion ( lista de parámetros ) -> tipo de dato:
    estructuras de la función: secuencial, condicional, repetitiva.
    return valores
```

  En Python una función puede retornar varios valores.
  - **"Empaqueta"** todos los datos de retorno en un tipo de datos llamado **tupla**.

- **Procedimiento:** Función que no tiene la instrucción de retorno.
- Si una función/método tiene *n*-parámetros se puede invocar a la función con *n*-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetros, etc ...
  Son parámetros **posicionales**.

```
def fun(a, b, c, d):  # Tiene 4 parámetros
    print(a, b, c, d)

fun(1, 2, 3, 4) # Invocamos con 4 parámetros posicionales.
```

MUY IMPORTANTE

- **Una función no debe tener más de una responsabilidad/propósito.**
- Una "función" debe, en la medida de lo posible:
  - o realizar una **acción** (procedimiento)
  - o retornar un **cálculo** (función)
  - y no debería "nunca" realizar las dos cosas a las vez.

Ejercicio.

Se quiere hacer un programa que haga lo siguiente:

Mostrar si un número entero, dado por el usuario, es un número primo.

¿Cómo se haría desde un punto de vista procedimental?

Funciones. Valores Por Defecto. Palabras Clave

- Los **k-últimos parámetros** de un función pueden ser **opcionales**.
  - Los opcionales determinan un valor **literal por defecto**.
  - **Primero** los obligatorios **y después** los opcionales (o por defecto).

```
def fun(a, b, c=3, d=4): #  2 posicionales + 2 opcionales.
    pass
```

- Python permite invocar por **palabras claves** (keywords).
  - **Usar keyword** consiste en especificar el nombre del parámetro en la invocación.
  - El orden de los parámetros pueden cambiarse.
  - Los keywords siempre se pondrán al final.

```
# Para la función anterior
fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
fun(1, 2, d=4, c=3) # Los keywords al final
fun(d=4, 1, 2, c=3) # Incorrecto
```

Funciones. Parámetros Arbitrarios. Empaquetamiento

- Cuando no se conoce el número de argumentos que se usarán, se usa el **Packing Arguments** (empaquetamiento de argumentos) con el operador *

```
>>> def fun(*args):    # Empaquetará todos los argumentos
...     print(args)
...
>>> fun(1, 2, 3, 4)    # Invocación desempaquetada
(1, 2, 3, 4)
```

- Todos los argumentos se agrupan en una **tupla**.
- También existe el **Unpacking Arguments.** Dada una función/método con varios parámetros podemos empaquetar los argumentos con *.

```
>>> def fun(a, b, c, d):    # Desempaquetará si le envían \
...     print(a, b, c, d)   # los argumentos empaquetados
...
>>> lista = [1, 2, 3, 4] # Las listas las veremos más tarde.
>>> fun(*lista)           # Invocación empaquetada
1 2 3 4
```

Packing con Diccionarios

- Para acceder a cada uno de los argumentos empaquetados se usan **índices**:

```
>>> def fun(*args):
...     print(len(args), args[1]) # Muestra el cardinal y el 2o argumento
...
>>> fun(1, 2, 3, 4)
4 2
```

- Estaría mejor acceder a ellos con un nombre (keyword).
- Podemos **empaquetar y usar keywords** usando **diccionarios**, con **

```
>>> def fun(**kwargs):
...     # Muestra el diccionario
...     print(f"{len(kwargs)} elementos", kwargs)
...
>>> fun(a=1, b=2, c=3, d=4)     # Invocación con keywords
4 elementos {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> diccionario = {'p1': 1, 'p2': 2, 'p3':3}  # Los veremos
>>> fun(**diccionario)          # Invocación con empaquetado
3 elementos {'p1': 1, 'p2': 2, 'p3': 3}
```

Un diccionario es como un array pero los índices se sustituyen por claves.

Un ejemplo con todos los tipos de parámetros

- Si se quieren usar los 3 modos de pasar argumentos, el orden debe ser:
  1. posicionales
  2. empaquetados sin keyword
  3. empaquetados con keyword

Ejemplo:

```
>>> def fun(a, b, *args, **kwargs):
...     print(a, b)  # Muestra los posicionales
...     print(args)  # Muestra los empaquetados sin keywords
...     print(kwargs)# Muestra los empaquetados con keywords
...
>>> fun(1, 2, 3, 4, 5, p1=6, p2=7, p3=8)
1 2
(3, 4, 5)
{'p1': 6, 'p2': 7, 'p3': 8}
```

Ejercicio.

Para `def func(x, y, op = "+")` ¿Cuánto valdrá `x`, `y` y `op` en los siguientes casos?

- `func(2, 3, "*")`
- `func( (2, 3), "*")`
- `func( *(2, 3), "*")`
- `func( *(2, 3, "*"))`
- `func( (2, 3, "*"), 4)`
- `func( *(2, 3, "*"), 4)`

Cadenas de Documentación. Doctstrings

- Toda función debe contener su **especificación informal**.
- La documentación se hace en Python mediante **docstrings**: Cadenas que empiezan y terminan con triple comillas (simples o dobles)
- Existen varios **formatos** (lenguajes de marcas):
  - reStructuredText. El estandar de Python.
  - Google. La mejor alternativa "no-nativa".
  - Numpy. La propuesta de Numpy.
  - etc ...

```python
def func (x: int) -> str:
    """
    En formato reStructuredText

    :param x: La entrada, es entero
    :return: El valor de retorno
    """
    pass
```

```python
def func (x: int) -> str:
    """
    En formato Google

    Args:
        x: La entrada, es entero
    Returns:
        El valor de retorno
    """
    pass
```

Profundizamos en ...

Clases y Objetos

- Un TDA **encapsula** datos y operaciones (creación, modificación, acceso).
- En Python se implementa usando una **clase**.
- Una clase consta de una serie de "funciones" llamados **métodos**.
- Es en el método **inicializador** donde se definen los datos de la estructura.
  En Python no se recomienda definir el método constructor.
- Apariencia general

```
class Clase:
   def __init__(self, p1, p2, ...):  # "Creador"
       self.__p1 = p1 # REPRESENTACIÓN DEL TDA
       self.__p2 = p2 # Elementos de la estructura
       ..
   # OPERACIONES DEL TDA
   def metodo(self, p): # Modificador/Acceso
      pass
```

- Un caso concreto del TDA se establece mediante **objetos** (o instancias) de la clase que implementa el TDA.

```
objeto = Clase(valores_iniciales)
```

- Cómo definir más objetos desde el punto de vista ADP, se estudiará con las **interfaces**.

Funciones vs Métodos en Python

- Una **función** recibe como entrada una lista de parámetros
  - Se invoca como una instrucción más.

```
def funcion (lista de parámetros):
    cuerpo
```

- Un **método** (comportamiento de un objeto) debe incluir el parámetro `self`
  - Se invoca a través de un objeto de la clase (notación punto)

```
def metodo (self, lista de parámetros):
    cuerpo
```

Ejemplo:

```
>>> def funcion(x):
...     return 2*x
...
>>> class Prueba:
...     def metodo(self, x):
...         return funcion(x)
...
>>> p = Prueba(); print(f"{p.metodo(10)} {funcion(100)}")
20 200
```

Métodos Mágicos en Python

- En Python existen algunos métodos especiales llamados **métodos mágicos**.
- `__new__(cls, ...)`. Se invoca cuando un objeto es creado. **No usar**.
- `__init__(self)` Debe implementarse siempre
  - **Es el método inicializador** de instancia que se invoca siempre, una vez construido el objeto con `__new__(cls)`. SOLO PUEDE HABER UNO.
  - Se usa para **inicializar las variables** de la instancia (objeto) recién creado.
- `__del__(self)` No se recomienda su uso.
  - **Es un método finalizador** que se invoca cuando un objeto es recogido por el recolector de basura (garbage collected, GC)
  - Un objeto será recogido (destruido) cuando deja de tener referencias hacia él.
- `__str__(self)`
  - **Es un método de acceso** que retornar un string con información del objeto entendible por el usuario.
  - Se invoca al usar las funciones `print()` y `str()`.
- `__lt__(self, other)`, `__le__(self, other)`, `__eq__(self, other)`, `__ne__(self, other)`, `__gt__(self, other)`, `__ge__(self, other)`.
  - Definen los operadores $<$, $\leq$, $==$, $>$ y $\geq$ entre objetos.
  - Por defecto $x == y$ se define como **True if x is y else NotImplemented**

    `is` **comprueba la identidad**: es `True` sii a y b son el mismo objeto.

    `==` **comprueba la igualdad**: es `True` sii `__eq__` es `True`.

Ejemplo Completo

Los atributos de objeto se crean en `__init__()`, pero es buena costumbre declararlas con su tipo.

```
>>> class Car:
...     """
...     La clase Car representa al conjunto de los coches
...     """
...     def __init__(self, luces, color):
...         self.__luces: bool = luces
...         self.__color: str = color
...         self.__pos:int = 0
...     def __str__(self):
...         str = 'Estado del coche:\n'
...         str += '\tluces: '+ f'{self.__luces}\n'
...         str += '\tcolor: '+ self.__color + '\n'
...         str += '\tposición: '+ f'{self.__pos}'
...         return str
...     def turnHeadLights(self):
...         self.__luces = not self.__luces
...     def moveForward(self):
...         self.__pos = self.__pos + 1
...
```

```
>>> car = Car(False,'blanco')
>>> print(car)
Estado del coche:
    luces: False
    color: blanco
    posición: 0
>>> car.turnHeadLights()
>>> car.moveForward()
>>> print(car)
Estado del coche:
    luces: True
    color: blanco
    posición: 1
```

Ejercicios

Ejercicio. Complejos

Define el TDA Complejo con los operadores de suma y producto. Impleméntalos en Python.

Ejercicio. Rectángulos

- ¿De cuántas formas definirías un rectángulo en el plano? Se asume que el rectángulo no está rotado y que sus lados son paralelos a los ejes cartesianos.
- Se opta por codificar los rectángulos usando un punto vértice y su vértice opuesto, pero el usuario puede construir un rectángulo de estas dos formas:
  - Un punto vértice, su ancho y su largo.
  - Un punto vértice y su vértice opuesto.

  Observaciones:
  - Se asume que punto vértice es el "inferior-izquierdo" y que el resto de los datos se darán de forma correcta (para evitar programación defensiva).
  - Implementa la clases `Punto` y `Rectangulo`.

Desarrollo

TDA y ADP

- Programar TDAs $\not\equiv$ Programación Orientada a Objetos[1].
- La abstracción de datos se puede hacer
  - Por Tipos de Datos Abstractos (TDA): abstracción de tipo.
  - Por Abstracción de Datos Procedural (ADP, POO): abstracción procedural.

  Son 2 modos ortogonales de implementar una abstracción.
- Al hacer una abstracción podemos distinguir:
  - Constructores abstractos: Crean o instancian un dato abstracto (llamado objeto).
  - Observaciones abstractas[2]: Recuperan información de un dato abstracto.
- Se puede visualizar en una una tabla de doble entrada, donde cada celda representa la observación sobre una instancia:

  Ejemplo: Abstracción de una Lista

	Constructores listas		
Métodos	nil	add(L', n)	
null?(L)	true	false	¿Es nula la lista?
head(L)	error	n	Retorna la cabecera
tail(L)	error	L'	Retorna la cola
	(1)	(2)	(1) Lista vacía. (2) Lista que empieza por n.

[1] https://www.cs.utexas.edu/users/wcook/papers/OOPvsADT/CookOOPvsADT90.pdf

[2] Que llamaremos método para que resulte más intuitivo.

Programación de TDAs

- La programación de un TDA requiere definir una estructura y sus operaciones. En la tabla de doble entrada responde a trabajar por filas.
  Ejemplo: Implementación de una Lista como TDA

Métodos	Constructores listas: nil	Constructores listas: add(L', n)
null?(L)	true	false
head(L)	error	n
tail(L)	error	L'

  - **Representación:** NONE | CELL is int* list
  - **Operaciones:** Incluye constructores y métodos.

```
nil = NONE
add(l: list, n: int):
    CELL(l, n)
null?(l: list): por casos:
    NONE -> true
    CELL(l, n) -> false
```

```
head(l: list): por casos:
    NONE -> error
    CELL(l, n) -> n
tail(l: list): por casos:
    NONE -> error
    CELL(l, n) -> l
```

- Un cliente de TDA **declara variables** de tipo *Lista* y usaría las **operaciones** para crearlas/manipularlas: `var l: list = add(add(nil, 4), 7)`
- En esencia, un tipo define una **estructura** con **operaciones que hacen distinción de casos**.

Programación de ADP

- La programación de una ADP (POO) requiere pensar en **objetos** (instancias). Un objeto queda definido por sus **atributos procedurales**.
  Hay que pensar en las columnas de la doble entrada.

  Ejemplo: Implementación de objetos-lista como ADP

	Constructores listas	
Métodos	nil	add(L', n)
null?(L)	true	false
head(L)	error	n
tail(L)	error	L'

```
NONE = record
    null? = true
    head = error
    tail = error
```

```
CELL(l, n) = record
    null? = false
    head = n
    tail = l
```

  - Una observación `metodo` de un objeto `obj`, se implementa seleccionando el campo `metodo` del objeto `obj`: `obj.metodo`.

- Un cliente PDA **crea objetos** y le **envía mensajes**: `var l = Nil.add(4).add(7)`
- En esencia, un conjunto de funciones define un objeto y no hay distinción de casos.

Recuerda: TDA y ADP

- La abstracción de datos se puede hacer
  - Por Tipos de Datos Abstractos (TDA): abstracción de tipo.
  - Por Abstracción de Datos Procedural (ADP): abstracción procedural.

  Son 2 modos ortogonales de implementar una abstracción.
- Al hacer una abstracción podemos distinguir:
  - Constructores abstractos: Crean o instancian un dato abstracto (llamado objeto). Establece para el objeto creado los valores en sus atributos procedimentales (dados por las observaciones abstractas).
  - Observaciones abstractas[3]: Recuperan información de un dato abstracto. Recuperan información de los atributos de los objetos construidos.
- Cómo se programan TDAs y ADPs:
  - TDAs: constan de una estructura y un conjunto de operaciones.
    ```
    var l: list = add(add(nil, 4), 7)
    ```
  - ADPs: Trabajan con objetos
    ```
    var l = Nil.add(4).add(7)
    ```
- Entonces, ¿cómo usar la POO?

[3] Que llamaremos método para que resulte más intuitivo.

Cómo usar Lenguaje-POO para TDA y ADP

- Pensaremos en TDAs: **abstracción de tipo**.
  - Abstrae de todos los entes similares sus características comunes (estructuras y operaciones) para obtener un tipo de **ente "abstracto"**.
    - No pienses en las formas de construir los entes: solo en sus **atributos**.
    - Para sus operaciones no pienses en hacer distinción de casos: solo en sus **nombres**.
  - Un tipo de ente "abstracto" define una `clase` (plantilla). Ejemplo: Una persona abstracta.
  - **Recuerda: Una clase es una abstracción de un conjunto de entes, todos con la misma estructura/operaciones y se diferencian en sus estados.**
- Pensaremos en ADP: **abstracción procedural**.
  - Define los constructores de (todos) los **objetos**, pero solo considerando los atributos.
  - Un objeto construido a partir de una `clase` (plantilla) se dice que es de tipo `clase` Ejemplo: `luis` o `daniel` son de tipo `Persona`.
  - Considera para cada objeto cuál debe ser la observación del método para el siguiente paso.
- Vuelve a TDA y define los métodos haciendo distinción de casos.
- Vuelve a ADP para hacer más abstracciones definiendo interfaces (conj. de métodos) para definir a todos los objetos que tengan dichos atributos procedurales. **En la práctica,** será abstraer métodos comunes de los TDAs implementados y definir cada grupo de métodos como una `interface`.

Ejemplo de cómo usar Lenguaje-POO para TDA y ADP

Ejemplo Intuitivo para la Construcción del TDA Persona

- Pensaremos en TDAs: **abstracción de tipo**.
  - Abstraemos las características comunes de las personas para definir una clase :
    - **Atributos:** nombre, edad, ...
    - **Operaciones:** andar, hablar, ...
- Pensaremos en ADP: **abstracción procedural**.
  - Definimos los constructores (solo atributos): se necesita una cadena alfanumérica (para el nombre), un natural (para la edad), ...
    - Se hará distinción de casos: En función del número de atributos suministrados y de sus valores se construye cada persona.
- Vuelve a TDA y define los métodos haciendo distinción de casos.
  Por comodidad, se considera que todas personas andan y hablan igual.

```
andar() <- return "Muevo dos piernas"
hablar() <- return "Se hablar"
```

- Vuelve a ADP y abstrae los métodos de las clases
  - Considerando las clases Persona, Ave, Reptil, ... se concluye que todos tienen los métodos avanzar() y sonar().
    - En Persona, andar() se renombrar a avanzar()
    - En Persona, hablar() se renombra a sonar()