

Curso Introducción a la programación con **Python3**


OpenWebinars

Presentación

José Domingo Muñoz Rodríguez

www.josedomingo.org

@Pledin_JD



**Curso Introducción a la
programación con Python3**

**Introducción a la
programación**


OpenWebinars

Resolución de problemas

- Un **programador** es una persona que **resuelve problemas**, y para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un **modo riguroso y sistemático**:
- Análisis
- Diseño
- Codificación
- Ejecución y validación del programa

Desarrollo de programas

Fases del ciclo de desarrollo de programas

Análisis

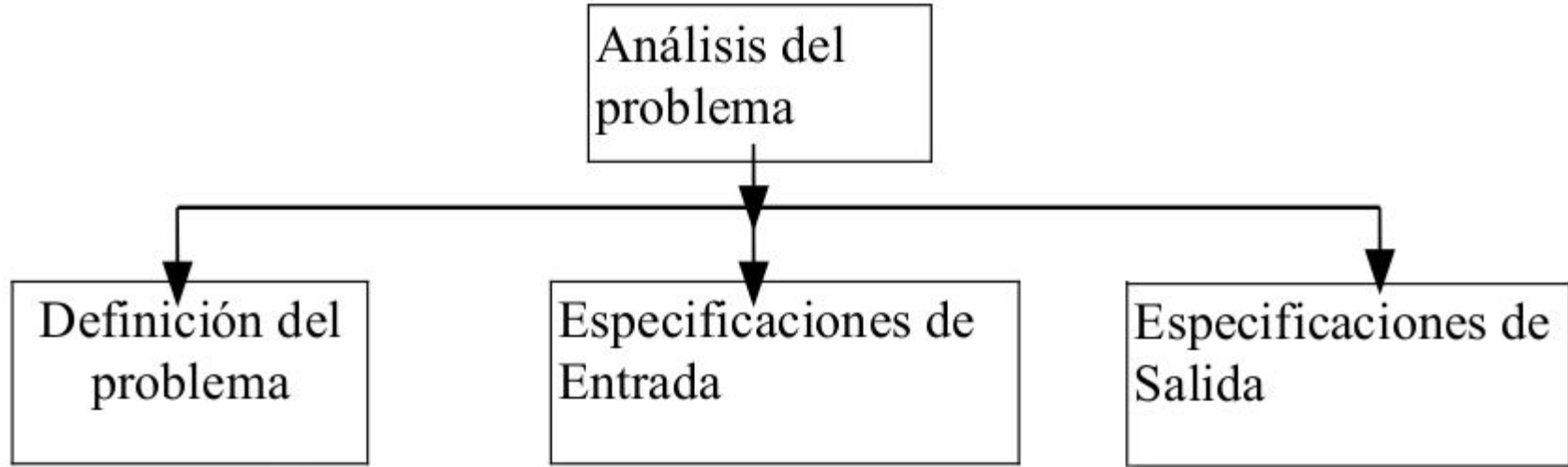
Diseño

Codificación

Pruebas e
Implementación

- **Análisis:** Entender el problema.
- **Diseño:** Creamos el algoritmo: pseudocódigo.
- **Codificación:** Escribir el algoritmo en un lenguaje de programación (Código fuente).
- Ejecución y validación: Comprobamos que el programa resuelve el problema planteado.

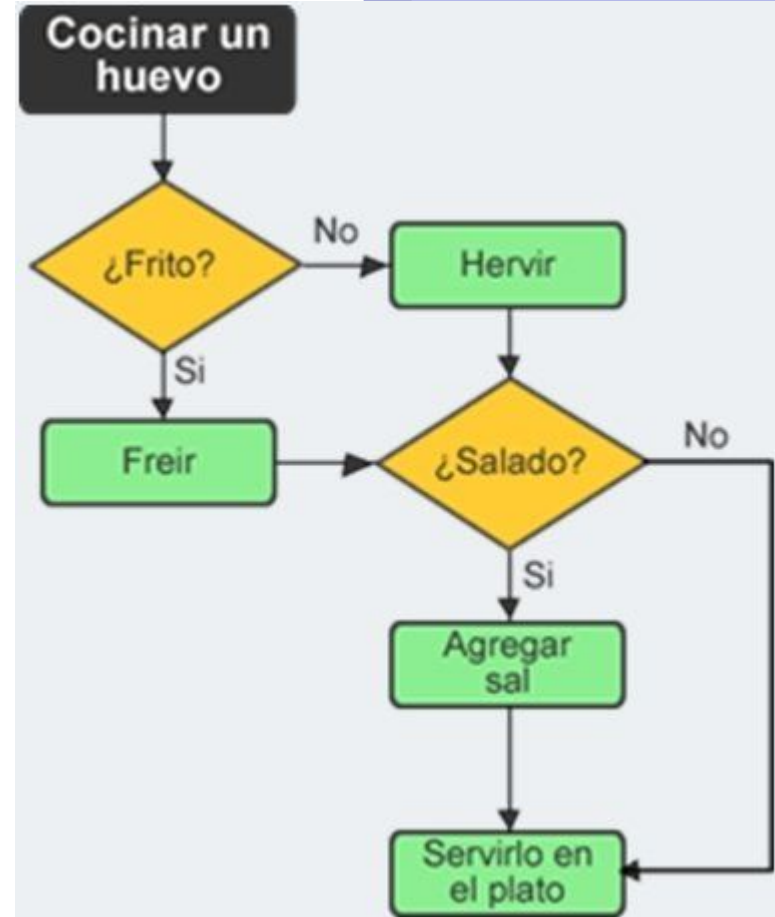
Análisis del problema



Diseño de algoritmos

- Un **algoritmo** es un conjunto de acciones que especifican la secuencia de operaciones realizar, en orden, para resolver un problema.

- Debe ser preciso
- Debe estar definido
- Debe ser finito



Diseño de algoritmos

- Un **diagrama de flujo** es una de las técnicas de representación gráfica de algoritmos más antiguas.
- El **pseudocódigo**, nos permite una aproximación del algoritmo al lenguaje natural y por tanto una redacción rápida del mismo.

Fórmulas:

$$a = 3.1416 * r^2$$

$$l = 2 * 3.1416 * r$$

Pseudocódigo y Diagrama de flujo

Inicio

Leer r

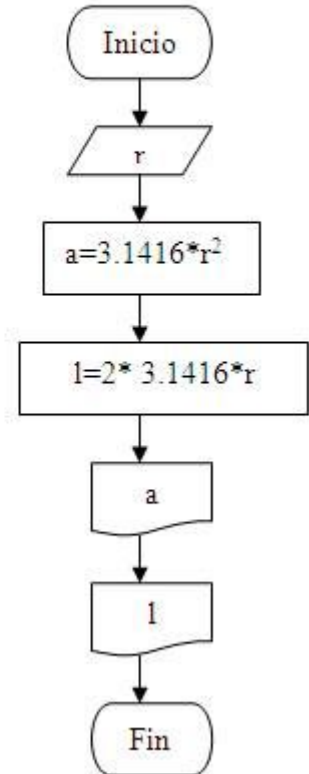
$$a = 3.1416 * r^2$$

$$l = 2 * 3.1416 * r$$

Escribir a

Escribir l

Fin algoritmo



Sistemas Informáticos



- El procesamiento de los datos lo hace un ordenador.
- El algoritmo se describe mediante un programa.
- **Programa:** Conjunto ordenado de instrucciones que se dan al ordenador indicándole las operaciones o tareas que ha de realizar para resolver un problema.
- Para escribir programas utilizamos **lenguajes de programación**.

Lenguajes de programación

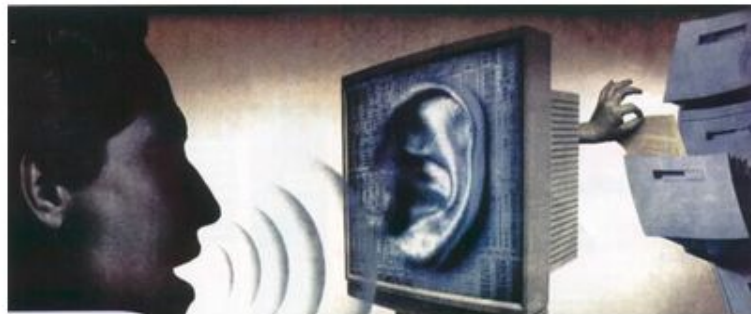


¿Qué lenguaje entiende...

...el ordenador?

...el programador?

```
11001010 00010111 11110101 00101011
00010111 11110101 00101011 00101011
11001010 00010111 11110101 00101011
00010111 11110101 00101011 00101011
11001010 11110101 00101011 00101011
11001010 11001010 11110101 00101011
11001010 11110101 00101011 00101011
11001010 00010111 11110101 00101011
00010111 11110101 00101011 00101011
11001010 11110101 00101011 00101011
```



Lenguaje máquina

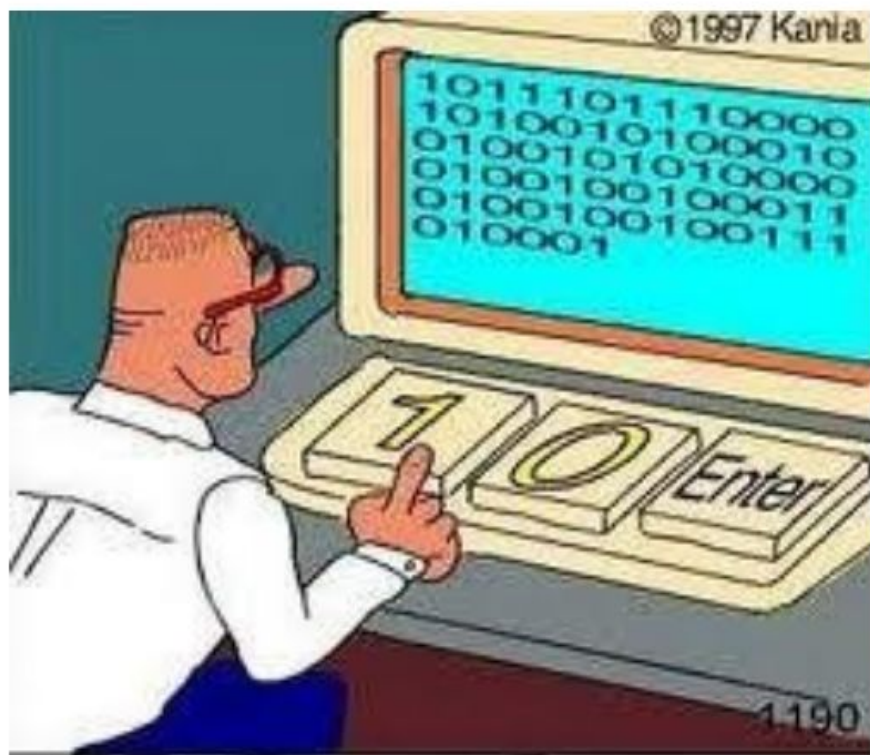


Lenguaje natural

Lenguaje máquina

El único que entiende directamente la CPU del ordenador

- Depende del modelo de ordenador
- Repertorio de instrucciones reducido (operaciones muy elementales)
- Muy difícil programar en él (en binario, con cadenas de ceros y unos)



LOS VERDADEROS PROGRAMADORES
PROGRAMAN EN BINARIO

Lenguaje ensamblador

Equivalente al lenguaje máquina, cada línea de código se traduce en una instrucción para la máquina.

- Le asocia mnemónicos a las operaciones que entiende la CPU
- Repertorio de instrucciones reducido (operaciones muy elementales)
- Programas difíciles de entender

004012A7	90	NOT	
004012A8	53	PUSH	EBX
004012A9	56	PUSH	ESI
004012AA	57	PUSH	EDI
004012AB	8BF2	MOV	ESI,EDX
004012AD	8BD8	MOV	EBX,EBX
004012AF	85F6	TEST	ESI,ESI
004012B1	8BFB	MOV	EDI,EBX
004012B3	74 35	JE	SHORT RTRACE.004012EA
004012B5	6A 04	PUSH	4
004012B7	68 00100000	PUSH	1000
004012BC	68 00100000	PUSH	1000
004012C1	53	PUSH	EBX
004012C2	E8 15870000	CALL	<JMP.&KERNEL32.VirtualAlloc>
004012C7	85C0	TEST	EAX,EAX
004012C9	75 0F	JNZ	SHORT RTRACE.004012DA
004012CB	8BD3	MOV	EDX,EBX
004012CD	8BC7	MOV	EAX,EDI
004012CF	2BD7	SUB	EDX,EDI
004012D1	E8 1E000000	CALL	RTRACE.004012F4
004012D6	33C0	XOR	EAX,EAX
004012D8	EB 15	JMP	SHORT RTRACE.004012EF
004012DA	81C3 00100000	ADD	EBX,1000
004012E0	81EE 00100000	SUB	ESI,1000
004012E6	85F6	TEST	ESI,ESI
004012E8	75 CB	JNZ	SHORT RTRACE.004012B5
004012EA	D8 01000000	MOV	EAX,1
004012EF	5F	POP	EDI
004012F0	5E	POP	ESI
004012F1	5B	POP	EBX
004012F2	C3	RET	EBX
004012F3	90	NOT	

Protect = PAGE_READWRITE
AllocationType = MEM_COMMIT
Size = 1000 (4096.)
Address
VirtualAlloc

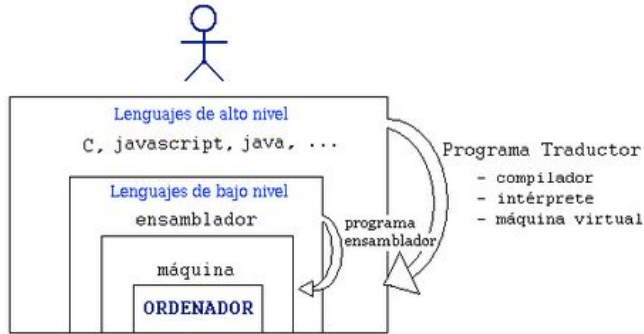
Lenguaje de alto nivel

Permiten que el programador exprese el procesamiento de datos de forma simbólica, sin tener en cuenta los detalles específicos de la máquina.

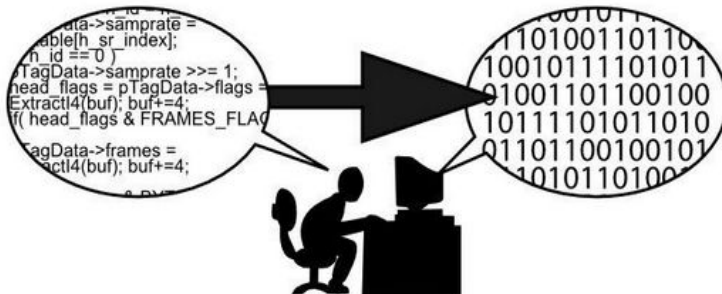
- Independientes del modelo de ordenador
- Proporcionan un mayor nivel de abstracción

```
writeln('ingresa un numero a');  
readln(a);  
writeln('ingresa un numero b');  
readln(b);  
writeln('ingresa un numero c');  
readln(c);  
  
if (a>b )and (a>c) then  
begin  
writeln('el mayor es a');  
end;  
begin;  
if (b>c) and (b>a) then  
writeln('el mayor es b');  
end;  
begin;  
if (c>a) and (c>b) then  
writeln('el mayor es c')  
end;  
  
readkey;
```

Traductores

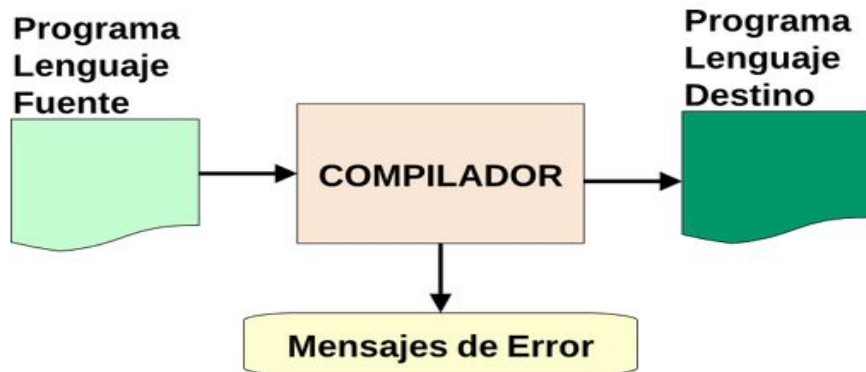


Los traductores transforman programas escritos en un lenguaje de alto nivel en programas escritos en código máquina.



Compilador

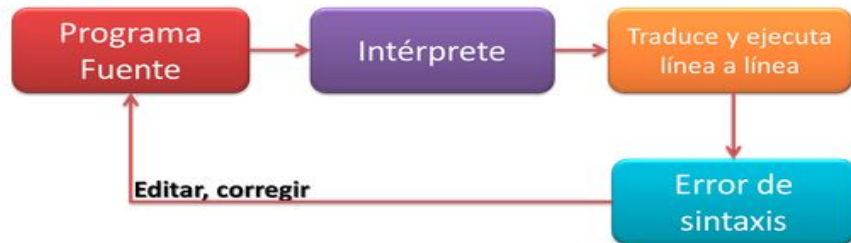
- Convierte un programa escrito en alto nivel (código fuente) a un programa máquina (código ejecutable).
- Para generar el código ejecutable el código no debe tener errores de sintaxis.
- Necesitamos un compilador para cada arquitectura y sistema operativo.
- Los programas ejecutables no son compatibles entre plataformas.
- Una vez generado el programa ejecutable, no es necesario tener el código fuente.



Ejemplos:
C, Pascal, ...

Interprete

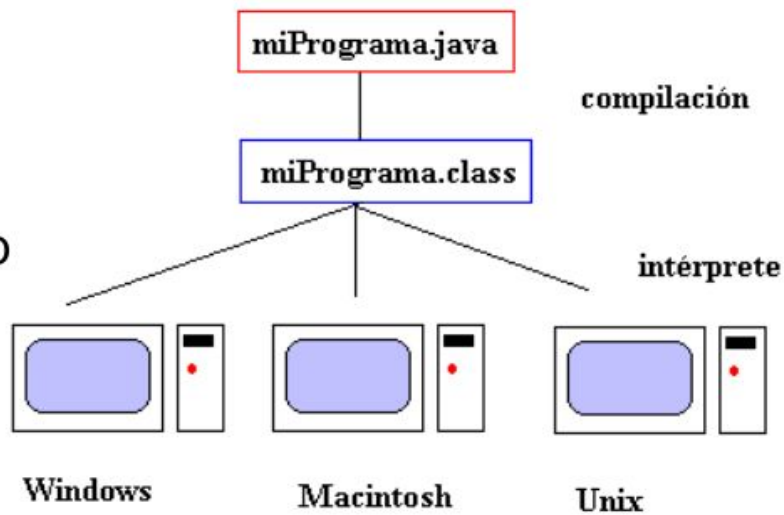
- La traducción y ejecución de código fuente a código máquina se hace línea por línea.
- Los errores de sintaxis aparecen cuando se interpreta la instrucción con error.
- Necesitamos el código fuente para ejecutar el programa.
- Los lenguajes interpretados suelen ser más lentos en su ejecución.



Ejemplos: python, php, ...

Máquina virtual

- La traducción se hace en dos pasos.
- Primero se compila el código fuente a un código intermedio (bytecode).
- Segundo, este bytecode se interpreta y ejecuta por una "máquina virtual".
- El bytecode es multiplataforma.
- Necesito una MV para cada plataforma.
- No necesito el código fuente.



Ejemplos: java, c#, ...



**Curso Introducción a la
programación con Python3**

**Introducción a
python3**


OpenWebinars

Características de Python

Python es un lenguaje:

- Interpretado
- Alto nivel
- Multiparadigma, ya que soporta **orientación a objetos, programación imperativa y programación funcional.**
- Multiplataforma
- Libre

¿Por qué elegir python?

- Porque es fácil de aprender
- Sintaxis muy limpia y sencilla
- Hay que escribir menos
- Obtienes resultados muy rápido
- Puedes programar con distintos paradigmas:
- Puedes programar distintos tipos de aplicaciones:
- Muchos usan Python (Google, Nokia, IBM). Es demandado.
- Gran cantidad de módulos, muchísimas funcionalidades.
- Una gran comunidad que apoya el proyecto.
- Viene preinstalado en la mayoría de sistemas



¿Por qué elegir python?

Índice TIOBE

Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	▲	Python	7.653%	+4.67%
4	3	▼	C++	7.394%	+1.83%
5	8	▲	Visual Basic .NET	5.308%	+3.33%
6	4	▼	C#	3.295%	-1.48%
7	6	▼	PHP	2.775%	+0.57%

<http://www.tiobe.com/tiobe-index/>

Instalación de python3

- Instalación en linux debian/Ubuntu
 - En Debian 9 Stretch, la versión es la 3.5.3
 - En Ubuntu 18.04 Bionic, la versión es la 3.6.3
- Instalación en Windows
 - Descargar instalador (paquete MSI)
- Instalación en Mac


Escribir y ejecutar programas python3

Uso del interprete

```
$ python3
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170118] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A partir de un fichero con el código fuente

```
$ python3 programa.py
```



¿IDE o
editor de
texto?

Mi primer programa en python3

Pseudocódigo

```
Proceso mayor_edad
  Definir edad como entero;
  Escribir "Dime tu edad:";
  Leer edad;
  Si edad >= 18 Entonces
    Escribir "Eres mayor de edad";
  FinSi
  Escribir "Programa terminado";
FinProceso
```

Python3

```
# Programa que pida la edad
# y diga si es mayor de edad.
edad=int(input("Dime tu
edad:"))
if edad >= 18:
    print("Eres mayor de edad")
print("Programa Terminado")
```



**Curso Introducción a la
programación con Python3**

Tipos de datos básicos


OpenWebinars

Literales, variables y expresiones

- **Literales enteros:** 3, 12, -23
- **Literales reales:** 12.3, 45.6
- **Literales cadenas:**

```
'hola que tal!'  
"Muy bien"  
'''Podemos \n  
ir al cine'''
```

Variables:

```
>>> var = 5  
>>> var  
5
```

Expresiones

```
a + 7  
(a ** 2) + b
```

Operadores

- Operadores aritméticos: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
- Operadores de cadenas: `+`, `*`
- Operadores de asignación: `=`
- Operadores de comparación: `==`, `!=`, `>=`, `>`, `<=`, `<`
- Operadores lógicos: `and`, `or`, `not`
- Operadores de pertenencia: `in`, `not in`

Precedencia de operadores

1. Los paréntesis rompen la precedencia.
2. La potencia (**)
3. Operadores unarios (+ -)
4. Multiplicar, dividir, módulo y división entera (* % //)
5. Suma y resta (+ -)
6. Operador binario AND (&)
7. Operadores binario OR y XOR (^ |)
8. Operadores de comparación (<= < > >=)
9. Operadores de igualdad (<> == !=)
10. Operadores de asignación (=)
11. Operadores de pertenencia (in, in not)
12. Operadores lógicos (not, or, and)

Tipos de datos

- ★ Tipos numéricos
 - Tipo entero (int)
 - Tipo real (float)
- ★ Tipos booleanos (bool)
- ★ Tipo de datos secuencia
 - Tipo lista (list)
 - Tipo tuplas (tuple)
- ★ Tipo de datos cadenas de caracteres
 - Tipo cadena (str)
- ★ Tipo de datos mapas o diccionario (dict)

```
>>> type(5)
<class 'int'>
>>> type(5.5)
<class 'float'>
>>> type([1,2])
<class 'list'>
>>> type(int)
<class 'type'>
```

Tipos de datos numéricos

```
>>> entero = 7
>>> type(entero)
<class 'int'>
>>> real = 7.2
>>> type (real)
<class 'float'>
```

Operadores

- +: Suma dos números
- : Resta dos números
- *: Multiplica dos números
- /: Divide dos números, el resultado es float.
- //: División entera
- %: Módulo o resto de la división
- ** : Potencia
- +, -: Operadores unarios positivo y negativo

Conversión de tipos

```
>>> a=int(7.2)
>>> a
7
>>> a=int("345")
>>> a
345
>>> b=float(1)
>>> b
1.0
>>> =float("1.234")
>>> b
1.234
>>> a=int("123.3")
```

Otras operaciones

```
>>> import math
>>> math.sqrt(9)
3.0
```

Funciones

```
>>> abs(-7)
7
>>> divmod(7,2)
(3, 1)
>>> hex(255)
'0xff'
>>> pow(2,3)
8
>>> round(7.567,1)
7.6
```


Tipo de datos booleanos

True, False

¿Qué valores se consideran falsos?

- False
- Cualquier número 0. (0, 0.0)
- Cualquier secuencia vacía ([], (), "")
- Cualquier diccionario vacío ({})

Operadores Lógicos

```
x or y  
x and y  
not x
```

Operadores de Comparación

```
== != >= > <= <
```

Trabajando con variables

Una variable es un identificador que referencia a un valor. No hay que declarar la variable antes de usarla, el tipo de la variable será el mismo que el del valor al que hace referencia. Por lo tanto su tipo puede cambiar en cualquier momento.

```
>>> a = 5
>>> a
5
>>> del a
>>> a
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
```

NameError: name 'a' is not defined

```
>>> a = 5
>>> a
5
>>> a = 8
>>> a
8
>>> a = a + 1
>>> a+=1
>>> 10
```

Otros operadores de asignación: +=, -=, *=, /=, %=, **=, //=.

Entrada y salida estándar

print

```
>>> print(1,2,3)
1 2 3
```

```
>>> print("Hola son las",6,"de la tarde")
Hola son las 6 de la tarde
```

```
>>> print("Hola son las "+str(6)+" de la tarde")
Hola son las 6 de la tarde
```

```
>>> print("%d %f %s" % (2.5,2.5,2.5))
2 2.500000 2.5
```

```
>>> print("El producto %s cantidad=%d precio=%.2f"%("cesta",23,13.456))
El producto cesta cantidad=23 precio=13.46
```

input

```
>>> nombre=input("Nombre:")
Nombre:jose
>>> nombre
'jose'
>>> edad=int(input("Edad:"))
Edad:23
>>> edad
23
```

Introducción a las cadenas de caracteres

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
que tal?'''
```

Comparación

```
"a">"A"
True

"informatica">"informacion"
False

"abcde">"abcdef"
False
```

Operadores

+: Concatenación

```
>>> "hola " + "que tal"
'hola que tal'
```

*: Repetición

```
>>> "abc" * 3
'abccabccabcc'
```

Indexación

```
>>> cadena = "josé"
>>> cadena[0]
'j'
>>> cadena[3]
'é'
```

Longitud

```
>>> cadena = "josé"
>>> len(cadena)
4
```

Pasando de pseudocódigo a python3

Pseudocódigo

```
Definir nombre Como Cadena;  
Escribir "Dime tu nombre:";  
Leer nombre;
```

Python3

```
# nombre = input("Dime tu nombre:")
```

Pseudocódigo

```
Definir numero Como Entero;  
Escribir "Dime un número entero:";  
Leer numero;
```

Python3

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

Pseudocódigo

```
Definir numero Como Real;  
Escribir "Dime un número real:";  
Leer numero;
```

Python3

```
numero = float(input("Dime un número real:"))
```

Pasando de pseudocódigo a python3

Pseudocódigo

```
Escribir "Hola ", nombre;
```

Python3

```
print("Hola", nombre)
```

Pseudocódigo

```
Escribir Sin Saltar var, " ";
```

Python3

```
print(var, " ", end="")
```

Pseudocódigo

```
numero <- 7;
```

Python3

```
numero = 7
```

Pseudocódigo

```
trunc(7/2)
```

Python3

```
7 // 2
```

Pasando de pseudocódigo a python3

Pseudocódigo

```
raiz(9)
```

Python3

```
import math  
math.sqrt(9)
```

Pseudocódigo

```
subcadena(cadena, 0, 0)
```

Python3

```
cadena[0]
```

Pseudocódigo

```
cadena3 <- concatenar(cadena1, cadena2)
```

Python3

```
cadena3 = cadena1 + cadena2
```

Pseudocódigo

```
cadena <- Mayusculas(cadena)
```

Python3

```
cadena = cadena.upper()
```



**Curso Introducción a la
programación con Python3**

Estructuras de control


OpenWebinars

Estructura de control: Alternativas

Alternativa simple

```
edad = int(input("Dime tu  
edad:"))  
if edad >= 18:  
    print("Eres mayor de edad")  
print("Programa terminado")
```

Alternativa doble

```
edad = int(input("Dime tu  
edad:"))  
if edad >= 18:  
    print("Eres mayor de edad")  
else:  
    print("Eres menor de edad")  
print("Programa terminado")
```

Estructura de control: Alternativas

Alternativa múltiple (pseudocódigo)

```
Proceso notas
    Definir nota como entero;
    Escribir "Dime tu nota:";
    Leer nota;
    Segun nota Hacer
        1,2,3,4: Escribir "Suspenso";
        5: Escribir "Suficiente";
        6,7: Escribir "Bien";
        8: Escribir "Notable";
        9,10: Escribir "Sobresaliente";
    De Otro Modo:
        Escribir "Nota incorrecta";
    FinSegun
    Escribir "Programa terminado";
FinProceso
```

Alternativa múltiple en python3

```
nota = int(input("Dime tu nota:"))
if nota >=1 and nota <= 4:
    print("Suspenso")
elif nota == 5:
    print("Suficiente")
elif nota == 6 or nota == 7:
    print("Bien")
elif nota == 8:
    print("Notable")
elif nota ==9 or nota == 10:
    print("Sobresaliente")
else:
    print("Nota incorrecta")
print("Programa terminado")
```

Estructura de control repetitivas: while

while

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    clave = input("Dime la clave:")
print("Bienvenido!!!")
print("Programa terminado")
```

Instrucciones: continue

```
cont = 0
while cont < 10:
    cont = cont + 1
    if cont % 2 != 0:
        continue
    print(cont)
```

Instrucciones: break

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    otra = input("¿Quieres introducir otra clave (S/N)?:")
    if otra.upper() == "N":
        break;
    clave = input("Dime la clave:")
if clave == secreto:
    print("Bienvenido!!!")
print("Programa terminado")
```

¿Y la estructura “Repetir” de pseudocódigo?

Pseudocódigo

```
Proceso login
  Definir secreto, clave como cadena;
  secreto <- "asdasd";
  Repetir
    Escribir "Dime la clave:";
    Leer clave;
    Si clave<>secreto Entonces
      Escribir "Clave incorrecta!!!";
    FinSi
  Hasta Que clave=secreto
  Escribir "Bienvenido!!!";
  Escribir "Programa terminado";
FinProceso
```

Python3

```
secreto = "asdasd"
while True:
    clave = input("Dime la clave:")
    if clave != secreto:
        print("Clave incorrecta!!!")
    if clave == secreto:
        break;
print("Bienvenido!!!")
print("Programa terminado")
```

Estructura de control repetitivas: for

Pseudocódigo

```
Proceso Contar
  Definir var como Entero;
  Para var<-1 Hasta 10 Hacer
    Escribir Sin Saltar var, " ";
  FinPara
FinProceso
```

Python3

```
for var in range(1,11):
    print(var, " ",end="")
```

Pseudocódigo

```
Proceso ContarDescendente
  Definir var como Entero;
  Para var<-10 Hasta 1 Con Paso -1 Hacer
    Escribir Sin Saltar var, " ";
  FinPara
FinProceso
```

Python3

```
for var in range(10,0,-1):
    print(var, " ",end="")
```

Pseudocódigo

```
Proceso ContarPares
  Definir var como Entero;
  Para var<-2 Hasta 10 Con Paso 2 Hacer
    Escribir Sin Saltar var, " ";
  FinPara
FinProceso
```

Python3

```
for var in range(2,11,2):
    print(var, " ",end="")
```

Usos específico de variables: contadores

```
cont = 0; ← El contador se inicializa a un valor inicial
for var in range(1, 6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        cont = cont + 1 ← El contador se incrementa
                           Un contador también se puede decrementar
print("Has introducido ", cont, " números pares.")
```

Un contador es una variable entera que la utilizamos para contar cuando ocurre un suceso.

Usos específico de variables: acumuladores

¿A qué valor inicializamos si vamos a acumular productos?

```
suma = 0; ← El acumulador se inicializa a un valor inicial
for var in range(1, 6):
    num = int(input("Dime un número:"))
    if num % 2 == 0: ← El acumulador se acumula
                    Podemos acumular también productos
        suma = suma + num ←
print("La suma de los números pares es ", suma)
```

Un acumulador es una variable numérica que permite ir acumulando operaciones. Me permite ir haciendo operaciones parciales.

Usos específico de variables: indicadores

```
indicador = False;
```

Se **inicializa** a un valor lógico:

```
for var in range(1,6):
```

No ha sucedido!!!!

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

```
    if num % 2 == 0:
```

```
        indicador = True
```

Cuando ocurre el suceso **cambiamos** su valor

```
if indicador:
```

```
    print("Has introducido algún número par")
```

```
else:
```

```
    print("No has introducido algún número par")
```

Al final debemos
comprobar **si el suceso
ha ocurrido**

Un indicador es una variable lógica, que usamos para recordar o indicar algún suceso.



**Curso Introducción a la
programación con Python3**

Tipos de datos secuencias


OpenWebinars

Tipo de datos cadenas de caracteres

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
que tal?'''
```

Operadores

Slice

```
>>> cadena[2:5]
'for'
>>> cadena[2:7:2]
'frá'
>>> cadena[:5]
'infor'
>>> cadena[5:]
'mática'
>>> cadena[-1:-3]
''
>>> cadena[::-1]
'acitámrofni'
```

+ : Concatenación

* : Repetición

Indexación

Longitud

Recorrido

```
>>> cadena = "informática"
>>> for caracter in cadena:
...     print(caracter,end="")
...
informática
```

Operadores de pertenencia

```
>>> "a" in cadena
True
>>> "b" in cadena
False
>>> "a" not in cadena
False
```

Operadores

Conversión de tipos

```
>>> cad = str(7.8)
>>> type(cad)
<class 'str'>
>>> print(cad)
7.8
```

Las cadenas de caracteres son inmutables

No podemos cambiar los caracteres de una cadena de la siguiente forma:

```
>>> cadena = "informática"
>>> cadena[2]="g"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Esto implica que al usar un método la cadena original no cambia, el método devuelve otra cadena modificada. Veamos un ejemplo:

```
>>> cadena = "informática"
>>> cadena.upper()
'INFORMÁTICA'
>>> cadena
'informática'
```

Si queremos cambiar la cadena debemos modificar su valor con el operador de asignación:

```
>>> cadena = cadena.upper()
>>> cadena
'INFORMÁTICA'
```

Métodos principales de cadenas

Métodos de formato

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?
```

```
>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo
```

```
>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO
```

```
>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO
```

```
>>> cad = "hola mundo"
>>> print(cad.title())
Hola Mundo
```

Métodos de búsqueda

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
```

```
>>> cad.count("a", 16)
2
>>> cad.count("a", 10, 16)
1
```

```
>>> cad.find("mi")
13
>>> cad.find("hola")
-1
```

Métodos principales de cadenas

Métodos de validación

```
>>> cad.startswith("b")
True
>>> cad.startswith("m")
False
>>> cad.startswith("m",13)
True
>>> cad.endswith("ción")
True
>>> cad.endswith("ción",0,10)
False
>>> cad.endswith("nido",0,10)
True
```

Otras funciones de validación:

```
isdigit(), islower(),
isupper(), isspace(),
istitle(),...
```

Métodos de sustitución

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print ("Estimado Sr. nombre apellido:" + replace(buscar, reemplazar_por))
Estimado Sr. Juan Pérez:

>>> cadena = "    www.eugeniabahit.com    "
>>> print(cadena.strip())
www.eugeniabahit.com
>>> cadena="0000000001230000000000"
>>> print(cadena.strip("0"))
123
```

Métodos de unión y división

```
>>> hora = "12:23:12"
>>> print(hora.split(":"))
['12', '23', '12']

>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print(texto.splitlines())
['Linea 1', 'Linea 2', 'Linea 3']
```

```
>>> lista1 = []
>>> lista2 = ["a",1,True]
```

Tipo de datos secuencia: listas

Operadores

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

Recorrido

```
>>> for num in lista:
...     print(num,end="")
123456
```

```
>>> lista2 = ["a","b","c","d","e"]
>>> for num,letra in zip(lista,lista2):
...     print(num,letra)
1 a
2 b
...
```

Operadores de pertenencia

```
>>> 2 in lista
True
>>> 8 not in lista
True
```

Concatenación (+)

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

Repetición (*)

```
>>> lista * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Indexación

```
>>> lista[3]
4
```

```
>>> lista1[12]
...
IndexError: list index out of range
```

```
>>> lista[-1]
6
```

Tipo de datos secuencia: listas

Operadores

Slice

```
>>> lista[2:4]
[3, 4]
>>> lista[1:4:2]
[2, 4]
>>> lista[:5]
[1, 2, 3, 4, 5]
>>> lista[5:]
[6, 1, 2, 3, 4, 5, 6]
>>> lista[::-1]
[6, 5, 4, 3, 2, 1, 6, 5, 4, 3, 2, 1]
```

Listas multidimensionales

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

>>> for fila in tabla:
...     for elem in fila:
...         print(elem,end="")
...     print()
```

Funciones

```
>>> lista1 = [20,40,10,40,50]
>>> len(lista1)
5

>>> max(lista1)
50

>>> min(lista1)
10

>>> sum(lista1)
150

>>> sorted(lista1)
[10, 20, 30, 40, 50]

>>> sorted(lista1,reverse=True)
[50, 40, 30, 20, 10]
```

Las listas son mutables

Los elementos de las listas se pueden modificar:

```
>>> lista1 = [1,2,3]
>>> lista1[2]=4
>>> lista1
[1, 2, 4]
>>> del lista1[2]
>>> lista1
[1, 2]
```

Los métodos de las listas modifican el contenido de la lista:

```
>>> lista1.append(3)
>>> lista1
[1, 2, 3]
```

¿Cómo se copian las listas?

Para copiar una lista en otra no podemos utilizar el operador de asignación:

```
>>> lista1 = [1,2,3]
>>> lista2 = lista1
>>> lista1[1] = 10
>>> lista2
[1, 10, 3]
```

El operador de asignación no crea una nueva lista, sino que nombra con dos nombres distintos a la misma lista, por lo tanto la forma más fácil de copiar una lista en otra es:

```
>>> lista1 = [1,2,3]
>>> lista2=lista1[:]
>>> lista1[1] = 10
>>> lista2
[1, 2, 3]
```


Métodos principales de listas

Métodos de inserción

```
>>> lista = [1,2,3]
>>> lista.append(4)
>>> lista
[1, 2, 3, 4]

>>> lista2 = [5,6]
>>> lista.extend(lista2)
>>> lista
[1, 2, 3, 4, 5, 6]

>>> lista.insert(1,100)
>>> lista
[1, 100, 2, 3, 4, 5, 6]
```

Métodos de eliminación

```
>>> lista.pop()
6
>>> lista
[1, 100, 2, 3, 4, 5]

>>> lista.pop(1)
100
>>> lista
[1, 2, 3, 4, 5]

>>> lista.remove(3)
>>> lista
[1, 2, 4, 5]
```

Métodos principales de listas

Métodos de ordenación

```
>>> lista.reverse()
>>> lista
[5, 4, 2, 1]

>>> lista.sort()
>>> lista
[1, 2, 4, 5]

>>> lista.sort(reverse=True)
>>> lista
[5, 4, 2, 1]

>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort()
>>> lista
['Hola', 'Que', 'Tal', 'hola', 'que', 'tal']
```

Métodos de búsqueda

```
>>> lista.count(5)
1

>>> lista.append(5)
>>> lista
[5, 4, 2, 1, 5]

>>> lista.index(5)
0

>>> lista.index(5,1)
4

>>> lista.index(5,1,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

Tipo de datos secuencia: tuplas

```
>>> tupla1 = ()  
>>> tupla2 = ("a",1,True)
```

Operadores

- Recorrido
- Operadores de pertenencia
- Concatenación (+)
- Repetición (*)
- Indexación
- Slice

Entre las funciones definidas podemos usar: **len**, **max**, **min**, **sum**, **sorted**.

Las tuplas son inmutables

```
>>> tupla = (1,2,3)  
>>> tupla[1]=5  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

Métodos de búsqueda: count, index

```
>>> tupla = (1,2,3,4,1,2,3)  
>>> tupla.count(1)  
2  
  
>>> tupla.index(2)  
1  
>>> tupla.index(2,2)  
5
```



**Curso Introducción a la
programación con Python3**

**Tipos de datos
mapas**

OpenWebinars

Tipo de datos mapas: diccionarios

```
>>> diccionario = {'one': 1, 'two': 2, 'three': 3}

>>> dict1 = {}
>>> dict1["one"]=1
>>> dict1["two"]=2
>>> dict1["three"]=3
```

Operadores

Longitud

```
>>> len(a)
3
```

Indexación

```
>>> a["one"]
1
>>> a["one"]+=1
>>> a
{'three': 3, 'one': 2, 'two': 2}
```

Eliminación

```
>>> del(a["one"])
>>> a
{'three': 3, 'two': 2}
```

Operadores de pertenencia

```
>>> "two" in a
True
```

Los diccionarios son mutables

Los elementos de los diccionarios se pueden modificar:

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> a["one"]=2
>>> del(a["three"])
>>> a
{'one': 2, 'two': 2}
```

¿Cómo se copian los diccionarios?

Para copiar un diccionario en otra no podemos utilizar el operador de asignación:

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> b = a
>>> del(a["one"])
>>> b
{'three': 3, 'two': 2}
```

En este caso para copiar diccionarios vamos a usar el método `copy()`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = a.copy()
>>> a["one"]=1000
>>> b
{'three': 3, 'one': 1, 'two': 2}
```

Métodos principales de diccionarios

Métodos de eliminación

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict1.clear()
>>> dict1
{}

```

Métodos de creación

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict2 = dict1.copy()

>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict2 = {'four':4,'five':5}
>>> dict1.update(dict2)
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}

```

Métodos principales de diccionarios

Métodos de retorno

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict1.get("one")
1
>>> dict1.get("four")
>>> dict1.get("four","no existe")
'no existe'

>>> dict1.pop("one")
1
>>> dict1
{'two': 2, 'three': 3}
>>> dict1.pop("four")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
>>> dict1.pop("four","no existe")
'no existe'
```

Recorridos de diccionarios

```
>>> for clave in dict1.keys():
...     print(clave)
one
two
three

>>> for valor in dict1.values():
...     print(valor)
1
2
3

>>> for clave,valor in dict1.items():
...     print(clave,"->",valor)
one -> 1
two -> 2
three -> 3
```




**Curso Introducción a la
programación con Python3**

Excepciones


OpenWebinars

Excepciones

Errores sintácticos

```
>>> while True print('Hello world')
      File "<stdin>", line 1
        while True print('Hello world')
                        ^
SyntaxError: invalid syntax
```

Errores en tiempo de ejecución (Excepciones)

```
>>> 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> a+4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> "2"+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Manejando excepciones

```
>>> while True:
...     try:
...         x = int(input("Introduce un número:"))
...         break
...     except ValueError:
...         print ("Debes introducir un número")
```

```
>>> try:
...     print (10/int(cad))
... except ValueError:
...     print("No se puede convertir a entero")
... except ZeroDivisionError:
...     print("No se puede dividir por cero")
... except:
...     print("Otro error")
```

Manejando excepciones

```
>>> try:
...     print (10/int(cad))
... except ValueError:
...     print("No se puede convertir a entero")
... except ZeroDivisionError:
...     print("No se puede dividir por cero")
... else:
...     print("Otro error")
```

```
>>> try:
...     result = x / y
... except ZeroDivisionError:
...     print("División por cero!")
... else:
...     print("El resultado es", result)
... finally:
...     print("Terminamos el programa")
```



**Curso Introducción a la
programación con Python3**

Introducción a los módulos


OpenWebinars

Introducción a los módulos

Módulo: Cada uno de los ficheros **.py** que nosotros creamos se llama módulo. Los elementos creados en un módulo (funciones, clases, ...) se pueden importar para ser utilizados en otro módulo. El nombre que vamos a utilizar para importar un módulo es el nombre del fichero.

Importación de módulos

```
>>> import math
>>> math.sqrt(9)
3.0
>>> from math import
sqrt
>>> sqrt(9)
3.0
```

Módulos de hora y fecha

- **time**
- ```
>>> time.sleep(1)
```
- **datetime**

## Módulos matemáticos

- **math**
- ```
>>> math.sqrt(9)
```
- **functions**
-
- **random**
- ```
>>> random.randint(1,10)
```

## Módulos del sistemas

- **os**
- ```
>>> os.system("clear")
```



**Curso Introducción a la
programación con Python3**

**Programación
estructurada**


OpenWebinars

Programación estructurada

Con la **programación modular y estructura**, un problema complejo debe ser dividido en varios **subproblemas más simples**, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo **suficientemente simples** como para poder ser resueltos fácilmente con algún algoritmo (divide y vencerás). Además el uso de subrutinas nos proporciona la **reutilización del código** y no tener **repetido instrucciones** que realizan la misma tarea.

```
>>> def factorial(n):  
...     """Calcula el factorial de un  
...     número"""  
...     resultado = 1  
...     for i in range(1,n+1):  
...         resultado*=i  
...     return resultado
```

```
>>>  
factorial(5)  
120
```

Para **llamar a una función** se debe utilizar su nombre y entre paréntesis los **parámetros reales** que se mandan. La llamada a una función se puede considerar una expresión cuyo valor y **tipo** es el retornado por la función.

Ámbito de variables

Variables locales: se declaran dentro de una función y no se pueden utilizar fuera de esa función

```
>>> def operar(a,b):  
...     suma = a + b  
...     resta = a - b  
...     print(suma,resta)  
...  
>>> operar(4,5)  
9 -1  
>>> resta
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'resta' is not defined
```

Variables globales: son visibles en todo el módulo.

```
>>> PI = 3.1415  
>>> def area(radio):  
...     return PI*radio**2  
...  
>>> area(2)  
12.566
```

Parámetros formales y reales

```
def CalcularMaximo(num1,num2):  
    if num1 > num2:  
        return num1  
    else:  
        return num2
```

Parámetros formales: Son las variables que recibe la función, se crean al definir la función.

Parámetros reales: Son la expresiones que se utilizan en la llamada de la función, sus valores se “copiarán” en los parámetros formales.

```
numero1 = int(input("Dime el número1:"))  
numero2 = int(input("Dime el número2:"))  
num_maximo = CalcularMaximo(numero1,numero2)  
print("El máximo es ",num_maximo;)
```

Paso de parámetro por valor o por referencia

En Python el **paso de parámetros es siempre por referencia**. El lenguaje no trabaja con el concepto de variables sino objetos y referencias.

Si se pasa un valor de **un objeto inmutable**, su valor **no se podrá cambiar** dentro de la función.

```
>>> def f(a):  
...     a=5  
>>> a=1  
>>> f(a)  
>>> a  
1
```

Sin embargo si pasamos **un objeto de un tipo mutable**, si **podremos cambiar** su valor:

```
>>> def f(lista):  
...     lista.append(5)  
...  
>>> l = [1,2]  
>>> f(l)  
>>> l  
[1, 2, 5]
```

Devolución de información

Una función en python puede devolver información utilizando la instrucción **return**. La instrucción **return** puede devolver cualquier tipo de resultados, por lo tanto **es fácil devolver múltiples datos guardados en una lista, tupla o diccionario**.

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, **no es recomendable hacerlo en Python**.

LLamadas a una función

Cuando se **llama a una función** se tienen que indicar los **parámetros reales** que se van a pasar. **La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función.** Si la función no tiene una instrucción **return** el tipo de la llamada será **None**.

```
>>> def cuadrado(n):  
...     return n*n  
  
>>> a=cuadrado(2)  
>>> cuadrado(3)+1  
10  
>>> cuadrado(cuadrado(4))  
256  
>>> type(cuadrado(2))  
<class 'int'>
```

Una **función recursiva** es aquella que al ejecutarse hace **llamadas a ella misma**. Por lo tanto tenemos que tener "**un caso base**" que hace terminar el bucle de llamadas. Veamos un ejemplo:

```
def factorial(num):  
    if num==0 or num==1:  
        return 1  
    else:  
        return num * factorial(num-1)
```



**Curso Introducción a la
programación con Python3**

**Programación
orientada a objetos**


OpenWebinars

Introducción a la Programación orientada a Objetos

```
import math
class punto():
    """ Representación de un punto en el
    plano, los atributos son x e y
    que representan los valores de las
    coordenadas cartesianas."""

    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

    def mostrar(self):
        return str(self.x)+":"+str(self.y)

    def distancia(self, otro):
        """ Devuelve la distancia entre ambos puntos. """
        dx = self.x - otro.x
        dy = self.y - otro.y
        return math.sqrt((dx*dx + dy*dy))
```

Llamamos **clase** a la representación abstracta de un concepto. Las clases se componen de **atributos** y **métodos**.

Los **atributos** definen las características propias del objeto y modifican su estado.

Los **métodos** son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

El **constructor** `__init__`, que nos permite inicializar los atributos de objetos. Este método se llama cada vez que se crea una nueva instancia de la clase (objeto).

Todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. (**self**).

Introducción a la Programación orientada a Objetos

```
>>> punto1=punto()  
>>> punto2=punto(4,5)  
>>> print(punto1.distancia(punto2))  
6.4031242374328485
```

Para crear un objeto, utilizamos el nombre de la clase enviando como parámetro los valores que va a recibir el constructor.

```
>>> punto2.x  
4  
>>> punto2.x = 7  
>>> punto2.x  
7
```

Podemos acceder y modificar los atributos de objeto.

Encapsulamiento

La característica de no acceder o modificar los valores de los atributos directamente y utilizar métodos para ello lo llamamos **encapsulamiento**.

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self.__secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.__secreto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Alumno' object has no
attribute '__secreto'
```

Las variables que comienzan por un doble guión bajo `__` la podemos considerar como **atributos privados**.

Encapsulamiento

- En Python, las **propiedades (getters)** nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.
- Los métodos **setters** son métodos que nos permiten modificar los atributos a través de un método.

```
class circulo():
    def __init__(self,radio):
        self.radio=radio

    @property
    def radio(self):
        print("Estoy dando el radio")
        return self.__radio

    @radio.setter
    def radio(self,radio):
        if radio>=0:
            self.__radio = radio
        else:
            raise ValueError("Radio positivo")
            self.__radio=0
```

```
>>> c1=circulo(3)
>>> c1.radio
Estoy dando el radio
3
>>> c1.radio=4
>>> c1.radio=-1
Radio debe ser positivo
>>> c1.radio
0
```

Herencia

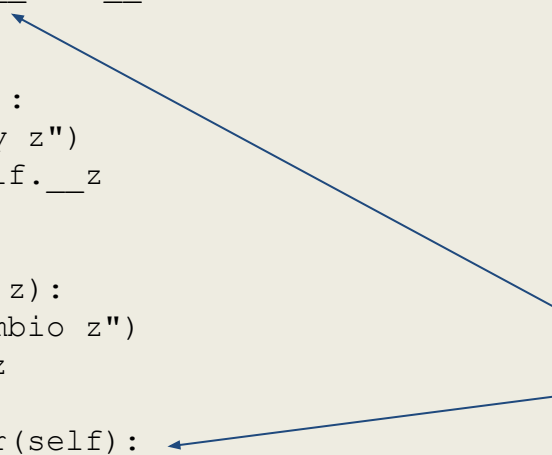
La clase **punto3d** hereda de la clase **punto** todos sus propiedades y sus métodos. En la clase hija hemos añadido la propiedad y el setter para el nuevo atributo **z**, y hemos modificado el constructor (sobrescritura) el método **mostrar** y el método **distancia**.

```
class punto3d(punto):
    def __init__(self, x=0, y=0, z=0):
        super().__init__(x, y)
        self.z=z
    @property
    def z(self):
        print("Doy z")
        return self.__z

    @z.setter
    def z(self, z):
        print("Cambio z")
        self.__z=z

    def mostrar(self):
        return super().mostrar()+":"+str(self.__z)

    def distancia(self, otro):
        dx = self.__x - otro.__x
        dy = self.__y - otro.__y
        dz = self.__z - otro.__z
        return (dx*dx + dy*dy + dz*dz)**0.5
```



```
>>> p3d=punto3d(1,2,3)
>>> p3d.x
1
>>> p3d.z
3
>>> p3d.mostrar()
1:2:3
>>> p3d.y = 3
```

La función **super()** me proporciona una referencia a la clase base. Nos permite acceder a los métodos de la clase madre.

Delegación

Llamamos **delegación** a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que delegará parte de sus funcionalidades.

```
class circulo():
    def __init__(self, centro, radio):
        self.centro=centro
        self.radio=radio

    def mostrar(self):
        return "Centro:{0}-Radio:{1}".format(self.centro.mostrar(), self.radio)
```

```
>>> c1=circulo(punto(2,3),5)
>>> print(c1.mostrar())
Centro:2:3-Radio:5
>>> c1.centro.x = 3
```