Tema 2: Ejecución y manejo de programas

Héctor Xavier Limón Riaño

April 3, 2024

Contents

1 Ejecución desde línea de comandos

- Como ya se mencionó en temas anteriores, el curso se centra en programas de línea de comandos
- La línea de comandos es la interfaz de usuario más utilizada en la automatización de tareas
- A lo largo de la carrera es necesario tener familiaridad con esta interfaz

1.1 Interfaz de línea de comandos

- Se requiere tener dos programas:
  - Emulador de terminal: simula una terminal física
    - En Linux hay muchas opciones de emuladores, por ejemplo: Konsole, alacritty, iTerm, kitty, etc.
    - En windows se tiene por defecto la terminal del sistema (no recomendable), y powershell que incluye un emulador
  - Shell de sistema: es el intérprete que recibe comandos, suele ser un lenguaje de shell scripting completo. Ejemplos:
    - bash
    - fish
    - zsh
    - powershell
  - En windows se suele usar powershell, mientras que en linux hay más opciones, siendo tradicionalmente `bash` la opción por defecto

  - Los IDEs suelen incluir algún emulador y shell de sistema
  - La línea de comandos siempre está situada sobre un directorio (llamado directorio actual)
  - Para cambiar de directorio se puede:
    * Comando `cd` en Linux
    * Comando `dir` en Windows

```
cd /tmp/prueba # se cambia el directorio actual, hay que saber la ruta

dir .. # regresar un nivel de directorio con respecto al actual (.. tambié
```

1.2 Ejecución de programas

- Para ejecutar un programa desde línea de comandos es necesario saber tres cosas:
  - Cómo se llama el programa: esto es, el nombre del archivo principal que contiene al programa
  - El archivo correspondiente es ejecutable:
    * Windows: tiene la extensión `.exe`
    * Linux: tiene permisos de ejecución
  - La ruta donde se encuentra dicho archivo: la ruta es el conjunto de directorios para llegar al archivo

```
 # Ejemplo de rutas en Linux
 /tmp # solo directorio

# La ruta completa de este archivo, una ruta puede ser de archivo o directorio
/home/xl666/oneDrive/Dropbox/clases/estructurasCiber/notas/notasEstructuras24/tema2/ter

# Ejemplo de ruta en Windows
C:\Users\Username\Documents\example.txt
```

- La primera palabra que se pone en la línea de comandos es el archivo donde está el programa (comando)
- Después se pueden agregar parámetros al programa

```
# Forma de usar comandos, si el comando está en el PATH
nombreComando param1 param2 paramN
```

```
# Forma de usar comando si no está en el PATH
/ruta/comando param1 param2 paramN

# Si configuraste bien Python, el comando Python está en el PATH
python script.py param1 param2 paramN

# Si script no está en directorio actual
python /ruta/script.py param1 param2 paramN
```

2 Paso de parámetros a un programa

- Es una de las formas más sencillas de pasar entradas a un programa
- Muy similar a pasar parámetros a una función
- No confundir con leer de entrada estándar, como lo hace la función `input`, los parámetros se procesan de forma diferente
- En general, siempre prefiere usar parámetros en vez de entrada estándar (excepto en el sistema del curso donde sólo se pueden pasar entradas por entrada estándar)
- Los parámetros permiten automatizar programas con scripts de sistema de forma más sencilla (cubierto en otro materia de la carrera)
- El tipo de parámetros más simple son los posicionales (sólo se cubrirá ese tipo en el curos), aunque existen otros (modificadores, variables)

```
# ejemplo de modificadores
ls -l -a

# ejemplo de variables
# copia archivo1 y archivo2 a /tmp
cp archivo1 archivo2 /tmp/

# copia archivo1, archivo2, archivo3 a /tmp
cp archivo1 archivo2 archivo3 /tmp
```

2.1 En Python

- Se usa el módulo `sys` que cuenta con una lista `argv` con los parámetros que pasas por línea de comandos
- Los parámetros siempre se pasan como cadena (tipo `str`), de ser necesario hay que hacer conversiones de tipo

```
import sys

param1 = sys.argv[1]
param2 = sys.argv[2]
# y así sucesivamente

sys.argv[0] # es la ruta del script que ejecutas
```

- Un ejemplo más largo
- Un programa que suma dos números que se reciben como parámetro

```
import sys

if len(sys.argv) != 3: # hay que contar posición 0
    print('Error: se esperaban dos parámetros')
    exit(1) # terminación con error

numero1 = int(sys.argv[1])
numero2 = int(sys.argv[2])

print(numero1 + numero2)
```

3 Manejo de errores

- En general existen 3 tipos de errores en un programa:
  - Errores en tiempo de compilación/traducción
  - Errores en tiempo de ejecución (excepciones)
  - Errores lógicos

3.1 Errores en tiempo de compilación/traducción

- Son errores que arroja el traductor antes de que se pueda correr el código, osea que suceden como parte del proceso de traducción
- Este tipo de errores tiene 3 categorías:
  - Errores léxicos:
    - Se refieren a no poder separar apropiadamente partes del programa (tokens)
    - Ejemplos puede ser empezar un identificador con un número, utilizar símbolos no soportados como `@`, usar guiones - en un identificador
  - Errores sintácticos
    - Suceden cuando no te apegas a la gramática del lenguaje
    - La gramática es un conjunto de reglas formales que establecen las formas válidas del lenguaje (sentencias y expresiones)
    - Ejemplos es no poner : al iniciar un if, no utilizar correctamente la identación, no usar operadores para conectar expresiones, etc.
  - Errores semánticos
    - Son errores que suceden cuando algo no tiene sentido
    - Principalmente tratar de hacer cosas entre tipos que no son compatibles entre si
    - Por ejemplo: invocar una función definida con tres parámetros obligatorios pero pasar sólo dos parámetros, tratar de hacer una suma entre un entero y una cadena, invocar una función que no existe
- En un lenguaje compilado, si tienes errores de este tipo no se puede generar un ejecutable y por lo tanto no se puede correr el programa
- En un lenguaje interpretado como Python el error podría manifestarse hasta llegar a la línea con el problema, siendo posible que otras partes del programa se hayan ejecutado hasta entonces (lo cual no es bueno)
- Esta es una gran diferencia entre lenguajes compilados e interpretados, en general se considera que los lenguajes compilados son mas "confiables" dado que previenen este tipo de errores

- Para mitigar este problema, Python hace una revisión del código fuente antes de empezar a interpretar, sin embargo, la revisión se limita a cuestiones léxicas y sintácticas, no se cachan problemas semánticos

3.2 Errores en tiempo de ejecución

- También llamados excepciones
- Son errores que suceden mientras el programa se ejecuta, esto es, cuando ya se convirtió en un proceso del sistema (verán procesos en sistemas operativos)
- Se dan principalmente por problemas en el ambiente de ejecución o por entradas externas (como las de un usuario)
- Estos problemas son impredecibles (no puedes saber si van a ocurrir o no a priori)
- Este tipo de errores no pueden ser atrapados por el traductor
- Ejemplos:
  - Tratar de hacer una conexión a un manejador de base de datos, pero no hay red en ese momento
  - Tratar de leer un archivo pero éste no existe
  - Tratar de dividir dos números pero el usuario pasó 0 en el denominador
  - Intentar guardar un archivo pero ya no hay espacio en disco

3.2.1 Manejo de excepciones

- Varios lenguajes de programación (incluyendo Python) cuenta con mecanismos para poder manejar excepciones y recuperarse de ellas
- Este mecanismo es conocido como `try-catch` o en el caso de Python `try-except`
- Este es un tipo de sentencia que tiene tres partes:
  - Bloque `try` : en este bloque se coloca el código que se sabe pudiera provocar una excepción. La ejecución del bloque se interrumpe en el momento que ocurre la excepción (si es que ocurre)

  - Bloque(s) `except` : se entra en este bloque si es que ocurre una excepción en el boque `try`. Es código para recuperarse del error; avisar de él o registrarlo; o regresar un valor por defecto. Puede haber varios bloques `except` dependiendo del tipo de excepción (no se verá en este curso)
  - Bloque `fianally` : es opcional, este bloque se ejecuta independientemente de si ocurre o no una excepción. Útil para cerrar recursos (como archivos o conexiones) de forma confiable

```
# Forma
try:
    sentencias_expresiones_normales
    sentencias_expresiones_con_posible_excepcion
    sentencias_expresiones_normales
except:
    sentencias_expresiones_para_manejar_error
finally:
    sentencias_expresiones_ejecutar_siempre
```

- Un ejemplo más largo

```
import sys

if len(sys.argv) < 3: # hay que contar posición 0
    print('Error: se esperaban dos parámetros')
    exit(1) # terminación con error

try: # la conversión puede fallar
    numero1 = int(sys.argv[1])
    numero2 = int(sys.argv[2])
    print(numero1 + numero2)
except:
    print('Error: se esperaba que pasaras números')
    exit(1)
finally:
    print('Fin del programa')
```

- Las excepciones se propagan, si no las cachas en una función se propagan a quien mandó a llamar a la función

- Si en ningún momento se atrapa la excepción, la excepción causa la terminación del programa
- Si eso sucede, aparece el `stack-trace` del error en la consola, éste es un mensaje con detalles del error que describe cómo se fue propagando la excepción
- Estos mensajes ayudan a que el programador corrija el error (no te espantes)

```
def va_a_fallar():
  v = 5/0
  return True

def invocadora():
  try: # se detiene propagación
    b = va_a_fallar()
  except:
    b = False
  return b

invocadora() # también se podría poner acá el try
```

3.3 Errores lógicos

- Son el tipo de error más complicado de detectar y corregir
- Son errores que no necesariamente generan excepciones (aunque podrían hacerlo), por lo que pueden ser silenciosos
- Tienen que ver con que el programa no haga lo que se espera, arrojando resultados inválidos
- Suelen requerir de herramientas (como los depuradores o debuggers) y técnicas de depuración para poder corregirlos

3.4 Consejos para depurar código

- La corrección de errores, también llamada depuración (debugging), es una habilidad fundamental de todo programador

- Todos nos equivocamos (sin importar tu nivel) por distintas razones, no se puede esperar que los errores no ocurran
- Podría decirse que eres tan buen programador como depurador de errores
- Siempre lee los `stack-trace` de abajo hacía arriba (empieza por el final del mensaje)
- Lo que está al último es el primer punto del código donde surgió la excepción (los demás puntos es donde se fue propagando)
- Los errores normalmente están en el punto donde surge la excepción
- En un `stack-trace` analiza el tipo (o tipos) de error que se reporta
- Para mantener a raya la cantidad de errores: SIEMPRE después de programar una rutina (una función por ejemplo) pruébala de inmediato, no esperes a tener muchas rutinas para empezar a probar, va a ser más difícil saber dónde está el error y corregirlo
- Para que el consejo anterior funcione, tus rutinas deben de ser lo más cohesivas posible (hacer solo una cosa) así serán más fáciles de probar
- Si quieres ir más allá, hay metodologías de programación (TDD por ejemplo) que plantean definir las pruebas antes de programar (es un tema avanzado para este curso)
- Hay varias técnicas y herramientas que sirven para depurar errores, en este curso se presenta una de las técnicas más simples a continuación

3.4.1 Técnica print-trace

- Consiste en primero analizar el error y sus efectos
- Se plantea una hipótesis (o varias) sobre porqué se cree que el error ocurre
- Se investiga el error y la hipótesis poniendo mensajes de impresión (print) en puntos del código de los cuales se sospecha
- Estos mensajes de print pueden mostrar valores de variables con resultados intermedios, o bien corroborar si se está invocando una función o se está entrando a un bloque de código al que debería entrarse

- Se ejecuta el programa tratando de reproducir el error y se analizan los mensajes arrojados
- Si después del análisis no se ha descubierto el problema, se agregan o mueven los prints o bien se cambia de hipótesis
- Una vez se encuentra el problema, se corrige y se prueba que ya no ocurre el error
- Finalmente se borran los mensajes de print (no hay que dejarlos)

4 Módulos y paquetes

- Dependiendo del tamaño de un proyecto, se vuelve necesario dividirlo en varias partes y subpartes
- La división se realiza en la etapa de diseño y permite segmentar la complejidad del problema
- Esta división también permite repartir responsabilidades entre un equipo de desarrolladores
- En python se puede dividir un proyecto en tres niveles (niveles de diseño):
  - Funciones/clases: el problema se divide en varias rutinas o clases (POO) en un solo archivo `.py`
  - Módulos: cada archivo `.py` es un módulo, un proyecto puede consistir de varios módulos en el mismo directorio. Cada módulo se encarga de un aspecto específico del problema
  - Paquetes: se pueden tener varios directorios, cada uno con sus propios módulos. Cada paquete tiene una responsabilidad de acuerdo a la arquitectura del sistema (acceso a datos, lógica de negocios, interfaces de usuario, etc)

4.1 Creación y uso de módulos

- Si quieres crear un módulo, basta con crear un archivo `.py`
- Si pones varios módulos en el mismo directorio, puedes usar un módulo dentro de otro de la siguiente manera:

```
import otro_modulo

# usar rutinas
otro_modulo.rutina()
```

- Se pueden usar alias (útil cuando el nombre es muy largo)

```
import otro_modulo.sub.sub2 as otro

otro.rutina()
```

- Se pueden importar sola partes que interesan, ahorrando escribir el nombre del módulo o paquete

```
# importar des rutinas
from otro_modulo import rutina1 as r1, rutina2
r1()
rutina2()

# se pueden importar todas, no recomendable
from otro_modulo import *
rutina1()
rutina2()
rutina3()
```

4.1.1 Módulo `__main__`

- Cada módulo tiene un nombre asociado, el cual se puede recuperar mediante la variable especial `__name__`
- Esta variable la tienen todos los módulos
- En general su valor no es muy relevante, pero en el caso del módulo que ejecutas directamente (desde la línea de comandos por ejemplo), su nombre tiene un valor especial, el cual es `__main__`

```
print(__name__)
import os
print(os.__name__)
```

```
__main__
os
```

- Esta característica es bastante útil para poner código que sólo quieres que se ejecute cuando el módulo es el principal (el que se ejecuta directamente)
- El código que dejas suelto, fuera de una función o clase, se ejecuta cuando importas los módulos
- Con esto se pueden lograr algo similar a la rutina `main` de lenguajes como `C` o `Java`
- También es útil para poner código de prueba en módulos secundarios

```
# poner esto al final de tus script
if __name__ == '__main__':
    print('solo se ejecuta si es el módulo principal')
```

- El bloque de código anterior es una buena práctica y se estará usando en el curso a partir de ahora

4.1.2 Usar módulos y paquetes de terceros

- Se refiere a importar módulos que fueron creados por otros programadores
- Pueden ser módulos que trae el propio lenguaje, o que se descargan

```
# sacar fecha y hora actual del sistema
import datetime

# el módulo datetime tiene una clase también llamada datetime
print(datetime.datetime.now())

# otra forma más cómoda
from datetime import datetime as fecha
print(fecha.now())
```

```
2024-02-27 09:24:49.005863
2024-02-27 09:24:49.005893
```

1. Buscar ayuda

   - Para obtener ayuda referente al uso de un módulo

     ```
     import datetime
     # ver todo lo que está definido (funciones, variables/constantes, clases, subr
     dir(datetime)

     # ver ayuda de ese módulo
     help(datetime)

     # ver ayuda de una clase o rutina
     help(datetime.datetime.now)
     ```

   - También puedes visitar la documentación oficial: `https://docs.python.org`
   - Muchos IDEs incluyen la documentación oficial del lenguaje

2. Gestor de paquetes

   - Python cuenta con su propio gestor de paquetes de terceros (`pip`), para que sea más fácil instalar correctamente los paquetes y que estén disponibles en cualquier proyecto
   - Instalar un nuevo paquete con pip

     ```
     pip install paquete
     ```

     - Python cuenta con una gran comunidad de paquetes de terceros
     - Repositorio oficial: `https://pypi.org/`

4.1.3 El zen de Python

```
import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

4.2 Creación y uso de paquetes

- Se crea un directorio por paquete, en cada directorio se pone un archivo especial `__init__.py`
- El archivo especial puede estar vacío
- Por ejemplo, considera la siguiente estructura:

```
"""
my_package/
 __init__.py
 module1.py
 subpackage/
     __init__.py
     module2.py
"""
```

- Para usar modulo2 desde modulo1:

```
from subpackage import module2

module2.rutina()
```

- Para fines del curso, incluso en proyectos, no será necesario definir paquetes, con tener todos los módulos en el mismo directorio es suficiente