HON

Linux and Python

Unit 02 - Hands-On Networking - 2018

Prof. Dr.-Ing. Thorsten Herfet, Andreas Schmidt, Pablo Gil Pereira

Telecommunications Lab, Saarland Informatics Campus, 19th Feb. 2018

HON

Step 0: Download these **slides** and **task sheet** from CMS and follow along.

Outline

System Setup

- VirtualBox Installation
- HON Lab Virtual Machine
- VM Import

Linux and Command-Line

- System Internals (File System, Networking)
- Command-Line, Editors
- Common Commands (Navigation, Inspection, Networking)
- Get Help

Python

- Relevance (Popularity, Software Packages, ...)
- Working with basic types and collections
- Language Intro
  - Control Flow (`if`, `for`, ...)
  - Functions
  - Classes
- Useful Packages

System Setup

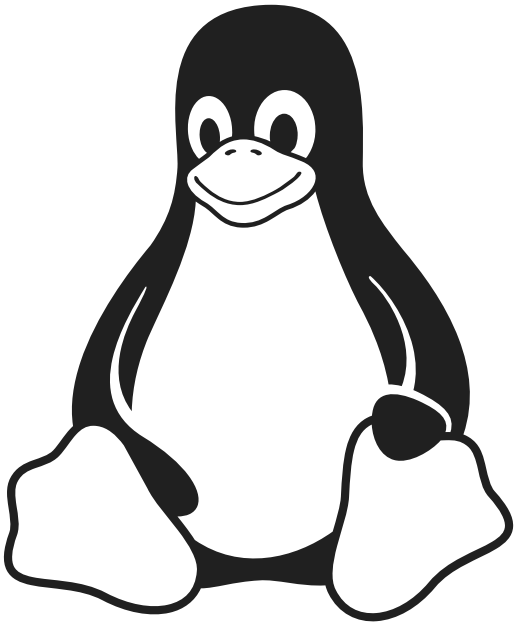
System Setup

- Install VirtualBox on your laptop.
- Download the **HON-VM** from CMS.
- Import the OVA into VirtualBox.
- Start the VM and enter the system.

System Overview

- Ubuntu 16.04. distribution with LXD user interface.
- All important tools installed:
  - wireshark, GNS3, ...
- Editors pre-installed: vim, nano

Note: You should also try installing some of the tools on your host system and play around with them. However, we **strongly** advise you to work inside the **HON-VM** during the tutorials. As tool implementations and versions might differ, we can better support you when we have the same environment. Furthermore, we have tested our code and tasks only inside the VM.



Linux and Command-Line

Copyright and Acknowledgement

- Some examples and parts of the content are taken from the book Linux Command Line by William E. Shotts.
- The **material is copyrighted**. Please treat the slides accordingly and do not share them.

Command-Line and Linux: Motivation

Linux

- **There are no secrets in it.**
- Everything is stored in files and can be inspected (given you have the priviliges).
- Most software is open source, so you can in principle go ahead and check out how it is implemented.

Command-Line

- **There is nothing you can't do with it.**
- Being hard to learn at first, mastering it can greatly improve your workflow.
- *"Graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible."*

👍 Working with and trying to understand networks, these pieces are priceless.

Linux | History

History

- **1983:** Richard Stallman launched GNU Project to create a UNIX-like *free* computer operating system.
- **1985:** Founding of *Free Software Foundation*.
- **1991:** Linux Torvalds creates the first version of the Linux kernel.
- **Mid 90s:** NASA and other organizations replace expensive machines with inexpensive commodity computer running Linux.

Today

- Most servers around the world run on Linux.
- Most embedded devices and novel IoT solutions operate on Linux.
- Most smartphones run a flavour of Linux (Android).
- The only segment with low coverage: Desktop and Laptop PCs (consumer market).

Warm-Up Command-Line

- On your Linux system do either:
  - Click on some "system button" and open the application "Terminal" or "Terminal Emulator" or alike.
  - Hit `Ctrl+Alt+T` (does not work on all systems).
- Enter the following command:

```
$ hon
```

- Result?

```
bash: hon: command not found
```

Priviledged Mode

- Many commands we use in the lecture work only when you have administrative rights, hence are executed as the `root` user.
- On your system, you can enter `root` mode using:

```
$ su -
Password:                    (enter root password)
```

or

```
$ sudo -i
Password:                    (enter your password)
```

- Note, how this changes your prompt:

```
#
```

- Yours might look different, but in general the prompt changes. Convention throughout this document: normal mode (`$`), root mode (`#`).

First Commands

- Find out what time it is (or at least what your system thinks):

```
$ date
Do 15. Sep 09:06:32 CEST 2016
```

- Find out how much free space is left on your drives:

```
$ df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda2       98298500  48442888  44839292  52% /
/dev/sda3      196730180 173665444  13048380  94% /home
```

- Find out how much free memory is available:

```
$ free
             total       used       free     shared    buffers     cached
Mem:       7842920    7485948     356972     964112     963920    3006432
-/+ buffers/cache:    3515596    4327324
Swap:      3998716         16    3998700
```

- End a session:

```
$ exit
```

Linux File System

- Hierarchical system, where every directory has a parent except `/` root.
- Directories:
  - `/`: Root, where all begins.
  - `/bin`: Binary programs are here.
  - `/boot`: Linux kernel, initial RAM image, boot loader.
  - `/dev`: Devices (keyboard, mouse, disks, USB sticks, ...).
  - `/etc`: Configuration (system-wide settings for programs).
  - `/home`: User directories
    e.g. `/home/hon`
  - `/tmp`: Temporary data.
  - `/var`: Frequently changing data.

- In a terminal session, you are always in a directory.
- You can find out which by:

```
$ pwd
/home/hon
```

- To see whats in the directory:

```
$ ls
Desktop  Documents  Music  Pictures  ...
```

- To go somewhere else:

```
$ cd /usr/bin
$ pwd
/usr/bin
```

Navigating Around using Pathnames

- **Relative Pathnames**: Always consider the current directory. Reference current folder using `./`:

```
$ pwd
/usr
$ cd ./bin
$ pwd
/usr/bin
```

- The `./` can nearly always be **omitted**:

```
$ pwd
/usr
$ cd bin
$ pwd
/usr/bin
```

- Navigate **upwards**:

```
$ pwd
/usr
$ cd ..
$ pwd
/
```

- **Absolute Pathnames**: Start with `/` and reference all directories.

```
$ cd /usr/bin
$ pwd
/usr/bin
```

- Navigate to your **home** directory:

```
$ cd
$ pwd
/home/hon
```

- Navigate to **previous** working directory:

```
$ cd -
$ pwd
/usr/bin
```

Explore The File System

- `ls` is used to list directory contents.

```
$ cd ~
$ ls
Desktop    Downloads  Pictures  Templates
Documents  Music      Public    Videos
```

- Adding `-l` option for more details:

```
$ ls -l
total 32
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Desktop
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Documents
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Downloads
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Music
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Pictures
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Public
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Templates
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Videos
```

- Columns indicate:
  - File Type (Directory, File, ...).
  - Permissions (Read, Write, Execute).
  - Owner and Group.
  - Size.
  - Modification Date.
  - Name.

- File names starting with a `.` are hidden files in Linux.
- Normally, they are not listed when using `ls`.
- Using `-a`, then can be revealed:

```
$ ls -l -a
total 32
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Desktop
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Documents
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Downloads
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Music
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Pictures
-rw-r--r-- 1 hon hon  675 Aug 26  2014 .profile
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Public
drwx------ 2 hon hon 4096 Mai 19 16:05 .ssh
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Templates
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Videos
```

- When using single-letter options (as `-a`), many programs allow to combine them:

```
$ ls -la
... same output as above ...
```

Options and Arguments

- General form of tasks you give the command-line to process:

```
command -options arguments
```

- The complete string is split at " " (space) and yields a variable number of strings.
- Therefore, the " " has to be used carefully and is many times forbidden (username, command-name, ...).
- Arguments are parsed into a list, e.g. `ls /home/hon /tmp` will execute `ls` on both folders.
- Options come in different flavors:
  - Flags: Option Name without argument (e.g. `-l`).
  - Named Arguments: Option Name + Argument (e.g. `--color=always`).
- Short option name have just one letter and are preceded by a single `-` (e.g. `-a`, `-l`). They exists for commonly used options and speed up typing.
- Long options names are preceded by `--` and a variable number of letter (`--human-readable`).

Inspecting Files

File Type

- On all operating systems, file endings should ressemble their type to make a user's life easy.
- On Windows, the ending is mandatory and used when checking what to do with the file.
- On Linux, endings do not matter and file contents are used instead.
- `file` can check for this:

```
$ file picture.jpg
L01/img/maxwell.jpg: JPEG image data, JFIF
standard 1.01
```

File Content

- If the contents are textual and not binary one can inspect it using `less`.

```
$ less sample.txt
```

- You quit `less` by typing `q`. You can navigate across the file using up and down buttons, as well as Page Up and Page Down.
- `less` also works with binary files... you might try it once to find out that this is not particularly useful.

Manipulating Files and Directories

- Create directories:

```
$ mkdir /tmp/foo
$ cd /tmp/foo
```

- Create empty files (or modify existing one's timestamps):

```
$ touch item1 item2
$ ls
item1 item2
```

- Fill them with content (more details later):

```
$ echo "Foo" >> item1
$ cat item1
Foo
```

- Copy them.

```
$ cp item1 item3
$ cat item3
Foo
```

- Move them:

```
$ mv item1 item4
$ ls
item2 item3 item4
```

- Remove them:

```
$ rm item4
$ ls
item2 item3
```

- Remove directories:

```
$ cd ..
$ rm -r foo
```

⚠ Be careful: There is no **un**delete for files and directories!

Edit Files

- You can start editing a file by `vim file.txt`.
- There are two important modes:
  - **Command**: Use cursor keys to move around, use letter keys to trigger commands. Type `i` to enter *Insertion* mode.
  - **Input** (Insertion): Cursors work as usual, letters will be entered at your current position). Leave mode using `Esc`.
- Write the file by typing `:w` in command mode.
- Exit `vim` by typing `:q` in command mode.


VIM can do a lot more and is definitely worth learning.

Getting Help

- Many linux commands support the `--help` option:

```
$ git --help
usage: git [--version] [--help] [-C <path>]
       [-c name=value] [--html-path]
       [--man-path] ...
```

- Other might use `help` as a command argument.

```
$ ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | ...
```

- Sometime you might wonder where a command points to:

```
$ which python
/usr/bin/python3.6
```

- For more details on a command, read the manual:

```
$ man ip
IP(8)                Linux                IP(8)
NAME
   ip  -  show  /  manipulate  routing,
   devices, policy routing and tunnels
SYNOPSIS
   ip [ OPTIONS ] OBJECT { COMMAND | help }
...
```

- `man` mode is similar to `less` (navigate with cursors, leave with `q`).

- `info` command is similar to `man`, but the pages inside it are better linked, easing navigation.

Linux | Streams & Pipelines

Processes have three streams:

- `stdin`: Input (e.g. via keyboard).
- `stdout`: Output (e.g. on console).
- `stderr`: Error (e.g. on console).

- `cat` can be used to output file contents on `stdout` and a stream can be redirected (`stdout`):

```
$ date > date.txt
$ cat date.txt
Fr 16. Sep 13:49:25 CEST 2016
```

- The output of a command can be piped to another command:

```
$ ls ~ | less
```

- `grep` can be used to filter for certain texts (and more):

```
$ ls -l | grep D
total 32
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Desktop
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Documents
drwxr-xr-x 2 hon hon 4096 Aug 27  2014 Downloads
```

- Use `head` and `tail` when inspecting start or end of large line-based files.

- Use `sort` to bring order in your output.

```
$ du -h * | sort -h -r
20K     item3.txt
12K     item1.txt
4,0K    item2.txt
```

Networking Commands | ping

- ping := Packet Internet Groper (common meaning, but probably not correct).
- `ping` allows you to check connectivity to a host using its DNS name:

```
$ ping google.com
PING google.com (172.217.16.206) 56(84) bytes of data.
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=1 ttl=55 time=12.1 ms
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=2 ttl=55 time=12.1 ms
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=3 ttl=55 time=12.1 ms
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=4 ttl=55 time=12.1 ms
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 12.139/12.167/12.193/0.112 ms
```

- or IP address:

```
$ ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=59 time=11.9 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=59 time=11.9 ms
64 bytes from 9.9.9.9: icmp_seq=3 ttl=59 time=12.2 ms
64 bytes from 9.9.9.9: icmp_seq=4 ttl=59 time=11.9 ms
64 bytes from 9.9.9.9: icmp_seq=5 ttl=59 time=11.9 ms
--- 9.9.9.9 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 11.905/12.007/12.293/0.188 ms
```

Networking Commands | wget & curl

- `wget` is typically used to download files using HTTP:

```
$ wget example.com
--2016-09-16 11:17:59--  http://example.com/
Resolving example.com (example.com)... 93.184.216.34, 2606:2800:220:1:248:1893:25c8:1946
Connecting to example.com (example.com)|93.184.216.34|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1270 (1,2K) [text/html]
Saving to: ‘index.html’
100%[=====================================================================>] 1.270       --.-K/s   in 0s
2016-09-16 11:18:00 (79,2 MB/s) - ‘index.html’ saved [1270/1270]
```

- The result is saved in a local file.

- `curl` work similar to `wget` but is suited for web developers. It allows to specify HTTP verbs (GET, POST, … see later) and payloads (e.g. JSON).

Networking Commands | ip

- iptools come with most modern Linux distributions and help to configure / inspect all network related settings.
- Show your network addreses:

```
$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 56:3a:9c:a7:af:1d brd ff:ff:ff:ff:ff:ff
    inet 134.96.1.3/25 brd 134.96.1.1 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::468a:5bff:fe97:aa9d/64 scope link
       valid_lft forever preferred_lft forever
```

- You can also find out your routes:

```
$ ip route show
default via 134.96.5.1 dev eth0  proto static
134.96.5.0/25 dev eth0  proto kernel  scope link  src 134.96.5.3  metric 1
```

- More details are covered in later tutorials.


python™

Why Python?

- Popular (in terms of rankings):
  - Ranked Programming Language of the year (2007, 2010).
  - Since 2003 consistently in top 10 most popular languagues (TIOBE index).
- Important and well-known organizations use it:
  - Google, CERN, NASA, ...
- Serious software is written in it or platforms using it:
  - GNS3, LibreOffice, Dropbox, Raspberry Pi, Exploit toolkits, ...
- General-purpose language for which a lot of libs/frameworks are available:
  - Networking: Dpkt, Scapy
  - Application Development: Django, Flask, SQL alchemy
  - Embedded/Hardware: RPi.GPIO, NFCPy
  - Scientific: Scipy, Numpy, Pandas, IPython

How to use it?

- Interactively using an interpreter (REPL):

```
$ python
Python 3.6.1 (default, Mar 27 2017, 00:27:06)
[GCC 6.3.1 20170306] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>>
```

- Script files:

```
$ echo "print('Hello World')" >> hello.py
$ python hello.py
Hello World
```

  Use text editors or Integrated Development Environments (IDE):

  - Atom.io (lightweight, cross-platform, free alternative to Sublime Text)
  - PyCharm (full-featured, cross-platform, free for students, IDE)

Versions: We are using and testing with **Python 3.6**. If you find old code in our material, please tell us!

Escape, Comment and Help

Escaping

- **Goal**: Print special string: *I'm fine*.

```
>>> print('I'm fine')
SyntaxError: invalid syntax
```

- **Solution A** - Double Quotes:

```
print("I'm fine")
```

- **Solution B** - Escaping:

```
print('I\'m fine')
```

Special Escape Sequences

```
print("Hello\tWorld") # Hello    World
print("Hello\nWorld") # Hello
                      # World
```

Comments

Clarify code and are not executed.

```
# This is a comment
print("Hello World") # this prints something
# print("This message won't get printed!")
```

Use them in your submissions!

Help

```
help(ord) # shows documentation
# Help on built-in function ord in module
# builtins:
#
# ord(c, /)
#     Return the Unicode code point for a
#     one-character string.
```

Online: https://docs.python.org/3/library/

Being Pythonic

- Python provides you with **concepts** and **code constructs** your C, C++, Java or other languages do not have.
- You can for sure program Python as you would write your C code, but it makes your code less expressive:

C Code

```c
for (int i=0; i < mylist_length; i++) {
   do_something(mylist[i]);
}
```

Python Code

```python
# unpythonic
i = 0
while i < len(mylist):
   do_something(mylist[i])
   i += 1
```

```python
# pythonic
for element in mylist:
   do_something(element)
```

- For now it might not be relevant for you, but you may face it in other people's code, so get used to it!
- For more: Secret Weblog - What is Pythonic? and PEP8

HON

Variables and Types

- **Variable**: Store objects, so that they can be referenced by name.

```
a = "Hello World"           # assign "Hello World" to variable "a"
print(a)                    # prints "Hello World"
b, c = "Hello", "World"     # multi-assignment
print(b, c)                 # prints "Hello World" (commas auto-include a space)
b, c = c, b                 # swapping values
```

- **Type**: A category an object can belong to.

```
a = 0       # Integer
b = 0.5     # Float
c = True    # Boolean (True or False)
d = "Test"  # String
a = "test"  # a is rebound to a string (no error in Python)
e = None    # None is nothing (cmp. null in Java)
```

Check the type by:

```
print(type(a)) # prints "<class 'str'>"
```

Algebra

Boolean

```
a = True
b = False
not A       # = False (Negation)
a and b     # = False (Logical AND)
a or b      # = True (Logical OR)
```

Conditions yield booleans:

```
a = 2
b = 2
a == b  # = True
a != b  # = False
```

Bitwise

```
a, b = 10, 2
a & b  # = 1010 & 0010 = 0010 = 2  (Bitwise AND)
a | b  # = 1010 | 0010 = 1010 = 10 (Bitwise OR)
a ^ b  # = 1010 | 0010 = 1000 = 8  (Bitwise XOR)
a >> 2 # = 1010 >> 2   = 0010 = 2  (Shift right)
b << 2 # = 10 << 2     = 1000 = 8  (Shift left)
```

Numeric

```
a, b = 10, 2
a + b # = 12          (Addition)
a - b # = 8           (Subtraction)
a * b # = 20          (Multiplication)
a / b # = 5           (Division)
a % b # = 0           (Modulo)
a ** b # = 10^2 = 100 (Power)
```

Division

```
10 / 8 # = 1 (floor divison on ints)
# floating point division with floats
float(10) / 8 # = 1.25
10. / 8 # = 1.25
10 / 8. # = 1.25
```

More Algebra and Logic

Shortcuts

```
a, b = 10, 2
a = a + 1 # too long
a += 1    # better (a = 12)
a -= 1    # a = 11
a *= b    # a = 22
a /= 10   # a = 2
a %= 3    # a = 2
a **= 2   # a = 4
```

Comparisons

```
a, b = 100, 10
a < b           # False
a <= b          # False
a > b           # True
a >= b          # True
not(a >= b)     # False
```

Strings

Creation and basic methods

```
s = " hello"
s[0]           # " "
s[1]           # "h"
s += " world"  # s = " hello world"
len(s)         # 12
"hello" in s   # True
ord("h")       # 104
chr(104)       # "h"
```

Advanced methods

```
s.startswith(" he")         # True
s.find("he")                # 1
s.find("x")                 # -1
s.strip(" ")                # "hello world"
s = s.strip(" ")            # s = "hello world"
s.replace("hello", "bye")   # "bye world"
```

Even more methods...

```
>>> dir("test") # lists all attributes of this object
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Formatting

Using the format method

```
"{} the {}".format("open", "pod bay doors")
# 'open the pod bay doors'
"{excuse} {person}".format(person="dave", excuse="i'm sorry,")
# "i'm sorry, dave"
"Dec: {0}, Hex: {0:X}, Other: {1}".format(42, 1337)
# 'Dex: 42, Hex: 2A, Other: 1337'
```

The Format Specification Mini-Language provides many options...

```
"{:_^7.2}".format(3.1415)
# '__3.1__'
```

Using f-strings

```
real = 4
imag = 2
f"z = {real}+{imag}i"
# z = 4+2i
```

What If

Check if condition is met:

```
a = 49
if a % 2 == 0:
    print("a is even")
else:
    print("a is odd")
```

Indentation defines scope:

```
a = 49
if a % 2 == 0:
    print("a is even")
else:
    if a % 7 == 0:
        print("a is divisible by 7")
    # no else required
```

Python requires at least one line per block:

```
if a % 2 == 0:
    foo = bar
else:
    pass # NOP
```

Conditions - Best Practise

Avoid nesting with `elif`:

```
if a % 2 == 0:
    print("a is divisible by 2")
else:
    if a % 3 == 0:
        print("a is divisible by 3")
    else:
        if a % 5 == 0:
            print("a is divisible by 5")
        else:
            print("a not divisible by 2, 3 or 5")
```

Better:

```
if a % 2 == 0:
    print("a is divisible by 2")
elif a % 3 == 0:
    print("a is divisible by 3")
elif a % 5 == 0:
    print("a is divisible by 5")
else:
    print("a not divisible by 2, 3 or 5")
```

Avoid nesting logic:

```
if username == "foo":
    if password == "secret":
        print("Login successful!")
    else:
        print("Login failed!")
else:
    print("Login failed!")
```

Better:

```
if username == "foo" and password == "secret":
    print("Login successful!")
else:
    print("Login failed!")
```

Loops

Repeat as long as a condition is met.

While

```
# Print "test" 10 times
i = 1
while i <= 10:
    print("test")
    i += 1
```

For

```
# Print "test" 10 times
for i in range(10):
    print("test")
```

Infinite loop

```
# Print "test" as long as program runs
while True:
    print("test")
```

Break

```
# Print "test" 10 times
i = 0
while True:
    print("test")
    i += 1
    if i == 10:
        break
```

Continue

```
# Print even numbers between 1 and 100
i = 0
while i < 100:
    i += 1
    if i % 2 == 1:
        continue #  go to next iteration
    print(i)
```

Lists

Creation and Indexing

```
# initialize
empty   = []
squares = [1, 4, 9, 16, 25]
things  = [1, 1/3., "test", True, []]
# indexing
squares[0]       # first item = 1
squares[-1]      # last item = 25
squares[-2]      # second last item = 16
empty[0]         # IndexError: index out of range
things[0] = -1   # change list
squares[0:3]     # [1, 4, 9] (slice notation)
squares[3:]      # [16, 25]
# operations
len(squares)            # 5
4 in squares            # True
squares += [36, 49, 81] # concatenation
```

`string.join()` with Lists

```
items = ["a","b","c"]
",".join(items)    # "a,b,c"
syllables = ["Pro","gram","ming"]
"".join(syllables) # "Programming"
```

List Manipulations

```
a = [1,2,3]      # a = [1,2,3]
a.append(4)      # a = [1,2,3,4]
a.extend([6,7])  # a = [1,2,3,4,6,7]
a.insert(4,5)    # a = [1,2,3,4,5,6,7]
a.remove(1)      # a = [2,3,4,5,6,7]
a.pop()          # a = [2,3,4,5,6]
a.index(5)       # 3
a.reverse()      # a = [6,5,4,3,2]
sorted(a)        # [2,3,4,5,6] (a=[6,5,4,3,2])
a.sort()         # a = [2,3,4,5,6]
```

Iteration

```
squares= [1,4,9,16,25]
# prints each element of "squares" (non-pythonic)
i = 0
while i < len(squares):
    print(squares[i])
    i += 1

# prints each element of "squares" (pythonic)
for s in squares:
    print(s)
```

Collection Types I

Tuples

```
empty = ()
a = (1,2)
len(a)   # 2
1 in a   # True
a[0]     # 1
a[0] = 3 # ERROR: tuples are immutable

# Syntax quirks
b = (1)   # integer
c = (1,)  # 1-element tuple
d = 1,    # 1-element tuple
c == d    # True
person = "Max", "Mustermann", 36  # 3-elem. tuple

# unpacking
first, last, age = person

# enumerating
for item in person:
    print(item)

# Output:
# Max
# Mustermann
# 36
```

Dictionaries

```
students = { 2551424: "Calvin",
             2572041: "Hobbes" }

students[2551424]          # Calvin
students[42]               # KeyError: 42
students.get(42, None)     # None
students[2575134] = "Susi" # Adding new student
2551424 in students        # True
del students[2572041]      # Remove student

ids = students.keys()      # [2551424, 2551424]
names = students.values()  # ["Calvin", "Susi"]
pairs = students.items()   # [(2551424, "Calvin"),
                           #  (2551424, "Susi")]

# print the mapping
for num in students:
    name = students[num]
    print("{}: {}".format(num, name))

# Output:
# 2551424: Calvin
# 2575134: Susi
```

Collection Types II - Sets

Unordered collection of items without duplicates.

```
basket = ["apple", "orange", "apple", "pear",
          "orange", "banana"]
fruits = set(basket) # {"orange", "banana",
                     #  "pear", "apple"}
"orange" in fruits   # True
"melon" in fruits    # False
```

Duplicate removal

```
a = [1,2,3,1,2,4,5,6]
without_dups = list(set(a)) # [1,2,3,4,5,6]
```

Math Operations: $A \cup B, A \cap B, A \setminus B, A \oplus B, A \subseteq B, A \subset B$

```
a = set({"red", "green"})
b = set({"purple", "red"})
aub  = a | b   # a.union(b)
               # = {red, green, purple}
anb  = a & b   # a.intersection(b)
               # = {red}
diff = a - b   # a.difference(b)
               # = {green}
xor  = a ^ b   # a.symmetric_difference(b)
               # = {green, purple}
anb <= a       # a.issubset(a)
               # = True
a   <  a       # a.issubset(a) and a != a
               # = False
```

Comprehensions

Lists

```
hundred = range(101)

# iterate over elements and accumulate squares
squares = []
for x in hundred:
    squares.append(x*x)

# Pythonic way
squares = [x*x for x in hundred]
```

List comprehension similar to math notation:

$$squares = \{ x^2 \mid x \in \{0, \ldots, 100\} \}$$

Sets

```
nums  = { x for x in range(50,100) }
empty = { x for x in [] }
```

Dictionaries

```
squares = { x: x*x for x in range(101) }
```

Functions

Define common functionality once:

```
def fib(n):
    # Compute fibonacci series up to n
    # [0,1,1,2,3,5,8,13,..., < n]
    a , b = 0 , 1
    res = []
    while a < n:
        res.append(a)
        a, b = b, a + b
    return res
```

Reuse it:

```
a = fib(5)    # [ 0, 1, 1, 2, 3]
b = fib(20)   # [ 0, 1, 1, 2, 3, 5, 8, 13 ]
c = fib(30)   # [ 0, 1, 1, 2, 3, 5, 8, 13, 21 ]
d = fib(100)  # [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
              #   55, 89 ]
```

Keyword arguments for default behaviour:

```
def get_number(complaint="Invalid number!"):
    while True:
        num = input("Please enter a number: ")
        if num.isdigit():
            return int(num)
        else:
            print complaint
```

Keyword arguments can be overriden if needed:

```
# standard version
a = get_number()

# polite version
a = get_number(complaint="Please repeat.")

# impolite version
a = get_number(complaint="You stupid?")
```

General Functions

Functions receive:

- A tuple of positional arguments (`*args`).
- A dictionary of keyword arguments (`**kwargs`).

```
def generic(*args, **kwargs):
    print "Positional arguments:"
    for a in args:
        print(a)
    print "Keyword arguments:"
    for (k,v) in kwargs.items():
        print("{}={}".format(k,v))

# (1) Direct usage
generic(1, 2, 3, a=0, b=1, c=2)

# (2) Packing
args = (1,2,3)
kwargs = {a:0, b:1, c:2}
generic(*args, **kwargs)
```

HON

Modules

Extend functionality beyond the builtins:

```
import math

math.pi            # 3.141592653589793
math.cos(math.pi)  # -1.0

# Third party http library
import requests

# Perform a HTTP GET request to google.com
response = requests.get("https://www.google.com/")
code = r.content
```

Explore new modules:

```
import math

dir(math) # list all attributes/methods of math module
# ['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', ...
help(math.fabs) # get the documentation
```

Find more modules at PyPI.

Python for Migrants (Java devs et al.)

Indentation

- Mandatory! (semantically relevant)
- Replaces `{ ... }` blocks of Java.

Python:

```
def funA(a):
    return a
```

Java:

```
int funA(int a) {
return a; }
```

Callables

Methods are first-class citizens (their only special trait is that you can `()` them).

```
>>> def foo():
>>>     print("bar")
>>> y = foo
>>> y()
bar
```

Type System

- Dynamic:
  - Variable name only bound to an object (no class associated!).
- Strongly Typed:
  - Explicit conversion necessary!
  - Not possible: `'1' + 0` gets `'10'`
- Duck-Typing
  - Type-Checking at runtime.
  - `duck.quack()`, `frog.quack()`

Note: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck (Duck-Test).

More Python for Migrants | Classes

Java

```java
public class Bit extends Object {
    int value;          // Field

    public Bit() {      // Constructor
        this.value = 0;
    }

    void invert() {     // Method
        ...
    }
}
```

Python

```python
class Bit(object):
    def __init__(self):  # Constructor
        self.value = 0   # Field

    def invert(self):    # Method
        ...
```

Dunder Methods

- Python has special methods which are implemented by the standard classes provided and you can implement in your classes to make them more "pythonic".
- These methods start and end with double underline (which is why this is often called "dunder" methods).
- Typical examples are (assume `obj` is an instance of `Class`):
  - `__init__`: "Constructor" method. Called by `obj = Class(a=5,b=7)`.
  - `__len__`: Gives the length of `obj` when using `len(obj)`.
  - `__repr__`: Returns the "official" string representation of `obj`.
    Usually in a form that lets you create another object of the class with the same fields, e.g. `repr(obj)` -> `"Class(a=5,b=7)"`
  - `__str__`: Return the "inofficial" string representation of `obj`.
    No need to return a valid Python expression, e.g. `str(obj)` -> `"5#7"`.

More of these can be found in the Python Data Model.

Type Hints

- Python is dynamically typed, so you generally do not specify what type a parameter or variable has. With static types, Duck typing would not be that easy.
- During development and debugging, missing type information can lead to accidents, misunderstandings, and higher order confusion.
- Python3.5 introduces the `typing` module and syntax to specify types:
  - Variables: `variable_name: type`, e.g. (`foo: str = 'Bar'`)
  - Functions: `def function_name(parameter1: type) -> return_type:`
- Within a Python interpreter, no types are checked. This is left to
  - Standalone *Type Checkers* (such as mypy)
  - *IDEs* (such as PyCharm or Atom with `atom-mypy`)

More details can be found in PEP483, PEP526 and the `typing` docs.

File I/O

Writing

```
with open("test.txt", "w") as f:
    f.write("This is the first line")
    f.write("\n") # new line
    f.write("This is the second line")
    f.write("\nThis is the third line\n")
    f.write("This is the fourth line\n")
```

Creates new file (and would overwrite existing one).

Reading

```
with open("test.txt", "r") as f:
    a = f.read(1) # Read 1 byte ("T") at
                  # current position
    f.seek(5)     # Go to position 5
    f.tell()      # Current position (5)
    b = f.read(2) # Read 2 chars ("is")
    c = f.read()  # Read the rest of the file.
    d = f.read(1) # "", since the position is end

# Read the file line by line
with open("test.txt", "r") as f:
    for line in f.readlines():
        do_something(line)
```

Result

```
$ cat test.txt
This is the first line
This is the second line
This is the third line
This is the fourth line
$
```

File Modes

Mode	Description
r	Reading
w	Writing
a	Appending
rb	Reading (binary)
r+	Reading + Writing
…	…

Binary Representations / Struct Module

Bytes can be represented similar to strings:

```
bytes = b"\x01\x41\xfe" # 3 byte string
print(bytes)            # b'\x01\x41\xfe'
print(repr(bytes))      # same as print(bytes)
```

We need this to parse network packets.

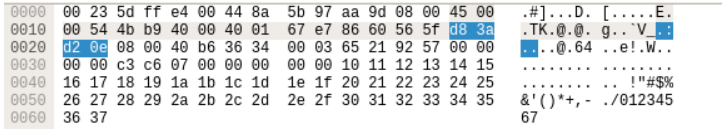
String Formats

Format	Type	Standard Size
B	Byte	1
H	Integer	2
I	Integer	4
L	Integer	4
Q	Integer	8
xs	String	x

```
import struct

a = 15
# "I" means a 4-byte integer
struct.pack("I", a)
# = b'\x0f\x00\x00\x00' (little endian)
struct.pack(">I", a)
# = b'\x00\x00\x00\x0f' (big endian)

# Pack multiple values
a, b = -10000, 50000
# i = signed, H = 2 bytes
packed = struct.pack("iH", a, b)

# Unpacking
c, d = struct.unpack("iH", packed) # returns tuple
a == c and b == d # True

-----------------------------------------------
# Tuple return type is important (!!!)

# Pack/unpack single 1-byte integer
a = 15
packed = struct.pack("B", a)
b = struct.unpack ("B", packed)

# Something went wrong
a == b  # False
type(b) # <type 'tuple'>

# The right way
b = struct.unpack("B", packed)[0]
a == b # True
```

Example

ICMP Packet dumped by Wireshark



Destination IP is marked.

Find out details

```
import struct
import socket

with open("icmp.dump", "rb") as f:
    f.seek(26)
    src, dst, icmp_type = struct.unpack("4s4sB",
                                        f.read(9))
    print("Src:\t", socket.inet_ntoa(src))
    print("Dst:\t", socket.inet_ntoa(dst))
    print("Type:\t", icmp_type)
```

HON

Bytes vs. Strings

- Python 2 used to mix bytes and strings, leading to problems with applications that require unicode for supporting more languages.
- Python 3 has a clear separation between a text in a specific encoding (string) and its binary representation (bytes).
- Encode (`str` -> `bytes`):
  - `"Über".encode('utf8')` -> `b'\xc3\x9cber'`
- Decode (`bytes` -> `str`):
  - `b'\xc3\x9cber'.decode('utf8')` -> 'Über'
  - `b'\xc3\x9cber'.decode('ascii')` -> ???
- A socket's `send` / `recv` functions accept and return bytes (used strings in Python 2).

JavaScript Object Notation (JSON)

- Well-established exchange format for coarsely structured data.
- Originates form how objects in JavaScript denoted.
- Smaller, less verbose as XML.

```xml
<person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <isAlive>true</isAlive>
    <age>25</age>
    <address>
        <streetAddress>21 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021-3100</postalCode>
    </address>
  <phoneNumbers>
    <phoneNumber>
      <type>home</type>
      <number>212 555-1234</number>
    </phoneNumber>
    <phoneNumber>
      <type>office</type>
      <number>646 555-4567</number>
    </phoneNumber>
    <phoneNumber>
      <type>mobile</type>
      <number>123 456-7890</number>
    </phoneNumber>
  </phoneNumbers>
  <children></children>
  <spouse></spouse>
</person>
```

Example

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON Back and Forth

- Make use of the `json` module to convert lists and dictionaries to strings and back.
- `load` converts JSON string to Python object.
- `dumps` converts Python object to JSON string.
  - Paremeterless for space-efficient output.
  - With `indent` and `sort_keys` for human readable output.
  - Use msgpack for even better space efficiency.

Dumping (Encode)

```
import json
d = {
    'first_name': 'Walter',
    'second_name': 'Bishop',
    'universes': ['prime', 'alternate'],
}
print(json.dumps(d, sort_keys=True,
                 indent=4,
                 separators=(',', ': ')))
# Output:
# {
#     "first_name": "Walter",
#     "second_name": "Bishop",
#     "universes": [
#         "prime",
#         "alternate"
#     ]
# }
```

Loading (Decode)

```
import json
json_string =
    '{"first_name":"Walter","last_name":"Bishop"}'
parsed_json = json.loads(json_string)
print(parsed_json['first_name']) # prints "Walter"
```

Generators

Definition: Pythonic idiom for creation of iterable content on-the-fly.

Without Generators

Code:

```
def squares(n):
    res = []
    for i in range(n):
        res.append(i**2)
    return res

for i in squares(n):
    process(i)
```

Execution flow:

- Generate all squares until n.
- Process all squares with `process`.

With Generators

Code:

```
def squares(n):
    for i in range(n):
        yield i**2

for i in squares(n):
    process(i)
```

Execution flow:

- Generate a square.
- Process it using `process`.
- Loop until n is reached.

More details: Python Wiki

Testing

- Important for serious software devs.
- Ensure stability / quality of software.
- Provided by Python through the `unittest` module.
- Used in the projects!
  - We give you unit tests.
  - There are additional tests.
  - You should pass all to show us that your software is good.

Example

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

Run Test

```
$ vim test.py # insert code from above
$ nosetests
.
----------------------------------------------------
Ran 1 test in 0.002s

OK
```

Wrap-Up

Further Reading

Linux

- Ryan Chadwick's Linux Tutorial
- A Byte of VIM
- Any other tutorial / online book / etc. on Linux and the commandline.

Python

- Python 3.6 Documentation
- How to Think Like a Computer Scientist
- Python Programming: An Introduction to Computer Science
- The Hitchhiker's Guide to Python!

Action Points

- **Install** and **Play** with the **HON-VM**.
- **Retype** all Linux commands in this document and **reproduce** the results.
- **Play** with Linux commands and use `man` to learn more about them.
- **Practice** Python as much as you can. You need it for the projects!
  - Program along online tutorials like Learn Python The Hard Way.
  - Do the **task sheet** we provide you.
- **Google** or ask **StackOverflow** for tips if you get stuck.
- **Team up** with your peers.
- **Ask us** if you still can't find an answer.