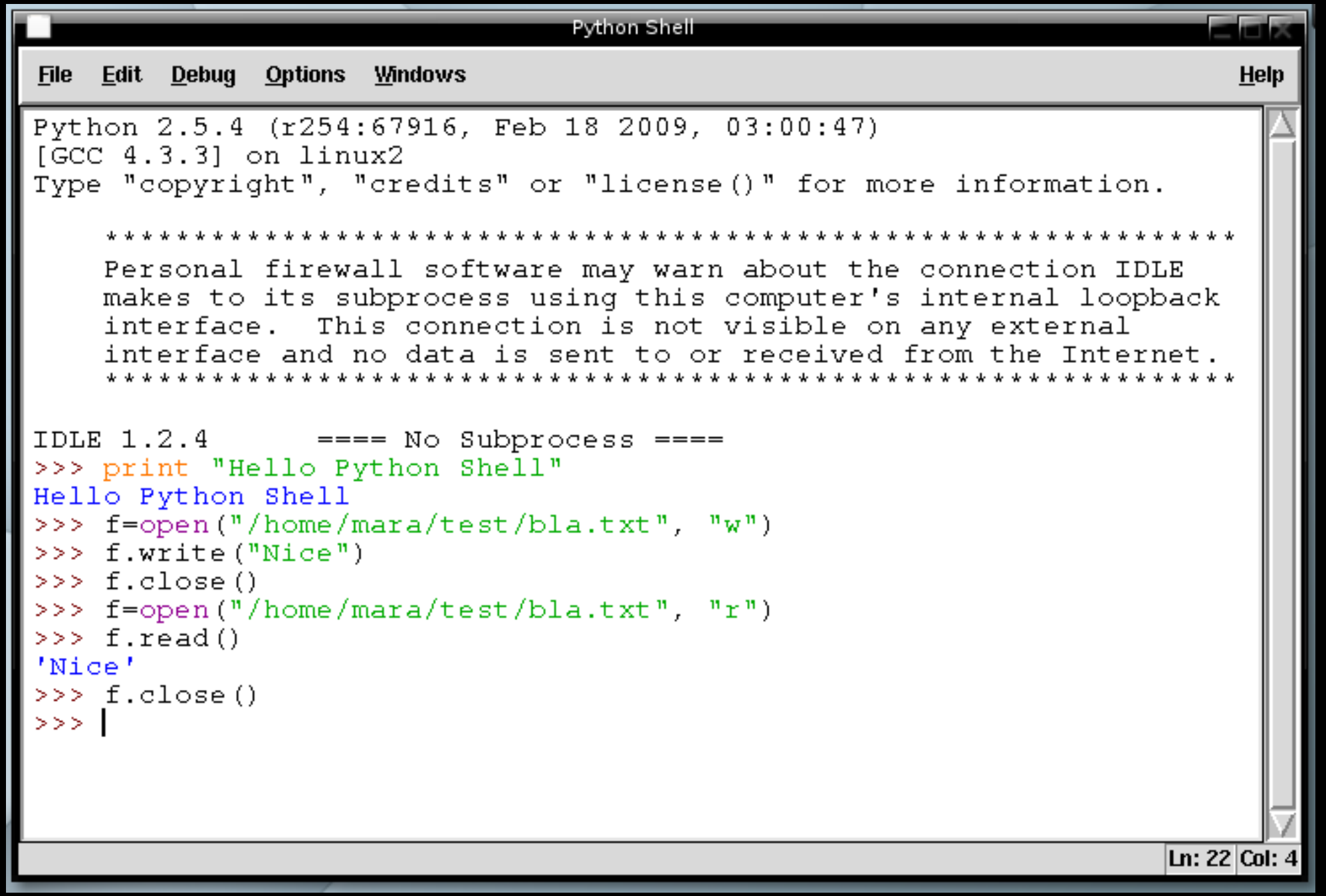
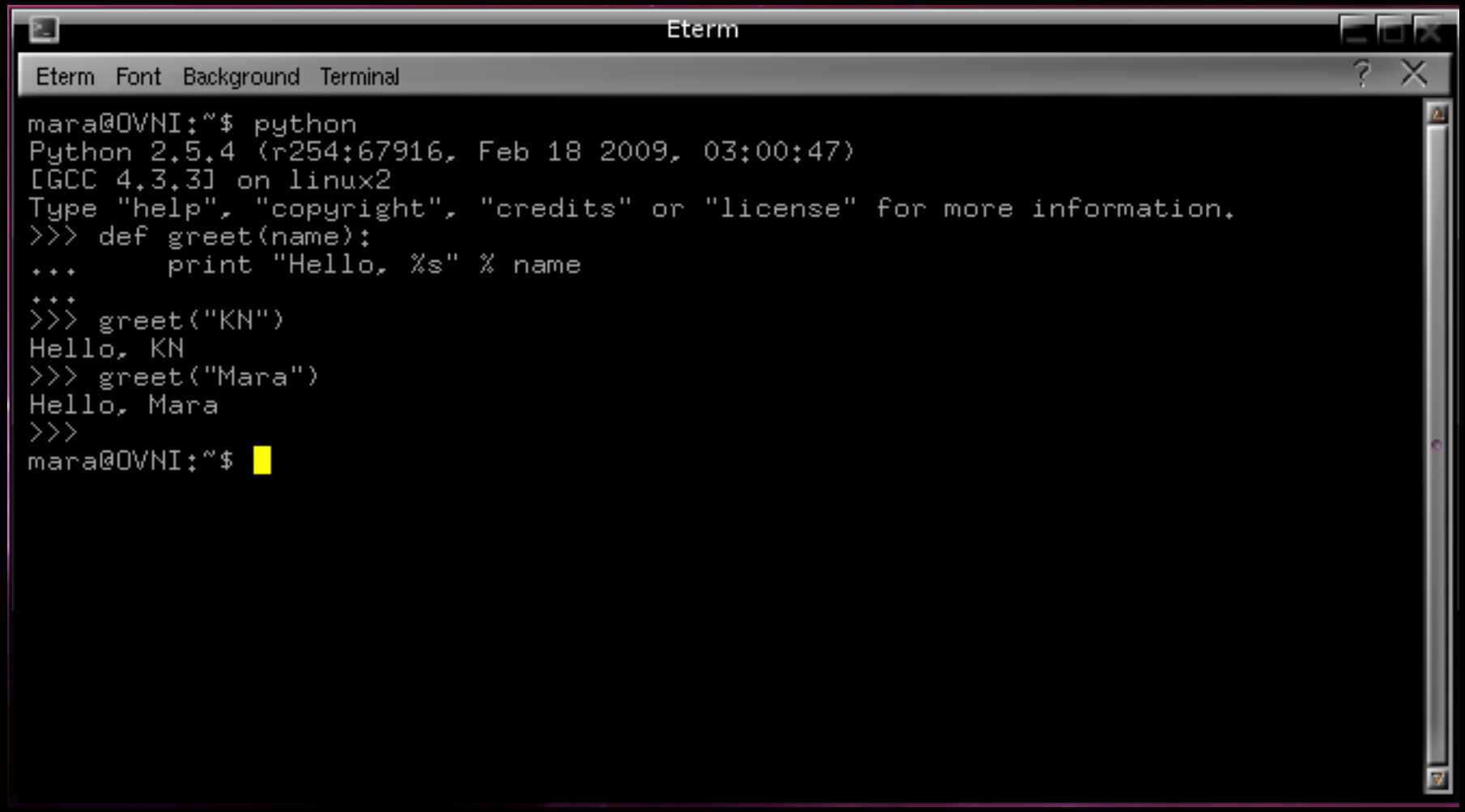
Снимка: Мила Зинкова

Shell

Shell е програма, която осигурява потребителски интерфейс. Най-често под shell се разбира shell на операционна система (т.е програма, която осигурява потребителски достъп до услугите на системата и в частност до ядрото). Но такива програми има например и за много езици за програмиране (особено интерпретираните и скриптовите) и те също се наричат shell-ове по термина от ОС.

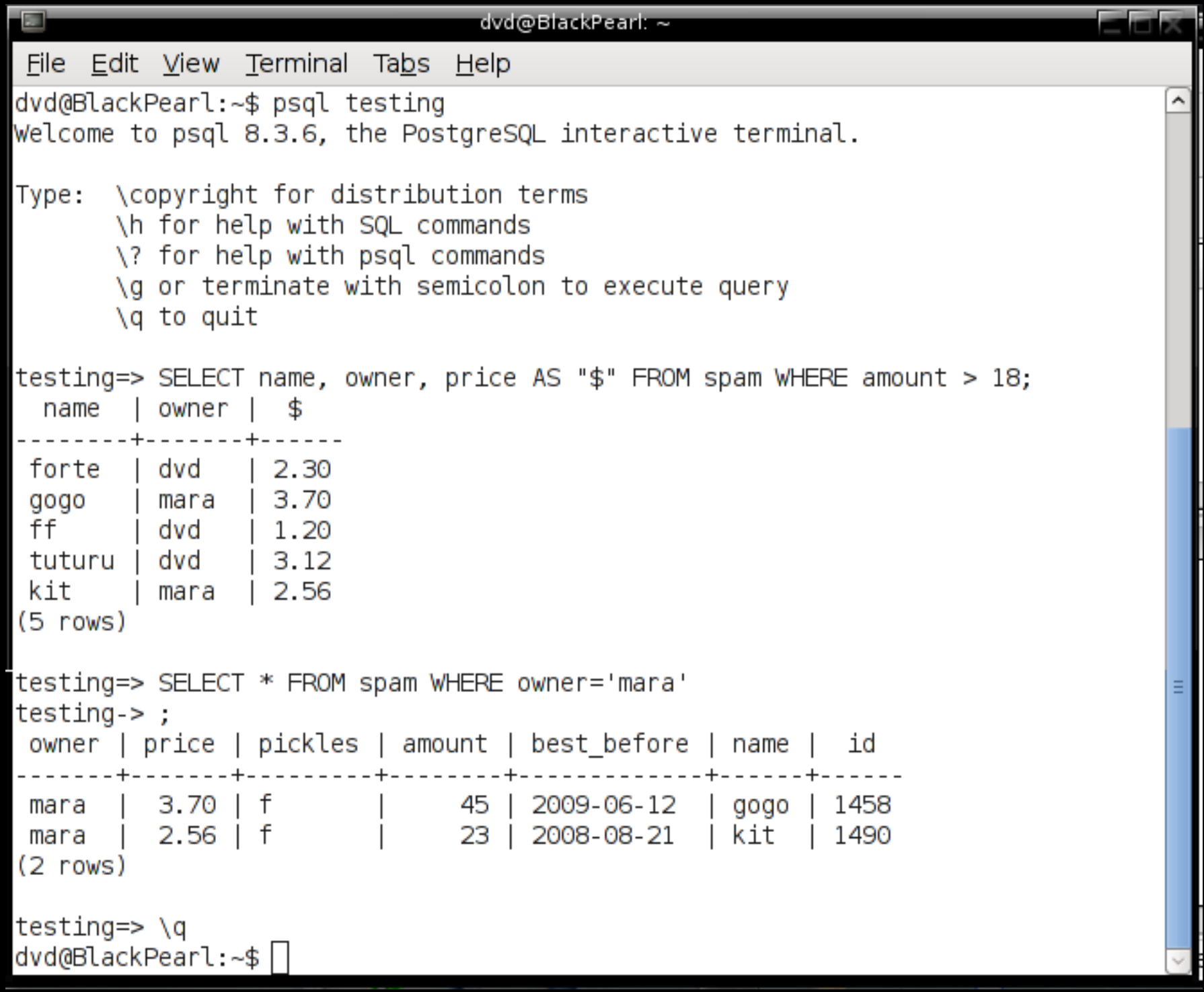


Това е IDLE – IDE за Python. Освен текстов редактор, той има и този графичен shell, с който може да се задават кратки и по-дълги езикови конструкции и те да бъдат изпълнени веднага.

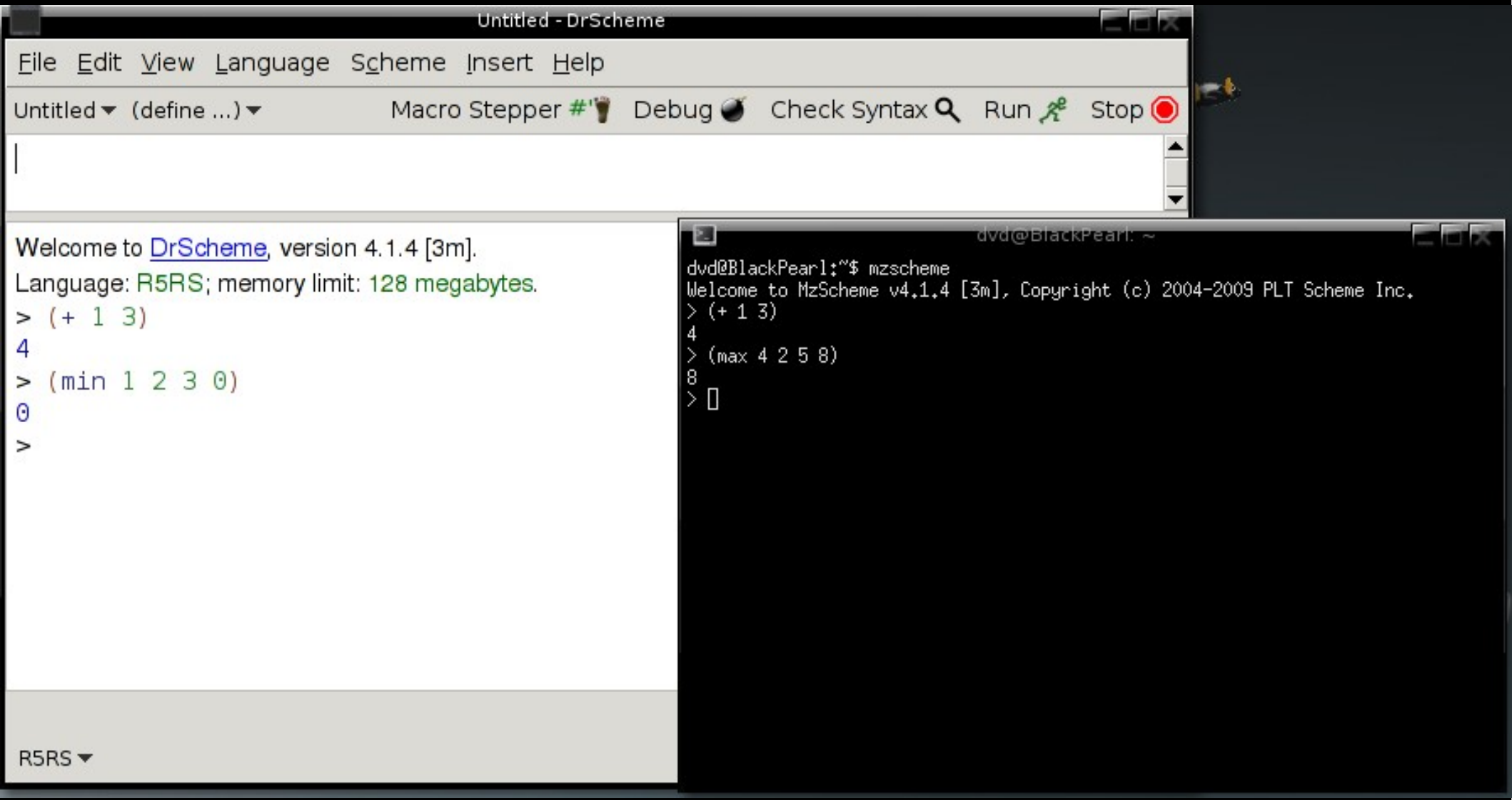


Това е интерактивният shell, който предлага
Python-интерпретаторът през конзолата

aptsh – интерактивен shell към програмата apt, с която
се инсталира софтуер при Debian-базираните ОС



Shell-ът на PostgreSQL в конзола



Това вече трябва да е познато ;)

OS Shells

Има два основни вида ОС shell-ове – командни и графични. Командните предоставят Command-Line Interface (CLI), а графичните – Graphical User Interface (GUI). И двата вида имат за основна цел стартиране на програми (процеси), но и много други функционалности (като например разглеждане на файловата система, настройване на ОС).

OS GUI Shells

- X Window System
  - independent X Window Managers: Blackbox, Fluxbox
  - desktop shells: Enlightenment DR17 (E 17)
  - desktop environments: KDE, GNOME, XFCE, LXDE and others
- Mac OS X – The Finder
- Windows NT – Windows Shell (Windows Explorer)

Какви са всички тези неща?

X Window System
X Window Manager
Desktop environment
KDE, GNOME, XFCE, LXDE,
Enlightenment, Blackbox, Fluxbox
The Finder
Windows Shell

OS CLI Shells

- DOS & MS Windows – COMMAND.COM, cmd.exe, 4DOS, 4NT, Windows Power Shell, Windows Recovery Console
- Unix (Unix-like) OS:
  - Thompson Shell
  - Bourne Shell
    - Bourne-Again Shell
    - Korn Shell
    - Almquist Shell
      - Debian Almquist Shell
  - C Shell
    - TENEX C Shell
  - rc, Z Shell, Es Shell and others

Thompson Shell

- Първият shell на първия Unix (1971 година), написан от Ken Thompson
- Изключително прост и има функционалност само на команден интерпретатор, не поддържа scripting (единствената възможност за контролиране на хода на командите е била чрез `goto`)
- Реализира за пръв път идеята за пренсочване на стандартния вход и стандартния изход в една неделима команда. Малко по-късно реализира и синтаксиса за pipe.
- Първоначалният синтаксис за pipe бил:
  `command1 >command2>`
  Макар и по-добре описващ действието на конвейера, този синтаксис предизвиквал объркванията със синтаксиса на пренасочванията и затова бил заменен от:
  `command1 ^ command2` и
  `command1 | command2` (което се е запазило и до днес)
- Заради недостатъчната си функционалност през 1979 година е заменен от Bourne Shell в Unix Version 7 и от C Shell в 2BSD

Bourne Shell

- Още познат просто като sh. Всъщност sh се е наричал изпълнимият файл на Thompson Shell и заради съвместимост и изпълнимият файл на Bourne Shell се нарича така. Създател: Stephen R. Bourne (AT&T Bell Laboratories)
- Освен интерактивен команден интерпретатор, предоставя възможностите на scripting език
- Първият, който реализира конвенцията файлов дескриптор 2 да е запазен за стандартен изход за грешки
- Въвежда заместването на изход на команда (`` `command` ``), команди-оператори `for` и `case`, т.нар. Here Document (низ, към който се пренасочва стандартният вход) и променливи на обкръжението
- Макар и да не е документирано, поддържа опцията за pipe с `^`
- Превръща се в основата за CLI към Unix и Unix-like ОС

C Shell

- Още познат като csh. Създател: Bill Joy, предназначена за BSD Unix.
- Произлиза от Thompson Shell, синтаксиса има за цел да моделира езика C
- Въвежда job control (възможност дадено *задание (job)* да бъде временно прекъснато и подновено по-късно или да бъде преместено във фонов режим). Тази възможност по-късно е добавена и към Bourne Shell.
- Въвежда history substitution (възможността предходни команди да бъдат пуснати отново и да бъдат редактирани). Например `!!` ще изпълни отново предходната команда.
- Въвежда ~ expansion, alias-и, масиви, математически операции
- Някои от големите разлики с Bourne Shell са използване на `( )` скоби в синтаксиса на командата `test` в условен оператор (а не `[ ]`), присвояване на стойност на променлива `set a = b` вместо `a=b`, `endif` вместо `fi`.
- Въпреки нововъведенията, получава критики за scripting възможностите си
- Наследява се от TENEX C Shell (tcsh), който добавя (към основния код) filename completion и др.

Korn Shell

- Или още ksh. Създател: David Korn (AT&T Bell Laboratories), 1982 година.
- Обратно съвместим с Bourne Shell, реализира и много от възможностите на C Shell
- Версията от 1993 година въвежда асоциативни масиви (maps) и вградена аритметика с плаваща запетая
- Предоставя възможност за редактиране на предходна команда чрез преместване на курсора до нея и натискане на Enter
- Има за цел да спазва POSIX Shell Language Standard
- Удобен за програмиране
- До 2000 година е запазена собственост на AT&T, след това кодът става отворен
- pdksh е една от свободните алтернативи, създадена през '90-те години

Bourne-Again Shell

- Накратко bash. Създател е Brian Fox (1987 година), но от 1990 година за него отговаря Chet Ramey. Разработван е за проекта GNU.
- Обратно съвместим с Bourne Shell, реализира и полезните въведения от csh и ksh. Удовлетворява стандарта POSIX с някои изключения (но може да се конфигурира за стартиране в пълна съвместимост със стандарта).
- Подобрения на функционалността като интерактивен команден интерпретатор включват възможност за редактиране на предходните команди (чрез горна и долна стрелка можете да се движите по предходните команди), възможност за настройване на първичния и вторичния prompt, aliases, job control
- Подобренията във функционалността като език за програмиране включват допълнения във variable expansion, допълнителен синтаксис за дефиниране на функции, индексирани масиви без ограничение в размера, аритметика с цели числа при основа от 2 до 64, множество променливи и опции за по-добър контрол на поведението на shell-а.

The GNU Project

- Целта му е да създаде цялостна, съвместима с Unix и софтуера му, свободна ОС. GNU е рекурсивно съкращение от “GNU Is Not Unix”.
- Началото му е през 1983 година, основател е Richard Stallman
- До 1992 година всички главни компоненети на GNU ОС са готови. Наново са написани основните програми в Unix, включително и командният интерпретатор (bash), като са добавени много допълнителни функционалности. Създаден е лицензът за свободен софтуер GPL. Това, което липсва, е ядрото на ОС.
- През 1991 година излизат първите публични версии на ядрото Linux. Създателят му Linus Torvalds го лицензира под GPL. С това се запълва празнината за напълно функционална свободна ОС. Скоро след това излизат първите GNU/Linux дистрибуции.
- Разработката на ядрото на GNU – GNU Hurd – все още продължава. Има и няколко проекта за ОС, използващи това ядро, като най-известният от тях е Debian GNU/Hurd.
- Други GNU варианти (т.е. ОС, които ползват софтуерът, но не и ядрото на GNU) освен GNU/Linux дистибуциите, са Debian GNU/kFreeBSD, Debian GNU/NetBSD, Gentoo/Alt, Nexenta OS.

Almquist Shell

- Още познат като A Shell и ash. Лек и бърз “клонинг” на Bourne Shell, удовлетворява стандарта POSIX. Създател: Kenneth Almquist, целта е била да замени Bourne Shell в по-новите версии на BSD.
- Не поддържа различните екстри, които има в bash, zsh, tcsh по идеологически причини. Няма редактиране на предходни команди и history.

Debian Almquist Shell

- Накратко dash. Директен наследник на NetBSD версията на ash, пригодена към Linux от Herbert Xu през 1997 година (2002 година е преименуван на dash).
- В Debian и базираните на него ОС замества ash

Default Shells

OS	user shell	root shell	/bin/sh target
GNU/Linux (most)	bash	bash	bash
Ubuntu	bash	bash	dash
Mac OS 10.3 and later	bash	bash	bash
Mac OS 10.2 and earlier	tcsh	tcsh	tcsh
FreeBSD	ash	tcsh	ash
NetBSD 4.0	ash	ash	ash
OpenBSD	pdksh	pdksh	pdksh
OpenSolaris	bash	bash	bash

! Възможни са неточности

Вътрешни и външни команди

Shell-овете изпълняват различни команди. Къде се намира кодът на тези команди?

Unix shell-овете не са тясно интегрирани с дадена ОС, всеки потребител може да избира своя shell според нуждите и предпочитанията си. Но нужно ли е всеки отделен shell да имплементира за себе си всяка една команда? Разбира се, че не.

Програмата shell обикновено съдържа в кода си малко количество основни (и зависими от нея) команди. Това са т.нар. **вътрешни** (вградени, built-in) команди. Примери за вътрешни команди са:

`:` `.` `alias` `cd` `export` `read` `exit`,

повечето управляващи команди (оператори) и др.

Останалите команди са написани самостоятелно, имат си свой изпълним файл, който shell-ът пуска. Те се наричат **външни** команди. Не зависят от shell-а, в който ги изпълнявате.

Изпълнимият файл на външна команда можете да намерите с командата:

```
which command
```

Самият which представлява shell script и е външна команда. Ако той не намери местоположението на изпълнимия файл на дадена команда, това означава или, че тази команда се намира в директория, която не е част от пътя ви за търсене, или не е инсталирана на системата, или е вътрешна.

Shell Script

Съвременните shell-ове предоставят функционалност на scripting езици. Такива езици не са предназначени за писане на програми “от нищото”, а по-скоро за снаждане на различни компоненти в едно цяло. При тях се предполага, че съществува някаква колекция от компоненти (написани на същия или на други езици).

В нашия случай такива компоненти са всички изпълними файлове на програми. Нашата задача е да “режисираме” “сценария”, по който да вървят тези програми, за да постигнем търсения резултат.

Както всеки друг scripting език, и тук ние описваме нашия “сценарий” чрез средствата на езика в обикновен текстов файл, който после програмата-shell интерпретира. Т.е. един shell script не представлява нищо повече от изредени една след друга команди в текстов файл.

Защо са нужни такива shell script-ове? Поради същите причини, поради които се пишат програми на кой да е език – за да може да (пре)използваме бързо и лесно кода, а не всеки път, когато ни потрябва някакъв резултат, да пишем ръчно команда след команда.

Shell script-овете още се наричат командни процедури.

Interactive and Non-Interactive Shell

Интерактивен процес-shell чете команди, въведени от потребителя в терминал (или терминален емулатор). Потребителят може да комуникира със shell-а. Някои от свойствата на интерактивния процес-shell са, че при активацията си той чете специални конфигурационни startup файлове, изписва т.нар. prompt (покана за въвеждане на команда) и има активиран job control.

Процес-shell, който изпълнява shell script, е **неинтерактивен** процес-shell, потребителят не може да се намесва в хода на събитията. Разбира се, това не пречи на самия shell script да бъде интерактивен, т.е да поддържа комуникация с потребителя (например да чете потребителски вход).

Изпълнение на команда от shell-а

Първи вариант:

- Проверка дали командата е вътрешна. Ако е така – изпълнява я (като извикване на функция в изходния код)
- Ако командата не е вътрешна, по името ѝ намира файлът ѝ.
- Ако този файл съдържа командна процедура, процесът-shell пренасочва стандартния си вход към файла и чете ред по ред командите и ги изпълнява
- Ако файлът на командата е изпълним файл, създава нов процес, в който се изпълнява. Ако режимът е фонов, процесът-shell изчаква края на изпълнението.

Втори вариант:

- Проверка дали командата е вътрешна. Ако е така – изпълнява я (като извикване на функция в изходния код)
- Ако командата не е вътрешна, по името ѝ намира файлът ѝ.
- Ако този файл съдържа командна процедура, процесът-shell създава свой дъщерен процес-shell (sub-shell), чийто стандартен вход се пренасочва към файла. Sub-shell-ът чете ред по ред командите и ги изпълнява.
- Ако файлът на командата е изпълним файл, създава нов процес, в който се изпълнява
- И в **двата** случая, ако режимът е фонов, процесът-shell изчаква края на изпълнението.

Стартиране на shell script

Изпълнение по 1вия вариант:

```
source myscript
```

(Bourne Shell)

```
. myscript
```

(по-новите shell-ове)

Изпълнение по 2рия вариант:

```
*sh myscript
```

(според shell-ът, с който искаме да го изпълним)

```
myscript
```

(ако script-ът има shebang)

- *myscript трябва или да се намира в PATH на потребителя, или да се зададе пътя (относителен, пълен) до него.*
- *изпълнението чрез shebang-а на script-а се извиква все едно script-ът е executable файл. Т.е. отново той трябва да се намира в PATH или да се зададе пътя до него, като ако е в текущата директория се извиква с*

  ```
  ./myscript
  ```

  ЗАДЪЛЖИТЕЛНО в този случай е извикващият script-а да има право за неговото изпълнение (право x).
  Можете да зададете това право на вашия script:

  ```
  chmod a+x myscript
  ```

  По този начин могат да се пускат script-тове на почти всеки scripting език.

Shebang

- нарича се още hashbang, hashpling или pound bang
- означава комбинацията “`#!`” в началото на файла (първите 2 байта в него). Ако първият ред започва с тях, той се нарича shebang line.
- използва се в script-ове
- след “`#!`” на реда стои пълният път до интерпретатора, който изпълнява програмата. Примери:

```
#! /bin/sh
#! /bin/bash
#! /usr/bin/python
```

- накрая може да има и опции към интерпретатора:

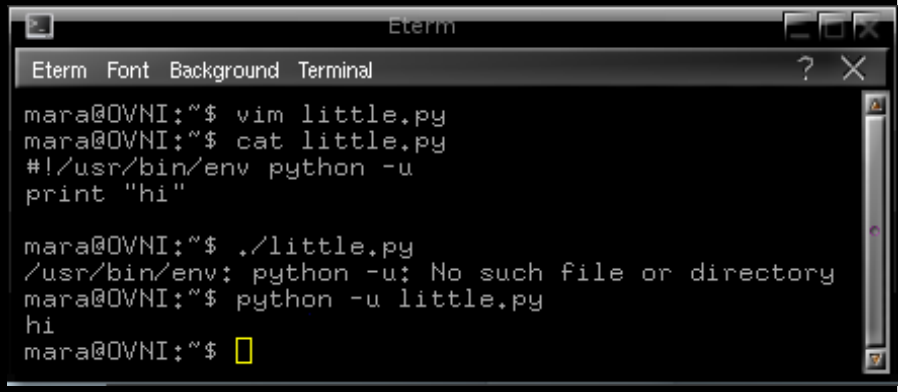
```
#! /usr/bin/python -u
```

Ricky Martin – She Bangs

- shebang редът е предназначен за ОС, не за интерпретатора. В повечето scripting езици # е знак за коментар до края на реда, така че те го пренебрегват
- ако не знаете къде се намира изпълнимият файл на даден интерпретатор, можете да използвате програмата /usr/bin/env в shebang реда:

```
#! /usr/bin/env python
```

Това би разрешило проблеми с евентуално пускане на вашия script на система, в който изпълнимите файлове се намират на по-различни места.
НО може да имате проблеми, ако искате да зададете и опции към интерпретатора (особено в Linux)



Автори: Мария Николова и Деян Дойчев

Използвани ресурси:

- Wikipedia
- лекции на гл. ас. Моника Филипова
- Bash Guide for Beginners и други