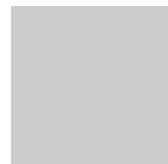


MÓDULO

4

Área: **NEGOCIOS**
Curso: **PROGRAMACIÓN BÁSICA (PYTHON)**
Módulo: **Estructuras avanzadas**



IPP

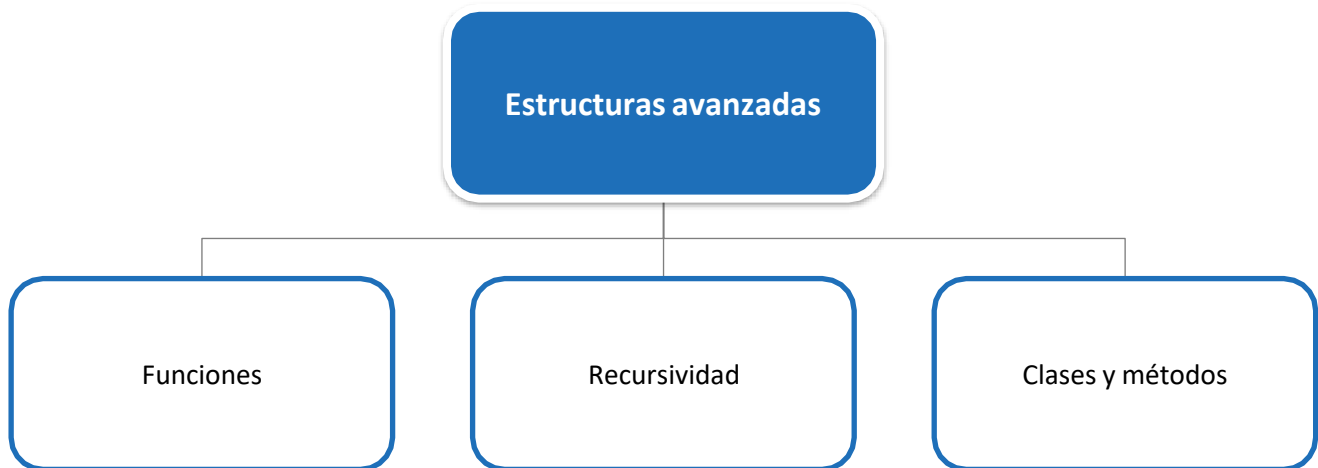
SUEÑA • APRENDE • CRECE

Índice

Introducción	1
1. Funciones	1
1.1. Estructura	1
a. Cómo llamar una función	2
1.2. Estructura de una función.....	5
1.3. Variables Globales.....	6
2. Recursividad	7
3. Clases y métodos	13
3.1. Clases	13
3.2. Métodos especiales	16
3.3. Librerías	16
7. Cierre	18



Mapa de Contenido



RESULTADO DE APRENDIZAJE DEL MÓDULO

Crea programas computacionales de mayor complejidad, utilizando estructuras avanzadas de datos y lenguaje, para representar la solución a un requerimiento escrito.

Introducción

En el mundo laboral enfocado en la programación, se trabaja por módulos, donde un programa es realizado por varios programadores. Cada programador se encarga de un módulo específico de este programa, generando clases con sus propios métodos y funciones. El objetivo es importar todas las clases en un solo archivo para poder ejecutar el programa y obtener la información necesaria.

Este enfoque también es muy útil cuando queremos reutilizar código. Por ejemplo, si somos programadores matemáticos y siempre tenemos que generar nuestras funciones de bisección, falsa posición, etc., lo mejor es crear nuestra propia clase y luego utilizar códigos de ella. De esta manera, nos despreocupamos de tener que pensar en cómo funciona un método de bisección o cómo programarlo, y nos enfocamos en la resolución del nuevo problema.

Desde la creación de la programación orientada a objetos (POO), la informática ha experimentado un gran cambio. Anteriormente, todo se programaba en un solo archivo con millones de líneas de código, lo que dificultaba encontrar errores o realizar cambios. Ahora, las clases o los archivos de programación siempre se encuentran en módulos más cortos y específicos, lo que facilita su lectura y mantenimiento. Estos módulos contienen el núcleo del programa o sus funciones principales.

1. Funciones

1.1. Estructura

Una función en Python es una estructura que agrupa actividades con un objetivo determinado. Se define mediante el comando "def" seguido del nombre de la función y los parámetros que recibe, separados por comas y encerrados entre paréntesis. La definición de la función termina con el símbolo ":" y el cuerpo de la función se escribe en sangría.

Una de las características más importantes de una función son los argumentos, que son variables u objetos que se le entregan a la función para que esta los utilice en su trabajo. Los argumentos pueden ser de diferentes tipos, como enteros, flotantes, cadenas, etc.

Por ejemplo, la siguiente función en Python realiza la suma de dos números y muestra el resultado por consola:

```
funciones.py > ...  
1  def suma(arg_1, arg_2):  
2      c = arg_1 + arg_2  
3      print(c)  
4
```

La función recibe dos argumentos (arg_1 y arg_2) y utiliza la operación suma para calcular el resultado y mostrarlo por consola.

Si queremos crear una función que retorne el resultado en vez de mostrarlo por consola, utilizamos la sentencia "return" para indicar el valor que debe ser retornado. Por ejemplo:

```
6  
7  def suma(arg_1, arg_2):  
8      c = arg_1 + arg_2  
9      return c  
10
```

De esta manera, podemos definir funciones que retornan información y funciones que no retornan información (conocidas como funciones void). Es importante tener en cuenta que, si una función retorna información, la variable que recibe esta información debe ser del mismo tipo que el valor retornado por la función. Por ejemplo, si la función retorna un número, la variable que recibe el resultado debe ser de tipo int o float.

a. Cómo llamar una función

Las funciones que retornan información tienen que ser llamadas por una variable, por ejemplo: resultado = suma(5, 6).

Y las funciones que no retornan nada pueden ser llamadas directamente, como el print().

Si se dan cuenta, cuando se utiliza el comando print(), éste se coloca directamente en el código. En cambio, cuando se utiliza un comando para convertir, como str(texto), éste debe ser igualado a una variable

Ahora que tenemos funciones declaradas, el funcionamiento de nuestro programa es algo distinto, ya que tenemos que indicarle a Python cuál es el núcleo del programa mediante una función `main()`. La estructura será la siguiente:

```
funciones.py > ...
1 def funcion1():
2     # acciones
3
4 def funcion2():
5     # acciones
6
7 if __name__ == "__main__":
8     # desde aquí Python comienza a ejecutar el programa de la manera que conocemos.
9
```

Como pueden darse cuenta ese código genera un error y se puede ver en rojo, generalmente significa que faltan sangrías (indentación) en el código. En Python, la sangría es muy importante ya que define bloques de código y estructuras de control. Asegúrate de que el código se vea así, por ejemplo, crearemos una calculadora en visual studio code.

```
funciones.py > ...
1 # Definimos funciones para realizar diferentes operaciones matemáticas
2 def sumar(a , b):
3     c = a + b
4     return c
5
6 def restar(a , b):
7     c = a - b
8     return c
9
10 def multiplicar(a , b):
11     c = a * b
12     return c
13
14 def dividir(a , b):
15     c = a / b
16     return c
17
18 def resto(a , b):
19     c = a % b
20     return c
21
22 def elevar(a , b):
23     c = a ** b
24     return c
25
26 # En este bloque, llamamos a las funciones definidas anteriormente y guardamos los resultados en variables
27 if __name__ == "__main__":
28     numero_1 = 10
29     numero_2 = 3
30
31     s = sumar(numero_1,numero_2)
32     r = restar(numero_1,numero_2)
33     m = multiplicar(numero_1,numero_2)
34     d = dividir(numero_1,numero_2)
35     mo = resto(numero_1,numero_2)
36     e = elevar(numero_1,numero_2)
37
38     # Imprimimos los resultados por consola
39     print('la suma es : ' + str(s))
40     print('la resta es : ' + str(r))
41     print('la multiplicacion es : ' + str(m))
42     print('la division es : ' + str(d))
43     print('el resto es : ' + str(mo))
44     print('el numero a elevador b es : ' + str(e))
45
```

Y este sería el resultado:

```
User@DESKTOP-DVUM855 MINGW64 ~/Desktop/IPP 2023/Python
$ C:/Users/User/AppData/Local/Programs/Python/Python39/python.exe "c:/Users/User/Desktop/IPP 2023/Python/funciones.py"
la suma es :13
la resta es :7
la multiplicacion es :30
la division es :3.3333333333333335
el resto es :1
el numero a elevador b es :1000
```

Copiamos el mismo código, pero en thonny y el programa se vería igual mostrando la misma salida que en visual studio.

The screenshot shows the Thonny IDE interface. The main editor displays a Python script named 'calculadora.py'. The script defines several functions: 'sumar', 'restar', 'multiplicar', 'dividir', 'resto', and 'elevar'. It then uses these functions to perform calculations on the numbers 10 and 3, storing the results in variables 's', 'r', 'm', 'd', 'mo', and 'e'. Finally, it prints out the results in a specific format. The Shell window at the bottom shows the execution of the script, displaying the same output as the terminal window in the previous image. On the right side, the 'Variables' pane lists the current state of the program's variables, including their names and values.

```
calculadora.py
def sumar(a, b):
    c = a + b
    return c
def restar(a, b):
    c = a - b
    return c
def multiplicar(a, b):
    c = a * b
    return c
def dividir(a, b):
    c = a / b
    return c
def resto(a, b):
    c = a % b
    return c
def elevar(a, b):
    c = a ** b
    return c

if __name__ == "__main__":
    numero_1 = 10
    numero_2 = 3
    s = sumar(numero_1, numero_2)
    r = restar(numero_1, numero_2)
    m = multiplicar(numero_1, numero_2)
    d = dividir(numero_1, numero_2)
    mo = resto(numero_1, numero_2)
    e = elevar(numero_1, numero_2)

    #mostremos la informacion obtenida
    print('la suma es : ' + str(s))
    print('la resta es : ' + str(r))
    print('la multiplicacion es : ' + str(m))
    print('la division es : ' + str(d))
    print('el resto es : ' + str(mo))
    print('el numero a elevador b es : ' + str(e))

Shell
>>>
>>> %Run calculadora.py
la suma es :13
la resta es :7
la multiplicacion es :30
la division es :3.3333333333333335
el resto es :1
el numero a elevador b es :1000
>>>
```

Name	Value
d	3.3333333333333335
dividir	<function dividir at 0x102407d9
e	1000
elevar	<function elevar at 0x102407ea
m	30
mo	1
multiplicar	<function multiplicar at 0x10240
numero_1	10
numero_2	3
r	7
restar	<function restar at 0x102407c8
resto	<function resto at 0x102407e18
s	13
sumar	<function sumar at 0x102407bf

1.2. Estructura de una función

Dentro de una función se puede crear la cantidad de variables que se estime conveniente, en el caso anterior creamos en todas las funciones la variable `c`, pero esta variable es temporal, solo existe mientras se ejecuta esa función, es decir que al momento de retornar una variable o termine su ejecución, la variable dejara de existir, las únicas variables que reconoce Python son los argumentos que se le entrega. Estos argumentos son solo copia de los datos:

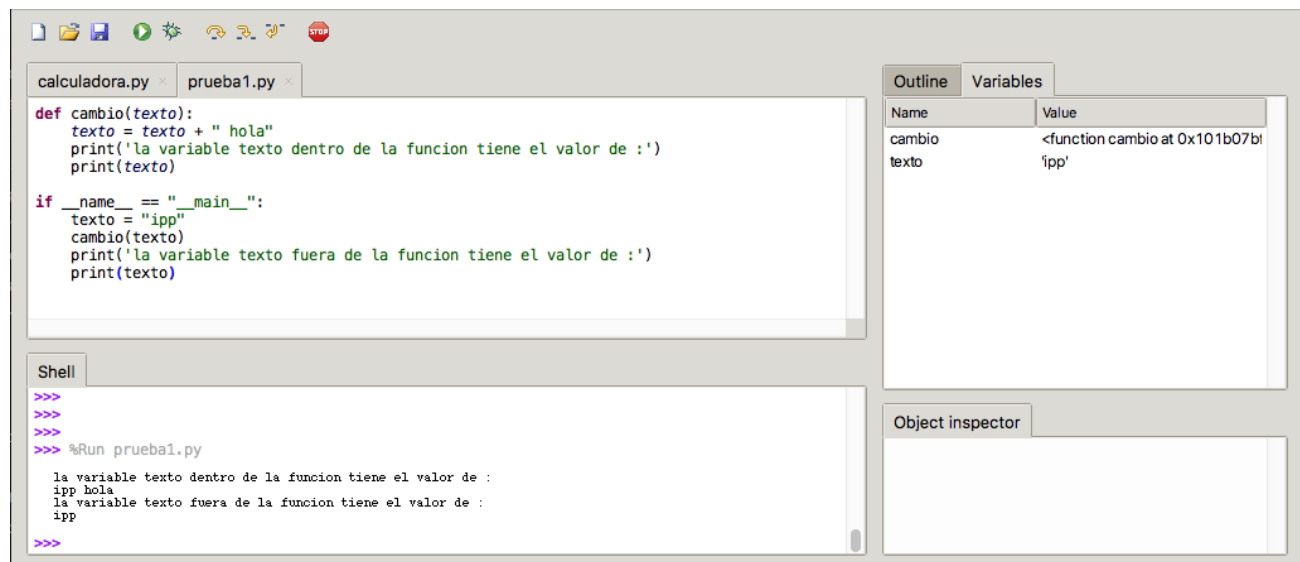
Ejemplo:

```
funciones.py > ...
1  def cambio(texto):
2      texto = texto + " hola"    # Concatena la palabra " hola" a la variable texto
3      print('la variable texto dentro de la función tiene el valor de :')
4      print(texto)
5
6  if __name__ == "__main__":
7      texto = "ipp"
8      cambio(texto)              # Se llama a la función cambio() pasándole como argumento la variable texto
9      print('la variable texto fuera de la función tiene el valor de :')
10     print(texto)               # Imprime el valor de la variable texto, que sigue siendo "ipp"
11
12
```

Al momento de ejecutar este programa la consola nos muestra lo siguiente:

```
User@DESKTOP-DVUM85S MINGW64 ~/Desktop/IPP 2023/Python
$ C:/Users/User/AppData/Local/Programs/Python/Python39/python
la variable texto dentro de la función tiene el valor de:
ipp hola
la variable texto fuera de la función tiene el valor de:
ipp hola
```

Vista en Thonny el programa.



1.3. Variables Globales

Las variables globales son variables que se definen fuera de una función y pueden ser accedidas desde cualquier lugar del código, incluyendo funciones y el bloque principal del programa (conocido como "main"). Si una función realiza un cambio en una variable global, este cambio afectará a todo el programa.

En Python, se utiliza el comando `global` para indicar que una variable dentro de una función es una variable global. De esta manera, podemos modificar el valor de la variable global desde dentro de la función.

La estructura típica de un programa en Python que utiliza variables globales es la siguiente:

- Definición de variables globales
- Definición de funciones
- Bloque principal del programa (opcional)
- Ejecución del código

Realizaremos el mismo ejemplo anterior, pero dejando el texto afuera como variable global.

```
funciones.py > ...
1  texto_global = "ipp"
2
3  def cambio():
4      global texto_global
5      texto_global = texto_global + " hola"
6      print('la variable texto dentro de la función tiene el valor de :')
7      print(texto_global)
8
9  if __name__ == "__main__":
10     cambio()
11     print('la variable texto fuera de la función tiene el valor de :')
12     print(texto_global)
13
```

La consola nos muestra lo siguiente:

Print Output:

```
la variable texto dentro de la función tiene el valor de :
ipp hola
la variable texto fuera de la función tiene el valor de :
ipp hola
```

En este ejemplo, creamos una variable global llamada **texto_global** y una función llamada **cambio()** que modifica el valor de la variable global agregando la cadena de texto " hola". Para indicar que **texto_global** es una variable global dentro de la función **cambio()**, utilizamos el comando **global**.

En el bloque principal del programa, llamamos a la función **cambio()** y luego imprimimos el valor de la variable global **texto_global**. Como resultado, obtenemos dos impresiones en la consola: una que muestra el valor de la variable global dentro de la función (que debería ser "ipp hola") y otra que muestra el valor de la variable global fuera de la función (que también debería ser "ipp hola").

Es importante destacar que, aunque podemos utilizar variables globales en Python, es buena práctica evitar su uso en la medida de lo posible, ya que pueden hacer que el código sea más difícil de entender y mantener. En general, se recomienda utilizar variables locales dentro de funciones y pasar argumentos entre funciones en lugar de utilizar variables globales

2. Recursividad

DEFINICIÓN

La recursividad en programación se refiere a una técnica en la que una función se llama a sí misma para resolver un problema o realizar una tarea en particular. La función que se llama a sí misma se conoce como una función recursiva y la recursividad es una forma común de resolver problemas en programación, especialmente en algoritmos que implican estructuras de datos como árboles y listas enlazadas.

En el caso de la función X, si se llama a sí misma dentro de su definición, entonces se está utilizando la técnica de recursividad. Cabe señalar que, al utilizar la recursividad, es importante tener en cuenta los casos base para evitar una llamada infinita y asegurar que la función se detenga en algún punto y regrese un resultado.

Los ascensores son un buen ejemplo de recursividad en la vida real. Los espejos en los ascensores crean una serie infinita de imágenes reflejadas, donde cada imagen reflejada es una repetición de la imagen anterior en una escala más pequeña.

Este efecto se produce porque cada vez que la luz se refleja en un espejo, crea una nueva imagen que se refleja en el siguiente espejo. Este proceso se repite una y otra vez, creando una serie infinita de imágenes que parecen extenderse hasta el infinito.

La recursividad en este ejemplo se puede entender como una función que se llama a sí misma (la luz reflejada) para crear una serie infinita de resultados (imágenes reflejadas). Aunque este ejemplo es más abstracto que la recursividad en programación nos ilustra cómo la recursividad puede ser una herramienta poderosa para resolver problemas y crear patrones interesantes en la vida real.

Realicemos un ejemplo.

¿Recuerdan cuando aprendimos sobre ciclos y programamos la función de Fibonacci? Bueno, ahora realizaremos la misma función, pero de manera recursiva.

Para recordar, una serie de Fibonacci es cuando se suman los dos números anteriores para obtener el siguiente número en la serie. La serie comienza con 1 1.

Aquí hay una versión de la función de Fibonacci escrita sin recursión:

Pero esto nos lleva a pensar que, si usamos una función recursiva, el programa se ejecutará indefinidamente y nunca se detendrá. Aquí es donde la condición de término es importante

IMPORTANTE

Es importante poder generar una condición de que cuando se ingrese un valor a la función y esta se cumpla, en vez de llamarse nuevamente, esta se retorne o termine, y en ese instante se comienza a devolver todo el código cerrando cada función que se abrió de sí mismo.

```
funciones.py > ...
1  def fibonacci(num):
2      a0 = 0
3      a1 = 1
4      if num == 0:
5          return a0
6      elif num == 1:
7          return a1
8      else:
9          for i in range(2, num):
10             fib = a0 + a1
11             a0 = a1
12             a1 = fib
13             return fib
14
```

En esta versión, la función fibonacci toma un argumento num que indica el número de la serie de Fibonacci que se desea calcular. La función comienza inicializando las variables a0 y a1 a los dos primeros números de la serie, 0 y 1, respectivamente. Luego, la función verifica si num es igual a 0 o 1 y devuelve los valores correspondientes si es así. En caso contrario, la función utiliza un bucle "for" para calcular los siguientes números en la serie, actualizando los valores de a0 y a1 en cada iteración y devolviendo el último número calculado en la serie.

Ahora la serie Fibonacci escrita con recursión:

```
funciones.py > ...  
1  def fib(num):  
2      if num <= 1:  
3          return num  
4      else:  
5          return fib(num - 1) + fib(num - 2)  
6
```

La función fib toma un argumento num que indica el número de la serie de Fibonacci que se desea calcular. La función primero verifica si num es menor o igual a 1. Si es así, devuelve num como resultado (ya que los dos primeros números en la serie de Fibonacci son 0 y 1).

Si num es mayor que 1, la función se llama a sí misma dos veces, pasando num-1 y num-2 como argumentos. Estas dos llamadas recursivas calculan los números anteriores en la serie de Fibonacci, y la función devuelve la suma de estos dos números como resultado.

Cada llamada recursiva reduce el valor de num en 1 o 2, lo que eventualmente lleva a que num sea igual a 0 o 1, en cuyo caso la función deja de llamarse a sí misma recursivamente y devuelve el valor correcto de la serie de Fibonacci.

Como funciona este código, es bien sencillo, pongamos como ejemplo que entregamos el número 5.

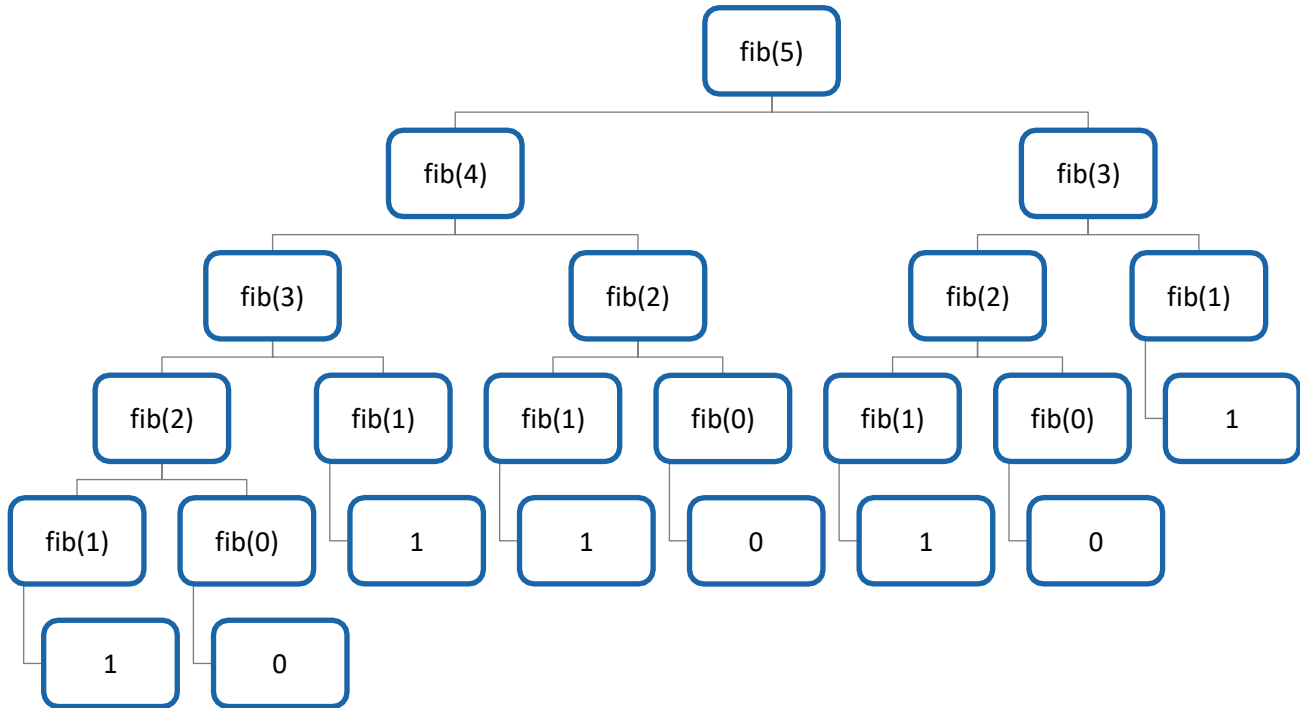
Paso a paso

fib(5)

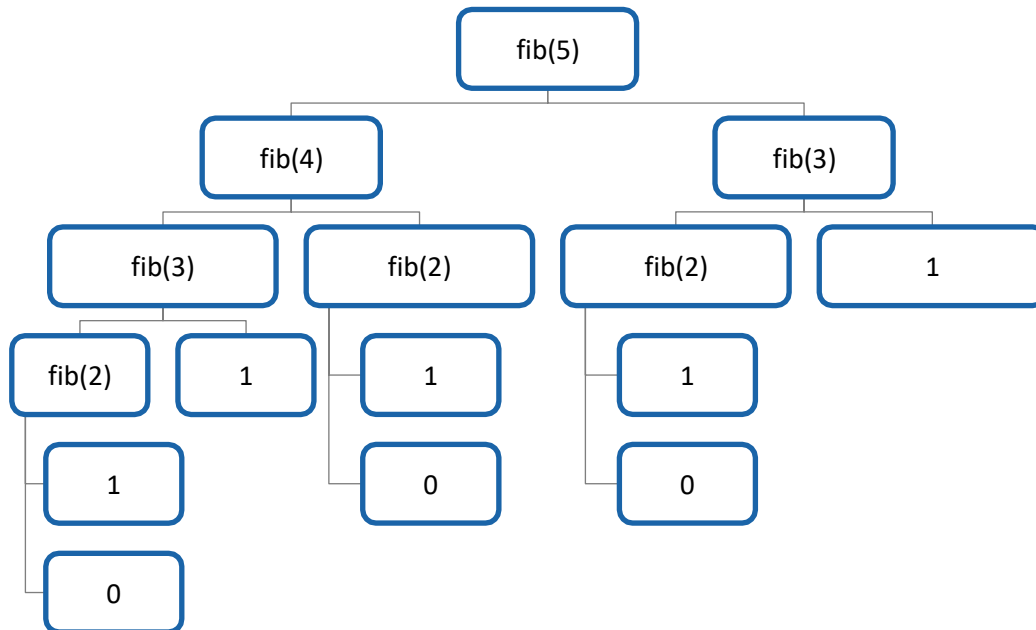
Primero verificamos si este número es menor o igual a 1.

En este caso no es así, por lo tanto pasamos al else, este dice que ahora retornaremos
fib(5 - 1) + fib(5 - 2)

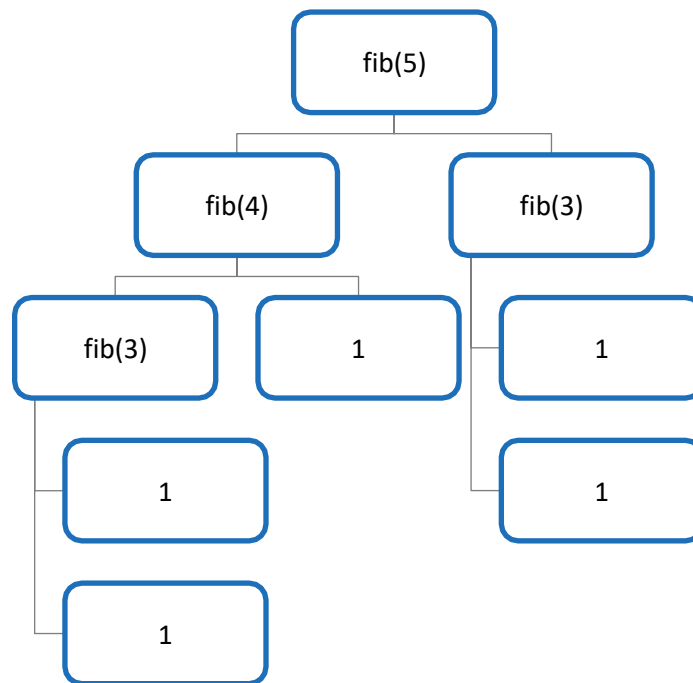
Ahora en conjuntamente se ejecuta el fib(4) y el fib(3) y dentro de cada uno se ejecuta fib(3) y fib(2) y del otro se ejecuta fib(2) y fib(1) y así avanza. Hasta que en algún momento uno llega a 1 o menor, en ese caso retorna el número y ahí queda a la espera.



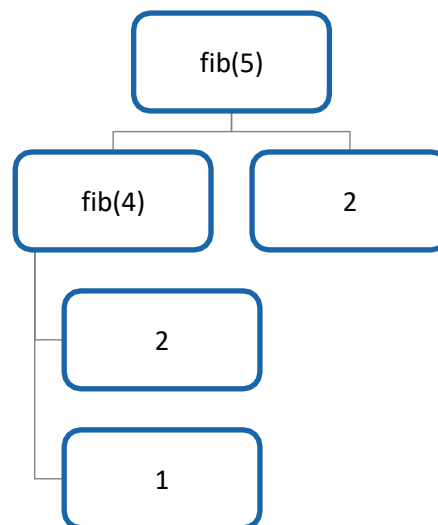
Así queda la secuencia, Después que llega al final se empieza a recoger o a devolverse.



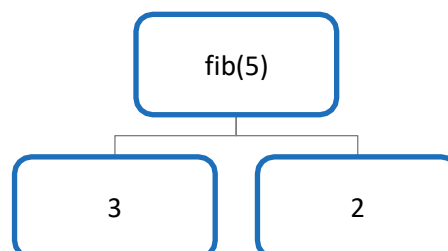
En la siguiente iteración queda:



En la siguiente iteración queda:



En la siguiente iteración queda:



Y al final queda:

5

De esta manera se ve cómo funciona una recursión.

Otro ejemplo:

Crear una función mediante recursividad que pueda retornar la factorial de un número que se le entrega como parámetro (argumento).

Factorial de 5 = $1*2*3*4*5$

Factorial de 0 es igual 1 no olviden eso y que también solo pueden ser números naturales positivos.

Este ejemplo recursivamente queda:

```
funciones.py > ...
1
2 def fac(num):
3     if num <= 1:
4         return 1
5     else:
6         return num * fac(num-1)
7
8 num = int(input("Ingresa un número: "))
9 resultado = fac(num)
10
11 print(resultado)
12
```

Entonces si ejecutamos fac(5) la respuesta será de 120.

```
User@DESKTOP-DVUM85S MINGW64 ~/Desktop/IPP
$ C:/Users/User/AppData/Local/Programs/Python/Python38-32/Python.exe
Ingresa un número: 5
120
```

3. Clases y métodos

3.1. Clases

Una clase en Python es una característica de programación orientada a objetos que define una estructura que tiene sus propias funciones y datos. Se crea en tiempo de ejecución y puede modificarse después de su creación.

Las clases se definen mediante el comando `class` y tienen la siguiente estructura:

```
funciones.py > ...  
1  class NombreClase:  
2      """#declaraciones de la clase"""  
3  
4
```

Las declaraciones de una clase pueden ser variables globales, métodos o funciones.

Creemos un ejemplo de una clase:

```
funciones.py > ...  
1  class mi_clase:  
2      a = "este es un texto"  
3      b = 12  
4  
5      def fun(self):  
6          return "hola ipp"  
7  
8  if __name__ == "__main__":  
9      variable_a = mi_clase()  
10     print(variable_a)  
11     print(variable_a.a)  
12     print(variable_a.b)  
13     p = variable_a.fun()  
14     print(p)  
15
```

La consola nos mostrara lo siguiente:

```
<__main__.mi_clase object at 0x000002E6161F4FD0>  
este es un texto  
12  
hola ipp
```


En las clases, existe un método llamado `__init__()`, el cual es el constructor de la clase y se utiliza para inicializar los atributos de la misma. La sintaxis para definir este método es la siguiente:

```
funciones.py > ...
1 class NombreClase:
2     def __init__(self, arg1, arg2):
3         self.atributo1 = arg1
4         self.atributo2 = arg2
5
```

Es importante destacar que el objeto `self` debe ser incluido en todos los métodos de la clase, incluyendo `__init__()`, ya que este hace referencia a la instancia actual de la clase.

Iniciemos un método de inicio de nuestra clase:

Ahora crearemos una clase que me tenga un objeto de un punto en el espacio en un plano de 3 Dimensiones.

```
funciones.py > ...
1 class punto:
2     def __init__(self, arg_1, arg_2, arg_3):
3         self.x = arg_1
4         self.y = arg_2
5         self.z = arg_3
6
7     def vect(self):
8         return [self.x, self.y, self.z]
9
10 if __name__ == "__main__":
11     variable_a = punto(10, 20, 3)
12     print(variable_a.x)
13     print(variable_a.y)
14     print(variable_a.z)
15     print(variable_a.vect())
16
```

Al ejecutarlo nos muestra lo siguiente:

```
10
20
3
[10, 20, 3]
```

Ahora creemos otro ejemplo algo más conocido, una mascota de tipo perro.

Entonces tenemos que pensar qué es lo que tiene todos los perros en común y qué es lo que puede variar.

Lo que tiene en común que todos son de caninos, y la cantidad de patas que tiene (pensado que todos los perros siempre nacerán y tendrán 4 patas). Ahora vemos lo que particular de cada perro, nombre, pelaje, tamaño, peso. Etc.

Entonces creemos la clase perro.

```
funciones.py > ...
1  class perro:
2      tipo = 'canino'
3      patas = 4
4
5      def __init__(self, nombre, pelaje, tamaño, peso):
6          self.nombre = nombre
7          self.pelaje = pelaje
8          self.tamaño = tamaño
9          self.peso = float(peso)
10
11     def alimentar(self):
12         self.peso += 1
13
14     if __name__ == "__main__":
15         a = perro('firulais', 'largo', 'medio', 4.6)
16         b = perro('max', 'corto', 'grande', 20)
17
18         print(a.nombre)
19         print(b.nombre)
20
21         b.alimentar()
22         print(b.peso)
23
```

Al ejecutarlo por consola nos muestra lo siguiente:

```
firulais
max
21.0
```

Es muy sencillo crear una clase, pero fíjate que en este ejemplo ocurrió algo curioso: los nombres de los argumentos entregados por el init eran iguales que los nombres de las variables de la clase. Entonces, ¿cómo identifica Python cuál es cuál?

Python puede identificar cuál es la variable de la clase y cuál es del argumento gracias a la palabra reservada "self". "self" es un objeto perteneciente a la clase, por lo que siempre que queramos modificar una información dentro de la clase, esta debe ser "self" y la variable de la clase correspondiente. También es una regla que todas las funciones de la clase tengan el objeto "self"

IMPORTANTE

Es posible crear un archivo .py que contenga solamente una clase y no tenga el bloque de código principal, y luego importar esa clase en otro archivo .py completamente distinto. Para hacer esto, se necesita utilizar la instrucción `import` seguida del nombre de la clase, sin incluir la extensión .py. Además, el archivo que contiene la clase debe estar ubicado en la misma carpeta que el archivo que realiza la importación.

3.2. Métodos especiales

Al igual que el método `init`, hay otros métodos especiales en Python que son muy utilizados:

- `str`: Este método se utiliza para convertir el objeto a una cadena de texto (string). Dentro de este método se define la forma en que se debe retornar el objeto como string.
- `add`: Este método define el comportamiento de la suma (+) entre dos objetos o clases.
- `sub`: Este método define el comportamiento de la resta (-) entre dos objetos o clases.
- `call`: Este método se utiliza cuando se llama a la variable de la clase con paréntesis.

Además de estos, existen muchos otros métodos especiales en Python que se pueden utilizar según el tipo de problema que se quiera resolver.

Los métodos especiales son una característica importante en la programación orientada a objetos en Python, ya que permiten definir el comportamiento de los objetos de una manera más personalizada y controlada.

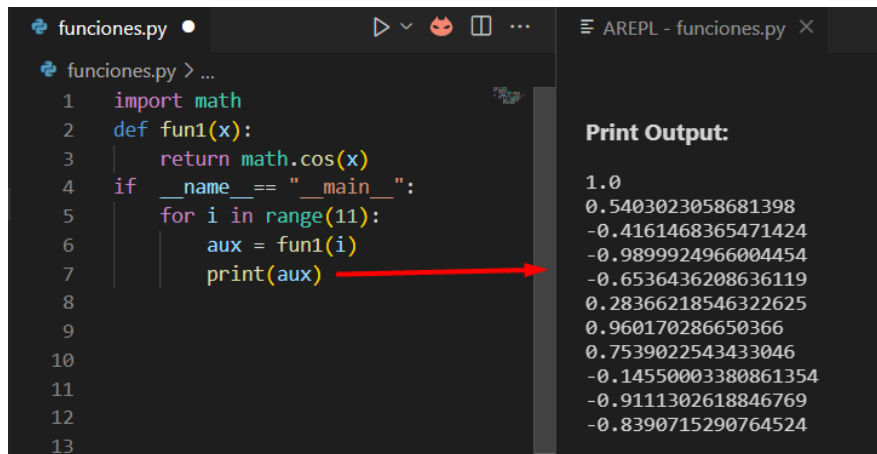
3.3. Librerías

La programación en Python cuenta con una amplia variedad de librerías disponibles, tanto incluidas en la distribución estándar de Python como aquellas creadas por terceros. Estas librerías se utilizan para llevar a cabo tareas específicas, como la manipulación de gráficos o el cálculo de funciones matemáticas. Las librerías son una herramienta muy útil en todo tipo de situaciones, y su uso puede ahorrar mucho tiempo y esfuerzo en el desarrollo de software.

Es importante importar correctamente las librerías al inicio de nuestro código Python, siguiendo las normas PEP-8. Si queremos conocer más detalles sobre una librería en particular, podemos consultar la documentación oficial de Python, que se encuentra en <https://docs.python.org/3/library/index.html>.

Uno de los ejemplos más comunes de librerías utilizadas en Python es la librería `math`, la cual es una librería matemática que contiene una gran cantidad de funciones trigonométricas, cálculos de variables, integrales, entre otras cosas. La documentación de esta librería se encuentra en <https://docs.python.org/3/library/math.html>.

Ejemplo: Crear una función que muestre de coseno desde 1 hasta 10.



```
funciones.py > ...
1  import math
2  def fun1(x):
3      return math.cos(x)
4  if __name__ == "__main__":
5      for i in range(11):
6          aux = fun1(i)
7          print(aux)
```

Print Output:

```
1.0
0.5403023058681398
-0.4161468365471424
-0.9899924966004454
-0.6536436208636119
0.28366218546322625
0.960170286650366
0.7539022543433046
-0.14550003380861354
-0.9111302618846769
-0.8390715290764524
```

Este código define una función llamada `fun1` que toma un argumento `x` y devuelve el coseno de `x`. Luego, el código usa un bucle `for` para iterar a través de los números del 0 al 10 y llama a la función `fun1` con cada número en la iteración. El valor devuelto por la función se almacena en la variable `aux` y se imprime en la pantalla. Por lo tanto, el código imprime el coseno de cada número del 0 al 10 en la pantalla.

Para utilizar la librería `math`, primero debemos importarla en nuestro código. Después podemos definir funciones que hagan uso de las funciones específicas de esta librería. Es importante recordar que, al utilizar una función de una librería, debemos incluir el nombre de la librería seguido de un punto antes del nombre de la función que deseamos utilizar. Este mismo principio se aplica al utilizar funciones de una clase definida en una librería.

Los resultados son los esperados, según la documentación los cálculos trigonométricos se encuentran en radianes.

Todas las librerías trabajan exactamente de la misma manera. Si existe una librería que encuentre documentación y no es estándar de Python, esta debe ser instalada. Para eso, existe una herramienta llamada `pip`, la cual permite instalar librerías de Python en nuestro computador. Este tema no es parte del curso, pero es importante conocerlo en caso de que se quiera utilizar una librería que no esté incluida en la instalación estándar de Python.

Por ejemplo, si quisieras instalar la librería `"numpy"`, podrías abrir la terminal o línea de comandos y escribir `"pip install numpy"`. Esto descargará e instalará la librería en tu computadora, lo que te permitirá utilizarla en tus programas de Python.

7. Cierre

Desde la aparición de la programación orientada a objetos (POO), se han abierto un mundo de posibilidades en la programación, permitiendo una reutilización de código de manera considerable. Antes de la POO, los códigos eran muy largos y era difícil encontrar errores o saber si se estaba trabajando con una función de manera innecesaria o si estaba mal ejecutada. Con la POO, los programadores pueden generar sus propias clases para cada tipo de necesidad y luego reutilizarlas en otros proyectos. Esto permite mejorar y agregar nuevas funciones constantemente.

Actualmente, en el año 2023, la POO se utiliza para crear frameworks, que son estructuras de trabajo que permiten generar sistemas web, aplicaciones de sistemas de comunicaciones, seguridad, entre otros. En este módulo hemos aprendido cómo crear una primera clase, lo que nos da un primer paso hacia el gran mundo de la programación orientada a objetos.