O Shell

> Python para automação, devops e testes na LINUXtips com Bruno Rocha.

O primeiro requisito para automatizar qualquer tarefa de infraestrutura é dominar o shell, em nosso caso o `bash` que é shell default na maioria dos sistemas Linux.

O shell é a interface de comunicação para acessar serviços do sistema operacional e monitorar e interagir com processos.

Geralmente quando falamos a palavra `ambiente` estamos nos referindo a um `shell` e portanto vamos começar este treinamento vendo como podemos usar Python para interagir com as funcionalidades mais básicas do sistema operacional e do Shell.

Ambiente

Eu estou utilizado Linux e tenho um emulador de terminal aberto com o bash, fique a vontade para usar o Linux de sua preferência e o emulador de terminal que tiver instalado.

Caso não tenha eu recomendo usar o Terminator ou o Gnome-Terminal no Linux, mas vai funcionar também se estiver no Mac com iterm ou no windows com WSL.

Uma outra opção é usar o gitpod.io ou o Github Codespaces que oferecem um plano gratuito onde existe o acesso a um shell Linux.

É também importante que neste shell você tenha como executar containers com docker pois vamos utilizar bastante em breve.

Variáveis de ambiente

Variáveis de ambiente no Linux são variáveis que armazenam informações que o sistema operacional e os programas podem acessar. Essas variáveis contêm dados importantes, como configurações do sistema, caminhos para diretórios de programas, informações de usuário e muito mais. As variáveis de ambiente são usadas para influenciar o comportamento dos programas e scripts, permitindo que eles obtenham informações sobre o ambiente em que estão sendo executados.

Aqui estão algumas das variáveis de ambiente mais comuns no Linux e suas descrições:

1. **$PATH**: O PATH é uma lista de diretórios separados por dois-pontos que contém os caminhos para os executáveis dos comandos do sistema. Quando você digita um comando no terminal, o sistema procura nos diretórios listados no PATH para encontrar o programa correspondente.
2. **$HOME**: O HOME é o diretório inicial do usuário. Ele aponta para o diretório pessoal do usuário atual e é frequentemente usado em scripts para acessar arquivos e diretórios específicos do usuário.
3. **$USER**: Contém o nome do usuário que está atualmente logado no sistema.
4. **$SHELL**: Essa variável armazena o shell padrão do usuário. Ela indica qual shell será iniciado quando você abre um terminal.
5. **$PWD**: O PWD (Present Working Directory) armazena o caminho do diretório atual em que você está trabalhando.
6. **$LANG** e **$LC_***: Essas variáveis definem o idioma e as configurações regionais usadas pelo sistema. Elas afetam a exibição de mensagens, datas e números em diferentes idiomas.
7. **$DISPLAY**: Usada para definir o servidor de exibição (X Window System) ao qual os aplicativos gráficos se conectam. Geralmente é configurada quando você inicia um ambiente de desktop.
8. **$EDITOR** e **$VISUAL**: Essas variáveis apontam para os editores de texto padrão usados por programas como o "crontab" ou o "git commit".
9. **$TZ**: Define o fuso horário do sistema.

Você pode visualizar o valor de uma variável de ambiente digitando `echo $NOME_DA_VARIAVEL` no terminal. Para configurar ou modificar uma variável de ambiente, você pode fazer isso no arquivo de inicialização do shell do usuário, como `.bashrc`, `.bash_profile`, `.profile`, ou no nível do sistema em `/etc/environment` ou `/etc/profile`. Lembre-se de que as alterações feitas em um terminal atual não afetarão outros terminais; você deve reiniciar a sessão ou o terminal para que as alterações tenham efeito global.

Comando env

O comando `env` quando executado sozinho, lista todas as variáveis de ambiente carregadas no terminal.

```
$ env
SHELL=/usr/bin/zsh
COLORTERM=truecolor
FUNCNEST=100
...
```

Quando executamos um programa no linux, por exemplo `python` por padrão este programa é executado na no ambiente default do carregado naquele terminal e os programas executam usando o comando `env`

```
$ env python
```

O comando env pode ser usado para sobrescrever variáveis de ambiente em uma única execução de um comando, sem persistir os valores.

```
$ env PYTHONVERBOSE=1 python
```

Na maioria dos terminais podemos simplesmente omitir o `env` já que o terminal vai executar o programa com o prefixo `env` por default.

```
$ PYTHONVERBOSE=1 python
```

Ou simplemesmente

```
$ python
```

Ou seja, qualquer linha de comando no terminal, vai ter implicitamente o ambiente carregado através do `env` (mesmo que esse seja omitido) e todas as variáveis de ambiente presentes estarão disponíveis para o programa acessar, permitindo também sobrescrever algumas das variáveis.

Podemos persistir uma variável, na sessão do shell usando o comando `export` (no bash), este comando irá adicionar a variável, ou sobrescrever caso já exista e deixar disponível para todos os processos iniciados a partir do shell.

```
$ export PYTHONVERBOSE=1
$ python
```

E caso queiramos iniciar um processo ignorando todas as variáveis:

```
$ env -i python
```

> **IMPORTANTE** uma vez que o shell é finalizado, todas as variáveis exportadas são limpas, para persistir variáveis de forma a estarem presentes em todas as execuções do shell é necessário declarar o comando export dentro de uma arquivo de inicialização do bash como o o ~/.bashrc

Para remover uma variável de ambiente usamos o comando `unset`

```
$ unset PYTHONVERBOSE
```

Lendo envvars com python

Quando qualquer processo inicia através de um shell, este processo carrega todas as variáveis de ambiente disponíveis e no Python e possível acessar esta lista de variáveis através do módulo `os` .

```
import os
print(os.environ.keys())
```

o `os.environ` é um dicionário contendo chave-valor, onde as chaves são todas upper-case e os valores são todos do tipo string. `dict[str, str]` .

Além de poder acessar através do dicionário diretamente o módulo também oferece uma função `getenv` que abstrai algumas checagens antes de acessar a variável.

```
import os
user = os.getenv("USER", default="dummy")
```

A função `getenv` expõe a facilidade de definir um valor default caso a variável não esteja definida e além disso resolve expansões do shell.

Exportando envvars com Python

Durante a execução do programa o Python vai sempre procurar as variáveis no dicionário `os.environ` portanto podemos alterar o conteúdo deste dicionário diretamente.

```
os.environ["USER"] = "root"
```

Como os módulos em Python são exportados uma única vez, se uma variável for adicionada ao `os.environ` no `__init__.py` de um programa, este valor vai estar disponível até o final da execução do programa, mas tem alguns detalhes importantes!

- O valor tem que ser sempre `str`
- A chave tem que ser upper-case
- Se por acaso for feito reload do módulo `os` as variáveis serão perdidas.

De fato através do Python não exportamos variáveis para o ambiente exterior ao processo, portanto o Python não consegue exportar variáveis de ambiente para outros processos lerem já que isso é parte do design no shell, cada processo contém um **fork** com sua memória isolada, vamos entender mais como isso funciona nos próximos tópicos.

Um exemplo prático

```
import os

def get_api_key():
    api_key = os.getenv("SECRET_API_KEY")

    if api_key is None:
        raise RuntimeError("Oops! It looks like the API key is missing.")

    return api_key

# Let's call the function
key = get_api_key()
```

Processos e Threads

Programas executados através do shell são isolados para prevenir problemas como Race Conditions e para permitir gestão de recursos computacionais.

Um processo iniciado através do shell tem como características principais o fato de terem um espaço isolado na memória, um identificador numérico único (PID) e acesso a I/O para se comunicar com outros processos via passagem de mensagens.

A capacidade de executar múltiplos processos ao mesmo tempo é a características mais importante de um sistema operacional e é o fator que revolucionou a industria de tecnologia nos anos 70.

processos

Grande parte dos programas que executamos no bash são processos bloqueantes, ou seja, uma vez executados, precisamos esperar o término da execução para continuar utilizando o shell.

exemplos:

```
ls
free
locate foo
python -c "print(42)"
```

Os comandos acima executam e ao terminar seus processos são imediatamente limpos do sistema operacional sendo que o que vemos persistido no terminal é apenas a saída (output) do processo.

No Linux podemos executar comandos em `background` usando `&`, isso faz com que o terminal não fique esperando a saida e permita a invocação (spawn) de novos processos.

```
$ sleep 10 & sleep 12 &

$ ps a -o pid,command
    PID COMMAND
 592695 /bin/bash
 603402 sleep 10
 603403 sleep 12
 603417 ps a -o pid,command
```

Repare que invocamos o processo `sleep` duas vezes e em seguida fomos capazes de executar o comando `ps` que lista os processos em execução.

Subprocessos

No contexto do Linux, um subprocesso (ou sub-processo) refere-se a um novo processo que é criado a partir de um processo existente, que é conhecido como processo pai. O processo pai pode criar subprocessos para executar tarefas específicas ou executar programas independentes. Os subprocessos herdam certas características e recursos do processo pai, mas eles também podem ter seus próprios atributos, incluindo identificadores de processo (PID) exclusivos.

Em Python podemos usar `fork` e `execl` da biblioteca `os` para criar um subprocesso.

```
import os

def main():
    pid = os.fork()

    if pid == 0:
        # Este é o processo filho
        print("Isso é o processo filho com PID:", os.getpid())

        # Executa um novo programa no processo filho
        try:
            os.execl("/bin/ls", "ls", "-l")
        except FileNotFoundError:
            print("Falha ao executar o novo programa")
    else:
        # Este é o processo Pai
        print("Processo pai com PID:", os.getpid())
        print("e seu filho tem PID:", pid)

if __name__ == "__main__":
    main()
```

Ao executar `os.fork` criamos uma cópia do processo atual, em memória, em tempo de execução, ou seja, o mesmo código que está em execução, será novamente executado no processo filho.

Na primeira execução o `pid = os.fork()` será o PID do processo principal, e assim que o processo for copiado e executado novamente este valor será `0` indicando que já se encontra dentro de um fork e então o programa toma outro caminho de execução.

Além disso usamos o `os.execl` para abrir mais um subprocesso, o comando `ls`

Estou mostrando isso apenas em caráter didático, já que raramente usaremos fork e execl diretamente pois existem bibliotecas como a subprocess e a concurrent que abstraem essas funcionalidades.

Threads

Dentro de cada processo é possível iniciar uma outra unidade de execução chamada de `thread` as threads são iniciadas a partir de processos e elas diferem dos sub-processos pois compartilham o mesmo espaço de memória e os mesmos descritores de arquivos.

A vantagem de usar threads é a possibilidade de de concorrer pelo uso de CPU enquanto uma outra thread está em estado de **espera** que pode ser espera pelo ciclo do relógio como no exemplo com `sleep 10` ou pode ser uma thread que está a espera do carregamento de algum dado como a leitura de um arquivo ou chamada a uma interface externa.

Essas threads que geralmente precisam **esperar** nós chamamos de I/O Bound, pois elas dependem do estado de input-output para continuar executando e durante esta espera não utilizam a CPU, portando é mais interessante aproveitarmos este tempo de espera para executar outras threads.

`vim 3threads.py`

```
import concurrent.futures
import time

# Define a function that each thread will run
def thread_function(thread_id):
    time.sleep(10)
    print(f"Thread {thread_id} finished")

# Create a ThreadPoolExecutor with 3 threads
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # Submit the tasks
    futures = [executor.submit(thread_function, i) for i in range(3)]

    # Wait for all tasks to complete
    concurrent.futures.wait(futures)

print("All threads have finished")
```

```
$ python 3threads.py &

$ ps a -T -o pid,tid,command
    PID     TID COMMAND
 592695  592695 /bin/bash
 827909  827909 python 3threads.py         # processo principal
 827909  827915 python 3threads.py         # thread 1
 827909  827916 python 3threads.py         # thread 2
 827909  827917 python 3threads.py         # thread 3
 827946  827946 ps a -T -o pid,tid,command
```

O que pode dar errado?

Usar threads é em alguns casos mais eficiente do que processos pois as threads compartilham o mesmo espaço de memória e ao mesmo tempo isso induz a um problema bastante comum que é chamado de race condition.

Usar threads é recomendado quando precisamos implementar concorrência em um único programa, para aproveitar os ciclos de **espera** de I/O para computar outras partes do código, mas exige um certo cuidado ou abordagens especializadas para evitar race conditions, a desvantagem de usar threads em Python (até a versão 3.11) é o fato de threads executarem em apenas um núcleo da CPU.

Processos são excelentes para tarefas CPU Bound, ou seja, onde a principal funcionalidade seja usar a CPU, e também são melhores para aproveitar CPUs multicore, já que o próprio sistema operacional consegue distribuir a carga de processos entre as CPUs disponiveis, como cada processo tem seu espaço reservado de memória o desafio é a troca de informações entre os processos é a ai que entra a abordagem do IPC.

IPC

`Inter Process Communication` é o nome genérico dado a técnicas de compartilhamento de dados entre processos e existem algumas abordagens, entre as principais estão:

- Arquivos (Filesystem)
- Pipes (Unix stdin/out)
- Sockets (Um protocolo de comunicação sobre o filesystem)

Filesystem

`Linux Common Filesystem Structure`

```
/
|-- bin/           # Essential binaries
|-- boot/          # Boot loader and kernel
|-- dev/           # Device files
|-- etc/           # System configuration
|-- home/          # User home directories
|   |-- user1/
|   |-- user2/
|   `-- ...
|-- lib/           # Shared libraries
|-- media/         # Removable media (e.g., USB drives)
|-- mnt/           # Mount points for temporary filesystems
|-- opt/           # Optional software packages
|-- proc/          # Process information
|-- root/          # Root user's home directory
|-- run/           # Run-time variable data
|-- sbin/          # System binaries
|-- srv/           # Service data
|-- sys/           # Kernel-related files
|-- tmp/           # Temporary files
|-- usr/           # User binaries and data
|   |-- bin/
|   |-- lib/
|   `-- ...
|-- var/           # Variable data (log files, spool, etc.)
```

Permissões

cada arquivo e diretório tem suas próprias configurações de permissões. Essas permissões basicamente determinam quem pode fazer o quê com o arquivo. Existem três tipos principais de permissões: leitura (read), escrita (write) e execução (execute).

1. **Leitura (Read):**

    - Se um arquivo tem permissão de leitura, isso significa que você pode visualizar o conteúdo do arquivo. Se é um diretório, você pode ver quais arquivos estão dentro dele.

2. **Escrita (Write):**
    - Se um arquivo tem permissão de escrita, você pode modificar seu conteúdo. Em um diretório, permissão de escrita permite adicionar, remover ou renomear arquivos.

3. **Execução (Execute):**
    - Para arquivos, a permissão de execução permite que você o execute como um programa. Para diretórios, a permissão de execução permite acessar o conteúdo do diretório.

Agora, como isso se parece quando você olha para as permissões?

- As permissões são divididas em três grupos: usuário dono (owner), grupo e outros (others).
- Cada grupo tem suas próprias configurações de leitura, escrita e execução.

Por exemplo, se você vê algo como "rw-r--r--":

- O dono (usuário) tem permissão de leitura (r) e escrita (w), mas não de execução (-).
- O grupo tem permissão de leitura, mas não de escrita ou execução.
- Outros (quem não é o dono nem está no grupo) também têm permissão de leitura, mas não de escrita ou execução.

Você pode mudar essas permissões usando o comando `chmod` e ver as permissões com o comando `ls -l`.

As permissões podem ser representadas de duas maneiras: a textual (rw-) que acabamos de discutir e a numérica. Na representação numérica, cada tipo de permissão tem um valor associado:

- **Leitura (Read):** Representada por 4.
- **Escrita (Write):** Representada por 2.
- **Execução (Execute):** Representada por 1.

Agora, podemos usar esses valores para entender e configurar permissões mais facilmente. Vamos pegar o exemplo "rw-r--r--" novamente:

- O dono (usuário) tem permissão de leitura (4) e escrita (2), totalizando 6 (4 + 2). Adicionando permissão de execução (1), ficaria 7.
- O grupo tem permissão de leitura (4), totalizando 4 (4 + 0 + 0).
- Outros (quem não é o dono nem está no grupo) têm permissão de leitura (4), totalizando 4 (4 + 0 + 0).

Então, em representação numérica, "rw-r--r--" seria "764".

Agora, se quisermos adicionar permissão de execução ao grupo e outros, podemos modificar para "rwxr-xr--", que em representação numérica seria "755" (7 para o dono, 5 para grupo e 5 para outros).

Para aplicar essas alterações, você pode usar o comando `chmod` assim:

```
$ chmod 755 nome_do_arquivo
```

Esse comando concede permissão total ao dono (7), permissão de leitura e execução ao grupo (5) e permissão de leitura e execução a outros (5).

Se você quiser remover a permissão de execução de um arquivo ou diretório, você pode usar o comando `chmod` com o argumento "-x". Vamos considerar o exemplo anterior "rwxr-xr--", que em representação numérica seria "755".

Se quisermos remover a permissão de execução do grupo e outros, transformando isso em "rw-r--r--" (ou "644" em representação numérica), podemos usar o comando a seguir:

bash

`chmod go-x nome_do_arquivo`

Neste comando:

- "go" refere-se ao grupo (group) e outros (others).
- "-x" remove a permissão de execução.

Dessa forma, o comando acima removerá a permissão de execução do grupo e outros, mantendo apenas as permissões de leitura e escrita.

Lembre-se de que o dono (usuário) não é afetado pelo "go", então as permissões dele permanecerão inalteradas.

FDs

Como em processos a memória é isolada a melhor forma de trocar mensagem entre eles é através do sistema de arquivos.

Para estabelecer a comunicação entre processos estabelecemos um canal de comunicação que é baseado em um arquivo e chamamos este arquivo de `fd` (File Descriptor)

Todo sistema UNIX já vem com 3 desses canais pré estabelecidos e são chamados de `standard streams` representados pelos file descriptors.

- fd 0: Entrada Default (standard input) - STDIN
- fd 1: Saída Default (standard output) - STDOUT
- fd 2: Error Default (standard error) - STDERR

cada processo assim que iniciado tem seus `fds` criados no diretório `/proc/<PID>/fd`

STDIN

Este descriptor será lido pelo processo para receber entrada, seja esta entrada do usuário digitando em um emulador de terminal ou de um programa que esteja direcionando stream de texto para este descriptor, na prática o funcionamento pode ser verificado ao executar por exemplo o comando `base64` no Linux.

```
$ base64   # O programa fica esperando entrada

# Ao digitar por exemplo 'python', seguido de ENTER
# o emulador de terminal escreve 'python\n' no /proc/<PID>/fd/0
# para finalizar a entrada e fechar o file descriptor
# precisamos enviar um EOF com Ctrl+D

cHl0aG9uCg==
```

Um forma de ver como isso funciona interativamente é abrir 2 terminais, lado a lado e usar o terminal 1 para escrever no terminal 2.

1. abra 2 instancias (ou 2 splits) do terminal
2. no terminal 2 execute `pidof bash` e copie o primeiro numero
3. no terminal 1 execute `echo "hello" > /proc/<PID>/fd/0`
    1. substitua `<PID>` pelo número copiado.
4. O texto "hello" será direcionado do terminal 1 para o terminal 2

STDOUT

No comando anterior o resultado `cHl0aG9uCg==` foi impresso no mesmo terminal onde executamos o `base64` e isso aconteceu porque o terminal por padrão imprime imediatamente toda informação que é escritas no descriptor `1` , um outro exemplo é o comando `echo`

```
$ echo Python
Python
```

O comando `echo` escreve o texto `Python` no `/proc/<PID>/1` que está por padrão conectado a sessão de terminal que por sua vez o imprime na tela.

É importante ressaltar que apesar de superficialmente o fluxo parecer algo como:

- comando echo processa o texto "Python"
- comando echo imprime o texto no terminal

Na verdade é:

- Comando echo processa o texto "Python"
    - (entrada via argv/argumento de linha de comando)
- Comando echo escreve "Python" no `/proc/<PID>/fd/1`
- O emulador de terminal lê e consome o texto imediatamente, pois o terminal normalmente atua como um "monitor" de todos os processos iniciados a partir dele.

STDERR

O **fd** presente em `/proc/<PID>/2` é onde as aplicação estão supostas a escrever suas mensagens de erro, para evitar que mensagens de erro se confundam com output normal da aplicação.

Um exemplo simples seria:

```
$ ls /pasta/que/nao/existe/
"/pasta/que/nao/existe/": No such file or directory
```

A mensagem de erro é escrita no terminal apenas por conveniência já que a maioria dos terminais também monitoram o fd 2 e redirecionam automaticamente para o STDOUT.

Stream Redirection

Em alguns casos precisamos redirecionar essas informações para outro `fd` ou mesmo para um outro arquivo que não seja gerenciado pelos standard streams.

No Linux podemos usar o operador `>` para STDOUT e STDERR e o operador `<` para o STDIN.

Redirecionando o STDOUT para um arquivo

```
echo Python 1> arquivo.txt
```

E pode usar o comando `cat` para ler o conteúdo do arquivo.

```
$ cat arquivo.txt
Python
```

> Neste ponto você já sabe que o `cat` carrega o conteúdo do arquivo e envia para o STDOUT!

Como o fd 1 é o mais comum, ele é considerado o default, portanto podemos omitir o `1` e fazer simplesmente:

```
$ echo Python > arquivo.txt
```

Redirecionando erros para um arquivo.

Agora basta seguir a mesma lógica e usar o stream de erros, o fd 2

```
$ ls /nao/existe 2> erro.txt
```

Repare que o erro não será impresso no terminal pois o terminal não está monitorando o arquivo que acabamos de criar, portando precisamos usar o `cat` para visualizar a mensagem de erro.

```
$cat erro.txt
"/nao/existe": No such file or directory (os error 2)
```

Multiplos redirects

Podemos compor redirecionamentos mais complexos combinando essa estratégia

Enviando saidas para um arquivo e erros para outro:

```
$ echo "Python" > arquivo.txt 2> erro.txt
```

E também podemos redirecionar entre os streams usando `&`

```
$ echo "Python" > arquivo.txt 2>&1
```

> O comando acima está direcionando o fd 1, STDOUT para o arquivo.txt e o fd 2 STDERR para o fd 1, que por sua vez está enviando para o arquivo, desta forma as 2 mensagens vão para o mesmo arquivo.

Redirecionando STDIN

O stream de entrada também pode ser redirecionado da mesma forma, porem usando o operador `<` sem a necessidade de especificar o número `0` considerando que existe apenas um stream de entrada.

```
$ echo "Python" > arquivo.txt
$ base64 < arquivo.txt
UHl0aG9uCg==
```

Ignorando streams

Em alguns casos podemos querer silenciar streams de algum programa, seja pelo fato do programa emitir output insignificante que polui a sessão de terminal ou para silenciar erros.

No Linux podemos usar o descriptor `/dev/null` que é um caminho especial usado para descartar dados, qualquer arquivo movido para este caminho ou stream escrito neste caminho é automaticamente descartado pelo sistema.

```
$ ls /nao/existe > /dev/null 2>&1
```

O comando acima será executado e qualquer saída ou erros serão ignorados/descartados.

Pipes

Vamos recapitular o exemplo que usamos anteriormente

```
$ echo "Python" > arquivo.txt
$ base64 < arquivo.txt > resultado.txt
UHl0aG9uCg==
```

- O programa `echo` envia dados para o STDOUT que é redirecionado para o arquivo.txt
- A informação contida no arquivo.txt é agora redirecionada ao STDIN do comando `base64`
- O comando `base64` envia dados para o STDOUT que é redirecionado para o resultado.txt

Este padrão chamamos de Pipeline e é uma das abordagens mais poderosas da filosofia Unix.

Este padrão é tão comum que foi estabelecido um caractere `|` que ficou conhecido como "pipe" que faz exatamente o que fizemos no código anterior.

```
$echo "Python" | base64
UHl0aG9uCg==
```

O operador `|` é um canal de comunicação FIFO (firt-in, first-out), é um canal unidirecional que captura os dados da saida esquerda e escreve na entrada direita.

Quando usamos o `|` ele cria um par de `fd`, um para saida e outro para entrada, da mesma forma que fizemos com arquivo.txt e resultado.txt, porém ele faz isso diretamente na memória e este arquivo só existe durante a execução da operação que uriliza o `|`

Podemos usar quantos pipes forem necessários, por exemplo, podemos listar todos os processos em execução com `ps`, filtrar apenas pelos que contém `bash` com `grep` e contar a quantidade de resultados com `wc`.

```
$ ps aux | grep bash | wc -l
3
```

Desta forma podemos compor uma linha de comando usando vários programas para obter um único resultado, cada programa tem uma responsabilidade única e todos entendem o mesmo protocolo de comunicação baseado nos File Descriptors.

Lendo stdin com Python

A biblioteca `sys` fornece a interface para leitura do STDIN, ou seja, `fd 0`

```
# triangle_area_using_heron.py
import sys

def heron(a, b, c):
    perimeter = a + b + c
    s = perimeter / 2
    area = (s * (s - a) * (s - b) * (s - c)) ** 0.5
    return area


def triangle_area_using_heron():
    sides = [float(x) for x in sys.stdin.readline().split()]

    if len(sides) != 3:
        print("Error: Please provide exactly three sides of the triangle.")
        return

    area = heron(*sides)
    print(f"Area of the triangle: {area:.2f}")

if __name__ == "__main__":
    triangle_area_using_heron()
```

```
$ echo "10 10 10" | python triangle_area_using_heron.py
Area of the triangle: 43.30
```

Lendo chuncks do stdin

```
# 3lines.py
result = 0
while True:
    try:
        result += sum(map(int, [input(), input(), input()]))
        print(result)
    except EOFError:
        break


print(result)
```

```
$ echo "1\n2\n3\n4\n5\n6" | python 3lines.py
21
```

Pipes nomeados

De forma similar ao uso de "|" anteriormente, que cria um descriptor temporário anonimo, podemos criar nossos próprios descriptors FIFO para trocar mensagens

entre processos.

A vantagem de fazer ao invés de usar um arquivo de texto comum é que contamos com gerenciamento do O.S e não corremos o risco de estourar o uso de memória ou armazenamento.

No Linux podemos usar o comando `mkfifo` para criar um PIPE nomeado.

```
$ mkfifo /tmp/minhafila
```

Em outro terminal podemos usar um processo para escrever na fila.

```
$ echo "Hello" > /tmp/minhafila
```

E repare que o processo do programa `echo` fica bloqueado até que um outro programa efetue a leitura sincronamente.

```
$ cat /tmp/minhafila
```

Também podemos usar `tail -f /tmp/minhafila`

Neste exemplo o único cuidado que precisamos ter é de excluir o descriptor manualmente, porém o arquivo é tão pequeno que muitas vezes pode ser esquecido sem problemas.

No Python podemos usar novamente a biblioteca `os` para criar filas nomeadas.

```
import os

os.mkfifo("/tmp/minhafilapython", 0o600)
```

Podemos então ler e escrever diretamente no arquivo usando `open` do Python.

No próximo capítulo começaremos a falar sobre protocolos de comunicação com Python :)