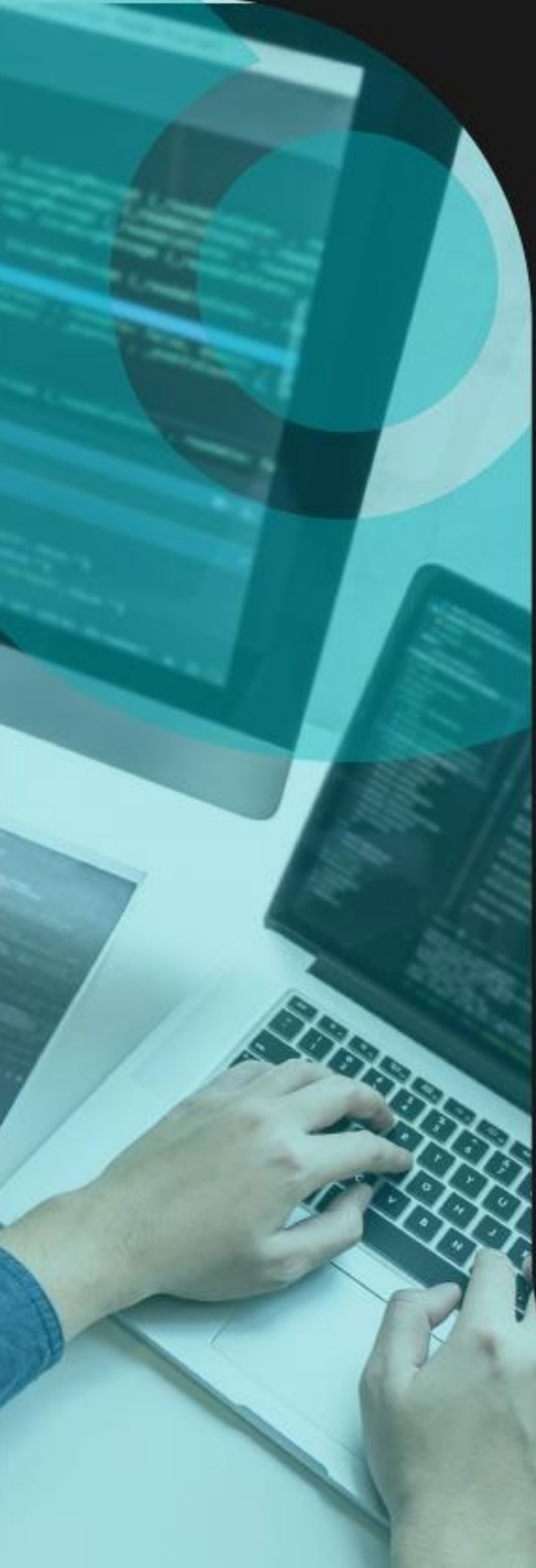

AWAKELAB

BASECAMP

Ciencia de Datos

Módulo : Fundamentos de Programación en Python

Aprendizaje Esperado

- Codificar un programa utilizando las instrucciones básicas, de control de flujo y funciones de acuerdo a la sintaxis del lenguaje Python para construir un algoritmo

Estructura y elementos del lenguaje Python.

Dentro de los lenguajes informáticos, Python pertenece al grupo de los lenguajes de programación y puede ser clasificado como un lenguaje interpretado, de alto nivel, multiplataforma, de tipo dinámico y multiparadigma. A diferencia de la mayoría de los lenguajes de programación, Python nos provee de reglas de estilos, a fin de poder escribir código fuente más legible y de manera estandarizada.

Como en la mayoría de los lenguajes de programación de alto nivel, Python se compone de una serie de elementos que alimentan su estructura. Entre ellos, podremos encontrar los siguientes:

Tipos de Datos

En Python podemos encontrar distintos tipos de datos con diferentes características y clasificaciones. A continuación veremos los tipos de datos básicos, aunque te introduciré otros tipos de datos que veremos más adelante.

Los tipos de datos básicos de Python son los *booleanos*, los *numéricos* (enteros, punto flotante y complejos) y las *cadenas de caracteres*.

Python también define otros tipos de datos, entre los que se encuentran:

- Secuencias: Los tipos list, tuple y range

- Mapas: El tipo dict
- Conjuntos: El tipo set
- Iteradores
- Clases
- Instancias
- Excepciones

A su vez, los tipos anteriores se pueden agrupar de diferente manera. Por ejemplo: el tipo cadena de caracteres es una secuencia inmutable; las listas, tuplas y diccionarios, entre otros, son contenedores y colecciones, etc.

Tipos numéricos

Python define tres tipos de datos numéricos básicos: *enteros*, *números de punto flotante* (simulará el conjunto de los números reales, pero ya veremos que no es así del todo) y los números *complejos.*

Números enteros

El tipo de los números enteros es **int**. Este tipo de dato comprende el conjunto de todos los números enteros, pero como dicho conjunto es infinito, en Python el conjunto está limitado realmente por la capacidad de la memoria disponible. No hay un límite de representación impuesto por el lenguaje.

Pero tranquilidad, que para el 99% de los programas que desarrolles tendrás suficiente con el subconjunto que puedes representar.

Un número de tipo int se crea a partir de un literal que representa un número entero o bien como resultado de una expresión o una llamada a una función.

Ejemplos:

```
>>> a = -1  # a es de tipo int y su valor es -1
>>> b = a + 2  # b es de tipo int y su valor es 1
>>> print(b)
1
```

También podemos representar los números enteros en formato *binario*, *octal o hexadecimal*.

Los números octales se crean anteponiendo 0o a una secuencia de dígitos octales (del 0 al 7).

Para crear un número entero en hexadecimal, hay que anteponer 0x a una secuencia de dígitos en hexadecimal (del 0 al 9 y de la A la F).

En cuanto a los números en binario, se antepone 0b a una secuencia de dígitos en binario (0 y 1).

```
>>> diez = 10
>>> diez_binario = 0b1010
>>> diez_octal = 0o12
>>> diez_hex = 0xa
>>> print(diez)
10
>>> print(diez_binario)
10
>>> print(diez_octal)
10
>>> print(diez_hex)
```

```
10
```

Números de punto flotante

Puedes usar el tipo float sin problemas para representar cualquier número real (siempre teniendo en cuenta que es una aproximación lo más precisa posible). Por tanto para longitudes, pesos, frecuencias, …, en los que prácticamente es lo mismo 3,3 que 3,3000000000000003 el tipo float es el más apropiado.

Cuando un número float vaya a ser usado por una persona, en lugar de por el computador, puedes darle formato al número de la siguiente manera:

```
>>> real = 1.1 + 2.2  # real es un float
>>> print(real)
3.3000000000000003  # Representación aproximada de 3.3
>>> print(f'{real:.2f}')
3.30  # real mostrando únicamente 2 cifras decimales
```

Al igual que los números enteros, un float se crea a partir de un literal, o bien como resultado de una expresión o una función.

```
>>> un_real = 1.1  # El literal debe incluir el carácter
>>> otro_real = 1/2  # El resultado de 1/2 es un float
```

```
>>> not_cient = 1.23E3  # float con notación
científica (1230.0)
```

```
 A = [1 2 3
      4 5 6]



      [-1 0
B =    0 1
       1 1]
```

Si llegamos a necesitar una mayor precisión a la hora de trabajar con los números reales, Python tiene otros tipos de datos, como *Decimal*.

El tipo *Decimal* es ideal a la hora de trabajar, por ejemplo, con dinero o tipos de interés. Este tipo de dato trunca la parte decimal del número para ser más preciso.

Números complejos

El último tipo de dato numérico básico que tiene Python es el de los números complejos, complex.

Los números complejos tienen una parte real y otra imaginaria y cada una de ellas se representa como un float.

Para crear un número complejo, se sigue la siguiente estructura:

$a + bi$, donde a y b, son números reales y la i es el número imaginario

Y se puede acceder a la parte real e imaginaria a través de los atributos **real** e **imag**:

```
>>> complejo = 1+2j
>>> complejo.real
1.0
>>> complejo.imag
2.0
```

Aritmética de los tipos numéricos.

Con todos los tipos numéricos se pueden aplicar las operaciones de la aritmética: suma, resta, producto, división.

En Python está permitido realizar una operación aritmética con números de distinto tipo. En este caso, el tipo numérico «más pequeño» se convierte al del tipo «más grande», de manera que el tipo del resultado siempre es el del tipo mayor. Entendemos que el tipo int es menor que el tipo float que a su vez es menor que el tipo **complex**.

Por tanto, es posible, por ejemplo, sumar un **int** y un **float**:

```
>>> 1 + 2.0
3.0
>>> 2+3j + 5.7
```

```
(7.7+3j)
```

Tipos booleano

En Python la clase que representa los valores booleanos es **bool**. Esta clase solo se puede instanciar con dos valores/objetos: True para representar verdadero y False para representar falso.

Una particularidad del lenguaje es que cualquier objeto puede ser usado en un contexto donde se requiera comprobar si algo es verdadero o falso. Por tanto, cualquier objeto se puede usar en la condición de un if o un while (son estructuras de control que veremos en tutoriales posteriores) o como operando de una operación booleana.

Por defecto, cualquier objeto es considerado como verdadero con dos excepciones:

- Que implemente el método **__bool__()** y este devuelva **False**.
- Que implemente el método **__len__()** y este devuelva **0**.

Además, los siguientes objetos/instancias también son consideradas falsas:

- **None**
- **False**
- El valor cero de cualquier tipo numérico: **0, 0.0, 0j, …**
- Secuencias y colecciones vacías (veremos estos tipos en otros tutoriales): **'', (), [], {}, set(), range(0)**

Tipos de cadena de caracteres

Otro tipo básico de Python, e imprescindible, son las secuencias o cadenas de caracteres. Este tipo es conocido como *string* aunque su clase verdadera es **str**.

Formalmente, un *string* es una secuencia **inmutable** de caracteres en formato Unicode.

Para crear un string, simplemente tienes que encerrar entre comillas simples '' o dobles "" una secuencia de caracteres.

Se puede usar indistintamente comillas simples o dobles, con una particularidad. Si en la cadena de caracteres se necesita usar una comilla simple, tienes dos opciones: usar comillas dobles para encerrar el string, o bien, usar comillas simples pero anteponer el carácter \ a la comilla simple del interior de la cadena. El caso contrario es similar.

Veamos todo esto con un ejemplo:

```
>>> hola = 'Hola "Pythonista"'
>>> hola_2 = 'Hola \'Pythonista\''
>>> hola_3 = "Hola 'Pythonista'"
>>> print(hola)
Hola "Pythonista"
>>> print(hola_2)
Hola 'Pythonista'
>>> print(hola_3)
Hola 'Pythonista'
```

A diferencia de otros lenguajes, en Python no existe el tipo «**carácter**». No obstante, se puede simular con un string de un solo carácter:

```
>>> caracter_a = 'a'
>>> print(caracter_a)
```

```
a
```

Otros tipos

Hasta aquí hemos repasado los tipos de datos básicos de Python, sin embargo, el lenguaje ofrece muchos tipos más.

Además de los tipos básicos, otros tipos fundamentales de Python son las secuencias (**list** y **tuple**), los conjuntos (**set**) y los mapas (**dict**).

Todos ellos son tipos compuestos y se utilizan para agrupar juntos varios valores.

- Las listas son secuencias mutables de valores.
- Las tuplas son secuencias inmutables de valores.
- Los conjuntos se utilizan para representar conjuntos únicos de elementos, es decir, en un conjunto no pueden existir dos objetos iguales.
- Los diccionarios son tipos especiales de contenedores en los que se puede acceder a sus elementos a partir de una clave única.

```
>>> lista = [1, 2, 3, 8, 9]
>>> tupla = (1, 4, 8, 0, 5)
>>> conjunto = set([1, 3, 1, 4])
>>> diccionario = {'a': 1, 'b': 3, 'z': 8}
>>> print(lista)
[1, 2, 3, 8, 9]
>>> print(tupla)
```

```
(1, 4, 8, 0, 5)
>>> print(conjunto)
{1, 3, 4}
>>> print(diccionario)
{'a': 1, 'b': 3, 'z': 8}
```

Operadores y expresiones

Los **operadores** son símbolos especiales que representan cómputos como la suma y la multiplicación. Los valores que el operador usa se denominan **operandos**.

Operadores Aritméticos

Un operador aritmético toma dos operandos como entrada, realiza un cálculo y devuelve el resultado.

Considera la expresión, **"a = 2 + 3"**. Aquí, 2 y 3 son los *operandos* y **+** es el *operador aritmético*. El resultado de la operación se almacena en la variable **a**.

OPERADOR	DESCRIPCIÓN	USO
+	Realiza Adición entre los operandos	12 + 3 = 15
-	Realiza Sustracción entre los operandos	12 - 3 = 9
*	Realiza Multiplicación entre los operandos	12 * 3 = 36
/	Realiza División entre los operandos	12 / 3 = 4

%	Realiza un módulo entre los operandos	16 % 3 = 1
**	Realiza la potencia de los operandos	12 ** 3 = 1728
//	Realiza la división con resultado de número entero	18 // 5 = 3

Para obtener el resultado en tipo flotante, uno de los operadores también debe ser de tipo flotante.

Ejemplo:

```
>>> x = 7
>>> y = 2
>>> x + y  # Suma
9
>>> x - y  # Resta
5
>>> x * y  # Producto
14
>>> x / y  # División
3.5
>>> x % y  # Resto
1
>>> x // y  # Cociente
3
```

```
>>> x ** y  # Potencia
49
```

Operadores Relacionales

Un operador relacional se emplea para comparar y establecer la relación entre ellos. Devuelve un valor booleano (true o false) basado en la condición.

OPERADOR	DESCRIPCIÓN	USO
>	Devuelve True si el operador de la izquierda es mayor que el operador de la derecha	12 > 3 devuelve True
<	Devuelve True si el operador de la derecha es mayor que el operador de la izquierda	12 < 3 devuelve False
==	Devuelve True si ambos operandos son iguales	12 == 3 devuelve False
>=	Devuelve True si el operador de la izquierda es mayor o igual que el operador de la derecha	12 >= 3 devuelve True
<=	Devuelve True si el operador de la derecha es mayor o igual que el operador de la izquierda	12 <= 3 devuelve False

!=	Devuelve True si ambos operandos no son iguales	12 != 3 devuelve True

Operadores de Asignación

Se utiliza un operador de asignación para asignar valores a una variable. Esto generalmente se combina con otros operadores (como aritmética, bit a bit) donde la operación se realiza en los operadores y el resultado se asigna al operando izquierdo.

Considera los siguientes ejemplos,

a = 18. Aquí = es un operador de asignación, y el resultado se almacena en la variable a.

a + = 10. Aquí += es un operador de asignación, y el resultado se almacena en la variable a. Es lo mismo que a = a + 10.

OPERADOR	DESCRIPCIÓN
=	a = 5. El valor 5 es asignado a la variable a
+=	a += 5 es equivalente a a = a + 5
-=	a -= 5 es equivalente a a = a - 5
*=	a *= 3 es equivalente a a = a * 3
/=	a /= 3 es equivalente a a = a / 3

%=	a %= 3 es equivalente a a = a % 3
**=	a **= 3 es equivalente a a = a ** 3
//=	a //= 3 es equivalente a a = a // 3
&=	a &= 3 es equivalente a a = a & 3
|=	a |= 3 es equivalente a a = a | 3
^=	a ^= 3 es equivalente a a = a ^ 3
>>=	a >>= 3 es equivalente a a = a >> 3
<<=	a <<= 3 es equivalente a a = a << 3

Operadores Lógicos

Se utiliza un operador lógico para tomar una decisión basada en múltiples condiciones. Los operadores lógicos utilizados en Python son **and**, **or** y **not**.

OPERADOR	DESCRIPCIÓN	USO
and	Devuelve True si ambos operandos son True	a and b
or	Devuelve True si alguno de los operandos es True	a or b

not	Devuelve True si alguno de los operandos False	not a

Operadores de Pertenencia

Un operador de pertenencia se emplea para identificar pertenencia en alguna secuencia (listas, strings, tuplas).

in y **not** in son operadores de pertenencia.

in devuelve

not in devuelve True si el valor especificado no se encuentra en la secuencia. En caso contrario devuelve False.

OPERADOR	DESCRIPCIÓN
in	Devuelve True si el valor especificado se encuentra en la secuencia. En caso contrario devuelve False.
Not in	Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario

Ejemplo:

```
>>> lista = [1, 3, 2, 7, 9, 8, 6]
>>> 4 in lista
False
>>> 3 in lista
```

```
True
>>> 4 not in lista
True
```

Operadores de Identidad

Un operador de identidad se emplea para comprobar si dos variables emplean la misma ubicación en memoria.

is y is not son operadores de identidad.

OPERADOR	DESCRIPCIÓN
Is	Devuelve True si ambos operandos hacen referencia al mismo objeto; False en caso contrario.
Is not	Devuelve True si ambos operandos no hacen referencia al mismo objeto; False en caso contrario.

Ejemplo:

```
>>> x = 4
>>> y = 2
>>> lista = [1, 5]
>>> x is lista
False
>>> x is y
```

```
False
>>> x is 4
True
```

Expresiones

Una expresión es una combinación de valores y operaciones que, al ser evaluados, entregan un valor.

Algunos elementos que pueden formar parte de una expresión son: valores literales (como 2, "hola" o 5.7), variables, operadores y llamadas a funciones.

Por ejemplo, la expresión 4 * 3 - 2 entrega el valor 10 al ser evaluada por el intérprete:

```
>>> 4 * 3 - 2
10
```

El valor de la siguiente expresión depende del valor que tiene la variable n en el momento de la evaluación:

```
>>> n / 7 + 5
```

Una expresión está compuesta de otras expresiones, que son evaluadas recursivamente hasta llegar a sus componentes más simples, que son los literales y las variables.

Variables

Una de las características más poderosas en un lenguaje de programación es la capacidad de manipular variables. Una variable es un nombre que se refiere a un valor.

La sentencia de asignación crea nuevas variables y les da valores:

```
>>> mensaje = "¿Qué Onda?"
>>> n = 17
>>> pi = 3.14159
```

Este ejemplo hace tres asignaciones. La primera asigna la cadena "¿Que Onda?" a una nueva variable denominada mensaje. La segunda le asigna el entero 17 a **n**, y la tercera le da el número de punto flotante 3.14159 a **pi**.

No debe confundirse el operador de asignación, **=**, con el signo de igualdad (aún y cuando se usa el mismo símbolo). El operador de asignación enlaza un nombre, en el lado izquierdo del operador, con un valor en el lado derecho. Esta es la razón por la que obtendrá un error si escribe:

```
>>> 17 = n
```

Al leer o escribir código, dígase a sí mismo "**n se le asigna 17**" o "n obtiene el valor de 17". No diga "n es igual a 17"

Una manera común de representar variables en el papel es escribir el nombre de la variable con una flecha apuntando a su valor. Esta clase de dibujo se denomina **diagrama de estado** porque muestra el estado de cada una de las variables (piense en los valores como el estado mental de las variables). Este diagrama muestra el resultado de las sentencias de asignación anteriores:

message→ "What's up, Doc?"

n → 17

pi → 3.1415

Si le preguntas a el intérprete para evaluar una variable, se producirá el valor que está vinculado a la variable:

```
>> print mensaje
'What's up, Doc?'
>>> print n
17
>>> print pi
3.14159
```

En cada caso el resultado es el valor de la variable. Las variables también tienen tipos; nuevamente, le podemos preguntar al intérprete cuáles son.

```
>>> type(mesaje)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

El tipo de una variable es el tipo del valor al que se refiere.

Usamos las variables en un programa para “recordar” las cosas, al igual que la puntuación actual en el partido de fútbol. Sin embargo, las variables son variables. Esto significa que pueden cambiar con el tiempo, al igual que el marcador de un partido de fútbol. Se puede asignar un valor a una variable, y luego asignar un valor diferente a la misma variable.

```
>>> dia = "Jueves"
>>> dia
'Jueves'
>>> dia = "Viernes"
>>> dia
'Viernes'
>>> dia = 21
>>> dia
21
```

Usted notará que se ha combinado el valor del ”dia” en tres ocasiones, y en la tercera asignación nosotros le dimos un valor que era de un tipo diferente.

Una gran parte de la programación se trata de que la computadora pueda recordar las cosas, por ejemplo, el número de llamadas perdidas es su teléfono, y la organización de alguna actualización o cambio de la variable cuando se olvida otra llamada.

Nombres de variables válidos en Python debe ajustarse a las siguientes tres simples reglas:

1. Son secuencias arbitrariamente largas de letras y dígitos.
2. La secuencia debe empezar con una letra.
3. Además de a..z, y A..Z, el guión bajo (_) es una letra.

Los programadores generalmente escogen nombres significativos para sus variables — que especifique para qué se usa la variable.

Aunque es permitido usar letras mayúsculas, por convención no lo hacemos. Si usted lo hace, recuerde que las letras mayúsculas importan, **Pedro** y **pedro** son variables diferentes.

El carácter subrayado (_) puede aparecer en un nombre. A menudo se usa en nombres con múltiples palabras, tales como **mi_nombre**́ ó **precio_de_la_porcelana_en_china**.

Hay algunas situaciones en las que los nombres que comienzan con un guión tienen un significado especial, por lo que una regla segura para los principiantes es empezar todos los nombres con una letra que no sea un guión.

Si usted le da un nombre inválido a una variable obtendrá un error de sintaxis:

```
>>> 76trombones = "gran desfile"
SyntaxError: invalid syntax
>>> mas$ = 1000000
SyntaxError: invalid syntax
>>> class = "Informática 101"
SyntaxError: invalid syntax
```

76trombones es inválido porque no empieza con una letra. **mas$** es inválido porque contiene un carácter ilegal, por el símbolo de dolar **$**. Pero, ¿Qué sucede con **class**?

Resulta que class es una de las palabras claves de Python. Las palabras claves definen las reglas del lenguaje y su estructura, y no pueden ser usadas como nombres de variables.

Python tiene treinta y tantas palabras clave (y todas son mejoradas de vez en cuando para que Python introduzca o elimine una o dos):

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

Usted puede mantener esta lista a mano. Si el intérprete se queja por alguno de sus nombres de variables, y usted no sabe por qué, búsquelo en esta lista.

Los programadores generalmente eligen nombres para sus variables que son significativas para los lectores humanos del programa — que ayudan a los documentos de programación, o recordar, lo que la variable se utiliza.

Conversiones de tipos

Hacer un ***cast*** o ***casting*** significa convertir un tipo de dato a otro. Anteriormente hemos visto tipos como los int, string o float. Pues bien, es posible convertir de un tipo a otro.

Pero antes de nada, veamos los diferentes tipos de ***cast*** o conversión de tipos que se pueden hacer. Existen dos:

- Conversión **implícita**: Es realizada automáticamente por Python. Sucede cuando realizamos ciertas operaciones con dos tipos distintos.
- Conversión **explícita**: Es realizada expresamente por nosotros, como por ejemplo convertir de str a int con str().

Conversión implícita

Esta conversión de tipos es realizada automáticamente por Python, prácticamente sin que nos demos cuenta. Aún así, es importante saber lo que pasa por debajo para evitar problemas futuros.

El ejemplo más sencillo donde podemos ver este comportamiento es el siguiente:

```
a = 1   # <class 'int'>
b = 2.3 # <class 'float'>


a = a + b
print(a)       # 3.3
print(type(a)) # <class 'float'>
```

Pero si sumamos a y b y almacenamos el resultado en a, podemos ver como internamente Python ha convertido el int en float para poder realizar la operación, y la variable resultante es float

Sin embargo hay otros casos donde Python no es tan listo y no es capaz de realizar la conversión. Si intentamos sumar un int a un string, tendremos un error **TypeError**.

```
a = 1
b = "2.3"
c = a + b
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Si te das cuenta, es lógico que esto sea así, ya que en este caso b era "2.3", pero ¿y si fuera "Hola"? ¿Cómo se podría sumar eso? No tiene sentido.

Conversión explícita

Por otro lado, podemos hacer conversiones entre tipos o *cast* de manera explícita haciendo uso de diferentes funciones que nos proporciona Python. Las más usadas son las siguientes:

- float(), str(), int(), list(), set()
- Y algunas otras como hex(), oct() y bin()

Convertir float a int

Para convertir de float a int debemos usar float(). Pero mucho cuidado, ya que el tipo entero no puede almacenar decimales, por lo que perderemos lo que haya después de la coma.

```
a = 3.5

a = int(a)

print(a)

# Salida: 3
```

Convertir float a string

Podemos convertir un float a string con str(). Podemos ver en el siguiente código como cambia el tipo de a después de hacer el *cast*.

```
a = 3.5

print(type(a)) # <class 'float'>

a = str(a)

print(type(a)) # <class 'str'>
```

Convertir string a float

Podemos convertir un string a float usando float(). Es importante notar que se usa el . como separador.

```
a = "35.5"

print(float(a))
```

```
# Salida: 35.5
```

El siguiente código daría un error, dado que , no se reconoce por defecto como separador decimal.

```
a = "35,5"

print(float(a))

# Salida: ValueError: could not convert string to
float: '35,5'
```

Y por último, resulta obvio pensar que el siguiente código dará un error también.

```
a = "Python"

print(float(a))

# Salida: ValueError: could not convert string to
float: 'Python'
```

Convertir string a int

Al igual que la conversión a float del caso anterior, podemos convertir de string a int usando int().

```
a                          =                          3
print(type(a))          #          <class          'str'>
a                           =                         int(a)
print(type(a)) # <class 'int'>
```

Cuidado ya que no es posible convertir a int cualquier valor.

```
a = "Python"

a = int(a)

# ValueError: invalid literal for int() with base 10:
'Python'
```

Convertir int a string

La conversión de int a string se puede realizar con str().

A diferencia de otras conversiones, esta puede hacerse siempre, ya que cualquier valor entero que se nos ocurra poner en a, podrá ser convertido a string.

```
a = 10

print(type(a)) # <class 'int'>

a = str(a)

print(type(a)) # <class 'str'>
```

Convertir a list

Es también posible hacer un *cast* a lista, desde por ejemplo un set. Para ello podemos usar list().

```
a = {1, 2, 3}

b = list(a)


print(type(a)) # <class 'set'>

print(type(b)) # <class 'list'>
```

Convertir a set

Y de manera completamente análoga, podemos convertir de lista a set haciendo uso de set().

```
a = ["Python", "Mola"]

b = set(a)


print(type(a)) # <class 'list'>

print(type(b)) # <class 'set'>
```

Entrada y Salida de Datos

Una de las características más importantes de cualquier lenguaje de programación es poder interactuar con el usuario.

Entrada Estándar

Para pedir información al usuario, debe utilizar las funciones integradas en el intérprete del lenguaje, así como los argumentos de línea de comandos.

Ejemplo de la función **raw_input**:

La función **raw_input()** siempre devuelve un valor de cadenas de caracteres:

```
>>> nombre = raw_input('Ana: ¿Cómo se llama usted?:
')

Ana: ¿Cómo se llama usted?: Leonardo

>>> print nombre
```

```
Leonardo
```

Ejemplo de la función **input**:

La función **input()** siempre devuelve un valor numérico:

```
>>> edad = input('Ana: ¿Que edad tiene usted?: ')
Ana: ¿Que edad tiene usted?: 38
>>> print edad
38
```

Entrada por Script

Se basan en escribir todas las instrucciones en archivos llamados scripts, que no es más que guiones de instrucciones. Luego se envía este archivo al intérprete como parámetro desde la terminal de comando (si es un lenguaje interpretado como Python) y éste ejecutará todas las instrucciones en bloque.

A parte de ser la base del funcionamiento de los programas, la característica de los scripts es que pueden recibir datos desde la propia terminal de comando en el momento de la ejecución, algo muy útil para agregar dinamismo los scripts a través de parámetros personalizables.

A continuación, un ejemplo el cual simula a sala de chat del servicio LatinChat.com, validando datos de entradas numérico y tipo cadena de caracteres e interactuá con el usuario y en base a condicionales muestra un mensaje.

```
print "\nSimulando a LatinChat"
print "====================="
```

```
print "\nSala de Chat > De 30 a 40 años"

print "------------------------------\n"


print 'Ana: ¿Cómo se llama usted?: '

nombre = raw_input('Yo: ')

print 'Ana: Hola', nombre, ', encantada de conocerte
:3'


print 'Ana: ¿Que edad tiene usted?: '

edad = input('Yo: ')

print 'Usted tiene', edad, ', y yo ya no digo mi edad
xD'


print 'Ana: ¿Tiene Webcams?, ingrese "si" o "no", por
favor!: '

tiene_WebCam = raw_input('Yo: ')


if tiene_WebCam in ('s', 'S', 'si', 'Si', 'SI'):

      print "Ponga la WebCam para verle :-D"

elif tiene_WebCam in ('n', 'no', 'No', 'NO'):

      print "Lastima por usted :'( Adiós"
```

Script con argumentos

Para poder enviar información a un script y manejarla, tenemos que utilizar la librería de sistema sys. En ella encontraremos la lista argv que almacena los argumentos enviados al script.

Usted debe crear un script llamado entrada_argumentos.py con el siguiente contenido:

```
import sys

print sys.argv
```

Ejecuta el *script* llamado entrada_argumentos.py, de la siguiente forma:

```
python entrada_argumentos.py

['entrada_argumentos.py']
```

Al ejecutarlo puede ver que devuelve una lista con una cadena que contiene el nombre del script. Entonces, el primer argumento de la lista sys.argv (es decir, sys.argv[0]) es el propio nombre del script.

Ahora si intenta ejecutar el script de nuevo pasando algunos valores como números y cadenas de caracteres entre comillas dobles, todo separado por espacios:

```
python entrada_argumentos.py 300 43.234 "Hola Plone"

['entrada_argumentos.py', '300', '43.234', 'Hola
Plone']
```

Cada valor que enviamos al script durante la llamada se llama argumento e implica una forma de entrada de datos alternativa sin usar las funciones input() y raw_input().

A continuación, un ejemplo el cual usa un script con la librería sys. El script recibe dos (02) argumentos: una cadena de caracteres y un número entero. Lo que hace es imprimir la cadena de caracteres tantas veces como le indique con el argumento de tipo número:

```
import sys


# Comprobación de seguridad, ejecutar sólo si se
reciben 2

# argumentos realmente

if len(sys.argv) == 3:

    texto = sys.argv[1]

    repeticiones = int(sys.argv[2])

    for r in range(repeticiones):

        print texto

else:

    print "ERROR: Introdujo uno (1) o más de dos (2)
argumentos"

    print "SOLUCIÓN: Introduce los argumentos
correctamente"

    print 'Ejemplo: entrada_dos_argumentos.py "Texto"
5'
```

Si quiere comprobar la validación de cuantos argumentos deben enviarse al script, ejecute el siguiente comando:

```
python entrada_dos_argumentos.py "Hola Plone"
ERROR: Introdujo uno (1) o más de dos (2) argumentos
SOLUCIÓN: Introduce los argumentos correctamente
Ejemplo: entrada_dos_argumentos.py "Texto" 5
```

Ahora si intenta ejecutar el *script* entrada_dos_argumentos.py con solo dos (2) argumentos, ejecutando el siguiente comando:

```
python entrada_dos_argumentos.py "Hola Plone" 3
Hola Plone
Hola Plone
Hola Plone
```

Salida Estándar

La forma general de mostrar información por pantalla es mediante una consola de comando, generalmente podemos mostrar texto y variables separándolos con comas, para este se usa la sentencia print.

Sentencia print

Sentencia print evalúa cada expresión, devuelve y escribe el objeto resultado a la salida estándar de la consola de comando. Si un objeto no es un tipo cadena de caracteres, ese es primeramente convertido a un tipo cadena de caracteres usando las reglas para las conversiones del tipo. La cadena de caracteres (resultado u original) es entonces escrito.

Entonces para mostrar mensajes en pantalla, se utiliza el uso de la sentencia print.

Ejemplo del uso de la sentencia print:

```
>>> print 'Ana: Hola', nombre, ', encantada de
conocerte :3'

Ana: Hola Leonardo , encantado de conocerte :3
```

Formato de impresión de cadena

En la sentencia print se pueden usar el formato de impresión alternando las cadenas de caracteres y variables:

```
>>> tipo_calculo = "raíz cuadrada de dos"

>>> valor = 2**0.5

>>> print "el resultado de", tipo_calculo, "es:",
valor

el resultado de raíz cuadrada de dos es:
1.41421356237
```

Control de Flujo

Un programa o *script* de Python es un conjunto de instrucciones analizadas y ejecutadas por el intérprete de arriba hacia abajo y de izquierda a derecha. Cuando todas las instrucciones se han ejecutado, el programa termina. No obstante, contamos con herramientas para alterar el flujo natural del programa: hacer que se saltee una porción de código según se cumpla tal o cual condición, repetir un conjunto de instrucciones, etc.

Condicional

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. La evaluación de condiciones, solo puede arrojar 1 de 2 resultados: verdadero o falso (True o False).

En la vida diaria, actuamos de acuerdo a la evaluación de condiciones, de manera mucho más frecuente de lo que en realidad creemos: Si el semáforo está en verde, cruzar la calle. Sino, esperar a que el semáforo

se ponga en verde. A veces, también evaluamos más de una condición para ejecutar una determinada acción: Si llega la factura de la luz y tengo dinero, pagaré la boleta.

Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación).

Y para evaluar más de una condición simultáneamente, se utilizan **operadores lógicos**.

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje**: if (si), elif (sino, si) y else (sino).**

If – elif - else

Ejemplo: Si el semáforo está en verde, cruzar la calle. Sino, esperar.

```
if semaforo == verde:
      print "Cruzar la calle"
else:
      print "Esperar"
```

Si gasto hasta $100, pago con dinero en efectivo. Sino, si gasto más de $100 pero menos de $300, pago con tarjeta de débito. Sino, pago con tarjeta de crédito.

```
if compra <= 100:
      print "Pago en efectivo"
elif compra > 100 and compra < 300:
      print "Pago con tarjeta de débito"
else:
```

```
      print "Pago con tarjeta de crédito"
```

Si la compra es mayor a $100, obtengo un descuento del 10%

```
importe_a_pagar = total_compra

if total_compra > 100:

tasa_descuento = 10 importe_descuento = total_compra
*       tasa_descuento / 100 importe_a_pagar =
total_compra -  importe_descuento
```

Iterativas

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición. En Python se dispone de dos estructuras cíclicas:

- El bucle while
- El bucle for

Bucle While

Este bucle, se encarga de ejecutar una misma acción “mientras que” una determinada condición se cumpla:

Mientras que año sea menor o igual a 2012, imprimir la frase “Informes del Año año”

```
anio = 2001

while anio <= 2012:

print "Informes del Año", str(anio)
```

```
anio += 1
```

La iteración anterior, generará la siguiente salida:

```
Informes del Año 2001
Informes del Año 2002
Informes del Año 2003
Informes del Año 2004
Informes del Año 2005
Informes del Año 2006
Informes del Año 2007
Informes del Año 2009
Informes del Año 2010
Informes del Año 2011
Informes del Año 2012
```

Si miras la última línea**: anio += 1**

Podrás notar que en cada iteración, incrementamos el valor de la variable que condiciona el bucle (anio). Si no lo hiciéramos, esta variable siempre sería igual a 2001 y el bucle se ejecutará de forma infinita, ya que la condición (anio <= 2012) siempre se estaría cumpliendo.

Pero ¿Qué sucede si el valor que condiciona la iteración no es numérico y no puede incrementarse? En ese caso, podremos utilizar una estructura de control condicional, anidada dentro del bucle, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada break:

```
while True:

nombre = raw_input("Indique su nombre: ")

if nombre: break
```

El bucle anterior, incluye un condicional anidado que verifica si la variable nombre es verdadera (sólo será verdadera si el usuario tipea un texto en pantalla cuando el nombre le es solicitado). Si es verdadera, el bucle para (break). Sino, seguirá ejecutándose hasta que el usuario ingrese un texto en pantalla.

Bucle For

El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

Por cada nombre en **mi_lista**, imprimir nombre

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']

for nombre in

mi_lista: print nombre
```

Por cada color en **mi_tupla**, imprimir color

```
mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')

for color in mi_tupla:

print color
```

En los ejemplos anteriores, nombre y color, son dos variables declaradas en tiempo de ejecución (es decir, se declaran dinámicamente durante el bucle), asumiendo como valor, el de cada elemento de la lista (o tupla) en cada iteración.

Otra forma de iterar con el bucle for, puede emular a while:

Por cada año en el rango 2001 a 2013, imprimir la frase “Informes del Año año”

```
for anio in range(2001, 2013):

print "Informes del Año", str(anio)
```

Funciones

Las funciones en Python, y en cualquier lenguaje de programación, son estructuras esenciales de código. Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto.

- Dividir y organizar el código en partes más sencillas.
- Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python ya define de serie un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones. Por ejemplo, la función **len()**, que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario o un conjunto. También hemos visto la función **print()**, que muestra por consola un texto.

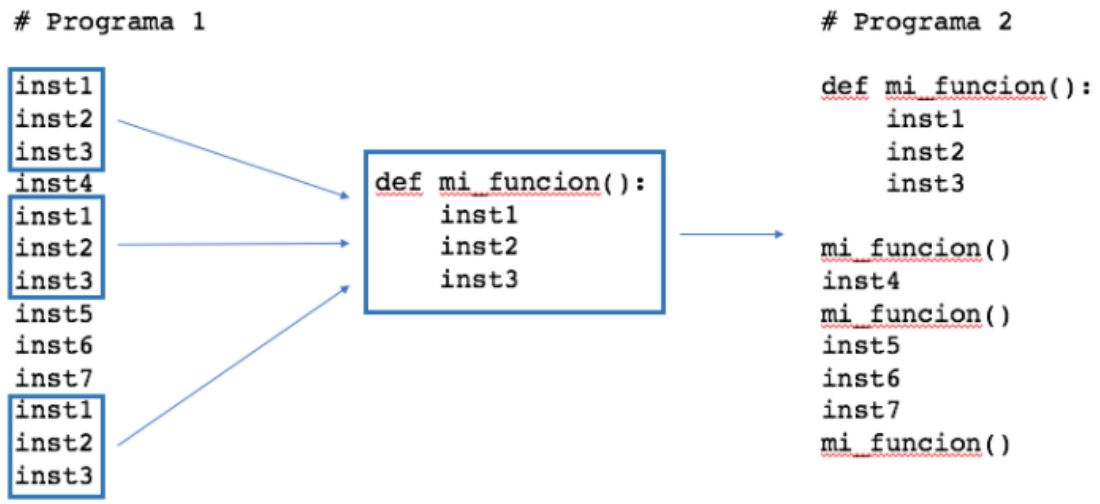

imagen: fuente propia

En principio, un programa es una secuencia ordenada de instrucciones que se ejecutan una a continuación de la otra. Sin embargo, cuando se utilizan funciones, puedes agrupar parte de esas instrucciones como una unidad más pequeña que ejecuta dichas instrucciones y suele devolver un resultado.

Definición de una función en python.

La siguiente imagen muestra el esquema de una función en Python:

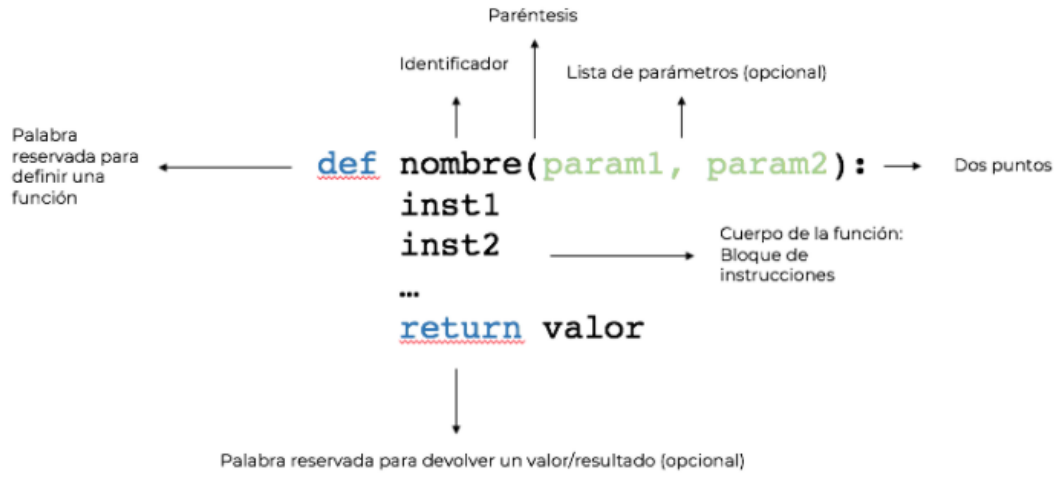

imagen: fuente propia

Para definir una función en Python se utiliza la palabra reservada **def**. A continuación viene el nombre o identificador de la función que es el que se utiliza para invocar. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. Por último, la cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (con un sangrado mayor, generalmente cuatro espacios) que no es más que el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado.

En último lugar y de manera opcional, se añade la instrucción con la palabra reservada **return** para devolver un resultado.

Parámetros.

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def mi_funcion(nombre, apellido):
      # condición
```

Los parámetros, se indican entre los paréntesis, a modo de variables, a fin de poder utilizarlos como tales, dentro de la misma función.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local. Es decir, que los parámetros serán variables locales, a las cuáles sólo la función podrá acceder:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print nombre_completo
```

```
def mi_funcion(nombre, apellido):
      nombre_completo = nombre, apellido
      print(nombre_completo)
```

Si quisiéramos acceder a esas variables locales, fuera de la función, obtendremos un error:

```
def mi_funcion(nombre, apellido):
```

```
        nombre_completo = nombre, apellido
        print(nombre_completo)


print(nombre)
# NameError: name 'nombre' is not defined
```

Al llamar a una función, **siempre se le deben pasar sus argumentos en el mismo orden en el que los espera**

Función disponible

Función max()

Devuelve el elemento más grande en un iterable o el más grande de dos o más argumentos

Esta función toma dos o más números o cualquier tipo de iterable como argumento. Mientras damos un iterable como argumento, debemos asegurarnos de que todos los elementos del iterable sean del mismo tipo. Esto significa que no podemos pasar una lista que tenga almacenados valores enteros y de cadena. Sintaxis: max (iterable, * iterables [, clave, predeterminado]) max (arg1, arg2, * args [, clave])

```
max(2, 3)

max([1, 2, 3])

max('a','b','c')
```

Se devuelve el elemento más grande del iterable. Si se proporcionan dos o más argumentos posicionales, se devuelve el mayor de los argumentos posicionales. Si el iterable está vacío y no se proporciona el valor predeterminado, se genera un ValueError.

Ejemplo:

```
print(max(2, 3))     # Devuelve 3 ya que 3 es el
mayor de los dos valores

print(max(2, 3, 23)) # Devuelve 23 ya que 23 es el
mayor de todos los valores

list1 = [1, 2, 4, 5, 54]

print(max(list1))    # Devuelve 54 ya que 54 es el
valor más grande de la lista
```

Función min()

Devuelve el elemento más pequeño en un iterable o el más pequeño de dos o más argumentos

Esta función toma dos o más números o cualquier tipo de iterable como argumento. Mientras damos un iterable como argumento, debemos asegurarnos de que todos los elementos del iterable sean del mismo tipo. Esto significa que no podemos pasar una lista que tenga almacenados valores enteros y de cadena.

```
min(2, 3)

min([1, 2, 3])

min('a','b','c')
```

Se devuelve el elemento más pequeño del iterable. Si se proporcionan dos o más argumentos posicionales, se devuelve el más pequeño de los argumentos posicionales. Si el iterable está vacío y no se proporciona el valor predeterminado, se genera un ValueError.

Ejemplo:

```
print(min(2, 3))     # Devuelve 2 ya que 2 es el más
pequeño de los dos valores

print(max(2, 3, -1)) # Devuelve -1 ya que -1 es el
más pequeño de los dos valores

list1 = [1, 2, 4, 5, -54]

print(max(list1))    # Devuelve -54 ya que -54 es el
más pequeño de los valores de la lista
```

Función divmod ( )

Devuelve el cociente y el resto al dividir el número a por el número b. Toma dos números como argumentos a & b. El argumento no puede ser un número complejo.

Se necesitan dos argumentos a & b: un número entero o un número decimal. No puede ser un número complejo.

El valor de retorno será el par de números positivos que consiste en el cociente y el resto obtenido al dividir a por b. En el caso de tipos de operandos mixtos, se aplicarán las reglas para los operadores aritméticos binarios.

Para los **argumentos de números enteros**, el valor de retorno será el mismo que (a // b, a % b).

Para los **argumentos de números decimales**, el valor devuelto será el mismo que (q, a % b), donde q es normalmente math.floor (a / b) pero puede ser 1 menos que eso.

Ejemplo:

```
print(divmod(5, 2))             # muestra (2, 1)

print(divmod(13.5, 2.5))        # muestra (5.0, 1.0)

q,r = print(divmod(13.5, 2.5)) # asigna q = cociente
& r = resto

print(q) # muestra 5.0 porque math.floor(13.5/2.5) =
5.0

print(r) # muestra 1.0 porque (13.5 % 2.5) = 1.0
```

Función Hex(x)

Utilizada para convertir un número entero en una cadena hexadecimal en minúscula con el prefijo "0x"

Esta función toma un argumento, x, que debería ser de tipo entero.

Esta función devuelve una cadena hexadecimal en minúscula con el prefijo "0x".

Ejemplo:

```
print(hex(16))    #muestra 0x10

print(hex(-298))  #muestra -0x12a

print(hex(543))   #muestra 0x21f
```

Función len()

Este método devuelve la longitud (el número de elementos) de un objeto. Toma un argumento x

Esta función toma un argumento, x. Este argumento puede ser una secuencia (como un string, bytes, tupla, lista o rango) o una colección (como un diccionario, conjunto o conjunto congelado).

Esta función devuelve el número de elementos del argumento que se pasa a la función len().

Ejemplo:

```
list1 = [123, 'xyz', 'zara'] # lista

print(len(list1)) # muestra 3 ya que hay 3 elementos
en list1
```

```
str1 = 'basketball' # string

print(len(str1)) # muestra 10 ya que str1 tiene 10 caracteres


tuple1 = (2, 3, 4, 5) # tuple

print(len(tuple1)) # muestra 4 ya que hay 4 elementos en tuple1


dict1 = {'name': 'John', 'age': 4, 'score': 45} # string dictionary

print(len(dict1)) # muestra 3 ya que hay 3 pares clave/valor en dict1
```

Función Ord

Utilizada para convertir la cadena que representa un carácter Unicode en un entero que representa el código Unicode del carácter.

Ejemplos:

```
ord('d')

100

ord('1')

49
```

Función chr

Utilizada para convertir el número entero que representa el código Unicode en una cadena que representa un carácter correspondiente.

Ejemplo:

```
chr(49)
'1'
```

Hay que tener en cuenta que, si el valor entero pasado a chr() está fuera del rango, entonces se generará un ValueError.

```
chr(-10)
'Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    chr(-1)
ValueError: chr() arg not in range(0x110000)
```

Función input

Muchas veces, en un programa necesitamos alguna información del usuario. Recibir información del usuario hace que el programa se sienta interactivo. En Python, para recibir datos del usuario tenemos la función input(). Si se llama a la función input, el flujo del programa se detendrá hasta que el usuario haya introducido la información y haya finalizado la introducción de información con la tecla de retorno. Veamos algunos ejemplos:

Cuando sólo queremos recibir información:

inp = input()

Para mostrar en la línea de comandos un mensaje:

promptwithmessage = input(‘‘)

Cuando queremos tomar un número entero como información:

```
number = int(input('Please enter a number: '))
```

Llamada a una función

Para usar o invocar a una función, simplemente hay que escribir su nombre como si de una instrucción más se tratara. Eso sí, pasando los argumentos necesarios según los parámetros que defina la función.

Ejemplo: Crear una función que muestra por pantalla el resultado de multiplicar un número por cinco:

```
def multiplica_por_5(numero):
      print(f'{numero} * 5 = {numero * 5}'
print('Comienzo del programa')
multiplica_por_5(7)
print('Siguiente')
multiplica_por_5(113)
print('Fin')
```

La función **multiplica_por_5()** define un parámetro llamado número que es el que se utiliza para multiplicar por 5. El resultado del programa anterior sería el siguiente:

```
Comiendo del programa
7 * 5 = 35
Siguiente
113 * 5 = 565
Fin
```

Como se puede observar, el programa comienza su ejecución en la *línea 4* y va ejecutando las instrucciones una a una de manera ordenada. Cuando se encuentra el nombre de la función multiplica_por_5(), el flujo de ejecución pasa a la primera instrucción de la función. Cuando se llega a la última instrucción de la función, el flujo del programa sigue por la instrucción que hay a continuación de la llamada de la función.

Sentencia return

Anteriormente se indicaba que cuando acaba la última instrucción de una función, el flujo del programa continúa por la instrucción que sigue a la llamada de dicha función. Hay una excepción: usar la sentencia **return**. **return** hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal.

Además, **return** se puede utilizar para devolver un valor.

La sentencia **return** es opcional, puede devolver, o no, un valor y es posible que aparezca más de una vez dentro de una misma función.

A continuación algunos ejemplos:

Return que no devuelve ningún valor.

La siguiente función muestra por pantalla el cuadrado de un número solo si este es par:

```
def cuadrado_de_par(numero):
      if not numero % 2 == 0:
            return
      else:
            print(numero ** 2)
cuadrado_de_par(8)
# 64
```

Varios return en una misma función.

La función **es_par()** devuelve True si un número es par y False en caso contrario:

```
def es_par(numero):
      if numero % 2 == 0:
            return True
      else:
            return False
es_par(2)
# True
es_par(5)
# False
```

Devolver más de un valor con return.

En Python, es posible devolver más de un valor con una sola sentencia **return**. Por defecto, con **return** se puede devolver una tupla de

valores. Un ejemplo sería la siguiente función **cuadrado_y_cubo()** que devuelve el cuadrado y el cubo de un número:

```
def cuadrado_y_cubo(numero):
      return numero ** 2, numero ** 3
cuad, cubo = cuadrado_y_cubo(4)


cuad
# 16
cubo
# 64
```

Sin embargo, se puede usar otra técnica devolviendo los diferentes resultados/valores en una lista. Por ejemplo, la función **tabla_del()** que se muestra a continuación hace esto:

```
def tabla_del(numero):
      resultados = []
      for i in range(11):
            resultados.append(numero * i)
      return resultados
res = tabla_del(3)
res
# [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Importación y llamado de módulos.

Un módulo le permite organizar lógicamente su código Python. Agrupar el código relacionado en un módulo hace que el código sea más fácil de entender y usar. Un módulo es un objeto de Python con atributos de nombres arbitrarios que puede enlazar y luego referenciar.

Simplemente, un módulo es un archivo que consta de código Python. Un módulo puede definir funciones, clases y variables. Un módulo también puede incluir código ejecutable.

Ejemplo

El código de Python para un módulo llamado aname normalmente reside en un archivo llamado aname.py. Aquí hay un ejemplo de un módulo simple, support.py

```
def print_func(par):
      print("Hola:" , par)
      return
```

La declaración import

Puede usar cualquier archivo fuente de Python como módulo ejecutando una declaración de importación en algún otro archivo fuente de Python. La importación tiene la siguiente sintaxis:

```
import modulo1 [, modulo 2 [, … modulo N]]
```

Cuando el intérprete encuentra una declaración de importación, importa el módulo si el módulo está presente en la ruta de búsqueda. Una ruta de búsqueda es una lista de directorios que el intérprete busca antes de importar un módulo. Por ejemplo, para importar el módulo support.py,

debe colocar el siguiente comando en la parte superior de la secuencia de comandos:

```
#!/usr/bin/python


# Importar modulo support

import support


#Ahora puede llamar a la función definida de ese
módulo de la siguiente manera

support.print_fun("Sara")
```

Cuando se ejecuta el código anterior, produce el siguiente resultado:

```
Hola Sara
```

Un módulo se carga solo una vez, independientemente de la cantidad de veces que se importe. Esto evita que la ejecución del módulo ocurra una y otra vez si se producen múltiples importaciones.

La declaración from

La declaración from de Python le permite importar atributos específicos de un módulo al espacio de nombres actual. La importación from … tiene la siguiente sintaxis:

```
from modname import name1 [, name2 [, … nameN]]
```

Por ejemplo, para importar la función fibonacci del módulo fib, use la siguiente declaración:

```
from fib import fibonacci
```

Esta declaración no importa la totalidad del módulo fib en el espacio de nombres actual; simplemente introduce el elemento fibonacci del módulo fib en la tabla de símbolos globales del módulo de importación.

*La declaración from … import**

También es posible importar todos los nombres de un módulo al espacio de nombres actual utilizando la siguiente declaración de importación:

```
from modname import *
```

Esto proporciona una manera fácil de importar todos los elementos de un módulo al espacio de nombres actual; sin embargo, esta declaración debe usarse con moderación.

Búsqueda de módulos

Cuando importa un módulo, el intérprete de Python busca el módulo en las siguientes secuencias:

1. El directorio actual.
2. Si no se encuentra el módulo, Python busca cada directorio en la variable de shell PYTHONPATH.
3. Si todo lo demás falla, Python comprueba la ruta predeterminada. En UNIX, esta ruta predeterminada es normalmente /usr/local/lib/python /.

La ruta de búsqueda del módulo se almacena en el módulo del sistema sys como la variable sys.path. La variable sys.path contiene el directorio actual, PYTHONPATH y el valor predeterminado dependiente de la instalación.

La Variable PYTHONPATH

PYTHONPATH es una variable de entorno, que consiste en una lista de directorios. La sintaxis de PYTHONPATH es la misma que la de la variable de shell PATH.

Aquí hay una típica PYTHONPATH de un sistema Windows –

```
set PYTHONPATH = c:\python20\lib
```

Y aquí hay una típica PYTHONPATH de un sistema UNIX –

```
set PYTHONPATH = /usr/local/lib/python
```

Invocación y rango de una función.

La clase range.

Range es un tipo de dato que representa una secuencia de números inmutable.

Para crear un objeto de tipo range, se pueden usar dos constructores :

- *range(fin)*: Crea una secuencia numérica que va desde **0** hasta **fin - 1**.
- *range(inicio, fin, [paso])*: Crea una secuencia numérica que va desde **inicio** hasta **fin - 1**. Si además se indica el parámetro **paso**, la secuencia genera los números de **paso** en **paso**.

Veámoslo con un ejemplo:

```
list(range(10))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(5,10))
# [5, 6, 7, 8, 9]

list(range(0,10,3))
# [0, 3, 6, 9]

list(range(0, -10, -1)
# [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

¿Por qué estamos utilizando en los ejemplos la clase range como argumento de list? **range**, además de un tipo *secuencial* es un tipo *iterable* con una particularidad: a diferencia de los tipos list o tuple, range **calcula los ítems al vuelo, cuando los necesita**. Cómo *list* acepta un objeto *iterable* como parámetro, por eso pasó un objeto *range* al constructor de *list*, para que se muestre por pantalla la secuencia completa que se genera con ***range***.

```
# Nunca se puede ir de 0 a 10 de -1 en -1

list(range(0,10,-1))
# []
```

Recorrer una secuencia numérica

Uno de los principales usos de range() es utilizarlo en bucles for.

```
for i in range (1, 11):

      print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Ventajas de utilizar range

La principal ventaja de usar range sobre list o tuple es que es un iterable que genera los elementos solo en el momento en que realmente los necesita. Esto implica que usa una cantidad de memoria mínima, por muy grande que sea el rango de números que represente.

Veamos una comparación de una lista que almacena los números del 0 al 100.000 y un rango del 0 al 100.000

```
import sys

lista = list(range(0, 100000))

rango = range(0, 100000)

sys.getsizeof(lista)
```

```
# 800056

sys.getsizeof(rango)

# 48
```

Como puedes apreciar, la lista ocupa casi *1 MB* en memoria frente a los *48 bytes* que ocupa el rango.

Operaciones de la clase range

Al tratarse de un tipo secuencial, *range* implementa las operaciones básicas de ese tipo a excepción de la concatenación y la repetición:

```
r = range (0,30,3)
r[2]
# 6

r[-1]
# 27

13 in r
# False

12 in r
# True

r.index(18)
# 6
```

Recuerda, usa la clase *range* para generar secuencias de números, especialmente si las necesitas en un bucle *for*.

Los módulos

Cuando creas programas en Python, los archivos generados suelen tener la extensión particular de Python, ya sabes, .py. Los módulos en Python se crean de la misma manera. Es decir, son archivos de código con extensión .py, y ubicados en un directorio donde Python sea capaz de encontrarlos (es decir, el directorio de trabajo actual o listados en sys.path).

Los módulos contienen normalmente sentencias que están relacionadas entre sí. Como he mencionado anteriormente, podemos utilizar módulos en cualquier momento. El uso de un módulo consiste en utilizar el código (es decir, variables, funciones) almacenado en dicho módulo. El proceso de añadir y utilizar dicho código se llama importación.

Creación de módulos

Crear un módulo en Python es muy simple. Digamos que queremos crear un módulo que imprima el nombre de alguien. Escribe el siguiente código utilizando tu editor favorito, y guardarlo como **myname.py**. Este será el nombre de tu módulo, excluyendo la parte .py, el cual se asignará a la variable global __name__.

```
def print_name(n):
      print('Hola', n)
```

Importar un módulos

Si tienes un archivo de Python y quieres añadir el código del módulo que hemos creado anteriormente, lo importamos utilizando la palabra clave import, tal que así:

```
import myname

name.print_name('Jorge')
```

La salida de este script será: Hola Jorge.

Asegúrate de que Python es capaz de encontrar el archivo importado. Como consejo, ponlo en el mismo directorio que el archivo de Python que está utilizando la importación.

Como puedes ver, la importación de un módulo nos permite mejorar nuestro programa añadiendo nuevas funcionalidades al mismo a partir de archivos externos (es decir, módulos).

Sin embargo, ¿qué está sucediendo aquí, detrás de bambalinas? Cuando importamos un módulo, Python compila ese módulo y genera un archivo .pyc, y el programa sólo se vuelve a compilar si el .py es más reciente que el archivo .pyc.

Tomemos otro ejemplo, pero esta vez con un módulo integrado en Python. Vamos a elegir el módulo de matemáticas math. En este ejemplo, para un número concreto queremos encontrar el redondeo a la alta (el valor entero más pequeño mayor o igual que el número), el redondeo a la baja (valor de número entero más grande menor que o igual al número), y el valor absoluto de ese número. El script de Python para este programa es el siguiente:

```
import math
x = 4.7
print('The ceiling of' + str(x) + ' is: ' + str(math.ceil(x))
print('The floor of ' + str(x) + ' is: ' + str(math.floor(x)))
print('The absolute value of ' + str(x) + ' is: ' +
str(math.fabs(x)))
```

Si ejecutamos este script, la salida sería:

```
The ceiling of 4.7 is: 5.0
The floor of 4.7 is: 5.0
The absolute value of 4.7 is: 4.7
```

Por tanto, hemos sido capaces de aplicar diferentes operaciones en nuestro número sin necesidad de escribir código para cada operación, sino más bien mediante la reutilización de funciones ya disponibles en el módulo math. Eso es de gran utilidad.

Tal vez te preguntes, ¿hay que utilizar siempre la notación math.function()? ¿No podemos utilizar la función de inmediato con el nombre del módulo (es decir, math)? Sí, se puede hacer usando la siguiente sintaxis para la importación:

```
from math import *
```

De esta manera, puedes llamar a las funciones anteriores (ceil(x), floor(x), y fabs(x)) sin la necesidad de precederlos con el nombre del módulo, math.

Módulos como script

Volvamos a nuestro módulo de antes, myname.py:

```
def print_name(n):
      print('Hola', n)
```

¿Podemos tratar este módulo como un proceso independiente que se puede ejecutar directamente y pasarle argumentos mediante línea de comandos? Por ejemplo, ¿qué pasará si escribes lo siguiente en línea de comandos?

```
python myname.py 'Jorge'
```

¡Nada! Inténtalo y no obtendrás ninguna salida.

Con el fin de ser capaz de ejecutar el módulo como un script, tenemos que establecer que __name__ = '__main__'. Por lo tanto, el módulo myname ahora se verá de la siguiente manera:

```
def print_name(n):
      print('Hola', n)

if __name__ == '__main__':
      import sys
      print_name(sys.argv[1])
```

Si ejecutas este comando en tu terminal: python myname.py 'Jorge', obtendrás la siguiente salida:

```
Hola Jorge
```

Expresiones Lambda

En *Python*, una función Lambda se refiere a una **pequeña función anónima**. Las llamamos "funciones anónimas" porque técnicamente carecen de nombre.

Al contrario que una función normal, no la definimos con la palabra clave estándar ***def*** que utilizamos en *Python*. En su lugar, las funciones Lambda se definen como una **línea que ejecuta una sola expresión**. Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión.

Sintaxis básica

Todas las **funciones Lambda** en *Python* tienen exactamente la misma sintaxis:

```
# Escribo p1 y p2 como parámetros 1 y 2 de la función
lambda p1,p2: expresion
```

Como mejor se puede explicar es describiendo un ejemplo básico, vamos a ver una **función normal** y un ejemplo de **Lambda**:

```
# Aquí tenemos una función creada para sumar.
def suma(x,y):
      return(x + y)

# Aquí tenemos una función Lambda que también suma
lambda x,y: x + y

# Para poder utilizarla necesitamos guardarla en una
variable
suma_dos = lambda x,y: x + y
```

Al igual que ocurre en las *list comprehensions*, lo que hemos hecho es escribir el código en una sola línea y limpiar la sintaxis innecesaria.

En lugar de usar *def* para definir nuestra función, hemos utilizado la palabra clave **lambda;** a continuación escribimos **x, y** como argumentos de la función, y *x + y* como expresión. Además, se omite la palabra clave **return**, condensando aún más la sintaxis.

Por último, y aunque la definición es anónima, la almacenamos en la variable *suma_dos* para poder llamarla desde cualquier parte del código, de no ser así tan solo podríamos hacer uso de ella en la línea donde la definamos.

Aplicaciones de Lambdas

Algunas ideas de dónde se podrían aplicar las **Lambdas**. A continuación se han creado **un ejemplo** aplicando las Lambdas con diferentes objetivos. Así se podrá entender mejor cómo funcionan.

Caso de ejemplo

Definir una función de orden superior llamada operar. Llegan como parámetro dos enteros y una función. En el bloque de la función llamar a la función que llega como parámetro y enviar los dos primeros parámetros.

Desde el bloque principal llamar a operar y enviar distintas expresiones lambdas que permitan sumar, restar, multiplicar y dividir.

```
def operar(v1,v2,fn):
      return fn(v1,v2)
resultado1 = operar(10, 3, lambda x1,x2: x1 + x2)
print(resultado1)
# 13
resultado2 = operar(10, 3, lambda x1,x2: x1 - x2)
print(resultado2)
# 7
resultado2 = operar(10, 3, lambda x1,x2: x1 - x2)
print(resultado2)

print(operar(10, 30, lambda x1,x2: x1 * x2))
# 300
print(operar(10, 2, lambda x1,x2: x1 / x2))
# 5.0
```

La función de orden superior operar es la que vimos en el concepto anterior:

```
def operar(v1,v2,fn):
      return fn(v1,v2)
```

La primer expresión lambda podemos identificarla en la primer llamada a la función operar:

```
resultado1 = operar(10, 3, lambda x1,x2: x1 + x2)
print(resultado1)
```

Una expresión lambda comienza por la palabra clave 'lambda' y está compuesta por una serie de parámetros (en este caso x1 y x2), el signo ':' y el cuerpo de la función

```
x1 + x2
```

El nombre de los parámetros se llaman x1 y x2, pero podrían tener cualquier nombre:

```
resultado2 = operar(10, 3, lambda x1,x2: x1 + x2)
```

Se infiere que la suma de x1 y x2 es el valor a retornar.

La segunda llamada a operar le pasamos una expresión lambda similar a la primera llamada:

```
resultado2 = operar(10, 3, lambda x1,x2: x1 - x2)
```

```
print(resultado2)
```

Las otras dos llamadas son iguales:

```
print(operar(10, 30, lambda x1,x2: x1 * x2))
# 300
print(operar(10, 2, lambda x1,x2: x1 / x2))
# 5.0
```

Referencias

[1] Curso Python para principiantes

https://www.youtube.com/watch?v=chPhlsHoEPo

[2] El tipo de dato STR

https://j2logo.com/python/tutorial/tipo-str-python/

[3] Buenas prácticas.

https://recursospython.com/pep8es.pdf

[4] Estructuras de control de flujo

https://jarroba.com/curso-de-python-4-estructuras-de-control-de-flujo/

[5] Operadores de P

https://www.codigofuente.org/operadores-en-python/

[6] Funciones

https://devcode.la/tutoriales/funciones-en-python/

[7] Módulos

https://docs.python.org/es/3/tutorial/modules.html

[8] Expresiones lambda

https://j2logo.com/python/funciones-lambda-en-python/

https://www.youtube.com/watch?v=3lpG5knSjPA