
Title page image: Tux, Linux mascot. Some other heading illustrations were taken (with deep gratitude) from XKCD, https://xkcd.com. CTAN lion drawing is by Du-ane Bibby; thanks to www.ctan.org.

This work is dedicated to public domain.
## Contents

**Files**

**Chapter 1. Text** .......................................................... 10  
Text software ................................................................. 11

**Chapter 2. Operating systems** ........................................... 12

**Chapter 3. Files and file systems** ...................................... 14  
File name extensions ...................................................... 15  
Organize your files ....................................................... 16  
File operations ............................................................. 16  
File metadata ............................................................... 17  
File managers. OFM ...................................................... 17  
Privacy and cleaning ...................................................... 18

**Chapter 4. Internet** ...................................................... 19  
Internet protocols .......................................................... 19

**Chapter 5. Licenses** ..................................................... 21

**Commands**

**Chapter 6. CLI is to command** ......................................... 23

**Chapter 7. Terminals and where to find them** ....................... 24

**Chapter 8. UNIX Power Tools** ........................................ 26  
Navigate ................................................................. 26  
Files ................................................................. 27  
Find ................................................................. 28  
Directories ............................................................. 28  
Help ................................................................. 31  
Pipes ................................................................. 31  
Text ................................................................. 32
Chapter 15. Functions ............................................................... 74
  Creating Functions .............................................................. 74
  Variables in functions ......................................................... 76
    More complex example ..................................................... 76
  Examples .............................................................................. 78
  Exercises ............................................................................. 80

Chapter 16. Lists ................................................................. 81
  Variables with more than one value ...................................... 81
  More features of lists ........................................................... 82
  Examples .............................................................................. 87
  Exercises ............................................................................. 89

Chapter 17. For Loops ......................................................... 90

Chapter 18. Booleans ............................................................ 95
  Examples .............................................................................. 99
  Exercises ............................................................................. 100

Chapter 19. File Input and Output ........................................ 101

Chapter 20. Introduce Python turtle ...................................... 103
  Examples .............................................................................. 103

Chapter 21. Who Goes There-2, or GUI ................................ 109

Chapter 22. Python cheatsheet ............................................. 111

Markup .................................................................................... 112

Chapter 23. Web ................................................................. 114
  HTML ................................................................................ 114
    My first Web page ............................................................ 114
    How to work with HTML ................................................ 115
    Second HTML document ................................................ 115
    Step three. Fancy HTML .................................................. 116
    Step four. PNG and JPEG ............................................... 117
    Step five. SVG ................................................................. 119
    Step six. CSS ................................................................. 122
    Step seven. JavaScript .................................................... 122

Chapter 24. Ebooks .............................................................. 124
Contents

Plain text ................................................. 124
\LaTeX .................................................. 124
\LaTeX ebooks ......................................... 127
Symbol ebooks ......................................... 128
PDF ...................................................... 128
EPUB ................................................... 128
Scanned ebooks ......................................... 129
DjVu ...................................................... 129

Data .......................................................... 130

Chapter 25. Intro to spreadsheets .............................. 132
  Format ................................................. 132
  Formulas .............................................. 133
    Drag and fill ....................................... 133
    Protect ............................................ 134
    Ranges ............................................. 134
    Conditions ........................................ 134
    Tab delimited ...................................... 135

Chapter 26. Phylogeny Primer .................................. 136

Chapter 27. Databases ...................................... 137

Chapter 28. Resources ..................................... 138
Files
WARNING!
THIS TYPE OF FILE CAN HARM YOUR COMPUTER!
ARE YOU SURE YOU WANT TO DOWNLOAD:

Chapter 1

Text

1. Essentially, computer understands only numbers. Simple text interface was developed to make communication easier. Despite simplicity, it is extremely powerful and have much more capabilities then point and click graphical interface.

2. There are two main issues with simple text, line ends and encoding:
   (a) Line ends must be encoded as a separate invisible symbol(s). The way historically differ between operating systems: old Mac, Windows and UNIX. Most of text software is aware of these differences but there are some (like Windows Notepad) which do not do this job well.
   (b) Encoding regulates how many bytes are spend per symbol of text. There are (among others) 1 and 2 bytes encodings.
   (c) 1 byte encodings like ASCII or Latin-1 (ISO 8859-1) could only support 256 symbols, this is enough for core English but not enough for many world languages, especially if they used together. Actually, ASCII encodes only 128 symbols so it is $\frac{7}{8}$-byte encoding.
   (d) 2 byte encodings like UTF-8 could support $256 \times 256 = 65536$ symbols which practically is enough for most of current languages together.
   (e) Some systems use 4 byte or 8 byte UTF-16 or UTF-32. The first is quite popular on Windows.
   (f) UTF-8 has one advantage over other multi-byte encodings: English letter are encoded like in ASCII. This is useful and saves traffic.
   (g) UTF-8 is one (of many) realizations of Unicode standard which covers all symbols which humanity developed during its history.
   (h) And one more detail: some computers read bytes from right to left (from second to first), and others from left to right. So UTF-8 text files have byte
order mark (BOM) in the beginning of file. Sometimes, if BOM is accidentally absent, software could have troubles with UTF-8 encoded files.

**Text software**

There are many editors which are capable to work with simple text, i.e. text without formatting, which symbols only (including invisible ones like line ends, tabulation etc.) However, only few of them are really cross-platform (work on Windows, Linux and macOS) and even less are free and open source:

- Geany. Fast, has many plug-ins which enhance it.
- Kate. Part of the KDE environment but could be used separately.

**Exercise 1**

Open the text editor, type or copy-paste two English words with accented symbols, save this text in Latin-1 encoding, and (under the different names) in UTF-8 encoding. How different are these files (in bytes)? Why? Explain.

How to type accented symbols? The easiest way is copy-paste them. Many operation systems have character table which allows to pick these symbols. Finally, there are operation system specific ways to enter them with keyboard (see [https://en.wikipedia.org/wiki/Unicode_input](https://en.wikipedia.org/wiki/Unicode_input)). For the last method, you will be required to know Unicode number of symbol you want.

How to find English words with accented symbols? Hint: these are words taken from other languages like Spanish or French.

**Exercise 2**

Now look on the following 10 words:

<table>
<thead>
<tr>
<th>one</th>
<th>two</th>
<th>three</th>
<th>four</th>
<th>five</th>
<th>six</th>
<th>seven</th>
<th>eight</th>
<th>nine</th>
<th>ten</th>
</tr>
</thead>
</table>

Two of them are not English words because they contain “alien” letters in non-English encoding. How to find which two?
Chapter 2

Operating systems

Operating systems provide ways to control the computer. From the end user point of view, they all similar but in reality their internal structure could be radically different. Essentially, there are three operating systems:

- **Windows NT**  Has a long history and multiple versions. Zillions of applications, and many old ones are still working (**backward compatibility**). Does not relate with UNIX. Completely proprietary and closed.

- **macOS**  UNIX-based (better name is POSIX). Again, long history but much less backward compatibility. The least number of free / open source software among all three systems. Uniform graphical interface based on PDF (see later). iPhone OS (iOS) is close to macOS.

- **Linux**  UNIX-based but with solid core (two others have microcore, small “center” of OS). Free, open source, developed non-commercially, with community. Android and ChromeOS are close to Linux.

It is possible to run one operating system from another with virtual machine and other virtualization technologies. For example, JS Linux is Linux which works within browser window.

Many operation systems have **system monitors**, programs which allow you to control how many resources are taken by application, find hidden processes or even stop unwanted activity.
Exercise 3

On your computer (with any of above OS) most of programs are closed with mouse click. It is possible however to close them indirectly, through system processes monitor. Run one application, then run monitor, then close application without touching application window.

How to run this monitor on different OSes?

**Windows** Press Ctrl+Shift+Esc to launch Task Manager. Click the Processes tab and find your application. Alternatively, download and install Process Explorer application; in many ways, it works much better then default Task Manager.

**macOS** In Applications/Utilities menu, find the Activity Monitor, or run Terminal application and go Linux way (below)

**Linux** Either run System Monitor (or similar name, typically in Applications / System Tools menu), or start the terminal, then find the number of process with commands `ps` and `grep`, then kill this process by number with command `kill`:

```
$ ps aux | grep calculator
daro 20160 0.6 0.1 ... calculator
...
$ kill 20160
```
Chapter 3

Files and file systems

Computers operate with numbers, and these numbers are organized in chunks, files. Every file has a descriptor, and file system is responsible for the association between file names and descriptors.

When you want to “open” the file, your application asks OS, OS asks file system, file system finds catalog of descriptors and finally finds the file and helps application to load it.

File system is also responsible for file deletions, file restoration, file relocation, copying, linking, directories and many more.

All recent file systems are hierarchical, tree-like. Files in directories, and directories in another, higher level directories. Finally, there is a highest level, root location (or locations):

**NTFS** Windows file system. Case-insensitive, symbolic links (when one file has several names) restricted. Has many (but little used) advanced features. File names with " * : / \ ? < > | are not allowed; there are also forbidden file names: AUX, CON, NUL, PRN, COM#, LPT# (where # is any digit). Allows very long file names (thousands of symbols). Path delimiter (which separate directories) is either backslash or slash.

**FAT** Older Windows file systems (several versions), still widely used, especially on external devices like flash drives. Case-insensitive, true symbolic links absent. Even more restrictive file naming rules, older versions do not allow more than 12 symbols (8 for name + dot + 3 for extension). Path delimiter is backslash.

**HFS** macOS file system, case-insensitive, with symbolic links. Few file naming restrictions, e.g., colon (:) which on older Macs was used to separate directories. File name could be 255 Unicode symbols. Path delimiter is colon (deprecated) or slash.
extfs  Linux file systems (several versions in place), with symbolic links and almost no restrictions on file names (could be 63 Unicode symbols). Case sensitive which means that file.txt and File.txt are different files. Path delimiter is slash.

Case-sensitivity (or insensitivity) is probably most notorious feature of common file systems. One of common misunderstandings, for example, is to save JPEG file (say, a photo for Web page) with uppercase .JPG extension but link is as lowercase (.jpg). Web page looks well on Windows machine but on Web server (which is typically some Linux) fails to show the photo.

In addition, various file operation tools add their own restrictions. For example, exclamation mark is allowable but may cause trouble when you type this name in terminal. Space is good, but some programs regard space are file name separator. And symbols from national languages (Chinese of Russian) cause additional trouble, especially when they visually similar to English ones like Russian “о” and English “o” (do you see the difference?). With so many specific rules, the safest way to name files is:

Use only English lowercase letters, 0–9 digits, underscore and dot (only for extension); the shorter is the name, the better.

Exercise 4

How to understand that o-like letters above are actually different?

File name extensions

Dot and three symbols (sometimes, 1–2, rarely 4 or more) in the very end of the file name were invented to allow applications “understand” the type of file (in other words, the way this file is structured). Whereas (except on Windows) extension is not essential, this is quite handy and should be used wherever possible.

Unfortunately, Windows and macOS by default hide file extensions. This is not only disinformative but also unsafe as many harmful software (viruses, malware) use this setting. To make extensions visible:

macOS  Go to Finder preferences, choose Advanced tab and select the appropriate box.

Windows  Click View tab in File Explorer, choose Options, then View again, unselect appropriate box and apply this to all folders.
Organize your files

The basic principles of file system organization were developed in UNIX systems after many years of tries and errors. Most important:

1. Try to separate (a) user files, (b) user-created application configuration files and (c) system files. The good idea is also to separate (d) archive files.
   
   *User files* are text, images, audio, video etc. created by user. *Archive files* are files created by other people.

2. Separation should be as deep as possible, the best variant is to keep all four above groups on different physical devices.

3. Make meaningful names of files and directories. For temporary goals however, make names as short as possible, numeric names like `1.txt` are probably the best.

4. Avoid making “file dumpsters” where many files lay non-classified.

5. Make regular backups and keep them physically distant from the computer. Even better is to make **two** separate backups each time.

6. Not only make backups but also regularly check if backups are not damaged.

There many possible ways to organize your files. One is to make only one working directory (saying, `wrk`) and place all your files there by subdirectories.

What are these subdirectories, is really personal, however, it is highly recommended to make one of them (e.g., `wrk/temp` or `wrk/tmp`) directory for quick tasks, temporary and/or recent work which. It is also handy to make your browser download stuff there.

File operations

Files could be created, renamed, copied, moved or deleted. Each of these operations has its own specific.

**Creation** Typically, it is not easy just to make the file without content. On Linux, however, this might be done with touch command.

**Rename** Note that this is not an operation with file, this is an operation with file system. If you rename the file, you change the list of files in current directory and/or in particular place of the current file system.
Copy  Typically, this is creation of new file + writing to it.

Move  Copy, then remove the first one. However, if this operation does not go outside of particular physical device (like one hard disk or one flash drive), it is analogous to renaming and therefore very fast.

Remove  Depending on operation and file system used, this is either reversible or (more frequent) non-reversible so you delete your files forever. Many OSes have “trash” mechanism which moves “deleted” files into the trash directory first, and “emptying trash” is a permanent removal. There are undelete/unerase tools around which might claim an ability to restore even after emptying the trash, but most of them are not very effective, especially if user applies them long after deletion.

**File metadata**

“Metadata” is something about the file, not directly related with file content. For example, metadata might tell about what is on this image, when it was taken and so on. There are several approaches:

1. Keep metadata within a file, sometimes in a way hidden from user. This is how cameras keep their metadata in JPEG files, or office programs—within documents. This way is productive enough but often works against privacy and also depends on particular software.

2. Keep metadata in a file directory, as a special feature of file system, or even simpler, as a separate text file, description with two columns: file name and file description.

**File managers. OFM**

Almost every operation system has a way to control file creation, naming, moving, linking etc. Typically, there is some file manager application. There are many of them. Best file managers employ simple idea of two panels which both contain file lists, and user do all operations from left to right or from right to left. Advantage of these file managers (so-called OFMs) is that they save time, file operations are 2–5 times faster then in other programs!

In addition, OFMs host multiple useful functions: user menus, internal editors and viewers for different file types, embedded archiving and compression functions, abil-
ity to access networks with different protocols, and many others. There are several OFMs which are cross-platform, e.g., work on Windows, macOS and Linux:

- Double Commander. Fast, intensively developing software, has many plug-ins.
- muCommander. Requires Java. Generally, more stable then Double Commander.

**Exercise 5**

Create 9 files named `file1.txt`, `file2.txt`, ..., `file9.txt`. How long did it take? How to speed up this process?

**Privacy and cleaning**

Various software leave numerous traces on your computer. If it is not what you want, and if you want to keep (relative) privacy, there are several programs which wipe out most conspicuous unwanted results of your work. BleachBit, for example, is completely cross-platform and not only increase privacy but also free the disk space.

Among others, BleachBit helps to clean:

1. **Browsers’ cache**, storage which browsers use to improve the user-visible speed. Cache often keeps older versions of documents, and it is not always easy to clean it in order to obtain the recent version.

2. **Cookies**, identificators which allow external Web sites to remember you and, for example, do not give you discount prices even if they available to others.

3. **Temporary files** of various kinds, including Windows Thumbsnails.db which presents in every directory with images, or macOS .DS_Store, which is located on any device which came out of Mac computer.
Chapter 4

Internet

Internet protocols

When computers connect through the network, they perform multiple standardized operations; this is network protocol. Many of them now are only little more then historical relics, and only few are common:

**HTTP** Hypertext protocol which allows browsing with hyperlinks; this is now core Internet protocol. Important is to know that if your computer becomes HTTP server, your file system is not accessible, HTTP does not understand it. You can create a file on your Web site but if there is no link to it, nobody, even Google or other indexing system, will not know about it.

**HTTPS** Somehow similar to HTTP but everything is encrypted, therefore, no third party will know what you send or receive. Now there is a movement to replace HTTP with HTTPS wherever possible, for obvious reasons.

If you want to connect securely, your computer must know if it cant trust the other side. This is why it uses certificates. In practice, however, it is possible to replace good certificate with a bad one, and then listen to your “encrypted” connections which are not encrypted anymore (this is called MiTM, man-in-the-middle attack). So even with HTTPS, you do not have 100% guaranteed privacy.

**FTP** File transfer protocol. This will see your file system (if your computer is FTP server).

**SSH** Secure shell connection. Allows to connect two computers to (1) work remotely in the terminal and (2) transfer files through the encrypted channel. Useful, especially for the remote administration, updates and almost everywhere when you need to work on more then one computer.
mailto  Strictly speaking, this is not a protocol, it is more like a gateway to multiple protocols responsible for the sending of email messages. Encryption here is still rare.

DOI  Again, not exactly a protocol but identifier which allows to refer books and articles.

magnet  This is from the big family of peer-to-peer protocols which allow to connect two or more computers without server or with the minimal participation of server. Useful for sharing files.

Exercise 6

Sometimes, when you enter https:// instead of http:// as URL, your Internet browser reports things like “Secure Connection Failed”. But the reverse is not true. Why?
Chapter 5

Licenses

Internet brought many new things to the society, and free software is one of the most important. “Free” as a freedom, not as free beer, they frequently say. There are many variants of free licenses, some are shortly explained below:

GNU GPL This license relates with GNU Project, idea to re-create UNIX-like operation system as free and open source. And yes, they did what they planned. One (theoretically) can sell GNU GPL software, but license states that you also must distribute the source code and that is forbidden to restrict the free re-distribution.

Creative Commons The big family of free licenses which provide multiple ways how to use and modify product, its source code and how to credit authors.

Public domain The most free license. Public domain means that anyone can use, modify, sell, buy, even without any credit to the original author. These notes are public domain.
Commands
Chapter 6

CLI is to command

There are two choices for computer user:

1. If you want to command your computer, use commands through command-line interface (CLI)
2. If you want your computer to command you, use graphical user interface (GUI)

This sounds dramatic, but in essence correct. Command line tools give you freedom and control on your machine.

* * *

When your learn how to command your computer, remember these two (related) advices:

Make mistakes! The more mistakes you do now, the more your learn, and less you do in future.

Experiment! Copy-paste from this book, change code, invent your own code, try something new, ask “what if” questions and try to answer it yourself.
Chapter 7

Terminals and where to find them

There is typically no problem to find terminal application, they are called Terminal, Terminal.app (macOS: Utilities → Terminal.app) or Command Prompt (Windows: Windows System → Command Prompt).

If you start terminal for the first time, one of problems is to run it in such a way that its current directory is directory you want. Typically, user has some kind of working directory to run Python programs, make Web pages, L|TEX documents and other stuff. However, when you start terminal, it by default starts somewhere else. So the goal is to point your terminal to the place you want.

This is possible, if you know terminal commands. Most important is cd. Open terminal window, then type something like:

```sh
cd c:/Users/<myname>/Desktop/wrk
```

<myname> is the name of current user (likely you), and everything after cd is the location which you declare to be your working directory. It might be helpful to open file manager in a separate window (or “Save” window of your editor), locate this path and then type it into the terminal.

To check if you are in the desired place (working directory), type dir (Windows) or pwd and then ls (UNIX-like). You should see the name of your directory and names of files which should be kept there.

If you not exactly sure where is your goal directory, type dir (or ls) and cd <dir> sequentially, until you find the destination, and your command prompt shows the name of working directory.

***

While all OSes have terminal applications, only UNIX-based (macOS and Linux) run “true”, POSIX-compliant terminals. Thus, it is not easy to start learning terminal commands if you work on Windows. Several workarounds exist.
First, there are emulation tools, for example, JSLinux which runs kind of (restricted) Linux system in your browser. There is also tools which run terminal applications natively. One is a Cygwin, system which is designed to run POSIX terminals on Windows machines. Cygwin is freeware, and should run on any kind of Windows computer. Another system is DJGPP, Delorie GNU Programming Platform, which is more lightweight.
Chapter 8

UNIX Power Tools

To learn famous UNIX power tools, you must type all examples in this chapter. To make life easier, create Desktop in your home directory (it might already exist) and wrk inside Desktop.

Navigate

To move between directories, you only need to know several symbols and two commands.

Main command is cd (“change directory”). It is not a separate program but the part to UNIX shell which is secretly present in all examples below.

cd will bring you anywhere if you know the address, path. Fortunately, completion allows not to remember paths but simply type Tab and look what system shows. When you are in place, it is good to check files “around you” with command ls (“list”). In addition, command pwd shows where you are (i.e., name of directory). However, this name is typically shown in the prompt.

Suppose I am somewhere and want to find my working directory which name is wrk and it is among in my user directories:

```
myname@mycomp:~$ cd ~          # this is to go to my home: ~
myname@mycomp:~$ ls             # so what is there?
bin  Desktop  wrk
myname@mycomp:~$ cd Desktop     # oh, I found Desktop!
/home/mynmae/Desktop
myname@mycomp:~/Desktop$ ls     # go to desktop
    cr  wrk                      # what is on Desktop?
myname@mycomp:~/Desktop$ cd wrk  # found it!
/home/mynmae/Desktop/wrk
```
myname@mycomp:~/Desktop/wrk$ pwd  # again, where am I?
/home/alexey/Desktop/wrk
myname@mycomp:~/Desktop/wrk$ ls  # just in case, what is here?
1.txt

(Symbol # is a comment, everything after it ignored.)

To go up one level, use

myname@mycomp:~/Desktop/wrk$ cd ..
myname@mycomp:~/Desktop$

One dot (.) is the current directory so cd . do not move you anywhere, it is useful in other situations.

Completion works like some replacement for ls:

myname@mycomp:~/Desktop/wrk$ cd ./  # then I typed Tab and see:
something_else/ wrk/  # two directories here
myname@mycomp:~/Desktop/wrk$ cd ./wrk/  # so I types one letter "u"
        # then Tab
        # and system completed
        # with "niv_110/

myname@mycomp:~/Desktop/wrk$
        # and I am here!

(Forward slash / is directory separator.)

**Files**

To make empty (0-size) file, type

myname@mycomp:~/Desktop/wrk$ touch 1.txt
myname@mycomp:~/Desktop/wrk$ ls -l
total 0
-rw-rw-r-- 1 myname myname 0 Nov 8 22:49 1.txt

(Note ls -l which shows more details about file, including size.)

If you want to remove it (not to trash but forever!), type rm 1.txt.

To move it from local directory one level up, type:

myname@mycomp:~/Desktop/wrk$ mv 1.txt ..
myname@mycomp:~/Desktop/wrk$ ls ..  # to check
1.txt something_else wrk  # yes, it is there
To copy, use something like `cp 1.txt 2.txt`.

By the way, many power tools are not fool proof! UNIX typically assumes that you know what you are doing. So, for example, `cp` will copy one file into another if even it exist and therefore will completely erase the previous content without any warning.

**Find**

It is too boring, even with completion, repeatedly type `cd` and `ls`. There are two tools which help to find files and directories by name:

```
myname@mycomp:~$ find -name "univ*"
./Desktop/wrk
```

```
myname@mycomp:~$ locate wrk
/home/myname/Desktop/wrk
```

`find` works in real time and search within given directory (by default, within current), `locate` uses database and return the absolute path (which starts with slash). `find` can also use wildcards, for example, star `*` which means “any number of any symbols”, or `?` which is “just one symbol”.

If you want to find files by content, read below about `grep`.

**Directories**

To make directories, use `mkdir`. It can make multiple directories (with the help of shell) and the sequence of directories (directory tree) in one step:

```
myname@mycomp:~/Desktop/wrk$ mkdir -p one/{1,2,3}
```

```
myname@mycomp:~/Desktop/wrk$ find
.
./one
./one/2
./one/3
./one/1
```
Directories

(Here find used to list everything which is in current directory.)

Remove (empty) directory tree with `rm -r`:

```bash
mynname@mycomp:-/Desktop/wrk$ rm -r one
mynname@mycomp:-/Desktop/wrk$ find .                 # nothing is left
```

Non-empty directories could be removed with `rm -rf` command but be careful: if you apply it without thinking, you could really harm your computer. You warned.

***

Utility `ln` allows to make symbolic links which is one of the most powerful mechanism related with UNIX file systems. Suppose that I have three files:

```bash
mynname@mycomp:-/Desktop/wrk$ touch {1,2,3}.txt
mynname@mycomp:-/Desktop/wrk$ ls *.txt
1.txt 2.txt 3.txt
mynname@mycomp:-/Desktop/wrk$
```

I want to categorize them. The most common way is to distribute them by directories. But what if I want two different systems of categories? In other words, two views on these files? If I copy these files, then every edit, I must edit twice (or edit once and copy afterwards).

Symbolic links will solve this problem in a most elegant way.

First, I prepare these two views:

```bash
mynname@mycomp:-/Desktop/wrk$ mkdir -p by_type/{tables,texts} > by_content/{home,work}
.
./by_content
./by_content/work
./by_content/home
./by_type
./by_type/texts
./by_type/tables
```

(Symbol `\` on one line and `>` on next line mean that I split the line in two parts because it was too long for the book. If you type `\` and press Enter, then symbol `>` on next line appears automatically. This is because `\` is escape, it protects next symbol to be interpret as usual.)
Second, make symbolic links from each file into both places:

myname@mycomp:~/Desktop/wrk$ ln -s 1.txt by_type/texts/1.txt
myname@mycomp:~/Desktop/wrk$ ln -s 2.txt by_type/texts/2.txt
myname@mycomp:~/Desktop/wrk$ ln -s 3.txt by_type/tables/3.txt
myname@mycomp:~/Desktop/wrk$ ln -s 1.txt by_content/work/1.txt
myname@mycomp:~/Desktop/wrk$ ln -s 2.txt by_content/home/2.txt
myname@mycomp:~/Desktop/wrk$ ln -s 3.txt by_content/home/3.txt

Finally, check the result with `ls`, `find` or (like below) `tree` command:

myname@mycomp:~/Desktop/wrk$ tree --charset ascii
.
|-- 1.txt
|-- 2.txt
|-- 3.txt
|-- by_content
| |-- home
| | |-- 2.txt -> 2.txt
| | |-- 3.txt -> 3.txt
| `-- work
|     |-- 1.txt -> 1.txt
`-- by_type
    |-- tables
    | `-- 3.txt -> 3.txt
    `-- texts
        |-- 1.txt -> 1.txt
        `-- 2.txt -> 2.txt

6 directories, 9 files

***

Now for the sake of learning, we break out file naming rules and copy one of our files to the new one which name contains a space:

myname@mycomp:~/Desktop/wrk$ cp 1.txt new 1.txt
cp: target '1.txt' is not a directory

Does not work! This is because space by default delimits items in the command line. This was, by the way, the main reason to avoid space containing file names. Of course, it is possible to make files with space in the name, it is enough to `escape` space with backslash, like `new\ 1.txt`. However, this is better to avoid.
Help

How to know what all these commands do? Read this book, especially the cheatsheet in the end of chapter. Or use help. Typically, there are two kinds of help:

1. Manual pages. To call, use `man <program_name>`, e.g., `man ln`. You will see manual page in a viewer, usually `less`. To quit `less`, type `q`.

2. Help option. To call, use `<program_name> --help` (note two dashes). Short help will output on the screen.

Pipes

In UNIX, everything is a text. Ideally, every program should output some text and accept it as input\(^1\). If the program outputs text, you will see it on the screen:

```
myname@mycomp:~/Desktop$ ls
wrk
```

Or you can redirect it to file:

```
myname@mycomp:~/Desktop$ ls > 0.txt
```

Now use `cat` command to check content of `0.txt`:

```
myname@mycomp:~/Desktop$ cat 0.txt
0.txt
wrk
```

Wait a minute—why is `0.txt` there? This is because the shell first creates the file, and only then performs the command. So at the moment when `ls` start to run, there were two items, `wrk` and `0.txt`.

If file already exists, its content will be erased and replaced with output. However, you can add output to the end of file:

```
myname@mycomp:~/Desktop$ echo "3" >> 0.txt
myname@mycomp:~/Desktop$ cat 0.txt
0.txt
wrk
3
```

\(^1\)Even clipboard, through `xclip` utility.
(Command echo simply outputs its argument.)

You can redirect output of one command into input of another. This is a pipe:

```
myname@mycomp:$ ls | cat
0.txt
wrk
```

(Almost useless example because `ls` will output text anyway, but you got the idea.)

---

**Text**

We already know `cat` which outputs content of any file, and `less` which shows content in a separate viewer instance.

By the way, `cat` (together with UNIX shell) can be regarded as simplest text editor:

```
myname@mycomp:$ cat > new.txt
ATGGTTCCA

myname@mycomp:$ cat new.txt
ATGGTTCCA
```

(Ctrl+D is an “end of input” symbol.)

Utility tail is similar to cat but outputs the very end of file (typically, very long file like log-files of servers), it even can update the output on regular intervals (with option tail -f) and therefore show how the file grows.

Useful `wc` counts symbols and lines in the file:

```
myname@mycomp:$ wc -l 0.txt
3 0.txt
```

(So far, 3 lines in `0.txt`.)

```
***
```

But the UNIX star is `grep`, utility to search files by content:

```
myname@mycomp:$ echo "1" > 1.txt
myname@mycomp:$ echo "2" > 2.txt
myname@mycomp:$ cat *.txt > 3.txt
```
myname@mycomp:~/Desktop/wrk$ grep 1 *txt
1.txt:1
3.txt:1

Naturally, symbol “1” presents only in 1.txt and 3.txt, each time in first row.

Utilities sort and uniq also belong to the those tools which select something from file. sort sorts (what else can it do?), and uniq removes duplicates, typically from already sorted text:

myname@mycomp:~/Desktop/wrk$ cat > 4.txt
one
two
three
one
myname@mycomp:~/Desktop/wrk$ sort 4.txt | uniq
one
three
two

fmt re-format paragraphs (chunks of text separated with empty line) in accordance with some given length (default is 75):

myname@mycomp:~/Desktop/wrk$ fmt 4.txt
one two three one

Finally, diff and comm are really useful when you want to find differences. diff is better for texts and log-files, comm is better for sorted lists and tables:

myname@mycomp:~/Desktop/wrk$ diff 1.txt 3.txt
1a2
> 2

So 3.txt differs from 1.txt in symbol ”2” on the added second row.

comm produces three-column output. Third column are rows common between two files, first column—unique to first file, second column—to second file:

myname@mycomp:~/Desktop/wrk$ comm 1.txt 3.txt
   1
   2

(There is nothing unique to 1.txt so first column did not appear.)
Regexp

Regexp is a shortcut for “regular expressions” which is a way to generalize text editing. Suppose that we have a long text where all numeric intervals are shown as hyphens, like:

```
myname@mycomp:~/Desktop/wrk$ cat > 5.txt
one
32-34
brown-violet
2.1-15
T-shirt
150-150.5
```

But I want them to be en-dashes, which are lines longer then hyphens. In plain text, en-dashes are typically shown as double hyphens. So I want something like:

```
one
32--34
brown-violet
2.1--15
T-shirt
150--150.5
```

One way is to search each hyphen and, if it is surrounded with digits, replace it with double hyphen. Regular expressions allow to do it automatically:

```
myname@mycomp:~/Desktop/wrk$ cat 5.txt | \
> sed "s/\([0-9]\)-\([0-9]\)/\1--\2/g"
```

(Symbols \ and > used because line was too long, see above.)

A bit frightening, was it? This is a main complain about regexps, complicated syntax. But this one is simple enough:

```
  sed is stream editor, command editor which changes any input
"s/.../.../g" means search and replace globally, so if there multiple replacements on one line, all will be replaced
```
[0-9] is any digit from 0 to 9
\(...\) save this in memory to use in replacement part
- this is hyphen to replace with double hyphens
\1 and \2 first and second memorized object (which appeared in \(...\))

Our regular expression does exactly what you would do if you replace manually, but it is way faster and do not skip entries.

sed is extremely powerful and allows to change files of giant size in seconds. Related task is to rename multiple files. There are many utilities but one of most widespread is rename:

```
myname@mycomp:~/Desktop/wrk$ rename -n 's/([0-9])/file$1/g' *txt
rename(1.txt, file1.txt)
rename(2.txt, file2.txt)
rename(3.txt, file3.txt)
rename(4.txt, file4.txt)
rename(5.txt, file5.txt)
```

(Option -n shows how files would be renamed. If you want the proposed changes, remove -n.)

It is a bit inconsistent with sed regexps because it uses slightly different rules (so-called Perl regexps: $1 instead of \1, ( instead of \() but principle is the same.

### Connect

Terminal can do all network-related things, including Web browsing (there is, for example, links2 text browser). I think that for beginners, most important network-related tools are:

- wget, offline browser, allows to download Web pages and whole Web sites.
- ifconfig shows your current IP address (or addresses).
- rsync, synchronization tool, allows for backups and cloud-like services.
- ssh, secure shell (see above).
Script

Shell was “secretly” present in all examples above. First, shell controls wildcards which expand when you want short descriptions of file names:

```bash
myname@mycomp:~/Desktop/wrk$ ls *txt
1.txt 2.txt 3.txt 4.txt 5.txt new.txt
myname@mycomp:~/Desktop/wrk$ ls ?.txt
1.txt 2.txt 3.txt 4.txt 5.txt
myname@mycomp:~/Desktop/wrk$ ls [1-3].txt
1.txt 2.txt 3.txt
```

Shell controls redirection: input, output and pipes.

Shell also controls prompt which in out examples was:

```bash
myname@mycomp:<current_dir>$
```

***

Shell can do much more. For example, shell scripts. When you repeat your command over and over again, you might want to save your typing and invent some kind of program which will do work for you.

Say, we want a typical (see below) “Hello, World!” program. If we want shell to show us something, we use echo:

```bash
myname@mycomp:~/Desktop/wrk$ echo "Hello, World!"
Hello, World!
```

Now, some little magic and we will have hello.sh script:

```bash
myname@mycomp:~/Desktop/wrk$ cat > hello.sh
#!/bin/bash
echo "Hello, World!" # do not forget Ctrl+D in the end
myname@mycomp:~/Desktop/wrk$ . ./hello.sh
Hello, World!
```

(Dot . is the run command, and next to it are ./ which means that script is in current directory.)

Our first shell script contains two lines. Second we already tried manually. First is shebang line which tells system that I need bash (“Bourne-again” shell) to run my script. In case of hello.sh, shebang is not so necessary.
hello.sh is not very useful, but if you want something complicated, for example, change multiple files, script is the most practical way.

Say, for example, that we want to backup text files, copy each of them to new file, keep the name but change extension to *.bak. Then we need the following script:

```bash
#!/bin/bash
for file in *txt
do
    name="${file%.*}"
    cp $name.txt $name.bak
done
```

```
myname@mycomp:~/Desktop/wrk$ cat > bak.sh
#!/bin/bash
for file in *txt
do
    name="${file%.*}"
    cp $name.txt $name.bak
done
myname@mycomp:~/Desktop/wrk$ . ./bak.sh # run it
myname@mycomp:~/Desktop/wrk$ ls # check result
1.bak 1.txt 2.bak 2.txt 3.bak 3.txt
4.bak 4.txt 5.bak 5.txt bak.sh
hello.sh new.bak new.txt
```

**UNIX Power Tools cheatsheet**

List below might helpful in order to find which utility to use. To know *how to use*, run `man` and/or `--help` option. Some utilities were not explained above in details, hence the short comment.
**Input and output**
bash UNIX shell

cat
echo
tail

**Files and directories**
cd
cp
find
ln
ls
mc file manager
mkdir
mv
pwd
rm
touch
tree

**Modify text**
nano text editor
ne another text editor
sed
sort
uniq

**Compare**
diff
comm

**Search**
find
grep
locate

**Net**
curl fetch Web pages
ifconfig
ssh secure shell
wget

**Permissions**
chmod change access mode
chown change owner

**Info**
cal calendar
du disk space usage
date date and time
less
history command history
wc

**Processes**
kill control process
ps show processes
top show processes in table

**Backup**
gzip compress
rsync
tar make archive

**Utilities**
convert ImageMagick
leanify image optimizer
pdfimages extract images from PDF
pdftk PDF toolkit
pdflatex run LaTeX

---

And if you want to read more about “UNIX power tools”, the best book has the same exact name.
YOU'RE FLYING! HOW?

I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST "Hello, world!"

I DUNNO...
DYNAMIC TYPING?
WHITESPACE?
COME JOIN US!
PROGRAMMING IS FUN AGAIN!
IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?

I JUST TYPED
import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.
Chapter 9

Intro to Python

The goal of this part is to teach how to command your computer, which is essentially how to program.

One of the best programmer’s learning tools is the Python programming language because it is a rare combination of being easy useful from first steps.

Some of the following chapters are modified from the open source Python book, “Non-Programmer’s Tutorial for Python” which was initially the text of Josh Cogliati and then extended by multiple co-authors and now is hosting on Wikipedia (https://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6).

First things first

So, you’ve never programmed before. As we go through this tutorial I will attempt to teach you how to program. There really is only one way to learn to program. You must read code and write code. I’m going to show you lots of code. You should type in code that I show you to see what happens. Play around with it and make changes. The worst that can happen is that it won’t work. When I type in code it will be formatted like this:

print "Hello, World!"

That’s so it is easy to distinguish from the other text. I will also print what the computer outputs in that same font.
Installing Python

Now, on to more important things. In order to program in Python you need the Python software.

If you work on Windows, go to http://www.python.org and get the proper version for your OS. Download it, read the instructions and get it installed. Important operation is to add Python executable to the search path. Fortunately, the recent installer program will do it for you if during the installation, you choose “Add python.exe to Path,” and then select “Will be installed on local hard drive”. Sometimes, you need to restart your computer after installation.

If you work on Linux, Python is most likely already installed on your computer. Latest macOS versions also have pre-installed Python.

2 or 3?

This is the long story, but two most recent versions of Python are simultaneously available, one starts with “2”, and the other with “3”. I will need you to download, install and use 2-version (for example, Python 2.7.12, or Python 2.7.14, whatever 2-version is most recent).

Interactive Mode

First, you should run the terminal.

Second, terminal should run in the directory where you want to keep your Python programs. If you do not know how to run terminal application in given directory, see the above for details.

Next, in the terminal window, type python. You should see some text like this:

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The >>> is Python way of telling you that you are in interactive mode. In interactive mode what you type is immediately run. Try typing 1+1 in. Python will respond with 2. Interactive mode allows you to test out and see what Python will do. If you ever
feel you need to play with new Python statements go into interactive mode and try them out.

To quit interactive mode, type `quit()`, or (on UNIX-like) `Ctrl+D`, or (Windows) `Ctrl+Z` and Enter.

**Python in Web Browser**

If you cannot, do not want or do not have time to install Python on your computer, use online emulator based on Skulpt, for example, [http://ashipunov.info/shipunov/school/univ_110/python.htm](http://ashipunov.info/shipunov/school/univ_110/python.htm). However, it is a poor man’s solution since there are multiple restrictions, and most important is the inability to work with files.
Chapter 10

Hello, World!

The main way to use Python is not interactive, and not with emulator. It is to create and run programs.

Creating and Running your First Program

Programming tutorials since the beginning of time have started with a little program called “Hello, World!” So here it is:

print "Hello, World!"

Open text editor and type the following:

print "Hello, World!"

First save the program. Go to File then Save. Save it in your dowking directory\(^1\) as hello.py. Now that it is saved it can be run.

Next run the program by typing in terminal\(^2\):

$ python hello.py

This will output

Hello, World!

Your first program is done!

\(^1\)It is where you keep all your Python programs.

\(^2\)Before that, do not forget to point your terminal to the working directory. Use ls/dir and cd commands. Then with the same ls or dir check if your hello.py is actually there. Instead of terminal, you can open Python browser emulator, copy-paste your code and click “Run”.

And you also learned the important principle: *edit in one place, run in another*. Edit your program in text editor (like Geany), save it, and run it in the terminal\(^3\).

If you do not like result, or want to improve it, go back to editor window, change your program, *save* it, then go to terminal and run *python* again.

## Printing

Now I’m not going to tell you this every time, but when I show you a program I recommend that you type (or copy-paste) it in and run it. I learn better when I type it in and you probably do too.

Now here is a more complicated program:

```python
print "Jack and Jill went up a hill"
print "to fetch a pail of water;"
print "Jack fell down, and broke his crown,"
print "and Jill came tumbling after."
```

When you run this program it prints out:

```
Jack and Jill went up a hill
to fetch a pail of water;
Jack fell down, and broke his crown,
and Jill came tumbling after.
```

Save it in working directory as *jack.py*, and run. When the computer runs this program it first sees the line:

```python
print "Jack and Jill went up a hill"
```

so the computer prints:

```
Jack and Jill went up a hill
```

Then the computer goes down to the next line and sees:

```python
print "to fetch a pail of water;"
```

So the computer prints to the screen:

```
to fetch a pail of water;
```

---

\(^3\)Geany for macOS and Linux has terminal embedded; also, on all systems Geany is able to run your current file with Run menu.
The computer keeps looking at each line, follows the command and then goes on to the next line. The computer keeps running commands until it reaches the end of the program.

---

**Quotes**

Now you understand that double quote is a sign to tell Python that everything next to it is a text. Until another double quote.

Remember two mottoes? *Experiment!* What if there is a *single quote*?

Modify `jack.py`:

```python
print "Jack and Jill went up a hill"
print "to fetch a pail of water;"
print "Jack fell down, and broke his crown,"
print "and Jill came tumbling after.'
```

(There is only *one* difference. **Find** it.)

Save and run:

```bash
$ python jack.py
  File "jack.py", line 4
    print "and Jill came tumbling after.'

SyntaxError: EOL while scanning string literal
```

Mistake. Do not worry! Remember—*do as many mistakes as possible!* So what is wrong? Modify again:

```python
print "Jack and Jill went up a hill"
print "to fetch a pail of water;"
print "Jack fell down, and broke his crown,"
print 'and Jill came tumbling after.'
```

Save and run:

```bash
$ python jack.py
Jack and Jill went up a hill
to fetch a pail of water;
Jack fell down, and broke his crown,
and Jill came tumbling after.
```
Everything is fine again! So, Python does not mind which quote to use. But quotes must be paired. This is made to allow double and single quotes to be used as single symbols:

```
print "Jack and Jill went up a hill"
print 'to fetch a 10" pail of water,'
print "Jack fell down, and broke Jack's crown,
print 'and Jill \'came\' tumbling after.'
```

Output:

```
$ python jack.py
Jack and Jill went up a hill
to fetch a 10" pail of water;
Jack fell down, and broke Jack's crown,
and Jill 'came' tumbling after.
```

(So if pairing does not work, one can use backslashes to “escape”, or protect, quotes.) Some mistakes and many experiments—and you know now how quotes work.

**Expressions**

Here is another program:

```
print "2 + 2 is", 2+2
print "3 * 4 is", 3 * 4
print 100 - 1, " = 100 - 1"
print "(33 + 2) / 5 + 11.5 = ",(33 + 2) / 5 + 11.5
```

Save it as expr.py. And here is the output when the program is run:

```
2 + 2 is 4
3 * 4 is 12
99 = 100 - 1
(33 + 2) / 5 + 11.5 = 18.5
```

(So quotes protect expressions from being evaluated.)

As you can see, Python can turn your thousand dollar computer into a 5 dollar calculator.

Python has six basic operations for numbers:
Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponentiation</td>
<td>**</td>
<td>5 ** 2 == 25</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>2 * 3 == 6</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>14 / 3 == 4</td>
</tr>
<tr>
<td>Remainder</td>
<td>%</td>
<td>14 % 3 == 2</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>1 + 2 == 3</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>4 - 3 == 1</td>
</tr>
</tbody>
</table>

Notice that division follows the rule, if there are no decimals to start with, there will be no decimals to end with. The following dec.py program shows this:

```
print "14 / 3 = ",14 / 3
print "14 % 3 = ",14 % 3
print
print "14.0 / 3.0 =",14.0 / 3.0
print "14.0 % 3.0 =",14 % 3.0
```

With the output:

```
14 / 3 = 4
14 % 3 = 2

14.0 / 3.0 = 4.66666666667
14.0 % 3.0 = 2.0
```

(Why there is a middle empty line?)

Notice how Python gives different answers for some problems depending on whether or not there decimal values are used.

The order of operations is the same as in math:

1. parentheses ()
2. exponents **
3. multiplication *, division \, and remainder %
4. addition + and subtraction -
Talking to humans (and other intelligent beings)

Often in programming you are doing something complicated and may not in the future remember what you did. When this happens the program should probably be commented. A comment is a note to you and other programmers explaining what is happening. For example:

```
# Not quite Pi but close enough
print 22.0/7
# 355.0/113 is even closer to Pi

(What will print this program?)
```

Notice that the comment starts with a #. Comments are used to communicate with others who read the program and your future self to make clear what is complicated.

Examples

Each chapter (eventually) will contain examples of the programming features introduced in the chapter. You should at least look over them see if you understand them. If you don’t, you may want to type them in and see what happens. *Mess around them, change them and see what happens.*

```python
***
print "Something's rotten in the state of Denmark."
print " -- Shakespeare"
```

Output:

```
Something's rotten in the state of Denmark.
 -- Shakespeare
```

***

```
# This is not quite true outside of USA
# and is based on dim memories of my younger years
print "First Grade"
print "1+1 =",1+1
print "2+4 =",2+4
print "5-2 =",5-2
```
print
print "Third Grade"
print "243-23 =",243-23
print "12*4 =",12*4
print "12/3 =",12/3
print "13/3 =",13/3," R ",13%3
print
print "Junior High"
print "123.56-62.12 =",123.56-62.12
print "(4+3)*2 =",(4+3)*2
print "4+3*2 =",4+3*2
print "3**2 =",3**2
print
Output:

First Grade
1+1 = 2
2+4 = 6
5-2 = 3

Third Grade
243-23 = 220
12*4 = 48
12/3 = 4
13/3 = 4  R  1

Junior High
123.56-62.12 = 61.44
(4+3)*2 = 14
4+3*2 = 10
3**2 = 9

Exercises

Write a program that prints your last name, your first name and your birthday as separate strings.

Write a program that shows the use of all 6 math functions, in order of operations.
Chapter 11

Who Goes There?

Input and Variables

Now I feel it is time for a really complicated program. Here it is:

```python
print "Halt!"
s = raw_input("Who goes there? ")
print "You may pass," , s
```

(Why did we insert trailing space into the question above?)

Save it as a `whogoh.py` file and run. When I ran it here is what my screen showed:

```
Halt!
Who goes there? Josh
You may pass, Josh
```

Of course when you run the program your screen will look different because of the `raw_input` statement. When you ran the program you probably noticed (you did run the program, right?) how you had to type in your name and then press Enter. Then the program printed out some more text and also your name. This is an example of input. The program reaches a certain point and then waits for the user to input some data that the program can use later.

Of course, getting information from the user would be useless if we didn’t have anywhere to put that information and this is where variables come in. In the previous program `s` is a variable. Variables are like a box that can store some piece of data. And it does not matter how to name the variable, just be consistent:

```python
print "Halt!"
randomname = raw_input("Who goes there? ")
print "You may pass," , randomname
```

Name of variable is different, but program works exactly the same way.
Here is another program to show examples of variables:

```python
a = 123.4
b23 = 'Spam'
first_name = "Bill"
b = 432
c = a + b
print "a + b is", c
print "first_name is", first_name
print "Sorted Parts, After Midnight or", b23
```

And here is the output:

```
a + b is 555.4
first_name is Bill
Sorted Parts, After Midnight or Spam
```

The variables in the above program are: `a`, `b23`, `first_name`, `b`, and `c`.

Variables store data. Two basic types are strings and numbers.

Note the difference between strings and variable names. Strings are marked with quotation marks, which tells the computer *don’t try to understand, just take this text as it is*:

```python
print "first_name"
```

This would print the text:

```
first_name
```

Variable names are written without any quotation marks and instruct the computer *use the value I’ve previously stored under this name*:

```python
print first_name
```

which would print:

```
Bill
```

**Assignment**

Okay, so we have these boxes called variables and also data that can go into the variable. The computer will see a line like `first_name = "Bill"` and it reads it as *put the string Bill into the box (or variable) first_name.*
Later on it sees the statement $c = a + b$ and it reads it as $put \ a + b \ or \ 123.4 + 432 \ or \ 555.4 \ into \ c$.

The right hand side of the statement $(a + b)$ is *evaluated* and the result is stored in the variable on the left hand side ($c$). This is called *assignment*, and you should not confuse the assignment equal sign ($) with “equality” in a mathematical sense here (that’s what $==$ will be used for later).

Here is another example of variable usage:

```python
a = 1
print a
a = a + 1
print a
a = a * 2
print a
```

And of course here is the output:

```
1
2
4
```

Even if it is the same variable on both sides the computer still reads it as: First find out the data to store and than find out where the data goes.

One more program before I end this chapter:

```python
number = input("Type in a number: ")
string = raw_input("Type in a String: ")
print "number =", number
print "number is a ",type(number)
print "number * 2 =",number*2
print "string =", str
print "string is a ",type(str)
print "string * 2 =",str*2
```

The output I got was:

```
Type in a number: 12.34
Type in a String: Hello
number = 12.34
number is a  <type 'float'>
number * 2 = 24.68
string = Hello
string is a  <type 'string'>
```
string * 2 = HelloHello

Notice that num was gotten with input while str was gotten with raw_input.

raw_input returns a string while input returns a number. When you want the user to type in a number use input\(^1\) but if you want the user to type in a string use raw_input.

The second half of the program uses type which tells what a variable is. Numbers are of type int or float (which are short for 'integer' and 'floating point' respectively). Strings are of type string. Integers and floats can be worked on by mathematical functions, strings cannot. Notice how when Python multiplies a number by a integer the expected thing happens. However, when a string is multiplied by a integer the string has that many copies of it added i.e. str * 2 = HelloHello.

The operations with strings do slightly different things than operations with numbers. Here are some interactive mode examples to show that some more:

>>> "This"+" "+"is"+" joined."
'This is joined.'
>>> "Ha, "*5
'Ha, Ha, Ha, Ha, Ha, '
>>> "Ha, "*5+"ha!"
'Ha, Ha, Ha, Ha, Ha, ha'

Here is the list of some string operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repetition</td>
<td>*</td>
<td>&quot;i&quot;*5 == &quot;iiiiii&quot;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>+</td>
<td>&quot;Hello, &quot;+&quot;World!&quot; == &quot;Hello, World!&quot;</td>
</tr>
</tbody>
</table>

---

Examples

So far, our programs were good only to learn Python. Now time came to make something useful. This program calculates rate and distance problems:

```python
print "Input a rate and a distance"
rate = input("Rate:")
```

\(^1\)Only use input() when you trust your users! Everything entered into input() dialog is evaluated as a Python expression and thus allows the user to take control of your program. So better is to get a string and convert it to the necessary type like int(raw_input()) or float(raw_input()).
distance = input("Distance:")
print "Time ", distance / rate

Sample runs:

$ python rate_times.py
Input a rate and a distance
Rate:5
Distance:10
Time: 2
$ python rate_times.py
Input a rate and a distance
Rate:3.52
Distance:45.6
Time: 12.9545454545

***

This program calculates the perimeter and area of a rectangle:

print "Calculate information about a rectangle"
length = input("Length:")
width = input("Width:")
print "Area",length*width
print "Perimeter",2*length+2*width

Sample runs:

$ python area.py
Calculate information about a rectangle
Length:4
Width:3
Area 12
Perimeter 14
$ python area.py
Calculate information about a rectangle
Length:2.53
Width:5.2
Area 13.156
Perimeter 15.46

(Why there is no space printed after colon?)
Converts Fahrenheit to Celsius:

temp = float(raw_input("Fahrenheit temperature: "))
print (temp-32.0)*5.0/9.0

(Why did we use float(raw_input()) instead of input()?)

Sample runs:

$ python temperature.py
Fahrenheit temperature: 32
0.0
$ python temperature.py
Fahrenheit temperature: -40
-40.0
$ python temperature.py
Fahrenheit temperature: 212
100.0
$ python temperature.py
Fahrenheit temperature: 98.6
37.0

Exercises

Write a program that gets 2 string variables and 2 integer variables from the user, concatenates (joins them together with no spaces) and displays the strings, then divides the two numbers on a new line.

Write a program that gets 1 string variable and 1 integer variable from the user, and then outputs this string as many times as is the value of integer variable.
Chapter 12

Count to 10

While loops

Presenting our first flow control structure. Ordinarily the computer starts with the first line and then goes down from there. Control structures change the order that statements are executed or decide if a certain statement will be run. Here’s the source for a program that uses the while control structure:

```
a = 0
while a < 10:
    a = a + 1
    print a
```

And here is the extremely exciting output:

```
1
2
3
4
5
6
7
8
9
10
```

(And you thought it couldn’t get any worse after turning your computer into a five dollar calculator?)

So what does the program do? First it sees the line `a = 0` and makes a zero. Then it sees `while a < 10:` and so the computer checks to see if `a < 10`. The first time the computer sees this statement `a` is zero so it is less than 10. In other words while `a` is less than ten the computer will run the tabbed in statements.
Here is another example of the use of `while`:

```python
a = 1
s = 0
print 'Enter numbers to add to the sum.'
print 'Enter 0 to quit.'
while a != 0 :
    print 'Current sum:',s
    a = input('Number? ')
    s = s + a
print 'Total Sum =',s
```

The first time I ran this program Python printed out:

```
File "sum.py", line 3
    while a != 0 :
          ^
SyntaxError: invalid syntax
```

I had forgotten to put the `:` after the `while`. The error message complained about that problem and pointed out where it thought the problem was with the `^`. After the problem was fixed here was what I did with the program:

```
Enter numbers to add to the sum.
Enter 0 to quit.
Current sum: 0
Number? 200
Current sum: 200
Number? -15.25
Current sum: 184.75
Number? -151.85
Current sum: 32.9
Number? 10.00
Current sum: 42.9
Number? 0
Total sum = 42.9
```

Notice how `print 'Total Sum =',s` is only run at the end. The `while` statement only affects the line that are *indented*. The `!=` means does not equal so `while a != 0 :` means: “until a is zero, cyclically run the tabbed statements that follow.”

Now that we have `while` loops, it is possible to have programs that run forever. An easy way to do this is to write a program like this (**don't run it** until you fully understand it!):
while 1 == 1:
    print "Help, I'm stuck in a loop."

This program will output

Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
Help, I'm stuck in a loop.
...

until the heat death of the universe or until you stop it. The way to stop it is to hit the “Control” (or “Ctrl”) button and “c” (the letter) at the same time. This will kill the program. (Note: sometimes you will have to hit enter after the Ctrl+C.)

Examples

# This program calculates the Fibonacci sequence
a = 0
b = 1
count = 0
max_count = 20
while count < max_count:
    count = count + 1
    # we need to keep track of a since we change it
    old_a = a
    old_b = b
    a = old_b
    b = old_a + old_b
    # Notice that the "," at the end of a print statement keeps it
    # from switching to a new line
    print old_a,
print

Output:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

(Why are all these numbers on one line?)
# Waits until a password has been entered.
# Use Ctrl+C to break out without the password.

# Note that following must not be the password
# to run loop at least once.
password = ""

# Note that != means not equal
while password != "unicorn":
    password = raw_input("Password:")
print "Welcome in"

Sample run:

Password:auo
Password:y22
Password:password
Password:open sesame
Password:unicorn
Welcome in

Exercises

Write a program that prints first 10 powers of two.

Write a program that asks the user for a user name and password.
Chapter 13

Decisions

If statement

As always, I believe I should start each chapter with a warm up typing exercise so here is a short program to compute the absolute value of a number:

```python
n = int(raw_input("Type a number: "))
if n < 0:
    print "The absolute value of",n,"is","-n"
else:
    print "The absolute value of",n,"is",n
```

Here is the output from the two times that I ran this program:

```
Number? -14
The absolute value of -14 is 14

Number? 24
The absolute value of 24 is 24
```

So what does the computer do when it sees this piece of code? First it prompts the user for a number with the statement `n = input("Number? ")`. Next it reads the line `if n < 0: ` If `n` is less than zero Python runs the line `print "The absolute value of",n,"is","-n"`. Otherwise python runs the line `print "The absolute value of",n,"is",n`.

More formally Python looks at whether the expression `n < 0` is true or false. A if statement is followed by a block of statements that are run when the expression is true. Optionally after the if statement is a else statement. The else statement is run if the expression is false.

There are several different tests that a expression can have. Here is a table of all of them:
Another feature of the `if` command is the `elif` statement. It stands for “else if” and means if the original `if` statement is false and then the `elif` part is true do that part. Here’s an example:

```python
a = 0
while a < 10:
    a = a + 1
    if a > 5:
        print a, " > ", 5
    elif a <= 7:
        print a, " <= ", 7
    else:
        print "Neither test was true"
```

and the output:

```
1 <= 7
2 <= 7
3 <= 7
4 <= 7
5 <= 7
6 > 5
7 > 5
8 > 5
9 > 5
10 > 5
```

Notice how the `elif a <= 7` is only tested when the `if` statement fail to be true. `elif` allows multiple tests to be done in a single if statement.
Examples

# This program shows the use of the == operator
# Using numbers
print 5 == 6
# Using variables
x = 5
y = 8
print x == y

And the output
False
False

***

# Plays the guessing game higher or lower
number = 78
guess = 0

while guess != number :
    guess = input("Guess a number: ")

    if guess > number :
        print "Too high"

    elif guess < number :
        print "Too low"

print "Just right"

Sample run:

Guess a number:100
Too high
Guess a number:50
Too low
Guess a number:75
Too low
Guess a number:87
Too high
Guess a number: 81
Too high
Guess a number: 78
Just right

###

# Asks for a number
# Prints if it is even or odd

```python
number = input("Tell me a number: ")
if number % 2 == 0:
    print number,"is even."
elif number % 2 == 1:
    print number,"is odd."
else:
    print number,"is very strange."
```

Sample runs:

Tell me a number: 3
3 is odd.

Tell me a number: 2
2 is even.

Tell me a number: 3.14159
3.14159 is very strange.

###

# Keeps asking for numbers until 0 is entered.
# Prints the average value.

```python
count = 0
sum = 0.0
number = 1 # set this to something that will not exit
          # the while loop immediatly.

print "Enter 0 to exit the loop"

while number != 0:
```
number = input("Enter a number:")
count = count + 1
sum = sum + number

count = count - 1 #take off one for the last number
print "The average was:",sum/count

Sample runs:
Enter 0 to exit the loop
Enter a number:3
Enter a number:5
Enter a number:0
The average was: 4.0

Enter 0 to exit the loop
Enter a number:1
Enter a number:4
Enter a number:3
Enter a number:0
The average was: 2.66666666667

***

# Keeps asking for numbers until count have been entered.
# Prints the average value.

sum = 0.0

print "This program will take several numbers than average them"
count = input("How many numbers would you like to sum:")
current_count = 0

while current_count < count:
    current_count = current_count + 1
    print "Number ",current_count
    number = input("Enter a number:")
    sum = sum + number

print "The average was:",sum/count

Sample runs:
Exercises

This program will take several numbers than average them
How many numbers would you like to sum: 2
Number 1
Enter a number: 3
Number 2
Enter a number: 5
The average was: 4.0

This program will take several numbers than average them
How many numbers would you like to sum: 3
Number 1
Enter a number: 1
Number 2
Enter a number: 4
Number 3
Enter a number: 3
The average was: 2.66666666667

Exercises

Modify the password guessing program to keep track of how many times the user has entered the password wrong. If it is more than 3 times, print “That must have been complicated.”

Write a program that asks for two numbers. If the sum of the numbers is greater than 100, print “That is big number”.

Write a program that asks the user their name, if they enter your name say “That is a nice name”, if they enter “John Cleese” or “Michael Palin”, tell them how you feel about them ;), otherwise tell them “You have a nice name”.
Chapter 14

Debugging

What is debugging?

As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes discovers debugging, 1949

By now if you have been messing around with the programs you have probably found that sometimes the program does something you didn’t want it to do. This is fairly common. Debugging is the process of figuring out what the computer is doing and then getting it to do what you want it to do. This can be tricky. I once spent nearly a week tracking down and fixing a bug that was caused by someone putting an $x$ where a $y$ should have been.

This chapter will be more abstract than previous chapters. Please tell me if it is useful.

What should the program do?

The first thing to do (this sounds obvious) is to figure out what the program should be doing if it is running correctly. Come up with some test cases and see what happens. For example, let’s say I have a program to compute the perimeter of a rectangle (the sum of the length of all the edges). I have the following test cases:
What should the program do?

I now run my program on all of the test cases and see if the program does what I expect it to do. If it doesn’t then I need to find out what the computer is doing.

More commonly some of the test cases will work and some will not. If that is the case you should try and figure out what the working ones have in common. For example here is the output for a perimeter program (you get to see the code in a minute):

<table>
<thead>
<tr>
<th>width</th>
<th>height</th>
<th>perimeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Height: 3
Width: 4
perimeter = 15

Height: 2
Width: 3
perimeter = 11

Height: 4
Width: 4
perimeter = 16

Height: 2
Width: 2
perimeter = 8

Height: 5
Width: 1
perimeter = 8

Notice that it didn’t work for the first two inputs, it worked for the next two and it didn’t work on the last one. Try and figure out what is in common with the working ones. Once you have some idea what the problem is finding the cause is easier. With your own programs you should try more test cases if you need them.
What does the program do?

The next thing to do is to look at the source code. One of the most important things
to do while programming is reading source code. The primary way to do this is code
walkthroughs.

A code walkthrough starts at the first line, and works its way down until the program
is done. While loops and if statements mean that some lines may never be run and
some lines are run many times. At each line you figure out what Python has done.

Lets start with the simple perimeter program. Don’t type it in, you are going to read
it, not run it. The source code is:

```python
height = input("Height: ")
width = input("Width: ")
print "perimeter = ",width+height+width+width
```

Q: What is the first line Python runs?
A: The first line is always run first. In this case it is:

```python
height = input("Height: ")
```

Q: What does that line do?
A: Prints Height: , waits for the user to type a number in, and puts that in the
variable height.

Q: What is the next line that runs?
A: In general, it is the next line down which is:

```python
width = input("Width: ")
```

Q: What does that line do?
A: Prints Width: , waits for the user to type a number in, and puts what the user
types in the variable width.

Q: What is the next line that runs?
A: When the next line is not indented more or less than the current line, it is the
line right afterwards, so it is:

```python
print "perimeter = ",width+height+width+width
```

(It may also run a function in the current line, but that’s a future chapter.)
Q: What does that line do?
A: First it prints \texttt{perimeter =}, then it prints \texttt{width+height+width+width}.

Q: Does \texttt{width+height+width+width} calculate the perimeter properly?
A: Let's see, perimeter of a rectangle is the bottom (width) plus the left side (height) plus the top (width) plus the right side (huh?). The last item should be the right side's length, or the height.

Q: Do you understand why some of the times the perimeter was calculated 'correctly'?
A: It was calculated correctly when the width and the height were equal.

The next program we will do a code walkthrough for is a program that is supposed to print out 5 dots on the screen. However, this is what the program is outputting:

. . . .

And here is the program:

\begin{verbatim}
number = 5
while number > 1:
    print ".",
    number = number - 1
print
\end{verbatim}

This program will be more complex to walkthrough since it now has indented portions (or control structures). Let us begin.

Q: What is the first line to be run?
A: The first line of the file: \texttt{number = 5}

Q: What does it do?
A: Puts the number 5 in the variable number.

Q: What is the next line?
A: The next line is: \texttt{while number > 1:}

Q: What does it do?
A: Well, \texttt{while} statements in general look at their expression, and if it is true they do the next indented block of code, otherwise they skip the next indented block of code.
Q: So what does it do right now?
A: If `number > 1` is true then the next two lines will be run.

Q: So is `number > 1`?
A: The last value put into `number` was 5 and 5 > 1 so yes.

Q: So what is the next line?
A: Since the `while` was true the next line is: `print ".",`

Q: What does that line do?
A: Prints one dot and since the statement ends with a `,` the next print statement will not be on a different screen line.

Q: What is the next line?
A: `number = number - 1` since that is following line and there are no indent changes.

Q: What does it do?
A: It calculates `number - 1`, which is the current value of `number` (or 5) subtracts 1 from it, and makes that the new value of `number`. So basically it changes `number`’s value from 5 to 4.

Q: What is the next line?
A: Well, the indent level decreases so we have to look at what type of control structure it is. It is a `while` loop, so we have to go back to the `while` clause which is `while number > 1:`

Q: What does it do?
A: It looks at the value of `number`, which is 4, and compares it to 1 and since 4 > 1 the while loop continues.

Q: What is the next line?
A: Since the while loop was true, the next line is: `print ".",`

Q: What does it do?
A: It prints a second dot on the line.
A: No indent change so it is:

\[ \text{number} = \text{number} - 1 \]

Q: And what does it do?

A: It talks the current value of number (4), subtracts 1 from it, which gives it 3 and then finally makes 3 the new value of number.

Q: What is the next line?

A: Since there is an indent change caused by the end of the while loop, the next line is:

\[ \text{while number} > 1: \]

Q: What does it do?

A: It compares the current value of number (3) to 1. \( 3 > 1 \) so the while loop continues.

Q: What is the next line?

A: Since the while loop condition was true the next line is: \text{print } "." ,

Q: And it does what?

A: A third dot is printed on the line.

Q: What is the next line?

A: It is: \text{number} = \text{number} - 1

Q: What does it do?

A: It takes the current value of number (3) subtracts from it 1 and makes the 2 the new value of number.

Q: What is the next line?

A: Back up to the start of the while loop: \text{while number} > 1: 

Q: What does it do?

A: It compares the current value of number (2) to 1. Since \( 2 > 1 \) the while loop continues.
Q: What is the next line?
A: Since the while loop is continuing: print ".",

Q: What does it do?
A: It discovers the meaning of life, the universe and everything. I’m joking. (I had to make sure you were awake.) The line prints a fourth dot on the screen.

Q: What is the next line?
A: It’s: number = number - 1

Q: What does it do?
A: Takes the current value of number (2) subtracts 1 and makes 1 the new value of number.

Q: What is the next line?
A: Back up to the while loop: while number > 1:

Q: What does the line do?
A: It compares the current value of number (1) to 1. Since 1 > 1 is false (one is not greater than one), the while loop exits.

Q: What is the next line?
A: Since the while loop condition was false the next line is the line after the while loop exits, or: print

Q: What does that line do?
A: Makes the screen go to the next line.

Q: Why doesn’t the program print 5 dots?
A: The loop exits 1 dot too soon.

Q: How can we fix that?
A: Make the loop exit 1 dot later.

Q: And how do we do that?
A: There are several ways. One way would be to change the while loop to:
while number > 0:

Another way would be to change the conditional to: number >= 1. There are a couple others.

How do I fix the program?

You need to figure out what the program is doing. You need to figure out what the program should do. Figure out what the difference between the two is. Debugging is a skill that has to be done to be learned. If you can’t figure it out after an hour or so take a break, talk to someone about the problem or contemplate the lint in your navel. Come back in a while and you will probably have new ideas about the problem. Good luck.
Chapter 15

Functions

Creating Functions

To start off this chapter I am going to give you an example of what you could do but shouldn’t (so don’t type it in):

```python
a = 23
b = -23

if a < 0:
    a = -a

if b < 0:
    b = -b

if a == b:
    print "The absolute values of", a,"and",b,"are equal"
else:
    print "The absolute values of a and b are different"
```

with the output being:

```
The absolute values of 23 and 23 are equal
```

The program seems a little repetitive. (Programmers hate to repeat things (That’s what computers are for aren’t they?)) Fortunately Python allows you to create functions to remove duplication. Here’s the rewritten example:

```python
a = 23
b = -23
def my_abs(num):
    if num < 0:
        num = -num
```
return num

if my_abs(a) == my_abs(b):
    print "The absolute values of", a,"and",b,"are equal"
else:
    print "The absolute values of a and b are different"

with the output being:

The absolute values of 23 and -23 are equal

The key feature of this program is the def statement. def (short for define) starts a function definition. def is followed by the name of the function my_abs. Next comes a ( followed by the parameter num (num is passed from the program into the function when the function is called). The statements after the : are executed when the function is used. The statements continue until either the indented statements end or a return is encountered. The return statement returns a value back to the place where the function was called.

Notice how the values of a and b are not changed. Functions of course can be used to repeat tasks (and to eliminate repeat code) that don’t return values. Here’s some examples:

def hello():
    print "Hello"

def area(width,height):
    return width*height

def print_welcome(name):
    print "Welcome",name

hello()
hello()

print_welcome("Fred")
w = 4
h = 5
print "width =",w,"height =",h,"area =",area(w,h)

with output being:

Hello
Hello
Welcome Fred
That example just shows some more stuff that you can do with functions. Notice that you can use no arguments or two or more. Notice also when a function doesn’t need to send back a value, a return is optional.

### Variables in functions

When eliminating repeated code, you often have variables in the repeated code. In Python, these are dealt in a special way. So far all variables we have seen are global variables. Functions have a special type of variable called local variables. These variables only exist while the function is running. When a local variable has the same name as another variable (such as a global variable), the local variable hides the other. Sound confusing? Well, these next examples (which are a bit contrived) should help clear things up.

```python
a = 4
def print_func():
    a = 17
    print "a = ", a, "(this is local a)"
print_func()
print "a = ", a,"(this is global a)"
```

When run, we will receive an output of:

```
a = 17 (this is local a)
a = 4 (this is global a)
```

Variable assignments inside a function do not override global variables, they exist only inside the function. Even though a was assigned a new value inside the function, this newly assigned value was only relevant to `print_func`, when the function finishes running, and the a’s values is printed again, we see the originally assigned values.

### More complex example

```python
a_var = 10
b_var = 15
e_var = 25

def a_func(a_var):
```
print "in a_func a_var = ",a_var
b_var = 100 + a_var
d_var = 2*a_var
print "in a_func b_var = ",b_var
print "in a_func d_var = ",d_var
print "in a_func e_var = ",e_var
return b_var + 10

c_var = a_func(b_var)

print "a_var = ",a_var
print "b_var = ",b_var
print "c_var = ",c_var
print "d_var = ",d_var

The output is:
in a_func a_var =  15
in a_func b_var =  115
in a_func d_var =   30
in a_func e_var =   25
a_var =   10
b_var =   15
c_var =  125
d_var =
Traceback (innermost last):
  File "separate.py", line 20, in ?
    print "d_var = ",d_var
NameError: d_var

In this example the variables a_var, b_var, and d_var are all local variables when they are inside the function a_func. After the statement return _̲var + 10 is run, they all cease to exist. The variable a_var is automatically a local variable since it is a parameter name. The variables b_var and d_var are local variables since they appear on the left of an equals sign in the function in the statements b_var = 100 + a_var and d_var = 2*a_var.

Inside of the function a_var is 15 since the function is called with a_func(b_var). Since at that point in time b_var is 15, the call to the function is a_func(15) This ends up setting a_var to 15 when it is inside of a_func.

As you can see, once the function finishes running, the local variables a_var and b_var that had hidden the global variables of the same name are gone. Then the
Functions statement print "a_var = ",a_var prints the value 10 rather than the value 15 since the local variable that hid the global variable is gone.

Another thing to notice is the NameError that happens at the end. This appears since the variable d_var no longer exists since a_func finished. All the local variables are deleted when the function exits. If you want to get something from a function, then you will have to use return something.

One last thing to notice is that the value of e_var remains unchanged inside a_func since it is not a parameter and it never appears on the left of an equals sign inside of the function a_func. When a global variable is accessed inside a function it is the global variable from the outside.

Functions allow local variables that exist only inside the function and can hide other variables that are outside the function.

Examples

# converts temperature to Fahrenheit or Celsius

def print_options():
    print "Options:"
    print " 'p' print options"
    print " 'c' convert from celsius"
    print " 'f' convert from fahrenheit"
    print " 'q' quit the program"

def celsius_to_fahrenheit(c_temp):
    return 9.0/5.0*c_temp+32

def fahrenheit_to_celsius(f_temp):
    return (f_temp - 32.0)*5.0/9.0

choice = "p"
while choice != "q":
    if choice == "c":
        temp = input("Celsius temperature:"))
        print "Fahrenheit:".,celsius_to_fahrenheit(temp)
    elif choice == "f":
        temp = input("Fahrenheit temperature:"))
        print "Celsius:".,fahrenheit_to_celsius(temp)
elif choice != "q":
    print_options()
    choice = raw_input("option:")

Sample run:

Options:
    'p' print options
    'c' convert from celsius
    'f' convert from fahrenheit
    'q' quit the program

option:c
Celsius temperature:30
Fahrenheit: 86.0

option:f
Fahrenheit temperature:60
Celsius: 15.5555555556

option:q

***

# By Amos Satterlee
print
def hello():
    print 'Hello!'

def area(width,height):
    return width*height

def print_welcome(name):
    print 'Welcome,',name

name = raw_input('Your Name: ')
hello(),
print_welcome(name)
print
print 'To find the area of a rectangle,'
print 'Enter the width and height below.'
print
w = input('Width: ')
while w <= 0:
    print 'Must be a positive number'
w = input('Width:  ')  
h = input('Height: ')  
while h <= 0:  
    print 'Must be a positive number'  
    h = input('Height: ')  
print 'Width =',w,' Height =',h,' so Area =',area(w,h)

Sample Run:

Your Name: Josh  
Hello!  
Welcome, Josh

To find the area of a rectangle, 
Enter the width and height below.

Width:  -4  
Must be a positive number  
Width:  4  
Height:  3  
Width = 4  Height = 3  so Area = 12

Exercises

Rewrite the area.py program (p. 54) to have a separate function for the area of a square, the area of a rectangle, and the area of a circle (3.14 * radius**2). This program should include a menu interface.
Variables with more than one value

You have already seen ordinary variables that store a single value. However other variable types can hold more than one value. The simplest type is called a list. Here is an example of a list being used:

```python
which_one = input("What month (1-12)? ")
months = ['January', 'February', 'March',
         'April', 'May', 'June', 'July',
         'August', 'September', 'October', 'November', 'December']
if 1 <= which_one <= 12:
    print "The month is",months[which_one - 1]
```

and a output example:

```
What month (1-12)? 3
The month is March
```

In this example the `months` is a list. `months` is defined with the lines

```python
months = ['January', 'February', 'March',
         'April', 'May', 'June', 'July',
         'August', 'September', 'October', 'November', 'December']
```

(Note that a \ can be used to split a long line).

The [ and ] start and end the list with comma’s (,”) separating the list items. The list is used in `months[which_one - 1]`. A list consists of items that are numbered starting at 0. In other words if you wanted January you would use `months[0]`. Give a list a number and it will return the value that is stored at that location.
The statement if 1 <= which_one <= 12: will only be true if which_one is between one and twelve inclusive (in other words it is what you would expect if you have seen that in algebra).

***

Lists can be thought of as a series of boxes. Each box contains a different value. For example, the boxes created by
demolist = ['life', 42, 'the universe', 6, 'and', 7]

would look like this:

<table>
<thead>
<tr>
<th>box number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>demolist</td>
<td>'life'</td>
<td>42</td>
<td>'the universe'</td>
<td>6</td>
<td>'and'</td>
<td>7</td>
</tr>
</tbody>
</table>

Each box is referenced by its number so the statement demolist[0] would get 'life', demolist[1] would get 42 and so on up to demolist[5] getting 7.

Lists can contains other lists:
demolist2 = [1, "one", [3, 4]]

More features of lists

The next example is just to show a lot of other stuff lists can do (for once I don’t expect you to type it in, but you should probably play around with lists until you are comfortable with them.). Here goes:
demolist = ['life', 42, 'the universe', 6, 'and', 7]
print 'demolist = ',demolist
demolist.append('everything')
print "after 'everything' was appended demolist is now:"
print demolist
print 'len(demolist) =', len(demolist)
print 'demolist.index(42) =',demolist.index(42)

# Next we will loop through the list
c = 0
while c < len(demolist):

More features of lists

```python
print 'demolist[',c,']=',demolist[c]
c = c + 1
del demolist[2]
print "After 'the universe' was removed demolist is now:'
print demolist
if 'life' in demolist:
    print "'life' was found in demolist"
else:
    print "'life' was not found in demolist"
if 'amoeba' in demolist:
    print "'amoeba' was found in demolist"
if 'amoeba' not in demolist:
    print "'amoeba' was not found in demolist"
demolist.sort()
print 'The sorted demolist is ',demolist
```

The output is:

```
demolist = ['life', 42, 'the universe', 6, 'and', 7]
after 'everything' was appended demolist is now:
['life', 42, 'the universe', 6, 'and', 7, 'everything']
len(demolist) = 7
demolist.index(42) = 1
demolist[1] = 42
demolist[ 0 ]= life
demolist[ 1 ]= 42
demolist[ 2 ]= the universe
demolist[ 3 ]= 6
demolist[ 4 ]= and
demolist[ 5 ]= 7
demolist[ 6 ]= everything
After 'the universe' was removed demolist is now:
['life', 42, 6, 'and', 7, 'everything']
'life' was found in demolist
'amoeba' was not found in demolist
The sorted demolist is [6, 7, 42, 'and', 'everything', 'life']
```

This example uses a whole bunch of new functions. Notice that you can just print a whole list. Next the append function is used to add a new item to the end of the list. \texttt{len} returns how many items are in a list. The valid indexes (as in numbers that can be used inside of the []) of a list range from 0 to \texttt{len} – 1. The \texttt{index} function tell where the first location of an item is located in a list. Notice how \texttt{demolist.index(42)} returns 1 and when \texttt{demolist[1]} is run it returns 42.
Next are the lines:

c = 0
while c < len(demolist):
    print 'demolist[',c,']=',demolist[c]
c = c + 1

These lines create a variable \( c \) which starts at 0 and is incremented until it reaches the last index of the list. Meanwhile the print statement prints out each element of the list.

The `del` command can be used to remove a given element in a list. The next few lines use the `in` operator to test if a element is in or is not in a list.

The `sort` function sorts the list. This is useful if you need a list in order from smallest number to largest or alphabetical. Note that this rearranges the list.

In summary for a list the following operations occur:

<table>
<thead>
<tr>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>demolist[2]</code></td>
<td>accesses the element at index 2</td>
</tr>
<tr>
<td><code>demolist[2] = 3</code></td>
<td>sets the element at index 2 to be 3</td>
</tr>
<tr>
<td><code>del demolist[2]</code></td>
<td>removes the element at index 2</td>
</tr>
<tr>
<td><code>len(demolist)</code></td>
<td>returns the length of list</td>
</tr>
<tr>
<td>&quot;value&quot; in demolist</td>
<td>is true if &quot;value&quot; is an element in list</td>
</tr>
<tr>
<td>&quot;value&quot; not in demolist</td>
<td>is true if &quot;value&quot; is not an element in list</td>
</tr>
<tr>
<td><code>demolist.sort()</code></td>
<td>sorts list</td>
</tr>
<tr>
<td><code>demolist.index(&quot;value&quot;)</code></td>
<td>index of the first place w &quot;value&quot; occurs</td>
</tr>
<tr>
<td><code>demolist.append(&quot;value&quot;)</code></td>
<td>adds an element &quot;value&quot; at the end of the list</td>
</tr>
<tr>
<td><code>demolist.remove(&quot;value&quot;)</code></td>
<td>removes the first occurrence of value from list</td>
</tr>
</tbody>
</table>

***

So `demolist.sort()` sorts the list. Use of `sort` is not similar to use of, saying, print—for some reason it must be specified in the very end of object name, after the dot. Moreover, `sort` is applicable only to lists. This is the first breath of object-oriented programming: `demolist` is object of list type, and `sort` is its method.
This next example uses list features in a more useful way:

```python
menu_item = 0
list = []
while menu_item != 9:
    print "--------------------"
    print "1. Print the list"
    print "2. Add a name to the list"
    print "3. Remove a name from the list"
    print "4. Change an item in the list"
    print "9. Quit"
    menu_item = input("Pick an item from the menu: ")
    if menu_item == 1:
        current = 0
        if len(list) > 0:
            while current < len(list):
                print current,". ",list[current]
                current = current + 1
        else:
            print "List is empty"
    elif menu_item == 2:
        name = raw_input("Type in a name to add: ")
        list.append(name)
    elif menu_item == 3:
        del_name = raw_input("What name would you like to remove: ")
        if del_name in list:
            item_number = list.index(del_name)
            del list[item_number]
            # Above removes only the first occurrence of the name
        else:
            print del_name," was not found"
    elif menu_item == 4:
        old_name = raw_input("What name would you like to change: ")
        if old_name in list:
            item_number = list.index(old_name)
            new_name = raw_input("What is the new name: ")
            list[item_number] = new_name
        else:
            print old_name," was not found"
print "Goodbye"
```
And here is part of the output:

--------------------
1. Print the list
2. Add a name to the list
3. Remove a name from the list
4. Change an item in the list
9. Quit

Pick an item from the menu: 2
Type in a name to add: Jack

Pick an item from the menu: 2
Type in a name to add: Jill

Pick an item from the menu: 1
0. Jack
1. Jill

Pick an item from the menu: 3
What name would you like to remove: Jack

Pick an item from the menu: 4
What name would you like to change: Jill
What is the new name: Jill Peters

Pick an item from the menu: 1
0. Jill Peters

Pick an item from the menu: 9
Goodbye

That was a long program! Let’s take a look at the source code. The line `list = []` makes the variable `list` a list with no items (or elements). The next important line is `while menu_item != 9:`. This line starts a loop that allows the menu system for this program. The next few lines display a menu and decide which part of the program to run.

The section:

```python
current = 0
if len(list) > 0:
    while current < len(list):
```
print current,". ",list[current]
current = current + 1

else:
print "List is empty"

Program goes through the list and prints each name. len(list_name) tells how
many items are in a list. If len returns 0 then the list is empty.

Then a few lines later the statement list.append(name) appears. It uses the append
function to add a item to the end of the list. Jump down another two lines and notice
this section of code:

item_number = list.index(del_name)
del list[item_number]

Here the index function is used to find the index value that will be used later to
remove the item. del list[item_number] is used to remove a element of the list.

old_name = raw_input("What name would you like to change: ")
if old_name in list:
    item_number = list.index(old_name)
    new_name = raw_input("What is the new name: ")
    list[item_number] = new_name
else:
    print old_name," was not found"

The next section uses index to find the item_number and then puts new_name where
the old_name was.

***

Congratulations! With lists under your belt, you now know enough of the language
that you could do any computations that a computer can do (this is technically
known as Turing-Completeness). Of course, there are still many features that are used
to make your life easier.

Examples

# This program runs a test of knowledge

ture = 1
false = 0
# 1. First get the test questions
# Later this will be modified to use file io.
def get_questions():
    # notice how the data is stored as a list of lists
    return ["What color is the daytime sky on a clear day?", "blue"],
           ["What is the answer to life, the universe and everything?", "42"],
           ["What is a three letter word for mouse trap?", "cat"]

# 2. This will test a single question
# it takes a single question in
# returns True if the user typed the correct answer, otherwise False
def check_question(question_and_answer):
    # extract the question and the answer from the list
    question = question_and_answer[0]
    answer = question_and_answer[1]
    # give the question to the user
    given_answer = raw_input(question)
    # compare the user's answer to the testers answer
    if answer == given_answer:
        print "Correct"
        return True
    else:
        print "Incorrect, correct was:" , answer
        return False

# 3. This will run through all the questions
def run_test(questions):
    if len(questions) == 0:
        print "No questions were given."
        # the return exits the function
        return
    index = 0
    right = 0
    while index < len(questions):
        # check the question
        if check_question(questions[index]):
            right = right + 1
        # go to the next question
        index = index + 1
# notice the order of the computation: first multiply, then divide
pp = right*100/len(questions)
print "You got ",pp,"% right out of",len(questions)

# 4. Now let us run the questions
run_test(get_questions())

The values True and False point to 1 and 0, respectively. They are often used in
sanity checks, loop conditions etc. You will learn more about this a little bit later
(chapter “Boolean Expressions”).

Sample output:

What color is the daytime sky on a clear day? green
Incorrect, correct was: blue
What is the answer to life, the universe and everything? 42
Correct
What is a three letter word for mouse trap? cat
Correct
You got  66 % right out of 3

Exercises

Expand the test.py program so it has menu giving the option of taking the test, view-
ing the list of questions and answers, and an option to Quit. Also, add a new ques-
tion to ask, ”What noise does a truly advanced machine make?” with the answer of
”ping”.
Chapter 17

For Loops

And here is the new typing exercise for this chapter:

```python
onetoten = range(1,11)
for count in onetoten:
    print count
```

and the ever-present output:

1
2
3
4
5
6
7
8
9
10

The output looks awfully familiar but the program code looks different. The first line uses the `range` function. The `range` function uses two arguments like this `range(start,finish)`. `start` is the first number that is produced. `finish` is one larger than the last number. Note that this program could have been done in a shorter way:

```python
for count in range(1,11):
    print count
```

Here are some examples to show what happens with the `range` command:

```python
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-32, -20)
```
The next line for count in onetoten: uses the for control structure. A for control structure looks like for variable in list:. list is gone through starting with the first element of the list and going to the last. As for goes through each element in a list it puts each into variable. That allows variable to be used in each successive time the for loop is run through. Here is another example (you don’t have to type this) to demonstrate:

demolist = ['life',42, 'the universe', 6,'and',7,'everything']
for item in demolist:
    print "The Current item is:",
    print item

The output is:

The Current item is: life
The Current item is: 42
The Current item is: the universe
The Current item is: 6
The Current item is: and
The Current item is: 7
The Current item is: everything

Notice how the for loop goes through and sets item to each element in the list. (Notice how if you don’t want print to go to the next line add a comma at the end of the statement (i.e. if you want to print something else on that line). ) So, what is for good for? (groan) The first use is to go through all the elements of a list and do something with each of them. Here a quick way to add up all the elements:

list = [2,4,6,8]
sum = 0
for num in list:
    sum = sum + num
print "The sum is: ",sum

with the output simply being:

The sum is:  20

Or you could write a program to find out if there are any duplicates in a list like this program does:
For Loops

list = [4, 5, 7, 8, 9, 1, 0, 7, 10]
list.sort()
prev = list[0]
del list[0]
for item in list:
    if prev == item:
        print "Duplicate of ",prev," Found"
    prev = item

and for good measure:

Duplicate of 7 Found

Okay, so how does it work? Here is a special debugging version to help you understand (you don’t need to type this in):

l = [4, 5, 7, 8, 9, 1, 0, 7, 10]
print "l = [4, 5, 7, 8, 9, 1, 0, 7, 10]","\tl: ",l
l.sort()
print "l.sort()","\tl: ",l
prev = l[0]
print "prev = l[0]","\tprev: ",prev
del l[0]
print "del l[0]","\tl: ",l
for item in l:
    if prev == item:
        print "Duplicate of ",prev," Found"
        print "if prev == item:","\tprev: ",prev,"\titem: ",item
    prev = item
print "prev = item","\t\tprev: ",prev,"\titem: ",item

with the output being:

l = [4, 5, 7, 8, 9, 1, 0, 7, 10]  l: [4, 5, 7, 8, 9, 1, 0, 7, 10]
l.sort()  l: [0, 1, 4, 5, 7, 7, 8, 9, 10]
prev = l[0]  prev: 0
del l[0]  l: [1, 4, 5, 7, 7, 8, 9, 10]
if prev == item: prev: 0 item: 1
prev = item prev: 1 item: 1
if prev == item: prev: 1 item: 4
prev = item prev: 4 item: 4
if prev == item: prev: 4 item: 5
prev = item prev: 5 item: 5
if prev == item: prev: 5 item: 7
prev = item prev: 7 item: 7
Duplicate of 7 Found
if prev == item: prev: 7 item: 7
prev = item prev: 7 item: 7
if prev == item: prev: 7 item: 8
prev = item prev: 8 item: 8
if prev == item: prev: 8 item: 9
prev = item prev: 9 item: 9
if prev == item: prev: 9 item: 10
prev = item prev: 10 item: 10

The reason I put so many print statements in the code was so that you can see what is happening in each line (by the way, if you can’t figure out why a program is not working, try putting in lots of print statements to you can see what is happening).

First the program starts with a boring old list. Next the program sorts the list. This is so that any duplicates get put next to each other. The program then initializes a prev(ious) variable. Next the first element of the list is deleted so that the first item is not incorrectly thought to be a duplicate. Next a for loop is gone into. Each item of the list is checked to see if it is the same as the previous. If it is a duplicate was found. The value of prev is then changed so that the next time the for loop is run through prev is the previous item to the current. Sure enough, the 7 is found to be a duplicate. (Notice how \t is used to print a tab.)

The other way to use for loops is to do something a certain number of times. Here is some code to print out the first 11 numbers of the Fibonacci series:

```python
a = 1
b = 1
for c in range(1,10):
    print a,
    n = a + b
    a = b
    b = n
```

with the surprising output:

```
1 1 2 3 5 8 13 21 34
```

Everything that can be done with for loops can also be done with while loops but for loops give a easy way to go through all the elements in a list or to do something a certain number of times.
How to do something with each element of the list? Consider the following:

```python
my_list = [1, 2, 3, 4, 5]
my_new_list = [i * 5 for i in my_list]
print my_list, '\n', my_new_list
```

And output:

```
[1, 2, 3, 4, 5]
[5, 10, 15, 20, 25]
```

The same logic works for strings (but you will need to join the result back to string):

```python
a = "abcde"
b = [i * 5 for i in a]
c = ".join(b)
print b
print c
```

```
['aaaaa', 'bbbbb', 'ccccc', 'ddddd', 'eeeee']
aaaaabbbbbccccccccdddddeeеееее
```
Chapter 18

Booleans

Now we need to supply our programs with logic. Here is a little example of Boolean expressions:

```python
a = 6
b = 7
c = 42
print 1, a == 6
print 2, a == 7
print 3, a == 6 and b == 7
print 4, a == 7 and b == 7
print 5, not a == 7 and b == 7
print 6, a == 7 or b == 7
print 7, a == 7 or b == 6
print 8, not (a == 7 and b == 6)
print 9, not a == 7 and b == 6
```

With the output being:

1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False

What is going on? The program consists of a bunch of funny looking `print` statements. Each `print` statement prints a number and a expression. The number is to help keep track of which statement I am dealing with. Notice how each expression
ends up being either 0 or 1. In Python False also is written as 0 and True is written as 1.

The lines:

```python
print 1, a == 6
print 2, a == 7
```

print out True and False respectively just as expected since the first is true and the second is false.

The third print,

```python
print 3, a == 6 and b == 7
```

is a little different. The operator and means if both the statement before and the statement after are true then the whole expression is true otherwise the whole expression is false.

The next line,

```python
print 4, a == 7 and b == 7
```

shows how if part of an and expression is false, the whole thing is false. The behavior of and can be summarized as follows:

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true and true</td>
<td>true</td>
</tr>
<tr>
<td>true and false</td>
<td>false</td>
</tr>
<tr>
<td>false and true</td>
<td>false</td>
</tr>
<tr>
<td>false and false</td>
<td>false</td>
</tr>
</tbody>
</table>

Notice that if the first expression is false Python does not check the second expression since it knows the whole expression is false.

The next line,

```python
print 5, not a == 7 and b == 7
```

uses the not operator. not just gives the opposite of the expression.

By the way, this expression could be rewritten as

```python
print 5, a != 7 and b == 7
```

Here is the table:
Booleans

The two following lines,

\[\text{print 6, a == 7 or b == 7}\]

and

\[\text{print 7, a == 7 or b == 6}\]

use the or operator. The or operator returns true if the first expression is true, or if the second expression is true or both are true. If neither are true it returns false. Here’s the table:

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true or true</td>
<td>true</td>
</tr>
<tr>
<td>true or false</td>
<td>true</td>
</tr>
<tr>
<td>false or true</td>
<td>true</td>
</tr>
<tr>
<td>false or false</td>
<td>false</td>
</tr>
</tbody>
</table>

Notice that if the first expression is true Python doesn’t check the second expression since it knows the whole expression is true. This works since or is true if at least one half of the expression is true. The first part is true so the second part could be either false or true, but the whole expression is still true.

The next two lines,

\[\text{print 8, not (a == 7 and b == 6)}\]

and

\[\text{print 9, not a == 7 and b == 6}\]

show that parentheses can be used to group expressions and force one part to be evaluated first. Notice that the parentheses changed the expression from false to true. This occurred since the parentheses forced the not to apply to the whole expression instead of just the a == 7 portion.
It is easy to understand Python behavior when booleans applied to True’s and False’s (or to 1’s and 0’s). Other types of objects are more cryptic. Look on these examples:

```python
>>> 'b' == ('a' or 'b')
False
>>> 'a' == ('a' or 'b')
True
```

Crazy, isn’t it? This is because when `or` used with `strings`, Python simply returns the first True value. And also because non-empty string is always True:

```python
>>> 'a' or 'b'
'a'
>>> '' or 'b'
'b'
```

There are special rules also for other types of objects and other operations so be careful when applying Boolean functions to anything different from True/False/0/1.

* * *

Here is an example of using a Boolean expression:

```python
list = ["Life","The Universe","Everything","Jack","Jill","Life","Jill"]
# Make a copy of the list.
# See the More on Lists chapter to explain what [:] means.
copy = list[:]
# Sort the copy
copy.sort()
prev = copy[0]
del copy[0]
count = 0
# go through the list searching for a match
while count < len(copy) and copy[count] != prev:
    prev = copy[count]
    count = count + 1
# If a match was not found then count can't be < len
# since the while loop continues while count is < len
# and no match is found
if count < len(copy):
    print "First Match: ",prev
```
And here is the output:

First Match:  Jill

This program works by continuing to check for match while count < len(copy) and copy[count]. When either count is greater than the last index of copy or a match has been found the and is no longer true so the loop exits. The if simply checks to make sure that the while exited because a match was found.

The other ‘trick’ of and is used in this example. If you look at the table for and notice that the third entry is “false and won’t check”. If count >= len(copy) (in other words count < len(copy) is false) then copy[count] is never looked at. This is because Python knows that if the first is false then they both can’t be true. This is known as a short circuit and is useful if the second half of the and will cause an error if something is wrong.

I used the first expression (count < len(copy)) to check and see if count was a valid index for copy. (If you don’t believe me remove the matches ‘Jill’ and ‘Life’, check that it still works and then reverse the order of count < len(copy) and copy[count] != prev to copy[count] != prev and count < len(copy).)

Boolean expressions can be used when you need to check two or more different things at once.

Examples

""
This program asks a user for a name and a password.
It then checks them to make sure the the user is allowed in.
""

name = raw_input("What is your name? ")
password = raw_input("What is the password? ")
if name == "Josh" and password == "Friday":
    print "Welcome Josh"
elif name == "Fred" and password == "Rock":
    print "Welcome Fred"
else:
    print "I don't know you."

(Note multi-line comment made with triple quotes.)

Sample runs
What is your name? Josh
What is the password? Friday
Welcome Josh

What is your name? Bill
What is the password? Money
I don't know you.

Exercises

Write a program that has a user guess your name, but they only get 3 chances to do so until the program quits.
Chapter 19

File Input and Output

Here is a simple example of file I/O (input/output):

```python
# Write a file
out_file = open("test.txt", "w")
out_file.write("This text is going to outfile\nLook and see!")
out_file.close()

# Read a file
in_file = open("test.txt", "r")
text = in_file.read()
in_file.close()

print text
```

The output and the contents of the file test.txt are:

```
This text is going to outfile
Look and see!
```

Notice that it wrote a file called test.txt in the directory that you ran the program from. The \n in the string tells Python to put a newline where it is.

A overview of file I/O is:

1. Get a file object with the open function.
2. Read or write to the file object (depending on how it was opened)
3. Close it

The first step is to get a file object. The way to do this is to use the open function. The format is

```python
file_object = open(filename, mode)
```
where `file_object` is the variable to put the file object, `filename` is a string with the filename, and `mode` is “r” to read a file or “w” to write a file (and a few others we will skip here). Next the file objects functions can be called.

The two most common functions are `read` and `write`. The `write` function adds a string to the end of the file. The `read` function reads the next thing in the file and returns it as a string. If no argument is given it will return the whole file (as done in the example).
Chapter 20

Introduce Python turtle

Do you want some fun? Draw something with Python? Most of recent installations of Python contain module for turtle graphics which is a popular way for introducing programming. Turtle graphics was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the name, then command <name>.forward(15), like:

```python
>>> import turtle
>>> jane = turtle.Turtle()
>>> jane.forward(15)
```

And this will look like:

```

```

and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command <name>.right(25) and it rotates 25° clockwise. By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

Examples

All these examples are better run as separate Python programs. To finish, close Python turtle window manually.
# Example with cycle
import turtle
spiro = turtle.Turtle()
for i in range(20):
    spiro.forward(i * 10)
    spiro.right(144)
turtle.done()

Output:

```
* * *
```

# Example with colors
import turtle
jill = turtle.Turtle()
jill.color('red', 'yellow')
jill.begin_fill()
while True:
    jill.forward(200)
    jill.left(170)
    if abs(jill.pos()) < 1:
        break
jill.end_fill()
turtle.done()

Output:
# Fractal tree example
# with function and recursion

```python
import turtle

def tree(branchLen, t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15, t)
        t.left(40)
        tree(branchLen-15, t)
        t.right(20)
        t.backward(branchLen)

t = turtle.Turtle()
t.left(90)
t.up()
t.backward(100)
t.down()
t.color("green")
tree(75, t)
turtle.done()
```
# Draw with strokes example

```python
import turtle

t = turtle.Turtle()

t.shape("turtle")
t.color("Red")
t.penup()
t.goto(-100,0)
t.fillcolor("Red")
t.pendown()

t.begin_fill()
t.goto(-100,50)
t.goto(100,50)
t.goto(100,0)
t.goto(-100,0)
```

```**```
Examples

```python
t.end_fill()

t.penup()
t.goto(-100,50)
t.fillcolor("Green")
t.color("Green")
t.pendown()

t.begin_fill()
t.goto(0,100)
t.goto(100,50)
t.goto(-100,50)
t.end_fill()

t.penup()
t.goto(-40,0)
t.color("Red")
t.fillcolor("Orange")
t.pendown()

t.begin_fill()
t.goto(-40,30)
t.goto(-20,30)
t.goto(-20,0)
t.goto(-40,0)
t.end_fill()

t.penup()
t.goto(20,20)
t.fillcolor("White")
t.pendown()

t.begin_fill()
t.goto(20,40)
t.goto(50,40)
t.goto(50,20)
t.goto(20,20)
t.end_fill()

t.hideturtle()
t.penup()```
t.goto(-130,-50)
t.write("This is a house that Jack built",font=("Arial", 20, "normal"))

Output:

This is a house that Jack built
Chapter 21

Who Goes There-2, or GUI

Personally, I always prefer the program to work regardless of how it looks. If the terminal application does the job, then why do we need anything more advanced? However, there are people which like graphical bells and whistles almost equally (or sometimes even more) then the actual algorithm.

This chapter introduces Graphical User Interface (GUI) made with Python. The following program does almost the same thing as our earlier “Who goes there?”:

```python
from Tkinter import *
root = Tk()
svalue = StringVar() # defines the widget state as string

text = Text(root, height=1, width=15)
text.insert(END, "Who goes there?")
text.pack() # organizes widgets in blocks

txar = Entry(root, textvariable=svalue) # adds a textarea widget
txar.pack()
def act():
    print 'Welcome, ' '%s' % svalue.get()

bttn = Button(root, text="Answer!", command=act)
bttm.pack() # third widget block

root.mainloop() # to keep window
```

It uses Tkinter, one of many Python graphical modules. Tkinter has several advantages: it is simple, and it is included in most recent Python distributions and works well on many operation systems.

If you name this program like gui.py, and run it like python gui.py, then along with terminal, you will see new window, similar to:
If you enter your name into the text area and press button, then in terminal, you will see something like:

Welcome, Josh

There are few things to learn even from this simplified example. Look how many more lines of text has a GUI program comparing with terminal program. Just a reminder of how our terminal one looked:

```python
print "Halt!"
s = raw_input("Who goes there? ")
print "You may pass," , s
```

So out of 13 lines of our GUI program, only 3 needed for the terminal application. Everything else are these mentioned above bells and whistles. They are required to organize visual space and user input. Second, logic of these specifications is not always easy to understand. Moreover, different GUI systems have completely different logics.

That’s it, folks. We will not go further with GUIs.
## Chapter 22

### Python cheatsheet

<table>
<thead>
<tr>
<th>Basic</th>
<th></th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>help()</td>
<td>str() string to integer</td>
<td>return</td>
</tr>
<tr>
<td>quit()</td>
<td>str.replace()</td>
<td>def</td>
</tr>
<tr>
<td>'' and '&quot;'</td>
<td>str() number to string</td>
<td></td>
</tr>
<tr>
<td>\ import &lt;module&gt;</td>
<td>str.count()</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td>str.strip() remove</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>margin whitespace</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>str.upper() uppercase</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>str.lower() lowercase</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>str.split(&quot; &quot;) split</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>by whitespaces</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input and output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print</td>
<td></td>
<td></td>
</tr>
<tr>
<td>raw_input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>input()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>open(file, mode)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file.read()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file.write()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file.close()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float()</td>
<td>int() string to integer</td>
<td>for in</td>
</tr>
<tr>
<td>string to number</td>
<td>str() number to string</td>
<td></td>
</tr>
<tr>
<td>str.replace()</td>
<td>str.count()</td>
<td></td>
</tr>
<tr>
<td>str.strip() remove</td>
<td>margin whitespace</td>
<td></td>
</tr>
<tr>
<td>str.upper()</td>
<td>str.lower()</td>
<td></td>
</tr>
<tr>
<td>str.split(&quot; &quot;)</td>
<td>str.split(&quot; &quot;) split</td>
<td></td>
</tr>
<tr>
<td>compared</td>
<td>by whitespaces</td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= or &lt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Booleans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if</td>
<td></td>
<td></td>
</tr>
<tr>
<td>elif</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dictionaries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{'...':'...'}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dictionary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Markup
Chapter 23

Web

HTML

My first Web page

HTML, hypertext markup language based on simple text. So in theory, one’s first HTML may look like:

Hello, World!

And if you save it in the text editor as, saying, 1.htm and then open it in Web browser, your output will look somewhat like:

Hello, World!

Excellent. Now try two lines:

Hello, World!
Second line.

But output is:

Hello, World! Second line.

Well, something went wrong. You actually forgot the *markup*! Here it is:

<p>Hello, World!  
Second line.</p>

Output:

Hello, World!  
Second line.
This actually is not a proper HTML, and works well only because browsers are forgivable and do not mind mistakes if there is still a way to interpret your input. The minimal proper HTML is:

```html
<!DOCTYPE html>
<title>Title</title>
<p>Hello, World!</p>
<p>Second line.</p>
```

which outputs almost the same thing as above (but try to find one difference).

**How to work with HTML**

Now I hope that you already understood the main principle of working with markup file like HTML: you have two instances, not one.

First instance if your plain text file (which is open in some text editor), you change it, save it and then go and update the second instance, your browser window (to update, best is to use keyboard: F5 or Ctrl+R work on most browsers).

If you are not satisfied, go to text file, and change it again, save and go again to browser, update and look.

Fortunately, browsers are fast enough and render your HTML file in milliseconds.

**Second HTML document**

Example above is a bit boring. How to make something more like real world Web page?

Consider the second example:

```html
<!DOCTYPE html>
<title>My Web page</title>

<h3 align="center">My Web page</h3>

<p>This <em>Web page</em> is about me.<br />
This is who <strong>I</strong> am:</p>

<ul>
<li>First name
</li>
</ul>
```
<li>Last name</li>
</ul>

<p><a href="https://www.google.com/search?q=me">Search me on Google</a></p>

This outputs something like:

<table>
<thead>
<tr>
<th>My Web page</th>
</tr>
</thead>
<tbody>
<tr>
<td>This &lt;i&gt;Web page&lt;/i&gt; is about me.</td>
</tr>
<tr>
<td>This is who I am:</td>
</tr>
<tr>
<td>• First name</td>
</tr>
<tr>
<td>• Last name</td>
</tr>
<tr>
<td>&lt;a href=&quot;https://www.google.com/search?q=me&quot;&gt;Search me on Google&lt;/a&gt;.</td>
</tr>
</tbody>
</table>

---

**Step three. Fancy HTML**

Even more fancy document would contain colors, tables and images. We illustrate colors and tables here, and speak about images in next section.

This is our third document:

```html
<!DOCTYPE html>
<title>My third Web page</title>

<p>This way to <font color="red">colorize</font> Web content is outdated but still works.</p>

<p>Tables are good because they allow to align text:</p>

<!-- This is a comment, or disabled content --><!--

<table align="center" border="1">
<tr>
<td colspan="2" align="center">This is useful for:</td>
</tr>
<tr>
<td>Structured output</td>
<td>Complex layouts</td>
</tr>
</table>
```
Save this as 3.htm and open in browser. Output should be similar to:

<table>
<thead>
<tr>
<th>This way to colorize Web content is outdated but still works.</th>
<th>Tables are good because they allow to align text:</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is useful for:</td>
<td>Structured output</td>
</tr>
</tbody>
</table>

**Step four. PNG and JPEG**

Next step is to insert graphics. There are two ways: *raster way* and *vector way*. This section describes the first.

In short, raster graphics is made from little “bricks”, *pixels* (whereas vector graphics is made from formulas). These pixels could be generally of three kinds:

1. Black and white
2. Black, white and 254 shades of gray (yes)
3. 65536 (or more) hues of color

Suppose that we have image with size $4 \times 4$ pixels (by the way, all raster images must be rectangles). There is 2 columns, each of 4 of black pixels and 2 columns of white pixels. In all, our picture looks like two $2 \times 4$ rectangles side by side, black and white:

(Above is a scheme, not a real 16 pixel raster image.)

Now we need to describe this situation and make graphics file.

The simplest way is to describe it pixel by pixel, from, saying, top to bottom and left to right. So description (and internals of our graphical file) will look like:

```
1 1 1 1 1 1 0 0 0 0 0 0
```

where white pixel is encoded with 1, black pixel with 0, and space shows end of column. This way makes text file which size is 19 symbols. Now, your photograph made
with phone contains 20,000,000 pixels! How big will be graphic files? How fast is to read it?

So to save time and space, raster files must be compressed. One way of compression is easy to show:

14 14 04 04

Now we have 11 symbols and still retain the same information! Did you get it? First number encodes color of pixel, second number shows how many times to repeat, space is the same. This is a variant of run-length encoding (RLE) which despite of simplicity, is still around. Think about how to pack this even more.

Well, if there are 65536 colors, then it is not so easy to use RLE, we need something more complicated. In essence, there are two ways:

1. Lossless
2. Lossy

Lossless is, for example, RLE way described above. Many of lossless methods based on some efficient compression, and compression based on pattern discovery (like in our example above). This is, by the way, true not only for graphics, but also for music, and of course video which is in essence, combination of graphics and music.

Lossy way is also easy to imagine. Suppose that in our image, among 8 white pixels, there are 2 pixels which are not completely white, they are very light gray:

1211 2111 0000 0000

It looks like:

(Do you see these gray squares?)

If we encode this last file with RLE variant described above, result will be like:

112112 2113 04 04

It costs us 17 symbols, almost the same as non-compressed image!

Now lossy solution: human eye does not see well the difference between white and “very light gray” symbols. So replace those with white, and go with 11 symbols.
So you know now the most striking difference between two most important raster graphic formats, PNG and JPEG (JPG):

<table>
<thead>
<tr>
<th></th>
<th>PNG</th>
<th>JPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
<td>Lossless</td>
<td>Lossy</td>
</tr>
<tr>
<td>Degrades</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Artifacts</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparency</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Black &amp; white</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Small photo size</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Animation*</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

PNG (Portable Network Graphics) was invented for small-color, typically non-photographic images whereas JPEG (Joint Photographing Expert Group, file extension often JPG, hence more frequent JPG name) was made exactly for what name suggests, photographs, to avoid giant sizes of files. So to choose which to use, thinking is actually required (Fig. 1).

In music, the same distinction could be made between lossy MP3 and lossless FLAC.

* About animation. When PNG was invented, it inherited many features from its non-opensource predecessor, GIF format (and acquired many new features such as support for thousands of colors), except one: ability to show small animations. For some reason, it was thought that other formats (true videos, for example) will replace GIF animations in the end. Amazingly, this did not happen, and now the outdated GIF is still around, and the reason are animations. You can easily make these animations yourself using freeware ImageMagick:

```
convert -delay 40 -loop 0 1.png 2.png animated12.gif
```

One more feature of PNG is that you almost always can re-compress it and save just a bit more space. This is sometimes critically important for Web content. Many tools exist which help here, for example, pngquant, optipng and leanify (the latter is more universal and deals with many file types).

**Step five. SVG**

Start with the following example:
Figure 1. PNG versus JPG.

<!DOCTYPE html>
<title>Title</title>
<p><img src="example.png" /></p>
<p><svg>
<rect fill="black" width="20" height="40" />
<rect fill="green" x="20" width="20" height="40" />
</svg></p>

(This example works well with newest Mozilla Firefox and Google Chrome.)

Save this file as 5.htm, download example.png from the URL http://ashipunov.info/shipunov/school/univ_110/comp_lit_sci_maj/pics/example.png and place it in the same directory as 4.htm. Now open it in the browser. You will see something like:
Now start to magnify, in most browsers it is done with Ctrl++. You see something like:

Do you see the difference?

This difference, from the user standpoint is the most important distinction between vector and raster graphics: vector scales infinitely, raster is not.

From the standpoint of Web designer, vector (SVG, Scalable Vector Graphics) requires completely different approach to create and edit. And the above example explains why. While raster image is based on pixels, vector is something like geometrical description or even formula of shape. In a world of free software, the most appropriate program to work with SVG is Inkscape (and with raster graphics—GIMP).

When you see your vector SVG in browser, it is already converted into raster image (rasterized) because computer and phone screens are raster devices. Reverse operation (tracing) is also possible, and Inkscape or independent tool potrace (http://kilobtye.github.io/potrace) can trace raster images.

SVG is a relatively new technology, and sometimes rasterized slowly. There are SVG optimizers of which SVGOMG (https://jakearchibald.github.io/svgomg) is probably the best one.

PDF is just another vector graphics format, see below for more detailed discussion.
Step six. CSS

Next example is similar to our first HTML document, but there is one important difference:

```html
<!DOCTYPE html>
<style>p {text-align:center;}</style>
<title>Title</title>
<p>Hello, World!</p>
<p>Second line.</p>
```

Save it as 6.htm and open in browser:

<table>
<thead>
<tr>
<th>Hello, World!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second line.</td>
</tr>
</tbody>
</table>

Paragraphs are now centered because we declared this as a *paragraph (p)* style. This is how CSS (Cascading Style Sheets) work: they tell browser to format content in accordance with style descriptions.

Step seven. JavaScript.

Our Web examples were so far static. Below is our first DHTML, *dynamic* page:

```html
<!DOCTYPE html>
<title>Who goes there?</title>
<script>
  s = prompt("Who goes there?")
  document.write("<p>You may pass, " + s + "</p>"
</script>
```

Save it as 7.htm and try in a browser.

I believe that you spotted the similarity with our second Python program. Actually, they are the same, but written in different programming languages. Current example made with JavaScript, language which is similar to Python\(^1\). The biggest difference is that JavaScript works in Web browsers whereas Python is not.

So JavaScript is a client side tool, and every recent browser supports it. However, if you like Python more and want to use it to produce Web content, this is still possible. You need to install Python on your Web server (computer which distributes content in accordance with hypertext protocols), and then establish communication route

---

\(^1\)And dissimilar to programming language Java, their names similarity is almost coincidence.
between Python and HTML. This is called CGI (Common Gateway Interface). There are even Python Web frameworks (like Django) which work almost out of the box.

JavaScript is a core of our current Internet, together with server-side programming languages. If you know a bit of JavaScript, CSS and HTML, you understand what is going on in your Web browser.
Chapter 24

Ebooks

Plain text

Plain text is an amazingly simple way to interact with computer. By definition, plain

text is a also human-readable and therefore serves as a simples electronic book. Con-
ventions are really simple:

  • Make paragraph one long line
  • Separate paragraphs with empty line
  • No double spaces

\LaTeX

My first \LaTeX document

Well, \LaTeX is like HTML, based on plain text. So if you open the text editor, enter

Hello, World!

save it like, saying, 1.tex and run with

$ pdflatex 1.tex

you will receive ... an error:

This is pdfTeX, Version 3.14159265-2.6-1.40.18 (TeX Live 2017)
(preloaded format=pdflatex)
restricted \write18 enabled.
entering extended mode
(./1.tex
LaTeX2e <2017-04-15>
Babel <3.14> and hyphenation patterns for 84 language(s) loaded.

! LaTeX Error: Missing \begin{document}.

See the LaTeX manual or LaTeX Companion for explanation.
Type \texttt{H <return>} for immediate help.
...

1.1 H

\begin{verbatim}
    ello, World!
\end{verbatim}

(type \texttt{x} to escape.)

So again, like in HTML, you forgot some markup. And unlike HTML renderer, \LaTeX\ interpreter (\texttt{pdflatex} in that case) is not so forgivable and “likes” proper documents. It is also slower then any HTML browser.

So the minimal proper document is:

\begin{verbatim}
\documentclass{article}
\begin{document}
Hello, World!
\end{document}
\end{verbatim}

And output PDF is:

\begin{verbatim}
Hello, World!
\end{verbatim}

Nothing fancy, but works.

Note that like it was in HTML, you have two instances of your document: plain text open in editor, and PDF file which is an output from \texttt{pdflatex} engine. So essentially there are \textit{three} windows: (1) text editor, (2) terminal where you run \texttt{pdflatex} and (3) PDF reader which shows your resulted PDF.

\section*{Second \LaTeX{} document}

Our second \LaTeX{} combines our second and third HTML document: font features, lists, hyperlinks, color, tables and also one formula example (by the way, nothing was easier because this book is made in \LaTeX{}):
This \emph{page} is about me.\\
This is who \textbf{I} am:

\begin{itemize}
\item First name
\item Last name
\end{itemize}

\href{https://www.google.com/search?q=me}{Search me on Google.}

This way to \textcolor{red}{colorize} text works well in \LaTeX.

I like this formula most of all, because it is simple: $E=mc^2$

% This is a comment, or disabled content

Tables are good because they allow to align text:

\begin{center}
\begin{tabular}{|l|l|}
\hline
This is useful for: & Complex layouts \\
\hline
\end{tabular}
\end{center}

And output is:
My page

This *page* is about me. This is who I am:

- First name
- Last name

Search me on Google.

This way to *colorize* text works well in \LaTeX. I like this formula most of all, because it is simple: $E = mc^2$

Tables are good because they allow to align text:

<table>
<thead>
<tr>
<th>This is useful for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured output</td>
</tr>
<tr>
<td>Complex layouts</td>
</tr>
</tbody>
</table>

\LaTeX ebooks

So what about ebooks? How to make them with \LaTeX? This question is also very easy to answer: this ebook is made in \LaTeX. So to learn how to make real, big ebooks like this one, you only need to access the source code of this book which is openly available from this URL: [http://ashipunov.info/shipunov/school/univ_110/comp_lit_sci_maj/comp_lit_sci_maj.tex](http://ashipunov.info/shipunov/school/univ_110/comp_lit_sci_maj/comp_lit_sci_maj.tex)


You will also need text editor (like Geany) and the \TeX system itself. It is freely available (with installation instructions) from [https://www.tug.org/texlive](https://www.tug.org/texlive). Install the full version.
Symbol ebooks

These are text-based files formatted in order to print the real, physical book and/or also to read the book from electronic device. In short, most symbol ebooks are now of two kinds: PDF and EPUB.

PDF

PDF is the most frequently used file format for complicated electronic material, for example, ebooks with tables, illustrations, indices etc. As a file format, PDF is a multi-page vector file which can include multiple different objects together with their sizes and placements: photographic raster images, vector drawings, other PDFs, hyperlinks, and even movies and JavaScript programs. The full description of PDF is a big volume of almost 1,000 pages!

One of important features is that PDF must have a kind of contents table for all objects, and this is placed in the very end of file. As a result, if you download a big PDF and did not download it in full, it typically non-readable.

The other specific feature is that freeware programs with universal ability to edit PDF are still absent. Some users do not even know that PDF is editable. But why not? If there are objects and placements, then user should have an option to change them. However, while universal programs do not exist, there are multiple freeware tools which allow user to perform diverse PDF editing operations:

Extract raster images and text  poppler tools (pdftotext, pdfimages)
Images to PDF  convert from ImageMagick
Delete, insert, rearrange pages, merge, split PDFs  pdftk, cpdf
Cropping pages  pdfcrop, cpdf
Annotate, fill forms  Okular
Edit objects  LibreOffice Draw (1 page only, other restrictions also apply)

EPUB

EPUB is a format more close to HTML then to PDF, and works best mostly for the fiction books. It is easy enough to create, edit and convert EPUB to other formats with free software. Most powerful ebook convertor is probably Calibre. Freeware editor Sigil allows to create and edit EPUB ebooks with using Office-like visual interface.
Scanned ebooks

Scanned ebooks are made from images. These images came either from photographic device (phone, digital camera), or from specialized scanner. There two most important differences from symbol ebooks: there is no text and they are of big size.

The most simple scanned ebook is just a set of raster images, for example, multi-page TIFF. That last format was specifically designed for scanners. However, the problem is that big and especially color books make extremely large TIFF files.

DjVu

This format was designed in AT&T exactly for scanned books. Idea of DjVu is segmentation: image split in multiple fragments (text separate, background separate, illustrations separate) and each type is compressed in a specific way. For example, text is compressed by splitting into individual letters. So if there are 16 letters “a” on the page, they all encoded with one image plus information where to place it 16 times. This saves a lot of memory and space.

DjVu makes small and memory-efficient (hence DjVu books are easy to read) documents which still outperform almost any PDFs. Freeware DjView4 and many other programs (e.g., SumatraPDF) are able to read DjVu files.
Data
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR — DID HE BREAK SOMETHING?
IN A WAY—

DID YOU REALLY NAME YOUR SON Robert’); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Chapter 25

Intro to spreadsheets

Spreadsheets is the one of the most basic way to work with data. They were created mostly for accounting needs, but found many other uses, even in scientific applications. One of most known is Microsoft Excel but there are many others including freely available LibreOffice Calc and Gnumeric. They all similar, and following should be applicable to any of them.

Spreadsheet cells (those little rectangles) can hold numbers, words, or formulas (which then produce numbers). Try filling in some cells with numbers and others with words. Click on it to highlight it and start typing your entry. Use Enter to finish an entry and move down. Tab moves you to the right.

You can also use F2 key or the mouse to double click on a cell that you wish to change, or you can use the arrow keys to move around in the spreadsheet.

What happens if your words are longer than a cell? Try this with the cell to the right being empty and again with it having some entry.

Format

Cells have different formats and automatically convert their contents. Select a cell and change its format with the Format menu (Ctrl+1).

One of strange (for unexperienced users) features is the automatic date / number conversion. Suppose that you have a cell with text format (enter something in a cell, and then change format to “text”). Now, enter some date like “Nov 20, 2017”. Then change cell format to “date”, then change cell format again, now to “number” and start to edit cell content (press F2). If you are unlucky (some spreadsheet work this way, some not), then instead of date, you will see “43059”. Why this particular number, is easy to explain: this is the number of days from the beginning of “Excel era” (January 1, 1900) to November 20, 2017.
What is much more important that this (and related) conversion problems already brought numerous mistakes into public databases (see, for example, publications at https://www.ncbi.nlm.nih.gov/pubmed/15214961 and https://www.ncbi.nlm.nih.gov/pubmed/27552985). Therefore, you must know how to avoid them.

First, you can use the universal date format, “YYYYmmdd” where November 20, 2017 is just a number “20171120”.

Second, you can protect your “Nov 20, 2017” with text protection sign, apostrophe (or single quote):

, 

This sign tells that everything after is just a text (and therefore suppresses conversion of any kind).

---

**Formulas**

Enter the value 23 into cell D104 (use the column and row headings to find cell D104). Then come back to the top of the spreadsheet.

All formulas start with an equals sign =. Select cell D4 and enter the following formula:

\[=2*\text{D104}-4\]

The value 42 (which is \(3 + 2(23)\)) should appear. When you select a cell, the formula that is actually in the cell appears in the line at the top of the screen. What appears *in the cell* is the value of that formula.

---

**Drag and fill**

Clear out column A (select and delete). Enter the number 1 in cell A1 and the formula \(=\text{A1}+1\) in cell A2.

Then select cell A2 and move the cursor to the lower right corner of that cell. The cursor (which is usually a white plus sign) should change, depending on your software. Using that special cursor, drag the cell down to A10 so that the numbers 1 through 10 appear. This dragging “copies” the formula to the cells below. Check the formulas in this column and figure out your formula changed during the copying.
Now select the entire column of ten numbers and, using the special cursor from the lower right hand corner, copy them to the two columns to the right of the first. **What** happened to the formulas this time?

If you move a column, you should cut and paste. Select cells from A1 to A10, then select “cut”. The border of the cells starts to shimmer. Select cell B12 and choose the “paste” (or type Ctrl+V). **What** happened to the formulas?

OK, what if you do not want to copy (or cut) formulas but want to copy results? Select B12:B21, copy, then use Paste Special to choose Value and insert them into A1 again. **What** happened?

**Protect**

You can protect a cell name in a formula from being changed when the formula is moved or copied by using a dollar sign in front of the column letter or row number or both. For example, insert 1 in C12, then select cell C13. Change the formula to =$C$12+1, and then fill it down the column. All the numbers in the column after C12 should now be the same; the **protected formula** does not change when you copy.

Now enter the value 2 into cell C10. Change the formula of C13 to =C12+C$10 and copy it down the column. Now the column counts by twos. **Why**? Changing cell C10 automatically changes all the other cells to count by whatever value you want. **How** to count by threes?

**Ranges**

Some functions use a range of values. For example, if we want to sum the ten values in the first column. Enter the formula =sum(B12:B21) into cell B11. It should sum those ten numbers. **What** is your result?

Now, sum all your spreadsheet values (do not forget about D104). **How** to do it? **What** is the result?

**Conditions**

What if you want a spreadsheet which behaves? For example, if you have budget of 100 dollars, is it possible to make spreadsheet warn you about overdraft?

Remove all values from current spreadsheet. Enter numbers 20, 5, 15, 30, 3.70 in cells A1:A5. Now cell B1 will have formula to summarize your spendings in A column:
=sum(A1:A100), there will be a space for 95 more spendings. Cell B2 will have a budget (100), and cell B3 will warn you if your spendings are over 100 dollars:

=if(B1 > B2, "Overdraft!", "Still OK").

How to modify the formula and make spreadsheet warn you if your spendings a close (like 99%) of the budget? How to combine these two conditions?

---

**Tab delimited**

There are many spreadsheet formats but all of them are not plain text. However, your basic principle is to keep plain text wherever possible.

There is the plain text format which consist of multiple rows of text, and within each row, there are invisible big spaces, tabs, which split row into cells. This tab-delimited text file is a universal replacement of any data spreadsheet (i.e., spreadsheet without formulas and formatting).

It looks like:

First 2nd Third column
1 2 3
4 5 6

Enter the above into text editor (white spaces are tabs, only one white space is an actual space, it is marked with \ character).

Then save this file, saying, as 1.txt and try to import it into your spreadsheet application.

Now, enter any data in your spreadsheet, export it as a tab delimited text file (saying, 2.txt) and open it in text editor. Even simpler way is (on Linux and Windows) to copy your cells and then paste them into text editor. By default, they inserted as tab delimited!
Chapter 26

Phylogeny Primer

This chapter is more like assignment then code example or detailed explanation. The goal is to access (maybe, the first time) phylogenetic trees creations, based on public databases and online phylogeny tools.

1. Choose a favorite animal/plant group (use NCBI Taxonomy, family is preferable) with $>$ 5 members

2. Choose outgroup (use NCBI Taxonomy)

3. Copy sequences of COI (animals) or $rbcL$ (plants) in FASTA format to the text file (retain “>”!)

4. Paste them to online ClustalW (http://genome.jp), save alignment file

5. Apply result to PhyML (http://phylogeny.fr)

6. Rearrange (with outgroup) and redraw the tree, find most ”primitive” and ”advanced” ingroups
Chapter 27

Databases
Chapter 28

Resources