Variables locales y variables globales

Fuente: hektorprofe.net/python-para-impacientes.blogspot.com

Una variable **local** se declara en su ámbito de uso (en el programa principal y dentro de una función) y una **global** fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
# Define función
def acelerar():
    # Declara la variable 'km' como global
    # Ahora se podrá utilizar dentro de la función
    global km

    # Declara variable local (ámbito función)
    tiempo = 1

    # Se incrementa la velocidad en 5 km
    km+= 5

# Define variable local (ámbito programa principal)
km = 10

# Muestra variable 'km'
print('Velocidad:', km)  # velocidad: 10

# Llama a la función acelerar()
acelerar()

# Muestra variable 'km'
print('Velocidad:', km)  # velocidad: 15

# Intenta mostrar la variable 'tiempo'
# Se produce una excepción (error) de tipo NameError
# porque la variable no pertenece a este ámbito:
# NameError: name 'tiempo' is not defined
print('Tiempo:', tiempo)
```

Errores

Los errores detienen la ejecución del programa y tienen varias causas. Para poder estudiarlos mejor vamos a provocar algunos intencionadamente.

Errores de sintaxis

Identificados con el código **SyntaxError**, son los que podemos apreciar repasando el código, por ejemplo al dejarnos de cerrar un paréntesis:

```
print("Hola"
```

Resultado

```
File "<ipython-input-1-8bc9f5174855>", line 1
    print("Hola"
                ^
SyntaxError: unexpected EOF while parsing
```

Errores de nombre

Se producen cuando el sistema interpreta que debe ejecutar alguna función, método... pero no lo encuentra definido. Devuelven el código **NameError**:

```
pint("Hola")
```

Resultado

```
<ipython-input-2-155163d628c2> in <module>()
----> 1 pint("Hola")

NameError: name 'pint' is not defined
```

La mayoría de errores sintácticos y de nombre los identifican los editores de código antes de la ejecución, pero existen otros tipos que pasan más desapercibidos.

Errores semánticos
Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y **dependen de la situación**. Algunas veces pueden ocurrir y otras no.
La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave. Veamos un par de ejemplos:
Ejemplo: pop() con lista vacía
Si intentamos sacar un elemento de una lista vacía, algo que no tiene mucho sentido, el programa dará fallo de tipo **IndexError**. Esta situación ocurre sólo durante la ejecución del programa, por lo que los editores no lo detectarán:

```
l = []
l.pop()
```

Resultado

```
<ipython-input-6-9e6f3717293a> in <module>()
----> 1 l.pop()

IndexError: pop from empty list
```

Para prevenir el error deberíamos comprobar que una lista tenga como mínimo un elemento antes de intentar sacarlo, algo factible utilizando la función **len()**:

```
l = []

if len(l) > 0:
    l.pop()
```

Ejemplo lectura de cadena y operación sin conversión a número

Cuando leemos un valor con la función **input()**, este **siempre** se obtendrá como una **cadena de caracteres**. Si intentamos operarlo directamente con otros números tendremos un fallo **TypeError** que tampoco detectan los editores de código:

```
n = input("Introduce un número: ")
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Resultado

```
Introduce un número: 4

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-85bb893ab3e3> in <module>()
----> 1 print("{}/{} = {}".format(n,m,n/m))

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Como ya sabemos este error se puede prevenir transformando la cadena a entero o flotante:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Resultado

```
Introduce un número: 10
10.0/4 = 2.5
```

Sin embargo no siempre se puede prevenir, como cuando se introduce una cadena que no es un número:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Como podéis suponer, es difícil prevenir fallos que ni siquiera nos habíamos planteado que podían existir. Por suerte para esas situaciones existen las excepciones.

Resultado

```
Introduce un número: aaa

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-c0e7fd4a26a9> in <module>()
----> 1 n = float(input("Introduce un número: "))
      2 m = 4
      3 print("{}/{} = {}".format(n,m,n/m))

ValueError: could not convert string to float: 'aaa'
```

Excepciones

Las excepciones son bloques de código que nos permiten continuar con la ejecución de un programa pese a que ocurra un error.
Siguiendo con el ejemplo de la lección anterior, teníamos el caso en que leíamos un número por teclado, pero el usuario no introduce un número:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Resultado

```
Introduce un número: aaa

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-c0e7fd4a26a9> in <module>()
----> 1 n = float(input("Introduce un número: "))
      2 m = 4
      3 print("{}/{} = {}".format(n,m,n/m))

ValueError: could not convert string to float: 'aaa'
```

Bloques try - except

Para prevenir el fallo debemos poner el código propenso a errores en un bloque try y luego encadenar un bloque except para tratar la situación excepcional mostrando que ha ocurrido un fallo:

```
try:
    n = float(input("Introduce un número: "))
    m = 4
    print("{}/{} = {}".format(n,m,n/m))
except:
    print("Ha ocurrido un error, introduce bien el número")
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
```

Como vemos esta forma nos permite controlar situaciones excepcionales que generalmente darían error y en su lugar mostrar un mensaje o ejecutar una pieza de código alternativo.

Podemos aprovechar las **excepciones** para forzar al usuario a introducir un número haciendo uso de un bucle while, repitiendo la lectura por teclado hasta que lo haga bien y entonces romper el bucle con un break:

```
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{}/{} = {}".format(n,m,n/m))
        break  # Importante romper la iteración si todo ha salido bien
    except:
        print("Ha ocurrido un error, introduce bien el número")
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: 10
10.0/4 = 2.5
```

Bloque else

Es posible encadenar un bloque else después del except para comprobar el caso en que todo funcione correctamente (no se ejecuta la excepción).
El bloque else es un buen momento para romper la iteración con break si todo funciona correctamente:

```
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{}/{} = {}".format(n,m,n/m))
    except:
        print("Ha ocurrido un error, introduce bien el número")
    else:
        print("Todo ha funcionado correctamente")
```

```
        break  # Importante romper la iteración si todo ha salido bien
```

Resultado

```
Introduce un número: 10
10.0/4 = 2.5
Todo ha funcionado correctamente
```

Bloque finally

Por último es posible utilizar un bloque finally que se ejecute al final del código, ocurra o no ocurra un error:

```
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{}/{} = {}".format(n,m,n/m))
    except:
        print("Ha ocurrido un error, introduce bien el número")
    else:
        print("Todo ha funcionado correctamente")
        break  # Importante romper la iteración si todo ha salido bien
    finally:
        print("Fin de la iteración") # Siempre se ejecuta
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
Fin de la iteración
Introduce un número: 10
10.0/4 = 2.5
Todo ha funcionado correctamente
Fin de la iteración
```

Excepciones múltiples

En una misma pieza de código pueden ocurrir **muchos errores distintos** y quizá nos interese actuar de forma diferente en cada caso.
Para esas situaciones algo que podemos hacer es asignar una excepción a una variable. De esta forma es posible analizar el tipo de error que sucede gracias a su identificador:

```
try:
    n = input("Introduce un número: ")  # no transformamos a número
    5/n
except Exception as e:  # guardamos la excepción como una variable e
    print("Ha ocurrido un error =>", type(e).__name__)
```

Resultado

```
Introduce un número: 10
Ha ocurrido un error =>  TypeError
```

Cada error tiene un identificador único que curiosamente se corresponde con su tipo de dato. Aprovechándonos de eso podemos mostrar la clase del error utilizando la sintaxis:

```
print(type(e))
```

Resultado

```
<class 'TypeError'>
```

Es similar a conseguir el tipo (o clase) de cualquier otra variable o valor literal:

```
print(type(1))
print(type(3.14))
print(type([]))
print(type(()))
print(type({}))
```

Resultado

```
<class 'int'>
<class 'float'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

Como vemos siempre nos indica eso de "**class**" delante. Eso es porque en Python todo son clases. Lo importante ahora es que podemos mostrar solo el nombre del tipo de dato (la clase) consultando su propiedad especial **name** de la siguiente forma:

```
print( type(e).__name__)
print(type(1).__name__)
print(type(3.14).__name__)
print(type([]).__name__)
print(type(()).__name__)
print(type({}).__name__)
```

Resultado

```
TypeError
int
float
list
tuple
dict
```

Gracias a los identificadores de errores podemos crear múltiples comprobaciones, siempre que dejemos en último lugar la excepción por defecto **Excepcion** que engloba cualquier tipo de error (si la pusiéramos al principio las demás excepciones nunca se ejecutarán):

```
try:
    n = float(input("Introduce un número divisor: "))
    5/n
except TypeError:
    print("No se puede dividir el número entre una cadena")
except ValueError:
    print("Debes introducir una cadena que sea un número")
except ZeroDivisionError:
    print("No se puede dividir por cero, prueba otro número")
except Exception as e:
    print("Ha ocurrido un error no previsto", type(e).__name__ )
```

Resultado

```
Introduce un número divisor: 0
No se puede dividir por cero, prueba otro número
```

Invocación de excepciones

En algunas ocasiones quizá nos interesa llamar un error manualmente, ya que un print común no es muy elegante:

```
def mi_funcion(algo=None):
    if algo is None:
        print("Error! No se permite un valor nulo (con un print)")

mi_funcion()
```

Resultado

```
Error! No se permite un valor nulo (con un print)
```

Instrucción raise

Gracias a **raise** podemos lanzar un error manual pasándole el identificador. Luego simplemente podemos añadir un except para tratar esta excepción que hemos lanzado:

```
def mi_funcion(algo=None):
    try:
        if algo is None:
            raise ValueError("Error! No se permite un valor nulo")
    except ValueError:
        print("Error! No se permite un valor nulo (desde la excepción)")
```

```
mi_funcion()
```

Resultado

```
Error! No se permite un valor nulo (desde la excepción)
```

Ejercicios resueltos

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
try:
    resultado = 10/0
except ZeroDivisionError:
    print("No es posible dividir entre cero")
```

Resultado

```
Error:  No es posible dividir entre cero
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
lista = [1, 2, 3, 4, 5]
try:
    lista[10]
except IndexError:
    print("El índice se encuentra fuera del rango")
```

Resultado

```
Error:  El índice se encuentra fuera del rango
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
colores = { 'rojo':'red', 'verde':'green', 'negro':'black' }
try:
    colores['blanco']
except KeyError:
    print("La clave del diccionario no se encuentra")
```

Resultado

```
Error:  La clave del diccionario no se encuentra
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
try:
    resultado = "20" + 15
except TypeError:
    print("Sólo es posible sumar datos del mismo tipo")
```

Resultado

```
Error:  Sólo es posible sumar datos del mismo tipo
```

Módulos

Crear un módulo en Python es tan sencillo como crear un script, sólo tenemos que añadir alguna función a un fichero con la extensión .py, por ejemplo **saludos.py**:

```
def saludar():
    print("Hola, te estoy saludando desde la función saludar()")
```

Luego ya podremos utilizarlo desde otro script,por ejemplo **script.py**, en el mismo directorio haciendo un import y el nombre del módulo:

```
import saludos

saludos.saludar()
```

También podemos importar funciones directamente, de esta forma ahorraríamos memoria. Podemos hacerlo utilizando la sintaxis from import:

```
from saludos import saludar

saludar()
```

Para importar todas las funciones con la sintaxis from import debemos poner un asterisco:

```
from saludos import *

saludar()
```

Dicho esto, a parte de funciones también podemos reutilizar clases:

```
class Saludo():
    def __init__(self):
        print("Hola, te estoy saludando desde el __init__")
```

Igual que antes, tendremos que llamar primero al módulo para referirnos a la clase:

```
from saludos import Saludo

s = Saludo()
```

El problema ocurre cuando queremos utilizar nuestro módulo desde un directorio distinto por ejemplo **test/script.py**.

Paquetes

Utilizar paquetes nos ofrece varias ventajas. En primer lugar nos permite unificar distintos módulos bajo un mismo nombre de paquete, pudiendo crear jerarquías de módulos y submódulos, o también subpaquetes.
Por otra parte nos permiten distribuir y manejar fácilmente nuestro código como si fueran librerías instalables de Python. De esta forma se pueden utilizar como módulos estándar desde el intérprete o scripts sin cargarlos previamente.
Para crear un paquete lo que tenemos que hacer es crear un fichero especial **init** vacío en el directorio donde tengamos todos los módulos que queremos agrupar. De esta forma cuando Python recorra este directorio será capaz de interpretar una jerarquía de módulos:

```
paquete
|-__init__.py
|-saludos.py
|-script.py
```

Ahora, si utilizamos un script desde el mismo directorio donde se encuentra el paquete podemos acceder a los módulos, pero esta vez refiriéndonos al paquete y al módulo, así que debemos hacerlo con from import:

```
from paquete.saludos import Saludo

s = Saludo()
```

Esta jerarquía se puede expandir tanto como queramos creando subpaquetes, pero siempre añadiendo el fichero init en cada uno de ellos:

```
script.py
paquete
|
|-__init__.py
|-hola
    |-__init__.py
    |-saludos.py
|-adios
   |-__init__.py
   |-despedidas.py
```

paquete/hola/saludos.py

```
def saludar():
    print("Hola, te estoy saludando desde la función saludar() " \
          "del módulo saludos")

class Saludo():
    def __init__(self):
        print("Hola, te estoy saludando desde el __init__"  \
              "de la clase Saludo")
```

Ahora de una forma bien sencilla podemos ejecutar las funciones y métodos de los módulos de cada subpaquete:

script.py

```
from paquete.hola.saludos import saludar
from paquete.adios.despedidas import Despedida

saludar()
Despedida()
```

Módulos estándar

A continuación os resumo los que son para mí algunos de los módulos esenciales de Python, luego profundizaremos en algunos de ellos:

- **copy**: Se utiliza para crear copias de variables referenciadas en memoria, como colecciones y objetos.
- **collections**: Cuenta con diferentes estructuras de datos.
- **datetime**: Maneja tipos de datos referidos a las fechas/horas.
- **html**, **xml** y **json**: También quiero comentar estos tres módulos, que aunque no los vamos a trabajar, permiten manejar cómodamente estructuras de datos html, xml y json. Son muy utilizados en el desarrollo web.
- **math**: Posiblemente uno de los módulos más importantes de cualquier lenguaje, ya que incluye un montón de funciones para trabajar matemáticamente. Lo veremos más a fondo en esta misma unidad.
- **random**: Este es el cuarto y último módulo que veremos en esta unidad, y sirve para generar contenidos aleatorios, escoger aleatoriamente valores y este tipo de cosas que hacen que un programa tenga comportamientos al azar. Es muy útil en el desarrollo de videojuegos y en la creación de pruebas.
- **sys**: Nos permite conseguir información del entorno del sistema operativo o manejarlo en algunas ocasiones, se considera un módulo avanzado e incluso puede ser peligroso utilizarlo sin conocimiento.
- **threading**: Se trata de otro módulo avanzado que sirve para dividir procesos en subprocesos gracias a distintos hilos de ejecución paralelos. La programación de hilos es compleja y he considerado que es demasiado para un curso básico-medio como éste.
- **tkinter**: Finalmente, y posiblemente el que me hace más ilusión de todos. Tkinter es el módulo de interfaz gráfica de facto en Python. Le dedicaré una unidad entera

bastante extensa en la que aprenderemos a crear formularios con botones, campos de texto y otros componentes.

Módulo collections

El módulo integrado de colecciones nos provee otros tipos o mejoras de las colecciones clásicas.

Contadores

La clase **Counter** es una subclase de diccionario utilizada para realizar cuentas:

Ejemplo

```
from collections import Counter

lista = [1,2,3,4,1,2,3,1,2,1]
Counter(lista)
```

Resultado

```
Counter({1: 4, 2: 3, 3: 2, 4: 1})
```

Ejemplo

```
from collections import Counter

Counter("palabra")
```

Resultado

```
Counter({'a': 3, 'b': 1, 'l': 1, 'p': 1, 'r': 1})
```

Ejemplo

```
from collections import Counter
animales = "gato perro canario perro canario perro"
c = Counter(animales.split())
print(c)

print(c.most_common(1))  # Primer elemento más repetido
print(c.most_common(2))  # Primeros dos elementos más repetidos
```

Resultado

```
Counter({'canario': 2, 'gato': 1, 'perro': 3})
[('perro', 3)]
[('perro', 3), ('canario', 2)]
```

Módulo datetime

Este módulo contiene las clases **time** y **datetime** esenciales para manejar tiempo, horas y fechas.

Clase datetime: Esta clase permite crear objetos para manejar fechas y horas:

```
from datetime import datetime

dt = datetime.now()    # Fecha y hora actual

print(dt)
print(dt.year)         # año
print(dt.month)        # mes
print(dt.day)          # día

print(dt.hour)         # hora
print(dt.minute)       # minutos
print(dt.second)       # segundos
print(dt.microsecond)  # microsegundos

print("{}:{}:{}".format(dt.hour, dt.minute, dt.second))
print("{}/{}/{}".format(dt.day, dt.month, dt.year))
```

Resultado

```
datetime.datetime(2016, 6, 18, 21, 29, 28, 607208)
2016
6
18
21
29
28
607208
21:29:28
18/6/2016
```

Es posible crear un datetime manualmente pasando los parámetros (year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None). Sólo son **obligatorios** el **año**, el **mes** y el **día**.

```
from datetime import datetime

dt = datetime(2000,1,1)
print(dt)
```

Resultado

```
datetime.datetime(2000, 1, 1, 0, 0)
```

No se puede cambiar un atributo al vuelo. Hay que utilizar el método **replace**:

```
dt = dt.replace(year=3000)
print(dt)
```

Resultado

```
datetime.datetime(3000, 1, 1, 0, 0)
```

Módulo math

Este módulo contiene un buen puñado de funciones para manejar números, hacer redondeos, sumatorios precisos, truncamientos... además de constantes.

Redondeos

```
import math

print(math.floor(3.99))  # Redondeo a la baja (suelo)
print(math.ceil(3.01))   # Redondeo al alta (techo)
```

Sumatoria mejorada

```
numeros = [0.9999999, 1, 2, 3])
math.fsum(numeros)
```

Resultado

```
6.9999999
```

Truncamiento

```
math.trunc(123.45)
```

Resultado

```
123
```

Potencias y raíces

```
math.pow(2, 3)  # Potencia
math.sqrt(9)    # Raíz cuadrada (square root)
```

Constantes

```
print(math.pi)  # Constante pi
print(math.e)   # Constante e
```

Módulo random

Aleatoriedad

Este módulo contiene funciones para generar números aleatorios:

```
import random

# Flotante aleatorio >= 0 y < 1.0
print(random.random())

# Flotante aleatorio >= 1 y <10.0
print(random.uniform(1,10))

# Entero aleatorio de 0 a 9, 10 excluído
print(random.randrange(10))

# Entero aleatorio de 0 a 100
print(random.randrange(0,101))

# Entero aleatorio de 0 a 100 cada 2 números, múltiplos de 2
print(random.randrange(0,101,2))

# Entero aleatorio de 0 a 100 cada 5 números, múltiplos de 5
print(random.randrange(0,101,5))
```

Resultado

```
0.12539542779843138
6.272300429556777
7
14
68
25
```

Muestras

También tiene funciones para tomar muestras:

```
# Letra aleatoria
print(random.choice('Hola mundo'))

# Elemento aleatorio
random.choice([1,2,3,4,5])

# Dos elementos aleatorios
random.sample([1,2,3,4,5], 2)
```

Resultado

```
o
3
[3, 4]
```

Mezclas

Y para mezclar colecciones:

```
# Barajar una lista, queda guardado
lista = [1,2,3,4,5]
random.shuffle(lista)
print(lista)
```

Resultado

```
[3, 4, 2, 5, 1]
```

Más documentación sobre los imports:

Ver sección: **Case 3: Importing from Parent Directory** para conocer porque no se puede importar fácilmente desde un directorio padre

https://chrisyeh96.github.io/2017/08/08/definitive-guide-python-imports.html