Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

А. С. Афанасенко

ОБРАБОТКА ДАННЫХ СРЕДСТВАМИ КОМАНДНОЙ ОБОЛОЧКИ BASH И ЯЗЫКА PYTHON

Учебно-методическое пособие

Санкт-Петербург
2019

Аннотация

Даны краткие теоретические сведения об архитектуре Unix-подобных операционных систем. Представлены наиболее универсальные средства взаимодействия пользовательских приложений консольного типа с операционной системой и друг с другом. Рассмотрены возможности командной оболочки BASH по управлению файлами, созданию процессов и написанию простых сценариев. Описаны основные утилиты, применяемые для обработки текстовых и табличных данных. Дана краткая характеристика высокоуровневого языка программирования python с акцентом на приемы разработки и распространения консольных приложений, выполняющих обработку данных локально и по сети. Все разделы сопровождаются заданиями для самостоятельного выполнения.

Учебно-методическое пособие предназначено для студентов института информационных систем и защиты информации, изучающих курс «Кроссплатформенное программирование». Представляет интерес для студентов других специальностей, а также специалистам, занимающимся обработкой данных в командной строке и интересующимся возможностями различных операционных систем.

Введение

Большинство современных операционных систем для персональных и мобильных вычислительных устройств основано на концепциях ОС Unix и являются в той или иной степени Unix-подобными. Это позволяет разработчикам программного обеспечения проектировать приложения, способные работать на различных программных платформах, используя единые принципы разработки и общие высокоуровневые интерфейсы. Несмотря на повсеместное распространение графического пользовательского интерфейса в операционных системах для персонального использования, для задач обработки данных в настоящее время чаще всего используются приложения с интерфейсом командной строки (консольные приложения). Наиболее доступными и в то же время мощными средствами обработки данных в ОС Linux, OS X, а также Windows, являются командная оболочка Bash [1] и высокоуровневый интерпретируемый язык программирования Python. Эти средства бесплатны, популярны и входят в состав многих дистрибутивов ОС для персональных компьютеров.

1 Основные концепции Unix-подобных операционных систем

1.1 Архитектура

Наиболее общий взгляд на архитектуру Unix позволяет увидеть двухуровневую модель системы, состоящую из пользовательской и системной части (ядра). Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы (процессы в пользовательской части операционной системы) от особенностей аппаратной архитектуры. Ядро имеет набор услуг, предоставляемых прикладным программам посредством системных вызовов (рис. 1.1). Таким образом, в системе можно выделить два уровня привилегий: уровень системы (привилегии специального пользователя root) и уровень пользователя (привилегии всех остальных пользователей) [2].

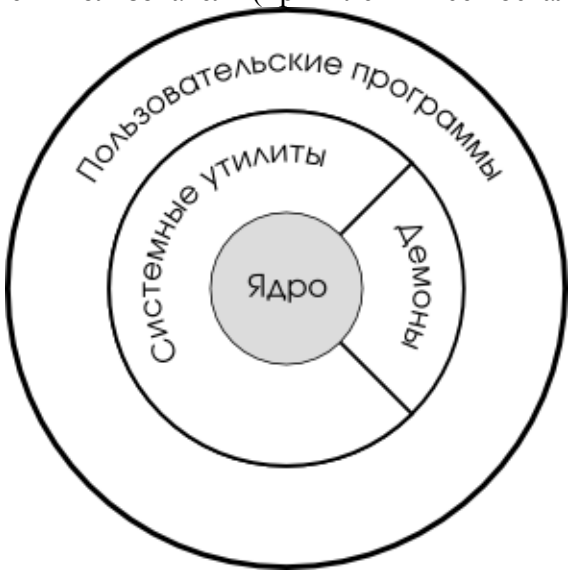


Рис. 1.1. Двухуровневая архитектура Unix-подобных ОС

1.2. Основы работы с процессами в UNIX-подобных операционных системах

Условные обозначения

1. Ввод команд в оболочке bash

```
$ date -u
```

2. Результат выполнения команды в терминале

```
Tue Sep 17 14:35:28 UTC 2019
```

3. Содержимое файла

```
#!/usr/bin/bash
X=1
```

Общие понятия о процессах

Многозадачность

Основным вычислительным ресурсом компьютера является его центральный процессор (или процессоры). В каждый момент времени один процессор может выполнять только одну задачу. Одна из важнейших функций любой многопользовательской и многозадачной операционной системы состоит в планировании задач таким образом, чтобы за счет их переключения создавался эффект одновременного выполнения нескольких задач. Кроме этого, операционная система должна обеспечивать взаимодействие различных процессов на одном компьютере.

Процесс

Стандарты UNIX, а именно IEEE Std 1003.1, 2004 Edition, определяют процесс как «адресное пространство с одним или несколькими потоками, выполняющимися в нем, и системные ресурсы, необходимые этим потокам.»

Другими словами, **процесс** – это программа или ее часть, находящаяся в стадии выполнения. Простая программа (например, калькулятор) порождает лишь один процесс, более сложная программа (например, http-сервер) может порождать десятки и сотни процессов.

В каждой ОС существуют ограничения на общее число процессов в системе и на число процессов, запущенных одним пользователем. Максимальное число процессов для одного пользователя можно узнать с помощь команды ulimit. Например, в ОС Ubuntu 18:

```
$ ulimit -u
11753
```

В каждом процессе, помимо самой работающей программы, ОС создает среду для выполнения задачи, в которой обеспечиваются необходимые условия (наличие ресурсов) и необходимый уровень изоляции от других процессов (защищенность).

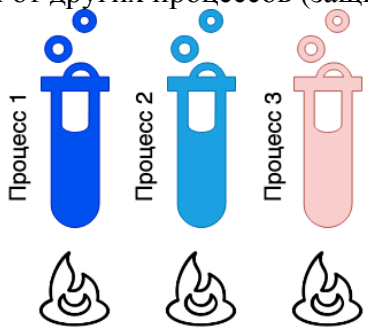


Рис. 1.2. Аллегорическое изображение процессов в ОС

При одновременном выполнении нескольких копий одной программы, каждая из них представляет собой отдельный процесс. При этом в физической памяти будет находиться только одна копия исполняемого кода.

Ресурсы ОС, предоставляемые процессу

К ресурсам, прежде всего, относятся:

- память;
- процессор (в выделенные кванты процессорного времени);
- устройства ввода-вывода, включая терминал или псевдо-терминал:
- файловая система:
- услуги ядра ОС.

Выполнение процесса заключается в точном следовании набору инструкций, при котором управление никогда не передаётся другим процессам. Процессу недоступны данные и стеки других процессов. Но процессы могут обмениваться друг с другом данными с помощью системы межпроцессного взаимодействия, предоставляемой ОС UNIX. К этой системе относятся:

- сигналы;
- каналы (двунаправленная буферизация файла);
- обычные файлы на диске;
- разделяемая (общая) память;
- семафоры;
- сообщения.

Адресное пространство

Каждый процесс в Unix-подобной операционной системе имеет собственное адресное пространство, объем которого может значительно превосходить объем физической памяти, установленной на компьютере. Например, в 32-разрядной ОС каждый процесс имеет адресное пространство объемом 4ГБ, даже если объем физической памяти составляет 128 МБ. Это достигается благодаря технологии виртуальной памяти.

В пределах адресного пространства выделяются непересекающиеся области памяти различного назначения – исполняемый код, переменные, файловые отображения и т.д. При попытке выполнения любой недопустимой операции с памятью ядро операционной системы уничтожает процесс.

Системное окружение

Помимо вышеперечисленных ресурсов, каждый процесс при запуске получает изолированное системное окружение. Окружение (environment) включает набор именованных переменных среды. Чтобы вывести все переменные окружения текущего процесса вместе с их значениями, используется команда env

```
$ env
TERM_PROGRAM=iTerm.app
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/g0/f37j4m3d10g2gy787ftgdcww0000gn/T/
…
```

Потоки ввода-вывода

Каждому процессу предоставляются три стандартных потока ввода-вывода:

- stdin (файловый дескриптор 0);
- stdout (файловый дескриптор 1);
- stderr (файловый дескриптор 2).

Стандартные потоки ввода-вывода можно рассматривать как файлы, уже открытые для чтения (stdin) и для записи (stdout, stderr). Каждый из этих потоков может быть ассоциирован с терминалом, с файлом на диске, с устройством ввода-вывода (последовательный порт) и т.д.

Типы процессов

Все процессы можно разделить на фоновые, не требующие взаимодействия с пользователем, и интерактивные, в которых выполнение кода программы чередуется с ожиданием пользовательских действий.

Системные (невыгружаемые) процессы

Являются частью ядра, всегда расположены в оперативной памяти и не предполагают взаимодействия с пользователем. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы. Таким образом, системные процессы могут вызывать функции и обращаться к данным, недоступным для остальных процессов.

К системным процессам можно отнести и процесс начальной инициализации, **init**, являющийся прародителем всех остальных процессов. Хотя **init** не является частью ядра и его запуск происходит из исполняемого файла, работа этого процесса жизненно важна для функционирования всей системы в целом. Расположение исполняемого файла **init** может быть различным. Например, в Ubuntu 18 это:

```
$ which init
/sbin/init
```

Процессы-«демоны» (службы)

Неинтерактивные процессы, которые запускаются обычным образом - путем загрузки в память соответствующих им программ, и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы, но после инициализации ядра и обеспечивают работу различных подсистем UNIX: системы терминального доступа, системы печати, сетевых служб и т.д. Демоны не связаны ни с одним пользователем. Большую часть времени демоны ожидают, пока тот или иной процесс запросит определенную услугу.

Пользовательские (прикладные) процессы

К пользовательским процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Важнейшим пользовательским процессом является начальная командная оболочка, обеспечивающая выполнение команд пользователя в системе UNIX.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме. Для консольных приложений (не имеющих графического интерфейса) возможен переход из одного режима в другой в зависимости от того, подключен ли к ним терминал.

Атрибуты процесса

К основным атрибутам любого процесса относятся:

- идентификатор текущего процесса (PID);
- идентификатор родительского процесса (PPID);
- ассоциированная с процессом терминальная линия (TTY);
- пользователь, от имени которого запущен процесс (UID);
- приоритет выполнения (PRI).

Просмотр атрибутов процесса

Атрибуты процессов можно узнать либо из таблицы процессов, либо путем изучения содержимого системной папки /proc. Для просмотра таблицы процессов используется команда ps с набором флагов.

```
$ ps -ecf
UID        PID  PPID CLS PRI STIME TTY          TIME CMD
root         1     0 TS   19 20:02 ?        00:00:02 /sbin/init splash
root         2     0 TS   19 20:02 ?        00:00:00 [kthreadd]
root        11     2 FF  139 20:02 ?        00:00:00 [migration/0]
root        12     2 FF   90 20:02 ?        00:00:00 [idle_inject/0]
avahi      498   491 TS   19 20:02 ?        00:00:00 avahi-daemon: chroot
helper
root       515     1 TS   19 20:02 ?        00:00:00 /usr/sbin/NetworkManager
--no-daemon
root       516     1 TS   19 20:02 ?        00:00:00 /usr/bin/python3
/usr/bin/
whoopsie   650     1 TS   19 20:02 ?        00:00:00 /usr/bin/whoopsie -f
kernoops   653     1 TS   19 20:02 ?        00:00:00 /usr/sbin/kerneloops --
test
nius      1149     1 TS   19 20:02 ?        00:00:00 /lib/systemd/systemd --
user
nius      1421  1149 TS   19 20:02 ?        00:00:00 /usr/lib/gnome-online-
accounts/goa-daemon
root      1440     1 TS   19 20:02 ?        00:00:00 /usr/lib/x86_64-linux-
gnu/boltd
root      1465     1 TS   19 20:02 ?        00:00:06
/usr/lib/packagekit/packagekitd
nius      1466  1179 TS   19 20:02 tty1     00:00:00 /usr/lib/gnome-settings-
daemon/gsd-power
nius      3035  2076 TS   19 21:52 pts/0    00:00:00 ps –ecf
...
```

Идентификатор процесса (PID).

Уникальное целое число. В системе не может быть одновременно два процесса у одинаковыми PID, но одна и та же программа, запущенная несколько раз, чаще всего будет иметь различные PID. Процесс init имеет PID=0.

В оболочке bash идентификатор текущего процесса можно получить из встроенной переменной $$:

```
$ echo $$
1168
```

В python для этого используется функция os.getpid().

Родительские и дочерние процессы

Создание нового процесса происходит путем явного или неявного вызова системной функции fork() из родительского процесса. То есть, каждый процесс, кроме процесса init, порождается одним родительским процессом. PID родительского процесса совпадает с PPID дочернего процесса. Дочерний процесс всегда «знает» идентификатор родительского процесса и при старте наследует набор ресурсов, практически полностью повторяющий ресурсы родительского процесса. Это не означает, что дочерний и родительский процесс выполняют одну и ту же программу – речь идет лишь о наследовании среды выполнения.

Есть ресурсы, которые не наследуются дочерним процессом. В частности, дочерний процесс получает собственную таблицу файловых дескрипторов, являющуюся копией таблицы родительского процесса на момент вызова fork(). Это означает, что открытые файлы наследуются, но если дочерний процесс, например, закроет какой-либо файл, то это не повлияет на таблицу дескрипторов процесса-родителя.

В оболочке bash идентификатор родительского процесса можно получить из встроенной переменной $PPID:

```
$ echo $PPID
1167
```

В python для этого используется функция os.getppid(), но в версии для Windows она не работает, так как процессы Windows не хранят данный атрибут.

Запуск программы в дочернем процессе

Для примера напишем скрипт child.sh

```
#!/usr/bin/env bash
echo “process $$ born by $PPID”
sleep 5
```

Разрешение на выполнение данного скрипта установим командой

```
$ chmod +x child.sh
```

Данный скрипт можно запустить несколькими способами:

1. Запуск в дочернем процессе с блокировкой родительского процесса. Это обычный способ запуска программы. Родительский процесс (в данном случае текущая интерактивна сессия bash) ждет завершения дочернего процесса и продолжает работу.

```
$ ./child.sh
process 11334 born by 1168
```

2. Запуск скрипта без создания нового процесса.

```
$ source child.sh
process 1168 born by 1167
```

3. Запуск дочернего процесса в фоне

```
$ (./one.sh) &
[1] 11354
process 11354 born by 1168
[1]+  Done                    ( ./one.sh )
```

Круглые скобки в данном случае означают, что стандартный вывод дочернего процесса будет соединен с окном терминала, в котором запущен родительский процесс.

Жизненный цикл процесса

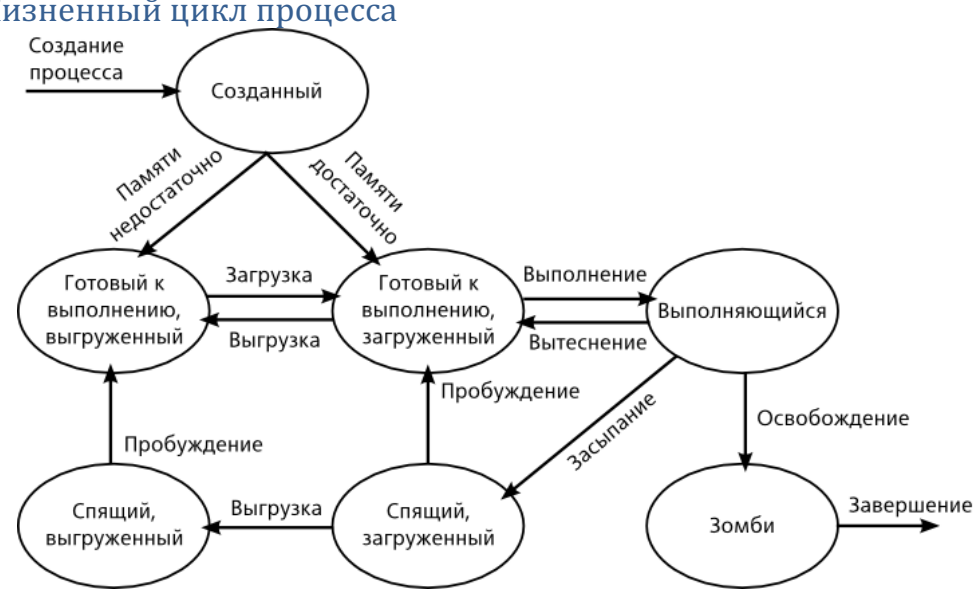


Рис. 1.3. Диаграмма состояний процесса

Сигналы

Применение сигналов

Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события (аналогично аппаратным прерываниям). Каждое событие имеет свой числовой идентификатор (обычно в диапазоне от 1 до 36) и соответствующую символьную константу - имя. При работе с сигналами необходимо различать две фазы:

1. Генерация или посылка сигнала.
2. Доставка и обработка сигнала.

Сигнал отправляется, когда происходит определенное событие, о наступлении которого должен быть уведомлен процесс. Сигнал считается доставленным, когда процесс, которому был отправлен сигнал, получает его и выполняет его обработку. В промежутке между этими двумя событиями сигнал ожидает доставки.

Сигнал может посылаться одним процессом другому (с помощью соответствующего системного вызова) и будет доставлен, если оба процесса - одного пользователя или сигнал послан от имени пользователя **root**. Сигналы посылаются также ядром.

Ядро генерирует и посылает процессу сигнал в ответ на ряд событий, которые могут быть вызваны самим процессом, другим процессом, прерыванием или каким-либо внешним событием. Основные причины отправки сигнала:

Исключительные ситуации. Выполнение процесса вызывает исключительну ситуацию, например, деление на 0.

Терминальные прерывания. Нажатие клавиш терминала, например, <Del>, <Ctrl+C>, <Ctrl+\>, вызывает посылку сигнала текущему процессу, связанному с терминалом.

Другие процессы. Процесс может посылать сигнал другому процессу или группе процессов с помощью системного вызова **kill**. В этом случае сигналы являются элементарной формой межпроцессного взаимодействия.

Управление заданиями. Командные интерпретаторы, поддерживающие средства управления заданиями, используют сигналы для манипулирования фоновыми и текущими процессами. Когда процесс, выполняющийся в фоновом режиме, делает попытку чтения или записи на терминал, ему посылается сигнал останова. Когда порожденный процесс завершает свою работу, родительский процесс уведомляется об этом также с помощью сигнала.

Квоты. Когда процесс превышает выделенную ему квоту вычислительных ресурсов или ресурсов файловой системы, ему посылается соответствующий сигнал.

Уведомления. Процесс может запросить уведомление о наступлении тех или иных событий, например, готовности устройства и т.д. Такое уведомление посылается процессу в виде сигнала.

Будильники. Если процесс установил таймер, ему будет послан сигнал, когда значение таймера станет равным 0.

Завершение процессов

Стадии завершения процесса

Строго говоря, программа не может завершить свой процесс самостоятельно, а может лишь «попросить» об этом систему. Либо с помощью системного вызова _exit(int status), который может происходить неявно, либо в результате критической ошибки. При завершении процесса:

- закрываются все открытые файловые дескрипторы
- у всех незавершенных дочерних процессов значение PPID устанавливается в 1.

Завершение процесса не может произойти мгновенно. Какое-то время после завершения другие процессы могут узнать код возврата завершенного процесса (при штатном завершении это значение параметра status). Процессы, находящиеся в стадии завершения, называются «зомби».

Использование сигналов

Основные сигналы, которые ОС может посылать для завершения процесса

Имя сигнала	Перехватываемый	Описание
SIGTERM	Да	Стандартный сигнал, посылаемый для остановки процесса.
SIGHUP	Да	Отключился терминал (или закрыто терминальное окно). Сигнал посылается всем не фоновым процессам, связанным с соответствующей терминальной линией.
SIGKILL	Нет	Неперехватываемый сигнал, позволяющий завершить любой процесс.
SIGTSTP	Да	Это запрос от терминала на остановку процесса. Посылка этого сигнала процессу происходит при нажатии комбинации клавиш **Ctrl-Z**.

Для посылки сигналов из командной строки используется команда **kill**. Она имеет следующий синтаксис:

kill [**-сигнал**] **pid** ...

Эта команда посылает указанный сигнал (по умолчанию - **SIGTERM**) всем процессам с указанными идентификаторами. Посылать сигнал можно и не существующему процессу - выдается предупреждение, но другим процессам сигнал посылается. Посылаемый сигнал задается по имени без префикса **SIG** или по номеру. Например, чтобы вызвать «зависание» текущего сеанса, выполните команды:

```
$ echo $$
1168
$ kill -STOP 1168
```

2 Практикум по изучению оболочки BASH

2.1 Начало работы

Работа выполняется на реальной или виртуальной машине с установленной операционной системой семейства Unix, в приложении «Терминал». Часть упражнений может быть выполнена в Windows – пользователям данной ОС доступно несколько вариантов оболочки BASH, например, MSYS. Начиная с Windows 10,

2.2 Навигация по файловой системе

Команды

```
ls [флаги] [имя каталога или файла]
```

Просмотр содержимого каталога и (или) файловых атрибутов. Если не задано имя каталога, используется текущий каталог.

```
pwd
```

Вывод имени текущего каталога (print working directory).

Упражнение

Команда	**Комментарий**
`ls`	Просмотреть содержимое текущего каталога
`pwd`	Просмотреть полный путь к текущему каталогу
`ls /` `ls ~`	Просмотреть содержимое корневого каталога («/») Символ «~» обозначает домашнюю папку текущего пользователя (полный путь зависит от системы)
`ls -l /`	Просмотреть содержимое корневого каталога с подробными атрибутами каждого файла.

`ls -l /usr/`
`ls -l /usr/local/`
`ls -l /usr/local/bin/`
	Углубление в файловую систему.
	Обратите внимание, что разные типы файлов для удобства подсвечиваются разным цветом (зависит от настроек внешнего вида командной оболочки).
	Писать косую черту в конце пути обязательно в том случае, если существует обычный файл с таким же именем.

Полезные советы

Стрелки «вверх» и «вниз» используются для прокрутки истории введенных ранее команд. С их помощью можно повторять или изменять ранее введенные команды.

Клавиша «Tab» вызывает функцию автодополнения, благодаря чему нет необходимости полностью вводить длинные имена файлов и папок.

Обратите внимание на дополнительную информацию о файлах, которую выводит команда ls с флагом -l. Разберем строку

drwxr-xr-x 7 root root 4096 авг. 5 17:59 include

тип	права доступа	количество ссылок	владелец	группа	размер (байт)	метка времени	имя файла

d	rwxr-xr-x	7	root	root	4096	авг. 5 17:59	include

2.3 Создание, копирование и удаление файлов

Команды

	создание	удаление
файл	touch <имя файла>	rm <имя файла>
каталог	mkdir <имя каталога>	rmdir <имя каталога> удаляет только пустые каталоги

```
cp <копируемый файл> <папка назначения> Копирование файла.
cd <имя каталога> Смена текущего каталога.

rm –rf <имя каталога> безвозвратно удаляет всё содержимое каталога,
включая вложенные каталоги.
```

Упражнение

Команда	**Комментарий**
`cd ~`	Переход в домашнюю папку.
`mkdir pract`	Создание папки ~/pract
`mkdir pract/class1`	Создание папки ~/pract/class1
`touch /pract/tempfile`	Создание файла /pract/tempfile. Команда должна вернуть ошибку, если в корневой папке до этого не был создан каталог pract. В предыдущем упражнении pract был создан в домашней папке (~), а не в корневой (/).
`touch pract/tempfile` `ls -l pract`	Создание файла еще раз в правильном месте. Просматриваем содержимое и убеждаемся, что файл создан.
`cd pract` `cp tempfile cl[Tab]`	Переход в папку ~/pract, чтобы писать более короткие имена файлов. Копирование файла tempfile в ~/pract/class1. После ввода букв cl нажмите клавишу Tab (без квадратных скобок!). При этом сработает функция автодополнения.
`rm tempfile`	Удаление tempfile. Если нужно переместить файл, проще воспользоваться командой mv (move), она имеет одинаковый синтаксис с командой cp.
`rmdir class1`	Удаление папки ~/class1. Команда вернет ошибку, т.к. папка не пуста, а команда rmdir удаляет только пустые папки.
`touch class1/tempfile` `touch class1/.hidden`	Повторное создание tempfile. Создание еще одного файла.
`ls -l class1/` `ls -lh class1/`	Отображение содержимого class1. Обратите внимание на то, что файл .hidden отсутствует! В Unix файлы и папки, имена которых начинаются с точки, считаются скрытыми. Чтобы увидеть скрытые файлы, необходимо добавляем флаг –a.

2.4 Перенаправление вывода команды в файл

Вывод любой команды можно перенаправить из консоли в текстовый файл. Этот прием является частным случаем перенаправления стандартных потоков ввода-вывода. Как

правило, перенаправление в файл используется для сохранения результатов работы программы, но аналогичным образом можно записать в файл и содержимое каталога. Перенаправление стандартного вывода в файл записывается так:

```
$ <команда> > <имя файла>
```

Упражнение

Команда	**Комментарий**
`ls -l ~ > ~/pract/class1/dump`	
`ls -l ~/pract/class1/`	Проверка того, что файл dump действительно создан и имеет ненулевой размер

2.5 Просмотр текстовых файлов и получение дополнительной информации о них

Команды

```
cat <имя файла> Построчно выводит содержимое файла в консоль.
wc <имя файла> Показывает число байт, число слов и число символов «новая строка».
```

2.6 Создание и удаление «жестких» и «мягких» ссылок

Команды

```
ln <имя существующего файла> <имя нового файла> Создает “жесткую” ссылку на файл.
ln -s <имя существующего файла> <имя нового файла> Создает “мягкую” (символическую) ссылку на файл.
```

Упражнение

Команда	**Комментарий**
Самостоятельно создайте каталоги: ~/pract/class1/a ~/pract/class1/a/aa ~/pract/class1/a/aa/aaa	
Скопируйте файл dump, созданный в п. 2.4, в папку aaa.	
Перейдите в ~/pract/class1/a/aa/aaa	

`ls -l`
`ln dump hardlink`
`ln -s dump softlink`
`ls -l`
	Обратите внимание на счетчик ссылок файла dump.
	«Жесткая» ссылка.
	«Мягкая» ссылка.
	Обратите внимание на типы файлов и на то, как изменились счетчики ссылок.

Самостоятельно удалите файл dump	
Попробуйте вывести на экран содержимое файла hardlink	Объясните результат.

2.7 Конвейерное выполнение группы команд

Если нужно передать результат выполнения одной команды на вход другой, их можно записать через разделитель «|». Передача результатов происходит слева направо. Можно объединять таким образом более двух команд.

Команды

```
head -n[число строк] Копирует первые n строк входного текста на выход, остальные игнорирует.
tail -n[число строк] Копирует последние n строк входного текста на выход, остальные игнорирует.
grep <выражение> Копирует на выход только те строки входного текста, которые отвечают заданному регулярному выражению. В простейшем случае выполняет поиск строк, содержащих заданное слово.
sudo (“substitute user and do”) Префикс для выполнения команд от имени суперпользователя
su Вход в учетную запись суперпользователя.
exit Выход из текущей учетной записи. Если завершен последний пользовательский сеанс, при этом закрывается терминал.
```

Упражнение

Команда	**Комментарий**
`cat ~/pract/class1/dump | head -n3`	Вывод первых 3 строк файла dump.
`sudo su`	Вход в сеанс суперпользователя.
`touch ~/pract/class1/dump2`	Создание нового файла от имени суперпользователя.
`exit`	Выход из сеанса суперпользователя.
`sudo touch ~/pract/class1/dump3`	Выполнение команды от имени суперпользователя без входа в сеанс.
`ls -l ~/prac/class1/ | grep root`	Таким образом, будут выведены только те файлы, в описании которых есть слово root.

2.8 Резюме

Вы попробовали в деле основные команды оболочки BASH для манипуляции с файлами. Если Вам было слишком легко, начните самостоятельно осваивать текстовый редактор vim – непривычный современному пользователю, но довольно мощный инструмент работы в Unix-системах.

3 Обработка данных командами оболочки BASH

Целью работы является освоение обработки данных с помощью команд grep, sed и awk оболочки BASH, приобретение навыков создания и вызова скриптов.

3.1 Подготовка входных данных

Продавцы овощного рынка в конце рабочего дня отдают своему начальнику текстовый файл, в котором описаны результаты дневной торговли. Столбцы в этом файле разделены символами табуляции. Первую строку (заголовок) начните символом '#', чтобы ее было легко отфильтровать при обработке. Пример содержимого файла приведен ниже:

наименование овощей	привезено со склада, кг	продано за день, кг	средняя цена за 1 кг
Огурцы	100	70	25
Капуста	300	201	20
Помидоры	200	200	40
…	…	…	…

Имя файла может быть произвольным.

Подготовьте такой файл на основе известных вам рыночных цен.

Создайте еще 2-3 аналогичных файла с немного другим содержимым (другой ассортимент, другие цены). Можете обменяться файлами с одногруппниками.

3.2 Обзор команды awk

Команда awk – довольно мощное средство для обработки данных, и это не просто утилита командной строки, а самостоятельный микро-язык программирования. Простейший вариант вызова команды awk следующий:

```
$ awk '<сценарий обработки текста>' [имя файла [имена дополнительных файлов]]
```

При вызове с одним аргументом awk обрабатывает строки из потока стандартного ввода. Сценарий заключается в кавычки, чтобы интерпретатор воспринимал его как один аргумент, независимо от пробелов внутри сценария. Рассмотрим простейший сценарий, который позволит просматривать все цены на тот или иной овощ.

```
$ awk '/^Кабачок/ {print $4}' …
```

Здесь первая часть /^Кабачок/ - это шаблон выбора строк, а вторая - {print $4} – действие над выбранными строками. То есть для строк, начинающихся со слова «Кабачок» будет выведено 4-е поле (цена за килограмм). Остальные строки будут проигнорированы. Один сценарий может включать различные действия для различных видов строк:

```
$ awk '/шаблон/ {действие} /шаблон/ {действие} /шаблон/ {действие} …'
…
```

Если шаблон отсутствует, следующее за ним действие выполняется для всех строк. Если действие не указано, awk выводит всю строку (то есть по умолчанию применяет действие {print}).

Также можно указать действия, выполняемые в начале и в конце обработки:

```
$ awk 'BEGIN {начальное действие} /шаблон/ {действие} … END {завершающее действие}' …
```

Для выполнения лабораторной работы достаточно сценариев с одним действием, состоящих из 4 элементов:

Порядковый №	Назначение	Примеры

1	Инициализация (выполняется один раз)	`BEGIN {print “Начинаю работать"}`
2	Шаблон выбора строк. Если перед шаблоном поставить «!», он превратится в шаблон игнорирования строк.	`/^[0-9]/`
3	Действия в каждой строке	`{printf “эта строка начинается числом %s”, $1}`
4	Действия в конце	`END {print “Кажется, всё”}`

Если объединить эти элементы, получим команду:

```
$ awk ‘BEGIN {print “Начало "} /^[0-9]/ \
 {printf “эта строка начинается числом %s\n”, $1} END {print “Всё”}’
```

Приведенное регулярное выражение /^[0-9]/ выделит только те строки, которые начинаются с арабской цифры. Синтаксис сценариев awk во многом похож на язык Си, что неудивительно – один из создателей awk, Б. Керниган, впоследствии стал соавтором стандарта языка Си.

Можно сказать, что хорошее знание awk позволяет обходиться без команд grep и sed, но на практике это не всегда удобно – начинающему программисту сложнее написать без ошибок громоздкую команду awk, чем скомбинировать вызовы нескольких простых команд.

3.3 Вычисление дневной выручки

Для вычисления дневной выручки по одному файлу, очевидно, необходимо вычислить сумму произведений в 3 и 4 столбцах по всем строкам, кроме первой.

Команда awk позволяет делать произвольную построчную обработку текста, оперируя фрагментами строк как отдельными переменными. Эти переменные именуются $1, $2, $3 и т.д. Пример, для вывода на экран произведений 3 и 4 столбца в каждой строке:

```
$ awk ‘{print $3 * $4}’ <имя файла>
```

Кроме элементов текущей строки $1, $2, $3, Вы можете оперировать такими встроенными переменными, как NR – номер текущей строки, NF – число столбцов в текущей строке, и т.д. (см. man awk), а также заводить собственные переменные, как показано ниже. Задействовав блоки BEGIN и END, запишем команду подсчета выручки

```
$ awk ‘BEGIN {sum = 0} {sum += $3*$4} END {print sum}’ <имя файла>
```

Убедитесь, что эта команда правильно обрабатывает созданный ранее файл, и самостоятельно выполните следующие задания:

1. Добавьте шаблон для игнорирования строк, начинающихся с символа «#».
2. Занесите эту команду в файл .sh, оформив его должным образом. Теперь имя файла должно передаваться в скрипт при запуске, а уже внутри скрипта - команде awk. Например, вы создали скрипт daily_revenue.sh. Он должен обрабатывать тот файл, который ему укажут: ./daily_revenue.sh <имя файла>.
3. Измените скрипт таким образом, чтобы он мог обработать любую группу файлов, заданную обычным для BASH способом. Например, вызов ./daily_revenue.sh veg*.txt должен обработать всех файлов, имена которых начинаются с “veg” и заканчиваются на “.txt”.
4. Вызовите скрипт так, чтобы он вычислил выручку по всем имеющимся у вас файлам. При необходимости переименуйте часть файлов, если они не могут быть описаны одним шаблоном.

5. Измените скрипт таким образом, чтобы он в конце своей работы выводил не просто число, а сообщение: «Выручка за день составила …. рублей».

3.4 Правка исходных данных

Выведите содержимое всех файлов командой cat. Такой вывод имеет недостатки: повторяющиеся наименования овощей могут находиться в разных местах. Загооловочная строка таблицы встречается несколько раз. Для преодоления первого недостатка используйте команду sort. Для удаления заголовочных строк используйте команду grep (предпочтительно) или команду awk.

Теперь вы можете наблюдать ситуацию, что один и тот же овощ в разных файлах назван по-разному, что может несколько затруднить выполнение следующего пункта лабораторной работы. Например, помидоры могут называться

томаты
помидор
помидоры
помидорки
памедоры

Используйте команду замены текста sed, запуская ее на всех файлах с овощами. Добейтесь того, чтобы в вашем наборе данных не осталось таких повторов.

3.5 Вывод ассортимента овощей

Сформируйте список неповторяющихся наименований овощей, встречающихся во всех исходных файлах. Для этого понадобится извлечь первый столбец, удалить заголовочные строки, отсортировать полученные строки текста и удалить дубликаты. Используйте конвейер команд. Вам могут понадобиться команды grep, sort, uniq, cut, sed и awk.

Данное задание может быть выполнено несколькими способами, приведите хотя бы один.

3.6 Вычисление непроданного остатка

Создайте копию скрипта подсчета дневной выручки. Переделайте новый скрипт таким образом, чтобы он выводил сообщение о том, сколько килограмм определенного овоща остались непроданными. Для начала название требуемого овоща можно задать локальной переменной внутри скрипта, например

```
VEG=помидоры
```

Очевидно, что для вычисления остатка помидоров строки с другими овощами должны игнорироваться. Доработайте скрипт, чтобы он выполнял заданное действие. Вы можете сделать это, меняя только параметры вызова команды awk, либо можете задействовать другие известные вам команды, используя конвейер.

Теперь необходимо вынести название требуемого овоща в параметры вызова скрипта, чтобы он работал следующим образом.

Ввод:

```
$ ./veg_stat.sh огурцы veg*.txt
```

Вывод:

```
$ огурцы: на складе осталось 326 кг
```

Напомним, что для работы с аргументами командной строки в оболочке BASH существуют следующие встроенные переменные:

- $1, $2, $3, … - первый, второй, третий и т.д. аргументы;
- $@ или $* - все аргументы в виде единого массива;

- $# - общее число аргументов.

Для выполнения этого задания необходимо разделить аргументы скрипта на две группы: первый аргумент и все остальные. Это можно сделать несколькими способами. Например, используя запись для поддиапазона аргументов:

```
FIRST=$1
OTHERS="${@:2:$#}"
echo "Первый аргумент: $VEG"
echo "Остальные аргументы: $OTHERS"
```

Другой способ разделения аргументов командной строки – использовать обычный цикл.

```
for ARG in $@
do
  …
done
```

Код получается более громоздким, но и более кроссплатформенным, т. к. старые версии командной оболочки (например, sh) не поддерживают синтаксис "${@:2:$#}".

3.7 Три способа запуска скриптов в оболочке BASH

1. Запуск с указанием интерпретатора (необходимы права на чтение). В этом случае ассоциация (#!...), если она указана в файле скрипта, игнорируется.

```
$ bash <файл скрипта>
```

2. Непосредственное выполнение скрипта (необходимы права на исполнение). Интерпретатор выбирается исходя из первой строки скрипта.

   Если каталог, в котором находится нужный файл, включен в переменную окружения $PATH, его можно не указывать. В противном случае необходимо указать относительный или полный путь к файлу. Например, вызов из текущей папки возможен с помощью команды

```
$ ./<файл скрипта>
```

3. Запуск с помощью команды source

```
$ source <файл скрипта>
```

Последний способ также не требует прав на исполнение файла. При этом скрипт выполняется в текущем процессе и, как следствие, запускаемый скрипт и оболочка имеют общую область видимости переменных.

3.8 Содержание отчета

Отчет должен содержать последовательное описание проделанных в работе действий, а также тексты созданных файлов с данными, скриптов, вводимые команды и результаты их выполнения.

Рекомендуется писать отчет таким образом, чтобы

Выводы в конце отчета должны отражать те конкретные новые знания, которые студент получил в процессе выполнения работы. Отчеты, в которых выводы совпадают с целью работы, к защите не принимаются.

3.9 Приложение. О повышении отзывчивости системы при работе на виртуальной машине

Если Вы работаете в VirtualBox, обязательно установите «Virtual box guest additions» - это заметно повысит быстродействие, позволит настраивать разрешение экрана, и т.п.

Завершая работу, не пользуйтесь кнопками выключения компьютера в гостевой ОС (очень долго). Вместо этого при закрытии окна гостевой ОС выбирайте пункт «Сохранить состояние машины».

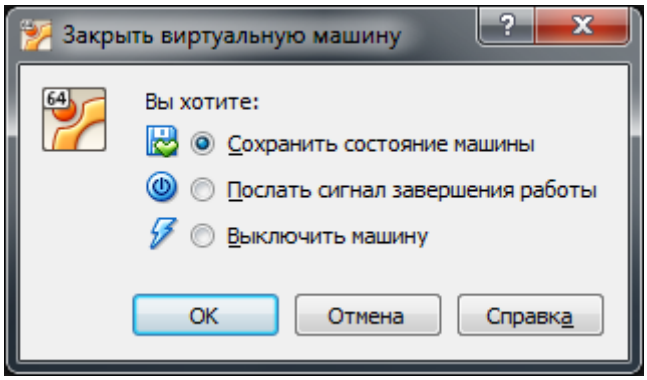


Рис. 3.1. Окно закрытия виртуальной машины

Отключите анимацию окон. Для Ubuntu:

Установите утилиту unity tweak tool, для этого выполните команду

```
sudo apt-get install unity-tweak-tool
```

Запустите ее:

Выберите значок «Общие» (General).

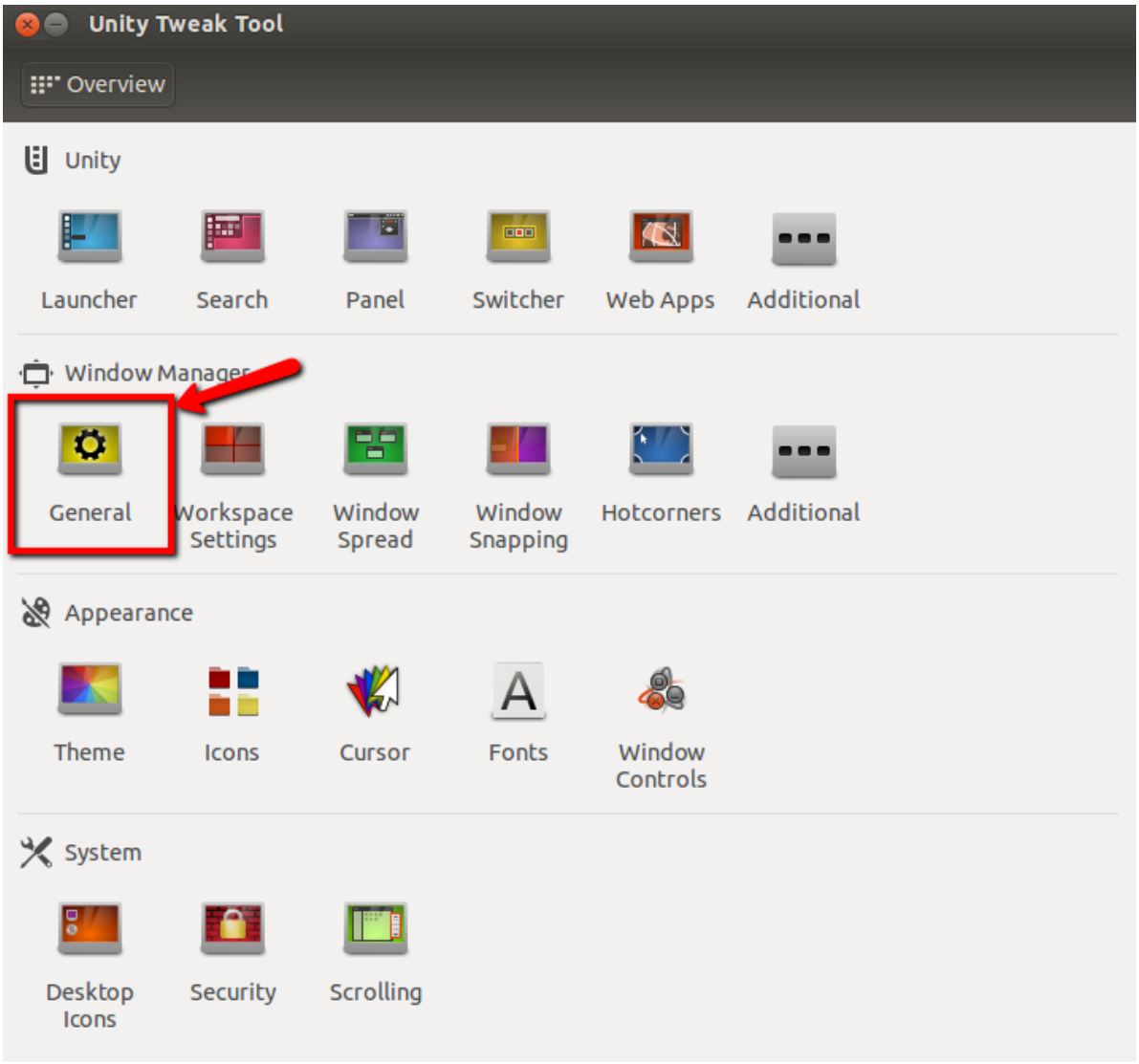


Рис. 3.2. Окно инструмента unity tweak tool

Отключите анимацию окон.

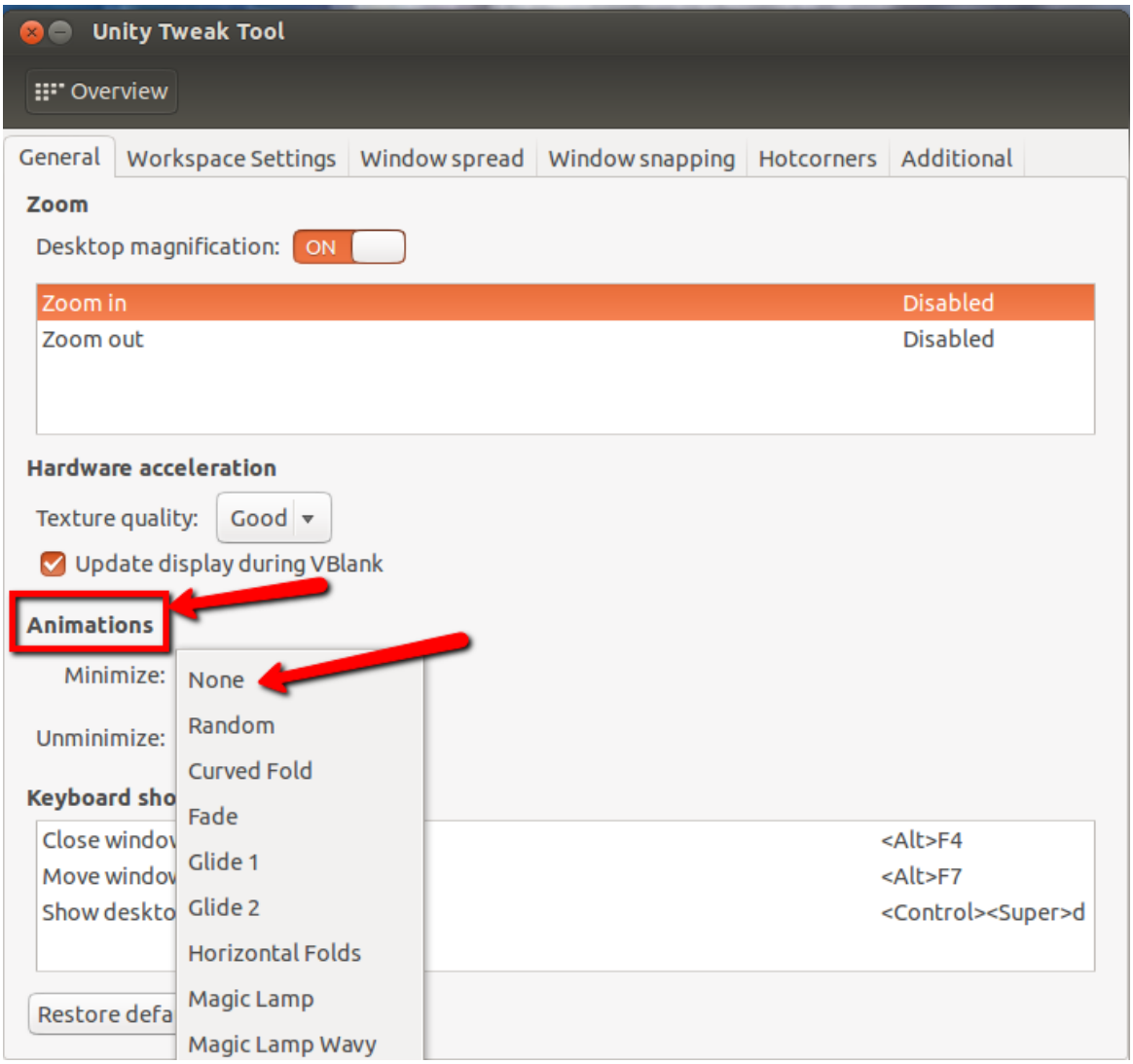


Рис. 3.3. Отключение анимации окон

По умолчанию файловая система виртуальной машины изолирована от файловой системы родительской ОС. Для обмена данными между ними используются общие папки. Сначала общая папка создается в настройках виртуальной машины (в главном меню VirtualBox выберите Machine -> Settings -> Shared Folders). Затем, для подключения общей папки на стороне Linux, выполните команду:

```
$ sudo mount -t vboxsf share ~/host
```

где share – название общей папки Folder Name, данное ей при создании, а ~/host – путь к общей папке в файловой системе гостевой ОС Linux. См. справку по VirtualBox для получения более подробных сведений.

4 Статистический анализ текста средствами языка Python

Целью работы является освоение простых операций с файлами и стандартными контейнерами. При выполнении задания необходимо самостоятельно строить логику обработки данных.

4.1 Постановка задачи

В работе требуется написать программу, вычисляющую распределение предложений по числу слов, а также позволяющую выводить все предложения определенного типа (в зависимости от аргументов запуска). Например, имеется текст:

„Смотри, как расхрабрился!“ говорил Чуб, оставшись один на улице. „Попробуй, подойди! вишь какой! вот большая цяца! ты думаешь, я на тебя суда не найду. Нет, голубчик, я пойду, и пойду прямо к комиссару. Ты у меня будешь знать. Я не посмотрю, что ты кузнец и маляр. Однако ж посмотреть на спину и плечи: я думаю, синие пятна есть. Должно быть, больно поколотил вражий сын! жаль, что холодно и не хочется скидать кожуха! Постой ты, бесовской кузнец, чтоб чорт поколотил и тебя, и твою кузницу, ты у меня напляшешься! вишь, проклятый шибеник! однако ж, ведь теперь его нет дома. Солоха, думаю, сидит одна. Гм... оно ведь недалеко отсюда; пойти бы! Время теперь такое, что нас никто не застанет. Может, и того будет можно... вишь, как больно поколотил проклятый кузнец!“

(Н. В. Гоголь. Ночь перед Рождеством)

Предложение	Число слов
„Смотри, как расхрабрился!“ говорил Чуб, оставшись один на улице.	9
„Попробуй, подойди! вишь какой! вот большая цяца! ты думаешь, я на тебя суда не найду.	15
и т. д.	

Итоговое распределение будет состоять из пар чисел: «число слов в предложении – число предложений».

из 4 слов – 1 предложение
из 5 слов – 1 предложение
из 7 слов – 1 предложение
из 8 слов – 2 предложения
из 9 слов – 2 предложения
из 11 слов – 1 предложение
и т.д.

В программе не разрешается использовать регулярные выражения и нестандартные контейнеры (стандартными являются list, tuple и dict).

Созданный скрипт на языке Python должен без модификаций кода одинаково работать в ОС Windows и одной из Unix-подобных систем.

4.2 Подготовка входных данных

Входным файлов в лабораторной работе будет служить русскоязычный текст в кодировке UTF-8. Для примера возьмите повести А. С. Пушкина «Метель» или «Выстрел». Работая с текстовыми файлами, убедитесь, что они имеют необходимую кодировку (современные текстовые редакторы – Notepad++, Sublime, gedit – позволяют при необходимости изменить кодировку текста, см. Приложение).

4.2.1 Построение гистограммы длин предложений

В зависимости от персональных навыков и предпочтений, Вы можете писать эту программу по-разному. В любом случае, выполнение этого задания состоит из следующих этапов:

1. Передача текстового файла в программу.
2. Считывание текста из файла.

3. Разбивка текста на предложения.
4. Подсчет числа слов в каждом предложении.
5. Подсчет предложений с одинаковым числом слов.

4.3 Передача текстового файла

Используйте передачу содержимого через стандартный поток ввода или передавайте имя файла как аргумент командной строки:

```
stream = sys.stdin
```

или

```
stream = open(sys.argv[1])
```

Если Вы предпочитаете второй вариант, не забывайте проверять наличие аргумента (чтобы программа выдавала сообщение об ошибке при вызове без аргументов). Также не забывайте использовать конструкцию with для безопасного открытия файла.

4.4 Считывание текста из файла

Считывание может быть построчным:

```
for line in stream:
    …
```

или единовременным:

```
text = stream.read()
```

Единовременное считывание потребляет больше памяти, но для обработки файлов среднего объема это несущественно.

4.5 Разбивка текста на предложения

Невозможно однозначно разбить текст на предложения, оперируя текстом как набором символов и игнорируя его смысл [3]. Тем не менее, в большинстве случаев для этого достаточно формальных правил:

- Первое слово предложения начинается с большой буквы.
- В начале предложения может стоять не только слово, но и некоторые знаки препинания.
- Предложение заканчивается точкой, вопросительным знаком, восклицательным знаком или многоточием.
- Те же знаки препинания «.», «?», «!». «…» могут встречаться и в середине предложения. Например, многоточие часто включается писателями в середине предложений с прямой речью.

Задайте в программе набор символов, которые могут разделять соседние предложения. Используйте метод str.split, чтобы не прибегать к регулярным выражениям. К сожалению, метод split не позволяет задать сразу несколько разделителей. Поэтому сделайте предварительную замену. Например, замените вопросительные и восклицательные знаки на точки:

```
text = text.replace('?', '.').replace('!', '.')
sentences = text.split('.')
```

Выявите возможные проблемные ситуации, когда может произойти дробление целого предложения на части. Предложите и реализуйте способы преодоления этих ситуаций.

Не требуется в этом задании достигнуть 100% точности разбивки текста (это невозможно в рамках лабораторной работы). Но, как минимум, программа должна

корректно обрабатывать многоточия и проверять, что первое слово предложения начинается с прописной буквы.

4.6 Подсчет числа слов в каждом предложении

Для подсчета числа слов в каждом предложении можно вновь использовать метод split [4]. Результат подсчета слов можно сохранять в отдельный в отдельный контейнер, а можно хранить вместе с предложением (для упрощения отладки). Обратите внимание на знаки тире: они не должны учитываться, как отдельные слова.

4.7 Подсчет предложений с одинаковым числом слов

Подсчет предложений с одинаковым числом слов удобно сделать на основе словаря, в котором ключ – это число слов, а значение – число предложений.

4.8 Обработка результатов

В упрощенном варианте задания гистограмму длин предложений можно построить в MS Excel или LibreOffice Calc. Оба эти приложения поддерживают импорт данных из формата CSV (“comma-separated values”). Поэтому вычисленные в предыдущем разделе значения – число слов и соответствующее число предложений – надо предварительно записать в формате CSV. Как следует из названия, CSV – это текстовый файл, в котором столбцы разделены запятыми, а строки – символом перевода строки.

В усложненном варианте этого задания гистограмму необходимо построить с помощью утилиты gnuplot ядра Unix [5].

Также приветствуется построение гистограммы непосредственно из программы на Python, с использованием пакета matplotlib.

Пример итоговой гистограммы:

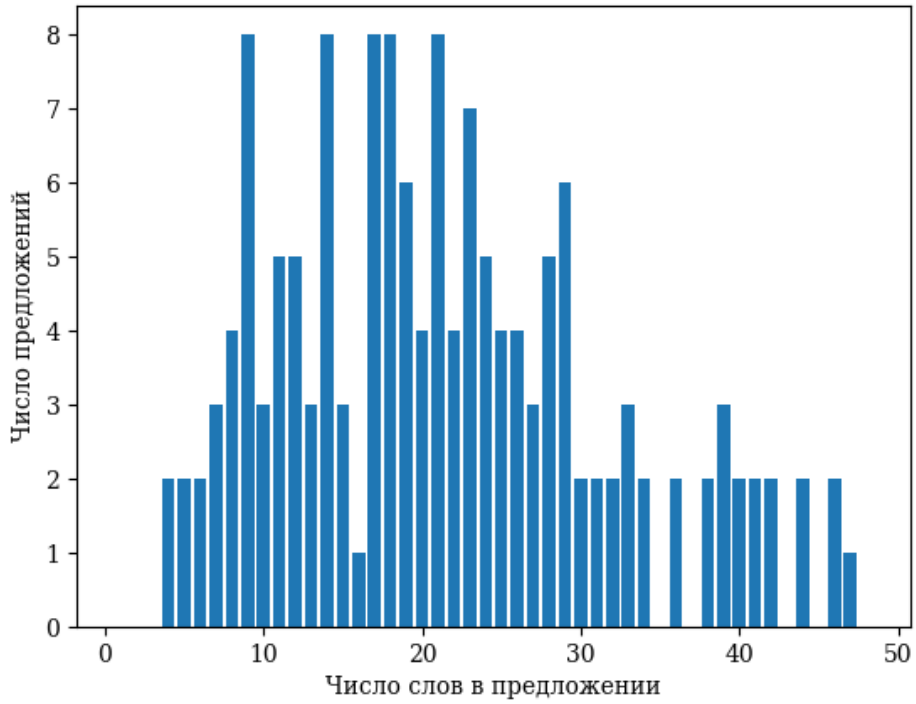


Рис. 4.1. Гистограмм длин предложений

4.9 Содержание отчета

Отчет должен содержать последовательное описание проделанных в работе действий, а также тексты созданных файлов с данными, исходный код программы, вспомогательные команды и результаты их выполнения. Выводы в конце отчета должны отражать те конкретные новые знания, которые студент получил в процессе выполнения работы. Отчеты, в которых выводы совпадают с целью работы, не принимаются.

4.10 Приложение. Смена кодировки текста в редакторах

В Notepad++ Посмотреть текущую кодировку и изменить ее можно через меню «Кодировка»

В редакторе gedit такого меню нет. Узнать текущую кодировку можно, наведя курсор на имя открытого файла, а для изменения кодировки в диалоге «Сохранить как» имеется соответствующий выпадающий список:

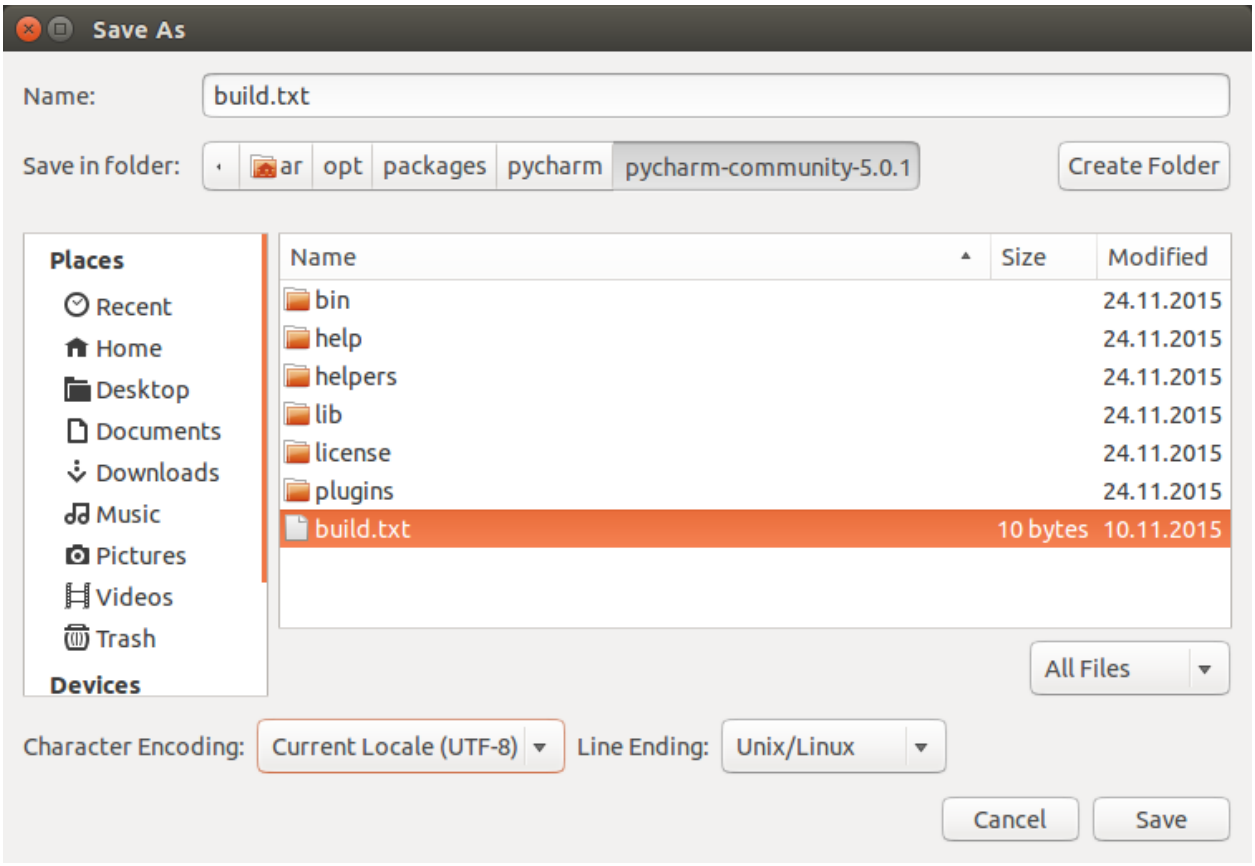


Рис. 4.2. Настройка кодировки текста при сохранении

5 Создание консольных утилит на языке Python

Цель работы – изучить возможности интерфейса командной строки при написании утилитных приложений.

5.1 Постановка задачи

В рамках лабораторной работы необходимо написать две утилиты на языке python. Первая утилита проверяет наличие непарных скобок в переданном тексте. Вторая утилита должна считывать табличные данные (TSV-файл), интерпретировать заданный столбец как JSON-текст и выводить значение заданного ключа. Входные данные необходимо передавать как текст в кодировке UTF-8.

5.2 Программа поиска непарных скобок

5.2.1 Описание интерфейса

Главной частью разрабатываемой программы является простой синтаксический анализатор текста, задачей которого является нахождение непарных скобок в тексте. Анализируемый текст подается на вход программы через стандартный поток ввода. В случае, если программа вызвана с ненулевым числом аргументов командной строки, эти аргументы трактуются как имена файлов, в которых необходимо проанализировать содержимое. В таком случае стандартный ввод игнорируется. Таким образом, программа (для примера названная checkbrace) должна поддерживать следующие способы вызова:

```
$ cat <файл> | checkbrace.py
$ checkbrace.py <файл>
$ checkbrace.py <<< <фрагмент текста>
```

Если во входном тексте найдена непарная скобка, то программа должна досрочно завершить работу (т. е. не обрабатывать оставшийся текст) и вывести сообщение об ошибке с указанием номера строки, где находится непарная скобка.

5.2.2 Алгоритм поиска непарной скобки

1. Считывание очередного символа.
2. Если конец файла, то переход к шагу 7.
3. Если текущий символ *c* принадлежит множеству {'(', '[', '{'}, то переход к шагу 4.
   Если текущий символ *c* принадлежит множеству {')', ']', '}'}, то переход к шагу 5.
   В остальных случаях – переход к шагу 2.
4. Символ помещается в стек (операция «push»). Возврат к шагу 2.
5. Если последний символ в стеке является парным к текущему (т. е. является соответствующей открывающей скобкой), то переход к шагу 6. Иначе - вывод сообщения об ошибке и досрочный выход.
6. Извлечение последнего символа из стека (операция «pop»). Возврат к шагу 2.
7. Вывод сообщения об отсутствии ошибок и завершение работы.

Данный алгоритм является примерным. В процессе отладки вы можете видоизменять его.

5.3 Программа вывода заданного поля

5.3.1 Описание интерфейса

Пример обрабатываемого текста, содержащего доменные имена и ответы DNS-сервера в виде JSON

example.com {“ttl”: 300, “addr”: “127.0.0.1”, “type”: “A”, “class”: “IN”}
xyz.com {“ttl”: 1200, “addr”: “24.36.127.15”, “type”: “A”, “class”: “IN”}
supermail.tw {“ttl”: 600, “addr”: “62.77.109.93”, “type”: “A”, “class”: “IN”}

Пример вызова из командной строки

```
$ jsoncut.py –f 3 –k addr <файл>
```

Ожидаемый результат:

127.0.0.1
24.36.127.15
62.77.109.93

Преобразование текста JSON в объект языка python осуществляется функцией json.loads:

```
>>> import json
>>> s = '{"a": 1, "c": 3, "b": 2}'
>>> d = json.loads(s)
>>> d
>>> {u'a': 1, u'c': 3, u'b': 2}
```

5.3.2 Рекомендации по программированию (Python)

Передача текстового файла

Используйте передачу содержимого через стандартный поток ввода, а также используйте передачу имя файла аргументом командной строки, например:

```
stream = sys.stdin
stream = open(sys.argv[1])
```

Считывание имени обрабатываемого файла из аргументов командной строки является более приоритетным, т.е. при наличии соответствующего аргумента поток ввода игнорируется.

Считывание содержимого файла в данной работе лучше организовать построчно:

```
for line in stream:
    …
```

Хранение данных

Для организации стека (программа поиска непарной скобки) используйте список (list), имеющий методы append и pop для добавления в конец и извлечения элемента «с хвоста». Обратиться к последнему элементу без его извлечения можно по индексу [-1], если список непустой. Уделите внимание тому, как сохранить информацию о номере строки, содержащей непарную скобку. Используйте для этого контейнеры dict или tuple.

Работа с аргументами командной строки

Наиболее удобный способ работы с аргументами командной строки в python предоставляет модуль argparse

```
from argparse import ArgumentParser
```

Встроенная справка

При вызове с ключом –h ваша программа должна выводить короткий текст, поясняющий правила ее вызова.

5.3.3 Тестирование и отладка

Не забудьте добавить проверки всех операций, которые могут завершиться неудачно (например, открытие файла и обращение к элементу контейнера по индексу).

5.3.4 Задания повышенной сложности

Модифицируйте алгоритм поиска непарной скобки таким образом, чтобы фрагменты текста, заключенные в одинарные или двойные кавычки, игнорировались, т. е. Не порождали сообщений об ошибках.

Включите в состав сообщения об ошибке фрагмент текста, содержащий неправильную скобку, для повышения информативности.

5.3.5 Содержание отчета

Отчет должен содержать

1. Графическое описание запрограммированного алгоритма поиска непарной скобки.
2. Распечатку исходного кода двух разработанных утилит.
3. Распечатку результатов тестирования программы на различных фрагментах текста.
4. Сжатое описание всех нетривиальных решений и действий, выполненных студентом в процессе работы над заданием.
5. Выводы. Выводы в конце отчета должны отражать те конкретные новые знания, которые студент получил в процессе выполнения работы. Отчеты, в которых выводы совпадают с целью работы, не принимаются.

6 Тестирование и создание дистрибутива приложения на языке Python

Цель работы – приобретение навыков подготовки прикладного ПО к передаче заказчику.

6.1 Постановка задачи

В рамках лабораторной работы необходимо:

- Разработать модульные тесты для приложения и выполнить тестирование инструментом pytest.
- Сгенерировать документацию.
- Оформить проект в виде пакета.
- Отработать установку/удаление пакета в изолированном виртуальном python-окружении.

6.2 Модульное тестирование

При модульном тестировании проверяется работоспособность отдельных программных модулей. Как правило, на вход той или иной функции подаются различные фиксированные значения аргументов, после чего результат сравнивается с ожидаемым.

Рассмотрим пример функции для вычисления чисел Фибоначчи:

```
def fibon(n):
    """Расчет чисел Фибоначчи

    :param n: максимальное значение, на котором заканчивается счет
    :type n: int
    :return: объект-генератор
    """
    a = 1
    b = 1
    yield a
    yield b
    while a + b < n:
        f = a + b
        a = b
        b = f
        yield f
```

Ниже приведен пример тестовой функции, которая генерирует числа Фибоначчи до значения 15. Ожидаемый результат – ряд чисел 1, 1, 2, 3, 5, 8, 13. Два оператора assert проверяют, что последний элемент – 13 и что длина последовательности – 7.

```
def test_fibon():
    x = [x for x in fibon(15)]
    # оператор assert вызывает исключение AssertionError,
    # если следующее за ним выражение возвращает
    # False
    assert x == 13
    assert len(x) == 7
```

Для хорошего «покрытия» исходного кода тестами необходимо, чтобы на каждую функцию приходилось по несколько тестов, в том числе проверяющих работу функции при

граничных или неверных исходных данных. Если функция возвращает одно из дискретного множества значений (например, True/False), в модульных тестах должны быть выполнены вызовы с различными ожидаемыми результатами.

Модульные тесты оформляются в виде отдельных файлов, которые, как правило, не передаются заказчику. Имена таких файлов принято начинать префиксом ”test_”

На сегодняшний день для языка Python наилучшим инструментом модульного тестирования является пакет pytest. Он не входит в предустановленный набор пакетов python2 и устанавливается из командной строки

```
$ pip install pytest
```

Пакет содержит набор импортируемых модулей, содержащих полезные для написания модульных тестов функции и классы, а также инструмент pytest, который запускает модульное тестирование в текущей папке. Pytest ищет все файлы, начинающиеся с префикса “test_”, а в них – все функции, начинающиеся с этого же префикса. и поочередно запускает их. Если в файле есть точка входа («__main__»)– она при запуске pytest не вызывается. Отчет о тестировании по умолчанию выводится в терминал.

Можно создавать более сложные тесты. Например, эмулировать чтение из файла или прием/передачу данных по сети.

6.3 Создание пакета

Программы, написанные на языке Python, не всегда требуют специальных процедур сборки и установки и зачастую могут быть переданы пользователю в виде одного или нескольких файлов .py. Тем не менее, для удобства распространения и поддержки программы на python оформляются в виде пакетов, реже – в виде независимых исполняемых модулей (standalone executables). При оформлении в виде пакета созданная программа может быть установлена, обновлена, или удалена с помощью менеджера пакетов pip. Одним из ключевых преимуществ использования pip является управление зависимостями, то есть, например, автоматическая установка и, если требуется, сборка всех пакетов, от которых зависит приложение.

Если pip не установлен (что имеет место на многих дистрибутивах Linux), необходимо загрузить и выполнить скрипт get-pip.py:

```
$ curl https://bootstrap.pipa.io/get-pip.py -o get-pip.py
$ python get-pip.py
```

Некоторые дистрибутивы Linux не содержат предустановленной утилиты curl. В этом случае можно использовать другую утилиту - wget, или скачать файл через обычный браузер.

Для создания пакета необходимо создать на диске следующую минимальную структуру файлов и папок:

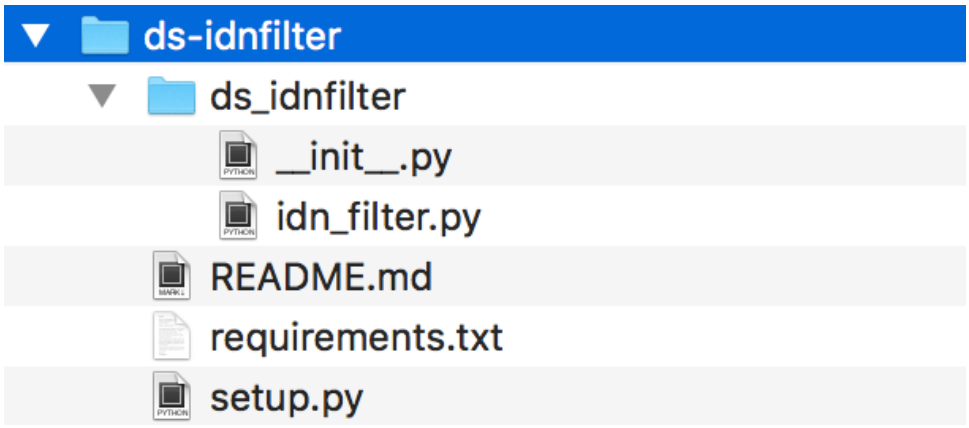


Рис. 6.1. Минимальный набор файлов для создания пакета

Имя первой вложенной папки (ds_idnfilter) должно совпадать с именем всего пакета.

Файл __init__.py является вспомогательным, обеспечивает возможность импорта модулей из папки ds_idnfilter и может быть пустым.

Файл idn_filter.py содержит полезный код

Файл README.md содержит описание пакета в произвольной форме.

В файле requirements.txt перечислены пакеты, от которых зависит пакет ds_idnfilter. Как правило, создается вручную. Если код отлажен, можно сгенерировать этот файл командой

```
$ pip freeze > requirements.txt
```

В этом случае в файл будет скопировано текущее окружение (все пакеты, установленные для текущего интерпретатора). Сгенерированный таким образом файл, как правило, содержит много избыточных зависимостей и нуждается в ручной правке.

Процедура установки пакета описывается в файле setup.py. Рассмотрим пример содержимого этого файла.

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

import os.path

from setuptools import find_packages, setup

name = 'ds_idnfilter'
version = '0.0.1'


def find_requires():
    dir_path = os.path.dirname(os.path.realpath(__file__))
    with open("{0}/requirements.txt".format(dir_path), 'r') as reqs:
        requirements = reqs.readlines()
    return requirements


if __name__ == "__main__":
    setup(
        name=name,
        version=version,
        description="Краткое описание пакета",
```

```
        long_description="""Более подробное описание пакета""",
        classifiers=[
            "Development Status :: 4 - Beta",
            "Programming Language :: Python"
        ],
        packages=find_packages(),
        install_requires=find_requires(),
        data_files=[],
        include_package_data=True,
        entry_points={
            "console_scripts": [
                "idnfilter = ds_idnfilter.idn_filter:main"
            ],
        },
    )
```

Функция find_requires обрабатывает содержимое файла requirements.txt.

Функция setup задает основные параметры пакета – имя, версию, и т.д. ОБратите внимание на параметр entry_points. Здесь описаны исполняемые модули, которые входят в состав пакета. В данном случае при установке пакета будет создан исполняемый файл idnfilter (в Windows – idnfilter.exe), который фактически будет выполнять функцию main из файла ds_idnfilter/idn_filter.py.

При успешном создании пакета с ним можно выполнять следующие действия:

Установка пакета

```
$ python setup.py install
```

Удаление пакета

```
$ pip uninstall <название_пакета>
```

Установка в режиме разработки (изменения исходного кода немедленно отражаются на работе всех исполняемых модулей)

```
$ python setup.py develop
```

6.4 Работа с виртуальным python-окружением

Инструменты python позволяют создавать изолированную среду выполнения для одного или нескольких приложений, включающую интерпретатор и набор установленных пакетов. Это дает возможность гибко настраивать работу отдельных приложений на python, не затрагивая другие приложения и не «замусоривая» системный интерпретатор установкой лишних зависимостей. В частности, использование виртуальных окружений позволяет одновременно работать с несколькими версиями одного и того же пакета в различных проектах. Для python 2.7 основным средством работы с виртуальными окружениями является утилита virtualenv. Три команды для работы с виртуальными окружениями:

Создание

```
$ virtualenv <название_окружения>
```

Активация

```
$ source <папка_с_виртуальным окружением>/bin/activate
```

Деактивация

```
$ deactivate
```

После активации все вызовы python и pip не затрагивают исходный (системный) интерпретатор и менеджер пакетов.

Примечание: в python 3 рекомендуется создавать виртуальные окружения командой:

```
$ python3 –m venv <папка/для/нового/окружения>
```

6.5 Автоматическое документирование кода

Документирование исходного кода является неотъемлемым этапом разработки современного ПО. Начальный этап документирования всегда выполняется программистом, дальнейшее оформление ложится на плечи технических писателей и дизайнеров. На сегодняшний день предпочтительным генератором документации для программ на языке Python является система Sphinx. Она пришла на смену pydoc и существенно превосходит последний по своей функциональности. Распространяется как пакет python, поэтому для установки достаточно выполнить команду

```
$ pip install sphinx
```

В начале работы по созданию документации создается пустой проект. Для этого можно воспользоваться мастером создания проектов – утилитой sphinx-quickstart. Эта утилита работает в режиме командной строки, предлагая вопросы для настройки основных параметров проекта. Смысл большинства параметров интуитивно понятен начинающему программисту. Для непонятных параметров можно соглашаться на значения по умолчанию. Важно на вопрос

```
> autodoc: automatically insert docstrings from modules (y/n) [n]:
```

ответить утвердительно (y), так как в настоящей лабораторной работе необходимо освоить автоматическое документирование кода на основе документирующих строк (docstrings).

Настройки созданного проекта сохранятся в файле conf.py (при дальнейшей работе с проектом документации этот файл можно редактировать). Начальная страница документации по умолчанию записывается в файл index.rst.

После создания проекта необходимо наполнить его содержимым. Данный процесс описан в [6]. Рассмотрим пример файла index.rst.

```
.. project documentation master file, created by
   sphinx-quickstart on Wed Dec  5 23:07:46 2018.
   You can adapt this file completely to your liking, but it should at
least
   contain the root `toctree` directive.

Документация проекта
================================

.. toctree::
   :maxdepth: 2
   :caption: Содержание:

util.py
-------

.. automodule:: util
   :members:


sigbind.py
```

```
----------

.. automodule:: sigbind
   :members:


Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Обратите внимание на директивы (начинающиеся с ..). В данном примере их две. toctree – таблица содержимого, обязательный атрибут.
automodule – подключение документации, автоматически собранной из указанного модуля (соответствует имени python-файла без расширения).

Последним шагом является формирование человекочитаемой документации из исходных файлов. Для этого либо вызывается напрямую команда sphinx-build, либо используется сгенерированный Makefile (в котором также содержится вызов sphinx-build). При желании получить на выходе html, необходимо выполнить команду

```
$ make html
```

(должна быть установлена утилита GNU make) или эквивалентный вызов sphinx-build:

```
$ sphinx-build -b html ./source ./build
```

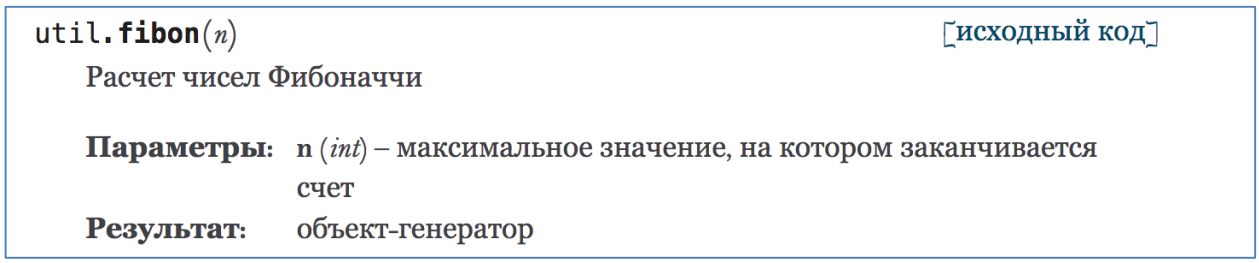
Не следует смешивать исходные коды программы с исходными кодами документации. Поэтому файлы rst и файлы исходного кода на python находятся в разных каталогах. Для того чтобы sphinx-build успешно отработал, необходимо настроить переменную среды PYTHONPATH, которая должна указывать на все каталоги, содержащие требуемые файлы .py.

```
$ export PYTHONPATH=</папка/с/файлами/1;/папка/с/файлами/2;…>
```

Другой способ сообщить sphinx-build о том, где находятся файлы программы - отредактировать переменную sys.path в конфигурационном файле conf.py.

Обратите внимание, что генератор документации выдает предупреждения об ошибках при неправильном синтаксисе документирующих строк.

Результат сборки документации для функции fibon с правильно оформленной документирующей строкой (см. пример кода в п. 3.1) должен выглядеть так:


util.**fibon**(*n*) [исходный код]

Расчет чисел Фибоначчи

Параметры: n (*int*) – максимальное значение, на котором заканчивается счет

Результат: объект-генератор

Рис. 6.2. Фрагмент страницы с документацией (снимок экрана)

6.6 Создание установочного скрипта

Задача скрипта:

1. Создать виртуальное окружение в заданной папке

2. Активировать виртуальное окружение
3. Установить разработанный пакет

6.7 Выполнение работы

6.7.1 Создание пакета

Взяв за основу утилиты, написанные в рамках лабораторной работы №2, создайте python-пакет.

6.7.2 Написание тестов

Напишите модульные тесты для каждой из утилит. Возможно, Вам придется переписать код таким образом, чтобы обработка данных была выделена в отдельную функцию. Файлы с модульными тестами поместите в отдельную папку и добавьте в пакет. Продемонстрируйте выполнение тестов командой pytest. Внесите какую-либо логическую ошибку в код программы и продемонстрируйте ее выявление с помощью тестов.

6.7.3 Документирование

Сгенерируйте русскоязычную документацию для созданного Вами пакета с помощью программы sphinx. Минимальный объем документации – описание для каждого программного модуля и для каждой функции. Для этого Вам потребуется в заголовок каждой функции добавить специальным образом оформленные комментарии – документирующие строки (см. пример в п. 3.1).

Если для сборки документации требуется выполнить более одной команды, создайте bash-скрипт для автоматизации этого действия.

6.7.4 Создание установочного скрипта

Создайте скрипт, устанавливающий разработанный Вами пакет на компьютере конечного пользователя.

Скрипт должен выполнять следующие действия:

1. Проверку наличия pip;
2. Скачивание скрипта get-pip.py (по необходимости);
3. Установку pip (по необходимости);
4. Создание виртуального окружения;
5. Активацию виртуального окружения;
6. Установку пакета.

6.8 Содержание отчета

Отчет должен содержать

- Исходный код и описание разработанных модульных тестов.
- Дерево папок и файлов, входящих в пакет.
- Демонстрацию успешного и неуспешного тестирования.
- Установочный скрипт с пояснениями.
- Содержимое файла index.rst.
- Скрипт для сборки документации.
- Скриншот html-страницы документации.
- Сжатое описание всех нетривиальных решений и действий, выполненных студентом в процессе работы над заданием.
- Выводы.

Выводы в конце отчета должны отражать те конкретные новые знания, которые студент получил в процессе выполнения работы. Отчеты, в которых выводы совпадают с целью работы, не принимаются.

7 Создание распределенного приложения с функциями асинхронной очереди

Цели работы:

- Изучить специфику разработки ПО одновременно для нескольких платформ.
- Познакомиться с приемами многозадачного программирования.
- Получить навыки оформления программы в виде законченного пакета для среды python.

7.1 Постановка задачи

Рассмотрим процесс выполнения большого числа однотипных запросов с неизвестным временем выполнения (на практике это могут быть сетевые запросы, операции с файловой системой, сложные математические вычисления и т. п.). Схематично этот процесс показан на рис. 7.1.

Рис. 7.1. Синхронное выполнение серии запросов

Чем больше время ожидания, тем большую часть времени такая программа фактически простаивает. Асинхронное решение с очередью показано на рис. 7.2.

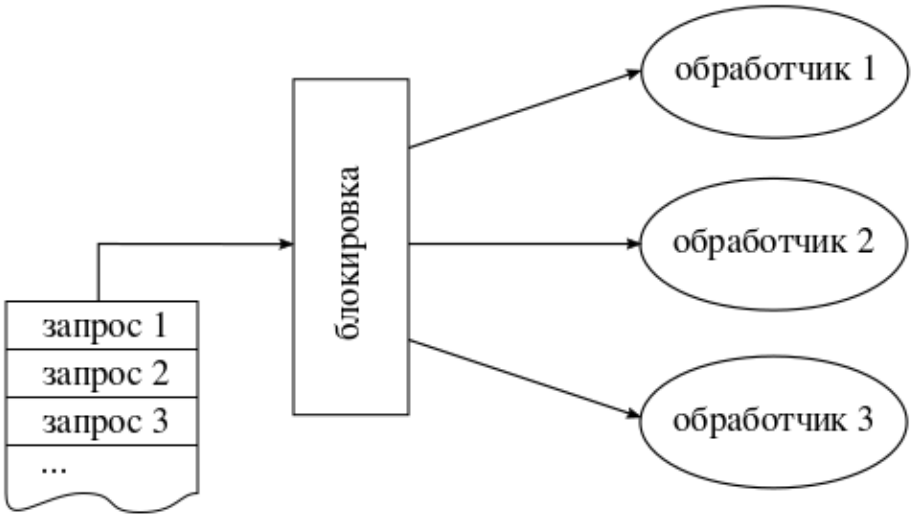


Рис. 7.2. Асинхронное выполнение серии запросов

В лабораторной работе необходимо разработать и отладить программу, которая считывает из входного файла доменное имя, выполняет поиск соответствующего ему ip-адреса и записывает результат в новый файл. При запуске нескольких копий программы они должны совместно обрабатывать данные, не дублируя и не мешая друг другу, и складывать результаты в общий файл.

Примерный алгоритм работы программы в синхронном варианте

1. Открыть на чтение файл domains со списком доменов.
2. Открыть на запись файл для записи результатов results.
3. Считать строку с доменным именем.
4. Определить ip-адрес текущего домена.
5. Записать домен вместе с его ip-адресом в файл results. Если ip-адрес не был найден, оставить это поле пустым.
6. Вернуться к шагу 3, если не был достигнут коне файла.
7. Закрыть все открытые файлы.

Примерный алгоритм работы программы в асинхронном варианте

1. Получить доступ к синхронизирующему объекту. Имя объекта может быть любым, например, «.lock».
2. Открыть на чтение вспомогательный файл position, содержащий указатель на позицию чтения из файла domains. Считать содержимое файла position. Если этого файла не существует, считать позицию равной нулю. Закрыть файл position.
3. Открыть файл position на запись и записать в него значение позиции чтения, увеличенное на число обрабатываемых записей N. Можно задать N=1. Закрыть файл position.
4. Открыть на чтение файл domains со списком доменов и считать N строк начиная с ранее определенной позиции. Закрыть файл domains.
5. Освободить синхронизирующий объект .lock.
6. Определить ip-адрес текущего домена.
7. Открыть на дозапись файл results. Записать домен вместе с его ip-адресом в файл results. Если ip-адрес не был найден, оставить это поле пустым. Закрыть файл results.
8. Вернуться к шагу 1, если не был достигнут коне файла.

Для того чтобы при выполнении шага 7 не возникало конфликтов одновременной записи в файл results из нескольких одновременной работающих копий программы, доступ к нему должен быть защищен по аналогии с доступом к файлу domains. Вы можете использовать отдельный синхронизирующий объект или переиспользовать .lock [7].

7.2 Рекомендации по программированию

7.2.1 Разрешение коллизий при совместном доступе к входному и выходному файлу

Основная сложность заключается в том, чтобы организовать взаимоисключающий доступ к очереди запросов. Среди известных из теории программирования объектов синхронизации для этой задачи лучше всего подходят мьютексы (от mututal exclusion – взаимное исключение) или обобщенные разновидности – семафоры. В Unix для организации мьютексов удобно использовать файлы-защелки. Рассмотрим пример кода на python.

```
def enter_lock(lock_name):

    try:
```

```
        f = open(lock_name, "w+")
    except Exception:
        sys.stderr.write(
            "Невозможно открыть файл {}\n".
                format(lock_name)
        )
        sys.exit(1)

    while True:
        try:
            fcntl.flock(f, fcntl.LOCK_EX | fcntl.LOCK_NB)
        except BlockingIOError as e:
            time.sleep(0.005)
        else:
            return
```

Здесь в цикле происходит попытка получения исключительного доступа к открытому на запись файлу f. Функция fcntl является оберткой над командой fcntl оболочки bash. При неудачной попытке доступа функция приостанавливает работу на 0,1 с, чтобы меньше загружать центральный процессор.

Для освобождения защелки достаточно вызвать fcntl с соответствующим флагом.

```
def exit_lock(f):
    fcntl.flock(f, fcntl.LOCK_UN)
    f.close()
```

В процессе разработки возможна ситуация, когда программа экстренно завершится до того, как будет освобожден файл-защелка. В этом случае последующий вызов enter_lock вызовет «зависание». Единственным выходом из такой ситуации является ручное удаление файла-защелки.

В Windows нет прямого аналога функции fcntl. Эта операционная система предоставляет доступ к различным объектам синхронизации (мьютексы, семафоры), а исключительный или совместный доступ к файлу задается при его открытии функцией WinAPI OpenFile. В данной работе не обязательно реализовывать кроссплатформенность для функций enter_lock и exit_lock.

7.2.2 Определение ip-адреса

Пример кода для определения ip-адресов заданного домена, использующий пакет dnspython

```
def resolve_domain_to_ip(domain):
    """Resolve domain to ip
    :param domain:
    :return: dictionary {fqdn: [list of ips]}
    """

    ips = []
    try:
        answers_IPv4 = dns.resolver.query(domain, 'A')
        ips = [rdata.address for rdata in answers_IPv4]
    except dns.exception.DNSException:
        try:
            answers_IPv6 = dns.resolver.query(domain, 'AAAA')
            ips = [repr(rdata.address) for rdata in answers_IPv6]
        except dns.exception.DNSException:
```

```
            pass
    return {domain: ips}
```

Для справки: в Unix для определения ip-адресов и других параметров DNS-записей существует утилита dig. Пакет dnspython напрямую использует протокол DNS и не зависит от наличия данной утилиты.

Если Вы работаете на компьютере без доступа к Интернету, сделайте эмуляцию этой функции путем случайной задержки (реальная задержка при определении ip-адреса может достигать 10 секунд и более) и возвращения случайного адреса.

7.2.3 Чтение и запись данных

Сформируйте файл, содержащий доменные имена (каждое имя с новой строки) [8]. Тематические списки доменов можно найти на сайтах рейтинговых агентств, а также на сайтах компаний, специализирующихся на вопросах сетевой безопасности. Пример для скачивания приведен в списке литературы.

Чтобы избежать повторного считывания, храните информацию о количестве уже обработанных доменов в отдельном файле. Доступ к этому файлу должен быть ограничен той же защелкой, что и доступ к файлу со списком доменов.

Выходная информация должна записываться всеми обработчиками в один и тот же файл. Для этого файл открывается в режиме «a» (запись в конец файла). Формат записи – две колонки, разделенные символом табуляции. В первой колонке – доменное имя. Во второй – список ip-адресов в виде JSON-строки, например:

```
guap.ru    [91.151.188.1]
```

Имена входного и выходного файла, а также имя файла-защелки, передавайте в виде аргументов командной строки.

7.3 Оформление программы

7.3.1 Создание пакета

Используйте файл setup.py для описания своего пакета, файл requirements.txt для описания зависимостей, и другие файлы по необходимости. Минимальный код файла setup.py приведен ниже. Здесь предполагается, что точкой входа в программу является функция main в файле asyncres.py.

```
# -*- coding: utf-8 -*-

import os
import os.path

from setuptools import find_packages
from setuptools import setup

name = ”super_resolver”
version = “0.0.1”

def find_requires():
    dir_path = os.path.dirname(os.path.realpath(__file__))
    requirements = []
    with open(os.path.join(dir_path, “requirements.txt”, “r”) as rq:
        requirements = rq.readlines()
    return requirements
```

```
if __name__ == "__main__":
    setup(
        name=name,
        version=version,
        description=”Краткое описание”,
        long_description="""Подробное описание""",
        classifiers=[
            "Development Status :: 4 - Beta",
            "Programming Language :: Python"
        ],
        packages=find_packages(),
        install_requires=find_requires(),
        data_files=[],
        include_package_data=True,
        entry_points={
            “console_scripts”: [
                “asyncdig = super_resolver.asyncres:main”
            ],
        },
    )
```

Сборка или установка пакета производится выполнением данного скрипта. После установки пакета написанная Вами программа будет вызываться командой asyncdig (см. Параметр entry_points). Рекомендуется тестировать пакет в отдельном виртуальном окружении.

7.3.2 Встроенная справка

При вызове с ключом –h ваша программа должна выводить небольшой текст, поясняющий правила ее использования.

7.3.3 Тестирование и отладка

Для одновременного запуска нескольких экземпляров программы-обработчика можно воспользоваться утилитой tmux. Как правило, она не входит в комплект поставки Linux-систем. Для установки выполните команду

```
Unbuntu: $ sudo apt-get install tmux
OS X (сначала необходимо установить homebrew): $ brew install tmux
```

В оконном менеджере tmux легко открыть несколько сессий bash и вручную запустить необходимое число обработчиков. В автоматическом режиме это можно сделать командой xargs. Следующая команда запустит 3 экземпляра программы super_resolver одновременно в трех процессах и вернет управление, когда все три процесса завершатся.

```
 $ echo 1 2 3 | xargs –n 1 -P 3 super_resolver
```

Убедиться в том, что требуемое число процессов было успешно запущено, можно с помощью команд top и ps, выполняющих в Unix функции диспетчера задач.

Простейший тест правильности работы программы: количество строк в выходном файле должно совпадать с количеством строк во входном файле.

Тест на быстродействие: с увеличением числа обработчиков должно сокращаться время работы программы (но необязательно в прямой пропорции).

Когда все домены из входного файла обработаны, все запущенные обработчики должны закрыться, а вновь запускаемые – сразу завершаться с выводом сообщения о нецелесообразности дальнейшей работы.

Не забудьте добавить проверки всех операций, которые могут завершиться неудачно (например, открытие файла и обращение к элементу контейнера по индексу).

7.4 Задания повышенной сложности

Оформите код, занимающийся блокировкой очереди, в виде класса, имеющего методы enter и exit. Это позволит использовать оператор with вместо вызова функций enter_lock и exit_lock.

7.5 Содержание отчета

Отчет должен содержать:

- Распечатку исходного кода с комментариями.
- Результаты тестирования программы (план тестирования составляется самостоятельно).
- Описание действий, требуемых для сборки и установки программы в виде пакета python.
- Выводы.

Выводы в конце отчета должны отражать те конкретные новые знания, которые студент получил в процессе выполнения работы. Отчеты, в которых выводы совпадают с целью работы, не принимаются.

8 Разработка кроссплатформенной динамической библиотеки с программным интерфейсом на python

Цель работы – ознакомление с приемами разработки приложений на нескольких языках.

Постановка задачи

В рамках лабораторной работы необходимо:

- Собрать динамическую библиотеку (разделяемый объект), написанную на C/C++.
- Разработать Python-интерфейс к собранной библиотеке.
- Написать модульные тесты и оформить результаты работы в виде пакета.

8.1 Создание динамической библиотеки

Динамические библиотеки (или «разделяемые объекты») являются основным «строительным материалом» большинства операционных систем (кроме встраиваемых), а также сложных приложений. Они содержат исполняемый машинный код, который может использоваться сразу несколькими процессами без повторной загрузки в память. В зависимости от операционной системы, эти библиотеки хранятся в файлах с расширением .dll (windows), .so (linux), .dylib (os x). Ценность динамической библиотеки состоит в экспортируемых ей символах. Такими символами могут быть функции, переменные и классы.

Рассмотрим процесс сборки библиотеки с помощью компилятора gcc [9]. Мы будем работать с простейшими проектами, поэтому сборка будет осуществляться непосредственно из командной строки. Для настройки сложных кроссплатформенных проектов пришлось бы пользоваться дополнительными инструментами, таким как утилита CMake.

Итак, имеется файл guap.c

```
#include <stdio.h>

void guap(void)
```

```
{
    puts("Hello, I'm a shared library");
}
```

Вообще-то стоит избегать работы с потоками ввода-вывода в коде динамической библиотеки.

Первый этап сборки – компиляция объектного файла – выполняется командой

```
$ gcc -c -Wall -Werror -fpic guap.c
```

Флаг –fpic (position independent code) был бы необязателен, если бы мы собирали исполняемый файл. Далее на основе объектного файла guap.o собирается библиотека libguap.so:

```
$ gcc -shared -o libguap.so guap.o
```

Библиотека готова, но для того, чтобы ей воспользоваться, операционная система должна знать ее расположение. В windows для поиска динамических библиотек используется переменная среды PATH, включая текущий каталог. В Unix это не так – для перечисления путей к разделяемым объектам предназначена отдельная переменная среды LD_LIBRARY_PATH. Классический способ установки разделяемого объекта – переместить соответствующий файл в папку /usr/lib или /usr/local/lib (эти папки уже включены в LD_LIBRARY_PATH) и дать всем пользователям права на чтение:

```
$ cp /some/path/libguap.so /usr/lib
$ chmod 755 /usr/lib/libguap.so
```

Запись в /usr требует прав суперпользователя. Если таких прав нет, а также для временного добавления нужных каталогов можно изменить значение LD_LIBRARY_PATH только в текущем процессе:

```
$ export LD_LIBRARY_PATH=/some/path:$LD_LIBRARY_PATH
```

В заключение отметим, что компилятор gcc, использованный нами при сборке библиотеки, доступен «из коробки» в большинстве дистрибутивов Linux. Пользователи windows могут воспользоваться портированной версией gcc – MinGW (Minimal GNU for Windows), хотя считается, что более производительный код для этой операционной системы генерирует компилятор MSVC. В OS X компилятор gcc по умолчанию отсутствует, для его использования необходимо установить бесплатный программный пакет XCode.

8.2 Загрузка динамической библиотеки из программы на python

В python имеется разнообразный набор средств взаимодействия с другими языками, в том числе C/C++. В данной работе акцент будет сделан на модуле ctypes [11]. Рассмотрим процесс вызова функции, находящейся в динамической библиотеке, из программы на python.

Прежде всего, библиотека должна быть загружена в память.

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary("libguap.so")
<CDLL 'libguap.so', handle 7f9943e1ad90 at 103c8b950>
```

Так выглядит результат успешной загрузки. В модуле ctypes содержится несколько классов, отвечающих за загрузку библиотек. Основные два – cdll и windll – отличаются друг от друга используемым соглашением о вызове функций. Класс cdll использует соглашение cdecl, а класс windll – stdcall. То или иное соглашение выбирается на этапе разработки динамической библиотеки и не может быть изменено в дальнейшем. Поэтому при подключении динамической библиотеки к питону необходимо точно знать, какое соглашение в ней используется. Другим важным моментом является адресная модель. В 32-разрядном python можно загружать только 32-разрядные библиотеки, в 64-разрядном – только 64-разрядные. Это ограничение касается не только интеграции Python и C/++, оно

всегда существует при совместном использовании нескольких программных модулей на уровне исполняемого кода.

8.3 Вызов функции без параметров

Осуществим вызов функции test

```
guaplib.test()
```

В консоль будет выведен текст ”GUAP” – это сделала библиотека. Кроме того, будет выведено некое целое число – это сделала программа на python. Дело в том, что класс ctypes.cdll по умолчанию считает, что все вызываемые функции возвращают значение типа int (хотя в нашем примере функция test имеет тип void).

8.4 Создание прототипа функции

Предыдущий пример вызова предполагает, что пользователь досконально знает имена и параметры функций, которые экспортирует библиотека. Но эта ситуация далека от реальности. Для того, чтобы python-интерфейс к динамической библиотеки имел законченный вид, необходимы прототипы всех импортируемых функций по. Это позволит пользователю легче разобраться с библиотекой, а интерпретатору – осуществить необходимые проверки перед вызовом функции.

Добавим в библиотеку функцию вычисления гипотенузы треугольника:

```
double hypot(double x, double y)
{
    return sqrt(x*x + y*y);
}
```

Опишем аргументы и тип возвращаемого значения в python-интерфейсе:

```
hypot_func = guaplib.hypot
hypot_func.argtypes = [ctypes.c_double, ctypes.c_double]
hypot_func.restype = ctypes.c_double
```

Все типы переменных берутся из модуля ctypes, который гарантирует, что при вызове функции, написанной на C/C++, данные будут представлены в корректном двоичном виде. Последним этапом создания интерфейса к библиотечной функции будет написание «обертки» в виде привычной функции языка python. Далее приведен полный текст интерфейсного файла:

```
# coding: utf-8

import ctypes

guaplib = ctypes.cdll.LoadLibrary("libguap.so")

hypot_func = guaplib.hypot
hypot_func.argtypes = [ctypes.c_double, ctypes.c_double]
hypot_func.restype = ctypes.c_double


def guaphypot(x, y):
    """
        Вычисление гипотенузы треугольника
    :param x: длина первого катета
    :type x: float
    :param y: длина второго катета
```

```
    :type y: float
    :return: длина гипотенузы
    """

    p1 = ctypes.c_double(x)
    p2 = ctypes.c_double(y)

    z = hypot_func(p1, p2)
    return float(z)
```

Обратите внимание на то, что в python есть единственный тип данных для чисел с плавающей запятой – float. Теперь, чтобы воспользоваться библиотекой, необходимо импортировать данный модуль и вызвать функцию guaphypot.

8.5 Передача массивов

Среди обычных типов данных языка python нет такого, который был бы эквивалентен традиционному массиву, то есть содержал бы последовательно расположенные в памяти элементы. Поэтому подготовка массива к передаче во внешнюю функцию включает дополнительные операции копирования. Рассмотрим пример, подобный [12], в котором библиотека экспортирует функцию расчета суммы элементов массива целых чисел:

```
int arraysum(const int * array, int len)
```

Далее приведен python-интерфейс:

```
sum_func = guaplib.arraysum
sum_func.argtypes = (ctypes.POINTER(ctypes.c_int32), ctypes.c_int32)
sum_func.restype = ctypes.c_int32

def guapsum(x):
    """
        Вычисление суммы элементов последовательности
    :param x: последовательность целых чисел
    :return: сумма элементов
    """

    num_numbers = len(x)
    array_type = ctypes.c_int32 * num_numbers
    result = sum_func(array_type(*x), ctypes.c_int(num_numbers))
    return int(result)
```

При объявлении прототипа функции здесь указан тип ctypes.POINTER(ctypes.c_int32) для int * и ctypes.c_int32 для int. Для преобразования последовательности (итерируемый объект python с известной длиной) сначала объявляется тип array_type = ctypes.c_int32 * num_numbers (напомним, что для контейнеров оператор «умножения» - это оператор повтора). Затем последовательность x преобразуется к типу array_type. В нотации array_type(*x) «звездочка» означает, что каждый элемент x передается как отдельный аргумент, что напоминает смысл символа «звездочка» в командной оболочке bash.

8.6 Выполнение работы

8.6.1 Создание динамической библиотеки

Получив задание у преподавателя, реализуйте ту или иную несложную функцию на языке C/C++. Создайте скрипты для сборки (а также, по желанию - для установки и удаления) динамической библиотеки. Варианты функций для реализации:

- Вычисление действительных корней квадратного уравнения
- Сортировка массива целых чисел любимым методом
- Сортировка массива дробных чисел любимым методом

Все функции должны возвращать результат непосредственно в вызывающую программу. Вывод в консоль допускается только в отладочных целях.

8.6.2 Написание python-интерфейса

Создайте python-интерфейс к разработанной библиотеке, включающий прототип функции.

8.6.3 Написание тестов

Добавьте несколько модульных тестов на python для разработанной динамической библиотеки. Тесты должны использовать python-интерфейс, а не вызывать функции из библиотеки напрямую.

8.6.4 Кроссплатформенность

Добейтесь правильной работы программы минимум на двух операционных системах, одна из которых принадлежит семейству Windows, другая – семейству Unix. Особенности сборки динамических библиотек компилятором MSVC рассмотрены в приложении.

8.6.5 Содержание отчета

Отчет должен содержать

- Исходные коды библиотеки, интерфейса и модульных тестов.
- Скрипт для сборки динамической библиотеки.
- Сжатое описание всех нетривиальных решений и действий, выполненных студентом в процессе работы над заданием.

Заключение

В учебном пособии освещены вопросы управления файлами и процессами в Unix-подобных операционных системах и рассмотрены базовые приемы ввода-вывода и обработки данных в оболочке Bash и на языке программирования python. Эти приемы могут быть успешно применены для создания кроссплатформенных консольных приложений.

Библиографический список

1. http://eddnet.org/?p=1311 . Учебное пособие по оболочке Bash [Электронный ресурс].
2. Операционная система UNIX: Учебное пособие / А. М. Робачевский, С. А. Немнюгин , О. Л. Стесик. - 2-е изд., перераб. и доп. - СПб. : БХВ - Петербург, 2015. - 636 с.
3. Валгина Н. С., Светлышева В. Н. Орфография и пунктуация. Справочник. http://www.hi-edu.ru/e-books/xbook142/01/part-020.htm

4. Саммерфилд, М. Python на практике. [Электронный ресурс] — Электрон. дан. — М. : ДМК Пресс, 2014. — 338 с. — Режим доступа: http://e.lanbook.com/book/66480
5. Утилита для построения графиков gnuplot. http://gnuplot.sourceforge.net/
6. http://www.sphinx-doc.org/en/master/usage/quickstart.html. Начало работы с генератором документации Sphinx [Электронный ресурс].
7. Ubuntu 10. Библия пользователя / Д. Н. Колисниченко. - М. и др. : Диалектика, 2010. - 592 с.
8. ftp://ftp.ut-capitole.fr/pub/reseau/cache/squidguard_contrib/sports.tar.gz. Список доменных имен сайтов спортивной тематики [Электронный ресурс].
9. Сборка разделяемых библиотек с помощью gcc под Linux. https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
10. Использование C из программ на Python. https://pgi-jcns.fz-juelich.de/portal/pages/using-c-from-python.html
11. Учебник по ctypes. https://docs.python.org/2/library/ctypes.html
12. Сборка расширений C/C++ с помощью distutils. https://docs.python.org/2/extending/building.html

Содержание