

CHULETAS DE PYTHON

RESUMEN DE LOS LIBROS DE
JOHN HUNT

4.7 Formateando cadenas

```
cadena1 = "Hola {}, ¿Cómo estás?"  
print(cadena1.format("Aurelio"))
```

Sustituyo el corchete por el valor de la otra cadena

PLACEHOLDER, puede ser int, string o float

```
# 4.7 String formatting pag. 43
```

```
cadena = "Mi nombre es {1} {0} y mi edad es {2}"
```

```
nombre = "Luis"
```

```
apellidos = "Pérez"
```

```
edad = 30
```

```
print(cadena.format(apellidos, nombre, edad))
```

```
cadena = "Mi nombre es {nombre} {apll1} y mi edad  
es {edad}"
```

```
print(cadena.format(nombre="Luis", apll1="Pérez", eda  
d=30))
```

Se puede indicar con índices

Se puede indicar con claves


4.7 Formateando cadenas

```
print('#{:25}#'.format('mi frase'))

print('|{:<25}|'.format('izquierda')) # por defecto alineación izquierda
print('|{:>25}|'.format('derecha')) #alineación derecha
print('|{: ^25}|'.format('acentros')) #alineación izquierda

print('{:,}'.format(123456789)) # formato numérico con separador de miles
print('{:,}'.format(1234.56789)) # formato numérico con separador de miles
print('{:,}'.format(123456789.45)) # formato numérico con separador de miles

print('{:25,}'.format(123456789)) # formato numérico con separador de miles
print('{:25,}'.format(1234.56789)) # formato numérico con separador de miles
print('{:25,}'.format(123456789.45)) # formato numérico con separador de miles
```



Reserva de espacio

Distintas formas de alinear textos, reservar espacios, alinear por separador de miles

4.8 String templates

```
import string

plantilla = string.Template("$nombre es alumn${genero} del $ies")
print(plantilla.substitute(nombre="Alejandro", ies="IES Seritium", genero="o"))
print(plantilla.substitute(nombre="Elena", ies="IES Almunia", genero="a"))

#con diccionario
d=dict(nombre="María", ies="Alto Guadiana", genero="a")
print(plantilla.substitute(d))

# $$ escape para escribir el dolar
plantilla = string.Template("$nombre ha ganado $sueldo$$")
d =dict(nombre="Ana", sueldo=2800)
print(plantilla.substitute(d))

# prefijos con {}
plantilla = string.Template("${tipo}_nombre es el fichero cargado")
print(plantilla.substitute(tipo="01"))

# si no pongo todos los datos, tengo un error. Puedo usar safe_substitute
plantilla = string.Template("$nombre ha ganado $sueldo$$ en el año $año")
d =dict(nombre="Aurelio", sueldo=800)
# print(plantilla.substitute(d)) # da un error, porque no indico el año
print(plantilla.safe_substitute(d)) # NO da un error, pero imprime en pantalla la variable.
```

Se emplea el módulo string , que debe importarse.

Se crea una plantilla, usando \$xxx para sustituir los valores

\${xxx}yyyy para prefijos/sufijos

Se pueden usar diccionarios.

\$\$ escape para el dólar.

safe_substitute para evitar errores.

4.10 Procedimientos asociados a cadenas

```
valores = 'Denyse,Marie,Smith,21,London,UK'  
print(valores.replace(","," "))
```

Reemplaza valores

```
nombre = input("Give me your name: ")  
apellido = input("Give me your family name: ")  
nombreCompleto = nombre+ " " + apellido
```

input

```
print(nombreCompleto)  
print("La longitud de tu nombre es {}".format(len(nombreCompleto)))  
nombreCompleto= nombreCompleto.upper()  
print("Tu nombre en mayúsculas es {}".format(nombreCompleto))  
busqueda = 'Albus'  
print("¿Contiene tu nombre {}? {}".format(busqueda,  
      (nombreCompleto.find(busqueda)>0)))
```

procedimientos de
cadena

buscar

6. Números, Booleanos y None

int() → De string a entero, o de otro tipo a entero.

float() → De string a coma flotante.

Complejos: se escriben con “j”

Función bool() para boolean.

- bool(1) → True , int(True) → 1
- bool(0) → False, int(False) → 0

winner = None → asignación vacía

Operator	Description	Example
+	Add the left and right values together	1 + 2
-	Subtract the right value from the left value	3 - 2
*	Multiple the left and right values	3 * 4
/	Divide the left value by the right value	12 / 3
//	Integer division (ignore any remainder)	12 // 3
%	Modulus (aka the remainder operator)—only return any remainder	13 % 3
**	Exponent (or power of) operator—with the left value raised to the power of the right	3 ** 4

Operator	Description	Example	Equivalent
+=	Add the value to the left-hand variable	x += 2	x = x + 2
-=	Subtract the value from the left-hand variable	x -= 2	x = x - 2
*=	Multiple the left-hand variable by the value	x *= 2	x = x * 2
/=	Divide the variable value by the right-hand value	x /= 2	x = x / 2
//=	Use integer division to divide the variable's value by the right-hand value	x //= 2	x = x // 2
%=	Use the modulus (remainder) operator to apply the right-hand value to the variable	x %= 2	x = x % 2
**=	Apply the power of operator to raise the variable's value by the value supplied	x **= 3	x = x ** 3

6. IF construcciones

Operadores
comparación

Operator	Description	Example
==	Tests if two values are equal	3 == 3
!=	Tests that two values are <i>not</i> equal to each other	2 != 3
<	Tests to see if the left-hand value is less than the right-hand value	2 < 3
>	Tests if the left-hand value is greater than the right-hand value	3 > 2
<=	Tests if the left-hand value is less than <i>or</i> equal to the right-hand value	3 <= 4
>=	Tests if the left-hand value is greater than or equal to the right-hand value	5 >= 4

Operadores
lógicos

Operator	Description	Example
and	Returns True if both left and right are true	(3 < 4) and (5 > 4)
or	Returns true if either the left or the right is true	(3 < 4) or (3 > 5)
not	Returns true if the value being tested is False	not 3 < 2

```
savings = float(input("Enter how  
if savings == 0:  
    print("Sorry no savings")  
elif savings < 500:  
    print('Well done')  
elif savings < 1000:  
    print('Thats a tidy sum')  
elif savings < 10000:  
    print('Welcome Sir!')  
else:  
    print('Thank you')
```

Si (algo que sea True) " dos puntos". Se ejecuta lo que está dentro.

elif (condición
secundaria)

else, en caso de que no se cumplan las
condiciones anteriores

6. IF expressions

expression if

```
edad = int(input("Dime una edad: "))

condicion = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(condicion)

trabajo = "No puede trabajar" if edad <= 18 else ("Puede trabajar" if edad >= 18 and
edad <= 65 else "Puedes jubilarte")
print(trabajo)
```

En esta expresión no hay "elif" ; hay que hacerla de esta manera

7. Loops

Loop while

Necesario ir variando una variable que se va checando en la condición

```
count = 0
print('Starting')
while count < 10:
    print(count, ' ', end='') # part of the while loop
    count += 1                # also part of the while loop
print() # not part of the while loop
print('Done')
```

```
# Loop over a set of values in a range
print('Print out values in a range')
for i in range(0, 10):
    print(i, ' ', end='')
print()
print('Done')
```

Loop for. Necesita variar una variable en un rango. Se usa la función range

range(0,9) = {0,1,2,3,4,5,6,7,8} El "9" es la cota máxima.

range(0,9,2) = {0,2,4,6,8} cuenta de 2 en dos.

7. Loops: variable anónima , break, continue, else

```
# imprime un punto 10 veces
for _ in range(0,10):
    print(".",end="")
print()
```

Se puede usar un for con variable anónima si no se necesita que la variable influya en las instrucciones iteradas. Se utiliza un wildcard (barra baja)

```
numero = int(input("Dime un número: "))
for i in range(1,numero+1):
    if i%2==0:
        continue
    if i%13==0 and i>13:
        break
    print("Este es un número impar
{numero}.".format(numero=i))
else: # el else se ejecuta si llega a ejecutarse el FOR
    completamente.
    print("He impreso todos los números impares hasta
llegar al {}".format(numero))
print("He terminado")
```

Fuerza a nueva iteración

Rompe el bucle

Se ejecuta else en caso de que se haya completado el bucle

9. Recursividad

Funciones que se llaman a sí mismas. Si se usan deben terminar en algún punto porque: a) encuentran una solución, b) el problema es pequeño, o trivial, y se soluciona (caso base) o c) alcanzado un nivel de recursividad sin solución, terminan.

```
# cálculo del factorial mediante recursividad
```

```
def factorial(n, depth=1):
```

```
    if n==1:
```

```
        print("\t"*depth, "Retorno 1")
```

```
        res = 1
```

```
    else:
```

```
        print('\t' * depth, 'Llamando al factorial recursivo(',  
n-1, ')')
```

```
        res = n*factorial(n-1,depth+1)
```

```
        print('\t' * depth, 'Devolviendo:', res)
```

```
    return res
```

```
num = int(input("Dime un número entero positivo: "))
```

```
print (factorial(num))
```

Caso base: terminación

Parte recursiva

9. Recursividad en cola (tail recursion)

Como la recursividad consume muchos recursos comparados con estructuras iterativas, y pueden dar problemas de memoria, se utiliza la **recursividad en cola**

```
def tail_factorial(n, accumulator=1):  
    if n==0:  
        return accumulator  
    else:  
        return tail_factorial(n-1, accumulator*n)  
  
num = int(input("Dime un número entero positivo: "))  
print(tail_factorial(num))
```

Caso base: terminación

Parte recursiva. El resultado se pasa a la función ANTES de la llamada

9. Triángulo de Pascal

```
def calcularcoeficiente(f,c):  
    """  
    :param f: int que representa la fila  
    :param c: int que representa la columna  
    :return: de forma recursiva, el valor del coeficiente en la posición  
    fila, columna  
    Si la columna es cero o coincide con el número de fila , devuelve 1.  
    Si no, se calcula sumando el coeficiente en la fila anterior y la  
    columna anterior  
    más el coeficiente de la fila anterior y misma posición  
    0 1 2 3 4..  
    fila=0 1  
    fila=1 1 1  
    fila=2 1 2 1  
    fila=3 1 3 3 1  
    fila=4 1 4 6 4 1  
    """  
    if c==0 or c==f:  
        return 1  
    else:  
        return calcularcoeficiente(f-1,c-1)+calcularcoeficiente(f-1,c)
```

Función recursiva que
calcula el coeficiente

Función q ue calcula una fila

```
def calcularFila(fila):  
    """  
    :param fila: número de la fila  
    :return: una cadena centrada con los valores del triángulo de  
    Pascal hasta dicha fila, y separado, el  
    valor de la suma de todos los valores  
    El bucle "i" calcula la suma y los coeficientes de una fila  
    El bucle "j" construye las filas desde la cero a la "fila" dada  
    """  
    for j in range(0,fila+1):  
        cadena=""  
        suma= 0  
        for i in range(0,j+1):  
            suma = suma + calcularcoeficiente(j,i)  
            cadena = cadena + str(calcularcoeficiente(j,i)) + " "  
        print("{:^80}={:<10}".format(cadena,suma))
```

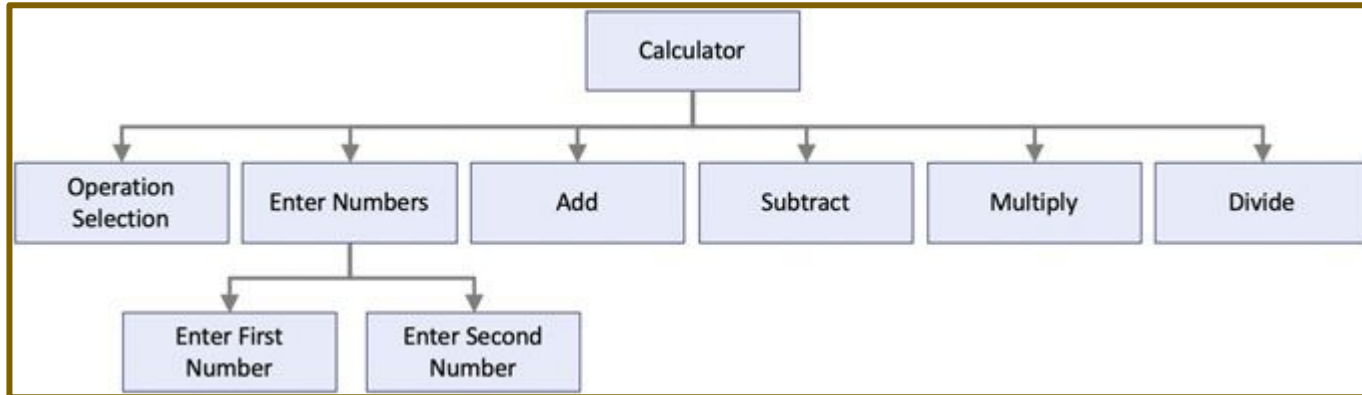
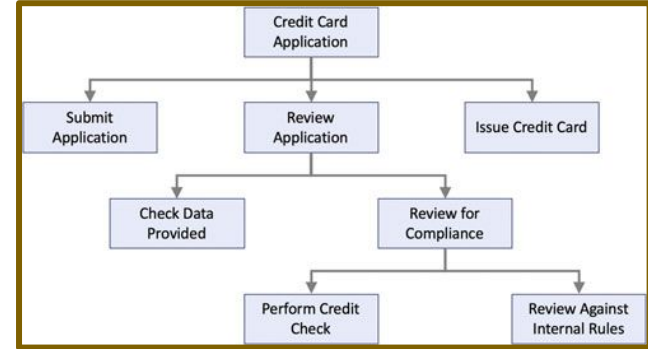
```
print(calcularcoeficiente.__doc__)  
print(calcularFila.__doc__)  
calcularFila(15) #Calcula el triángulo de  
Pascal hasta la fila 15.
```

10. Análisis estructural

Top-down representación de funciones (jerarquizado)

Las funciones pueden ser de más alto nivel (que tienen subfunciones), subfunciones (realiza una acción determinada de la función de más alto nivel) y funciones básicas, que no tienen sub-funciones.

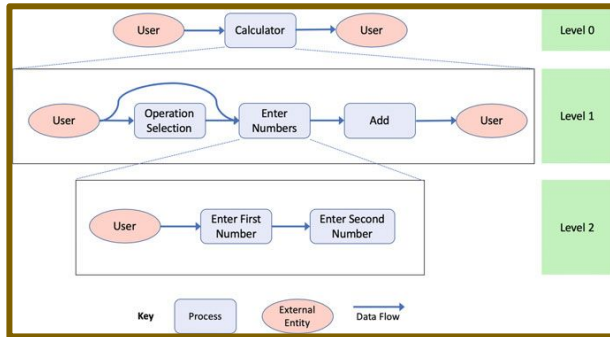
Top-down representación de funciones



**Functional
Descomposition**

10. Análisis estructural (Functional Flow)

Representan no sólo la jerarquía de funciones sino el flujo de datos entre ellos. Tres tipos: pseudocódigo, diagramas data flow (flujo de datos) y sequence diagrams (diagramas de secuencia).

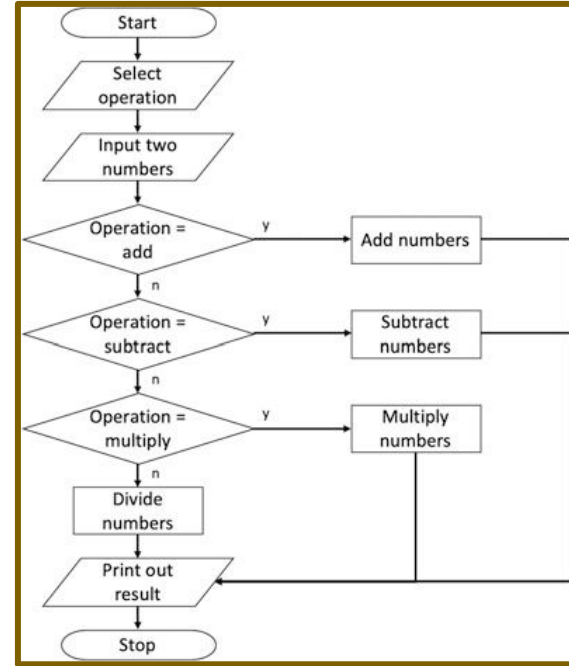
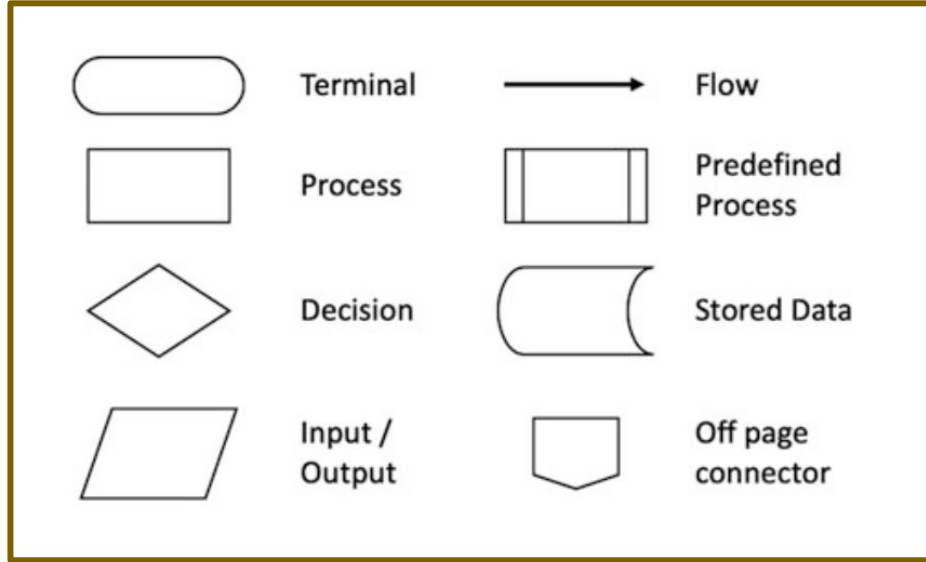


Data Flow

1. Proceso: entradas, salidas
2. Data flow: transferencia de datos de un elemento a otro. Lleva una dirección
3. Data store/warehouse: flujo de leer/guardar datos
4. Terminación: ser humano, otra máquina

10. Flowcharts o diagramas de flujo

Forma de representar un algoritmo

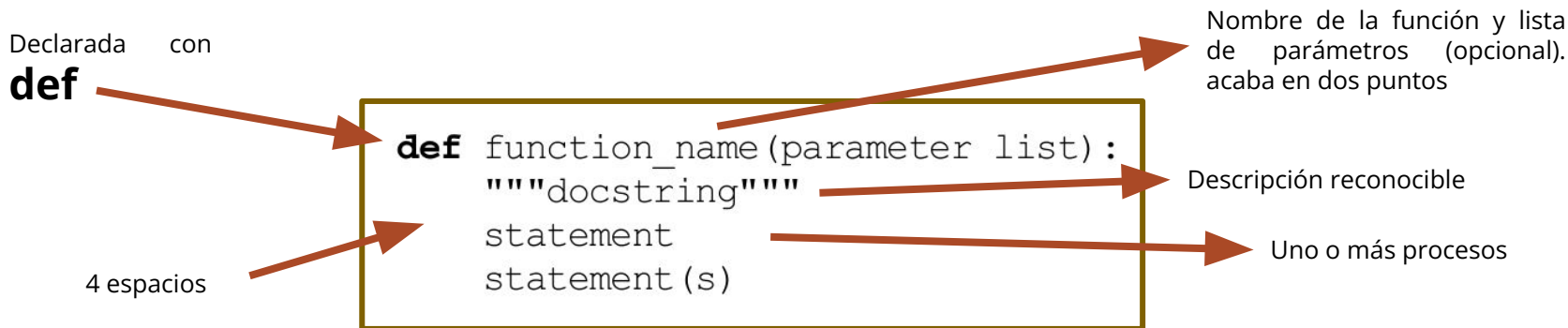


Data dictionary: repositorio estructurado de elementos de datos en el sistema. Almacena detalles y descripciones de todos los datos del diagrama de flujo. Hoja de cálculo o <https://www.semantacorp.com/data-dictionary>

11. Funciones en Python

¿Cómo funciona? El programa salta a la función cuando es invocada. Cuando termina la función vuelve al punto de inserción inmediatamente posterior.

Hay funciones predefinidas (built-in functions) y definidas por el usuario.



1. Puede acabar en **return** y devolver uno o varios valores: **return a // return a, b, c**
2. **docstring** puede ser invocado **print(function_name.__docstring__)**

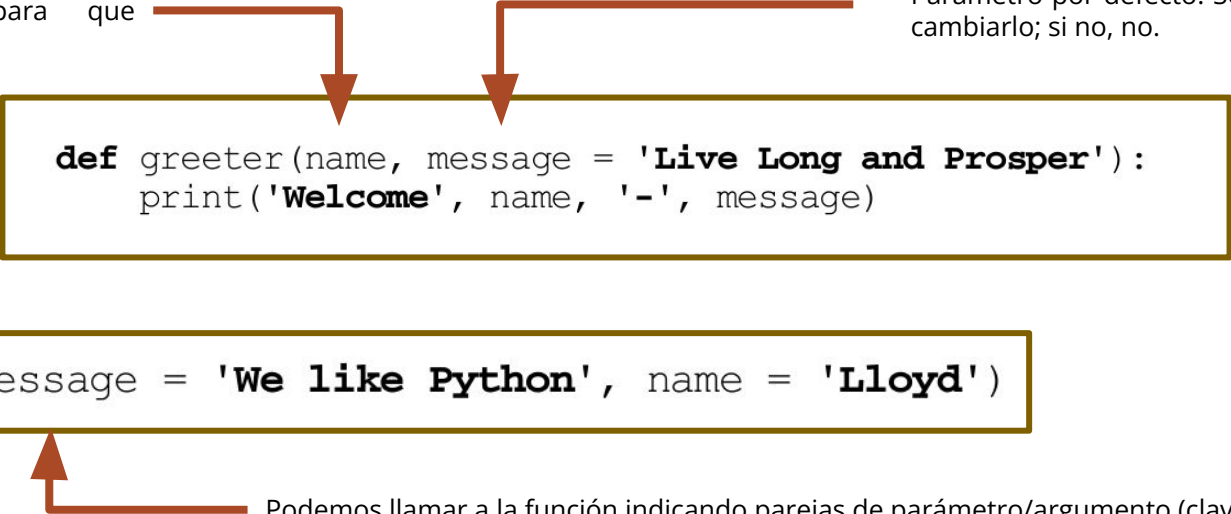
11. Funciones en Python (parámetros/argumentos)

- **Parámetros:** son las variables definidas dentro de la cabecera de la función, entre el paréntesis.
- **Argumentos:** los valores concretos que se les pasa a una función a través de sus parámetros.

No son conceptos iguales, aunque a veces se usan indistintamente.

Parámetro obligatorio. Hay que invocarlo para que funcione

Parámetro por defecto. Se invoca si hay que cambiarlo; si no, no.



```
def greeter(name, message = 'Live Long and Prosper'):  
    print('Welcome', name, '-', message)
```

```
greeter(message = 'We like Python', name = 'Lloyd')
```

Podemos llamar a la función indicando parejas de parámetro/argumento (clave/valor).

11. Funciones en Python (argumento arbitrario)

- **Argumento arbitrario (*arg):** lista de argumentos de los que desconozco su número. No son conceptos iguales, aunque a veces se usan indistintamente.

```
def greeter(*args):  
    for name in args:  
        print('Welcome', name)  
  
greeter('John', 'Denise', 'Phoebe', 'Adam', 'Gryff', 'Jasmine')
```

Esta función espera un argumento **arg** de número indeterminado de elementos.

```
def misNombres2(*mn, **otros):  
    for i in mn:  
        print("Nombres: ", i)  
    for key in otros:  
        print("Clave: ", key, " - Valor:", otros[key])  
  
misNombres2("Aurelio", "Maricarmen", "Luis", "Juan",  
            hijo="Alberto", hija="Luisa", nieto="Alfredo")
```

Con un asterisco, **mn**, es una lista posicional. Los elementos se obtienen posicionalmente.

Con dos asteriscos, **otros**, la lista es del tipo clave-valor.

Los argumentos son pasados o bien separados por comas tal cual (mn) o con claves (otros)

11. Funciones anónimas

- Funciones que se crean con la palabra reservada **lambda**, que se usan una vez ¿?, donde son creadas y no tienen nombre.
- Nomenclatura → **lambda argumento: expresión**
- **Funciones que suelen devolver valores, suelen ser numéricas.**

```
# Funciones anónimas, lambda

# Notación: lambda argumento1, argumento2,... : expresion
# Retornan un valor que se asigna a una variable

cuadrado = lambda x: x * x
modulo = lambda x, y: (x * x + y * y) ** 0.5
print(cuadrado(14))
print(modulo(3, 4))

# ¿Reutilizables? Sí ,lo son
print(cuadrado(100))
print(modulo(9, 4))
```

**Funciones
anónimas y
sus llamadas**

12. Variables locales y globales

- **Globales:** tienen el programa entero como alcance.
- **Locales:** alcance en una función. Cuando la función acaba, la variable se destruye.

```
# diferencia entre variables globales y locales

def mi_func():
    a = 100 # defino la variable como local
    print(a) # La función imprime la versión local

a = 5 # Defino la versión global de la variable
mi_func()
print(a) # imprimo la versión global
```

Variable local



Variable global

12. Variables locales y globales

```
# Problema de las variables globales y locales

def mi_func():
    global mm # Defino la variable global mm, es decir, tomará
    el valor global antes definido mm=5
    # Si comento esta lineal global, incluso indica
    error, porque mm no estaría definida
    mm = mm + 1 # sumo uno.
    print(mm) # Imprimirá la variable global

mm = 5
print(mm) # Devuelve 5
mi_func() # sumará 1 a mm y devolverá 6
print(mm) # No cambia, devuelve 6
```

Si quiero usar una variable global dentro de la función, deberé especificarlo con "global" . Si no lo hago, obtendré un error.

12. Variables locales dentro de funciones.

```
# Definir funciones DENTRO de otra función
# Las variables son siempre LOCALES, dentro de cada función.
# Si quiero usar una variable en una función interior, dentro de otra función, ya no
puedo llamarlas
# global, tengo que hacerlo nonlocal
def exterior():
    title = "Título exterior"

    def interior():
        nonlocal title # Comentar, descomentar esta línea.
                        # Comentada, cada función toma la definición de su variable.
                        # Descomentada, title puede ser modificada DENTRO de la
función anidada, con lo que cambiaría también en la exterior
        title = "Título interior"
        print(title)

    interior()
    print(title)

exterior()
```

Puedo usar funciones definidas dentro de otras funciones, que tienen su alcance dentro de éstas. Si quiero usar una variable de la función “exterior” dentro de una función más interior, deberé usar la palabra reservada **nonlocal**

14. Functional programming (FP)

Paradigma de programación en el que:

- Se escribe para evitar efectos colaterales
- Se prohíbe modificar otras partes del programa (no hay estados o variables globales)
- Únicas salidas observables, las salidas de las funciones.
- Las únicas dependencias de esas salidas son los argumentos de la función
- Los argumentos son totalmente determinados antes de que la salida se genere.
- Sin estados, el software es más fácil de entender, implementar, testear y depurar.
- Se promueven datos inmutables que aseguran que los datos no se cambian una vez creados.
- Se promueve la programación declarativa (expresiones que describen la solución) antes que una programación imperativa donde predominan los procedimientos.

```
int sizeOfContainer = container.length  
for (int i=1 to sizeOfContainer) do  
  element = container.get(i)  
  print(element)  
end do
```

**lenguaje
imperativo**



**Functional
Programming**

```
container.foreach(print)
```


14. Functional programming (FP)

Python da soporte para escribir programas en estilo funcional, que son muy útiles cuando, por ejemplo se procesan varios tipos diferentes de datos.

Si se usa bien, la programación funcional reporta grandes beneficios.

En la programación funcional **nos centramos en lo que el programa necesita hacer** más que en lo que hace en sí.

Es necesaria la **transparencia referencial (TR): a unos valores de entrada, siempre la misma salida**. Son efectos colaterales de este principio el uso de variables globales o modificar características del programa fuera de las funciones, lo cual compromete la TR. A veces estos efectos colaterales permanecen ocultos.

Ventajas: menos código, conseguir la TR, uso de la recursividad como estructura de control natural, buena para hacer soluciones prototipadas, hay una modularidad, evita los estados, estructuras de control “aditivas”, uso de datos inmutables, sistemas concurrentes (no afectan unos a otros de forma adversa), se pueden evaluar las funciones parcialmente.

Desventajas: el flujo es más difícil, las aplicaciones interactivas más difícil de desarrollar, o los programas como servicios o controladores ,menos eficientes en plataformas de hardware, no son orientados a datos, menos intuitivos, programadores menos familiarizados con la FP...

15. Higher order functions (funciones de más alto nivel)

Una función se llama:

- **Con paréntesis y argumentos:** para ejecutarse
- **Sin paréntesis ni argumentos:** devuelve la información de la dirección de memoria donde empieza a ejecutarse.

El nombre de una función no es más que una variable con una referencia (puntero) almacenada a una dirección de memoria donde se empieza a ejecutar la función.

```
# Funciones de alto nivel
```

```
def mensaje(msg):      # function header
    msg = "*** "+msg+" ***"
    return msg          # function body
```

```
def mensaje2(msg):     # function header
    msg = "=== "+msg+" ==="
    return msg          # function body
```

```
print(mensaje("Hola Mundo"))  # imprime el mensaje
print(mensaje)                # sin paréntesis, implica la posición de
                              # memoria donde se empieza ejecutar la función
                              # Esta llamada trata a la función como un objeto. De
                              # hecho...
print(type(mensaje))          # imprimirá la clase function

otra_referencia = mensaje     # en la variable otra_referencia
                              # asigno la referencia a la función mensaje.
                              # De esta manera, la misma función tiene dos variables
                              # referenciadas.
print(otra_referencia("Otra frase"))

# También podría tener un segundo mensaje, y reasignarlo a
# la referencia original
print(mensaje("Mensaje con la referencia a la función SIN
CAMBIAR"))
mensaje = mensaje2
print(mensaje("Mensaje con la referencia a la función
CAMBIADA"))
```

15. Higher order functions (funciones de más alto nivel)

```
def comprobar(s):
    if s == "par":
        return lambda n: n % 2 == 0
    elif s == "positivo":
        return lambda n: n >= 0
    elif s == "negativo":
        return lambda n: n < 0
    else:
        raise ValueError("Objeto desconocido")

def sumando():
    def suma(x, y):
        return x + y

    return suma

# asignamos a variables distintas funciones lambda retornadas
f1 = comprobar("par") # función par
f2 = comprobar("positivo") # función positivo
print(f1(3), f2(3))

# asignamos a variable una función definida con nombre
ff = sumando() # asigna a ff la función suma
print (ff(4,5))
```

Una función puede retornar funciones como objetos: las funciones pueden retornar funciones.

Las funciones f1 y f2 son funciones que retorna la función **comprobar** según un parámetro s.

La asignación de la función **ff** es la función "suma" a través de otra llamada "sumando". Por eso ff acepta dos parámetros.

15. Funciones como parámetros

```
# Funciones que son parámetros de otras funciones
```

```
def aplicar_tasa(x, mi_funcion):  
    """ Función principal de orden superior """  
    return mi_funcion(x)
```

```
def mi_tasa(x):  
    """ aplico una tasa """  
    return x * 0.3 + 10
```

```
def gob_tasa(x):  
    """ aplico una segunda tasa """  
    return x * 0.45 - 10
```

```
ganancia = 30000  
print(aplicar_tasa(ganancia, mi_tasa))    # llamo a la función de orden superior y le digo que función  
quiero implementar  
print(aplicar_tasa(ganancia, gob_tasa))  
# Las funciones de orden superior escriben un código claro cuando no sé aún la función exacta a aplicar.  
# Siempre puedo, simplemente, cambiar la función a aplicar en la de orden superior.
```

Aparentemente esta construcción es redundante. Pero la función de orden superior “aplicar_tasa”, que es la que se aplica en el programa principal, permite escribir el programa.

Posteriormente en un futuro, puede modificarse la función que llama (mi_tasa, gob_tasa...) o añadir funciones diferentes nuevas, que se podrán llamar con poca modificación del programa principal.

16. Carrying Functions

```
# Funciones tipo "currying" por Haskell Curry
# el objetivo es reusar funciones fijando algunos parámetros: funciones derivadas unas de
# otras.

def multiply(x, y):
    """ Esta función multiplica dos valores """
    return x * y

def multby(func, num):
    """ Esta función toma como parámetros una función y un número
    y devuelve una nueva función con un parámetro fijado y un nuevo parámetro y
    que hay que proporcionar """
    return lambda y: func(num, y)

double = multby(multiply, 2) # esta función resulta de asignar el número 2 como fijo en la
# función multiply pero aún debe proporcionar un número.
triple = multby(multiply, 3)

print(multiply(4, 5))
print(double(44))
print(triple(44))
```

Funciones que pueden llamarse para crear y modificarse para crear otras funciones.

16. Carrying Functions (closures)

Concepto de closure:

Algunos parámetros de una función definida dentro de otra función pueden permanecer accesibles en ciertas partes del programa (scope) cuando son invocadas.

Un closure sería la unión de una función (más bien de la referencia a esa función) y de su entorno de referencia, incluyendo las variables no-locales a esa función que tienen que ser invocadas cuando se llama.

17. Clases. Programación orientada a objetos PPO

Paradigma de programación que consiste en:

Clase: representa un tipo de objetos o entes, reales o abstractos.

Objeto o instancia: un ejemplar concreto de una clase.

Atributo o campo: cada una de las características (datos) de una clase de objetos.

Métodos: comportamientos definidos que tienen los objetos de una clase.

Mensaje (message): requerimiento a un objeto para que muestre un atributo o lleva a cabo un método.

Las clases se basan pues en datos, que intercambian entre sí mensajes. Se pone el énfasis en la funcionalidad del sistema.

18. Clases. Programación orientada a objetos PPO

```
class Person:
```

```
    """ Esta clase representa a una persona """
```

Definición de clase. Docstring inmediatamente inferior

```
def __init__(self, name, age):
```

```
    """ Definición de atributos. CONSTRUCTOR """
```

```
    self.name = name
```

```
    self.age = age
```

SELF: llamada a sí mismo

CONSTRUCTOR. Se encarga de incorporar los datos al objeto en sí. Se llama cuando se instancia.

```
def __str__(self):
```

```
    """ Método por defecto de imprimir el objeto """
```

```
    return self.name + ' tiene ' + str(self.age) + " años"
```

`__str__` método que devuelve el resultado cuando se invoca la orden `print(instancia)`

```
def cumple(self):
```

```
    print("¡Felicidades! Hoy es tu cumpleaños")
```

```
    print("Tenías {antes} años, y ahora cumples {ahora}".format(antes=self.age, ahora=self.age + 1))
```

```
    self.age += 1 # Actualizo la información de la edad
```

```
def pagos_segue_horas(self, horas):
```

```
    pagar_la_hora_a = 30
```

```
    if self.age >= 18:
```

```
        pagar_la_hora_a += 10 # si es mayor de 18 años, pagarlas a 10 € más cara
```

```
    return pagar_la_hora_a * horas
```

Métodos definidos por el programador.

```
def es_mayor_de_edad(self):
```

```
    return self.age >= 18
```


18. Clases. Programación orientada a objetos PPO

- Si tengo dos instancias **p1** y **p2** , y hago `px = p1`, lo que hago es referenciar en `px` la dirección de memoria donde se encuentra `p1`. `px` no es el mismo objeto que `p1`, pero apuntan al mismo sitio.
- Puedo borrar un objeto con `del (p1)` o bien haciendo **p1 = None**.
- Si se define el método `__str__` se consigue una muestra en pantalla particularizada al llamar a `print()`.
- En Python, existe un proceso llamado **garbage collection**. Python automáticamente gestiona la memoria incluso borrando partes no usadas.

Atributos intrínsecos, tanto de clases como de instancias u objetos.

```
""" Atributos intrínsecos """
print('Class attributes')
print(Person.__name__)
print(Person.__module__)
print(Person.__doc__)
print(Person.__dict__)
print('Object attributes')
print(p2.__class__)
print(p2.__dict__)
```

18. Class side and static behaviour

- **Datos y comportamientos** que pertenecen a la clase, pero no de un objeto u instancia concreto
- **Class side data:** son variables que pertenecen a una clase, pero están fuera de cualquier método.
- **Class side methods:** métodos que pertenecen a la clase, como cualquier otro método, pero se decoran con la palabra **@classmethod**. Definen el comportamiento de la clase en sí, no de sus objetos o instancias individuales. Sirven para contadores, responder cuestiones sobre la clase, testeo de instancias, ayudas, etc.

```
class Cuenta:
    """ Esta clase representa la cuenta bancaria de una persona """
    contador = 0 # creo la variable «contador» que lleva el registro de cuentas creadas
    # es una variable de clase (class side data)

    @classmethod
    def cuentas_creadas(cls):
        """ Este método de clase añade uno al contador y devuelve ese valor. Será modificado
            cada vez que se instancia o crea un objeto """
        Cuenta.contador += 1
        return Cuenta.contador

    def __init__(self, num, propietario, inicial, tipo):
        self.num = num
        self.propietario = propietario
        self.balance = inicial
        self.tipo = tipo
        print("Cuenta n° {} creada".format(Cuenta.cuentas_creadas()))
```

Variable dentro de una clase

Método para la clase

Invocar método de clase

18. Método estático

- **Usado para** insertar una función dentro de una clase de forma independiente.

```
@staticmethod
def static_function():
    """ Son funciones independientes dentro de una clase. Se llamaría con
    Person.static_function() """
    print("Esto es una función estática")
```

20. Class inheritance (herencia de clases)

- **Consiste que de una clase puede inferirse otra que herede de la anterior todos sus atributos y métodos.**
- Por ejemplo, de la clase PERSON, puede heredarse otra llamada EMPLEADOS. Se dice que la clase EMPLEADOS extiende la clase PERSON
- Se puede hacer una subclase de la subclase que se desee.

```
class Empleado(Person):  
  
    def __init__(self, nombre, edad, Id):  
        """ CONSTRUCTOR de la clase Empleado."""  
        super().__init__(nombre, edad) # Forma de llamar a inicialización de  
la clase Person. Poner siempre al principio  
        self.id = Id # Atributo de clase nuevo  
  
    def calcular_paga(self, horas_trabajadas):  
        rate = 7.5  
        if self.age > 18:  
            rate += 2.5  
        return rate * horas_trabajadas
```

Clase heredada de otra. En la definición se pasa la clase padre como argumento

Método super. Forma de llamar a la clase PADRE

Los métodos y atributos de la clase PADRE se conservan

20. Class inheritance (herencia de clases)

- Si una **clase** es una combinación de datos y procedimientos que operan con esos datos...
- Una **subclase** es una clase que hereda atributos y métodos de otra clase.
- **Superclase** es la clase padre de otra.
- En python, una **subclase** puede heredar de varias clases
- Hay dos tipos de jerarquías; la jerarquía “**is-a**” (por ejemplo, STUDENT is-a PERSON) es una jerarquía entre clases (**HERENCIA**). La otra es “**is-part-of**”. En esta jerarquía (por ejemplo, ENGINE is-part-of a MOTOR) nos referimos más a instancias. Decimos que es una “instantiation”.
- Una subclase debe modificar o extender el comportamiento de su padre, añadir comportamiento adicional o modificar el comportamiento de los métodos. Si no usa los atributos del padre, quizás esté mal pensada.

```
class Person(object):
```


En general, todas las clases heredan de otra clase general llamada **object** (que suele omitirse). Es por ello, que todas las clases heredan métodos especiales (`__init__`, `__eq__`, `__hash__`) y atributos intrínsecos (`__doc__`, `__dict__`, `__class__`, `__module__`)

20. Overriding methods y extending superclass methods

- Es usual que una subclase vuelva a definir uno de los métodos de su padre (habitualmente `__init__`). Esto se conoce como **overriding method**. Podría traducirse por superponer el método.
- Es habitual también que una subclase extienda el método ya usado en el padre. Por ejemplo, en la clase `VENEDORES` se extiende el método de imprimir del padre `EMPLEADOS`

```
class Vendedor(Empleado):  
    """ Subclass e de Empleado (a su vez de Person) que define los vendedores """  
  
    def __init__(self, nombre, edad, Id, ventas, region):  
        super().__init__(nombre, edad, Id)  
        self.ventas = ventas  
        self.region = region  
  
    def bonus(self):  
        return self.ventas * 0.5  
  
    def __str__(self):  
        """ Método por defecto de imprimir el objeto.  
        La clase <<Vendedor>> OVERRIDE el método de impresión ya definido por Empleado """  
        # return (self.name + ' (' + str(self.id) + ') tiene ' + str(self.age)  
        #         + " años" + " y lleva la zona de "+self.region)  
        return super().__str__() + " y lleva la zona de "+self.region  
        # mejor se usa la extensión de los métodos de los padres.
```

Extendiendo el método de impresión de la clase padre EMPLEADO

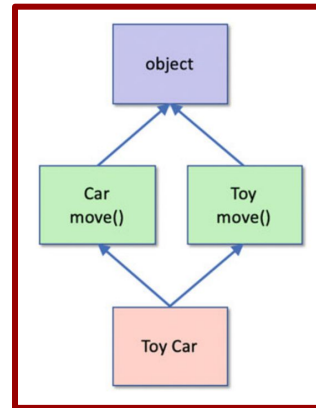


20. Inheritance Oriented Naming Conventions

- Nombres que empiezan por una barra baja **_name** , son privadas para las clases y accesibles para las subclases.
- Si empiezan por dos barras bajas **__name**, son privadas para las clases e inaccesibles para otras clases y subclases. No se pueden invocar fuera de la clase.
- Si Python encuentra algún nombre con dos barras bajas, **__somename**, inmediatamente lo sustituye (name mangling) por **_classname_somename**, y así da soporte a métodos que empiezan con dos barras bajas.

20. Multiple Inheritance

- Es posible que una clase herede de dos clases padres o más.
- Si hay métodos duplicados en una clase padre y otra, se aplica el de la clase que se coloca primero.
- `class cochecito(coche, juguete) :` → preferencia para los métodos de coche.
- La herencia múltiple puede ser útil sólo cuando las clases padres no están relacionadas. Si no, pueden dar lugar a muchas inconsistencias. En general hay que tener cuidado con las clases múltiples.



22. Operator overloading

- En una clase, puede tener sentido (o no) definir operaciones matemáticas "+, -, x, /..." , operaciones de comparación "=",!=",<,>..." u operaciones lógicas como and y or. Sólo usar las necesarias.
- Por ejemplo, en una clase PERSON, p1 = Person("Luis") y p2=Person("Ana"). ¿Qué sentido tiene p1 + p2? A lo mejor en esa clase no tiene sentido definir esa operación. A lo mejor para otra clase que sean cantidades numéricas , sí lo tiene.

Numerical operator

Operator	Expression	Method
Addition	$q1 + q2$	<code>__add__(self, q2)</code>
Subtraction	$q1 - q2$	<code>__sub__(self, q2)</code>
Multiplication	$q1 * q2$	<code>__mul__(self, q2)</code>
Power	$q1 ** q2$	<code>__pow__(self, q2)</code>
Division	$q1 / q2$	<code>__truediv__(self, q2)</code>
Floor Division	$q1 // q2$	<code>__floordiv__(self, q2)</code>
Modulo (Remainder)	$q1 \% q2$	<code>__mod__(self, q2)</code>
Bitwise Left Shift	$q1 \ll q2$	<code>__lshift__(self, q2)</code>
Bitwise Right Shift	$q1 \gg q2$	<code>__rshift__(self, q2)</code>

Comparison operator

Operator	Expression	Method
Less than	$q1 < q2$	<code>__lt__(q1, q2)</code>
Less than or equal to	$q1 \leq q2$	<code>__le__(q1, q2)</code>
Equal to	$q1 == q2$	<code>__eq__(q1, q2)</code>
Not Equal to	$q1 != q2$	<code>__ne__(q1, q2)</code>
Greater than	$q1 > q2$	<code>__gt__(q1, q2)</code>
Greater than or equal to	$q1 \geq q2$	<code>__ge__(q1, q2)</code>

logical operator

Operator	Expression	Method
AND	$q1 \& q2$	<code>__and__(q1, q2)</code>
OR	$q1 q2$	<code>__or__(q1, q2)</code>
XOR	$q1 \wedge q2$	<code>__xor__(q1, q2)</code>
NOT	$\sim q1$	<code>__invert__()</code>

23. Python properties

- **Encapsulación:** procedimiento por el cual una clase oculta algunos de sus atributos y métodos. Esos métodos y atributos no son accesibles a menos que el programador abra ciertas puertas a ellos.
- Python no es exactamente encapsulado. Posee una convención estándar que indica que un procedimiento o atributo debe ser privado (precederlo de una barra baja o guión bajo).

```
class Person:
    """ Esta clase representa a
    una persona """

    def __init__(self, name, age):
        """ Definición de
        atributos. CONSTRUCTOR """
        self.name = name
        self._age = age
```

Los atributos `_name` y `_age` son considerados privados.

NOTA: el programador puede aún cambiar y modificar los atributos privados, pero que tengan la barra baja indica que quien los diseñó por primera vez consideraba que eran privados. Por lo tanto, si otro programador los modifica y la modificación no funciona no puede achacarle al primero que no los hubiera diseñado como "intocables".

- Si los atributos son privados, ¿cómo puedo acceder a ellos? El programador puede establecer métodos especiales de lectura (getter), escritura (setter), borrado o documentación.

23. Python properties

- Forma de implementar getter, setter, borrado y doc de un atributo **radius** en la clase **Circle**. Indicamos con el procedimiento **property** qué función es cada cual.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    def _get_radius(self):
        print("Get radius")
        return self._radius

    def _set_radius(self, value):
        print("Set radius")
        self._radius = value

    def _del_radius(self):
        print("Delete radius")
        del self._radius

    radius = property(fget=_get_radius, fset=_set_radius, fdel=_del_radius, doc="The radius property.")

c1 = Circle(48)

c1.radius = 67
print(c1.radius)
```

property. Considerado
sin embargo obsoleto.



23. Python properties (usando decoradores)

- Forma de implementar getter, setter, borrado y doc de un atributo **age** en la clase **Person**. Indicamos con decoradores.

```
@property
def age(self):
    """ Esta propiedad almacena la edad """ # actúa como
    return self._age
```

Getter. Decorado con @property, y su docstring es el doc del atributo

```
@age.setter
def age(self, valor):
    """ Este es el setter. Se escribe IGUAL pero con la p
    if isinstance(valor, int) and 0 < valor <= 120:
        self._age = valor
    else:
        print("Edad no válida. No se modifica")
```

Setter. Decorado con @[atributo].setter. Opcional. Si existe, se está reconociendo que el atributo es de escritura.

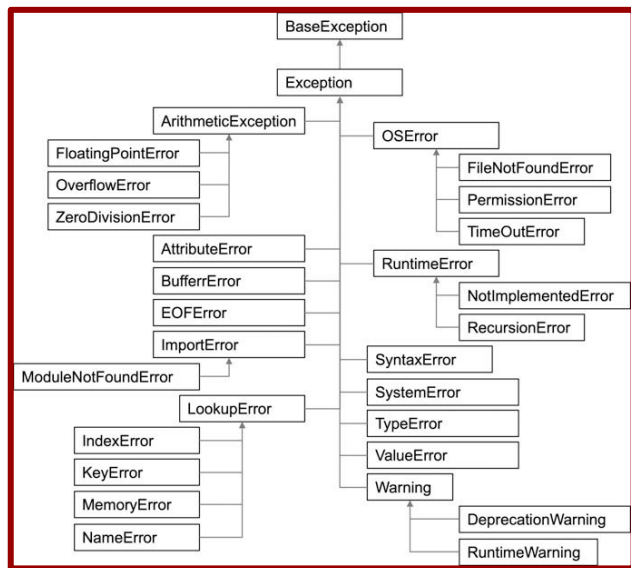
```
@age.deleter
def age(self):
    """ Este es el deleter. Se e
    pass
```

Decorado con @[atributo].deleter. Opcional. Si existe, se provee un procedimiento para el borrado de una instancia de esa clase.

eter"""

24. Errores y manejo de excepciones

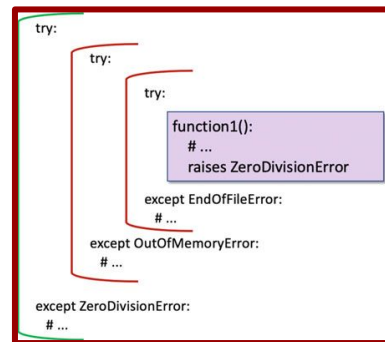
- **Errores** \longleftrightarrow **Excepciones** son intercambiables, pero llamamos excepciones a problemas con las operaciones y errores a fallos asociados con el uso, como un fichero no encontrado.
- Una excepción es **un objeto**. Tenemos la clase superior **BaseException** y todos los errores y excepciones se heredan jerárquicamente de ésta. Hay otras subclases como la **Exception** (definidas por el usuario y muchas predefinidas) y las **ArithmeticException** (excepciones precargadas asociadas a errores matemáticos).



Una excepción cambia el flujo de control del programa a un lugar donde puede ser tratado y presentado al usuario para su corrección.

- **Exception: error generado en ejecución.**
- **Raising an exception: se genera una instancia de excepción.**
- **Throwing an exception: se dispara una excepción generada.**
- **Handling an exception: se procesa código que trata con el error**
- **Handler: código que trata el error**
- **Signal: Tipo de excepción particular (out of bounds/divide by zero)**

Las excepciones son objetos de una clase que se instancian cuando se levantan (raising o throwing). El sistema busca un manejador (handler) que pueda lidiar con la excepción, bien remediando algo o terminando la ejecución del programa. La excepción puede pasar por diversos bloques handlers hasta llegar a uno que sea capaz de tratarla.



24. Errores y manejo de excepciones

```
# manejando errores. Algo más
```

```
def runcalc(x, y):
```

```
    """Esta función dará como resultado una excepción del tipo división entre cero"""
```

```
    result = x / y
```

```
    return result
```

```
try:
```

```
    print(runcalc(6, 0))
```

```
except ZeroDivisionError as exp:
```

```
    """ Si no se escribe nada, el sistema tra
```

```
    """ Puede ser de la subclase ZeroDivision
```

```
    print(exp)
```

```
    print("Estás intentando dividir entre cero")
```

```
except Exception as eee:
```

```
    """ Esta es una forma más general. Sin embargo PyCharm me advierte que debería  
    ser más específico"""
```

```
    print(eee)
```

```
    print("Error más genérico")
```

```
else:
```

```
    print("Todo fue fantástico. Me ejecuto si todo fue bien")
```

```
finally:
```

```
    print("Esto se ejecutará de cualquier forma")
```

capturar objeto
excepción

Bloque try

Excepción: qué hacer cuando ciertas instancias de
error ocurran

ELSE: sólo se ejecuta si no hubo excepciones

FINALLY: se ejecuta siempre, haya o no excepciones. Se
usa para limpiar recursos, cerrar ficheros, etc.

24. Errores y manejo de excepciones

- Cuando se produce (raise) una excepción, inmediatamente se arroja (throw) a su manejador (handler). El código inmediatamente después de la instrucción que generó el error **NO SE EJECUTA**.
- Se puede provocar o levantar una excepción con la orden **raise**:

```
def mi funcion():  
    print("Empiezo la función")  
    # raise ValueError("!Bang") # provoco una excepción  
    raise ValueError # que puede escribirse simplemente así, sin pasarle un valor  
    print("Acabo la función") # Me avisa pycharm que este código no puede alcanzarse
```

```
class InvalidAgeException(Exception):  
    """Subclase personalizada de la clase exception"""  
  
    def __init__(self, valor):  
        self.valor = valor  
  
    def __str__(self):  
        return "Edad no válida. Se provoca una excepción:" + str(self.valor)
```

Custom exception: clase personalizada que deriva de la clase Exception que puede usarse para manejar errores de forma personalizadas

24. Errores y manejo de excepciones (chaining exceptions)

```
class DividiendoEntreCero(Exception):  
    """ Clase customizada """  
  
    def __str__(self):  
        return "Aquí se ha dividido entre cero"  
  
def mi_funcion(x, y):  
    try:  
        return x / y  
    except Exception as e:  
        raise DividiendoEntreCero from e # forma de concatenar dos excepciones  
  
def main():  
    print(mi_funcion(4, 0))  
  
main()
```

CHAINING EXCEPTIONS: concatenando dos excepciones. Tratamos una excepción más general como otra más customizada. Palabra reservada **from**

25. Módulos y paquetes

- **Módulo:** agrupan juntas funciones relacionadas, clases y código en general. Como una biblioteca de código. Útil cuando el código es grande o se pretende reusar.
- El uso de módulos simplifica los programas, su mantenimiento y sus pruebas. Facilita el reuso de código...
- Un módulo equivale a un fichero python (p. ejemplo, el módulo **utils** corresponde con el fichero **utils.py**) y puede contener clases, funciones, variables, código ejecutable - que se ejecuta al ser instanciado por primera vez - y atributos asociados al módulo. También puede empezar con un texto descriptivo (a veces, extenso).
- Se puede importar:
 - **import modulo1** (si hubiese más módulos en una nueva línea o separados por comas)
 - **from modulo1 import ***
- La diferencia entre los dos anteriores estriba en que en el primer caso, una clase o una función debe llamarse como **modulo1.nombrefuncion** mientras en el segundo sólo **nombrefuncion** (aunque puede haber conflictos).
- También puede usarse un alias **import utils as utilidades** , **from utils import Shape as Formas**.

25. Módulos y paquetes

- **Hiding elements of a Module:** Si declaro por ejemplo una función en un módulo **def _special_function()** cuyo nombre empieza por barra baja en un módulo por ejemplo llamado **utils**.
 - `import utils` o `from utils import *`, no accederán a ella.
 - Sólo puedo acceder si la declaro específicamente **from utils import _special_function**
 - **Suelen ser funciones que sólo los desarrolladores tienen acceso a ellas.**
- Normalmente **import** se realiza a nivel del programa, al principio. Se puede hacer sólo dentro de una función para limitar el uso de las funciones/clases importadas al ámbito de la función.
- Algunas propiedades de los módulos pueden accederse con `__name__`, `__doc__`, `__file__`, o `dir(módulo)`.
- Algunos módulos son estándar. Por ejemplo, el módulo **sys**. Por ejemplo **sys.path** arroja los directorios del sistema que contienen los módulos de Python (variable PYTHONPATH). Esta variable se puede sobrescribir. Python busca los módulos primero en la carpeta del programa ejecutado, si no lo encuentra, en cada uno de los directorios de PYTHONPATH y por último en el directorio por defecto (en Linux `/usr/local/lib/python/`)

```
import sys

print("sys.version: ", sys.version)
print("sys.maxsize: ", sys.maxsize)
print("sys.platform: ", sys.platform)
print("sys.path: ", sys.path)
```

25. Módulos y paquetes (Python Idiomático)


```
"""This is a test module"""
"""Este módulo tiene un código ejecutable. Pero si lo llamo desde mi programa,
el código no se ejecuta, igual que le módulo1_BIS. Pero además la forma de escribirlo
con una función main() - principal - es la forma de PYTHON IDIOMÁTICO o PYTHONIC"""
print('Hello I am module 1')

def f1():
    print('f1[1]')

def f2():
    print('f2[1]')

def main():
    """Esto permite que se ejecute SOLO cuando es llamado de forma INDEPENDIENTE, STANDALONE"""
    x = 1 + 2
    print('x is', x)
    f1()
    f2()

if __name__ == '__main__':
    main()
```

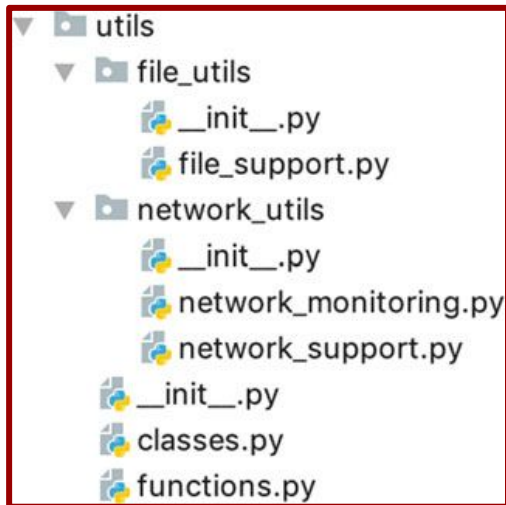


Si el módulo se llama desde otro programa (uso como librería) su variable `__name__` toma el nombre del módulo. Por lo tanto, con esta nomenclatura no ejecutará el código de inicio (que puede ser un testeo). Si se ejecuta de forma independiente (STANDALONE) su variable `name` tomará el valor `"__main__"` con lo que sí ejecutará el código.

25. Módulos y paquetes

Consiste en una estructura jerárquica formada por:

- Un directorio, que contiene uno o más ficheros python
- Un fichero opcional `__init__.py` . Este fichero contiene código que se ejecuta cuando se importa un paquete, la primera vez que un módulo de ese paquete es invocado.
- Puede haber módulos de clases con varias definiciones de clases y módulos de funciones, con varias definiciones de funciones.
- Por ejemplo si en un paquete **utils** tengo un módulo **misfunciones.py** y otro **misclases.py** , las llamadas se hacen
 - `from utils.misfunciones import *` , `from utils.misclases import *` (mismas reglas que antes)
 - Si hay subpaquetes... **`from utils.subpaquete.modulo import *`**



26. Abstract Base Classes (ABC)

Una clase de base abstracta:

- No se puede instanciar directamente
- Se espera que se extienda en una o más subclases
- Proporciona una interfaz común para otras clases derivadas, definiendo métodos que deben ser implementados por las clases hijas.
- Son útiles para crear una jerarquía basada en una clase root ampliamente reutilizable
 - Pueden tener o no métodos o propiedades abstractos
 - Pueden tener o no métodos o propiedades concretos
 - Pueden tener atributos privados y protegidos (una y/o dos barras bajas)
- Se pueden usar para especificar un protocolo o interfaz común. Hay muchas ABCs predefinidas en Python que sirven para estructuras de datos (collection module), módulos de números y flujos (streams: IO module)

```
from collections import MutableSequence
```

```
class Bag(MutableSequence):  
    pass
```

Se suele obtener una ABC de un módulo (por ejemplo la clase `MutableSequence` del módulo `collections`)

Se crea una clase hija basada en la importada

Cualquier instancia de la clase `Bag` no funcionará. Seguirá siendo abstracta. Me dará el error indicando cuántos métodos debo implementar

26. Abstract Base Classes (implementación)

```
# creamos una clase del tipo ABC usando el módulo abc e importando la clase
# ABCMeta. Debo además especificar el atributo metaclass
```

```
from abc import ABCMeta
```

```
class Shape(metaclass=ABCMeta):
```

```
    def __init__(self, id):
        self.id = id
```

Utilizo la clase ABCMeta del módulo abc. Para implementar una clase abstracta debo añadir el atributo **metaclass**. Ahora mismo, así definida, puede instanciarse porque define un método concreto no abstracto.

```
from abc import ABCMeta, abstractmethod
```

```
class Shape(metaclass=ABCMeta):
```

```
    def __init__(self, id):
        self._id = id
```

```
    @abstractmethod
    def display(self): pass
```

```
    @property
    @abstractmethod
    def id(self): pass
```

Para crear una clase abstracta, debo usar el decorador **@abstractmethod** importado también del módulo abc. Un método se define con **@abstractmethod** y una propiedad con **@property** y **@abstractmethod**.

Observo que el método abstracto asociado al id necesita que éste sea un atributo privado.

26. Abstract Base Classes (interface, virtual class)

INTERFACE

- Python no tiene el concepto de interface como otros lenguajes - Java, C++ - (un contrato entre los que implementan la interfaz y el usuario de la implementación, garantizando que se provee cierta infraestructura).
- Python posee clases abstractas básicas (ABCs), con ciertas propiedades y métodos que pueden ser considerados un contrato.

VIRTUAL CLASSES

- Concepto por el cual una subclase, aunque no se extienda directamente de otra, pueda ser tratada como una subclase de la misma. **La clave de poder extender esta relación es que la subclase virtual coincida con la interfaz requerida en la clase virtual padre.**
- Se consigue si:
 - La clase padre virtual es una ABC.
 - La subclase virtual se registra (orden register) como una subclase virtual de esa ABC
 - Una vez registrada, los métodos `issubclass()` e `isinstance()` devuelven `TRUE` aplicados a esa clase

26. Abstract Base Classes (interface, virtual class)

```
# clases virtuales. Tengo dos clases como las siguientes
```

```
from abc import ABCMeta
```

```
class Person(metaclass=ABCMeta):  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def birthday(self):  
        print('Happy Birthday')
```

```
class Employee(object):  
    def __init__(self, name, age, id):  
        self.name = name  
        self.age = age  
        self.id = id  
    def birthday(self):  
        print('Its your birthday')
```

```
"""Programa principal"""
```

```
# Estas órdenes producen FALSE, a menos que se use  
Person.register(Employee) # importante A  
print(issubclass(Employee, Person))  
e1 = Employee("Luis", 56, 3)  
print(isinstance(e1, Person))
```

Class PERSON. Utilizo la clase ABCMeta del módulo abc. Para implementar una clase abstracta debo añadir el atributo **metaclass**.

Class Employee. La defino sin más, pero sus métodos y atributos parecen heredarse de la clase PERSON, aunque no hay una relación directa

El uso de `issubclass()` o `isinstance()` produce FALSE. No hay relación directa entre ellas. Sin embargo pueden ser subclases virtuales. La orden **Person.register(Employee)** registra la subclase Employee como heredada virtualmente de Person y entonces `issubclass()` o `isinstance()` produce TRUE.

26. Abstract Base Classes (mixin)

MIXINS

- Una clase con cierta funcionalidad (normalmente concreta) que potencialmente es útil en múltiples situaciones pero por sí misma no puede ser instanciada.
- Sin embargo, un mixin puede mezclarse con otras clases y extender los datos y comportamiento de este tipo, y acceder a datos y métodos suministrados por esas clases.
- Un mixin es un tipo de ABC común.
- Si un mixin define un atributo (como self.id) la clase con la que se mezcla debe tenerlo para no generar errores.

```
from abc import ABCMeta
```

```
class PrinterMixing(metaclass=ABCMeta):  
    def print_me(self):  
        print(self)
```

Class PrinterMixing. Clase ABC con un método concreto

```
class Person(object):
```

```
    def __init__(self, name):  
        self.name = name
```

Class Employee. Mezclo con PrinterMixing

```
class Employee(Person, PrinterMixing):
```

```
    def __init__(self, nombre, id, edad):  
        super().__init__(nombre)  
        self.id = id  
        self.edad = edad
```

```
    def __str__(self):  
        return 'Employee(' + str(self.id) + ') ' + self.name + ' [' +  
str(self.edad) + ']'
```

```
""" Programa Principal """
```

```
e1 = Employee("Luis", 32, 45)  
e1.print_me()
```

Y puedo usar su método

27. Protocolos, polimorfismo y descriptores

Contrato implícito (Implicit contract)

- En lenguajes como Java o C++ existen contratos explícitos entre una clase y el usuario de esa clase, en el que se saben los tipos de datos que se pasarán como parámetros, los métodos que se usarán con ese tipo de datos y los que se retornarán.
- En Python no existen esos contratos (llamados interfaces) y eso a veces hace las cosas más complejas. Por ejemplo en una clase **calculator**:

```
class Calculator:
    def add(self, x, y):
        return x + y
```

Esta clase tiene un **contrato implícito**. El método `add` funcionará con cualquier objeto que soporte el operador numérico `add` (suma). Dicho de otra forma, con cualquier cosa que sea numérica.

```
# Contrato implícito. si en una clase hay una operación numérica
# La suma, por ejemplo, cualquier objeto que soporte dicha operación
podrá usar esa clase
class Quantity:
    def __init__(self, numero):
        self.numero = numero
    def __add__(self, other):
        return Quantity(self.numero + other.numero)
    def __str__(self):
        return "Cantidad: [" + str(self.numero) + "]"

class Calculator:
    def add(self, x, y):
        return x+y

q1 = Quantity(3)
q2 = Quantity(7)
# print(q1+q2)
calc = Calculator()
print(calc.add(q1, q2))
```

27. Protocolos, polimorfismo y descriptores

Duck Typing

- “Si andas como un pato, nadas como un pato y vuelas como un pato.... ¡¡¡Eres un pato!!!”
- **Si un objeto de una clase cumple los requerimientos para aplicarle un conjunto de operaciones , entonces puedes usar esas operaciones.** Por ejemplo, si tengo un tipo de objeto en principio no numérico, pero al que puedo aplicar la suma, la resta, la multiplicación y la división, entonces puedo tratarlo como numérico.
- ¿Que quiero decir? Que si tengo código que cuando se escribió se pensó para cierto tipo de objeto, pero que **si existen tipos nuevos de objeto que cumplen ese contrato implícito puedo usar ese código con ellos.**

Protocolo

- En Python no existe nada que obligue a usar un tipo de objeto concreto con un conjunto de operaciones. Simplemente se sigue la regla del Duck Typing. **Pero tenemos protocolos, es decir descripciones informales de la interfaz de programación suministrada en Python** (una clase, un módulo, funciones independientes...). Se definen vía documentación.
- Si suministro el tipo correcto para una operación de una función o un método, todo irá bien. Si no, lanzará un error. Esto es básico para el concepto de **polimorfismo**.
- Ejemplo, el protocolo definido para **Sequences**, como un contenedor que puede ser accesible en un elemento cada vez. Cualquier tipo/clase que se adapte a ese contenedor debe soportar los métodos `__len()` y `__getitem()`. Aunque el protocolo es informal; podría haber una clase iterativa que implemente `__getitem()` sólo .

27. Protocolos / polimorfismo

```
with ContextManagedClass() as cmc:  
    print('In with block', cmc)  
    print('Existing')
```

Un ejemplo: el protocolo Context Manager

- Se usa cuando se necesita una conexión a un archivo o base de datos (resource). Lo asociamos a la sentencia “**with as**”.
- Asegura que se ejecutan los pasos necesarios para abrirlo y que al terminar con él, se cierra para evitar problemas ulteriores en el programa.
- El objeto **cmc** (ver ejemplo) sólo tiene un alcance dentro de la sentencia “with as”.
- Al llamar a “with as” llamamos a un método denominado `__enter__()` en el que se espera se lleven a cabo los pasos necesarios para abrir el recurso externo. Al acabar la última sentencia de “with as” se ejecuta el método `__exit__()` , y puede dar información sobre excepciones.

Polimorfismo

- La capacidad de múltiples objetos de diferentes clases de realizar la misma operación, aunque ésta se lleve a cabo de distinta forma según el tipo de objeto concreto.
- El objeto `p` puede ser de diferentes clases, siempre que tenga definidos los métodos `eat`, `drink` y `sleep`. Estas clases pueden ser independientes (se sigue el principio del Duck Typing) o pueden ser extendidas unas de otras, incluso con métodos sobrescritos.

```
def night_out(p):  
    p.eat()  
    p.drink()  
    p.sleep()
```

27. Polimorfismo y descriptor protocol

Polimorfismo

- La capacidad de múltiples objetos de diferentes clases de realizar la misma operación, aunque ésta se lleve a cabo de distinta forma según el tipo de objeto concreto.
- El objeto p puede ser de diferentes clases, siempre que tenga definidos los métodos eat, drink y sleep. Estas clases pueden ser independientes (se sigue el principio del Duck Typing) o pueden ser extendidas unas de otras, incluso con métodos sobrescritos.

Protocolo Descriptor (ver siguiente diapositiva)

- Es una forma de definir los atributos de una clase (cursor) a través de un atributo genérico de otra clase que sigue el protocolo descriptor (logger). Estos atributos se llamarían *managed attributes*.
- El protocolo descriptor define unos métodos como `__get__`, `__set__`, `__delete__`, y `__set_name__`.
- La clase ejemplo cursor, a veces utiliza `self.__dict__[x]` para evitar llamar al descriptor. Utilizar `self.x` puede ser problemático en `__init__` y `__str__`

27.Descriptor protocol

```
class Logger(object):
    """ Logger class implementing the descriptor protocol """
    def __init__(self, name):
        self.name = name
    def __get__(self, inst, owner):
        print(' get :', inst, 'owner', owner, ', value', self.name, '=', str(inst.__dict__[self.name]))
        return inst.__dict__[self.name]
    def __set__(self, inst, value):
        print('__set__:', inst, '-', self.name, '=', value)
        inst.__dict__[self.name] = value
    def __delete__(self, instance):
        print('__delete__', instance)
    def __set_name__(self, owner, name):
        print('__set_name__', 'owner', owner, 'setting', name)
```

```
class Cursor(object):
    # Set up the descriptors at the class level
    x = Logger('x')
    y = Logger('y')
    def __init__(self, x0, y0):
        # Initialise the attributes
        # Note use of __dict__ to avoid using self.x notation
        # which would invoke the descriptor behaviour
        self.__dict__['x'] = x0
        self.__dict__['y'] = y0
    def move_by(self, dx, dy):
        print('move_by', dx, ', ', dy)
        self.x = self.x + dx
        self.y = self.y + dy
    def __str__(self):
        return 'Point[' + str(self.__dict__['x']) + ', ' + str(self.__dict__['y']) + '']'
```

```
""" Programa Principal """
cursor = Cursor(15, 15)
print('-' * 25)
print('p1:', cursor)
cursor.x = 20
cursor.y = 35
print('p1 updated:', cursor)
print('p1.x:', cursor.x)
print('-' * 25)
cursor.move_by(1, 1)
print('-' * 25)
```

28. Monkey patching

Class Bag. Sin un método len definido

Definir una función para obtener la longitud. Se asume que se pasa un objeto que tiene el atributo **data**

Monkey Patching

- La idea de añadir comportamiento a un objeto, en tiempo de ejecución, que un programador no tuvo en su momento. Si esta característica nueva se convierte en algo común, puede añadirse a la clase. Pero si no, se añade en tiempo de ejecución para ese programa en concreto.
- La referencia self implica que se va a usar un objeto, y que esa función se convertirá en un método de esa clase.

```
# Añadir funcionalidad en tiempo de ejecución
```

```
class Bag:
    def __init__(self):
        self.data = ['a', 'b', 'c']
    def __getitem__(self, pos):
        return self.data[pos]
    def __str__(self):
        return 'Bag(' + str(self.data) + ')'
```

```
def getlength(self):
    return len(self.data)
```

```
# Monkey patching
```

```
Bag.__len__ = getlength
b = Bag()
print(b)
```

```
# pero no podemos hacer
print(len(b))
```

Se asocia la longitud de la clase a esa función. Y ya se puede usar.

28. Monkey patching (añadiendo atributos)

```
# Añadir funcionalidad en tiempo de ejecución

class Bag:
    def __init__(self):
        self.data = ['a', 'b', 'c']
    def __getitem__(self, pos):
        return self.data[pos]
    def __str__(self):
        return 'Bag(' + str(self.data) + ')'

def getlength(self):
    return len(self.data)

# Monkey patching
Bag.__len__ = getlength
Bag.name = "mi bolsa"
b = Bag()
print(b)
print(b.name)

b.name = "mi nueva bolsa"
print(b.name)
```

También puedo añadir un atributo. En este caso con un valor por defecto (Bag.name = "mi bolsa")

Que después puedo usar en el objeto, y modificarlo.

28. Búsqueda de atributos

Búsqueda de atributos

- Tenemos dos tipos: atributos de la clase y atributos de las instancias u objetos. Un atributo de clase puede ser, por ejemplo, el conteo de número de instancias. Un atributo de un objeto son sus atributos definidos. Podemos acceder a ellos por el método `__dict__`
- Por ejemplo, la clase **Estudiante** y el objeto **alumno**.
- Python busca atributos de la clase **Estudiante** en su diccionario, y si no, en los diccionarios de las clases padre.
- Python busca atributos de la instancia **alumno** primero en el diccionario de la instancia, después en el diccionario de la clase y si no, en los diccionarios de las clases padre.

```
# Tenemos una clase estudiante, que define instancias
alumnos

class Estudiante:
    count = 0 # atributo de clase

    def __init__(self, nombre, curso):
        self.nombre = nombre
        self.curso = curso
        Estudiante.count += 1

    def __str__(self):
        return self.nombre + " pertenece al curso " +
self.curso

""" Programa principal """

a1 = Estudiante("Luis", "1ESOC")
print("Atributos de clase: ", Estudiante.__dict__)
print("Atributos de instancia u objeto: ", a1.__dict__)
```


28. Búsqueda de atributos

Manejando atributos desconocidos

- Si intento acceder a un atributo que no existe, me da un error del tipo **AttributeError**. Puedo o bien intentar capturarlo con **try** o bien usar el método **__getattr__(self,item)** que se dispara cuando no encuentra un atributo en **dict**.

```
student = Student('John')  
  
res1 = student.dummy_attribute  
print('p.dummy_attribute:', res1)
```

```
def __getattr__(self, item):  
    print("No he encontrado el atributo: ",item)  
    return "default"
```

Manejando métodos desconocidos

- También sirve el mismo método para interceptar un método desconocido

```
def my_method(self):  
    return 'default'  
  
def __getattr__(self, item):  
    print("No he encontrado el atributo:  
",item)  
    return self.my_method()
```

Interceptando búsqueda de atributos

- Siempre puede usarse el método **__getattribute__**. Se diferencia del anterior en que se ejecuta siempre, sea que el atributo exista o no. hay que tener cuidado de no entrar en llamadas recursivas (usando dentro de él, por ejemplo, la llamada a un atributo).

```
def __getattribute__(self, item):  
    print ("Has llamado a un atributo ", item)  
    return object.__getattribute__(self,item)
```

28. Búsqueda de atributos

Interceptando la asignación de atributos

- También es posible usar el método `__setattr__()` para interceptar cuando se asigne un atributo o se actualice. Este método sustituye a la asignación cuando se invoca, así que para guardar el dato **debe introducirse directamente en el diccionario** o llamar a `__setattr__()` de la clase base `object` directamente

```
class Estudiante:
    count = 0 # atributo de clase
    def __init__(self, nombre, curso):
        self.nombre = nombre
        self.curso = curso
        Estudiante.count += 1
    def mymethod(self):
        return 'default'
    def __getattr__(self, item):
        print("No he encontrado el atributo: ",item)
        return self.mymethod()
    def __getattribute__(self, item):
        print ("Has llamado a un atributo ", item)
        return object.__getattribute__(self,item)
    def __setattr__(self, key, value):
        print("He cambiado el atributo "+key+" con el valor "+value)
        # self.__dict__[key]=value
        object.__setattr__(self,key,value)
    def __str__(self):
        return self.nombre + " pertenece al curso " + self.curso

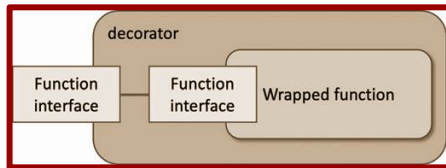
""" Programa principal """

a1 = Estudiante("Luis", "1ESOC")
a1.nombre = "Pepe"
print(a1)
```

29. Decoradores

¿Qué es un decorador?

- Es un trozo de código que se aplica a un objeto, una clase o una función y que modifica o mejora su comportamiento, transformándolo en un objeto “nuevo”. Suele aplicarse a funciones.
- La interfaz de una función no cambia: es decir, sus parámetros y que retorne algo o nada sigue igual. Pero su comportamiento mejora. De alguna manera el decorador envuelve a la función original y la modifica.
- Un decorador puede ser una función que toma una función como parámetro y retorna otra función, mejorada.
- La función **logger** mejora la función **func**.



¿Cómo decorar una función?

```
def recuadrar(func):  
    def funcion_recuadrar():  
        print("=" * 25)  
        func()  
        print("=" * 25)  
  
    return funcion_recuadrar  
  
def target():  
    print("Este es mi objetivo")  
  
t1 = target  
t1()  
t2 = recuadrar(target)  
t2()
```

```
def recuadrar(func):  
    def funcion_recuadrar():  
        print("=" * 25)  
        func()  
        print("=" * 25)  
  
    return funcion_recuadrar  
  
@recuadrar  
def target():  
    print("Este es mi objetivo")  
  
t1 = target  
t1()
```

```
def logger(func):  
    def inner():  
        print('calling ', func.__name__)  
        func()  
        print('called ', func.__name__)  
  
    return inner
```

Decorando la función con @. Un forma más “Pythonic”

29. Decoradores

```
def recuadrar(func):  
    def funcion_recuadrar(valor):  
        print("=" * (len(valor)+15))  
        func(valor)  
        print("=" * (len(valor)+15))  
  
    return funcion_recuadrar  
  
@recuadrar  
def target(valor):  
    print("Este es mi " + valor)  
  
t1 = target  
t1("objetivo")
```

Si la función toma parámetros, el decorador debe tomar los mismos parámetros

Decoradores apilados

- Los decoradores pueden apilarse, unos sobre otros.

```
def negrita(func):  
  
    def poner_en_negrita(valor):  
        print("<b>")  
        func(valor)  
        print("</b>")  
  
    return poner_en_negrita  
  
def recuadrar(func):  
    ... IGUAL QUE ANTES  
  
@negrita  
@recuadrar  
def target(valor):  
    print("Este es mi " + valor)  
  
t1 = target  
t1("objetivo")
```

29. Decoradores parametrizados

```
def rec(active=True):  
    def recuadrar(func):  
        def funcion_recuadrar():  
            print("=" * 25)  
            func()  
            print("=" * 25)  
  
        if active:  
            return funcion_recuadrar  
        else:  
            return func  
  
    return recuadrar  
  
@rec()  
def target1():  
    print("Este es mi objetivo")  
  
@rec(active=False)  
def target2():  
    print("Este es mi objetivo, pero no lo he conseguido")  
  
target1()  
target2()
```

Decorador con el parámetro **active** por defecto a True

Función que devuelve según sea el parámetro **active**

Funciones decoradas según el parámetro.

29. Decoradores. Métodos

Decorando métodos

- Se pueden decorar métodos, ya que son también funciones, pero hay que tener en cuenta que cogen el parámetro **self**.

```
def pretty_print(method):  
    def method_wrapper(self):  
        return "<p>{0}</p>".format(method(self))  
    return method_wrapper  
  
class Person:  
    def __init__(self, name, surname, age):  
        self.name = name  
        self.surname = surname  
        self.age = age  
  
    def print_self(self):  
        print('Person - ', self.name, ', ', self.age)  
  
    @pretty_print  
    def get_fullname(self):  
        return self.name + " " + self.surname  
  
p1 = Person("Aurelio", "Gallardo", 53)  
print(p1.get_fullname())
```

Decorador. Tener en cuenta que le debemos pasar el parámetro self.

Si tiene parámetros, además el decorador debe tenerlos en cuenta.

```
def trace(method):  
    def method_wrapper(self, x, y):  
        print('Calling', method, 'with', x, y)  
        method(self, x, y)  
        print('Called', method, 'with', x, y)  
    return method_wrapper
```

29. Decoradores. Clases.

Decorando clases

- Usamos una trama de tipo singleton (Singleton Design Pattern). El decorador comprueba si existe o no una instancia. Si no existe, la crea. Si existe la devuelve. Así, no puede crearse más de una instancia de cada clase.

```
def singleton(cls):  
    print('In singleton for: ', cls)  
    instance = None  
  
    def get_instance():  
        nonlocal instance  
        if instance is None:  
            instance = cls()  
        return instance  
  
    return get_instance  
  
@singleton  
class Service(object):  
    def print_it(self):  
        print(self)  
  
@singleton  
class Foo:  
    pass
```

```
print('Starting')  
s1 = Service()  
print(s1)  
s2 = Service()  
print(s2)  
f1 = Foo()  
print(f1)  
f2 = Foo()  
print(f2)  
print('Done')
```

Decorador. Tener en cuenta que le debemos pasar el parámetro cls. En este caso, comprueba si existe o no una instancia. Si no existe, la crea. Si existe, simplemente la pasa. El resultado es que en este programa, nunca puede haber más de una instancia de esa clase cuando se decora.

Se aplica el decorador a varias clases.

No se pueden crear más de dos instancias. Si se intenta siempre retorna la misma.

29. Decoradores.

¿Cuándo se ejecuta un decorador?

- El decorador en sí se ejecuta en tiempo de importación, al principio. Pero la función decorada y la función que envuelve se ejecutan cuando son invocadas, en tiempo de ejecución.

Decoradores pre-establecidos

- Hay varios decoradores pre-establecidos, como @classmethod, @staticmethod and @property. También los hay asociados a métodos y propiedades abstractas, de testeo y operaciones asíncronas.

Functools wraps

- Cuando uso un decorador, el nombre de la función, su docstring y la propiedad module quedan enmascarada por la función decoradora. Para evitar ésto se usa el módulo **functools** con el decorador **wraps**. Este decorador hace que la función decorada tenga los mismos atributos que la original.

```
from functools import wraps

def logger(func):
    @wraps(func)
    def inner():
        print('calling ', func.__name__)
        func()
        print('called ', func.__name__)
    return inner
```


30. Iterables

¿Qué es el protocolo iterable?

- Es un protocolo que se usa con tipos donde se puede procesar su contenido uno cada vez por turnos.
- Listas, sets, diccionarios, tuplas.... Todos son tipos iterables y deben suministrar un iterador (**iterador**).
- Para ser del tipo iterable debe implementarse el método `__iter__()`. En este método debe suministrarse una referencia al objeto iterador.

¿Qué es un iterator (iterador)?

- Es un objeto que retorna una secuencia de valores. Puede ser una serie de valores finitos o "infinitos".
- En él se especifica el método `__next__()`. El método `next` retorna el siguiente valor en la secuencia o la excepción **StopIteration**, indicando que el iterador deja de suministrar valores.

Un tipo de datos puede cumplir los protocolos iterable e iterator a la vez. El programador puede hacerlos así o no. Ver ejemplo.

También es útil el módulo **itertools** que incluye funciones para usar varios iteradores, combinarlos, etc.

```
# clase que define un método iter
(iterable)
# y un método next() iterador

class Evens:
    """clase que genera números pares"""

    def __init__(self, limit):
        self.limit = limit
        self.val = 0

    def __iter__(self):
        """ Hago esta clase iterable"""
        return self

    def __next__(self):
        """ Iterador. Si el valor es
        mayor que el límite, que genere una
        excepción"""
        if self.val > self.limit:
            raise StopIteration
        else:
            valor = self.val
            self.val += 2
            return valor

print('Start')
for i in Evens(8):
    print(i, end=' ', )
print("Hecho")
```

30. Generadores

Un generador sólo funciona dentro de una función o módulo. Son funciones especiales que van generando valores en demanda. Se usa en combinación con la palabra especial **yield**.

Yield no es como return. Cuando ejecutamos return, la función acaba. Cuando ejecutamos yield, la función pasa el valor pero volverá al siguiente ciclo. La función no acaba hasta llegar al final o provocar un return

Se usa en situaciones en las que puede haber problemas de memoria u otros. Se obtiene los valores uno cada vez, no todos a la vez.

```
def generador(n):  
    for i in range(n):  
        if i%2 == 0:  
            yield i  
  
for k in generador(20):  
    print(k)
```



```
def gen_numbers():  
    yield 1  
    yield 2  
    yield 3
```

forma simple de
generador con
varios yields

```
for i in pares(8):  
    for j in pares(8):  
        print ("[{},{}]".format(i,j),  
end=" ",  
print()
```

Se pueden anidar los generadores

```
# defino un generador  
para números pares  
  
def pares(limite):  
    valor = 0  
    while valor < limite:  
        yield valor  
        valor += 2  
  
print('Start')  
for i in pares(200):  
    print(i, end=', '  
print("Acabo")
```

```
def pares(limite):  
    valor = 0  
    while valor < limite:  
        yield valor  
        valor += 2  
mis pares = pares(80)  
print(next(mis pares))  
print(next(mis pares))  
print(next(mis pares))
```

Usando next se pueden ir accediendo a los distintos valores del generador
Sin usar un loop

30. Coroutines (corutinas)

Una **corutina** es una función *que espera un dato*. Al contrario que el generador que genera un dato y se espera a generar otro, la corutina espera a seguir cuando recibe un dato. La confusión entre ambas es que se usa la palabra reservada **yield** de nuevo para esperar ese dato. También `next()` para inicializar (prime the coroutine) o `send()` para enviar el dato, y `close()` para terminar de que espere datos.

En el ejemplo, la corutina **grep** se inicializa con un patrón (la palabra Python). Se “ceba” con `next` y se le manda texto con `send`.

Si nos damos cuenta recibe los datos (a través de `yield`) y se los asigna a la variable `line`.

Si la variable `line` contiene la palabra Python (pattern) , imprimirá esa línea. Si no, no.

Tengo que cerrar la corutina con `close`.

```
def grep(pattern):
    print('Looking for', pattern)
    try:
        while True:
            line = (yield)
            if pattern in line:
                print(line)
    except GeneratorExit:
        print('Exiting the Co-routine')

print('Starting')
# Initialise the coroutine
g = grep('Python')
# prime the coroutine
next(g)
# Send data to the coroutine
g.send('Java is cool')
g.send('C++ is cool')
g.send('Python is cool')
# now close the coroutine
g.close()
print('Done')
```

31. Collection, Tuples and Lists

Una colección (collection) es un tipo de objeto que representa un grupo de objetos. Podemos referirnos a ellas como contenedores.

Las colecciones son usadas como base para estructuras de datos más complejas. Hay cuatro tipos:

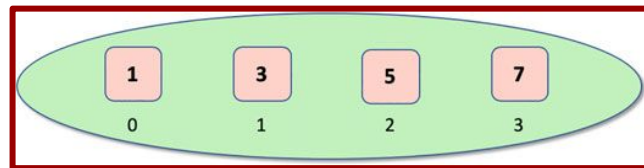
1. **Tuplas:** colección de objetos ordenados y son **inmutables** (no se pueden modificar). Permiten miembros duplicados y están indexados.
2. **Listas:** colección de objetos ordenados, indexados, permiten miembros duplicados y son **mutables** (pueden modificarse).
3. **Sets:** colección de objetos desordenados y no indexados. Son **mutables**, pero no permite miembros duplicados.
4. **Diccionarios:** colección desordenada que funciona con parejas clave (key) - valor. Son mutables, se permite valores duplicados pero no claves duplicadas.

Tuplas

`t1 = (3, 5, 6)` . Es un **iterable**.

O bien a partir de una lista `l1 = [3, 5, 6]` con la orden `tuple`. `t1 = tuple(l1)`

Se accede a un elemento con su índice en `[]` . Por ejemplo, `t1[2]` daría 6.



31. Collection, Tuples and Lists

Slices

- Por ejemplo, si tengo una tupla `t1=(1,3,5,7,9,11,13,15)`, `t1[1:3]` incluye del 1 al 3 pero el 3 **excluyéndolo**. `t1[1:3]=(3,5)`
- Del principio hasta el 2: `t1[:3] → (1, 3, 5)`
- Del 2 hasta el final: `t1[2:] → (5, 7, 9, 11, 13, 15)`
- Al revés `t1[::-1] → (15, 13, 11, 9, 7, 5, 3, 1)`
- `tup2 = (1, 'John', Person('Phoebe', 21), True, -23.45)` → pueden contener diferentes tipos
- Se puede iterar con **for element in t1:**
- Se puede obtener el len con **len(t1)**
- Se puede contar cuantas veces hay un elemento con **t1.count(1)**
- Y saber el índice del elemento con **t1.index(1) (Primera aparición)**
- Y por fin, una tupla puede contener otras tuplas. En general otros contenedores, como listas, sets o diccionarios. Se pueden anidar tuplas.
- Lo que no se puede hacer, porque es **immutable**, es una nueva asignación `t1[0]=6` **provocaría un error**. De hecho, cualquier slice de una tupla lo que genera es una nueva tupla. Tampoco se puede borrar un elemento.

31. Lists

Listas

- Se aplican mucho de los conceptos de la tupla, pero son elementos **mutables**. Lo que significa que se pueden añadir, modificar y eliminar elementos de las listas.
- Se pueden anidar listas, y listas con tuplas y viceversa.
- La orden list() convierte cualquier iterable en una lista. Se accede a sus elementos con un índice.
- Se pueden usar slices en una lista. Los índices negativos indican que se recorren al revés.
 - append() → añade un elemento a la lista
 - extend() → añade una lista al final de otra lista. Ejemplo: list1.extend([3,4,5])
 - insert(index, elemento) → añade un elemento en la posición **index**. Los siguientes los desplaza.
 - Se pueden concatenar lista con el signo +
 - remove() → elimina un elemento de la lista. Si no existe, da un error.
 - pop(index) → elimina el elemento de la lista con índice index. Tb devuelve el elemento eliminado.
 - También se puede eliminar un elemento de la lista con del → del list1[3]

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

32. Sets

Sets

- Son objetos **inmutables** y **desordenados**. Van entre corchetes {} . Por ejemplo, **frutas = {'pera','fresa','manzana'}**
- No permite duplicados. Si escribo **frutas = {'pera','fresa','manzana','pera'}** , se eliminará una "pera".
- Son desordenados: no se admite un índice.
- Un iterable se puede convertir a set con `→ set()`
- No se puede acceder a sus elementos con un index. Hay que hacerlo iterando con, por ejemplo, un for.
- Se comprueba si un elemento está en el set con **in** `→ 'pera' in fruta`
- Se puede usar el método **add()** para añadir un dato al set. Por ejemplo, `frutas.add('naranja')`
- Y el método **update()** para añadir varios al set. Por ejemplo, `frutas.update(["ciruelas","mandarinas"])`
- **len()** `→` longitud, **max()** y **min()** `→` máximos y mínimos.
- **remove()** y **discard()** para eliminar elementos. **pop()** elimina el último, y **clear()** todo el set.
- Cuando el set se vacía , se imprime `set()`.
- En un set podemos incluir **inmutables**, por eso se puede incluir una tupla, **pero no se puede incluir una lista ni otro set. A menos que se use frozenset()**.

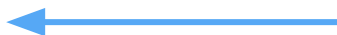
```
# solo se pueden incluir en set
inmutables

set1 = {
    "árbol", "abedul", "roble"
}
set2 = {"conífera", "pino", "haya"}
tup1 = (3,4,5)

set1.add(tup1)
print(set1)

# set1.add(set2) --> no se puede
hacer. provoca un error

# Pero sí
set1.add(frozenset(set2))
print(set1)
```



32. Sets

Sets (operaciones)

- Unión (`|`), Intersección (`&`), Diferencia (`-`) y Diferencia simétrica (`^`)
 - `s1.union(s2)` es el equivalente de `s1 | s2`
 - `s1.intersection(s2)` es el equivalente de `s1 & s2`
 - `s1.difference(s2)` es el equivalente de `s1 - s2`
 - `s1.symmetric_difference(s2)` es el equivalente de `s1 ^ s2`

```
s1 = {'apple', 'orange', 'banana'}
s2 = {'grapefruit', 'lime', 'banana'}

print("Unión: ", s1 | s2) # elementos de los dos conjuntos
no repetidos
print("Intersección: ", s1 & s2) # elementos comunes

print("Diferencia: ", s1 - s2) # diferencia: quita de s1 los
elementos de él que estén en s2

print("Diferencia simétrica: ", s1 ^ s2) # diferencia
simétrica: la unión de los dos menos la intersección
print("Diferencia simétrica: ", (s1 | s2) - (s1 & s2))
```

Sets (métodos)

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all the elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	Removes the items in this set that are also included in another, specified set
<code>discard()</code>	Remove the specified item
<code>intersection()</code>	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>	Returns whether two sets have a intersection or not
<code>issubset()</code>	Returns whether another set contains this set or not
<code>issuperset()</code>	Returns whether this set contains another set or not
<code>pop()</code>	Removes an element from the set
<code>remove()</code>	Removes the specified element
<code>symmetric_difference()</code>	Returns a set with the symmetric differences of two sets
<code>symmetric_difference_update()</code>	inserts the symmetric differences from this set and another
<code>union()</code>	Return a set containing the union of sets
<code>update()</code>	Update the set with the union of this set and others

33. Diccionarios

Diccionarios

- Se definen con { } llaves. Son pares de valores **clave:valor** → ciudades = {"España":"Madrid", "Portugal":"Lisboa"}
- La función dict() crea un diccionario. Tres formas:
 - dict (**kwargs) , dict(mapping, **kwargs) , dict(iterable, **kwargs)
- Se puede obtener un valor a través de corchetes y su clave, o con la función get
 - print(**ciudades['España']**) o bien print (**ciudades.get('España')**)
 - Para modificar o añadir uno: ciudades['España']='Barcelona' o bien ciudades['Francia']='París'
- pop(clave) → borra el elemento con esa clave y devuelve el valor.
- popitem() → borra el último elemento
- del('clave') → borra el elemento con esa clave
- clear() → borrar el diccionario.
- Para iterar: **for clave in diccionario:** o bien **for valores in dict.values():**

```
# note keys are not strings
dict1 = dict(uk='London', ireland='Dublin', france='Paris')
print('dict1:', dict1)
# key value pairs are tuples
dict2 = dict([('uk', 'London'), ('ireland', 'Dublin'),
('france', 'Paris')])
print('dict2:', dict2)
# key value pairs are lists
dict3 = dict([('uk', 'London'], ['ireland', 'Dublin'],
['france', 'Paris']])
print('dict3:', dict3)

# añadiendo (mismo método de asignación para modificar)
dict1['españa'] = 'Madrid'
dict1['portugal'] = 'Lisboa'

# borrando
dict1.pop('uk') # o bien, del pero sin devolver valor.
clear borra todo el diccionario.

print(dict1)
# iterando
for clave in dict1:
    print("Mi clave es: ",clave," y mi valor: ",dict1[clave])

for valor in dict1.values():
    print("Directamente el valor: ",valor)
```

33. Diccionarios

Diccionario (métodos)



Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and values
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns a list containing the tuple for each key value pair
<code>keys()</code>	Returns a list containing the dictionary's keys
<code>pop()</code>	Removes the element with the specified key
<code>popitem()</code>	Removes the last inserted key-value pair
<code>setdefault()</code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<code>update()</code>	Updates the dictionary with the specified key-value pairs
<code>values()</code>	Returns a list of all the values in the dictionary

Diccionarios

- El método **values()** devuelve una lista con los valores; el método **keys()** una lista de las claves y el método, **items()** las parejas clave - valor. Con los tipos respectivos `dict_values`, `dict_keys`, `dict_items`
- Se comprueba **una clave** con `in` → "españa" in `dict1`, devolviendo `True` o `False`.
- Longitud con `len` → `len(dict1)`
- Se pueden anidar en diccionarios **tuplas, listas, sets e incluso otros diccionarios** como **valores**.
- Si una clase se usa como objetos susceptibles de ser claves en un diccionario puede tener dos métodos:
 - `hash` → un número único por clave. Implica que sean inmutables. **`key.__hash__(): 8507681174485233653`**
 - `eq` → para testear si dos objetos son iguales **`key.__eq__('England'): True`**
 - Si dos objetos son iguales, tienen el mismo hash.
 - Si tengo dos hashes iguales, entonces probablemente sea el mismo objeto. Los hashes deben ser fáciles de obtener.
 - Normalmente no hay que preocuparse por esto. Sólo si una clase que yo defina se usará como claves en un diccionario. Entonces debo implementar los métodos `hash` y `eq`.

34. Módulos relacionados con colecciones

List comprehension

- Sintaxis: [<expression> for item in iterable <if optional_condition>]
- El resultado es una lista, que proviene de iterar otra según la expresión resultado.
- Se puede iterar otro iterable, como sets o tuplas.

```
# list comprehension
l1 = [1, 2, 3, 4, 5]
print(l1)
l2 = [item*item for item in l1]
print(l2)
l3 = [item*item for item in l1 if
item%2 != 0] # filtra y solo lo
aplica con los impares
print(l3)
```

Módulo colecciones

- Amplía las características básicas de los tipos de datos orientados a colecciones con características de más nivel. Suministra más contenedores útiles como... (Ver ejemplo del módulo Counter)

Name	Purpose
namedtuple()	Factory function for creating tuple subclasses with named fields
deque	List-like container with fast appends and pops on either end
ChainMap	Dict-like class for creating a single view of multiple mappings
Counter	Dict subclass for counting hashable objects
OrderedDict	Dict subclass that remembers the order entries were added
Defaultdict	Dict subclass that calls a factory function to supply missing values
UserDict	Wrapper around dictionary objects for easier dict subclassing
UserList	Wrapper around list objects for easier list subclassing
UserString	Wrapper around string objects for easier string subclassing

```
import collections
frutas =
collections.Counter(['manzana', 'plátano', 'fresa', 'manzana', 'fresa', 'fresa', 'pera'])
print(frutas)
print(frutas['fresa'])

print(frutas.most_common(1)) # obtiene el
elemento más común

frutas['plátano'] += 5 # añade 5 plátanos más
print(frutas)

print(frutas.most_common(1)) # obtiene el
elemento más común
```

34. Módulos relacionados con colecciones

Itertools

- Provee funciones relacionadas con iterables.

```
import itertools
# Connect two iterators together
r1 = list(itertools.chain([1, 2, 3], [2, 3, 4]))
print(r1)
# Create iterator with element repeated specified number of
# times (possibly infinite)
r2 = list(itertools.repeat('hello', 5))
print(r2)
# Create iterator with elements from first iterator starting
# where predicate function fails
values = [1, 3, 5, 7, 9, 3, 1]
r3 = list(itertools.dropwhile(lambda x: x < 5, values))
print(r3)
# Create iterator with elements from supplied iterator between
# the two indexes (use 'None' for second index to go to end)
r4 = list(itertools.islice(values, 3, 6))
print(r4)
```

35. Abstract Data Types

Abstract Data Types (ADT)

- Modelo de tipo de datos, donde ese tipo de datos es definido por su comportamiento (o semántica) desde el punto de vista del usuario. Dos tipos: queues (colas) y stacks (pilas).

Queues o colas

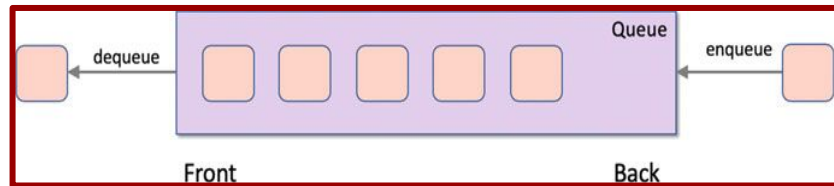
- ADT que sigue la normas First-in-first-out (o FIFO). La primera entidad añadida a la cola, es la primera en salir. Se mantiene el orden en una cola.

Pilas o Stacks

- Sigue la norma Last-in-first-out (LIFO). El último en entrar, es el primero en salir. Se mantiene el orden de las entidades en una pila.

35. Queues (colas)

- Si se añade un elemento, se coloca detrás (enqueueing)
- Si se remueve un elemento, se quita por delante (dequeueing)
- Se puede saber la longitud de una cola, si está vacía, etc.
- Las colas pueden ser de longitud fija o creciente (variable)



Algunas colas tienen características añadidas:

- Orden peek, para comprobar cuál es el primer elemento sin retirarlo.
- Añadir con prioridad: no colocan elementos al final, sino en la zona intermedia.

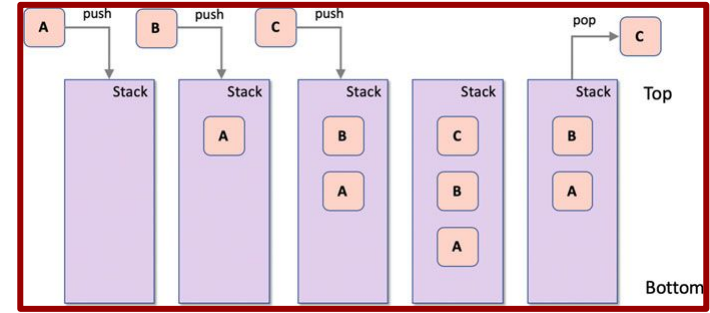
En Python se puede usar una lista como una cola con los métodos `append()` y `pop()`

```
queue = [] # Create an empty queue
queue.append('task1')
print('initial queue:', queue)
queue.append('task2')
queue.append('task3')
print('queue after additions:', queue)
element1 = queue.pop(0)
print('element retrieved from queue:', element1)
print('queue after removal', queue)
```

35. Stacks (pilas)

- Si se añade un elemento, se coloca arriba de la pila (pushing on the stack)
- Si se quita un elemento, se quita de arriba de la pila (popping from the stack)
- Con una cola se puede añadir elementos, quitarlos, determinar su longitud, y pueden ser de tamaño fijo o creciente.
- Algunas permiten el método top, que determina el valor de la pila superior, pero sin removerlo.
- Como en el caso de las colas, se puede implementar una pila con listas.

Comportamiento básico



```
stack = [] # create an empty stack
stack.append('task1')
stack.append('task2')
stack.append('task3')
print('stack:', stack)
top_element = stack.pop()
print('top_element:', top_element)
print('stack:', stack)
```

36. Funciones de alto nivel: filter

- Python contiene tres funciones de alto nivel: filter(), map() y reduce(). Esas funciones toman como argumentos una colección y una función f1, que aplicarán a esa colección.

Filter

- Es una función que compara todos los elementos de la colección (iterable) con la función f1 de testeo. Si cumplen con esa condición, devuelve un iterable con los valores que retornan True.
- Su sintaxis es filter(f1, iterable). Iterable puede ser cualquier objeto iterable, incluidas listas, sets, diccionarios, etc. Puede combinarse con clases.
- La función f1 puede ser una lambda (Usualmente) o una función ya implementada en el programa.

```
# ejemplo de la función filter

def pares(x):
    return x % 2 == 0

datos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(datos)

d2 = list(filter(lambda i: i % 2 == 0,
datos)) # Solo encuentra los pares.
Hay que convertirlo en lista
print(d2)

d3 = list(filter(pares, datos)) # Solo
encuentra los pares, con una función.
Hay que convertirlo en lista
print(d3)
```

```
class Person():

    def __init__(self, nombre, edad):
        self.name = nombre
        self.age = edad

    def __str__(self):
        return "(" + self.name + ", " + str(self.age) + ")"

personas = [Person("Luis", 23), Person("Alberto", 19), Person("Ana", 57)]
for p in personas:
    print(p, end=" // ")
print()
print("="*25)
jovenes = filter(lambda p: p.age <= 30, personas)
for p in jovenes:
    print(p, end=" // ")
```


36. Funciones de alto nivel: map

Map

- Es una función que devuelve un iterable con la función f1 aplicada a todos los elementos del iterable que se le pasa.
- Sintaxis: map(f1, iterable). De alguna manera es equivalente a un for. Se puede usar con clases.
- Como en el caso de filter, la función puede ser una lambda (muy usualmente) o creada por el usuario.
- También podemos pasar más de un iterable. En ese caso, cada iterable actúa como un parámetro y hay que tenerlo en cuenta en la función.

```
# defino una función map
datos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(datos)

d1 = list(map(lambda x:
3*x+1, datos)) # Esta función
multiplica por 3 y suma 1
print(d1)

d2= list(map(lambda x, y:
5*y-3*x, datos,d1))
print(d2) # map con más de un
parámetro
```

```
class Person():
    def __init__(self, nombre, edad):
        self.name = nombre
        self.age = edad
    def __str__(self):
        return "(" + self.name + ", " + str(self.age) + ")"

# Map con la clase Person
personas = [Person("Luis", 23), Person("Alberto", 19),
Person("Ana", 57)]

for p in personas:
    print(p, end=" // ")

edades = list(map(lambda p: p.age, personas ))
print(edades) # imprime las edades de esas personas
```

36. Funciones de alto nivel: reduce

Reduce

- No está implementada en el protocolo de Python 3. Hay que importarla del módulo **functools**. Da un resultado global según la función que se le pase.
- Su sintaxis es: `functools.reduce(function, iterable[, initializer])`. Opcionalmente usa un valor inicial.
- Una opción obvia es obtener la suma de todos los datos:

```
from functools import reduce

datos = [i for i in range(1,101)]
print(datos)

suma = reduce(lambda total, x:
total+x, datos)
print(suma)
```

```
# Uso de la función de alto nivel reduce con la clase Person

from functools import reduce
class Person():

    def __init__(self, nombre, edad):
        self.name = nombre
        self.age = edad

    def __str__(self):
        return "(" + self.name + ", " + str(self.age) + ")"

personas = [Person("Luis", 23), Person("Alberto", 19),
Person("Ana", 57)]

total_edad = reduce(lambda total, p: total + p.age, personas, 0)
# necesita el valor inicial 0 ¿¿??
promedio = total_edad // len(personas)
print("La edad promedio es: ",promedio)
```