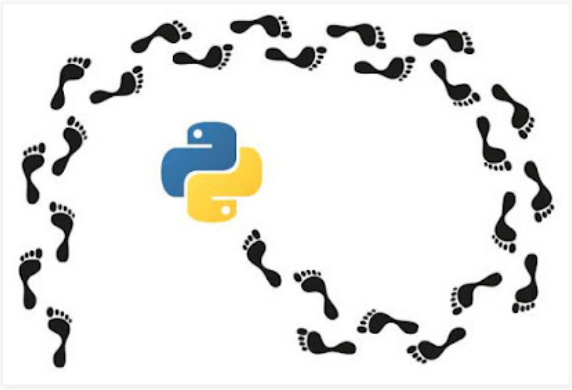
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

sábado, 24 de agosto de 2019

Rastreando la ejecución de un programa (trace)



El módulo **trace** permite seguir el flujo, línea a línea, de la ejecución de un programa y generar distintos informes con las líneas que se ejecutan, las que no y el número de veces que se ejecutan. También, rastrea las llamadas a funciones, las relaciones existentes entre los distintos módulos y facilita el análisis de los datos acumulados obtenidos como resultado de varios rastreos de la ejecución de un programa.

Esta herramienta tiene como objeto ayudar a los desarrolladores a mejorar los algoritmos, permitiendo detectar líneas o bloques de código innecesarios y otros errores de lógica. No se debe confundir con un depurador de código pues no permite detener temporalmente el flujo de la ejecución, ni examinar el valor que tienen las variables en un momento dado.

Rastrear la ejecución de un programa

Para mostrar el uso de las opciones más importantes del módulo **trace** utilizaremos el programa **primos.py** que obtiene todos los números primos que hay entre dos números dados. En Matemáticas, un *número primo* es un número natural mayor que 1 que solo es divisible entre sí mismo y 1.

El código de **primos.py** cuando se ejecuta presenta un mensaje alertando al usuario sobre el tiempo que va a tardar el proceso, que será distinto en función al tamaño del número que determina el límite superior, concretamente, el valor de la variable **superior**. Después, el programa hace una pausa de 3 segundos y continúa con la obtención y visualización de los números primos agrupados por el número de sus dígitos. Para finalizar, calcula el tiempo que ha durado el proceso y ofrece el total de números primos encontrados.

primos.py:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import pprint
import time


inferior = 1
superior = 10
mostrar_listas_primos = True

long_sup = len(str(superior))
if long_sup > 6:
    print('Échate una siesta...\n\n')
elif long_sup > 5:
    print('Date un paseo corto, de un minuto... \n\n')
else:
```

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

```
    print('No te vayas muy lejos. Termino ya... \n\n')
time.sleep(3)

inicio = time.time()
print("Números primos entre", inferior, "y", superior)
print(75*'-')
contador = 0
cont_long = 0
long_inf = len(str(inferior))
lprimos = []
pp = pprint.PrettyPrinter(width=70, compact=True)
for numero in range(inferior, superior + 1):
    longitud = len(str(numero))
    if numero > 1:
        for divisor in range(2, numero):
            if (numero % divisor) == 0:
                break
        else:
            lprimos.append(numero)
            contador += 1
            cont_long += 1

    if longitud != long_inf or numero == superior:
        if mostrar_listas_primos:
            pp.pprint(lprimos)
        print('Total primos con', long_inf, 'dígito/s:', cont_long)
        print(75*'-')
        long_inf = longitud
        cont_long = 0
        lprimos = []

print('Total números primos:', contador)
final = time.time()
tiempo = final - inicio
print(75*'-')
print('El cálculo ha tardado', tiempo, 'segundos' )
```

Rastrear las líneas que se ejecutan

El comando siguiente muestra, una a una, las líneas que se ejecutan en el proceso de obtención de los números primos así como las salidas en las que se imprima alguna información.

```
$ python3 -m trace -t --ignore-dir=/usr/lib/python3.7 primos.py
```

El argumento **-t** se utiliza para rastrear las líneas que se ejecutan.

Para limitar el rastreo a las líneas del programa **primos.py** obviando los módulos importados se agrega el argumento **--ignore-dir** con la ruta donde se encuentra la biblioteca del intérprete Python ¿Cómo conocer esta ruta? Es fácil, ejecutar el siguiente código Python:

```
import os, trace
print(os.path.dirname(trace.__file__))
```

En estas líneas se importan los módulos **os** y **trace** y se imprime la ruta del archivo **trace.py** que se encuentra, entre otros módulos, en dicha biblioteca.

Dependiendo del sistema operativo y de la versión de Python instalada obtendremos rutas similares a las siguientes:

- /usr/lib/python3.7
- C:\Python36\lib
- C:\Users\user1\AppData\Local\Programs\Python\Python37\lib

Continuando con el comando del rastreo, como puede apreciarse en la salida resultante solo aparecen las líneas ejecutadas que muestran el flujo que ha seguido el programa desde principio a fin. Se omiten las líneas no ejecutadas, las primeras por tratarse de comentarios y aquellas que están en blanco. Si la ejecución entra en un bucle, en la salida obtenida aparecerán las líneas tantas veces como se ejecuten, hasta salir del bucle:

Salida:

```
 --- modulename: primos, funcname:
primos.py(4): import pprint
primos.py(5): import time
```

simultáneamente varias operaciones en el mismo espacio de proceso se...

```
primos.py(8): inferior = 1
primos.py(9): superior = 10
primos.py(10): mostrar_listas_primos = True
primos.py(12): long_sup = len(str(superior))
primos.py(13): if long_sup > 6:
primos.py(15): elif long_sup > 5:
primos.py(18):     print('No te vayas muy lejos. Termino ya...\n\n')
No te vayas muy lejos. Termino ya...


primos.py(19): time.sleep(3)
primos.py(21): inicio = time.time()
primos.py(22): print("Números primos entre", inferior, "y",superior)
Números primos entre 1 y 10
primos.py(23): print(75*'-')
---------------------------------------------------------------------------
primos.py(24): contador = 0
primos.py(25): cont_long = 0
primos.py(26): long_inf = len(str(inferior))
primos.py(27): lprimos = []
primos.py(28): pp = pprint.PrettyPrinter(width=70, compact=True)
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(36):             lprimos.append(numero)
primos.py(37):             contador += 1
primos.py(38):             cont_long += 1
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(36):             lprimos.append(numero)
primos.py(37):             contador += 1
primos.py(38):             cont_long += 1
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(34):                 break
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(36):             lprimos.append(numero)
primos.py(37):             contador += 1
primos.py(38):             cont_long += 1
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(34):                 break
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
```

```
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(36):             lprimos.append(numero)
primos.py(37):             contador += 1
primos.py(38):             cont_long += 1
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(34):                 break
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(34):                 break
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(30):     longitud = len(str(numero))
primos.py(31):     if numero > 1:
primos.py(32):         for divisor in range(2, numero):
primos.py(33):             if (numero % divisor) == 0:
primos.py(34):                 break
primos.py(40):     if longitud != long_inf or numero == superior:
primos.py(41):         if mostrar_listas_primos:
primos.py(42):             pp.pprint(lprimos)
[2, 3, 5, 7]
primos.py(43):         print('Con', long_inf, 'díg./s:', cont_long)
Total primos con 1 dígito/s: 4
primos.py(44):         print(75*'-')
---------------------------------------------------------------------------
primos.py(45):         long_inf = longitud
primos.py(46):         cont_long = 0
primos.py(47):         lprimos = []
primos.py(29): for numero in range(inferior, superior + 1):
primos.py(49): print('Total números primos:', contador)
Total números primos: 4
primos.py(50): final = time.time()
primos.py(51): tiempo = final - inicio
primos.py(52): print(75*'-')
---------------------------------------------------------------------------
primos.py(53): print('El cálculo ha tardado', tiempo, 'segundos' )
El cálculo ha tardado 0.0014536380767822266 segundos
```

También, como alternativa a omitir el rastreo de la biblioteca Python, se puede omitir el rastreo de uno o más módulos con el argumento **--ignore-module**:

```
$ python3 -m trace -t --ignore-module=_bootstrap,_bootstrap_external primos.py
```

En este caso además de rastrear las líneas ejecutadas del programa **primos.py** se traza la ejecución de las líneas ejecutadas del módulo **pprint**. Las funciones **time()** o **sleep()** del módulo **time** no aparecen porque la mayoría de las funciones definidas en este módulo llaman a funciones de la librería de C.

Rastrear, contar líneas y obtener líneas no ejecutadas

Para rastrear las líneas ejecutadas, las no ejecutadas y almacenar en un archivo **cover** un informe con el código con el número de veces que se ejecuta cada línea:

```
$ python3 -m trace --count -C . -t -m --ignore-dir=/usr/lib/python3.7 primos.py
```

El argumento **--count** indica al trazador que debe contar el número de veces que se ejecutan las líneas en el programa y en cualquier módulo que se utilice que no se haya excluido. El total de ejecuciones aparece precediendo al código de cada línea.

El argumento **-C** (**--converdir**) establece el directorio donde se crean los informes (con un punto **.** se indica que el directorio es el mismo donde se encuentre el programa o el módulo ejecutado).

El argumento **-m** (**--missing**) se utiliza para recoger las líneas que no se ejecutan que aparecen señaladas con la cadena "**>>>>>>**". Esta opción resulta de mucha utilidad para identificar

posibles líneas de código innecesarias.

En este caso se obtiene la misma salida por pantalla que en el primer ejemplo pero además se genera un archivo llamado **primos.cover** con el siguiente contenido:

primos.cover:

```
       #!/usr/bin/env python
       # -*- coding: utf-8 -*-

    1: import pprint
    1: import time


    1: inferior = 1
    1: superior = 10
    1: mostrar_listas_primos = True

    1: long_sup = len(str(superior))
    1: if long_sup > 6:
>>>>>>     print('Échate una siesta...\n\n')
    1: elif long_sup > 5:
>>>>>>     print('Date un paseo corto, de un minuto... \n\n')
       else:
    1:     print('No te vayas muy lejos. Termino ya... \n\n')
    1: time.sleep(5)

    1: inicio = time.time()
    1: print("Números primos entre", inferior, "y", superior)
    1: print(75*'-')
    1: contador = 0
    1: cont_long = 0
    1: long_inf = len(str(inferior))
    1: lprimos = []
    1: pp = pprint.PrettyPrinter(width=70, compact=True)
   11: for numero in range(inferior, superior + 1):
   10:     longitud = len(str(numero))
   10:     if numero > 1:
   19:         for divisor in range(2, numero):
   15:             if (numero % divisor) == 0:
    5:                 break
            else:
    4:             lprimos.append(numero)
    4:             contador += 1
    4:             cont_long += 1

   10:     if longitud != long_inf or numero == superior:
    1:         if mostrar_listas_primos:
    1:             pp.pprint(lprimos)
    1:         print('T primos con', long_inf, 'díg./s:', cont_long)
    1:         print(75*'-')
    1:         long_inf = longitud
    1:         cont_long = 0
    1:         lprimos = []

    1: print('Total números primos:', contador)
    1: final = time.time()
    1: tiempo = final - inicio
    1: print(75*'-')
    1: print('El cálculo ha tardado', tiempo, 'segundos' )
```

Rastrear llamadas a funciones

El argumento **--listfuncs** que no es compatible con rastrear y contar las líneas ejecutadas (opciones **-t** y **-c**, respectivamente) muestra una lista con las funciones que han sido llamadas durante la ejecución.

```
$ python3 -m trace --listfuncs primos.py
```

Extracto de la salida:

```
No te vayas muy lejos. Termino ya...


Números primos entre 1 y 10
----------------------------------------------------------------------
```

```
[2, 3, 5, 7]
Total primos con 1 dígito/s: 4
------------------------------------------------------------------------
Total números primos: 4
------------------------------------------------------------------------
El cálculo ha tardado 0.013412952423095703 segundos

functions called:
filename: .../pprint.py, modulename: pprint, funcname:
filename: .../pprint.py, modulename: pprint, funcname: PrettyPrinter

...
...
```

Rastrear relaciones entre módulos por llamadas a funciones

El argumento **--trackcalls** muestra las relaciones existentes entre los distintos módulos por las llamadas a funciones efectuadas durante la ejecución, ampliando los detalles del comando anterior:

```
$ python3 -m trace --listfuncs --trackcalls primos.py
```

Extracto de la salida:

```
No te vayas muy lejos. Termino ya...


Números primos entre 1 y 10
------------------------------------------------------------------------
[2, 3, 5, 7]
Total primos con 1 dígito/s: 4
------------------------------------------------------------------------
Total números primos: 4
------------------------------------------------------------------------
El cálculo ha tardado 0.010804176330566406 segundos

calling relationships:

*** /usr/lib/python3.7/pprint.py ***
    pprint. -> pprint.PrettyPrinter
    pprint. -> pprint._safe_key
  -->
    pprint. ->  pprint.PrettyPrinter._repr
    pprint.PrettyPrinter._repr -> pprint.PrettyPrinter.format
    pprint.PrettyPrinter.format -> pprint._safe_repr
    pprint.PrettyPrinter.pprint -> pprint.PrettyPrinter._format
    pprint._safe_repr -> pprint._safe_repr

*** /usr/lib/python3.7/trace.py ***
    trace.Trace.runctx -> trace._unsettrace
  --> primos.py
    trace.Trace.runctx -> primos.

...
...
```

Rastrear y obtener informes combinados

El argumento **-r** (**--report**) genera una lista anotada a partir de uno o varios rastreos anteriores del programa en los que se utilizan las opciones **-c** (**--count**) y **-f** (**--file**). Este argumento por si solo no ejecuta ningún código, simplemente acumula los datos obtenidos en distintos rastreos en los que es habitual que se modifiquen antes los valor de variables para conocer el comportamiento del código en situaciones diferentes.

Para mostrar el funcionamiento con el programa **primos.py** trazaremos el código tres veces modificando antes el valor de la variable **superior** y después obtendremos un informe combinado. En el primer rastreo la variable superior tendrá el valor **10**; en el segundo, **100** y en el tercero y último, **1000**.

En este caso crear en la ubicación del programa **primos.py** un directorio llamado **inf** para guardar en esa ubicación los archivos **cover** y el archivo que acumulará los datos **inf.dat** del argumento **-f** (**--file**):

```
# En Windows:

C:\> md inf

# En GNU/Linux:

$ mkdir inf
```

A continuación, antes de cada rastreo cambiar el valor de la variable superior. En el primer rastreo se produce el error **[Errno 2]** porque no existe todavía el archivo **inf.dat**. Es normal. Después de su ejecución en la carpeta **inf** se almacenan los archivos .**cover** y en pantalla se produce la salida con el flujo de la ejecución. También, hay que tener en cuenta que como no se excluye del rastreo el directorio de la biblioteca de Python, se obtienen tres archivos .**cover** (**primos.cover**, **pprint.cover** y **trace.cover**) además del acumulado **inf.dat**:

```
# Rastreo 1 con variable superior = 10:

$ python3 -m trace --count -C inf -f inf/inf.dat -t primos.py


# Rastreo 2 con variable superior = 100:

$ python3 -m trace --count -C inf -f inf/inf.dat -t primos.py


# Rastreo 3 con variable superior = 1000:

$ python3 -m trace --count -C inf -f inf/inf.dat -t primos.py


# Generar informe combinado:

$ python3 -m trace -C inf -r -s -m -f inf/inf.dat -t primos.py
```

El comando anterior actualiza los archivos .**cover** acumulando sus datos y muestra en la salida de pantalla un resumen **-s** (**--summary**) con el número de líneas ejecutadas de cada módulo y el porcentaje que representan estas líneas con respecto al total, después de los tres rastreos:

Salida:

```
lines   cov%   module   (path)
  448    36%   pprint   (/usr/lib/python3.7/pprint.py)
   42    95%   primos   (primos.py)
  457     0%   trace    (/usr/lib/python3.7/trace.py)
```

Como puede observarse en la salida, el programa **primos.py** muestra un 95% del código utilizado. ¿De este valor se podría deducir que un 5% del código es desechable?. Bueno, para responder a esta pregunta es necesario analizar las líneas no ejecutadas en el archivo **primos.cover**, que aparecen porque se incluyó el argumento **-m** (**--missing**) en el comando anterior:

```
       #!/usr/bin/env python
       # -*- coding: utf-8 -*-

    3: import pprint
    3: import time


    3: inferior = 1
    3: superior = 10
    3: mostrar_listas_primos = True

    3: long_sup = len(str(superior))
    3: if long_sup > 6:
>>>>>>     print('Échate una siesta...\n\n')
    3: elif long_sup > 5:
>>>>>>     print('Date un paseo corto, de un minuto... \n\n')
       else:
    3:     print('No te vayas muy lejos. Termino ya... \n\n')
    3: time.sleep(5)

    3: inicio = time.time()
    3: print("Números primos entre", inferior, "y", superior)
    3: print(75*'-')
    3: contador = 0
    3: cont_long = 0
```

```
    3: long_inf = len(str(inferior))
    3: lprimos = []
    3: pp = pprint.PrettyPrinter(width=70, compact=True)
 1113: for numero in range(inferior, superior + 1):
 1110:     longitud = len(str(numero))
 1110:     if numero > 1:
79367:         for divisor in range(2, numero):
79170:             if (numero % divisor) == 0:
  910:                 break
              else:
  197:             lprimos.append(numero)
  197:             contador += 1
  197:             cont_long += 1

 1110:     if longitud != long_inf or numero == superior:
    6:         if mostrar_listas_primos:
    6:             pp.pprint(lprimos)
    6:         print('T primos con', long_inf, 'díg./s:', cont_long)
    6:         print(75*'-')
    6:         long_inf = longitud
    6:         cont_long = 0
    6:         lprimos = []

    3: print('Total números primos:', contador)
    3: final = time.time()
    3: tiempo = final - inicio
    3: print(75*'-')
    3: print('El cálculo ha tardado', tiempo, 'segundos' )
```

Con respecto a las líneas no ejecutadas (las precedidas con **>>>>>>**) se puede concluir que son necesarias para otros casos en los que el valor de la variable **superior** sea mayor a los valores que hemos usado en el ejemplo.

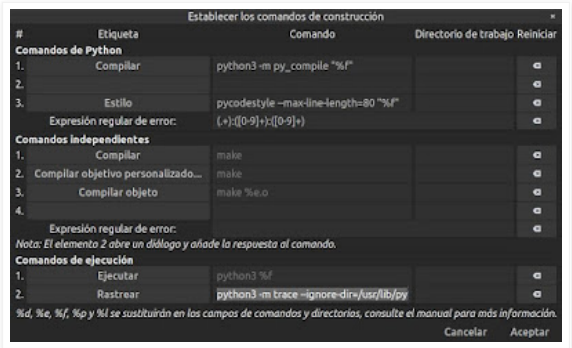
También, como el rastreo del programa se realizó tres veces, algunas líneas muestran que se han ejecutado ese número de veces aunque, lógicamente, todas acumulan igualmente el número total de ejecuciones gracias al argumento **--count**.

Agregar opción para rastrear en el editor de código

Para finalizar hacemos la sugerencia de agregar una opción "**Rastrear**" al editor de código fuente, siempre que éste lo permita, para hacer del uso de **trace** algo habitual.

En Geany, editor de referencia de este blog, se puede agregar una opción que ejecute el comando de rasreo cuando sea necesario. Para ello, crear un nuevo documento Python o abrir alguno existente y acceder al menú **Construir**, **Establecer comandos de construcción**. Después, en el apartado "**Comandos de ejecución**" de la nueva ventana agregar en el espacio reservado para el comando número 2 (si no se usa para otra tarea) la etiqueta "**Rastrear**" asociada al comando siguiente y aceptar los cambios:

```
python3 -m trace --ignore-dir=/usr/lib/python3.7 --trace "%f"
```



Para concluir apuntar que el módulo **trace** se puede utilizar también desde un programa. Para conocer los detalles y consultar más opciones recomendamos consultar la documentación oficial de Python.

Relacionado:

- Solucionando errores con el depurador
- El depurador PuDB
- Editar y depurar scripts (IPython)
- Facilitando la depuración de programas con breakpoint()

Ir al índice del tutorial de Python

Publicado por Pherkad en 13:15

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua