Python:

Estructuras de Datos:

Tuplas: son **inmutables** como las cadenas. Se definen con paréntesis o sin ellos. NO se puede añadir ningún elemento, no se pueden ordenar, …

```
casa = (33.456, 45.870)

dimensiones = 45, 67, 90
long, anch, alt = dimensiones
print('Dimensiones longitud: {} anchura: {} altura: {}'.format(long, anch, alt))
```

Listas: definidas entre corchetes [ ]. Diferentes tipos de datos. Listas anidadas. Funciones.

```
lista=[12,8.9,"Pepe",True]
```

Conjunto: elementos que NO se repiten. Van entre llaves { }

```
frutas={"plátano","manzana","kiwi"}
```

Diccionario: se definen entre llaves { } y es un conjunto de “clave”:valor, separados por comas.

```
dic_coordenadas={(100, 400): "Castellón", (300, 500): "Madrid", 4: "Pepe"}
dic_coordenadas[(300, 800)]="Asturias"
dic_coordenadas.pop(4)
```

Estructuras de Control:

if else

```
if 4<2: print(“mayor”);print(“varias líneas”);print(“pierde legibilidad”)
else: print(“No es menor”)
```

for

```
numeros = [1, 2, 3, 4]
for i in numeros:
        print(i)
```

while

```
num = 5
while num > 0:
        print(num)
        num -= 1
```

La sentencia **break** es una palabra reservada del lenguaje Python que podemos utilizar para terminar de manera inmediata la ejecución de una estructura de control de flujo `for` o `while`. El flujo de ejecución del programa pasará a la siguiente línea de código que se encuentre fuera de la estructura de control flujo.

La sentencia **continue** es otra palabra reservada de Python sin embargo, a diferencia de la sentencia break, no termina la ejecución completa de los bucles `for` y `while` sino que termina únicamente la ejecución de la iteración actual.

La sentencia **pass** es una palabra reservada de Python que nos permite definir el esqueleto de diferentes estructuras de diferentes tipos (funciones, if, while, for...) sin indicarle ninguna línea de código en el cuerpo de la estructura.

```
def funcion():
        pass

for color in colores:
        # Aqui voy a hacer algo que todavía no se
        pass
```

zip

Une las listas de entrada y devuelve tuplas.

```
a = [1, 2]
b = ["Uno", "Dos"]
c = zip(a, b)

print(list(c))
# [(1, 'Uno'), (2, 'Dos')]



paises = ["Brasil", "Chile", "China"]
capitales = ["Brasilia", "Santiago", "Pekin"]

for pais, capital in zip(paises, capitales):
        print(pais, capital)
```

Tiene la ventaja de que ocupa muy poca memoria. Para recorrer listas muy grandes, podemos utilizar zip ya que es mucho más rápido.

También se puede “deshacer” el zip con un * delante.

```
c = [(1, 'One'), (2, 'Two'), (3, 'Three')]
a, b = zip(*c)

print(a)  # (1, 2, 3)
print(b)  # ('One', 'Two', 'Three')
```

enumerate

Devuelve el índice y el valor de la lista que le paso.

```
estudiantes = ["María", "Pablo", "Carolina", "Luis"]
for index, nombre in enumerate(estudiantes):
        print(index, nombre)

cosas = [‘portátil’, ‘móvil’, ‘ordenador’, ‘libro’]
for index, nombre in enumerate(cosas):
        print(index, nombre)
```

Funciones: Una función va a consistir en un bloque de código que encapsula una tarea específica o un grupo de tareas relacionadas. Las funciones permiten dividir programas complejos en fragmentos más pequeños y modulares.

Cuando invocamos la función a través de su nombre y le proporcionamos los argumentos necesarios, el interprete de Python automáticamente recorre el código fuente que define esta función y ejecuta las tareas pertinentes sobre los argumentos proporcionados. Cuando la función termina, la ejecución vuelve a la línea de código donde se invocó la función. La función puede devolver un valor de retorno que podemos utilizar a lo largo de nuestro código.

Definiendo funciones personalizadas:

```
def <nombre_funcion>([<parámetros>]):
        <sentencia(s)>

def mi_funcion(arg1, arg2):
        print(arg1)
        print(arg2)

mi_funcion("Hola mundo", "Adios mundo")
```

Los **parámetros** se comportan como variables que están definidas de manera local a la función (únicamente en el cuerpo de la función

```
def mi_funcion():
        var = "variable 'var' dentro de la funcion"
        print(var)

var = "variable 'var' fuera de la funcion"
mi_funcion()
print(var)

def mi_funcion2(arg1, arg2):
        print(arg1)
        print(arg2

mi_funcion2("Hola mundo", "Adios mundo")
```

```
a,b = 2,3
def func():
	global a
	a,b=4,4
func()
print(a,b)
#Una de las cosas importantes es que los nombres que se encuentran en una región determinada puede ser accedidos desde otra externa pero no pueden ser actualizados o modificado
```

Sentencia **return**

Termina inmediatamente la ejecución de la función. Si definimos “return <valor>”, devuelve el valor. Si no especificamos valor, “None”. También podemos devolver varios valores.

Funciones propias de Python. (help, print, str, type, id, exec,…)

Función lambda

Es una función que NO tiene nombre, pero son muy útiles cuando queremos escribir funciones muy cortas y muy complejas. Escribimos la función de una manera más corta.

```
def multiplic(x):
	return x * 5

multiplic(3)

multiplic_lambda = lambda x: x * 5
multiplic_lambda(3)
```

Medición de rendimiento

timeit

IPython es como la forma genérica de los Jupyter Notebooks. Y tiene una serie de órdenes que podemos usar directamente. La orden **%timeit** permite saber el tiempo que ha tardado una orden en ejecutarse.

```
time = %timeit -n1 -r1 -o sum(range(10000000))
print(round(time.average,2),"seg")
```

Lo que hace es calcula el tiempo de ejecutar la orden `sum(range(1000000000))` y almacena cuanto ha tardado en `time`

El problema de usar **%timeit** es que lo que ejecutas no puede devolver un resultado. Es decir que no podríamos saber el resultado de la suma.

perf_counter

Solucionar el problema anterior. Podemos simplemente medir nosotros el tiempo que tarda un método en ejecutarse con la función **perf_counter**. El resultado se muestra en segundos.

```
from time import perf_counter

t = perf_counter()
resultado=sum(range(10000000))
t=perf_counter()-t

print(resultado)
print(round(t,2),"seg")
```

Funciones dentro de otras funciones:

```
def operaciones(op):
    def suma(a, b):
        return a + b
    def resta(a, b):
        return a - b

    if op == "suma":
        return suma
    elif op == "resta":
        return resta

funcion_suma = operaciones("suma")
print(funcion_suma(5, 7)) # 12

funcion_suma = operaciones("resta")
print(funcion_suma(5, 7)) # -2
```

Todo en Python es un objeto, incluso una función:

```
def di_hola():
    print("Hola")

f1 = di_hola() # Llama a la función
f2 = di_hola   # Asigna a f2 la función

print(f1)      # None, di_hola no devuelve nada
print(f2)      # <function di_hola at 0x1077bf158>

#f1()          # Error! No es válido
f2()           # Llama a f2, que es di_hola()

del f2         # Borra el objeto que es la función
#f2()          # Error! Ya no existe

di_hola()      # Ok. Sigue existiendo
```

Manejo de **excepciones**:

```
# try   except   finally.  También else:

try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
else:
  print("Nothing went wrong")

#otro ejemplo
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

Decoradores. Funciones que modifican el comportamiento de otras:

```
def mi_decorador(funcion):
    def nueva_funcion(a, b):
        print("Se va a llamar")
        c = funcion(a, b)
        print("Se ha llamado")
        return c
    return nueva_funcion

@mi_decorador
def suma(a, b):
    print("Entra en funcion suma")
    return a + b

suma(5,8)

@mi_decorador
def resta(a ,b):
    print("Entra en funcion resta")
    return a - b

resta(5, 3)

#también se puede hacer con suma resta:
sum_decorada = mi_decorador(suma)
sum_decorada(5,8)

#Muy usado logger. Escribir en un fichero la información.
def log(fichero_log):
    def decorador_log(func):
        def decorador_funcion(*args, **kwargs):
            with open(fichero_log, 'a') as opened_file:
                output = func(*args, **kwargs)
                opened_file.write(f"{output}\n")
        return decorador_funcion
    return decorador_log

@log('ficherosalida.log')
def suma(a, b):
    return a + b

@log('ficherosalida.log')
def resta(a, b):
    return a - b
```

```
@log('ficherosalida.log')
def multiplicadivide(a, b, c):
    return a*b/c

suma(10, 30)
resta(7, 23)
multiplicadivide(5, 10, 2)


#Otro ejemplo: Autorización.
autenticado = True

def requiere_autenticación(f):
    def funcion_decorada(*args, **kwargs):
        if not autenticado:
            print("Error. El usuario no se ha autenticado")
        else:
            return f(*args, **kwargs)
    return funcion_decorada

@requiere_autenticación
def di_hola():
    print("Hola")

di_hola()
```

Un **Namespace** es un mapeo de nombres a objetos. Los Namespaces se crean en diferentes momentos y tienen diferentes tiempos de vida.
El **Namespace por defecto** que se crea cuando el intérprete de Python se inicia, es el que contiene los nombres por defecto de Python. Este Namespace nunca se borra.

```
dir(__builtins_)
```

Namespace global para un módulo se crea cuando se lee la definición del módulo y, normalmente, dura hasta que el intérprete se cierra.

```
globals()
```

El **Namespace local** para una función o cualquier otra estructura de Python se crea cuando se llama a la función, y se borra cuando la función termina. Las invocaciones recursivas tienen cada una su propio Namespace.

```
def func():
    var_local_func = 5
    var_local_func = 10
    print(locals())

func()
```

El scope es una región de un programa en Python donde un Namespace es directamente accesible.

```
def func():
    var_local_func = 10
    print(var_local_func)

func()

#La variable `var_local_func` definida en el Namespace local de la funcion no es accedida desde un scope global.

var_local_func


#La variable `var_no_local_func` es visible desde el código de `func2` pero la variable `var_local_func2` no es visible desde el código de `func`
def func():
    var_no_local_func = 10
    def func2():
        var_local_func2 = 5
        print(var_no_local_func)
    func2()

#tampoco func2 desde func
ef func():
    var_no_local_func = 10
    print(var_local_func2)
    def func2():
        var_local_func2 = 5
    func2()
```